Programming Jabber

Table of Contents

List of Tables

List of Figures

List of Examples

Next
Preface

Preface. Preface

Table of Contents

Chapter 1. Preface

Table of Contents

What Is Jabber?

Well, that is certainly a good question with which to start this book. Depending on who you ask, the following answers may be forthcoming:

- Jabber is a technology
- Jabber is a protocol (or set of protocols)
- Jabber is an XML-based Instant Messaging (IM) system
- Jabber is an *implementation* of the set of protocols
- Jabber is an idea whose time has come.

In fact, all these answers are right. Jabber is a set of protocols expressed within XML that provide people and applications with the ability to converse. Sure, TCP sockets, STDIN/STDOUT, infrared, voice input and teletype mechanisms all allow people and applications to converse; the difference is that Jabber provides a structured, extensible framework for exchanging all kinds of information.

This is all rather abstract. What do we mean by "extensible framework for exchanging information?" Taking it one word at a time:

Extensible

> Jabber's substrate is XML. XML is inherently extensible in the sense that tags can be added in a hierarchical sequence. Namespaces in XML allow us to keep track of the meaning and

organization of these tags.

Framework

You can put together a system for exchanging information using many different tools. The point about the "framework" is that the information exchanged and the entities that are exchanging it are contextualized, bringing meaning and structure to the interactions.

Exchanging

Conversation is two-way, and it takes many forms: question and answer, notification, compartmentalized discussion, and simple chat. Jabber supports all these different types of conversation, and more.

Information

One doesn't really say that *information* is exchanged in a conversation, but when you bring applications into the mix of conversing entities, it may well be the case. And it's not just information exchanged in the form of conversations, but also information about the entities themselves that flows across this context framework.

It goes without saying:

- Jabber is an IM system

IM was Jabber's original raison d'etre. Many deployments of Jabber software are to provide IM services, but Jabber is more than IM. Certainly more than the phrase "Instant Messaging" represents. In this book you'll find out why this is so, and how you can deploy solutions with Jabber that are more than mere chat. But most importantly, Jabber is fun!

Like chess, which has a small set of rules but countless game possibilities, the technologies employed in Jabber and the Jabber protocol itself are straighforward. The possibilities are almost limitless. Furthermore, because of a fundamental design feature [1] (you might call it a "philosophical" angle), implementing Jabber-based solutions to your problems can be fun—really!

Notes

[1] That the complexity of a Jabber implementation should always remain in the *server*, leaving the clients simple and clean

The History Of Jabber

Jeremie Miller started the Jabber project in early 1998 and it was announced to the public in January 1999. To understand why Jabber came about, and in the form it took, let's look briefly at what existed in the IM world before Jabber.

The pre-Jabber history

IM existed as a concept and a handful of systems from companies such as Mirabilis, AOL, Microsoft and Yahoo!. These systems (ICQ from Mirabilis, AIM from AOL, MSN from Microsoft, and Yahoo!IM from Yahoo!) allowed their users to chat to one another and avail themselves of IM-related services. However, an AIM user couldn't chat with an ICQ user, and MSN users couldn't interact with Yahoo!IM users. Each system was effectively closed to the outside world.

Furthermore, the protocols that these systems used were also closed —proprietary—which meant it was difficult to find clients for these IM systems other than the ones supplied by the IM system owner.

Finally, the systems themselves were monolithic: multiple clients but a single server (or server farm). Although the companies were able to invest time and money into the problem, the fact remained that a monolithic architecture presented a scaling problem. Perhaps more relevant than that, companies who wanted to use IM services internally had to accept the fact that the conversations would be carried through the systems of a third party—namely the owners of these public IM systems. This was no more desirable than for a company to run their internal email using a public email service such as Hotmail.

Of course, these systems did have their advantages. The clients were accomplished and easy to learn and use; and as long as your correspondents were using the same IM system, and you didn't mind your messages being carried by another organization (for private individuals these wouldn't be unique circumstances; again, we are led to the email services parallel) then you could leave the system management to someone else and get on with chatting.

Scratching an itch

Having all your contacts use the same IM system is all well and good in theory, but in practice is rarely the case. (If, like me, you have few friends, then this is not so much of a problem). Jeremie Miller had correspondents in different IM systems, and consequently had to have different IM clients running on his desktop to keep up with them all. Many great software projects stem from a personal "itch" that someone wanted to scratch. This was the primary itch that Jeremie had. A single client for all IM interaction: *panacea*.

Of course, one obvious solution would be to build a single client that supported all of the IM system protocols, but this approach had two drawbacks:

- The proprietary nature of the protocols made it harder to implement the support required and would make the client overly complicated.
- Every time the protocol, which wasn't under his control, changed, or a new one came along, the client would have to be modified—a task not practical for a large user base.

On top of that, GUI programming isn't everyone's cup of tea, and Jeremie preferred a solution that allowed him to concentrate on the underlying problems at hand and let others build the GUIs.

And then came Jabber

So Jeremie resolved to create a solution that had the following characteristics:

- It would have its own internal protocol, based upon XML.
- This protocol should be:
    - Simple to understand and implement.
    - Easy to extend.
    - Open.
- The complexity of bridging the disparate proprietary IM protocols would remain at the server, each bridge being a plug-in module.
- All the clients would only have to implement the single, simple internal protocol; everything else would be implemented at the server.

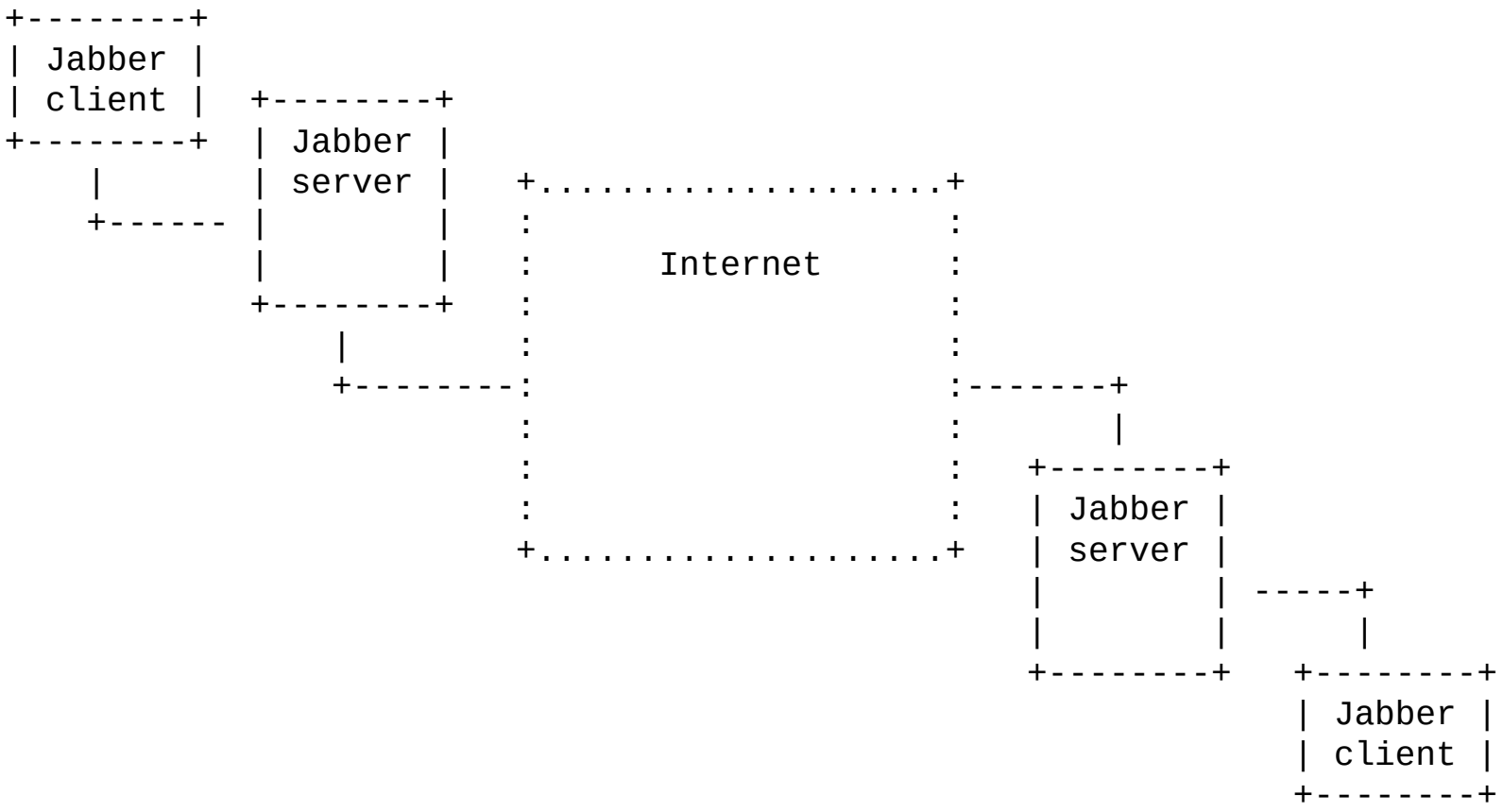
He called this solution "Jabber."

At the same time, perhaps because he didn't consider himself a large organization with the resources to run a centralized server service, this architecture feature was fundamental:

- Anyone could implement a server of their own.

In the same way that email is not a centralized service— each mail user has an address that corresponds to where their mailbox is held—the system that Jeremie envisioned was a decentralized one. This meant that individuals, companies and public organizations could run their own servers—especially relevant for internal-only, IM-style corporate communication. Just as email servers exchange mail using Simple Mail Transfer Protocol (SMTP), so the Jabber servers would connect and exchange IM traffic when necessary.

Figure 1-1. The Distributed Architecture of Jabber



Being *open* meant that Jabber could benefit from the help of anyone who wished to lend a hand, and administrators were empowered to be able to find and fix problems themselves if they so wished.

Being *XML-based*, as opposed to some other binary format for example, meant that the protocol streams were easy for humans to read, extensible, and readily integrated (there is a great range of XML parsing and construction tools already available).

Being *distributed* meant that the Jabber system would belong to the people, and that some of the scalability problems would be avoided. [1]

All of these features made for a good IM system design. But why stop at IM? Consider the client as an implementation of a simple protocol to exchange messages and presence information in XML structures, and use plug-in services at the server, and what do you have?

A language and platform agnostic XML routing framework.

Good grief, what a mouthful! This is why my response to "What is Jabber?" is usually just:

A really great *technology!*

Notes

[1] There remain some scalability issues of course. Client-server communication that is TCP socket based suffers from limitations of this technology. There are however initiatives to overcome these limitations with multiplexing techniques such as `jpolld` and `dpsm`.

Prev Preface | Home | Up | Next IM System Features

IM System Features

This book assumes you have a basic knowledge of features commonly found in IM systems. In case you haven't, here's a brief rundown of features relevant to what we'll be covering:

Presence

In many cases, there's not much point in sending a quick message to someone if they're not there. *Presence* is a term used in IM to describe the technique of exchanging information, in a controlled manner, about availability (or unavailability).

The idea is that when you connect to your IM server, your client sends an "I'm here" message which is relayed to your correspondents. It does the opposite when you disconnect. During the time you're connected you can vary the information about your availability to reflect your immediate situation ("just popped out for coffee," "working on my resume—don't disturb me!").

Buddy List/Roster

Both terms (the former comes from the original IM systems, the latter from Jabber) refer to a list of correspondents with whom you regularly communicate and receive presence information from. Depending on the IM system, the list may be stored on the client or on the server. Storing the list on the client has the (tenuous) advantage of being accessible when you're not connected to the server. Storing the list on the server means that you have a consistent roster content regardless of the client or workstation you happen to use—the list travels with you.

Jabber stores this list—the roster—at the server.

Push and Pull

When you connect to an IM system, there may be information the client needs to retrieve—*pull*—from the server (the roster, for instance). This is under direct control of your client as it decides when to make the retrieval. During the course of the connection, you'll receive messages from your correspondents. You don't request these messages by making a *retrieve call* to the server, as you would with the Post Office Protocol (POP) or Internet Message Access Protocol (IMAP) protocols to retrieve email messages from the mail server; they're *push*ed to you as they occur.

In other words, you could say that the client must implement an event-based system, to listen out for and subsequently handle the incoming information, by displaying a popup window containing

the chat message, for example.

The push/pull system lends itself well to traffic other than IM traffic.

Client-Server

It almost goes without saying that, like other IM systems, Jabber has a client-server architecture. Clients, and client libraries that implement the Jabber protocol, such as Net::Jabber, JabberPy, and Jabberbeans, are available for many languages (here, for Perl, Python, and Java, respectively).

With Jabber, it is especially relevant to stress that the "weight balance" in complexity terms, between the client and the server, comes down heavily on the server side. Not only does this mean that the complexity remains where it should be—on the server—but also makes the task of writing clients easier and the resulting software lighter.

Multiple vs Single Server

We've already mentioned that the Jabber architecture does not dictate a single, centralized server. Not only does this mean that organizations can implement their own private system, but also that developers are free to install their own server and develop new plug-in services in addition to the IM bridges already available.

What's Inside

This book is not particularly about IM per se. Nor is it about the bridges to other IM systems. It's about the essence, the ideas, the potential behind that concept and reality called *Jabber*.

You will learn about the Jabber protocol, and how to use Jabber's technology not only to implement IM based solutions but also solutions that don't involve inane chat. You'll learn how to install and configure your own Jabber server. You will discover more about the features of Jabber that give it its propensity for being an ideal messaging glue for many communication solutions; all of Jabber's technology features—the building blocks and the protocol itself—are explained; and you'll get to know how Jabber can be implemented in a variety of situations—some involving IM, others not—through a series of application and problem scenarios with fully working code examples, or *recipes*, in Perl, Python and Java.

Here's a brief overview of what's in the book.

Part I, *Getting Started with Jabber*

> The first part of this book provides you with an introduction to Jabber; you'll learn about its features, why it's more than an IM system, and how to install and configure a Jabber server of your own.

Chapter 1, *A Taste of Things to Come*

> We begin with an imaginary conversation with human and application participants, which shows how Jabber provides the supporting messaging "plasma". A short script shows how simple it is to make use of Jabber's power.

Chapter 2, *Inside Jabber*

> We take a look at some of the features—the nature—of Jabber, to understand why Jabber is more than just an IM system. The features introduced in this Chapter will be revisited as core building material for our recipes in Part II.

Chapter 3, *Intalling the Jabber Server*

> Here you'll learn how to retrieve and install the Jabber server, and perform minimal configuration, enough to be able to fire it up and use it as a basis for the recipes in Part II. Some troubleshooting and monitoring tips are also included.

Chapter 4, *Server Architecture and Configuration*

Once we have our Jabber server installed and running, we take a closer look at how the server has been designed. We focus on the server makeup, and the different ways it can be deployed. A detailed tour of the standard configuration is also in this Chapter.

Part II, *Putting Jabber's Concepts to Work*

The second part of this book provides a series of recipes—practical solutions to everyday problems—deployed in Jabber. The recipes use various Jabber features as a way of illustration.

Chapter 5, *Jabber Technology Basics*

We take a detailed look at what Jabber looks like under the hood. We examine JIDs, resource and priority, XML streams, and namespaces, as well as the basic Jabber building blocks (`<message/>`, `<iq/>` and `<presence/>`).

Chapter 6, *Identification, Authorization and Registration*

This Chapter looks at the steps needed to create and authenticate with a user, including the different types of authentication. We will also build a utility to create users.

