Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Damiano Boppart

# XMPProbe

XMPP Server Monitoring

Semester Thesis SA-2013-66
December 2013 to February 2014

Advisors: Xenofontas Dimitropoulos, Brian Trammell
Supervisor: Prof. Dr. Bernhard Plattner

## Abstract

The *Extensible Messaging and Presence Protocol* is an open, widely adopted protocol for instant messaging. In practice, many operators that offer the use of their XMPP server to their users free of charge do not manage to provide a very reliable service. In addition, different software implementations of XMPP cause systematic connectivity issues between certain clients and servers, or between certain servers. And yet, so far there are no tools known to the authors that allow an end user to monitor the reliability or availability of a server, or systematically attempt to uncover incompatibilities. We design and implement such a tool and discuss measurements made with it. We conclude that service quality indeed varies drastically between different servers.

**Acknowledgements**

I would like to thank Prof. Plattner, Fontas and Brian for letting me have a go at a project idea of my own and for their support throughout this thesis, to Raoul for his stimulating input on the SSD that led to an excellent solution and his graph images, to Volker for showing me that PIP solves everything (yes, *everything*) and to Silvio and Christoph for their remarks on the draft of this paper.

# Contents

# Chapter 1

# Introduction

*XMPP* (the *Extensible Messaging and Presence Protocol* also commonly known as *Jabber*[1]) is an open communications protocol. It is one of the dinosaurs of instant messaging protocols. Originally developed in 1999, it was standardized by the *Internet Engineering Task Force* starting in 2002[2]. Continued development of the protocol is ongoing today, and is coordinated by the XMPP Standards Foundation (*XSF*).
XMPP has a userbase of over ten million people worldwide[3]. Various large companies that offer instant messaging services are compatible with XMPP, including Google (with *Google Talk*), AOL (with its *AOL Instant Messenger*), Facebook (with *Facebook Chat*) and Microsoft (with *Microsoft Messenger Service* and *Skype*).
Despite this wide adoption, very little academic research into how Jabber fares in practice has been conducted.

## 1.1 Motivation

The motivation for this thesis comes from a very practical need of one of the authors of this thesis: the need for an XMPP server that offers excellent reliability. Since XMPP is a communication medium of paramount importance for him, sub-par service of his service providers used so far have left him dissatisfied. Countless servers are available that allow any user to create and use an account with them, free of charge. Naturally, the question of which provider offers the best service poses itself. It turns out that so far no-one has made a serious attempt to benchmark the reliability of the *freely usable servers*[4]. Thus the idea for this thesis was born: create a tool that allows an end-user to find a particularly reliable Jabber server.

## 1.2 The Task

This thesis investigates the *availability*[5] of free XMPP servers. Personal experience from one of the authors has shown that some of these freely available servers commonly suffer from brief outages. Unfortunately, there is currently no monitoring software available that would allow to capture the state of these servers such that objective assertions about availability can be made.

[1] The terms *XMPP* and *Jabber* are used interchangeably throughout this thesis.
[2] As of 2011, XMPP is discussed in RFC 3922 http://tools.ietf.org/html/rfc3922, RFC 3923 http://tools.ietf.org/html/rfc3923, RFC 6120 http://tools.ietf.org/html/rfc6120, RFC 6121 http://tools.ietf.org/html/rfc6121 and RFC 6122 http://tools.ietf.org/html/rfc6122.
[3] According to the XSF: http://xmpp.org/xsf/press/2003-09-22.shtml
[4] By this term we mean Jabber servers that allow any Internet user to create an account and use it free of charge.
[5] Or, put differently: *uptime* or *reliability*

We therefore implement a monitoring tool, that measures the availability of XMPP servers. This new monitoring application allows to determine availability and functionality of features of an XMPP server that an end user commonly uses.
Firstly, this monitoring application is designed to be used for research, as one tool of a toolbox that allows to paint a more comprehensive picture of the landscape of the Internet as far as XMPP is concerned. Secondly, this application is useful for system administrators that operate XMPP servers since it allows them to be alerted to issues with the service they provide. Thirdly, this tool empowers end users to inform themselves about the quality of service their Jabber server of choice provides.
Using this monitoring tool, we collect data about the state of a number of freely usable hosted XMPP servers.

## 1.3 Related Work

The available methods and tools to monitor XMPP servers can be classified into two categories:

**Remote tools** This category contains the tools that interface with the XMPP server only by way of the XMP Protocol. There is no requirement to run such tools on the machine that runs the XMPP server, and no special interface or feature needs to be configured or present on the server.

**Local tools** This category contains the tools that are designed to interface with the server not necessarily through XMPP, but through other means as well. This usually means that they are server implementation specific, and also that it might be required to install them on the machine the server runs on.

The following two subsections showcase the tools of the respective categories that we investigated.

### 1.3.1 Remote Tools

*XMPPoke* is a tool that falls into this category. XMPPoke is free software, and is available on their website [1].
XMPPoke is a tools for “testing the encryption strength of XMPP servers” [1]. It “can test the TLS configuration and the DNSSEC deployment of XMPP servers, give warnings about issues with certificate chains, show the list of ciphersuites used by a server and their strength, check DANE records, and [other features]” [2].
The *IM Observatory* at `https://xmpp.net/` showcases what XMPPoke has to offer.
XMPPoke aims to provide an easy way for end users to judge the security level of a public XMPP server by assigning a grade to the security of client-to-server and server-to-server connections, respectively.
The IM Observatory serves as ongoing progress report to the implementation of the *Jabber Manifesto* [5]. The signing parties of this declaration of intent aim to “establishing ubiquitous encryption over our network [of XMPP servers by] May 19, 2014.” [5]
However, XMPPoke exclusively deals with security aspects of communication with XMPP. Features such as testing that instant messaging functionality is in fact in working order for a given server are not part of its scope.

### 1.3.2 Local Tools

Due to their nature, local tools are actually not in the scope of this thesis. However, for completeness’ sake, one representative for this category is illustrated here.

*munin-prosody*[6] is a plugin for the widely used monitoring software Munin[7]. This plugin works only for the XMMP server software Prosody[8]. The communication between Munin and Prosody happens through Prosody's text-based maintenance console.
XMPP Server software implementation specific monitoring are obviously limited to only working with specific XMPP servers.

## 1.4 Overview

The basic terminology and architecture of XMPP is introduced in Chapter 2.
The observed issues with XMPP in practice are outlined in Chapter 3. The occurrence of the issues described here is what we hope to quantify using the tool we developed.
An overview and in-depth, technical discussion of the monitoring application we developed is presented in Chapter 4.
The use of our tool, and the insights gained from the data it collects are investigated in Chapter 5.
Finally, in Chapter 6 we give our conclusion and an outlook.

[6]https://github.com/jarus/munin-prosody
[7]Munin; a networked resource monitoring tool; http://munin-monitoring.org/
[8]Prosody; a modern XMPP communication server; http://prosody.im/

# Chapter 2

# Jabber Basics

This chapter introduces some of the basic concepts and the terminology of XMPP.

## 2.1 Architecture

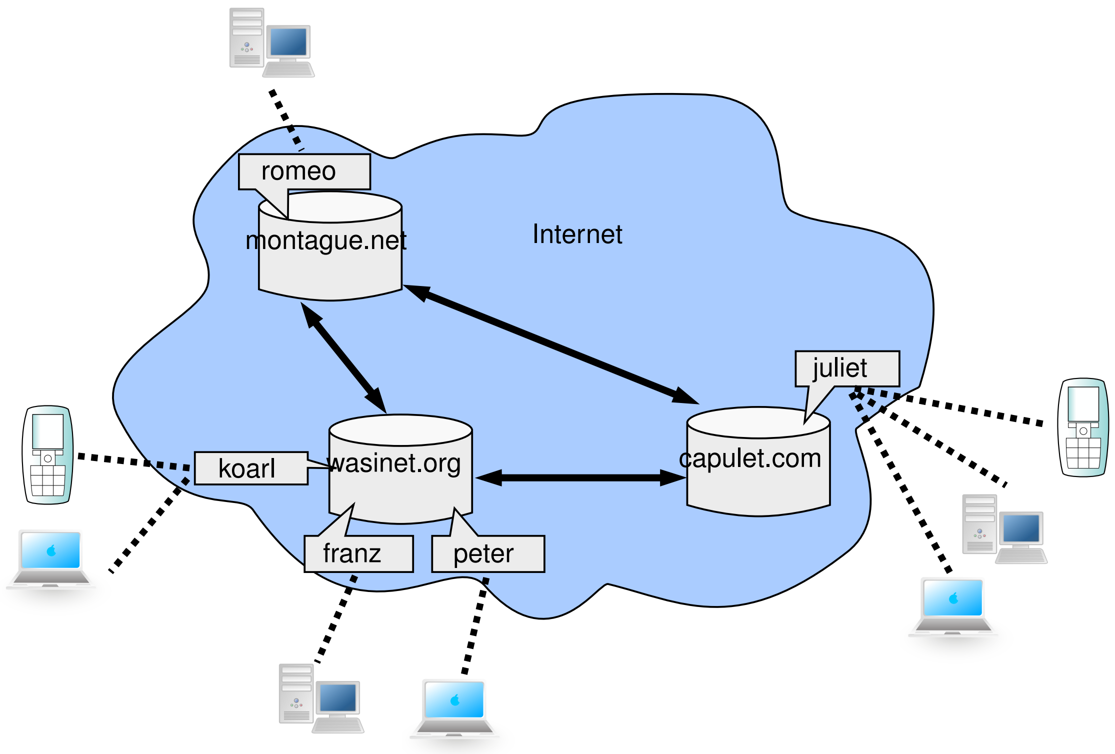

Figure 2.1: A View of the XMPP Network [4]

XMPP is built on a decentralized client-server infrastructure. The servers connect to their clients on one side, and to other servers on the other. There is no hierarchical distinction between servers.
In Figure 2.1 the different components of the XMPP network are shown:

**Servers** montague.net, wasinet.org and capulet.com.

**User accounts** romeo on montague.net, koarl, franz and peter on wasinet.org and juliet on capulet.com.

**Clients** The various desktop computers, laptops and phones, each associated with one account.

In a similar fashion to email (by way of *SMTP*[1]), an end user sends messages with his client software over servers acting as relays to reach an other user. So, a message from romeo to juliet is routed through romeo's home server montague.net, then passed on to the recipients home server capulet.com and from there to juliet's client. Unlike with email, the route from a user's home server to the recipient's home server is always direct. A message will therefore take at most three hops on the XMPP layer to be delivered to the recipient.
A server operator configures his server to either allow or deny his users to connect to users on other XMPP servers. If users are allowed to contact other servers, the server is called *federated*.

