Simple Jabber - Divide And Conquer XMPP

Jan Klemkow

30.08.2015

Abstract

The Extensible Messaging and Presence Protocol (XMPP)[1] is like the web. It is far too complex to be implemented in one program with Unix philosophy in mind. But like the web, you have to deal with it. As of this writing, it is the only open and widely used instant messaging protocol on the internet. Its extensibility is the main reason that an implementation in a single program is nearly impossible. Most implementations of XMPP deal with this by omitting extensibility and features or by embedding extensibility via plug-ins. Implementations like pidgin[2] try to implement as much as possible of the XMPP feature set. This leads to a large and inflexible program. Third party programs that want to interact with pidgin have to depend on the pidgin plug-in API or the D-Bus[3] communication channel. Other much more minimalistic implementations like jj[4] trade extensibility of the XMPP protocol for simplicity. This paper describes an approach to master this problem. It provides a minimal implementation of the core protocol of XMPP and keeps the possibility to extend it with third party tools without plug-in APIs or a special communication channel.

1 Protocol

The XMPP protocol mainly consists of three XML tags, called stanzas (*message*, *presence* and *iq*). The *message* and *presence* stanza are used for tasks that their names suggest. *iq* does everything else. The *iq* stanza mainly handles the extensibility part of the protocol. Beside these stanzas there a some other XML tags which handles things like connection, authentication and error messages.

At this level the protocol appears to be simple. But this is just the basic structure. The complexity starts within the sub XML tags of these three stanzas.

[1] http://xmpp.org/
[2] http://pidgin.im/
[3] http://freedesktop.org/wiki/Software/dbus/
[4] http://23.fi/jj/

2 Design

The design goal of *sj*[5] is to delegate as much knowledge of the inner XML structure as possible to other programs. The *sj* implementation consists of four daemons. One daemon for every stanza and one to handle connections, authentication and stanza routing. The following subsections describe the functionality of these core daemons.

2.1 sj

The program *sj* handles a network connection, its authentication and provides stanza routing. It spawns the other three daemons after the connection to an XMPP server is established. It communicates over pipes with the other daemons. If *sj* receives a stanza from the XMPP server, it forwards the whole tag over an unidirectional pipe to the responsible daemon. For outbound communication, *sj* opens a named pipe named *in*. If a daemon or any other program wants to send a stanza to the XMPP server, it just opens the *in* file and writes its XML tag into it.

2.2 messaged

The *messaged* daemon handles all message tags and the interface for the chat front end. It extracts the sender and the message text from all incoming message stanzas. It delivers the message text to the *out* file of the corresponding sender similar to IRC[6] client *Irc It(ii)*[7]. It also opens named pipes for every known chat contact. These files are also named *in* files similar to the *in* file of *sj*. To send a message to a chat contact, a front end program simply opens the corresponding *in* file, writes the chat message into it and closes it. *messaged* encapsulates this plain text message within a well formed message stanza and writes it into the *in* file of *sj*.

2.3 presenced

The *presenced* handles the presence stanzas in the same way like *messaged* does for message stanzas. There are two more files inside of a contacts directory beside of *in* and *out* which are named *presence* and *mypresence*. *presence* contains the presence status of the corresponding contact. *mypresence* contains ones own presence status that should be seen by the contact. If there is no *mypresence* file inside of a contacts directory the presence status of a global *mypresence* file is used.

[5] http://klemkow.net/sj.html
[6] http://tools.ietf.org/html/rfc1459
[7] http://tools.suckless.org/ii/

2.4 iqd

The *iqd* handles the extensions. *iqd* itself knows nothing about any extension. Like *sj*, it just routes the iq stanzas to the programs which know how to handle them.

If an extension program wants to send an iq request tag to the XMPP server it just writes the whole iq stanza into the *in* file of *sj*. Every iq stanza has an *id* attribute by which it is identified. When the iq response arrives at *iqd*, it opens a file with the name of this *id* and writes the whole answering iq stanza into it. The extension program just opens this file and reads the answer.

With this mechanism, extension programs just have to deal with file handling and they have to known how to handle their XML tags. This allows to write portable extension programs without any other requirement.

3 Interfaces

This section describes the front end and back end interfaces of the sj tools suite.