Chapter 7, *Simple Messaging and Presence Applications*

This Chapter looks at some simple examples of Jabber deployment using basic features of message and presence, including presence subscription.

Chapter 8

chapter 8

Chapter 9

chapter 9

The Software

The recipes in this book come in varying flavors, some in Perl, some in Python, and some in Java. These examples—to a greater or lesser degree—make use of prewritten libraries that at least provide the basic services needed to connect to a Jabber server and exchange data with it. Here's a summary of the versions of the languages used in this book, along with those libraries that are used, what features they offer, and where they're available. In addition, references to all of these libraries can be found on the Jabber development website http://dev.jabber.org. The installation instructions for the libraries can be found in the library packages themselves.

Java

The Java recipes in this book are written in Java 2 (J2SE - the Java 2 Standard Edition), specifically with the 1.3.1 version of the Java Development Kit (JDK).

JabberBeans is the name of the Java library for Jabber. It offers comprehensive coverage of the features needed to write programs that interact with Jabber servers: connection, authentication, and the management of Jabber elements passed between your program and the Jabber server.

The *JabberBeans* library can be obtained from http://jabberbeans.org.

Perl

The recipes have been built and tested with Perl 5.6.0, although earlier and later versions of release 5 will probably work just fine.

There are two libraries available for programming Jabber solutions in Perl. They both come in the form of installable modules, and are both of the object-oriented persuasion.

`Net::Jabber`

> This module is available on the Comprehensive Perl Archive Network (CPAN). `Net::Jabber` [1] provides basic functionality for connecting to and interacting with a Jabber server, in addition to a host of higher level features for manipulating all of the Jabber elements and making use of standard and custom namespaces.

It relies upon a companion module `XML::Stream`, also available on CPAN, that provides the underlying mechanisms for creating connections to a Jabber server, as well as sending, receiving, and interpreting (parsing) the fragments of conversation between your script and that Jabber server.

`Jabber::Connection`

The `Jabber::Connection` module is available at http://www.pipetree.com/jabber/ and provides the same basic features as `Net::Jabber` does, albeit in a more 'RISC' (Reduced Instruction Set Computing) way. While it provides similar functionality for connecting to and exchanging data with a Jabber server, it offers, via a companion module called `Jabber::NodeFactory`, a lower-level API—similar to that in the Jabber server itself—for constructing and manipulating the Jabber elements. There are no high-level features; instead, you build your own using the building blocks that the module provides.

Python

The Python examples have been written with Python 2.0.

JabberPy is the name of the Python Jabber library project that supplies the libraries used in the Python recipes in this book. As with Perl's `Net::Jabber` library set, *JabberPy* provides its feature set from two separate libraries—`Jabber`, which provides connectivity, authorization and callback functions like `Net::Jabber` and `Jabber::Connection`, and `XMLStream` which provides the basic connectivity and parsing functions like `Net::Jabber`'s companion `XML::Stream`.

The *JabberPy* libraries are available from its project site, at http://sourceforge.net/projects/jabberpy.

Notes

[1] at http://www.cpan.org

Conventions Used in This Book

The following typographical conventions are used in this book:

Bold

Used for commands, programs, and options. All terms shown in bold are typed literally.

Italic

Used to show arguments and variables that should be replaced with user-supplied values. Italic is also used to indicate new terms and URLs, filenames and file extensions, directories, and to highlight comments in examples.

`Constant Width`

Used to show the contents of files or the output from commands.

`Constant Width Bold`

Used in examples and tables to show commands or other text that should be typed literally by the user.

`Constant Width Italic`

Used in examples and tables to show text that should be replaced with user-supplied values.

Note: These signify a tip, suggestion, or general note.

Warning
These indicate a warning or caution.

How to Contact Us

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions to future editions, by writing:

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472
(800) 998-9938 (in the U.S. or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

You can also send us messages electronically. To be put on the mailing list or to request a catalog, send email to:

<info@oreilly.com>

To ask technical questions or comment on the book, send email to:

<bookquestions@oreilly.com>

We have a web site for the book, were we'll list examples, errata, and any plans for future editions. The site also includes a link to a forum where you can discuss the book with the author and other readers. You can access this site at: http://www.oreilly.com/catalog/progjab

For more information about this book and others, see the O'Reilly web site: http://www.oreilly.com

Acknowledgments

I. Getting Started with Jabber

Table of Contents

Chapter 1. A Taste of Things to Come

Table of Contents

This book is about Jabber. The technology, the protocols, the ideas, and the philosophy. Jabber is about connecting things - users, applications, etc. - in an all-pervasive message plasma that flows between clients and servers alike, carrying content, carrying structure, carrying conversations.

The spirit of Jabber lies in its openness, its extensibility, and its lean but generic nature. That it finds itself in the midst of all that technology that will provide the backdrop to the dawn of the next generation Internet is no accident. Web services, peer-to-peer, XML encapsulation, presence, distributed computing—these are all phrases that describe key intiatives and developments that lie at the core of the coming transition, and Jabber can and does play a central role.

In this book, we'll learn about how Jabber works, what makes it tick, and how to bend and shape it into solutions that join applications and users together. Part I is a guide to Jabber's technology and building blocks. Part II is a series of scenarios and scripts we call 'recipes', to show you how to deploy Jabber in all manner of situations.

To help us get in the mood, there follows an imaginary Jabber-based conversation between a couple of friends, Jim and John, and two 'assistant' applications at their respective places of work.

Imaginary Conversation

Jim and John work at two different companies, and are both interested in stocks and shares, using Jabber-based services to check and monitor prices, buy stock and manage their portfolios. There's a workflow assistant at John's company that can do things like monitor incoming email and coordinate work items between colleagues.

Jim sends a quick chat message to John:

```
<message type='chat' from='jim@company-a.com/home'
    to='john@company-b.com'>
  <thread>01</thread>
  <body>Hey John, you seen the latest story on Megacorp earnings?</body>
</message>
```

John responds:

```
<message type='chat' to='jim@company-a.com/home'
    from='john@company-b.com/Desk'>
  <thread>01</thread>
  <body>No, where is it?</body>
</message>
```

Jim sends John the URL:

```
<message type='chat' from='jim@company-a.com/home'
    to='john@company-b.com/Desk'>
  <thread>01</thread>
  <body>Here's the link</body>
  <x xmlns='jabber:x:oob'>
    <url>http://www.megacorp.co.uk/earnings3q.html</url>
    <desc>Third Quarter Earnings for Megacorp</desc>
  </x>
</message>
```

John receives an alert about the price of ACME Holdings (ACMH) falling below a threshold he previously set:

```
<message to='john@company-b.com' from='alert@stocks.company-b.com'>
  <subject>ACMH Fallen below 250p</subject>
  <body>ACME Holdings stock price 248p as at 10:20am today</body>
</message>
```

He checks the price of Megacorp stock (MEGC) by sending an empty message to the stock and shares assistant application:

```
<message type='chat' to='MEGC@stocks.company-b.com'
    from='john@company-b.com/Desk>
  <thread>T20</thread>
</message>
```

In reply, the stocks and shares assistant application sends the required information:

```
<message type='chat' from='MEGC@stocks.company-b.com'
    to='john@company-b.com/Desk'>
  <thread>T20</thread>
  <subject>MEGC Current Price</subject>
  <body>
    Megacorp stock price 1287p at 10:25am today
  </body>
</message>
```

John likes the price and decides to buy 100 more shares:

```
<message type='chat' to='MEGC@stocks.company-b.com'
    from='john@company-b.com/Desk'>
  <thread>T20</thread>
  <body>buy 100 at 1287 now</body>
</message>
```

He sends a message to Jim telling him of his new investment. Jim checks his own portfolio:

```
<iq type='get' to='portfolio.company-a.com'
    id='port_01'
    from='jim@company-a.com/home'>
  <query xmlns='jabber:iq:browse'/>
</iq>
```

He sees that he already holds 1200 Megacorp shares, and decides against buying any more:

```
<iq type='result' from='portfolio.company-a.com'
```

```
    id='port_01'
    to='jim@company-a.com/home'>
  <portfolio xmlns='jabber:iq:browse' type='personal'
       jid='jim@portfolio.company-a.com'>
    <stock type='standard' name='Megacorp'
          jid='MEGC@portfolio.company-a.com' holding='1200'/>
    <stock type='standard' name='ACME Holdings'
          jid='ACMH@portfolio.company-a.com' holding='500'/>
  </portfolio>
</iq>
```

Workassist, the workflow assistant application, sends John a notification of an important email:

```
<message from='workassist@company-b.com'>
  <subject>New mail from: Alastair B</subject>
  <body>
      You have a new mail waiting; details are as follows:
      Subject: Incident last week
      From:    Alastair B
  </body>
</message>
```

He also receives an invite to a meeting:

```
<message from='joanne@company-b.com/laptop'>
  <body>
    Hey John, you're supposed to be helping us decide where to hold
    this year's Christmas party!
  </body>
  <x xmlns='jabber:x:conference' jid='room2@meeting.company-b.com'/>
</message>
```

John decides to leave the mail until later and join the rest of his colleagues in the meeting room.

There's a great deal we see in this imaginary but not unlikely conversation.

- There are human and application participants involved in this conversation; Jabber makes no distinction, nor holds any prejudice against either participant type.
- XML is used effectively to segregate the conversational chunks (Jim's opening gambit, John's response, the stock price alert, and so on) and to structure and identify data within those chunks (the URL for the earnings story, the portfolio information, the conference room invitation).
- Conversational strands are kept in context by use of a `<thread/>` tag, so that Jim's and John's clients have a chance to present what's said in an appropriate way.

- Each conversational chunk is either a `<message/>` or a `<iq/>` tag - two of the three Jabber building blocks (`<message/>`, `<iq/>` and `<presence/>`) on which everything is based.
- The conversation takes place across two Jabber servers - the one at `company-a.com` and the other at `company-b.com`.
- The conversation flow is asynchronous, in the sense that out of nowhere, a message alerting John to a stock price fall below threshold appears seemingly in the middle of John's conversation with Jim, as does the message from the 'workassist' agent.
- All Jabber entities - human or otherwise - are identified by Jabber IDs such as John's (`john@company-a.com`) or the stock alert mechanism (`alert@stocks.company-b.com`).
- `company-b.com`'s stock system is multi-faceted. As well as being able to send (and receive settings for) threshold alerts through the `alert@stocks.company-b.com` address, it can also interact using different identities to reflect the context of the stock being discussed, for example `MEGC@stocks.company-b.com` represents the Megacorp (MEGC) stock.

  This stock system is in fact a Jabber component, a single element which takes on each of the stock guises in the conversation.

Jabber is a set of protocols, a technology, a system that is extremely capable of providing IM facilities, including bridging services to other IM systems.

But it is capable of much more than that. Paring down what Jabber is to its bare essentials, what do we have? An extensible client-server architecture in which XML can be exchanged, be *routed*, between clients and services that plug in as components to the Jabber server. The original and core set of components provide the IM features (and supporting services) that were briefly described in the Preface; the current 'off-the-shelf' Jabber clients are generally orientated towards IM.

But of course, IM is just one of countless domains to which XML-based messaging technology such as Jabber's can be applied.

As we'll learn, the XML structures that make up the Jabber protocol fall into three categories, each represented by a uniquely named top-level tag (also referred to as an *element*). But these tags mustn't necessarily carry human-generated IM message content—as long as the resulting XML is well-formed, anything goes. [1] Furthermore, the Jabber protocol design makes use of an XML feature that allows total flexibility of extention: *Namespaces.*

Bearing this in mind, it's clear that Jabber can be deployed to provide solutions outside the IM space as well as within it.

Notes

[1] To prevent flooding of the server with large amounts of data, there are mechanisms in place to 'throttle' heavy connections. These mechanisms can be configured in the server configuration, described in the section called *io Section* in Chapter 4.

A Simple Script

Before moving on, let's have a look how simple it is to interact with Jabber. Example 1-1 shows a simple Perl script that connects to a Jabber server, authenticates, checks who's online, and sends those people a reminder message. It uses the Net::Jabber library, which provides a high-level API to many Jabber-related functions such as handling the connection to the server (this is via another library that Net::Jabber uses—XML::Stream), authentication, events, and all the mechanisms to parse and create Jabber traffic.

Example 1-1. A simple Jabber script

```
#!/usr/bin/perl

use Net::Jabber qw(Client);
use strict;

# List of addressees for our reminder
our @addressees;

# What we want to send
my  $reminder = $ARGV[0] or die "No reminder!";

# Connect to our Jabber server
my $c= Net::Jabber::Client->new();
$c->Connect('hostname' => 'yak',
            'port'     => 5222);

# Authenticate
$c->AuthSend('username' => 'reminder',
             'password' => 'secret',
             'resource' => 'reminder');

# Set handler to deal with presence packets
# that might (will) be pushed to us (we're
# not interested in any other type of packet)
$c->SetCallBacks('presence' => \&handle_presence);

# Send out our own presence, and run an
# event loop for up to 5 seconds to
```

```
# catch any packets pushed to us
$c->PresenceSend();
$c->Process(5);

# Create a new message with our reminder text
my $m = Net::Jabber::Message->new();
$m->SetBody($reminder);

# Send the message to each of the addressees collected
# in the handle_presence() subroutine
foreach my $jid (@addressees) {

  $m->SetTo($jid);
  $c->Send($m);

}

# Disconnect from the Jabber server and exit
$c->Disconnect;
exit(0);


# Deal with presence packets
sub handle_presence {

  my ($sid, $presence) = @_;

  # Get the presence
  my $show = $presence->GetShow() || 'online';

  # If the user is around, add to addressee list
  # 'around' here means 'online' or 'chat'
  push @addressees, $presence->GetFrom()
    if $show eq 'online' or $show eq 'chat';

}
```

The script is fairly self-explanatory. For now, we'll leave the script's description to the comments embedded within it; by the end of the book, you should have a good understanding of how to put together complete applications and utilities using Jabber libraries in Perl, Python, and Java.

Chapter 2. Inside Jabber

Table of Contents

Jabber has a number of features that are fundamental to its design philosophy. This *design philosophy* outlines Jabber as a much more flexible and generic solution to the original problem of connecting to disparate IM systems. These *features* give Jabber the potential to exist and act in the P2P (peer to peer, or person to person), A2P (application to person), A2A (application to application) - in fact in any of the three-letter TLAs that have a '2' in the middle - spaces that have conversation at their core.

Understanding Jabber's features is fundamental to seeing how it fits into the bigger picture. In this Chapter, we explore these features, and discover in what ways Jabber is *not* simply a cross-IM mechanism. In this exploration, you'll get a feel for how capable Jabber is of being integrated into 'conversational solutions'.

In Part II of the book, we'll revisit each of these features and see how they can be used in many different programming scenarios.

XML-Based

Arguments abound for and against XML in the arena of data representation. XML is suited extremely well to Jabber, which is suited extremely well to XML. This is for many reasons.

The alternatives for representing data in Jabber are: binary and ASCII text. Binary? Well, perhaps binary data is more space efficient, but where is that advantage in the general scheme of things these days? Near the bottom of my list, anyway, especially as it's always at the cost of readability. ASCII? Well, yes, of course, ASCII is human readable, but because Jabber data flow consists of a series of conversational chunks—independent constructions in their own right—we need some sort of boundary mechanism to separate these chunks. XML affords us a very nice way of packaging individual chunks of data and giving their content meaning and context: These individual chunks of information have structure too, and this structure doesn't require any fixed-length madness either; XML allows the chunks, or fragments, to bend and stretch as required, while still retaining their meaning.