## 2.2 Addressing

Entities on an XMPP network all have a unique address. XMPP relies on the *Domain Name System* for address resolution. Identifiers of users, so called *Jabber IDs* (abbreviated JIDs) resemble email addresses: koarl@wasinet.org.
Each JID contains a domain name that identifies the XMPP server the account is registered to, wasinet.org in the above example. The user part of the JID, koarl in this example, uniquely identifies an account on a server. Jabber IDs of the form user@domain.tld are referred to as *bare* JIDs.
When an end user connects to the XMPP network with his client software, his JID will have a so-called *resource identifier* added to the Jabber ID. The resource part of the JID identifies a particular instance of client software. A user may be connected using an arbitrary number of clients simultaneously. In the above example, Koarl runs a Jabber client on both his phone and his laptop, and so the Jabber IDs that identify these two "end-points" might be koarl@wasinet.org/myphone and koarl@wasinet.org/computer. JIDs that include resource names, and thus identify an "end-point" and not just an account, are referred to as *full* JIDs. Each full JID (and the corresponding XMPP client it stands for) has individual presence information and capabilities.

## 2.3 Presence Information

*Presence* (referred to in many clients as *Status*) is one of the fundamental building blocks of XMPP. Presence information offers the possibility for a user to share information about his availability and willingness to engage in communication with other users. XMPP defines the four basic availability statuses "chatty", "away", "extended away" and "do not disturb". Sharing presence information with another user requires preceding explicit consent. Granting this consent is called *presence subscription*.
For example, for franz to be able to see peter's presence information a two-way exchange is required: franz must send a presence subscription request to peter, who must reply with an affirmative answer. Sharing presence information is not inherently bi-directional, but most XMPP client software implements the authorization of presence subscription requests in such a way that information will be shared both ways after one user has initiated presence subscription in one direction[2].

[1] The *Simple Mail Transfer Protocol*

[2] That is, after having authorized franz to access his presence information, peter's client automatically sends a presence subscription request to franz.

The set of accounts that one user receives presence information from is referred to as the *roster* or *buddy list*. A user's server keeps track of the state of presence subscriptions. When the state of a presence subscription changes, or on request, the server provides the connected clients with an updated roster. The roster is managed by the client, but stored on the server.
The presence information is considered by a server when deciding which end-point of one account a message should be sent to. If a message defines a bare JID as the recipient, then the servers choses the resource with the highest priority. If a message defines a full JID as the recipient, then the specified resource will receive the message. If the specified resource can not be reached, then the server caches the message for later delivery or returns an error. If a message only has a bare JID as a recipient and no resources are available, then the server caches the message for later delivery or returns an error.

## 2.4 Protocol Extensions

As the “extensible” in XMPP suggests, the protocol is designed to have features added. Extensions may of course be designed by anyone, but extensions published by the XMPP Standards Foundation enjoy particularly wide-spread adoption. The XSF published standard extensions are called *XMPP Extension Protocol* or *XEP*.
Standards may be purely procedural[3], informational[4], or add new features to XMPP[5].
Server software that does not support numerous XEPs out of the box and just offers “pure” RFC-compliant XMPP is more the exception than the rule[6].

[3] For example `XEP-0001`: The standards process followed by the XMPP Standards Foundation.

[4] For example `XEP-0160`: Best practices to be followed by XMPP servers in handling messages sent to recipients who are offline.

[5] For example `XEP-0166`: An extension for initiating and managing peer-to-peer media sessions between two XMPP entities with session management semantics compatible with SIP.

[6] An overview of the status of the support of various XEPs for XMPP server software can be found at `http://en.wikipedia.org/wiki/Comparison_of_XMPP_server_software`

# Chapter 3

# Problem Statement

In everyday use, issues with using Jabber occur frequently. Examples include:

1. Every so often a Jabber account might be disconnected for a couple of minutes. Jabber client software will usually make the user aware of this, as shown in Figure 3.1.
2. Individual messages of a longer conversation get lost when one of participants goes offline for a short time.
3. More subtle issues surround the propagation of presence information. In Figure 3.2, the client is configured to use two different Jabber accounts at the same time. Each receives presence information from one other account shown twice in the buddy list as *P*. The presence information (the account is currently *Away*) should be the same for both buddies in the roster, and yet one buddy is correctly shown as being away while the other seems to be offline.
4. Some connectivity issues don't just involve individual accounts, but all communication between two particular servers. What this can look like is shown in Figure 3.3. For all users that have accounts at one particular domain, the presence information reads `404: Remove Server Not Found`[1].

Generally, it is not clear which of the involved XMPP implementations (the user's client, the user's server, the buddy's server or the buddy's client) is at fault when issues like this occur. The monitoring tool we develop helps in pinpointing the origin of the problem for such issues.

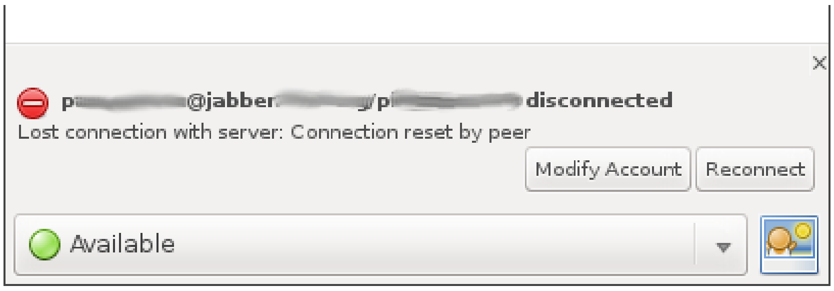
p[redacted]@jabber.[redacted]/p[redacted] disconnected
Lost connection with server: Connection reset by peer

Figure 3.1: Failed Connections Attempt

[1] In this particular case it was revealed that an upgrade of the software on the seemingly "not found" server removed some workaround for incorrectly implemented features in the author's server that subsequently generates these error messages.

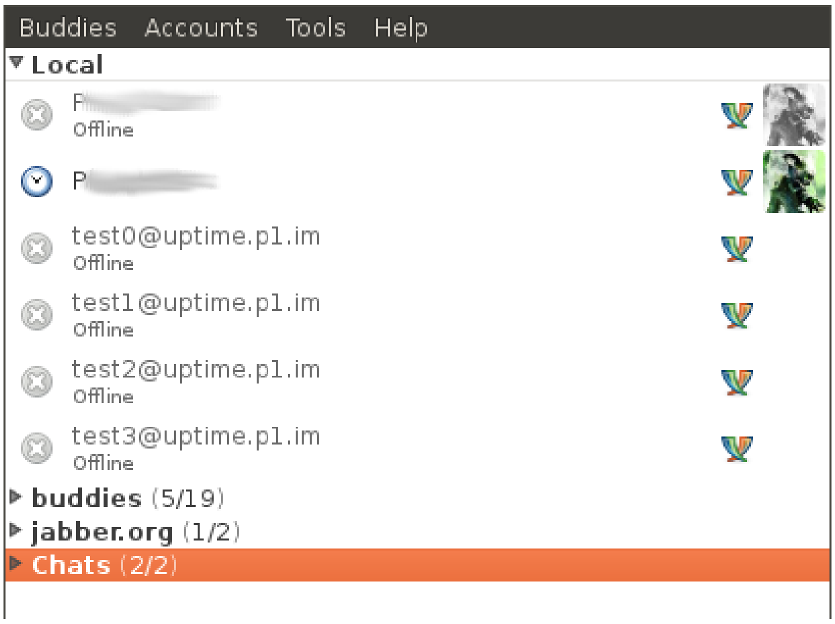
Buddies Accounts Tools Help
Local
P
Offline
P
test0@uptime.pl.im
Offline
test1@uptime.pl.im
Offline
test2@uptime.pl.im
Offline
test3@uptime.pl.im
Offline
buddies (5/19)
jabber.org (1/2)
Chats (2/2)

Figure 3.2: Some Presence Subscriptions Are More Equal Than Others

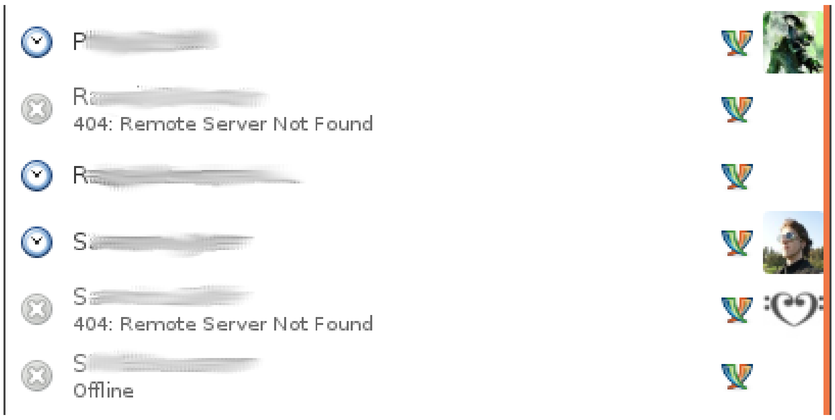
P
R
404: Remote Server Not Found
R
S
S
404: Remote Server Not Found
S
Offline

Figure 3.3: Swissjabber Connectivity Issues

# Chapter 4

# XMPProbe

The tool we developed, *XMPProbe*, allows the monitoring of certain features of a Jabber server, and so can detect and document issues such as the ones described in Chapter 3. Collecting information to pinpoint the exact problem is the first step in solving it.
The information that XMPProbe collects should allow its user to determine:

- How frequently certain problems manifest,
- How long they persist, and
- What domain is affected by a problem.

XMPProbe interacts with the Jabber servers it monitors exclusively over Jabber. That is, it is not dependent on any features specific to particular server software implementation. Furthermore, no special configuration or support by the server operator is required to use XMPProbe, which makes it easy to deploy. Most importantly, this allows any end user to do the monitoring by herself.

## 4.1 Overview

XMPProbe is designed to evaluate one of the fundamental features of an XMPP server: sending instant messages.
XMPProbe essentially mimics the typical behavior of an end user: Having registered a Jabber account with a server, a user adds some of her friends to her buddy list, and will occasionally send them instant messages and receive messages in response. In essence, XMPProbe is nothing more than an implementation of a Jabber client software, albeit one that doesn't allow it's user to type the messages to be sent herself.
In a nutshell, XMPProbe takes some configuration information (including a list of Jabber account credentials) from the user, periodically exchanges messages between these accounts and creates log files of what has happened for offline analysis.
The following sections each go into more detail about one aspect of the architecture of XMPProbe.

## 4.2 Backend

Since the authors are proficient in *Python*[1], and a variety of implementations of XMPP are available in Python, we decided to write all code for this thesis in Python.

[1]The Python programming language: http://www.python.org/

The following list gives an overview over the various XMPP libraries[2] considered for the use in XMPProbe.

**jabber.py**[3,4] Python 2.0+. GPLv2 License. Last release in 2003-11.

**xmpppy**[5,6] Python 2. GPLv2 License. Last code change in 2009-04.

**headstock**[7,8] Python 2.5+. BSD License. Last code change in 2011.

**PyXMPP**[9,10] Python 2.6. LGPL 2.1 License. Last code change in 2011-08.

**PyXMPP2**[11,12] Python 2.7 (using automated conversion also 3.2). LGPL 2.1 License. Last release in 2013-10. “This code is far from being complete and is not actively developed.”

