FREE eBook

LEARNING xmpp

Free unaffiliated eBook created from **Stack Overflow contributors.**

#xmpp

Table of Contents

About 1
Chapter 1: Getting started with xmpp 2
Remarks 2
Versions 2
Examples 2
Connecting and sending a message 2
SleekXMPP (Python) 2
Smack (Java / Android) 3
Creating a Chat Session and sending a message 3
Create Xmpp Client Connection Using agsxmpp library 3
Send a message using agsxmpp library 4
Chapter 2: Architecture 6
Remarks 6
Addressability 6
Stateful Streams 6
Routing 6
Servers 6
Examples 7
Visualizing the XMPP Network as a Graph 7
Chapter 3: Stream Negotiation 10
Remarks 10
Examples 10
Closing a stream 10
Starting a stream 10
Close XMPP connection using agsxmpp library 11
Chapter 4: XMPP Addresses aka. JIDs (Jabber Identifiers) 12
Syntax 12
Parameters 12
Remarks 12
Examples 12

Splitting a JID (generic) 12

JID Types 13

Validating a JID (generic) 13

Splitting a JID (Go) 14

Splitting a JID (Rust) 15

Credits 16

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: xmpp

It is an unofficial and free xmpp ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official xmpp.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with xmpp

Remarks

The Extensible Messaging and Presence Protocol (XMPP) is a network protocol that uses XML to exchange structured data between two or more network connected entities in near-real-time. XMPP was created to satisfy the IETFs guidelines for instant messaging and presence protocols (RFC 2779), but its purpose goes far beyond IM. It is also used as a message-oriented middleware, for machine-to-machine (M2M) communication and for the Internet of Things (IoT).

The lightweight XMPP core protocol provides users with

- strong authentication
- global addresses
- structured and extensible format for data exchange

The extensible approach makes it possible to build custom protocols on top of XMPP core.

The core XMPP protocol is defined in RFC 6120 and is managed by the Internet Engineering Task Force (XMPP). The instant messaging extensions are defined in RFC 6121, and a third document (RFC 7622) defines the format of XMPP addresses, also called "Jabber Identifiers" (JIDs). Additional functionality is specified in the form of XMPP Extension Protocols (XEPs), which are created by the community and maintained by the XMPP Standards Foundation (XSF).

Versions

Version	Notes	Release Date
1.0	Core: RFC 6120, IM: RFC 6121, Address: RFC 7622	2011-03-01
0.9	Core: RFC 3920, IM: RFC 3921, Address: RFC 6122	2004-10-01

Examples

Connecting and sending a message

SleekXMPP (Python)

```
import sleekxmpp

client = sleekxmpp.Client("address@example.net", "password")
client.connect()
client.process(blocking=False)
client.send_message(mto="remote@example.net", mbody=self.msg)
```

Smack (Java / Android)

```
XMPPTCPConnection connection = new XMPPTCPConnection("user", "password", "example.org")
connection.connect().login();
Message message = new Message("otheruser@example.net", "Hi, how are you?");
connection.sendStanza(message);
connection.disconnect();
```

Creating a Chat Session and sending a message

Smack (Java)

- Using Smack 4.1
- It is recommended to include Smack as Maven dependency in your project (e.g. by using gradle or Maven).
- Otherwhise the following Smack artifacts/jars have to be added manually to the classpath: smack-core, smack-extensions, smack-experimental, smack-im, smnack-tcp, smack-java7

```
import org.jivesoftware.smack.ConnectionConfiguration.SecurityMode;
import org.jivesoftware.smack.SmackException;
import org.jivesoftware.smack.XMPPException;
import org.jivesoftware.smack.chat.Chat;
import org.jivesoftware.smack.chat.ChatManager;
import org.jivesoftware.smack.chat.ChatMessageListener;
import org.jivesoftware.smack.packet.Message;
import org.jivesoftware.smack.packet.Presence;
import org.jivesoftware.smack.tcp.XMPPTCPConnection;
import org.jivesoftware.smack.tcp.XMPPTCPConnectionConfiguration;

public void sendMessage() {

XMPPTCPConnectionConfiguration config =
  XMPPTCPConnectionConfiguration.builder()
            .setServiceName("mydomain.local")
            .setHost("127.0.0.1")
            .setPort(5222)
            .build();

XMPPTCPConnection connection = new XMPPTCPConnection(config);

connection.connect();
connection.login("test1", "test1pwd");

ChatManager chatManager = ChatManager.getInstanceFor(connection);
String test2JID = "test2@domain.example";
Chat chat = chatManager.createChat(test2JID);
chat.sendMessage("Hello, how are you?");

connection.disconnect();
}
```