This flexibility also comes in the form of extensibility. It's straightforward to add distinct "extensions" to a fragment in a way that does not compromise the integrity of that fragment, and provides a structure to the extension added.

So why reinvent the wheel when there are tools that can be taken off the shelf to parse the data? There are many tried and tested XML libraries out there, and to be able to receive (from the parser) the Jabber data in a native format of your choice is a definite advantage.

Some of these arguments, concerning XML fragments and extensibility, will become clearer in Chapter 5. Until then, consider that Jabber makes good use of an XML feature called *namespaces*. [1] Namespaces are used in XML to segregate, or qualify, individual chunks of data, giving tags a reference to which they belong. What the reference is, in many ways is of secondary importance - the point is the delineation that allows us to manage content within an XML fragment that is logically divided into sub-fragments. Consider Example 2-1, which shows a section of the imaginary conversation from Chapter 1.

Example 2-1. Qualifying a fragment extension with a namespace

```
<message type='chat' from='jim@company-a.com/home'
    to='john@company-b.com/Desk'>
  <thread>01</thread>
  <body>Here's the link</body>
  <x xmlns='jabber:x:oob'>
```

```
    <url>http://www.megacorp.co.uk/earnings3q.html</url>
    <desc>Third Quarter Earnings for Megacorp</desc>
  </x>
</message>
```

The main part of the fragment is the `<message/>` element containing the `<thread/>` and `<body/>` tags. The `<message/>` element is the "carrier" part of the fragment:

```
<message type='chat' from='jim@company-a.com/home'
    to='john@company-b.com/Desk'>
  <thread>01</thread>
  <body>Here's the link</body>
</message>
```

But the fragment has been embellished by an extension that is qualified by the '`jabber:x:oob`' namespace:

```
<x xmlns='jabber:x:oob'>
  <url>http://www.megacorp.co.uk/earnings3q.html</url>
  <desc>Third Quarter Earnings for Megacorp</desc>
</x>
```

The `xmlns` attribute of the `<x/>` tag declares that the tag, and any children of that tag, belong to, or are qualified by, the `jabber:x:oob` namespace. [2] This namespace is different to the namespace that qualifies the carrier `<message/>` tag, and the other elements `<presence/>` and `<iq/>` that appear at the same level. The namespace that qualifies *these* tags is not explicitly specified as an `xmlns` attribute; rather it is declared when the XML stream is established. It is over this XML stream that these elements flow. See Chapter 5 for more details on XML streams and namespaces.

The general point is that the `jabber:x:oob` qualified extension is recognizable as an extension (by us, and more importantly, by the XML parser) and can be dealt with appropriately—we are likely to want to handle the information contained in the extension separately from the rest of the message.

So Jabber uses the extensible XML format to contain and carry data between endpoints.

"XML between endpoints"? That sounds rather generic to me—not something that's limited to providing an IM experience. Indeed, that's the whole idea.

"XML Router" is a moniker often used to describe Jabber, by people who have made this logical leap. Remove the IM mantle, and underneath we find a system, an architecture, capable of being deployed to exchange and distribute all manner of XML-encoded data.

Notes

[1] http://www.w3.org/TR/REC-xml-names/

[2] This namespace is used in Jabber to carry information about "out of band" (OOB) data; data that moves *outside* of the main client-server-client pathways. When a client sends a file directly to another client without sending that file via the server, this is said to be "out of band".

Prev | Home | Next
Inside Jabber | Up | Asynchronous Nature

Asynchronous Nature

The exchange and distribution of information in real world scenarios requires more than a synchronous (sequenced) request/response framework. In IM, people originate chat messages in a spontaneous and unpredictable manner (especially if there's alcohol involved). In a loose network of independent applications, messages originate on a similar 'random' event basis. This asynchronous activity requires a design equally asynchronous in nature—and that is what Jabber has.

To allow for the generation—and more importantly the receipt—of messages in an asynchronous fashion, Jabber's programming model for client and server alike is an *event-based* one. An event-based programming model is nothing to be afraid of. In simple terms it's just a loop that iterates either doing nothing in particular or doing something regularly (like checking for user input and acting upon it if required) while waiting for something—an event—to happen. In Jabber's case, it will be the receipt of an XML fragment. Depending on what type of fragment it is, a *handler* will be called to deal with that fragment. We saw a simple example of this in the Perl script in Chapter 1.

In all but the simplest deployment examples of Jabber, the event model pervades. We will see the model in action in Part II. And while we've reduced the receipt of chat messages to a rather dry and generic event idea, let's also look at some concepts that often go hand in hand with event-based messaging systems.

Store and forward

Depending on circumstances, if you send a message to someone who's not currently connected, that message will be held and passed to the recipient when he does connect.

Likewise, in the wider context of application to application (A2A) or application to person (A2P) messaging, this store and forward concept is often useful in non-time-critical situations, such as a centralized logging mechanism where the log-writing component might be temporarily unavailable, or a workflow scenario where an application passes a message to a supervisor for approval (in this case, the message would be similar to an email).

Queuing

Indeed, in the case of the recipient being offline and messages being handled by a store and forward

mechanism, or simply where the recipient cannot handle the messages as fast as they arrive, the nature of the XML stream in which these message fragments are transmitted (see Chapter 5 for details) means that the messages are naturally *queued*, to be handled in the order that they arrive.

Message receipt

In many cases, it's not much use sending a message to an application, and have it queued or stored offline, without having an idea of whether the recipient actually did (eventually) receive and handle it. Jabber supports the concept of *message receipt*, whereby the sender can request an acknowledgment of receipt from the recipient. This request is sent along with the message itself. We'll see message receipt in action in the recipes in Part II.

Messaging

We've been using the term "message" in quite a general sense - to represent data passing from one Jabber entity to another. In fact, as we'll see in Chapter 5, there are different types, and subtypes, of message—and each one has a certain role within the whole context of the Jabber protocol. These messaging types are sometimes referred to as elements, and there are three of them - `<message/>`, `<iq/>` and `<presence/>`. [1]

The `<message/>` element has five subtypes - *normal*, *chat*, *groupchat*, *headline*, and *error*, for example. The `<iq/>` and `<presence/>` elements also have subtypes to distinguish and describe their usage and context. (The `<iq/>` element has types *get*, *set*, *result*, and *error*, while the `<iq/>` element has types *available* and *unavailable*.) Furthermore, we already know that these elements can be extended using namespaces. Each type and subtype, and each of the pre-defined namespaces (those that begin '`jabber:`') have been designed with specific scenarios in mind.

There's a certain amount of consideration to be given in this respect when designing our messaging solutions and applications; how should we employ the basic message types, and do we need to create our own custom extensions qualified by our own namespaces?

For the most part, the answers to these questions will depend on what sort of solution needs to be developed; however, it is also important to consider what support is already available 'off the shelf' in the form of Jabber clients. This is especially the case if the application is A2P or P2A. These Jabber clients provide varying levels of features supporting the different message types and subtypes. For example, WinJab (for Win32 platforms) and Jarl (cross-platform, written in Perl/Tk) both support the `<message/>` subtype *headline* and can display URL information, which typically comes attached to the message in a `jabber:x:oob` qualified extension, in a useful way. If you're writing a news headline alert mechanism, for example, you may want to consider aiming development with a target of WinJab or Jarl in mind; the alternative is to develop your custom news headline viewer client.

Notes

[1] Actually, there are four - but the fourth—`<route/>`— is only used by the server, to route messages between the various components.

Payload Carrier

Earlier in this chapter we saw how a basic message was embellished with a structured and parseable 'attachment' in the form of a `jabber:x:oob` qualified `<x/>` tag, and its child tags:

```
<x xmlns='jabber:x:oob'>
  <url>http://www.megacorp.co.uk/earnings3q.html</url>
  <desc>Third Quarter Earnings for Megacorp</desc>
</x>
```

How does this actually work in practice? Well, partly by requirement, partly by convention.

- All three of the elements `<message/>`, `<iq/>` and `<presence/>` can carry these attachments.
- These attachments usually server one of two purposes
    - to bring primary (or secondary) meaning to the element that contains it
    - to act as the driving force behind the element, such as that the element really only exists to carry the attachment

This distinction is rather subtle, so let's look at a couple of examples.

Example 2-2. A `jabber:x:delay` extension adds meaning to a `<presence/>` element

```
<presence from='jim@company-a.com/home' to='john@company-b.com'>
  <show>chat</show>
  <status>having a break from work</status>
  <priority>1</priority>
  <x xmlns='jabber:x:delay' from='jim@company-a.com/home'
     stamp='20010611T13:13:04'/>
</presence>
```

Example 2-2 shows an attachment in the `jabber:x:delay` namespace. This works like a timestamp, and in this context indicates when that presence element appeared, in other words, from what time Jim started on his break.

Example 2-3. A `jabber:x:oob` extension is the heart of a `<message/>` element

```
<message type='headline' from='jabbercentral@news.company-b.com'
         to='john@company-b.com'>
  <x xmlns='jabber:x:oob'>
    <url>http://www.jabbercentral.com/news/view.php?news_id='989358658</url>
    <desc>
      Tomorrow, May 9th, a meeting regarding the Jabber Foundation
```

```
      will be held.
    </desc>
  </x>
</message>
```

Example 2-3 shows us a message element containing a URL attachment. In this case the element serves to carry the URL attachment; there is not really any other purpose to it. Delivering the `<message type='headline'/>` carrier without the payload wouldn't make much sense.

- The outermost tag of each attachment, by convention, is either `<x/>` or `<query/>` although they can be any valid XML tag name. The Jabber namespaces that are used to qualify these `<x/>` and `<query/>` attachments follow a convention, beginning

  `jabber:x:`

  for those qualifying the `<x/>` attachments, and

  `jabber:iq:`

  for those qualifying the `<query/>` attachments.

  `<x/>` is used as a generic extension mechanism to add information to any of the Jabber elements, and `<query/>` is used to extend the `<iq/>` element.

Chapter 5 lists the standard namespaces used in Jabber to qualify the `<x/>` and `<query/>` attachments. But we're not restricted to just those namespaces - we're free to build our own attachments qualified by our own namespaces, if we think that what we want to achieve isn't covered by anything 'out of the box'. In this situation, there are two things to note:

- the namespaces cannot begin 'jabber:'
- the receiving recipient of the extended element must be able to appropriately interpret the attachments (!)

Request/Response

The HyperText Transfer Protocol (HTTP) is a great example of a simple, effective request/response model used on the Web to request, and respond to requests for HTML and other content. Example 2-4 shows a typical HTTP request/response pairing.

Example 2-4. A typical HTTP request/reponse

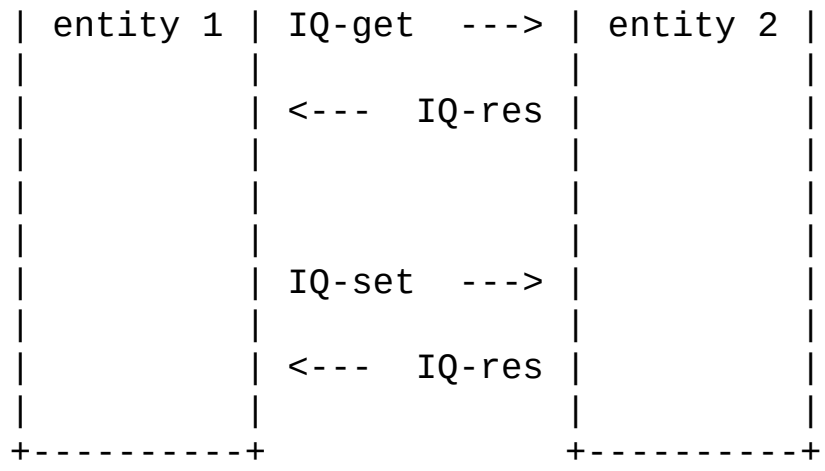
```
GET /home.html HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 11 Jun 2001 13:43:13 GMT
Server: Apache/1.3.12 (Unix) mod_perl/1.24
Last-Modified: Fri, 09 Jun 2000 13:47:56 GMT
ETag: "8a69-6a-3940f58c"
Accept-Ranges: bytes
Content-Length: 306
Connection: close
Content-Type: text/html

<html>
<head>
...
```

It shows us the request verb `GET`, and the specification of what to retrieve (the `/home.html` document), and it shows us what is returned in response.

While we've already seen that the Jabber protocol is asynchronous in nature, there is a similar request/response model available too, which tends to the synchronous (i.e., first request, then response) although unlike HTTP, the response is not guaranteed to *immediately* follow the request. Other unrelated Jabber fragments may be received in the stream in the time between request and response. This request/response model is called *Info/Query*, or IQ for short; the Jabber element employed in the implementation of this model is `<iq/>`.

This IQ model has many uses in providing Jabber's basic IM features. Figure 2-1 shows the element traffic in a typical IQ-based conversation.

Figure 2-1. Element traffic in an IQ-based conversation



Example 2-5, Example 2-6 and Example 2-7 show typical uses of the `<iq/>` element to effect the IQ request/response model.

Example 2-5. A simple client version query via the IQ model

```
SEND: <iq type='get' from='jim@company-a.com/home'
          to='mark@company-a.com/Laptop-7' id='v1'>
        <query xmlns='jabber:iq:version'/>
      </iq>

RECV: <iq type='result' to='jim@company-a.com/home'
          from='mark@company-a.com/Laptop-7' id='v1'>
        <query xmlns='jabber:iq:version'>
          <name>Jabber Instant Messenger</name>
          <version>1.7.0.14</version>
          <os>95 4.10</os>
        </query>
      </iq>
```

In Example 2-5 we see that the equivalent of HTTP's `GET` verb is the `type='get'` attribute of the `<iq/>` element, and that the equivalent of *what* to get (`/home.html`) is our namespace specification in the `<query/>` tag.

Example 2-6 shows a two-stage request/response model. First, a query is made, using the `<iq/>` 'get' type with an id of 'reg1', to discover which fields are required for user registration. Then, the actual user registration attempt is made with the `<iq/>` 'set' (id 'reg2'). In this second stage, we can see a link with HTTP's `POST` verb - whereby data is sent along with the request. [1]

Example 2-6. A multiple-phase IQ to register a user

```
SEND: <iq type='get' id='reg1'>
        <query xmlns='jabber:iq:register'/>
```

```
      </iq>

RECV: <iq type='result' id='reg1'>
        <query xmlns='jabber:iq:register'>
          <instructions>Choose a name and pass to register.</instructions>
          <username/>
          <password/>
        </query>
      </iq>

SEND: <iq type='set' id='reg2'>
        <query xmlns='jabber:iq:register'>
          <username>helen</username>
          <password>tr0y</password>
        </query>
      </iq>

RECV: <iq type='result' id='reg2'/>
```

While HTTP is primarily a client-server protocol (although these words have less and less meaning in a P2P (peer-to-peer) environment where each participating entity can be both client and server), Example 2-5 shows that the IQ model can be used in a client-to-client (or server-to-server, or server-to-client, for that matter) context as well.

Example 2-7 shows how an IQ-based conversation can extend beyond the *get ... result* or *set ... result* model. It shows how the IQ model can be used to deliver results of a search request in stages.