**Wokkel**[13,14] Python 2.4+. MIT License. Last stable release in 2013-01.

**SleekXMPP**[15,16] Python 2.6+ or 3.1. MIT License. Last release in 2014-02.

**Twisted**[17,18] Python 2.6+. MIT License. Last release in 2013-11. “Twisted is an event-driven networking engine” and not actually a library with any XMPP-specific features implemented.

**GAE XMPP Python API**[19] Python 2.7. Proprietary License. Google App Engine is a PaaS platform, and this is not an XMPP library that can be used like the others.

We had three main criteria for the evaluation of the available libraries:

**Python 3 Support** Python 3.0 was released at the end of 2008. As a rule of thumb, we assume that any library that has not managed to be compatible with Python 3 after 5 years has either been abandoned or is at the very least not maintained enthusiastically any more. The release dates or update dates of the libraries above are consistent with this rule of thumb.

**XEPs Supported** The selection of XEPs supported by each library, as well as the work needed for adding support for additional XEPs was considered. A wide variety of XEPs is actually supported and in and used by many server and client implementations, and a library that offers “bare” RFC-compliant XMPP would not allow us to test any functionality introduced in XEPs easily.

**Documentation** Given the short duration of this Semester Thesis, we needed a library with documentation that allows swift implementation of a working product.

[2]The XSF maintains a list of libraries for various programming languages at http://xmpp.org/xmpp-software/libraries/
[3]jabber.py Code: http://sourceforge.net/projects/jabberpy/files/
[4]jabber.py Documentation: http://jabberpy.sourceforge.net/
[5]xmpppy Code: http://sourceforge.net/projects/xmpppy/files/
[6]xmpppy Documentation: http://xmpppy.sourceforge.net/
[7]headstock Code: https://github.com/Lawouach/headstock
[8]headstock Documentation: http://www.defuze.org/oss/headstock/docs/0.4.1/
[9]PyXMPP Code: https://github.com/Jajcus/pyxmpp
[10]PyXMPP Documentation: http://pyxmpp.jajcus.net/pyxmpp.html
[11]PyXMPP2 Code: https://github.com/Jajcus/pyxmpp2
[12]PyXMPP2 Documentation: http://jajcus.github.io/pyxmpp2/api/
[13]Wokkel Code: http://hg.ik.nu/wokkel
[14]Wokkel Documentation: http://wokkel.ik.nu/
[15]SleekXMPP Code: https://github.com/fritzy/SleekXMPP
[16]SleekXMPP Documentation: http://fritzy.github.io/SleekXMPP/
[17]Twisted Code: http://twistedmatrix.com/trac/browser
[18]Twisted Documentation: http://twistedmatrix.com/trac/
[19]GAE XMPP Python API Documentation: https://developers.google.com/appengine/docs/python/xmpp/

Given these criteria, only two viable candidates of the above list remained: Wokkel and SleekXMPP. In the end, we selected SleekXMPP because it supports more XEPs and the documentation is of high quality.

## 4.3 Prerequisites of XMPProbe

XMPProbe monitors a given Jabber server by mimicking the interaction with that server that a normal user has. Like a normal user, XMPProbe needs to have a working Jabber account on the server in question, and a networked machine to run her client software on.
Originally, XMPProbe was designed to register the required accounts on the servers to be monitored itself[20]. However, it turns out that this approach is not viable in practice. Many XMPP servers do not allow in-band registration as a matter of policy. Of those servers that do, many require interaction in the form of solving a CAPTCHA or verifying account creation by using a code sent by email. Also, the servers that easily allow automated in-band registration typically have a limit of only allowing one new account per 24 hours and IP address to be registered. Many server operators point out on their website that these restrictions have been put in place precisely because account creation is easy to automate otherwise, and some of them have suffered from spam and DDoS attacks as a result. The approach of having the user of XMPProbe provide a list of server names was thus abandoned in favor of the user having to provide a list of at least one working Jabber account per server to be monitored. This approach removes a lot of the complexity from setting up monitoring, and the user can control exactly what accounts are used for monitoring. In addition to account credentials, the user also has to provide the Jabber resource used for all instances of XMPProbe and some more options regarding logging. The complete specification of the required information to run XMPProbe is detailed in Appendix A.
By using full Jabber IDs for all chat messages, XMPProbe ensures that the messages sent between a set of accounts being used for monitoring do not affect the use of these accounts for other resources. In particular, this means that a user can use her account "normally" while using it for monitoring with XMPProbe at the same time without any messages being routed to the "wrong" connected client. This use of full Jabber IDs also allows a user to use one account for multiple independent monitoring tasks with XMPProbe. An application of this would be to run multiple monitoring tasks of XMPProbe on different physical machines, so that problems and outages specific to one machine do not result in missing log data (since for the "gap" in data caused by one machine data from another machine can be used). In fact, XMPProbe runs a separate instance of itself per account it monitors. Also, one monitoring task can be split up to run on more than one machine to work around resource constraints.
This "compartmentalization" of running XMPProbe instances by using full JIDs has its limits, though: since the roster is not particular to a connected client software, but to an account, the buddy list is shared between all resources. This means that the buddies added by one instance of XMPProbe will show up for all other instances (and in the end user's other connected clients, where applicable). This does however not impact the behavior or functionality of XMPProbe. XMPProbe does not rely on the content of the roster or presence information when deciding who to send messages to.

## 4.4 Basics

Given a list of accounts to monitor, XMPProbe derives a schedule of "conversations". Over time, each JID will exchange messages with each other JID in the set of accounts configured, thus testing connectivity for every possible pair of chat partners.
To reduce the complexity and the probability that issues in XMPProbe lead to wrong or incomplete data, XMPProbe was designed in a way that each instance (tending to one Jabber

[20] By using the in-band registration feature specified by XEP-0077.

account) running as part of a monitoring task works independently of all other instances and there is no control information that needs to be communicated between instances.
This means that the scheduling of these conversations that test connectivity between a pair of accounts (and therefore between different instances of XMPProbe) has to be deterministic. The schedule we use is given by a solution to the *Speed Speed Dating Problem*.

## 4.5 The Speed Speed Dating Problem

The Speed Speed Dating Problem (abbreviated SSD) is named after an illustrative story that was made up by one of the authors to have an obvious example of its application. The story goes as follows:
*You are organizing a speed dating event for $n$ bisexual people. Every attendee should have one conversation (or "date") with every other attendee at some point. All conversations have a fixed duration, and start synchronously. So, all people sit down in pairs for having a conversation simultaneously, and all people switch chairs simultaneously. At any given time, one attendee may only be part of up to one conversation. How can you as the organizer schedule the conversations in such a way that the complete event takes as little time as possible?*
A more formal description of the same problem can be given in terms of graph theory:
An algorithm that solves the Speed Speed Dating Problem is an algorithm that returns a set of matchings $M$ on the complete graph $K_n = V, E$ such that the union of all edges in the matchings in $M$ is equal to $E$:

$$K_n = \{V, E\}, \text{ a complete graph with } n \text{ nodes} \tag{4.1}$$

$$\forall m \in M, m \text{ is a matching on } K_n \tag{4.2}$$

$$\bigcup_{\forall m \in M} m = E \tag{4.3}$$

The cardinality of $M$ expressed in terms of $n$ is the complexity of the solution to the SSD.
A trivial algorithm that solves the problem is the following one:
Each matching in $M$ consists of exactly one edge, that is:

$$M = \{\{a\}, \{b\}, \{c\}, ...\} \forall a, b, c, ... \in E \text{ and } a, b, c, ... \text{ pair-wise unequal} \tag{4.4}$$

Or, more formally:

$$M = \{\{e\} \big| e \in E\} \tag{4.5}$$

Given that the complete graph $K_n$ has $\frac{n \cdot (n-1)}{2}$ edges, the complexity of this solution is:

$$\frac{n \cdot (n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = \mathcal{O}(n^2) \tag{4.6}$$

A theoretical lower bound can be derived using the assumption that every matching $m \in M$ is a maximum matching. A maximum matching contains $\lfloor \frac{n}{2} \rfloor$ edges. The lowest-possible complexity of a solution assuming every matching is a maximum matching is therefore:

$$\frac{\frac{n \cdot (n-1)}{2}}{\lfloor \frac{n}{2} \rfloor} = \begin{cases} \frac{\frac{n \cdot (n-1)}{2}}{\frac{n}{2}} = n - 1 = \mathcal{O}(n) & n \text{ even} \\ \frac{\frac{n \cdot (n-1)}{2}}{\frac{n-1}{2}} = n = \mathcal{O}(n) & n \text{ odd} \end{cases} \tag{4.7}$$

For an even number of nodes, all $n$ nodes are part of a maximum matching, for an odd number of nodes $n - 1$ nodes are part of a matching.
For XMPProbe, we designed an algorithm that is more efficient that the trivial approach outlined above.

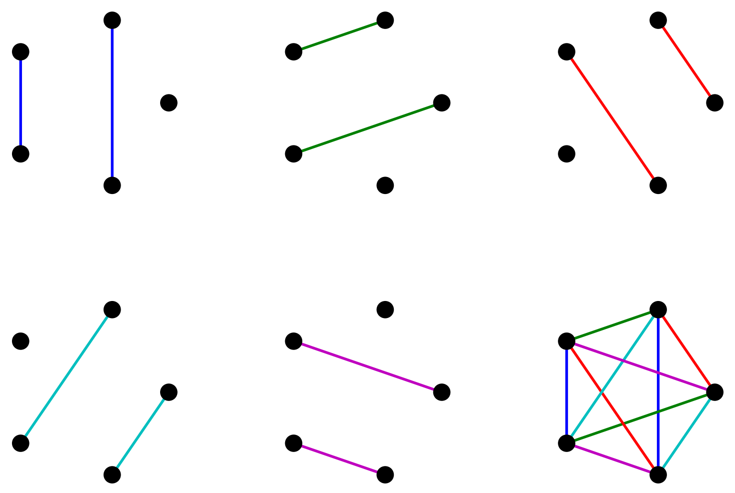
Figure 4.1: Matchings for $n$ odd.[3]

### 4.5.1 The Algorithm

To better visualize the algorithm, we arrange the $n$ nodes of the complete graph equally spaced along a circle line. We also define the *distance* between nodes along this circle line: Each node has two nodes closest on the circle line. These are the neighbors of distance 1. The next closest pair of nodes has distance 2, etc.
We distinguish between the cases of $n$ odd and $n$ even.

#### $n$ Odd

For an odd $n$, one node is not paired up with any other node for each matching. One matching of the solution is constructed in the following way:

1. One node is labeled as *inactive*.
2. Each possible pair of nodes with the same distance from the inactive node is communicating in this matching.