Create Xmpp Client Connection Using agsxmpp library

```
public void OpenXmppConnection(int port, bool useSsl, string serverJid, string userName,
string password)
        {
            try
            {
                _xmppClientConnection.AutoResolveConnectServer = true;
                _xmppClientConnection.Port = port;
                _xmppClientConnection.UseSSL = useSsl;
                _xmppClientConnection.Server = serverJid;
                _xmppClientConnection.Username = userName;
                _xmppClientConnection.Password = password;
                _xmppClientConnection.Resource = "web";


                //authenticate and open connection with server
                _xmppClientConnection.Open();
            }
            catch (Exception ex)
            {
            }
        }
```

Send a message using agsxmpp library

```
        public class ConversationManager
        {
            #region ClassMemeber

            private XmppClientConnection _xmppClientConnection = null;



     public ConversationManager(XmppClientConnection con)
            {
                _xmppClientConnection = con;
             }

 public void SendMessage(string message, string to, string guid, string type)
        {
            try
            {
                if (_xmppClientConnection != null)
                {
                    Jid jidTo = new Jid(to);
                    agsXMPP.protocol.client.Message mesg = new
agsXMPP.protocol.client.Message(jidTo, _ConnectionWrapper.MyJid,
                        agsXMPP.protocol.client.MessageType.chat,
                                      message);

                    mesg.Id = guid;
                    mesg.AddChild(new
agsXMPP.protocol.extensions.msgreceipts.Request());//request delievery
                    _xmppClientConnection.Send(mesg);
                }
            }
            catch (Exception ex)
            {
            }
        }
}
```

Read Getting started with xmpp online: https://riptutorial.com/xmpp/topic/2451/getting-started-with-xmpp

Chapter 2: Architecture

Remarks

XMPP allows for the full-duplex exchange of structured data and concurrent processing of requests between globally addressable clients and servers on the network. Unlike HTTP and the "Representational State Transfer" (REST) architecture widely deployed on the web, XMPP connections are stateful and concurrent, and an unlimited number of transactions may occur in the context of a single session. This architecture is sometimes refered too as "Availability for Concurrent Transactions" (ACT).

Addressability

To faciliate routing across the network, all XMPP addresses are globally addressable. Like email, this is acomplished with DNS and a federated client/server architecture. Addresses are of the form `localpart@domainpart/resourcepart` where the localpart is optional and corresponds to a user of the network, the domainpar is required and corresponds to a server, and resourcepart is optional and refers to a specific connected client for that user (in XMPP users may be signed in from many different locations, eg. a phone and a laptop in the case of instant messaging, or many sensors using one account in the case of internet-of-things enabled devices). XMPP also provides facilities for discovering the presence (availability) of other addresses on the network.

Stateful Streams

XMPP connections are long lived TCP connections that transport XML streams from a client to a server (c2s) or from a server to a server (s2s). Having these sessions be long lived and stateful allow nodes in the network to transmit data at any time and have it routed or delivered immediately.

Routing

Streams form a direct link on the network between a client and a server or a server and a server. If a client wishes to communicate with a remote client on the network, they first send the information to their server which forms a server-to-server connection with the remote server which then delivers the information to its client.