Example 2-7. Staggered response from IQ-based search request

The search request:

```
SEND: <iq type='set' to='ldap.company-a.com' id='801'>
        <query xmlns='jabber:iq:search'>
          <group>Support</group>
        </query>
      </iq>
```

The first part of the response—the first record found:

```
RECV: <iq type='set' from='ldap.company-a.com'
          to='jim@company-a.com/home' id='801'>
        <query xmlns='jabber:iq:search'>
          <name>John Aston</name>
          <phone>4701</phone>
          <group>Support</group>
```

```
        </query>
      </iq>
```

The second record found:

```
RECV: <iq type='set' from='ldap.company-a.com'
          to='jim@company-a.com/home' id='801'>
        <query xmlns='jabber:iq:search'>
          <name>Katie Smith</name>
          <phone>4711</phone>
          <group>Support</group>
        </query>
      </iq>
```

The last record found:

```
RECV: <iq type='set' from='ldap.company-a.com'
          to='jim@company-a.com/home' id='801'>
        <query xmlns='jabber:iq:search'>
          <name>Jeremy Taylor</name>
          <phone>4702</phone>
          <group>Support</group>
        </query>
      </iq>
```

No more records have been found and no more results are sent; instead, an "end marker", signified by the `type='result'` attribute, is sent.

```
RECV: <iq type='result' from='ldap.company-a.com'
          to='jim@company-a.com/home' id='801'>
        <query xmlns='jabber:iq:search'/>
      </iq>
```

Because of the inherently asynchronous nature of Jabber, we need some way of matching the responses received with the original request. After all, if we were to fire off two requests almost simultaneously to ask for the local time on our server and on a client in New Zealand:

```
SEND: <iq type='get' from='jim@company-a.com/home'
          to='piers@company-a.com/emacs'>
        <query xmlns='jabber:iq:time'/>
      <iq/>

SEND: <iq type='get' from='jim@company-a.com/home'
          to='company-a.com'>
```

```
      <query xmlns='jabber:iq:time'/>
    <iq/>
```

we might find that the responses come back out of sequence (in our view), because of the comparative network distances over which the two conversations must travel. First this:

```
RECV: <iq type='result' to='jim@company-a.com/home'
          from='company-a.com'>
        <query xmlns='jabber:iq:time'/>
          <utc>20010611T17:59:13</utc>
          <tz>CET</tz>
          <display>Mon Jun 11 19:59:13 2001</display>
        </query>
      </iq>
```

and then this:

```
RECV: <iq type='result' to='jim@company-a.com/home'
          from='piers@company-a.com/emacs'>
        <query xmlns='jabber:iq:time'/>
          <utc>20010611T17:59:14</utc>
          <tz>UTC+1200</tz>
          <display>Tue Jun 12 06:59:14 2001</display>
        </query>
      </iq>
```

That's the reason for the `id` attribute in the examples earlier in this section. Between a request and a response, any `id` attribute value in the `<iq/>` element is preserved, allowing the requester to match up `<iq/>` request / response pairs. Using the `id` attribute, we can piece together related fragments of individual conversations, which in this case were a pair of client time queries.

Notes

[1] Okay, data can also be sent with an HTTP `GET` request, but let's just not go there, for the sake of the comparison. ;-)

Component/Service Architecture

The original problem for Jabber to solve was to provide bridges to different IM systems; the upshot of the solution was a server design that is ultimately as flexible as the imagination allows. Each of the IM bridges, or *transports* as they are often called, is a pluggable component; the Jabber server architecture, examined in detail in Chapter 4, is a component-based architecture. The standard Jabber server distribution comes with the components required to provide IM services, data storage and server-to-server communication, for example. Each component is separately configurable and is made known to the server through the inclusion of that configuration into the main configuration structure. [1]

Components, also known as *modules* or *services*, can be, to a large extent, platform agnostic. There are different methods by which components can connect to and interact with the Jabber server. One method of which uses low-level Jabber library functions (in C) to bind the component—built in the form of a shared library—to the server. The other methods use either standard I/O (STDIO) in a process-spawning mechanism, or TCP sockets.

The former "library load" method and the STDIO method both require that the component runs on the same host as the Jabber server itself; this isn't in fact as restrictive as it sounds; as you'll find out in Chapter 4 it is possible to run multiple 'instances' of a Jabber server across different hosts, each instance containing one or more components, in much the same way as an SAP R/3 system can exist as multiple instances, each instance running on a separate host and providing different services (dialog, update, enqueue, message, background, gateway, spool) according to configuration.

Would it surprise you to learn that the binding fluid that flows between the Jabber server and components (and ultimately of course between Jabber clients and components) is XML? Of course not. In fact, it's the same XML that flows between Jabber clients and servers. [2] The `<message/>`, `<iq/>` and `<presence/>` elements can all flow to and from components—how a component handles or generates these elements reflects the purpose of that component. As we saw in the imaginary conversation in Chapter 1, components can also be addressed in the same way that other Jabber entities can be addressed:

- As a whole, the JID is simply the name of the component, for example `stocks.company-a.com`;
- Using individual "users" as if they existed as separate entities within that component, for example `MEGC@stocks.company-a.com`;

A version of this imaginary conversation will become reality in Part II as we learn how to build our own components and attach them to the Jabber server.

Although components are addressed in the same way that clients are addressed, and the interaction is similar, there is one significant difference between writing or using a *client* (which could just as well be a stub connector for a service), and writing or using a *component*.

This difference is rooted in the Jabber server architecture and becomes clear when we consider the nature of how the components provide their services. Messages sent to a client entity that is not connected will be stored and forwarded to that entity the next time it connects and becomes available. This is beause the client's connection is inherently hosted, or managed, by the standard component that provides IM services such as availability handling and message queueing. This is the Jabber Session Manager (JSM) component. All clients are handled this way and automatically partake of these features. All entities that connect over a stream with the `jabber:client` namespace are considered clients, and their XML-based interaction is handled by the JSM.

Because other components connected to the Jabber server are themselves peer components to the JSM component, no availability or message queueing is available, as the JSM is only designed to handle client-connected entities. [3] To put it another way:

- Send a message to a client that's not connected and it will get stored until that client connects and becomes available again.
- Send a message to a component that's not connected, and an error will occur.

From a philosophical standpoint, this is fair enough, as you're trying to address a (temporarily) nonexistent feature of the server. From a practical standpoint, this is not usually a problem unless you're in the habit of bouncing your Jabber server to randomly connect or disconnect components.

Notes

[1] XML is indeed all-pervasive in Jabber; the configuration is also written using XML.

[2] There are extra message types that flow inside the server and between the components,including `<route/>`, `<xdb/>`, and `<log/>`. These message types are explained in Chapter 4.

[3] The components that connect as separate entities (i.e., not those that are written as shared libraries) do so over a stream that is described by one of two namespaces, each beginning `jabber:component:`. See Chapter 4 and Chapter 5 for more details).

Custom Clients

Earlier in this chapter we discussed features of 'off the shelf' clients such as WinJab and Jarl, clients that natively support the headline `<message/>` element. Considering this, in combination with the features we know Jabber posesses, and the solution potential that these features offer in presenting a wider deployment vista than IM services, we come to an interesting conclusion:

> A Jabber client is a piece of software that implements as much of the Jabber protocol as required to get the job done.

What does this mean? WinJab supports a Jabber feature called *browsing* (see later in this chapter), Jarl supports connections to the server via an HTTP proxy. Some clients merely support a limited subset of Jabber as we know it; for example the client sjabber supports only the conferencing features of Jabber. (As a somewhat biased observer, I would of course call this "extremely focused").

Our definition of a Jabber client deliberately omits any mention of a user interface (let alone a GUI!). Indeed, human interaction in a Jabber solution is only an assumption formed from the hangover of the IM idea. Various efforts are underway to use Jabber as a transport for A2A messaging—the Jabber As Middleware (JAM) project is one such effort; [1] an extension to the Perl libraries for Jabber (Net::Jabber) to carry XML-RPC-encoded messages also exists. [2]

So, a Jabber client doesn't *need* to follow any particularly conventional model, except if you're developing an IM client, of course, and even then, flexibility and fitness for purpose is key. We will see this philosophy in action in Part II of the book, where we develop just enough Jabber client code to be able to connect to a server, authenticate, and deal with one-way alert-style messages. This is a key idea: you use as much, or as little, of what Jabber has to offer to build your solutions. This doesn't just refer to the Jabber building blocks, but also to the existing software, in particular the clients. For example, if you wanted to develop a Jabber IM-based approval cycle workflow process you could carry the approval data in a custom namespace-qualified message extension. In this case, you're going to have to build a custom client to interpret that extension. However, if you carry the approval data inside the body of a normal message then you can more or less take your pick of ready-made clients. Furthermore, if you include URLs in the message body —which many graphical clients can render into an active link— you can easily bind in the power of existing web-based interactivity to complete the solution.

Notes

[1] XXX where JAM exists

[2] XXX where Net::Jabber::XMLRPC exists

XML-RPC and SOAP

Realising that if Jabber is an XML router that can carry XML-based custom payloads in synchronous and asynchronous modes, we can immediately start to imagine scenarios where Jabber fits as a transport glue in other already established application domains.

The XML Remote Procedure Call (XML-RPC) specification and the Simple Object Access Protocol (SOAP) both formalize method calls and responses in XML. XML-RPC was designed to use HTTP to carry those encoded calls and responses between endpoints, and SOAP can use HTTP too. What if we carry XML-RPC or SOAP payloads in Jabber? We immediately see the step-change increase in contextual richness; XML-RPC interactivity becomes part of a wider, all-encompassing conversation between applications and humans. Traditional IM-based clients, applications using client stub libraries, and components can all make use of the power of what these technologies have to offer, without having to leave the comfort of their Jabber environment, which can serve as a messaging plasma between all sorts of entities and services.

We'll have a look at embedding XML-RPC and SOAP calls in Jabber messages in Part II of the book.

Browsing

"All sorts of entities and services." The more pervasive Jabber becomes, and the more uses to which it is put, the larger the world of entities grows. And therein lies the challenge. How can we identify, organize and navigate these entities in any useful way? Within the relatively narrow world of IM, the entities that exist (the IM users) and the hierarchies and relationships that are formed between them (where the only hierarchies are in the form of user rosters) don't really present much of a problem; as all the entities are users, there's no classification problem, and there's no hierarchical navigation problem. But within a system that regards users, applications, and services as equals (the unifying concept of a JID is an especially powerful device here), we need to have some way of searching, traversing, discovering, and querying these entities that is consistent regardless of what those entities are. Enter *Jabber Browsing*.

Browsing was introduced with the 1.4 Jabber server to solve some specific problems with service discovery: how clients found out about what services (say, what IM transports) were available on the Jabber server that they were connected to. The namespace mechanism that supported this discovery (`jabber:iq:agents`) was found to be too restrictive, and more importantly, too specific. What was needed was a more generic way of describing entities in the Jabber world.

Browsing has since grown from that single problem space and can now be found in Jabber software everywhere. Want to find out what a user's client is capable of (so you can interact with it) or what it otherwise offers? Just browse to it. Want to find out what conference rooms currently exist on a particular conference service? Browse that service. Want to take a peek to see who's in one of the rooms? Navigate down one level to browse to the room. [1]

There are three key elements that make browsing so flexible and so powerful:

Categorization

> Browsing follows the MIME model in defining categories and subcategories, but rather than *content* being categorized (`text/plain`, `image/png`, and so on), the categorization in browsing is of of entities that can be browsed to. Categories are used to describe users, services, agents, conferencing mechanism, applications, and so on. Within these categories, the subcategories are used to make finer distinctions.

Identification

> Every entity described in browsing is identifiable by a JID; the world is homogenized, so to speak. The JID is the key to browsing navigation as, when listed in a browse result as identifiation

for an entity, that JID can be used as the target of the *next* browse request.

Hierarchy

> Browsing describes entities in the Jabber world. But the world isn't *flat* ... it's hierarchical! Relationships between entities are easily established and described by placing browse information in hierarchies. These hierarchies represented in XML. A typical browse response might contain descriptions of entities on one or two levels. These levels can be navigated simply by choosing the JID of the desired node and making a further browse request, creating an instant "drill-down" method of accessing information. 'drill-down'.

So, Jabber entities can be classified, given identities, organized into hierarchies, and navigated. What takes place from there is really down to the imagination. But what *is* a Jabber entity? For the sake of argument, let's say it's anything that has a JID. Does having a JID presume having an IM user account with a Jabber server? Not necessarily, as we'll see. So as long as we assign a JID to something we wish to include in the Jabber world, most anything goes. In the section called *Browsing LDAP* in Chapter 9 we build an LDAP 'reflector' service that enables us to browse LDAP information from within our Jabber clients.

Browsing is a combination—a culmination even—of many of the Jabber features. Browsing is carried out in the context of the IQ request/response mechanism, and uses a namespace-qualified payload to carry the data whose hierarchy is naturally expressed in XML. And it bridges, philosophically and technically, the distances between the Jabber and non-Jabber spaces.

Notes

[1] If this is permitted by the room.

Chapter 3. Installing the Jabber Server

Table of Contents

This chapter explains what you have to do to obtain, install, configure and start up a Jabber server of your own with the minimum of fuss.

It's certainly possible to learn about the Jabber protocols and technology and develop solutions using someone else's Jabber server, but for real understanding and control, it's definitely worth setting up one of your own. By installing and configuring a Jabber server, you will gain a valuable insight into how the Jabber server, and its components, work together. Understanding how components are controlled and configured allows us to build Jabber solutions in the context of the 'big picture'.

Installation of earlier versions (1.0, 1.2) of the Jabber server were often complex affairs, and while the installation process has become much more straightforward, some people still shrink back from installing and configuring their own. This chapter shows how straightforward it is.

If you already have a server set up, you might want to skip this chapter and go on to Chapter 4 where the configuration and system architecture is explained in more detail.s

Although the Jabber development platform is Linux, the Jabber Server will compile and run on many flavours of Unix, including FreeBSD, Solaris, AIX, and IRIX. Versions of the C compiler and **make** utility from the GNU project [1] are recommended if you don't already have them installed.

The examples shown in this and other chapters are taken from a Linux platform; consult your local documentation for equivalent commands on your Unix OS.

Notes

[1] The GNU project is at http://www.gnu.org

The Jabber Server

The incarnation of the Jabber Server at the time of writing is version 1.4, more specifically 1.4.1. Version 1.4 represents a major step towards the 2.0 release and brings stabilisation of the server code and increases in performance and reliability over earlier versions. 1.4.1 is the version of the Jabber Server we will obtain and install here, and this will be used as the server for the recipes in the rest of this book.

Getting It

The Jabber Server package can be obtained from the Jabber project site - http://www.jabber.org; the 1.4.1 version is available in the downloads area:

http://download.jabber.org/dists/1.4/final/jabber-1.4.1.tar.gz

The tarball `jabber-1.4.1.tar.gz` contains everything that you need to get a Jabber Server up and running. [1] Previous versions of the Jabber server came in multiple packages - it was necessary to separately obtain and install GNU's portable threads library (**pth**) and the asynchronous DNS package (**ADNS**), as well as obtaining and installing various Jabber-specific libraries such as **libxode**, **libjabber** and **libetherx**. Now some of these libraries and packages have become obsolete as far as the Jabber Server is concerned (**ADNS** and **libetherx**) and others have been combined into the main Jabber Server tarball.

Installing It

Once you have downloaded the Jabber Server tarball, you need to unpack, configure the build environment, and compile it. The general idea is that the Jabber server will be compiled and run from wherever you decide to unpack it; that is, there is no separate 'install' step.