The complete set of matchings that make up a solution is constructed by executing above procedure $n$ times and picking a different node to be marked as inactive in the first step every time. In Figure 4.1 the visualization of this procedure is given for $n = 5$. The bottom right graph combines the 5 matchings depicted into the complete graph.

#### $n$ Even

For an even $n$, either all or all but two nodes are paired up with another node for each matching. The matchings of a solution are constructed in two different ways. The matchings with two idle nodes are constructed in the manner of the “odd” solution:

1. One node is labeled as *inactive*.

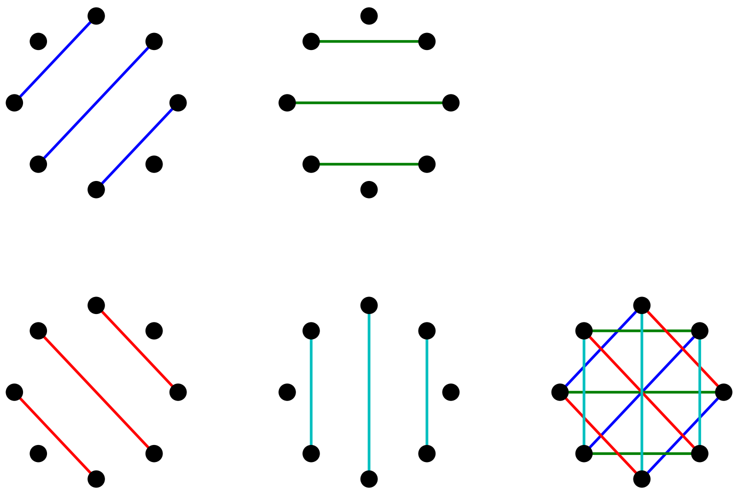

Figure 4.2: Matchings for $n$ even with two idle nodes per matching.[3]

2. Each possible pair of nodes with the same distance from the inactive node is communicating in this matching.

3. The one node with the highest distance from the inactive node is also labeled as *inactive*.

Half the set of matchings that make up a solution are constructed by executing above procedure $\frac{n}{2}$ times and picking a different node to be marked as inactive in the first step every time such that all nodes were marked as inactive exactly once after $\frac{n}{2}$ matchings.
In Figure 4.2, the visualization of this procedure is given for $n = 8$. The bottom right graph combines the 4 matchings depicted into one graph.
The remaining matchings with no idle nodes are constructed in the following way:

1. Label two neighboring nodes with *start*.

2. Each possible pair of nodes with the same distance to the closest node labeled *start* is communicating in this matching.

3. The two nodes with the highest distance to the closest node labeled with *start* are also labeled with *start*.

Half the set of matchings that make up a solution are constructed by executing the above procedure $\frac{n}{2}$ times and picking a different pair of starting nodes to be marked with *start* in the first step every time, such that every pair of neighboring nodes was labeled with *start* in exactly one matching after $\frac{n}{2}$ matchings.
In Figure 4.3, the visualization of this procedure is given for $n = 8$. The bottom right graph combines the 4 matchings depicted into one graph.
In Figure 4.4 these two different sets of matchings are combined into one graph, resulting in the complete graph $K_n$.

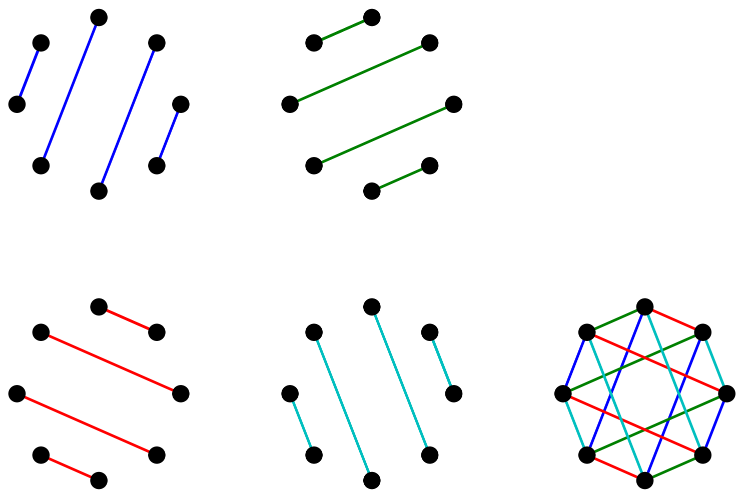

Figure 4.3: Matchings for $n$ even with no idle nodes per matching.[3]

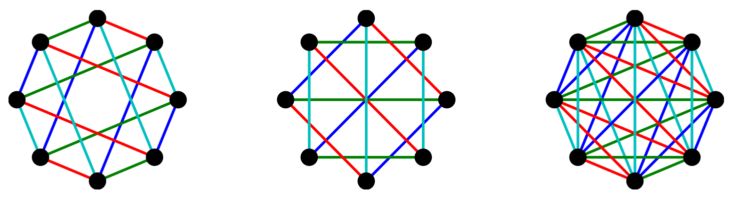

Figure 4.4: Matchings for $n$ even assembled from the two subsets of matchings.[3]

### 4.5.2 Implementation in XMPProbe

In XMPProbe the class `SpeedSpeedDating` provides an implementation of the above algorithm. The essential part of the algorithm implementation in Python is shown in Listing 4.1. The case for two nodes has to be handled separately.

```python
l = ['Alice', 'Bob', 'Mallory', 'Eve']
xs = []  # Aggregator
if len(l) == 2:
  xs = [l]
elif len(l) % 2 == 0:
  m = len(self._l) // 2  # Half the length of the agent list.
  for i in range(m):
    xs.append(l[i:] + l[:i])
  for i in range(m):
    xs.append(l[i + 1:i + m] + l[i + m + 1:] + l[:i])
else:
  for i in range(len(l)):
    xs.append(l[i + 1:] + l[:i])

matchings = [list(zip(x[:len(x) // 2], x[len(x) // 2:][::-1])) for x in xs]
```

Listing 4.1: Speed Speed Dating in Python

XMPProbe extends the Speed Speed Dating problem above by one small detail: We assume that even though for all conversations between two accounts messages are sent in both directions, that there might be a difference depending on who starts a conversation. That is, not only are we interested in having a conversation scheme where Alice strikes up a conversation with Bob every so often, in addition we want that Bob initiates a conversation with Alice.
We have implemented this by interpreting the edges of a matching as directed edges, and generating a "reciprocal" matching for each matching in the solution of the SSD: That is, for all matchings $m$ of the solution $M$ we add a matching $m\prime$ where all directed edges are reversed. An edge $(a, b)$ of $m$ is transformed to an edge $(b, a)$ in $m\prime$. By this extension, the size of our set of matchings is always $2 \cdot n$, but the order in which the matchings are parsed ensures that any two accounts have a conversation every $n$ rounds anyway.
A configuration parameter of XMPProbe defines a period, and the conversations of one matching are conducted every period.

## 4.6 Conversations

A conversation between two accounts consists of two messages: The initiating party, Alice, sends an "ask" message to the recipient party, Bob, who replies with an "answer" message.
A message is a serialized representation of a number of attributes. The following table gives the details on the attributes and their purpose:

**magic** A "magic number" identifying this message as being sent by XMPProbe.

**version** The version of the format of the message. Currently there is only version 1.

**nonce** A number picked randomly for the ask message of a conversation and incremented for every subsequent message of a conversation. The presence of the nonce makes it unlikely that messages from one conversation will ever be mistaken as being part of another conversation.

**t_sent** A timestamp of when the message was generated.

**t_rcvd** A timestamp of when the message was generated to which this message is a response to.

**report** The function of this message. Currently, the functions "ask" and "answer" are defined.

**comment** An arbitrary string that is passed along with each message. This is used to provide a sysadmin investigating an account used for monitoring with XMPProbe with contact information[21].

**checksum** A checksum of all the above fields for verifying the integrity of the attributes of a message.

Messages are serialized using JSON, because the serialized form has good readability and Python has good support for handling JSON.

## 4.7 Logging

All the activity of XMPProbe and the information it obtains about the servers it monitors are logged in three ways:

**Standard output** For human-readable log messages with very high verbosity.

**Log file** A duplicate of the standard output log messages, for later reference.

***CSV***[22] **file** For some very particular events only. The log file's format is completely specified, and the log files are suitable for machine-reading.

While the first two log types are only suitable for manual reference, the third type was designed specifically to make offline data analysis easy. The CSV file log contains information about 14 particular events that occur during the run of XMPProbe[23]. These events can be grouped into three categories:

**Online status** Events in this category represent connecting ("going online") and disconnecting ("going offline") of the Jabber account.

**Presence** Events in this category represent changes in presence subscription between two accounts.

**Instant Messaging** Events in this category cover sending an receiving chat messages of the format specified above as well as arbitrary unsolicited messages.

A discussion of what information can be extracted from the log files, as well as an example using collected data is given in Chapter 5.

## 4.8 Limitations

XMPProbe has limitations that fall into two categories: Features of the XMPP landscape that can not be monitored due to the way the protocol is designed, and issues with XMPProbe that can be remedied by extending the functionality of XMPProbe.

### 4.8.1 Issues Inherent to XMPP

Many XMPP service operators use load-balancing to be able to use multiple physical machines to run one virtual instance of an XMPP server. Load balancing is often implemented using DNS record priorities[24] or using transport layer load balancing. Since load balancing is transparent to clients (and for the most part to load-balanced server itself) it is impossible to reliably detect

[21] Assuming, of course, that the sysadmin in question can read the messages sent by the account
[22] *Comma-Separated Values* a more or less standardize format for tabular data.
[23] The details on all these events and the format of the log entries are presented in Appendix D.
[24] XMPP allows this to be implemented easily by design through its use of DNS SRV and TXT records

problems such as an outage of some (but not all) of the backend servers using monitoring on the XMPP layer.

Another problem is the standard-compliant behavior of servers regarding the handling of chat messages for offline recipients: Servers should cache messages for accounts that are offline and deliver them once the account connects again. This makes it impossible to measure the end-to-end path delay for messages without relying on information from the server. If a server wrongly assumes a client to be offline for a period of time, and all messages to that client are cached, then the path delay of those messages will be measured to be longer by the time the messages were cached. Indeed, unrealistically long path delays as a result of message caching have been observed in the dataset discussed in Section 5.1.

### 4.8.2 Issues Inherent to the Implementation of XMPProbe

Through the development we have seen that the library we use, SleekXMPP is not without faults: There are accounts that SleekXMPP can never connect to, even though other XMPP Clients can connect with the same account credentials. Network outages are "hidden" internally, and XMPProbe is not notified about them. These issues can most likely be addressed by setting up SleekXMPP differently than it is used now in XMPProbe or by minimal modifications to the library. A way of simultaneously addressing the network connectivity issue and gather additional useful information would be to extend XMPProbe to periodically send out XMPP *ping* messages[25] which would offer a way to verify connectivity as well as provide a separate way to measure path delay.