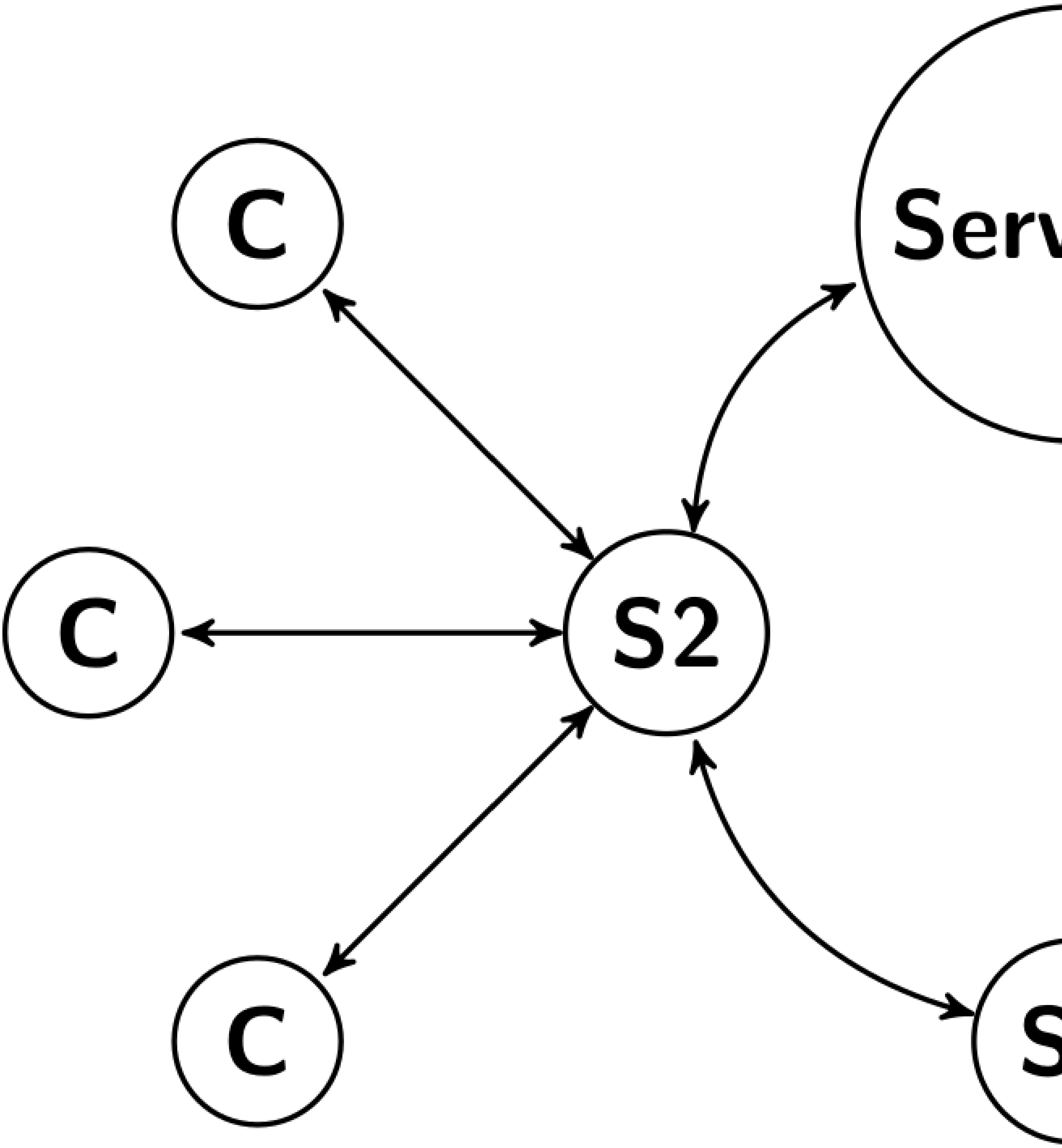
Servers

Servers in the XMPP network route data, but also have a number of other responsibilities including maintaining session state, storing client data (chat history, files, messages sent when no client was online to receive them, contact lists, etc.). They are where most of the business logic of handling an XMPP connection lives. This allows clients to remain as "dumb" as possible (containing very little logic).