For this reason, and because it's also often useful to be able to install and start up a different version of the Jabber Server for testing and comparisons, create a generic 'jabber' directory somewhere central but local, for example in `/usr/local/`:

```
yak:/usr/local# mkdir jabber
```

The Jabber Server does not and should not be run as `root`; so create a new user `jabber` (group `jabber`) to be used as the Jabber Server administrator and make that user the owner of the generic Jabber Server directory:

```
yak:/usr/local# groupadd jabber
yak:/usr/local# useradd -g jabber -d /usr/local/jabber jabber
yak:/usr/local# passwd jabber
Changing password for jabber
Enter the new password (minimum of 5, maximum of 127 characters)
Please use a combination of upper and lower case letters and numbers.
New password: ********
Re-enter new password: ********
Password changed.
yak:/usr/local# chown jabber:jabber jabber
yak:/usr/local#
```

Once you've created the generic Jabber Server directory, switch to the new Jabber Server administration user `jabber`, unpack the tarball you downloaded and enter the resulting directory:

```
yak:/usr/local# su - jabber
yak:~$ tar xzf jabber-1.4.1.tar.gz
yak:~$ cd jabber-1.4.1/
yak:~/jabber-1.4.1$
```

configure

Examining the contents of the `jabber-1.4.1` directory, we see the following files:

- `configure` (the configuration script)
- `jabber.xml` (the server configuration file)
- `Makefile` (the Makefile for the Jabber server)
- `README` (some basic instructions)
- `UPGRADE` (information on upgrading from an earlier server version)

as well as a number of directories that contain the source code.

The first step is to run the **configure** script:

```
yak:~/jabber-1.4.1$ ./configure
```

to determine your platform's compiler settings.

If you want SSL support in the Jabber Server, run the script with the `--enable-ssl` switch:

```
yak:~/jabber-1.4.1$ ./configure --enable-ssl
```

If you specified the `--enable-ssl` switch, the **configure** script will look for your SSL installation and add the appropriate compiler flags. If it doesn't find your SSL installation, it will say so and your Jabber Server will be compiled *without* SSL support.

Next, it will try to determine whether you have **pth** installed, and if so will use the **pth-config** command to glean the extra compiler options for building the Jabber Server. **pth** is required, so if it *isn't* already installed, it will be set up within your current `jabber-1.4.1` directory tree (as **pth** is included in the `jabber-1.4.1.tar.gz` tarball) and the appropriate compiler options added.

(If **pth** is set up during the course of running **configure**, you may see a message: "`Now please type 'make' to compile. Good luck.`" which comes at the end of the **pth** configure procedure; you can ignore this because there is *only one* **make** step, for the Jabber Server, that must be carried out as we are merely preparing the **pth** build environment for binding into the Jabber Server build.)

Finally, after extra platform specific compiler settings are determined, a shell script to set the build environment variables is created with the name `platform-settings`. This is used in the next step.

Example 3-1 shows typical output from the **configure** script.

Example 3-1. Typical output from configure

```
Running Jabber Configure
========================
```

```
Getting pth settings...         Done.
Setting Build Parameters...     Done.
Generating Settings Script...   Done.

You may now type 'make' to build your new Jabber system.
```

make

Once the platform settings have been determined by the **configure** script, we are ready to build the Jabber Server with **make**.

```
yak:~/jabber-1.4.1$ make
```

Example 3-2 shows abbreviated typical output from the **make** command.

Example 3-2. Typical output from make

```
Making all in pthsock
make[1]: Entering directory `/usr/local/jabber/jabber-1.4.1/pthsock'
gcc -g -Wall -fPIC -I. -I.. -I/usr/local/include -I../jabberd/   -c client.c -o
client.o
gcc -g -Wall -fPIC -I. -I.. -I/usr/local/include -I../jabberd/ -shared -o pthsoc
k_client.so client.o -L/usr/local/lib -lpth -ldl -lresolv
make[1]: Leaving directory `/usr/local/jabber/jabber-1.4.1/pthsock'
Making all in xdb_file
make[1]: Entering directory `/usr/local/jabber/jabber-1.4.1/xdb_file'
gcc -g -Wall -fPIC -I. -I.. -I/usr/local/include -I../jabberd   -c xdb_file.c -o
 xdb_file.o

...

gcc -g -Wall -fPIC -I. -I.. -I/usr/local/include -DHOME="\"/usr/local/jabber/jab
ber-1.4.1\"" -DCONFIGXML="\"jabber.xml\"" -o jabberd config.o mio.o mio_raw.o mi
o_xml.o mio_ssl.o deliver.o heartbeat.o jabberd.o load.o xdb.o mtq.o static.o lo
g.o lib/expat.o lib/genhash.o lib/hashtable.o lib/jid.o lib/jpacket.o lib/jutil.
o lib/karma.o lib/pool.o lib/pproxy.o lib/rate.o lib/sha.o lib/snprintf.o lib/so
cket.o lib/str.o lib/xmlnode.o lib/xmlparse.o lib/xmlrole.o lib/xmltok.o lib/xst
ream.o lib/xhash.o base/base_connect.o base/base_dynamic.o base/base_exec.o base
/base_stdout.o base/base_accept.o base/base_file.o base/base_format.o base/base_
stderr.o base/base_to.o -Wl,--export-dynamic -L/usr/local/lib -lpth -ldl -lresol
v
make[2]: Leaving directory `/usr/local/jabber/jabber-1.4.1/jabberd'
make[1]: Leaving directory `/usr/local/jabber/jabber-1.4.1/jabberd'
make[1]: Entering directory `/usr/local/jabber/jabber-1.4.1'
make[1]: Nothing to be done for `all-local'.
make[1]: Leaving directory `/usr/local/jabber/jabber-1.4.1'
```

Running From Build Environment?

You may be wondering where the **make install** step is: there isn't one— the Jabber Server is run from within its build environment. One of the reasons for this is that additional components, such as transports, which may be installed at any time after the basic server installation, must be compiled with reference to various Jabber Server header file information. One of the simplest ways of making this happen is to have the source for those components unpacked in a subdirectory within the `jabber-1.4.1` directory tree, and at compilation time component-level references to header files at the Jabber server level can be made using relative directory names that point back up the directory hierarchy.

Configuring The Jabber Server

The nature and behavior of a Jabber Server is controlled by the contents of a configuration file (with a default name of `jabber.xml`) which you will find in the `jabber-1.4.1` directory. As you can probably guess from the filename's extension, the configuration is formatted in XML, and affords a very powerful way of expressing the nature and features of your Jabber Server and associated services and components.

Details on how to navigate, interpret, and edit this configuration file are given in the next chapter; here we will just look at the basic settings that can be modified before you start up the Jabber Server.

For an experimental Jabber Server (such as for the purposes of this book) there isn't actually anything you *need* to change in the configuration. The out-of-the-box configuration settings are pretty much what we need to experiment with our recipes later in the book; nevertheless, let's look at some of the settings which you may wish to change right now.

- *Server Hostname*

  The `<host/>` parameter specifies the Jabber Server's hostname. As delivered, the `jabber.xml` configuration has this set to `localhost`:

  ```
  <host><jabberd:cmdline flag="h">localhost</jabberd:cmdline></host>
  ```

  You can change this to the name of your server hostname - in the case of our examples this would be `yak`.

  The `localhost` setting occurs elsewhere in the configuration too - as a literal in the welcome message that is sent to users after a successful registration with the server. You may wish to replace this occurrence of `localhost`; furthermore, you will find other occurrences but they are within sections of the configuration that are commented out in the standard delivered version of `jabber.xml` (specifically administration JIDs, and definitions for various add-on agents and transports; we will cover these in the next chapter).

  One other place that `localhost` occurs is in the `<update/>` section, which is explained next.
- *Server Software Update Notification Mechanism*

  The Jabber Server development team offer a facility for servers to check for updated versions of the Jabber Server software. The facility is addressed with this configuration setting:

  ```
  <update><jabberd:cmdline flag="h">localhost</jabberd:cmdline></update>
  ```

  which causes a versioning module (`mod_version`) to send a `<presence/>` packet (which carries the server version—in our case 1.4.1) from the server to the Jabber ID `jsm@update.jabber.org` when the Jabber Server starts up.

If your server is purely internal, and / or behind a firewall, it makes no sense to have this facility switched on (you can check for updates to the server on the http://www.jabber.org website) as the `<presence/>` packet will never reach its intended destination. You can comment it out like this:

```
<!--
<update><jabberd:cmdline flag="h">localhost</jabberd:cmdline></update>
-->
```

- *Automatic User Directory Update*

  The configuration as delivered contains a directive:

  ```
  <vcard2jud/>
  ```

  which means that any vCard data that is maintained by a Jabber client will be automatically passed on to the central user directory (the "JUD" - Jabber User Directory), defined elsewhere in the `jabber.xml` as the one at `jabber.org`, `users.jabber.org`. [2]

  If you've commented out the update notification mechanism because you're not going to be able to (or want to) reach the servers at `jabber.org`, then you might as well comment this out to avoid error messages being sent to Jabber clients when vCard data is modified: [3]

  ```
  <!--
  <vcard2jud/>
  -->
  ```

  Alternatively, instead of commenting out the `<vcard2jud/>`, you could of course comment out the definition of the JUD service in the `<browse/>` section:

  ```
  <!--
  <service type="jud" jid="users.jabber.org" name="Jabber User Directory">
    <ns>jabber:iq:search</ns>
    <ns>jabber:iq:register</ns>
  </service>
  -->
  ```

  because the mechanism looks in the `<browse/>` section for a reference to a JUD service; if there isn't one there, no vCard update will be sent.

You may have noticed that the values for each of these two settings (`<host/>` and `<update/>`) were wrapped in another tag:

```
<jabberd:cmdline flag="h">...</jabberd:cmdline>
```

This means that you can override the setting with a command-line switch (or 'flag'), in this case `-h`. So in fact, you don't even need to modify the `jabber.xml` configuration at all, if you specify your hostname when you start the server up (the welcome message will not be changed, of course).

Starting and Stopping the Jabber Server

At this stage, we have a Jabber server with enough basic configuration to be able to start it up and have it do something useful (like accept client connections). If you're curious about the rest of the configuration you encountered while editing the `jabber.xml` file,

you can jump to Chapter 4. Otherwise, let's start it up!

Starting the Server

The basic invocation looks like this:

```
yak:~/jabber-1.4.1$ ./jabberd/jabberd
```

but if you haven't bothered to change `localhost` anywhere in the configuration (as described earlier) you can use the `-h` switch to specify the hostname:

```
yak:~/jabber-1.4.1$ ./jabberd/jabberd -h yak
```

As it stands, there's a directive in the standard `jabber.xml` configuration file that specifies that any server error messages are to be written out to STDERR:

```
<log id='elogger'>
  <host/>
  <logtype/>
  <format>%d: [%t] (%h): %s</format>
  <file>error.log</file>
  <stderr/>
</log>
```

So either comment the directive out:

```
<!--
  <stderr/>
-->
```

Or redirect STDERR to `/dev/null`:

```
yak:~/jabber-1.4.1$ ./jabberd/jabberd -h yak 2>/dev/null
```

You won't lose the error messages - as you can see they're also written to the `error.log` file.

Assuming you wish to free up the terminal session after starting the server, you can send it to the background:

```
yak:~/jabber-1.4.1$ ./jabberd/jabberd -h yak 2>/dev/null &
```

Stopping the Server

To stop the server, just kill the processes, and it will shut down gracefully:

```
yak:~/jabber-1.4.1$ killall jabberd
```

or:

```
yak:~/jabber-1.4.1$ kill `cat jabber.pid`
```

`jabberd` Command Line Switches

We've seen the `-h` switch to specify the host when starting the server up; there are other switches available on the command line too; here is a list:

Table 3-1. Command Line Switches

Switch	Relating to	Description
`-c`	Alternate configuration	Use this to specify an alternative configuration file if you don't want to use `jabber.xml`
`-D`	Debugging info	Specifying this switch will cause (a large amount of) debugging information to be sent to STDERR.
`-h`	Hostname	The hostname of the Jabber Server
`-H`	Home folder	Used to specify 'home' folder or directory
`-s`	Spool area	The directory where the Jabber Server stores data via the `xdb_file` module
`-v`	Show version	Reports Jabber Server version and exits
`-V`	Show version	Same as `-v`
`-Z`	Debugging info	Limit the debugging information to certain 'zones' (comma separated) [a]

Notes:
a. the 'zones' are the filenames that immediately follow the timestamp in the debug log records, for example 'xdb_file' or 'deliver'.

Starting the Jabber Server with any unrecognized switches will cause it to show you a list of valid switches:

```
[yak: ~/jabber-1.4.1]$ ./jabberd/jabberd -badswitch
Usage:
jabberd &
 Optional Parameters:
 -c             configuration file
 -D             enable debug output
 -H             location of home folder
 -v             server version
 -V             server version
```

Yes, the list isn't complete. If the common switch `-h` were present in the list, we could almost consider the unlisted switches as undocumented, but it is isn't present, so we won't.

Monitoring and Troubleshooting

We've already seen a glimpse of the configuration relating to logging of messages in the previous section. As standard, the Jabber Server configuration describes two types of logging record and a recipient file for each type:

Error logging

Error log records are written to `error.log` in the current directory, as determined thus:

```
<log id='elogger'>
  <host/>
  <logtype/>
  <format>%d: [%t] (%h): %s</format>
  <file>error.log</file>
  <stderr/>
```

```
</log>
```

Statistical logging

Statistical log records used for tracking purposes are written to `record.log` in the current directory, as determined thus:

```
<log id='rlogger'>
  <host/>
  <logtype>record</logtype>
  <format>%d %h %s</format>
  <file>record.log</file>
</log>
```

Log records of this type are written when a client connects to the server, and when a client disconnects.

Furthermore, we can use the debugging switch (`-D`) when we start the server and have debugging and trace output written to STDERR.

If your Server Doesn't Start

There are a number of things to check that are likely candidates for preventing your server from starting.

Bad XML configuration

It is not difficult to make errors (typographical or otherwise) in the server configuration. The first line of defense is to be careful when editing your `jabber.xml` file. After that, the Jabber Server isn't going to be too forthcoming with information if you have broken the well-formedness of the XML:

```
yak:~/jabber-1.4.1$ ./jabberd/jabberd -h yak
Configuration parsing using jabber.xml failed
```

Help is at hand in the shape of Perl and the XML::Parser module, which is a wrapper around the XML parser, **expat**.

Providing you have Perl and the XML::Parser module installed, you can get **expat** to give you a clue where the XML is broken:

```
yak:~/jabber-1.4.1$ perl -MXML::Parser -e 'XML::Parser->new->parsefile("jabber.xml",
ErrorContext => 3)'

not well-formed (invalid token) at line 47, column 35, byte 1750:
    be on one line, the server doesn't like it otherwise! :)
    -->

    <host><jabberd:cmdline flag="h"yak</jabberd:cmdline></host>
==================================^

    &lt;!--
    This is the custom configuration section for the
 at /usr/local/lib/perl5/site_perl/5.6.0/i586-linux/XML/Parser.pm line 185
yak:~/jabber-1.4.1$
```

This shows us exactly where our problem is. In this case I inadvertently removed the close-tag symbol ">" when replacing `localhost` with `yak`.

No XML is Bad XML!

If you don't use the `-c` switch to specify which configuration file to use, the standard `jabber.xml` will be used. But if that file cannot be found, you get *exactly the same error* as if your XML was not well-formed. You have been warned!