One assumption that the current implementation of XMPProbe makes is that the monitored XMPP servers are indifferent to the order in which they send messages to other servers as part of the pair-wise matching scheme that XMPProbe uses[26]. In other words, XMPProbe makes the assumption that the server behaves in exactly the same way when Alice talks to Bob first and then to Carol compared to when Alice talks to Carol before Bob. We do not actually know that this assumption holds true.

Currently, XMPProbe only tests that sending chat messages works correctly, but there are of course countless other interesting features of the XMP Protocol to investigate, such as the speed and correctness of the propagation of presence information or the time needed to connect to and authenticate with the server.

One way in which the completion of one set of conversations could be sped up (from the current $n$ rounds) would be to "collapse" multiple conversations into one time slot. This feature is already partially implemented. The implementation of the SSD used by XMPProbe returns a list of conversation partners for each round, and at the moment the length of the list is capped to one. By increasing the length of this list to $k$ (and thus having each account be involved in $k$ conversations per period) the time needed to have a conversation with every buddy could be sped up significantly.

Lastly, another feature that would make XMPProbe even more interesting to use would be if it offered a way to crawl the web to find additional XMPP servers to monitor, as opposed to relying on a user-provided list of accounts[27].

[25]XMPP layer ping messages are standardized in `XEP-0199`.

[26]That is, the order in which the matchings constituting a solution to the SSD problem are parsed

[27]Since XMPP uses TCP port numbers officially recognized by the IANA in addition to identifying DNS records, automatic identification of XMPP servers is reasonably possible.

# Chapter 5

# Measurements

Using XMPProbe we ran a test using 58 Jabber accounts on 30 different servers. Each server had one or two registered accounts[1]. The test ran for 17+[2] hours with a period of 10 seconds. The interval between repeat communication of any pair of accounts was thus:

$$10\text{seconds} \cdot 58\text{accounts} = 9\text{minutes}40\text{seconds} \tag{5.1}$$

The test generated a total of 379.6MiB of log file data, of which 81.2MiB was in CSV log files. All instances of XMPProbe ran on one machine, since this allowed for most accurate possible path delay measurements. No activity that might have disrupted data collection was detected on the machine for the duration of the test.
For data analysis, we developed the Python script `analysis.py`. The details about the script are discussed in Appendix E.
The following section shows and comments the information extracted from the dataset.

## 5.1 Analysis of the Dataset

The total number of 582198 records is split up by event as shown in Table 5.1. An overview over the percentage of correctly answered messages per account is given in Table 5.2. In Table 5.3, the numbers for selected event types are given by account.
Of the 58 accounts, 51 were able to connect successfully (i.e. a `connect` event was registered on each of these accounts) and send ask and answer messages.
By the data shown in Table 5.3 we can separate the remaining 7 accounts into two categories:

**Many disconnects** Accounts in this category disconnect on average a lot more frequently than once per period. This behavior is a bug in XMPProbe, since XMPProbe is designed to reconnect at most once per period. The authentication is handled entirely by SleekXMPP.

**Few disconnects** Accounts in this category either have incorrect information in the configuration file, or SleekXMPP has trouble for unknown reason for authenticating with this particular account.

The small number of single authentication failure events registered are of no concern as they all immediately precede a successful connection attempt. We regard these `nauth` events as flukes.
Since we have 51 working accounts in this data set (that originally had an empty roster), we expect each possible pair of these accounts to subscribe to each other's presence information.

[1] We try to take nothing for granted, including that communication between two accounts on the same server works correctly.

[2] Specifically, the log entries were made between 2014-02-26 06:08:10.205568 UTC and 2014-02-26 23:13:31.225412 over a duration of 17:05:21.019844 hours.

Given that SleekXMPP handles reciprocal presence subscriptions[3] automatically in XMPProbe, we expect to see one `rsubscribe` event for each possible pair:

$$\left.\frac{n \cdot (n-1)}{2}\right|_{n=51} = \frac{51 \cdot (51-1)}{2} = 1275 \tag{5.2}$$

According to the value of `rsubscribe` events in Table 5.1 we calculate that 94.51% of the expected subscription requests actually happened correctly.

| Event | Count |
|---|---|
| connect | 51 |
| disconnect | 92896 |
| nauth | 19 |
| ranswer | 106582 |
| rask | 108599 |
| rsubscribe | 1205 |
| rsubscribed | 554 |
| rsurprise | 33 |
| runknown | 13985 |
| sanswer | 108599 |
| sask | 140553 |
| skip | 7341 |
| ssubscribe | 1781 |
| ssubscribed | 0 |

Table 5.1: Event Count by Type

The correctly working accounts all sent out around 2400 ask messages each, of which between 7.19% and 99.88% were answered correctly per account. At this stage of maturity of XMPProbe we are not concerned about the quality of service of low-scoring servers in this metric. The low values are more likely due to some unlucky bug affecting XMPProbe only in combination with particular XMPP server software implementations than due to low service quality of the respective servers. We can however say that in the best-case scenario XMPProbe is definitely not responsible for dropping a significant amount of messages: rates go as low as around 1 failure to answer per 600 messages for the best of accounts.
For the events of the types `rask`, `ranswer` and `rsurprise` that signify the reception of a valid message, the path delays have the following statistical metrics:

**minimum** 0.012371s

**5th percentile** 0.026598s

**50th percentile** 0.0760875s

**95th percentile** 0.318321s

**maximum** 6383.632378s

Since we send messages between Jabber account residing on the same server as well as half-way around the globe[4] the interval given by the minimum and the 95th percentile seem perfectly reasonable. The maximum value of over 1:45 hours suggests that XMPP servers do indeed cache messages to be delivered later for accounts that are offline, as is defined in the protocol specification.

[3]That is, if Alice successfully subscribes to Bob's presence information, then Alice will automatically grant Bob access to hers in return.

[4]From Germany to New Zealand, for instance.

| JID | asks_received | answers_sent | success rate | remaining_asks |
|---|---|---|---|---|
| xmpprobe—0@jabber.ccc.de/longxmpprobe | 2407 | 2404 | 99.88% | 3 |
| damiano1@koalatux.ch/longxmpprobe | 2431 | 2427 | 99.84% | 4 |
| xmpprobe—1@jabber.meta.net.nz/longxmpprobe | 2430 | 2426 | 99.84% | 4 |
| xmpprobe—1@jabber.ccc.de/longxmpprobe | 2424 | 2420 | 99.83% | 4 |
| xmpprobe—0@im.apinc.org/longxmpprobe | 2418 | 2414 | 99.83% | 4 |
| xmpprobe—1@im.apinc.org/longxmpprobe | 2416 | 2412 | 99.83% | 4 |
| xmpprobe—0@jabber.meta.net.nz/longxmpprobe | 2403 | 2399 | 99.83% | 4 |
| xmpprobe—1@jabber.rueckgr.at/longxmpprobe | 2433 | 2428 | 99.79% | 5 |
| damiano0@koalatux.ch/longxmpprobe | 2432 | 2427 | 99.79% | 5 |
| xmpprobe—1@jabb3r.de/longxmpprobe | 2419 | 2414 | 99.79% | 5 |
| xmpprobe—0@jabb3r.de/longxmpprobe | 2413 | 2408 | 99.79% | 5 |
| xmpprobe—1@jabber.minus273.org/longxmpprobe | 2433 | 2427 | 99.75% | 6 |
| xmpprobe—0@creep.im/longxmpprobe | 2421 | 2415 | 99.75% | 6 |
| xmpprobe—1@creep.im/longxmpprobe | 2412 | 2406 | 99.75% | 6 |
| xmpprobe—0@jabber.minus273.org/longxmpprobe | 2402 | 2396 | 99.75% | 6 |
| xmpprobe—0@jabber.no-sense.net/longxmpprobe | 2402 | 2396 | 99.75% | 6 |
| xmpprobe—0@jabberafrica.org/longxmpprobe | 2403 | 2351 | 97.84% | 52 |
| xmpprobe—0@jabme.de/longxmpprobe | 2401 | 2349 | 97.83% | 52 |
| xmpprobe—1@jabberafrica.org/longxmpprobe | 2436 | 2383 | 97.82% | 53 |
| xmpprobe—1@jabme.de/longxmpprobe | 2437 | 2383 | 97.78% | 54 |
| xmpprobe—0@draugr.de/longxmpprobe | 2418 | 2310 | 95.53% | 108 |
| xmpprobe—1@draugr.de/longxmpprobe | 2412 | 2303 | 95.48% | 109 |
| xmpprobe—1@coderollers.com/longxmpprobe | 2410 | 2300 | 95.44% | 110 |
| xmpprobe—1@jabber.de/longxmpprobe | 2428 | 2317 | 95.43% | 111 |
| xmpprobe—0@jabbim.cz/longxmpprobe | 2400 | 2290 | 95.42% | 110 |
| xmpprobe—0@coderollers.com/longxmpprobe | 2420 | 2309 | 95.41% | 111 |
| xmpprobe—1@jabbim.cz/longxmpprobe | 2437 | 2325 | 95.40% | 112 |
| xmpprobe—0@jabber.de/longxmpprobe | 2407 | 2296 | 95.39% | 111 |
| xmpprobe—1@xmppnet.de/longxmpprobe | 2435 | 2253 | 92.53% | 182 |
| xmpprobe—0@jabber-br.org/longxmpprobe | 2410 | 2198 | 91.20% | 212 |
| xmpprobe—1@jabber-br.org/longxmpprobe | 2419 | 2206 | 91.19% | 213 |
| xmpprobe—1@is-a-furry.org/longxmpprobe | 2415 | 2199 | 91.06% | 216 |
| xmpprobe—0@is-a-furry.org/longxmpprobe | 2415 | 2199 | 91.06% | 216 |
| xmpprobe—1@tigase.im/longxmpprobe | 2431 | 2180 | 89.68% | 251 |
| xmpprobe—0@tigase.im/longxmpprobe | 2402 | 2150 | 89.51% | 252 |
| xmpprobe—0@alpha-labs.net/longxmpprobe | 2425 | 2109 | 86.97% | 316 |
| xmpprobe—1@alpha-labs.net/longxmpprobe | 2405 | 2089 | 86.86% | 316 |
| xmpprobe___0@ch3kr.de/longxmpprobe | 2429 | 1959 | 80.65% | 470 |
| xmpprobe___1@ch3kr.de/longxmpprobe | 2428 | 1958 | 80.64% | 470 |
| xmpprobe—0@chatme.im/longxmpprobe | 2433 | 1942 | 79.82% | 491 |
| xmpprobe—1@chatme.im/longxmpprobe | 2418 | 1917 | 79.28% | 501 |
| xmpprobe—0@blah.im/longxmpprobe | 2422 | 1904 | 78.61% | 518 |
| xmpprobe—1@blah.im/longxmpprobe | 2405 | 1889 | 78.54% | 516 |
| xmpprobe—1@jabber.iitsp.com/longxmpprobe | 2417 | 1897 | 78.49% | 520 |
| xmpprobe—0@jabber.iitsp.com/longxmpprobe | 2402 | 1882 | 78.35% | 520 |
| xmpprobe—0@jabber.yeahnah.co.nz/longxmpprobe | 2398 | 1629 | 67.93% | 769 |
| xmpprobe—0@lightwitch.org/longxmpprobe | 2418 | 1620 | 67.00% | 798 |
| xmpprobe—1@jabber.at/longxmpprobe | 2430 | 772 | 31.77% | 1658 |
| xmpprobe—0@jabber.at/longxmpprobe | 2432 | 760 | 31.25% | 1672 |
| xmpprobe—1@jappix.com/longxmpprobe | 2485 | 192 | 7.73% | 2293 |
| xmpprobe—0@jappix.com/longxmpprobe | 2448 | 176 | 7.19% | 2272 |
| test1@uptime.p1.im/longxmpprobe | 2448 | 0 | 0.00% | 2448 |
| xmpprobe—1@jabber.smash-net.org/longxmpprobe | 2447 | 0 | 0.00% | 2447 |
| test0@uptime.p1.im/longxmpprobe | 2447 | 0 | 0.00% | 2447 |
| xmpprobe—1@jabber.yeahnah.co.nz/longxmpprobe | 2446 | 0 | 0.00% | 2446 |
| xmpprobe—0@jabber.rueckgr.at/longxmpprobe | 2446 | 0 | 0.00% | 2446 |
| xmpprobe—0@xmppnet.de/longxmpprobe | 2446 | 0 | 0.00% | 2446 |
| xmpprobe—0@jabber.smash-net.org/longxmpprobe | 2446 | 0 | 0.00% | 2446 |