3.1 Front end

All user interfaces used to chat over ii[8] or Ratox[9] should also work with sj. Like ii *messaged* provides an *in* and *out* file to communicate with the front end programs. In order to utilize the possibilities of the XMPP protocol this interface has to be extended. To handle presence information of the user and its contacts, the two files *presence* and *mypresence* have been defined during the work on sj. The extensions of XMPP provide further possibilities like the presence status. For example, there are mechanisms to handle avatar pictures or information about the mood of a user.

To keep front end programs generic and usable, this filesystem based interface should be standardized. This way programs for other chat protocols are able to use the same features with the same user interface programs.

3.2 Back end

Like other network programs, an XMPP client has to deal with IPv4 and IPv6 sockets, domain names and ports as well as traffic encryption. If a program additionally needs to handle TLS[10] certificate validation or needs to make use of proxy servers, than just this part becomes a monster.

[8] http://tools.suckless.org/ii/
[9] http://ratox.2f30.org/
[10] Transport Layer Security

Using the Unix Client Server Programming Interface (UCSPI)[11], the sj program is able to outsource this infrastructure to other programs. The domain name is already resolved, the network connection established and the TLS certificate validated by the UCSPI tool suite[12], just before the sj program is started. sj simply uses the two pre-opened file descriptors six and seven to communicate with the XMPP server.

4 Future work

This chapter describes tasks which have to be done in order to move this approach into a usable program suite.

4.1 Incoming iq queries

To handle incoming iq queries, the *iqd* has to read the name space of the tag inside of iq requests. With this information *iqd* is able to launch a program that is able to handle this kind of request.

4.2 new chat contacts

Thus far, the *messaged* program is unable to detect new chat contacts. A portable mechanism to signalize or detect new directories should be implemented. This problem should be solved for user interfaces, too.

4.3 Integration of OTR

"Off the record" (OTR)[13] is a widely used mechanism to provide private communication over XMPP and other chat protocols. A generic solution should be implemented to utilize this encryption protocol for other ii-like chat programs, too.

4.4 User front end integration

sj, ii and ratox are just back end tools. To make them usable for end users, many front end programs for GUI and terminal have to be implemented. Three proof of concepts for terminal[14], X11[15] and web[16] environments were implemented for

[11] http://cr.yp.to/proto/ucspi.txt
[12] https://github.com/younix/ucspi
[13] https://otr.cypherpunks.ca/
[14] https://github.com/younix/cii
[15] https://github.com/younix/xii
[16] https://github.com/younix/wii

the ii-like file system based chat front ends. Every named program represents just one chat session. The missing part is some glue which connects these user front ends.

5 Appendix

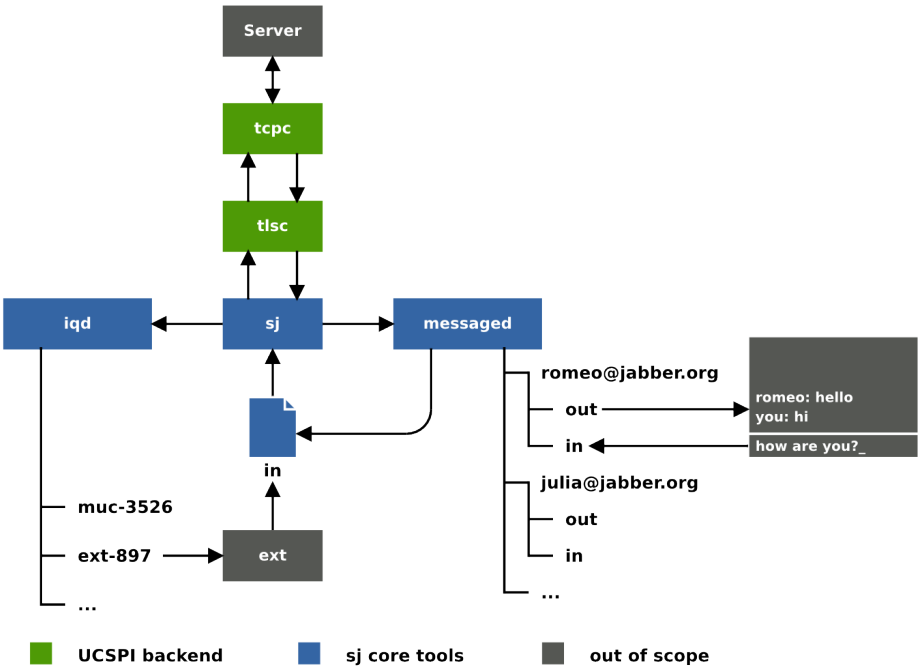

Figure 1: sj communication structure