Unable to Listen On Port(s)

Taking the standard `jabber.xml` configuration, the Jabber Server will try to bind to and listen on two ports: 5222 (for incoming client connections) and 5269 (for server to server connections). If other processes are already listening to these ports, then the Jabber Server cannot start and you will see something like this in the `error log`:

```
20010407T12:11:06: [alert] (-internal): io_select unable to listen on 5222 [(null)]
20010406T12:11:06: [alert] (-internal): io_select unable to listen on 5269 [(null)]
```

If this is the case, you can check the status of the ports with the **netstat** command:

```
yak:~/jabber-1.4.1$ netstat -an | grep -E '5222|5269'
tcp        0      0 0.0.0.0:5269            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:5222            0.0.0.0:*               LISTEN
```

If you see entries like this, it means that processes have been bound to these ports on *all* IP addresses. [4] For example, if `0.0.0.0:5222` is being listened to then you may have another instance of a Jabber Server already running.

On BSD systems, you cannot bind to the 'default' null address; the same error messages will be issued as if the ports were already bound. In the standard `jabber.xml` configuration file, a bind to the null address is specified for each port as standard; what you must do is change this and specify an explicit IP address for each of the ports in the configuration. That is, instead of:

```
<ip port="5222"/>
```

do something like this:

```
<ip port="5222">127.0.0.1</ip>
```

Notes

[1] If you want the Jabber Server to support SSL connections, you will need to have installed an SSL package; see the next chapter for more details.

[2] A "virtual" business card containing contact information.

[3] Some Jabber clients such as Jabber Instant Messenger require vCard information to be entered when registering for a new account, which means that an attempt to contact `users.jabber.org` would be made the first time a user connects.

[4] This "all" relates to the `(null)` shown in the "unable to listen" error messages shown earlier.

Chapter 4. Server Architecture and Configuration

Table of Contents

If you followed the previous chapter through you should now have a Jabber server of your own up and running. If you did go ahead and make the configuration changes described there, you may be curious to find out about the other 99 percent of the configuration file contents - what they do, what sort of structure (if any) exists, and how you might modify the configuration to suit your own requirements.

Despite the initially daunting and seemingly random nature of the `jabber.xml` file contents, there *is* a structure to the configuration. This chapter will take you through that structure, explaining how all the pieces fit together, and describing what those pieces do. In order to understand the configuration structure, we examine the nature of the server architecture itself. This architecture is reflected in that structure, and if we are to understand the latter, it helps to understand the former.

Indeed, in order to take the best advantage of what Jabber has to offer in terms of being a basis for many a messaging solution, it's important to understand how the server works, and how you as a programmer fit in. Jabber programming solutions can exist at different levels within the Jabber architecture; understanding this architecture can help you make better decisions about what needs to be done to build a solution.

So in this chapter, we'll take a look at the Jabber server architecture, and follow that by an in-depth tour of the server configuration in `jabber.xml`. Finally we'll have a look at some of the server "constellation" possibilities—how you can organize parts of the server to run over different hosts, and how you can make the server host multiple virtual server identities.

The Jabber Server Up An Overview of the Server Architecture

An Overview of the Server Architecture

In order to understand the configuration directives and how they work, it is necessary to take a step back and look what the Jabber Server really is.

jabberd and Components

The Jabber Server is a *daemon* **jabberd** that manages the flow of data between various *components*, components which collectively make up the *Jabber service*. There are different components, each of which perform different kinds of tasks, and there is a basic set of components that are required for a simple Jabber Server such as the one we configured and installed in the previous chapter.

The following list shows what the basic Jabber components are and what services they provide. It's worth considering the original and most well-known application of Jabber—instant messaging—and a Jabber design feature (distributed server architecture) to put this list into context and make better sense of it.

Session management

: We need to be able to manage users' sessions while they're connected to the server. The component that does this is called the Jabber Session Manager (JSM), and provides IM features such as message store and forward and roster management as well as managing sessions.

Client (to server) connections

: This is the component that manages the connections between clients and the server. It is known internally as '`c2s`'.

Server (to server) connections

: If there's a requirement to send a message from a user on one Jabber server to a user on another Jabber server, we need a way for the servers to connect to each other. This component establishes and manages server-to-server connections, and is known internally as '`s2s`'.

Logging

: As in any server system, the ability to log events (error messages, notices, alerts and so on) is essential. The logging component allows us to effect such logging.

Data storage

: There will be some server-side storage requirements, for example, to hold authentication

information and data such as last connect time; not to mention storage of rosters, personal details, and private data. The Data storage component does this for us. It is known internally as the '`xdb`' component. `xdb` stands for *X*ML *D*ata *B*ase.

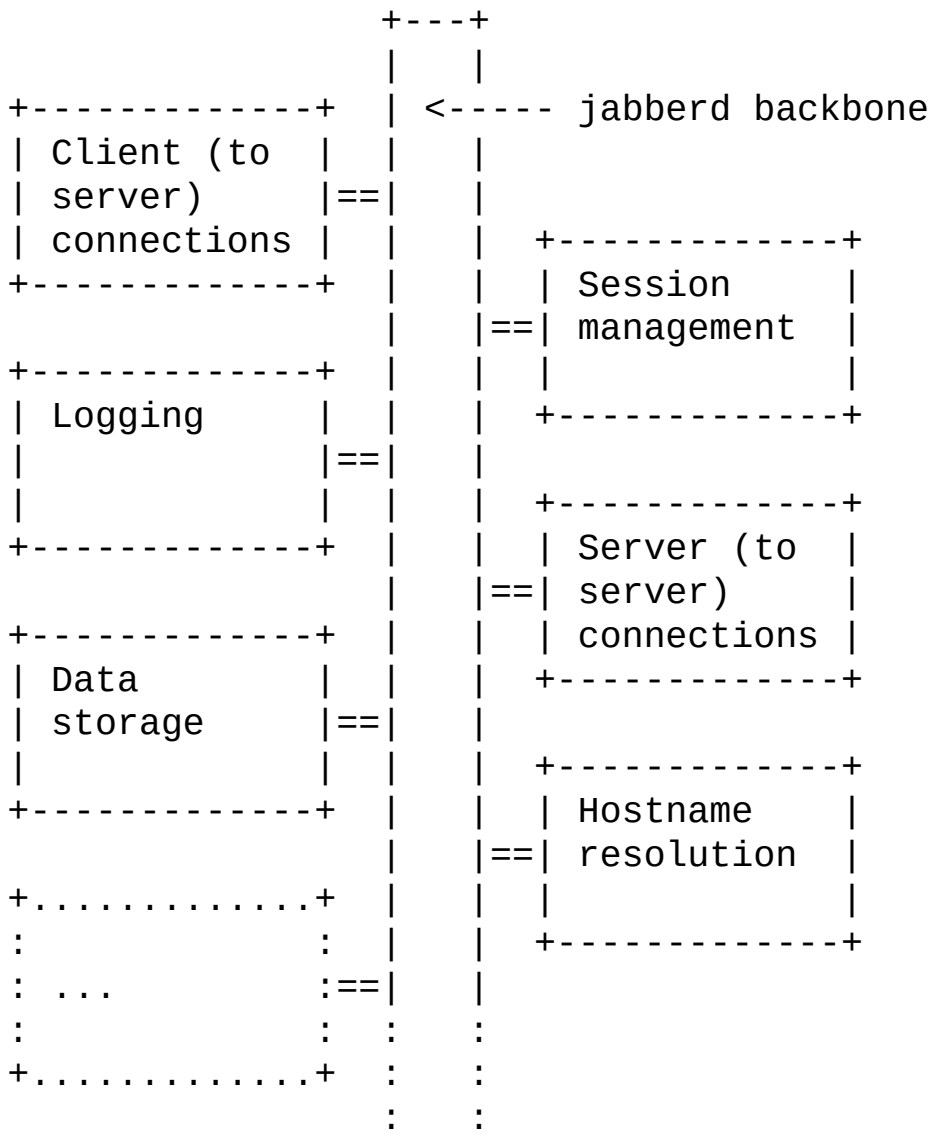
Hostname resolution

Last but not least, we may need some way to resolve names of hosts that the Jabber server doesn't recognize as "local", as in the Server (to Server) Connection context. This component is known internally as '`dnsrv`'.

Component types

Figure 4-1 shows the relationship between **jabberd** and the components. These components are the engines that serve and process XML messages, providing their services, and **jabberd** is the backbone along which messages are routed.

Figure 4-1. jabberd and the components



```
+---+
```

As seen in Figure 4-1, the **jabberd** backbone acts as the central artery or "hub", managing the "peripheral" components that are attached to it. The management of these components encompasses controlling and overseeing how they connect and coordinating the flow of data between them. Certain types of components receive only certain types of data. There is a distinction made between three different *types* of component:

- log
- xdb
- service

The different component types handle different types of data packets. Each packet is in the form of a distinct, fully-formed XML fragment and is identified by the outermost element name in the XML fragment. This element name is matched up to a particular component type.

log components

log components handle `<log/>` packets; you can guess that these are the components that provide *logging* services.

On receipt of a `<log/>` data packet a logging component will (hopefully) do something useful with it, like write it to a file or to STDERR.

The `<log/>` packet shown in Example 4-1 is being used to record the successful connection and authentication of user *dj*, on `yak`, using the Jabber client JabberIM.

Example 4-1. A `<log/>` packet

```
<log type='record' from='dj@yak'>login ok 192.168.0.1 JabberIM</log>
```

xdb components

xdb components handle `<xdb/>` packets. The `<xdb/>` packets carry data and storage/retrieval requests to and from the xdb components which provide the *Data Storage* services.

On receipt of an `<xdb/>` data packet an xdb component will retrieve data from or write data to a storage system such as a collection of flat files or an RDBMS.

The `<xdb/>` packet shown in Example 4-2 is carrying a request from the session manager to retrieve the preferences stored in a private namespace for the user *dj* (on Jabber server `yak` by the Jabber client JabberIM).

Example 4-2. An <xdb/> data packet

```
<xdb type='get' to='dj@yak' from='sessions' ns='jabberim:prefs' id='5'/>
```

service components

service components handle the three main building blocks on which the Jabber functionality is based (the <message/>, <presence/>, and <iq/> packets). You can find out more about these building blocks in Part II of this book.

In addition, the *service* component also handles the <route/> packets, which are used internally by **jabberd** to move packets around between components. For example, the *Session Management* component is the component that usually handles client authentication. It receives any incoming authorization requests received by and passed on from the *Client (to Server) Connections* component. However, it may be that the administrator has configured the Jabber server to use a different (third party) component, developed by another group or company, to handle the authorizations. In this case the request is *routed* from one component (*Session Management*) to another (the third-party authorization component).

So unlike the log and xdb components, which handle data packets whose element names match the component type (<log/> and <xdb/>), the service component is an umbrella component designed to handle packets with different element names.

Example 4-3 shows two typical service packets.

Example 4-3. Two service packets

```
<route to='dj@yak/81F2220' from='15@c2s/80EE868'>
  <presence>
    <status>Online</status>
  </presence>
</route>

<message id="jim_id_7" to="sabine@merlix" type="chat">
  <x xmlns="jabber:x:event">
    <composing/>
  </x>
  <thread>3A378DF2B70F6A53A9C317CF526C6B7A</thread>
  <body>Hi there</body>
</message>
```

The first is an internal <route/> packet, which is carrying a <presence/> packet from the *Client (to Server) Connections* component, identified by the 'c2s' part of the from attribute, to the *Session*

Management component (where the session identifier 81F2220 [1] is significant in the `to` attribute.). [2]

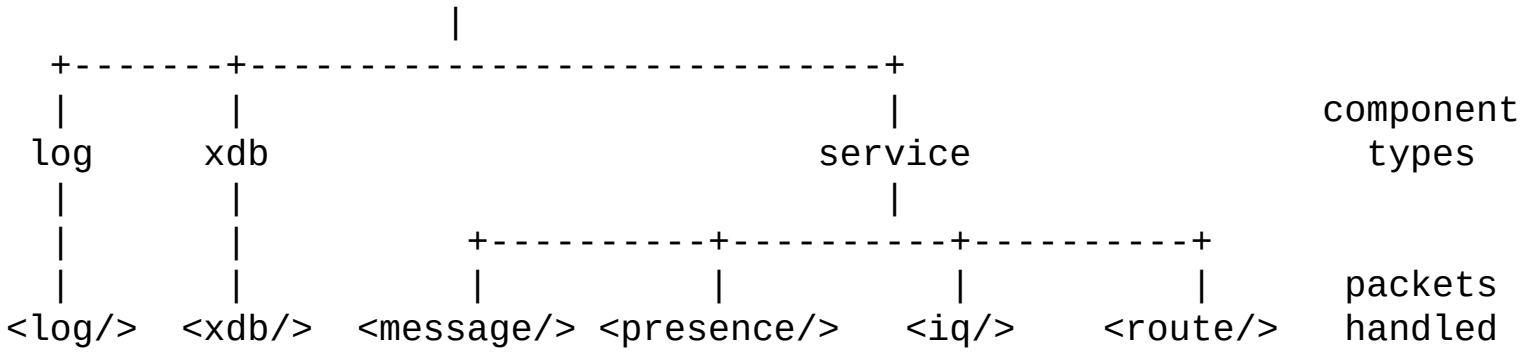
The second is a `<message/>` packet, which contains the message itself ("Hi there") as well as other information (a message event and a conversation thread identifier; these are examined in detail in Part II of the book).

It isn't necessarily the case, however, that *all* xdb components will handle *all* `<xdb/>` packets, or *all* service components will handle *all* `<presence/>` packets. The configuration, described later in this chapter, determines how the components announce themselves and state their readiness to receive and handle packets.

Delivery trees

The phrase "delivery tree" is often used in Jabber terminology to signify a component or components that handle certain types of packet. The path a packet makes as it *descends* the collection of decision branches that guide it to the component or components that will handle it. For example, an 'xdb type component' is sometimes referred to as an *xdb Delivery Tree*. Considering the division of components into different types that handle different packet types is perhaps easier to visualize as a tree, as shown in Figure 4-2.

Figure 4-2. The Jabber Delivery Tree



The Jabber Delivery Tree shows which component types can handle what sorts of packets in the Jabber world.

Component Connection Methods

The notion of components providing distinct services and being coordinated by a central mechanism (**jabberd**) suggests a certain amount of independence and individuality—a plug-in architecture—and that is what Jabber is. The components described earlier, and others too, are "plugged in" to the Jabber backbone according to the requirements of the server.

The idea is that once you have the basic services like *Session Management, Client (to Server) Connectivity*

and *Data Storage*, you plug in whatever you need to suit the server's requirements. For example if you need conferencing facilities, you can plug in the *Conferencing* component. If you need user directory facilities, you can plug in the *Jabber User Directory (JUD)* component. If you need a bridge to the Yahoo! Instant Messaging system, you can plug in the *Yahoo! Transport* component. [3] Indeed, you can write your own components to provide services that are not available off the shelf and plug those in too.

So, how are components 'plugged in' to the Jabber server backbone? [4] Well, there are three methods:

- Library load
- TCP sockets
- STDIO

Let's examine each one in turn.

library load

The core components of a Jabber Server providing IM services are connected using the *library load* method. This simply means that the component sources are compiled into shared object (`.so`) libraries and loaded into the main Jabber process (**jabberd**).