Table 5.2: Messages per Account

| Account | Connect | Disconnect | Nauth |
|---|---|---|---|
| xmppprobe-xmppprobe—1@jabber.smash-net.org | 0 | 28104 | 0 |
| xmppprobe-xmppprobe—0@jabber.smash-net.org | 0 | 28096 | 0 |
| xmppprobe-test1@uptime.p1.im | 0 | 18212 | 0 |
| xmppprobe-test0@uptime.p1.im | 0 | 18204 | 0 |
| xmppprobe-xmppprobe—0@jappix.com | 1 | 95 | 1 |
| xmppprobe-xmppprobe—1@jappix.com | 1 | 79 | 1 |
| xmppprobe-xmppprobe—0@lightwitch.org | 1 | 43 | 2 |
| xmppprobe-xmppprobe—1@chatme.im | 1 | 33 | 0 |
| xmppprobe-xmppprobe—0@chatme.im | 1 | 27 | 0 |
| xmppprobe-xmppprobe—0@jabber.rueckgr.at | 0 | 1 | 4 |
| xmppprobe-xmppprobe—0@xmppnet.de | 0 | 1 | 4 |
| xmppprobe-xmppprobe—1@jabber.yeahnah.co.nz | 0 | 1 | 3 |
| xmppprobe-damiano0@koalatux.ch | 1 | 0 | 0 |
| xmppprobe-damiano1@koalatux.ch | 1 | 0 | 0 |
| xmppprobe-xmppprobe___0@ch3kr.de | 1 | 0 | 0 |
| xmppprobe-xmppprobe___1@ch3kr.de | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@alpha-labs.net | 1 | 0 | 1 |
| xmppprobe-xmppprobe—0@blah.im | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@coderollers.com | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@creep.im | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@draugr.de | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@im.apinc.org | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@is-a-furry.org | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@jabb3r.de | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@jabber-br.org | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@jabber.at | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@jabber.ccc.de | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@jabber.de | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@jabber.iitsp.com | 1 | 0 | 1 |
| xmppprobe-xmppprobe—0@jabber.meta.net.nz | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@jabber.minus273.org | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@jabber.no-sense.net | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@jabber.yeahnah.co.nz | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@jabberafrica.org | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@jabbim.cz | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@jabme.de | 1 | 0 | 0 |
| xmppprobe-xmppprobe—0@tigase.im | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@alpha-labs.net | 1 | 0 | 1 |
| xmppprobe-xmppprobe—1@blah.im | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@coderollers.com | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@creep.im | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@draugr.de | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@im.apinc.org | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@is-a-furry.org | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@jabb3r.de | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@jabber-br.org | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@jabber.at | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@jabber.ccc.de | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@jabber.de | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@jabber.iitsp.com | 1 | 0 | 1 |
| xmppprobe-xmppprobe—1@jabber.meta.net.nz | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@jabber.minus273.org | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@jabber.rueckgr.at | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@jabberafrica.org | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@jabbim.cz | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@jabme.de | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@tigase.im | 1 | 0 | 0 |
| xmppprobe-xmppprobe—1@xmppnet.de | 1 | 0 | 0 |

Table 5.3: Counts for Selected Event Types per Account

# Chapter 6

# Conclusion and Outlook

We have shown that, even within the limited scope of a semester thesis, it is possible to create a tool that provides interesting insights into the XMPP landscape of the Internet. Our tool is straight-forward to use, and we hope that by making it available under a free license we inspire others to use and possibly even extend it.
By running tests using XMPProbe, we were able to confirm the suspicions of some of the authors that the quality of service is indeed lacking for some servers that are usable for free[1]. On the other hand, we have also confirmed that there are servers with excellent service. We hope that results obtained with this measurement tool inspire service operators around the world to provide even better service now that the quality of their work can readily be compared to that of others.
As has been discussed in Section 4.8 our solution XMPProbe is not without flaws. Last but not least, our methods of analysis of the collected data are not yet very sophisticated. We hope that we can extend the functionality of XMPProbe to become an even more powerful and useful tool in the future.

[1] "And what happens on the day that you find out?" — "Well, we all know how much you love to say 'I told you so.'"

# Appendix A

# Using XMPProbe

`XMPProbe` is a tool for monitoring the availability of XMPP (also known as *Jabber*) servers. To do this, it requires at least one working account on each of the servers to be monitored.
To use XMPProbe a configuration file is required. The specifics of the configuration file are documented in Appendix B.
For XMPProbe, the specification of the command line arguments is printed below:

```
usage: xmpprobe [-h] [-c CONFIG] [-b BUDDY] account resource

XMPProbe: Monitoring XMPP Servers.

positional arguments:
account               The account configuration to be used by this instance.
                        An object with the \PYGZdq{}jid\PYGZdq{} property set to this
                            value
                        must exist in the \PYGZdq{}accounts\PYGZdq{} array in the
                            specified
                        configuration file.
resource              The jabber resource to be used by this instance.
                        Becomes a part of the log file name.

optional arguments:
-h, --help            show this help message and exit
-c CONFIG, --config CONFIG
                        The path of the configuration file (default:
                        \PYGZdq{}./xmpprobe.conf\PYGZdq{})
-b BUDDY, --buddy BUDDY
                        The JID to send the ask message to. This argument is
                        only evaluated when the \PYGZdq{}account\PYGZdq{} is of usage
                            type
                        \PYGZdq{}ask\PYGZdq{}.
```

Listing A.1: XMPProbe Usage

For the use in production, only the arguments `--config`, `account` and `resource` are of use.
Invocation example:

```
$ xmpprobe -c config/xmpprobe-config.json "test0@example.com" "test" &
$ xmpprobe -c config/xmpprobe-config.json "test1@example.com" "test" &
$ xmpprobe -c config/xmpprobe-config.json "test2@example.com" "test" &
$ xmpprobe -c config/xmpprobe-config.json "test3@example.com" "test" &
```

Listing A.2: Running XMPProbe

Each Jabber account that should be monitored has to run its own instance of xmpprobe. Each account also needs an appropriate configuration section in the config file.
The typical use case is to start one instance of XMPProbe for every account with usage type `normal` in the configuration file.
The individual instances should be started as close together as possible, but this is not required.

Note that no chat messages might be exchanged for up to (number of accounts) · (period) seconds, that is, XMPProbe can have a long initialization phase.
Return codes of XMPProbe are documented in the class `Ret`. XMPProbe should only quit on its own right at the start. After the configuration is deemed to be valid, it runs indefinitely.

# Appendix B

# XMPProbe Configuration File Format

XMPProbe requires a correct configuration file to run. All the settings that are not likely to change between different runs of XMPProbe can only be set by options in the configuration file, and not by command-line arguments.
The configuration file is written in JSON.

## B.1 Sample Configuration

The following example is a complete and valid configuration file. The semantics of the individual parameters are explained below.

```
{
    "loglevel": "DEBUG",
    "stdoutloglevel": "DEBUG",
    "dataloglevel": "DEBUG",
    "logfile": "./logs/xmpprobe-{account}-{resource}.log",
    "datafile": "./logs/xmpprobe-{account}-{resource}.csv",
    "period": 3,
    "accounts": [
        {
            "jid": "test0@example.com",
            "password": "secret0",
            "usage": "normal"
        },
        {
            "jid": "test1@xmpp.uk.to",
            "password": "secret1",
            "usage": "normal"
        }
    ]
}
```

Listing B.1: Sample Configuration File

## B.2 Configuration Parameters

The configuration is represented by one JSON object. This object must have all of the following keys:

**loglevel (string):** The logging level for the log data written to general *logfile*. This key must have one of the following values: `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`.

**stdoutloglevel (string):** The logging level for the log data written to standard output. This key must have one of the following values: `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`.

**dataloglevel (string):** The logging level for the data written to the CSV files. This key must have one of the following values: `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`.

Note that the only sensible value to currently use is `DEBUG`, this may change in future versions.

**logfile (string):** The schema for the path and filename of the general log file. This string must include the substrings `{account}` (which will be replaced by the Jabber ID of the used account) and `{resource}` (which will be replaced by the resource name used). These two parameters are command-line arguments.

**datafile (string):** The schema for the path and filename of the data file (CSV format). This string must include the substrings `{account}` (which will be replaced by the Jabber ID of the used account) and `{resource}` (which will be replaced by the resource name used). These two parameters are command-line arguments.

**period (integer):** The sampling period (in seconds). This is the period with which new conversations are started.

**accounts (array):** Array of account objects. The array must contain at least one value. To be able to use most features of XMPProbe at least two account object are required.

The structure of account objects is defined in the following definition list.

Account objects have the following keys (all keys are mandatory):

**jid (string):** The Jabber ID of the account. Must be a bare Jabber ID, without a resource.

**password (string):** The password for *jid*.

**usage (string):** The usage type for this Jabber account. Must be one of the following: `answer`, `ask`, `debug_periodicbot`, `debug_timebot`, `echo`, `fixme`, `normal`.

To use the *jid* for monitoring, this value must be set to `normal`. All other accounts that *jid* should attempt to exchange messages with need to be set to `normal` too. At least two accounts with a *usage* of `normal` are required for monitoring to work properly.