Examples

Visualizing the XMPP Network as a Graph

The XMPP network can be thought of as a bidirected graph with servers (S) operating in a mesh, clients (C) clustered about their local server, and streams represented by extraverted edges:



When a client wants to send data (eg. a message or presence information) across the network to another client, the message is always routed along the shorted possible path (from a client to its server, then to the remote client if they are on the same server or to the remote clients server and then to the client if the remote client is on a different server).

Read Architecture online: https://riptutorial.com/xmpp/topic/3038/architecture

Chapter 3: Stream Negotiation

Remarks

XMPP connections comprise two XML streams: one for ingress and one for egress. These streams are generally sent over the same TCP connection (although sometimes multiple connections may be used, especially for server-to-server connections) and share certain features for which negotiation is required (eg. authentication with SASL).

Examples

Closing a stream

A stream is closed by sending a closing `</stream>` tag. After the closing stream tag is sent, no more data should be sent on the stream (even in response to data received from the other party). Before closing the connection, the sending entity should wait for a response `</stream>` tag to give the other party time to send any outstanding data and should time out (and terminate the underlying TCP connection[s]) if a closing stream tag is not received within a chosen amount of time.

```
</stream:stream>
```

If the stream is encrypted with TLS, the parties must cleanly terminate TLS by sending a TLS `close_notify` alert and receiving one in response. Your TLS library probably does this for you.

Starting a stream

Once a TCP connection is established, the initial stream header is sent by the initiating entity. Similarly, whenever a stream restart is required (eg. after negotiating a security layer such as TLS) a stream header must also be sent:

```
<?xml version='1.0'?>
<stream:stream
    from='juliet@im.example.com'
    to='im.example.com'
    version='1.0'
    xml:lang='en'
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'>
```

The XML header is optional, but if it exists it must not specify anything other than XML version 1.0 with UTF-8 encoding.

In response, the receiving entity will send its own opening stream tag containing a unique session ID:

```
<?xml version='1.0'?>
<stream:stream
    from='im.example.com'
    id='++TR84Sm6A3hnt3Q065SnAbbk3Y='
    to='juliet@im.example.com'
    version='1.0'
    xml:lang='en'
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'>
```

Close XMPP connection using agsxmpp library

```
  public class ConnectionManager
    {
         private XmppClientConnection _xmppClientConnection = null;

         public ConnectionManager()
                {
                    if (_xmppClientConnection == null)
                    {
                        _xmppClientConnection = new XmppClientConnection();
                    }
                }
         public void CloseXmppConnection()
                {
                    try
                    {
                        if (_xmppClientConnection != null)
                        {
                                //Close xmpp Client Connection
                             _xmppClientConnection.Close();
                        }


                    }
                    catch (Exception ex)
                    {
                    }
                }
    }
```

Read Stream Negotiation online: https://riptutorial.com/xmpp/topic/4248/stream-negotiation

Chapter 4: XMPP Addresses aka. JIDs (Jabber Identifiers)

Syntax

- [localpart "@"] domainpart ["/" resourcepart]

Parameters

Part	Common Usage
Localpart	Identifies an XMPP entity (optional)
Domainpart	Identifies the XMPP service
Resourcepart	Identifies a session of an XMPP entity (optional)