The components are written specially with the Jabber backbone in mind and contain standard registration routines that utilize functions in the core Jabber libraries. These routines are used to bind the component relationship with **jabberd** (for example there is a 'heartbeat' mechanism through which the components are monitored) and to specify packet receipt requirements. [5]

The *library load* method is represented in the configuration by the `<load/>` tag, which wraps the library (or libraries) that should be loaded. Example 4-4 and Example 4-5 show excerpts from the standard `jabber.xml` configuration file where we can see components being plugged in using the *library load* method.

Example 4-4 shows the *Client (to Server) Connections (c2s)* component, which has been written and compiled as a `.so` library, being connected using the *library load* method.

Example 4-4. Loading of the c2s component with *library load*

```
<load>
  <pthsock_client>./pthsock/pthsock_client.so</pthsock_client>
</load>
```

In this example, we see the "simpler" form of the `<load/>` tag: inside the tag we have:

```
<pthsock_client>./pthsock/pthsock_client.so</pthsock_client>
```

which specifies two things:

- Which library to load (`./pthsock/pthsock_client.so`).
- The name of the component registration routine that should be called by **jabberd** once the library has been loaded. The name of the routine is the name given to the tag that wraps the library filename; in this example it's `pthsock_client()`, denoted by `<pthsock_client/>`.

Example 4-5 shows multiple `.so` libraries being loaded when a component is connected; the form of the `<load/>` tag is slightly more involved.

Example 4-5. Loading of the jsm component with *library load*

```
<load main="jsm">
  <jsm>./jsm/jsm.so</jsm>
  <mod_echo>./jsm/jsm.so</mod_echo>
  <mod_roster>./jsm/jsm.so</mod_roster>
  <mod_time>./jsm/jsm.so</mod_time>
  <mod_vcard>./jsm/jsm.so</mod_vcard>
  <mod_last>./jsm/jsm.so</mod_last>
  <mod_version>./jsm/jsm.so</mod_version>
  <mod_announce>./jsm/jsm.so</mod_announce>
  <mod_agents>./jsm/jsm.so</mod_agents>
  <mod_browse>./jsm/jsm.so</mod_browse>
  <mod_admin>./jsm/jsm.so</mod_admin>
  <mod_filter>./jsm/jsm.so</mod_filter>
  <mod_offline>./jsm/jsm.so</mod_offline>
  <mod_presence>./jsm/jsm.so</mod_presence>
  <mod_auth_plain>./jsm/jsm.so</mod_auth_plain>
  <mod_auth_digest>./jsm/jsm.so</mod_auth_digest>
  <mod_auth_0k>./jsm/jsm.so</mod_auth_0k>
  <mod_log>./jsm/jsm.so</mod_log>
  <mod_register>./jsm/jsm.so</mod_register>
  <mod_xml>./jsm/jsm.so</mod_xml>
</load>
```

Here we see multiple libraries being loaded to form the *Session Management* (the JSM) component, known as "jsm".

What happens is this:

1. **jabberd** loads the library in the tag that's pointed to by the `main=""` attribute of the `<load/>` tag; in this example it's the library `./jsm/jsm.so`:

```
<jsm>./jsm/jsm.so</jsm>
```

2. **jabberd** then invokes the registration routine called `jsm()`
3. jsm loads the rest of the modules defined within the `<load/>` tag (`mod_echo`, `mod_roster`, `mod_time`, and so on), invoking each module's registration routine (`mod_echo()`, `mod_roster()`, `mod_time()`, and so on) as they're loaded.

In case you're wondering, all the modules that belong to the jsm are actually compiled into a single `.so` library - which is why all the `.so` references in this example are the same.

TCP sockets

Another method for connecting components to the Jabber backbone uses a TCP sockets connection. This means that a component connected in this way can reside on the same or a different server to the one running **jabberd**. So instead of being loaded directly into the **jabberd** backbone, TCP sockets-connected components exist and run as separate entities and can be started and stopped independently. [6]

The configuration syntax for defining a connection point for a component that is going to connect to the backbone via TCP sockets looks like this: [7]

```
<accept>
  <ip>127.0.0.1</ip>
  <port>9001</port>
  <secret>shhh</secret>
</accept>
```

As `<load/>` is to the *library load* method, so `<accept/>` is to the *TCP sockets* method.

The `<accept/>` tag usually has three child tags - `<ip/>`, `<port/>` and `<secret/>`. [8] Specify an IP address and port to which the component will connect; if you want the socket to be network interface independent, you can write `<ip/>` (an empty tag) to listen on your specified port on all (INADDR_ANY) IP addresses. The `<secret/>` tag is used in the handshake when the component connects to the backbone, so that it can be authenticated.

More information on connecting components with `<accept/>` can be found in Part II.

XML Streams

Entity to entity connections in the Jabber world pass data to each other using a technique called *XML streams*, which is essentially the exchange of data in the form of XML fragments in "streaming" mode over a network connection.

During the lifetime of a connection between two entities, two complete XML documents will be passed between the entities (if A connects to B, A will send B a document, fragment by fragment, and B will send A a document, fragment by fragment). At the start of the connection, the first things to be exchanged are the document headers—the outermost (document root) XML tags. The root tag in each document is `<stream/>`.

XML namespaces are defined for the content of the XML documents to be exchanged. These namespaces represent what sort of entity connection is taking place. In the case of a Jabber client to Jabber server connection, the namespace is `jabber:client`. In the case of connections based on the TCP sockets connection method, (using `<accept/>`), the namespace is `jabber:component:accept`.

To initiate a connection and conversation, the component will initiate the conversation by sending something like this to Jabber:

```
<?xml version="1.0"?>
<stream:stream xmlns:stream="http://etherx.jabber.org/streams"
               xmlns="jabber:component:accept"
               to="component.name">
```

to Jabber.

More detailed information on how XML streams work can be found in Chapter 5.

STDIO

The TCP sockets component connect method is used to connect an external component to the Jabber backbone via a socket connection through which streamed XML documents are exchanged. There is another way for components to connect and exchange XML document streams with the Jabber backbone—using the *STDIO* connection method.

While the *TCP sockets* method requires external components to be independently started and stopped, the *STDIO* method represents a mechanism whereby the **jabberd** process starts the external component itself. The component to start is specified inside an `<exec/>` tag. Example 4-6 shows how the *STDIO* method is

specified in the configuration.

Example 4-6. Invoking an external component with *STDIO*

```
<exec>/path/to/component.py -option a -option b</exec>
```

Here we see that the component is a Python program and is being passed some switches at startup.

So where's the socket connection in this method? There isn't one. The XML documents are exchanged through standard I/O (STDIO). The component writes XML fragments to STDOUT, and these are received on the Jabber backbone. The component receives XML fragments destined for it on STDIN, fragments which are written out from the Jabber backbone.

Just as a component connected using the *TCP sockets* method sends an opening document fragment, the component connected with this *STDIO* method sends an opening document fragment to initiate a connection and conversation:

```
<?xml version="1.0"?>
<stream:stream xmlns:stream="http://etherx.jabber.org/streams"
               xmlns="jabber:component:exec"
               to="component.name">
```

Notice how the namespace that describes this type of conversation is:

```
jabber:component:exec
```

No secret is required in this case because it is assumed that the component can be trusted if it is specified in the configuration and execution is initiated by **jabberd** itself.

Notes

[1] This identifier is a hexadecimal representation of the user's session id within the JSM, carried internally as a JID resource in the routing information.

[2] the '15' is the identifier for the socket on which the the pertinent client connection has been made

[3] Actually, you can also connect to a component running on another Jabber server - see later in this Chapter.

[4] The *log* component(s) are actually *part of* the backbone, and as such do not need to be 'plugged in'

[5] components receive packets based upon their type (log, xdb or service), upon a `<host/>` configuration specification that we'll see later in this chapter, and can also specify when in the delivery sequence they are to receive the packets

[6] It is actually possible to modify the contents of the configuration while the Jabber server is running, and then send a hangup (HUP) signal to the processes, which causes a re-read of the configuration and new *library load*ed components can be started and old ones can be stopped. There are however currently some problems with this, that will be addressed in the next round of bug fixes.

[7] The tag name for this TCP sockets configuration stanza is `<accept/>`, reflecting the low-level socket library call `accept()` to which it directly relates.

[8] There is a fourth tag `<timeout/>` with which you can control the heartbeat monitor of this component connection, which defaults to a value of 10 if not explicitly specified (it seldom is).

Prev: Server Architecture and Configuration
Home
Up
Next: Server Configuration

Server Configuration

At this stage, we should be fairly comfortable with the notion of a **jabberd** backbone and a set of components that combine to provide the features needed for a complete messaging system. We've looked at fragments of configuration in the previous section; now it's time to examine the configuration directives in more detail.

It's not uncommon for people installing a Jabber server for the first time to be daunted (I was terrified!) by the contents of the `jabber.xml` configuration file. But really, for the most part, it's just a collection of component descriptions—what those components are, how they're connected, what packets they are to process, and what their individual configuration is.

Component instances

There's a concept that encompasses Jabber's configuration approach that is taken from the object-oriented (OO) world—the concept of objects (and classes) and instances thereof. In Jabber server configuration, specifically the description of the components that are to make up a particular Jabber server, we talk about *instances* of components, not components directly.

In other words, a component is something generic that is written to provide a specific service or set of services; when we put that component to use in a Jabber server, we customize the characteristics of that component by specifying detailed configuration pertaining to how that component will *actually* work. We're creating an "instance" of that component.

A typical component instance description

Each component instance description follows the same approximate pattern:

- Declaration of the component type
- Identification (name) of the component
- Specification of the host filter for packet reception
- Definition of how the component is connected
- Custom configuration for the component

Of course, for any generalized rule, there's always an exception. The *log* component type, as mentioned

earlier in this chapter, is defined slightly differently —while there is a host filter defined, a component connection definition is not relevant nor present, and the custom configuration is limited— we'll see this later when we take a tour of the `jabber.xml`.

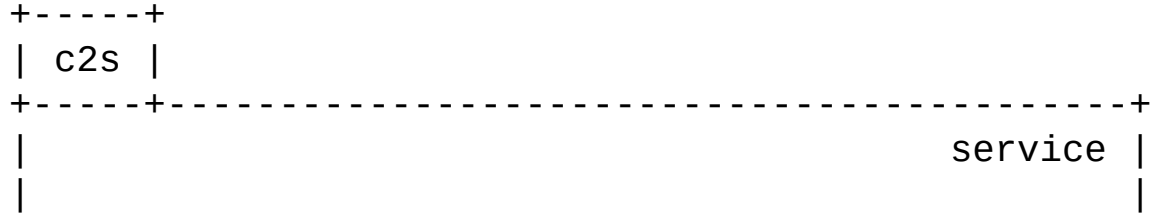
Let's have a closer look at the *Client (to Server) Connections* (c2s) component and how an instance of it is specified in the `jabber.xml`. We're going to use the one which comes delivered in the Jabber 1.4.1 server distribution tarball. Example 4-7 shows how the c2s is defined. The definition includes details of how the component code is connected (using the *library load* method), and contains some custom configuration covering authentication timeout (the `<authtime/>` tag), traffic flow control (the `<karma/>` section), and what port c2s is to listen on (the `<ip/>` tag). We'll look at these custom configuration tags in detail later.

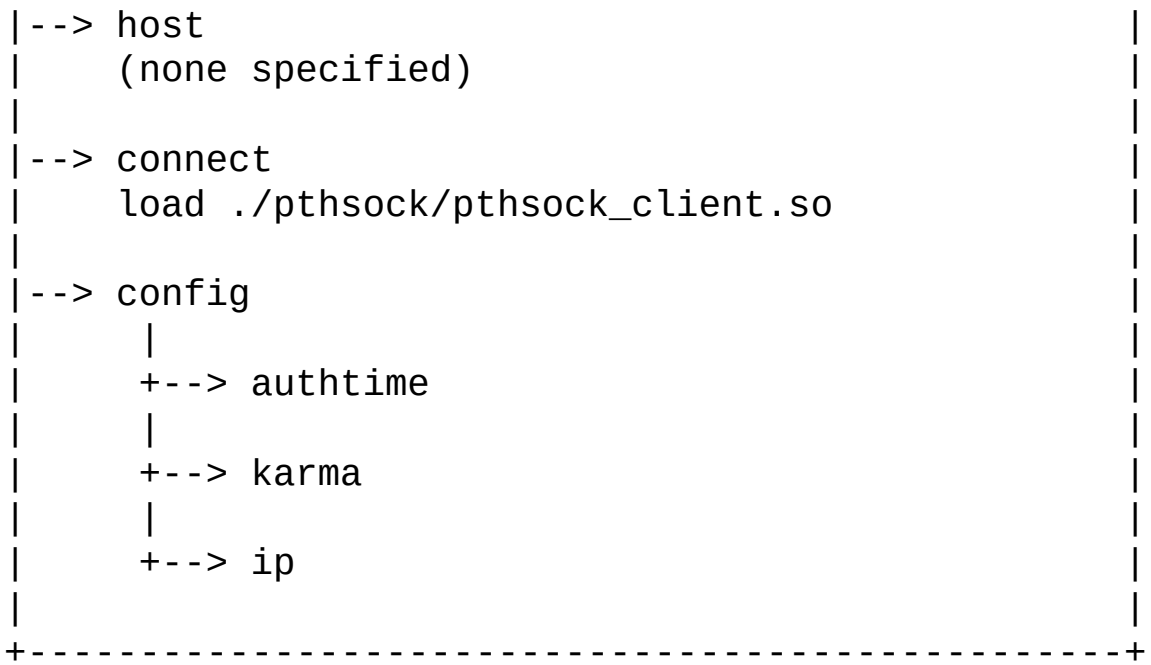
Example 4-7. The `c2s` instance configuration in `jabber.xml`

```
<service id="c2s">
  <load>
    <pthsock_client>./pthsock/pthsock_client.so</pthsock_client>
  </load>
  <pthcsock xmlns='jabber:config:pth-csock'>
    <authtime/>
    <karma>
      <init>10</init>
      <max>10</max>
      <inc>1</inc>
      <dec>1</dec>
      <penalty>-6</penalty>
      <restore>10</restore>
    </karma>
    <ip port="5222"/>
  </pthcsock>
</service>
```

Now let's arrange this instance configuration in diagram form. Figure 4-3 highlights the pattern we're expecting to see.

Figure 4-3. A diagram of the `c2s` instance configuration in `jabber.xml`





If we look at the component instance descriptions in this way, it's easy to understand how the configuration is put together, and we can begin to see the pattern emerging. Taking each of the elements of the pattern in turn let's examine what the XML tells us.

Component type

The component type is *service*. We know that from looking at the outermost tag in the XML:

```
<service id="c2s">
  ...
</service>
```

So we know that this component instance will handle `<message/>`, `<presence/>`, `<iq/>` and `<route/>` packets.

Identification

Each component instance must be uniquely identified within the space of a single Jabber server (configuration). **jabberd** uses this identification to address the components and deliver packets to the right place. In this case, the identification of this component instance is *c2s*; it's taken from the `id` attribute of the component type tag:

```
<service id="c2s">
```

Host filter

The diagram shown in Figure 4-3 states *"none specified."* So what happens now? Well, a host filter is usually one or more `<host/>` tags containing hostnames to which the component instance will "answer". It's a way of specifying that packets destined for a certain hostname will be received by that component instance.