A value of `fixme` indicates that this account is not used for any purpose. All other possible values are for testing and debugging purposes only. Refer to the documentation of the classes `AskBot` (`ask`), `AnswerBot` (`answer`), `PeriodicBot` (`debug_periodicbot`), `TimeBot` (`debug_timebot`) or `EchoBot` (`echo`), respectively.

# Appendix C

# Message Format

Class `Message`: Generate and validate messages.

All messages XMPProbe exchanges between accounts are generated by instances of this class.

A message (serialized as JSON) contains the following information:

| Field Name | Type | Interpretation |
|---|---|---|
| magic | string | A constant used to identify a message as a representation of an instance of `Message`. |
| version | integer | Version of the message format. |
| nonce | integer | An integer number picked randomly at the start of an exchange of messages. Incremented (modulo 0x10000000000) for every further message in the same thread. |
| t_sent | ISO 8601 | Timestamp when message was generated. |
| t_rcvd | ISO 8601 | Timestamp when the message to which this message is a reply to was received. |
| report | string | The function (purpose) of this message. |
| comment | string | A human-readable string that is not processed. Intended to message nosy XMPP server administrators that check what apparent spam XMPProbe bots send around. |
| checksum | integer | A checksum of all the above data fields (excluding the checksum field). |

The message exchanges that XMPProbe conducts work in the following way:

One agent sends an *ask* message, and the receiving agent responds with an *answer* message.

**Note** The checksumming is performed on the string representation of python attributes/objects and not the JSON string representation of the attributes/objects.

*Ask* message specification:

| Field Name | Value |
|---|---|
| magic | ‘xmpP’ |
| version | 1 |
| nonce | uniformly distributed random number in the range [0x0, 0xffffffffff] (corresponds to 40 bits of entropy) |
| t_sent | Timestamp when message was generated in UTC including timezone information (Example: ‘2014-02-06T06:34:49.211885+00:00’) |
| t_rcvd | None |
| report | ‘ping’ |
| comment | refer to the value of `COMM` |
| checksum | `zlib.adler32(str(magic) + str(version) + str(nonce) + str(t_sent) + str(t_rcvd) + str(report) + str(comment))` |

*Answer* message specification:

| Field Name | Value |
|---|---|
| magic | ‘xmpP’ |
| version | 1 |
| nonce | `((nonce of *ask* message) + 1) % 0x10000000000` |
| t_sent | Timestamp when message was generated in UTC including timezone information |
| t_rcvd | `t_sent of *ask* message` |
| report | ‘pong’ |
| comment | refer to the value of `COMM` |
| checksum | `zlib.adler32(str(magic) + str(version) + str(nonce) + str(t_sent) + str(t_rcvd) + str(report) + str(comment))` |

Usage example:
Alice wants to start a conversation with Bob, so she creates a message object.

```
>>> a = xmpprobe.Message()
```

Listing C.1: Using the Message Class

When she is ready, she generates the message. Since the message contains a timestamp, it should then actually be sent out as soon as possible.

```
>>> a.ask()
'{"nonce":144974668543,"comment":"XMPP monitoring project. Contact mail:
    xmpprobe@student.ethz.ch for information.","t_rcvd":null,"version":1,"magic":"xmpP
    ","report":"ping","checksum":2065312730,"t_sent":"2014-02-19T18
    :38:55.137526+00:00"}'
```

Listing C.2: Using the Message Class

Now, Alice sends this message to Bob. Bob receives this message as `m`.

```
>>> try:
...     b = xmpprobe.Message(m)
... except ValueError as e:
...     print('fail!')
```

Listing C.3: Using the Message Class

Alternatively, if Bob wants to reuse a previously created Message object, he can instead use:

```
>>> try:
...     b.decode(m)
... except ValueError as e:
```

```
...     print('fail!')
```

Listing C.4: Using the Message Class

If either way of parsing the message does not raise an error, the message has the correct syntax and meaningful values. Bob can now generate an answering message. Again, because this answer contains a timestamp, it should be sent out as soon as possible.

```
>>> b.answer()
'{"nonce":144974668544,"comment":"XMPP monitoring project. Contact mail:
    xmppprobe@student.ethz.ch for information.","t_rcvd":"2014-02-19T18
    :38:55.137526+00:00","version":1,"magic":"xmpP","report":"pong","checksum
    ":2954899668,"t_sent":"2014-02-19T18:39:09.088472+00:00"}'
```

Listing C.5: Using the Message Class

On receiving Bob's answering message as `m`, Alice can now verify it as being a correct *answer* message to the *ask* message she sent Bob earlier.

```
>>> try:
...     a.mark(m)
... except ValueError as e:
...     print('fail!')
```

Listing C.6: Using the Message Class

If no error is raised, the *answer* message was valid.

# Appendix D

# CSV Log Format

This section outlines the details on the format used for the CSV output of XMPProbe. Each record has an event type. The event type determines how many fields a record has, and what their meaning is.
The following definition list shows what each field is used for, for each event type. Some fields are common between various message formats: `time` is the timestamp (floating point number; seconds since Unix time epoch) when the log entry was generated and `event` is the name of the event type.
Note that timestamps other than `time` may be in a different format.

**connect:** time, jid, event

`jid` has connected.

**nauth:** time, jid, event

`jid` could not connect, because the authentication failed.

**disconnect:** time, jid, event

`jid` has disconnected.

**ssubscribe:** time, jid, event, other_jid

`jid` has sent a presence subscription request to `other_jid`.

**rsubscribe:** time, jid, event, other_jid

`jid` has received a presence subscription request from `other_jid`.

**ssubscribed:** time, jid, event, other_jid

`jid` has sent an affirmative answer to a presence subscription request from `other_jid`. Since subscription management happens automatically in parts, this event will never appear in the log file.

**rsubscribed:** time, jid, event, other_jid

`jid` has received an affirmative answer from `other_jid` to a presence subscription request from `jid`.

**skip:** time, jid, event, other_jid

`jid` took too long handling all that needed to be done in one period. One or more periods were skipped, and as a result `other_jid` was not sent an *ask* message.

**sask:** time, jid, event, other_jid, nonce, t_sent

`jid` has sent an *ask* message to `other_jid`. `nonce` and `t_sent` are the respective fields from the sent message.

**sanswer:** time, jid, event, other_jid, nonce, t_sent

> `jid` has sent an *answer* message to `other_jid` in reply to the *ask* message with the respective fields `nonce` and `t_sent`.

**runknown:** time, jid, event, other_jid, msg

> `jid` has received the message `msg` from `other_jid` that does not conform to the specification of *ask* and *answer* messages. This can mean that the message was a corrupt *ask* or *answer* message, or that it was an unrelated message.

**rask:** time, jid, event, other_jid, nonce, t_sent

> `jid` has received a valid *ask* message from `other_jid`. `nonce` and `t_sent` are the respective fields from the received message. The path delay can be calculated from `time` and `t_sent`.

**ranswer:** time, jid, event, other_jid, nonce, t_sent, t_rply

> `jid` has received a valid *answer* message from `other_jid`. `nonce` and `t_sent` are the respective fields from the *ask* message that this *answer* message is a reply to, that is the *nonce* field from the *answer* message is decremented, and the `t_sent` log entry field is actually *t_rcvd* from the message. `t_rply` is *t_sent* from the *answer* message. The path delay can be calculated from `time` and `t_rply`.

**rsurprise:** time, jid, event, other_jid, nonce, t_sent, t_rply

> `jid` has received an *answer* message from `other_jid`. The message appears to be valid, but it can not be checked whether or not it's an *answer* to an *ask* message that was actually sent. The further log entry fields are defined in the manner of the *ranswer* log entry fields.

## D.1 Log File Sample

The following short snipped of a CSV log file illustrates what log entries look like for some of the above event types.

```
1393449830.012724,xmpprobe---0@example.com/longxmpprobe,sask,xmpprobe---0@example.net/
    longxmpprobe,642973785857,2014-02-26 21:23:50.012054+00:00
1393449830.163635,xmpprobe---0@example.com/longxmpprobe,ranswer,xmpprobe---0@example.
    net/longxmpprobe,642973785857,2014-02-26 21:23:50.012054+00:00,2014-02-26
    21:23:50.090813+00:00
1393449840.00329,xmpprobe---0@example.com/longxmpprobe,sask,xmpprobe---0@example.org/
    longxmpprobe,54361358735,2014-02-26 21:24:00.002678+00:00
1393449840.046191,xmpprobe---0@example.com/longxmpprobe,runknown,xmpprobe---0@example.
    org/longxmpprobe,"{""nonce"":54361358735,""comment"":""XMPP monitoring project.
    Contact mail:xmpprobe@student.ethz.ch for information."",""t_rcvd"":null,""version
    "":1,""magic"":""xmpP"",""report"":""ping"",""checksum"":1365126023,""t_sent
    "":""2014-02-26T21:24:00.002678+00:00""}"
1393449850.011563,xmpprobe---0@example.com/longxmpprobe,sask,xmpprobe---0@jabber.
    example.com/longxmpprobe,277493021238,2014-02-26 21:24:10.011015+00:00
1393449860.009387,xmpprobe---0@example.com/longxmpprobe,sask,xmpprobe---0@jabber.
    example.net/longxmpprobe,1040098151630,2014-02-26 21:24:20.008674+00:00
1393449860.071305,xmpprobe---0@example.com/longxmpprobe,ranswer,xmpprobe---0@jabber.
    example.net/longxmpprobe,1040098151630,2014-02-26 21:24:20.008674+00:00,2014-02-26
     21:24:20.044593+00:00
```

Listing D.1: Log Sample

# Appendix E

# Analysis Script

Parse CSV files generated by XMPProbe and derive some statistics.
`analysis.py` is passed a directory of CSV log files and one function. It then computes the requested values by going through all the CSV files found.
Currently, the following functions are supported:

**`files`** Print the filenames of the files that will be parsed for all other functions.

**`records`** Print the number of event records per type and the total.

**`interval`** Print timestamps of first and last event (of any type) recorded in the data set, and calculate the time difference between the two.

Output format:

- `Lowest timestamp`, human readable timestamp, Unix time with microsecond resolution.
- `Highest timestamp`: human readable timestamp, Unix time with microsecond resolution.
- `Runtime`: human readable time interval with microsecond resolution (may contain rounding errors due to conversion to float), time interval in seconds with microsecond resolution

**`delay`** Calculate the minimum, 5th percentile, 50th percentile, 95th percentile and maximum of the path delay for all instant messages sent recorded in the logs as events of the types `rask`, `ranswer` and `rsurprise`. These three event types cover all messages that are valid `ask` or `answer` messages. Other messages are ignored for this metric. Internally all calculations are done in fixed point arithmetic as far as possible, but the library used for calculating the percentile numbers requires the use of Python's `float` type. So, unlike the minimum and maximum values the percentile values are subject to rounding errors.