Remarks

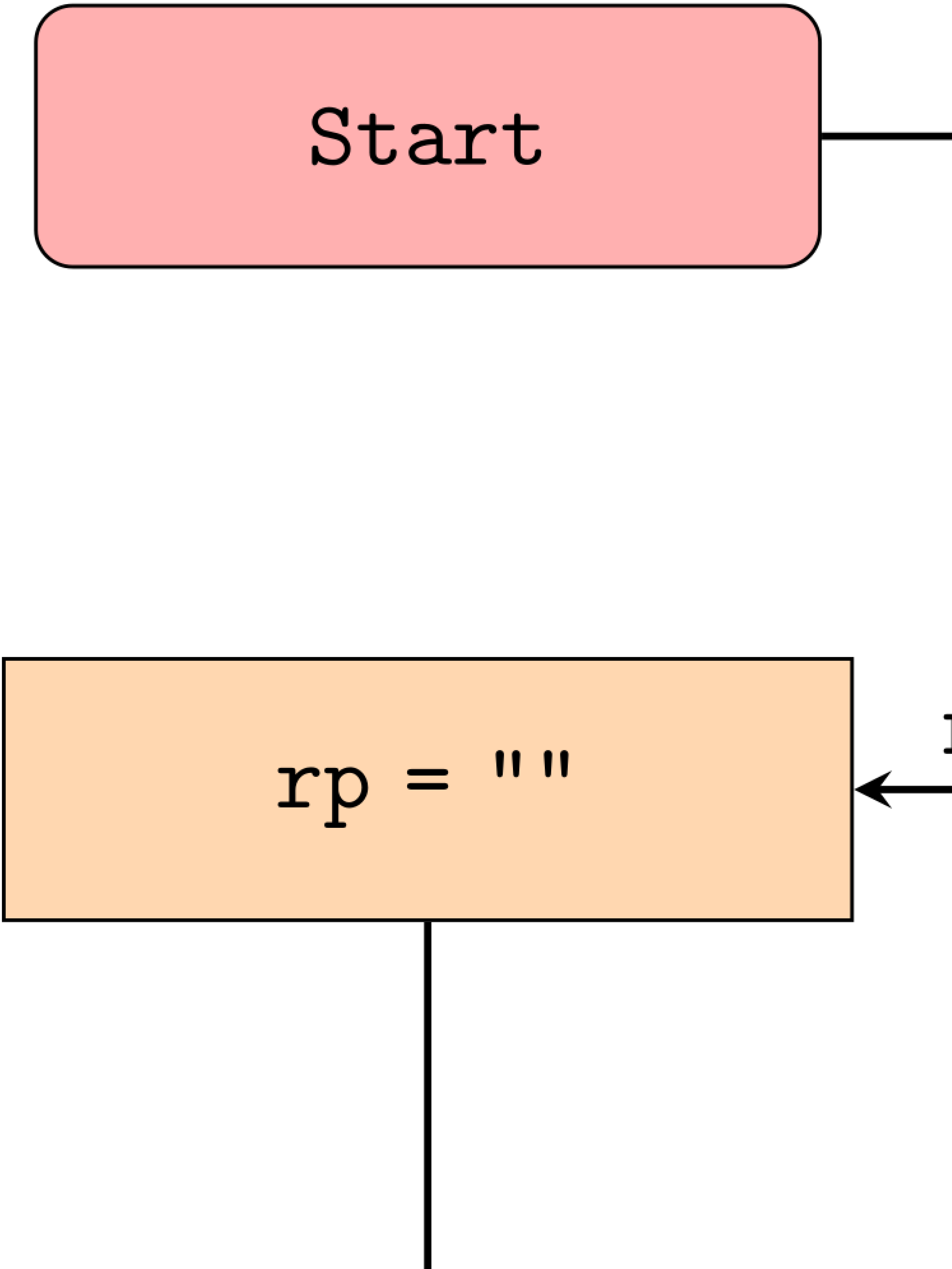
XMPP addresses, more commonly known as JIDs (Jabber Identifiers) are defined in RFC 7622 and act as addresses on the XMPP network. They look like an email address, but sometimes have an optional "resourcepart" at the end that identifies a particular client logged in as the account represented by the rest of the address (since XMPP may have multiple clients connected per account). An example of an XMPP address with the resourcepart (a client) `xyz` is:

```
romeo@example.net/xyz
```

Examples

Splitting a JID (generic)

To split a JID into its component parts (the localpart, domainpart, and resourcepart), the following algorithm should be used (where the localpart is represented by `lp`, the resourcepart by `rp`, and the domainpart by `dp` and `∈` is used to check if the given character is included in the string):



Start
rp = ""