Output format: The following list gives the captions and type of the values printed in the output.

- `minimum`, fixed point
- `5%%ile`, float
- `50%%ile`, float
- `95%%ile`, float
- `maximum`, fixed point

**msg_quota** Calculate per-Jabber-account percentage of correctly answered messages, that is, if Alice has sent an `ask` message to Bob (Alice records a `sask` event to the logs), the message will be deemed answered correctly when Alice receives an appropriate `answer` message from Bob (Alice records a `ranswer` or `rsurprise` event to the logs). Note that if Bob has a bad success rate this does not imply that Bob is at fault, as Alice's XMPP server may be dropping or mangling the `ask` messages.

Output format: The calculated values are listed as one CSV record per account in the input data. Each record contains the following fields:

**JID (string)** The Jabber ID.

**asks_received (integer)** The number of `ask` messages sent to this JID.

**answers_sent (integer)** The number of correct answer messages received from this JID.

**quota (float)** The ratio of `answers_sent` over `asks_received`. A value of `-1` indicates NaN.

**remaining_asks (integer)** The number of `ask` messages sent to this JID that have no recorded correct answers. A value of `-1` indicates NaN.

**remaining_answers (integer)** The number of `answer` messages sent from this JID that have no `ask` message that is answered. A value other than `-1` indicates either a bug in XMPProbe or `analysis.py` or manipulated data.

Usage example:

```
$ ./analysis.py ../data/run-1 files
58 files will be parsed.
58 files found with the following filenames:
[   '../data/run-1/xmpprobe-xmpprobe---0@im.apinc.org-longxmpprobe.csv',
    '../data/run-1/xmpprobe-xmpprobe---1@xmppnet.de-longxmpprobe.csv',
    '../data/run-1/xmpprobe-xmpprobe---0@creep.im-longxmpprobe.csv']

$ ./analysis.py ../data/run-1 records
58 files will be parsed.
Processing files ...
..........................................................
Done parsing data. Execution time: 0:00:03.614930
Total number of records: 582198.
Number of records per event type:
{   'connect': 51,
    'disconnect': 92896,
    'ssubscribe': 1781,
    'ssubscribed': 0}

$ ./analysis.py ../data/run-1 interval
58 files will be parsed.
Processing files ...
..........................................................
Done parsing data. Execution time: 0:00:14.437214
Lowest timestamp : 2014-02-26 06:08:10.205568 (1393391290.205568).
Highest timestamp: 2014-02-26 23:13:31.225412 (1393452811.225412).
Runtime          : 17:05:21.019844 (61521.019844).

$ ./analysis.py ../data/run-1 delay
58 files will be parsed.
Processing files ...
..........................................................
Done parsing data. Execution time: 0:01:06.116309
Interesting Delay Values:
                minimum: 0.012371s
                5%ile:   0.026598s
                50%ile:  0.0760875s
                95%ile:  0.318321s
                maximum: 6383.632378s
Done. Execution time: 0:00:02.431927

$ ./analysis ../data/run-1 msg_quota
58 files will be parsed.
Processing files ...
..........................................................
Done parsing data. Execution time: 0:00:06.182874
{JID}, {asks_received}, {answers_sent}, {quota}, {remaining_asks}, {remaining_answers}
xmpprobe---0@lightwitch.org/longxmpprobe,2418,1620,0.6699751861042184,798,-1
xmpprobe___1@ch3kr.de/longxmpprobe,2428,1958,0.8064250411861614,470,-1
xmpprobe---0@coderollers.com/longxmpprobe,2420,2309,0.9541322314049587,111,-1
xmpprobe---0@chatme.im/longxmpprobe,2433,1942,0.7981915330867242,491,-1
Done. Execution time: 0:00:00.001104
```

Listing E.1: Analysis Usage

Note that the output in this usage example has been truncated in parts and is therefore not consistent.

# Appendix F

# Message Quotas per Account

```
JID,asks_received,answers_sent,quota,remaining_asks,remaining_answers
xmpprobe---0@jabber.ccc.de/longxmpprobe,2407,2404,99.88%,3,-1
damiano1@koalatux.ch/longxmpprobe,2431,2427,99.84%,4,-1
xmpprobe---1@jabber.meta.net.nz/longxmpprobe,2430,2426,99.84%,4,-1
xmpprobe---1@jabber.ccc.de/longxmpprobe,2424,2420,99.83%,4,-1
xmpprobe---0@im.apinc.org/longxmpprobe,2418,2414,99.83%,4,-1
xmpprobe---1@im.apinc.org/longxmpprobe,2416,2412,99.83%,4,-1
xmpprobe---0@jabber.meta.net.nz/longxmpprobe,2403,2399,99.83%,4,-1
xmpprobe---1@jabber.rueckgr.at/longxmpprobe,2433,2428,99.79%,5,-1
damiano0@koalatux.ch/longxmpprobe,2432,2427,99.79%,5,-1
xmpprobe---1@jabb3r.de/longxmpprobe,2419,2414,99.79%,5,-1
xmpprobe---0@jabb3r.de/longxmpprobe,2413,2408,99.79%,5,-1
xmpprobe---1@jabber.minus273.org/longxmpprobe,2433,2427,99.75%,6,-1
xmpprobe---0@creep.im/longxmpprobe,2421,2415,99.75%,6,-1
xmpprobe---1@creep.im/longxmpprobe,2412,2406,99.75%,6,-1
xmpprobe---0@jabber.minus273.org/longxmpprobe,2402,2396,99.75%,6,-1
xmpprobe---0@jabber.no-sense.net/longxmpprobe,2402,2396,99.75%,6,-1
xmpprobe---0@jabberafrica.org/longxmpprobe,2403,2351,97.84%,52,-1
xmpprobe---0@jabme.de/longxmpprobe,2401,2349,97.83%,52,-1
xmpprobe---1@jabberafrica.org/longxmpprobe,2436,2383,97.82%,53,-1
xmpprobe---1@jabme.de/longxmpprobe,2437,2383,97.78%,54,-1
xmpprobe---0@draugr.de/longxmpprobe,2418,2310,95.53%,108,-1
xmpprobe---1@draugr.de/longxmpprobe,2412,2303,95.48%,109,-1
xmpprobe---1@coderollers.com/longxmpprobe,2410,2300,95.44%,110,-1
xmpprobe---1@jabber.de/longxmpprobe,2428,2317,95.43%,111,-1
xmpprobe---0@jabbim.cz/longxmpprobe,2400,2290,95.42%,110,-1
xmpprobe---0@coderollers.com/longxmpprobe,2420,2309,95.41%,111,-1
xmpprobe---1@jabbim.cz/longxmpprobe,2437,2325,95.40%,112,-1
xmpprobe---0@jabber.de/longxmpprobe,2407,2296,95.39%,111,-1
xmpprobe---1@xmppnet.de/longxmpprobe,2435,2253,92.53%,182,-1
xmpprobe---0@jabber-br.org/longxmpprobe,2410,2198,91.20%,212,-1
xmpprobe---1@jabber-br.org/longxmpprobe,2419,2206,91.19%,213,-1
xmpprobe---1@is-a-furry.org/longxmpprobe,2415,2199,91.06%,216,-1
xmpprobe---0@is-a-furry.org/longxmpprobe,2415,2199,91.06%,216,-1
xmpprobe---1@tigase.im/longxmpprobe,2431,2180,89.68%,251,-1
xmpprobe---0@tigase.im/longxmpprobe,2402,2150,89.51%,252,-1
xmpprobe---0@alpha-labs.net/longxmpprobe,2425,2109,86.97%,316,-1
xmpprobe---1@alpha-labs.net/longxmpprobe,2405,2089,86.86%,316,-1
xmpprobe___0@ch3kr.de/longxmpprobe,2429,1959,80.65%,470,-1
xmpprobe___1@ch3kr.de/longxmpprobe,2428,1958,80.64%,470,-1
xmpprobe---0@chatme.im/longxmpprobe,2433,1942,79.82%,491,-1
xmpprobe---1@chatme.im/longxmpprobe,2418,1917,79.28%,501,-1
xmpprobe---0@blah.im/longxmpprobe,2422,1904,78.61%,518,-1
xmpprobe---1@blah.im/longxmpprobe,2405,1889,78.54%,516,-1
xmpprobe---1@jabber.iitsp.com/longxmpprobe,2417,1897,78.49%,520,-1
xmpprobe---0@jabber.iitsp.com/longxmpprobe,2402,1882,78.35%,520,-1
xmpprobe---0@jabber.yeahnah.co.nz/longxmpprobe,2398,1629,67.93%,769,-1
xmpprobe---0@lightwitch.org/longxmpprobe,2418,1620,67.00%,798,-1
```

```
xmpprobe---1@jabber.at/longxmpprobe,2430,772,31.77%,1658,-1
xmpprobe---0@jabber.at/longxmpprobe,2432,760,31.25%,1672,-1
xmpprobe---1@jappix.com/longxmpprobe,2485,192,7.73%,2293,-1
xmpprobe---0@jappix.com/longxmpprobe,2448,176,7.19%,2272,-1
test1@uptime.p1.im/longxmpprobe,2448,0,0.00%,2448,-1
xmpprobe---1@jabber.smash-net.org/longxmpprobe,2447,0,0.00%,2447,-1
test0@uptime.p1.im/longxmpprobe,2447,0,0.00%,2447,-1
xmpprobe---1@jabber.yeahnah.co.nz/longxmpprobe,2446,0,0.00%,2446,-1
xmpprobe---0@jabber.rueckgr.at/longxmpprobe,2446,0,0.00%,2446,-1
xmpprobe---0@xmppnet.de/longxmpprobe,2446,0,0.00%,2446,-1
xmpprobe---0@jabber.smash-net.org/longxmpprobe,2446,0,0.00%,2446,-1
```

Listing F.1: Complete List

# Bibliography

[1] Thijs Alkemade, *XMPPoke — testing the encryption strength of XMPP servers*, https://bitbucket.org/xnyhps/xmppoke, 2013, [Online; accessed 2014-03-04].

[2] ______, *IM observatory; testing the security of the jabber/XMPP network*, https://xmpp.net/about.php, 2014, [Online; accessed 2014-03-04].

[3] Raoul Bourquin, *SSD*, Unpublished images created in discussion of the SSD, 2014.

[4] D0ktorz, *XMPP network*, http://de.wikibooks.org/wiki/Datei:Jabber-Netzwerk.svg, 2006, [Online; accessed 2014-03-04].

[5] Peter Saint-Andre, *A public statement regarding ubiquitous encryption on the XMPP network*, https://github.com/stpeter/manifesto/blob/master/manifesto.txt, 2013, [Online; accessed 2014-03-04].

[6] Peter Saint-Andre, Kevin Smith, and Remko Troncon, *XMPP: The definitive guide; buiding real-time applications with Jabber technologies*, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2009, *The* reference book on XMPP.