- Check that the localpart doesnot contain any of `"&'/:<>@`
- Check that the resourcepart is less than 1024 bytes
- Check that the domainpart is greater than zero bytes and less than 1024 bytes (and possibly validate that the individual parts of the domain fit into DNS requirements)
- If the domain is a valid IPv6 address, ensrue that it uses bracketed notation (eg. `[::1]` instead of `::1`)
  instead of `::1`)

Splitting a JID (Go)

The `mellium.im/xmpp/jid` package implements operations on JIDs. To split a JID string into its component parts the `SplitString` function may be used:

```
lp, dp, rp, err := SplitString("romeo@example.net")
```

No validation is performed by the function and the parts are not guaranteed to be valid.

To manually split a string without depending on the `jid` package, the underlying code looks like this:

```
// SplitString splits out the localpart, domainpart, and resourcepart from a
// string representation of a JID. The parts are not guaranteed to be valid, and
// each part must be 1023 bytes or less.
func SplitString(s string) (localpart, domainpart, resourcepart string, err error) {

    // RFC 7622 §3.1.  Fundamentals:
    //
    //    Implementation Note: When dividing a JID into its component parts,
    //    an implementation needs to match the separator characters '@' and
    //    '/' before applying any transformation algorithms, which might
    //    decompose certain Unicode code points to the separator characters.
    //
    // so let's do that now. First we'll parse the domainpart using the rules
    // defined in §3.2:
    //
    //    The domainpart of a JID is the portion that remains once the
    //    following parsing steps are taken:
    //
    //    1.  Remove any portion from the first '/' character to the end of the
    //        string (if there is a '/' character present).
    sep := strings.Index(s, "/")

    if sep == -1 {
        sep = len(s)
        resourcepart = ""
    } else {
        // If the resource part exists, make sure it isn't empty.
        if sep == len(s)-1 {
            err = errors.New("The resourcepart must be larger than 0 bytes")
            return
        }
        resourcepart = s[sep+1:]
        s = s[:sep]
    }
    //    2.  Remove any portion from the beginning of the string to the first
    //        '@' character (if there is an '@' character present).

    sep = strings.Index(s, "@")

    switch sep {
    case -1:
        // There is no @ sign, and therefore no localpart.
        localpart = ""
        domainpart = s
    case 0:
        // The JID starts with an @ sign (invalid empty localpart)
        err = errors.New("The localpart must be larger than 0 bytes")
        return
    default:
        domainpart = s[sep+1:]
        localpart = s[:sep]
    }

    // We'll throw out any trailing dots on domainparts, since they're ignored:
    //
    //    If the domainpart includes a final character considered to be a label
    //    separator (dot) by [RFC1034], this character MUST be stripped from
    //    the domainpart before the JID of which it is a part is used for the
    //    purpose of routing an XML stanza, comparing against another JID, or
    //    constructing an XMPP URI or IRI [RFC5122].  In particular, such a
    //    character MUST be stripped before any other canonicalization steps
    //    are taken.

    domainpart = strings.TrimSuffix(domainpart, ".")

    return
}
```

Splitting a JID (Rust)

In Rust the `xmpp-addr` (docs) crate can be used to manipulate JIDs. To split a JID into its component parts (without validating that those parts are valid), the `Jid::split` function may be used:

```
let (lp, dp, rp) = Jid::split("feste@example.net")?;
assert_eq!(lp, Some("feste"));
assert_eq!(dp, "example.net");
assert_eq!(rp, None);
```

Read XMPP Addresses aka. JIDs (Jabber Identifiers) online: https://riptutorial.com/xmpp/topic/3036/xmpp-addresses-aka--jids--jabber-identifiers-

Credits

S. No	Chapters	Contributors
1	Getting started with xmpp	Afsheen126, bilal, Community, Flow, ge0rg, khan, Sam Whited
2	Architecture	Sam Whited
3	Stream Negotiation	khan, Sam Whited
4	XMPP Addresses aka. JIDs (Jabber Identifiers)	Flow, ge0rg, Sam Whited