However, if there are no `<host/>` tags specified as in this c2s example, then the component instance's identification is taken as the hostname specification. In other words, the `<service id="c2s">` declaration in this example, coupled with the lack of any explicit `<host/>` tag, *implies* a host filter of "c2s". This component instance wants to receive all packets with addresses that have "c2s" as the hostname. It's the equivalent of this host filter specification:

```
<host>c2s</host>
```

Packets and Hostnames

Each packet travelling within the Jabber universe has a destination address, which at the very least contains a hostname specification. This hostname can be a "real" hostname in the DNS sense, or it can be a Jabber-internal hostname, such as '`jud.yak`'.

When you send a packet, such as a `<message/>` element from your Jabber client, you're sending it as an XML fragment within the context of the `jabber:client` XML stream running between your client and the Jabber server. Apart from the initial activity to connect to the Jabber server's listening socket, there's no hostname resolution on the client's part—the hostname in the recipient address is just specified as an attribute (`to`) of the `<message/>` tag *inside* the stream.

Once a packet reaches the Jabber server, **jabberd** will try to deliver it to its destination. In doing so it will check an internal list of component instances derived from the configuration file specified at startup, and work out which instance (or instances; more than one may have registered to receive packets destined for the same hostname) should receive it.

If no matching instance can be found (i.e., there are no instances filtering for the hostname specified in the packet's destination address) then the Jabber server will assume it's for a non-local destination and attempt to resolve the hostname into an IP address using the *Hostname Resolution* component and send it on using the *Server (to Server) Connections* component.

The `<host/>` tag

There is some degree of flexibility in how you specify a hostname in the `<host/>` tag.

You can specify an absolute hostname like this:

```
<host>conference.yak</host>
```

You can specify more than one hostname like this:

```
<host>conference.yak</host>
<host>talk.yak</host>
```

For example, if this pair of `<host/>` tags appeared in an instance specification for the *Conferencing* component, you could address the component instance using either hostname.

You can use a wildcard character to specify all hostnames within a domain, for example:

```
<host>*.pipetree.com</host>
```

will match on all hosts with the domain name `pipetree.com`.

If you want the component instance to receive packets regardless of the hostname, you can specify an empty tag thus:

```
<host/>
```

Component connection method

What is the component? Where do we load it from, or how does it connect to the Jabber backbone? There is a *component connection method* (see the section called *Component Connection Methods* earlier in this Chapter) specified in each of the component instance definitions. In our example of the c2s component instance, we see that the *library load* method is being used to load the `./pthsock/pthsock_client.so` shared object library, and that the component registration routine `pthsock_client()` should be called once loading is complete:

```
<load>
  <pthsock_client>./pthsock/pthsock_client.so</pthsock_client>
</load>
```

Custom configuration

Once we've dealt with the (optional or implied) `<host/>` tag hostname filters, and the component connection method, all that is left is the custom configuration for the component instance itself. This will look different for different components, but there is still a pattern that you can recognize. The

configuration always appears in a "wrapper" tag that, like the <host/> and <load/> tags above, appears as an immediate child of the component type tag (that's <service/> in our c2s example):

```
<service id="c2s">
  ...
  <pthcsock xmlns='jabber:config:pth-csock'>
     ... [configuration here] ...
  </pthcsock>
</service>
```

There are two things to note here:

- The tag name (<pthcsock/>)
- The namespace declaration (xmlns='jabber:config:pth-csock')

To put it bluntly, one is important, the other isn't. The name of the tag enclosing the instance's configuration—"pthcsock" is not important. There must be a name (otherwise it wouldn't be valid XML) but it might as well be "banana" for all the effect it would have. So as it might as well be "banana," it might as well be something meaningful—hence "pthcsock."

The important part of the configuration wrapper tag is the namespace declaration:

```
xmlns="jabber:config:pth-csock"
```

because that is what the component actually uses to search for and retrieve the configuration.

As for the actual configuration elements for the c2s component instance that we see here (<authtime/> and <karma/>), we'll take a look at them in the next section.

A Tour of jabber.xml

Now we know what patterns to look out for, we're well prepared to dive into a jabber.xml configuration file. As an example, we'll take one that's very similar to the default jabber.xml installed with version 1.4.1 of Jabber, but we'll plug in some extra components: the Conferencing component and a local JUD component.

The entire configuration content, with comment lines dividing up each section, can be found in Appendix A. It's definitely worth turning briefly to have a look at the XML before continuing, to get a feel for how the configuration is laid out.

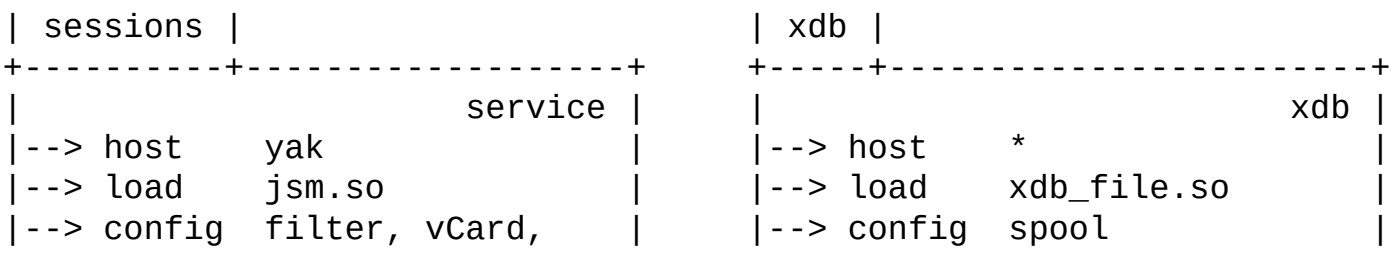
In order to deal with it without going crazy, let's break down the XML into manageable chunks. We'll build configuration diagrams for each of the top-level tags that are children of the root tag <jabber/>. The opening tags for each of these chunks are as follows:

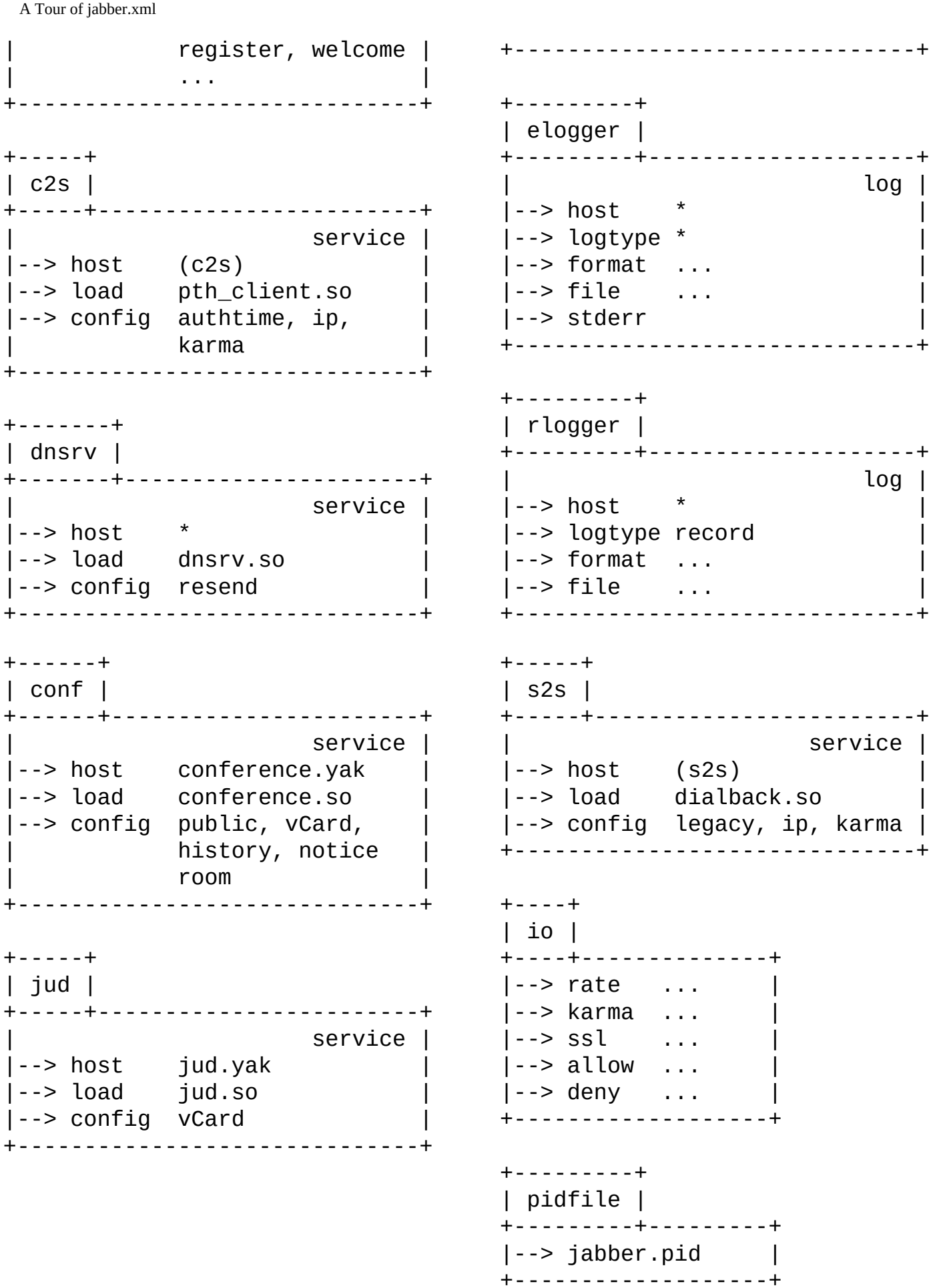
- <service id="sessions">
- <xdb id="xdb">
- <service id="c2s">
- <log id="elogger">
- <log id="rlogger">
- <service id="dnsrv">
- <service id="s2s">
- <service id="conf">
- <io>
- <pidfile>

Most of these should be recognisable by now, but there are two chunks that we haven't come across yet: <io> and <pidfile>. These aren't components but nevertheless are part of the configuration for **jabberd**; there are also the two *Logging* component instances that we have not paid much attention to until now.

Figure 4-4 provides an overview of how the Jabber server is configured. It represents the contents of the jabber.xml configuration file in Appendix A in diagram form.

Figure 4-4. Configuration file in diagram form




register, welcome
...
c2s
service
host (c2s)
load pth_client.so
config authtime, ip, karma
dnsrv
service
host *
load dnsrv.so
config resend
conf
service
host conference.yak
load conference.so
config public, vCard, history, notice room
jud
service
host jud.yak
load jud.so
config vCard
elogger
log
host *
logtype *
format ...
file ...
stderr
rlogger
log
host *
logtype record
format ...
file ...
s2s
service
host (s2s)
load dialback.so
config legacy, ip, karma
io
rate ...
karma ...
ssl ...
allow ...
deny ...
pidfile
jabber.pid

We can see that the bulk of the Jabber server functionality described here is in the form of components. Let's take each of these components one by one and have a closer look.

Component instance: *sessions*

The *sessions* component, described by the configuration XML shown in Example 4-8 and shown in diagram form in Figure 4-5, provides Session Management features for users (the word "users" is employed in the widest possible sense—a user could be a person or a script) connecting with Jabber clients - through XML streams identified with the `jabber:client` stream namespace.

Figure 4-5. Diagram view of *sessions* component instance

```
+----------+
| sessions |
+----------+--------------------+
|                       service |
|--> host    yak                |
|--> load    jsm.so             |
|--> config  filter, vCard,     |
|            register, welcome  |
|            ...                |
+-------------------------------+
```

It also provides the services that give Jabber its IM capabilities— services such as roster management, message filtering, store-and-forward ("offline") message handling, and so on. These IM services are loaded individually as part of the component connection phase.

Example 4-8. `jabber.xml` configuration for the *sessions* component instance

```
<service id="sessions">

  <host><jabberd:cmdline flag="h">yak</jabberd:cmdline></host>

  <jsm xmlns="jabber:config:jsm">
    <filter>
        <default/>
        <max_size>100</max_size>
        <allow>
            <conditions>
                <ns/>
                <unavailable/>
                <from/>
                <resource/>
                <subject/>
                <body/>
                <show/>
```

```
            <type/>
            <roster/>
            <group/>
        </conditions>
        <actions>
            <error/>
            <offline/>
            <forward/>
            <reply/>
            <continue/>
            <settype/>
        </actions>
    </allow>
</filter>
<vCard>
  <FN>Jabber Server on yak</FN>
  <DESC>A Jabber Server!</DESC>
  <URL>http://yak/</URL>
</vCard>
<register notify="yes">
  <instructions>Choose a userid and password to register.</instructions>
  <name/>
  <email/>
</register>
<welcome>
  <subject>Welcome!</subject>
  <body>Welcome to the Jabber server on yak</body>
</welcome>
<!--
<admin>
  <read>support@yak</read>
  <write>admin@yak</write>
  <reply>
    <subject>Auto Reply</subject>
    <body>This is a special administrative address.</body>
  </reply>
</admin>
-->
<update><jabberd:cmdline flag="h">yak</jabberd:cmdline></update>
<vcard2jud/>
<browse>
  <service type="jud" jid="jud.yak" name="yak User Directory">
    <ns>jabber:iq:search</ns>
    <ns>jabber:iq:register</ns>
  </service>
  <conference type="public" jid="conference.yak" name="yak Conferencing"/>
</browse>
```

```
  </jsm>

  <load main="jsm">
    <jsm>./jsm/jsm.so</jsm>
    <mod_echo>./jsm/jsm.so</mod_echo>
    <mod_roster>./jsm/jsm.so</mod_roster>
    <mod_time>./jsm/jsm.so</mod_time>
    <mod_vcard>./jsm/jsm.so</mod_vcard>
    <mod_last>./jsm/jsm.so</mod_last>
    <mod_version>./jsm/jsm.so</mod_version>
    <mod_announce>./jsm/jsm.so</mod_announce>
    <mod_agents>./jsm/jsm.so</mod_agents>
    <mod_browse>./jsm/jsm.so</mod_browse>
    <mod_admin>./jsm/jsm.so</mod_admin>
    <mod_filter>./jsm/jsm.so</mod_filter>
    <mod_offline>./jsm/jsm.so</mod_offline>
    <mod_presence>./jsm/jsm.so</mod_presence>
    <mod_auth_plain>./jsm/jsm.so</mod_auth_plain>
    <mod_auth_digest>./jsm/jsm.so</mod_auth_digest>
    <mod_auth_0k>./jsm/jsm.so</mod_auth_0k>
    <mod_log>./jsm/jsm.so</mod_log>
    <mod_register>./jsm/jsm.so</mod_register>
    <mod_xml>./jsm/jsm.so</mod_xml>
  </load>

</service>
```

Component type and identification

The opening tag:

```
<service id="sessions">
```

identifies this component instance to the backbone as a service type component and gives it a name ("sessions") that can be used for internal addressing and to distinguish it from other component instances.

Host filter

Assuming that our hostname isn't "sessions," it's just as well that we have a `<host/>` specification in this component instance description:

```
<host><jabberd:cmdline flag="h">yak</jabberd:cmdline></host>
```

which means that this session management component instance will handle packets addressed to the host "yak." [1] The `<jabberd:cmdline flag="h"> ... </jabberd:cmdline>` wrapper around the hostname means that this value ("yak") can be overridden by specifying a switch `-h <hostname>` when **jabberd** is invoked, as is described in Chapter 3. If you're sure you'll never want to override the hostname setting here, this

`<jabberd:cmdline/>` wrapper can safely be removed from the configuration, to leave:

```
<host>yak</host>
```

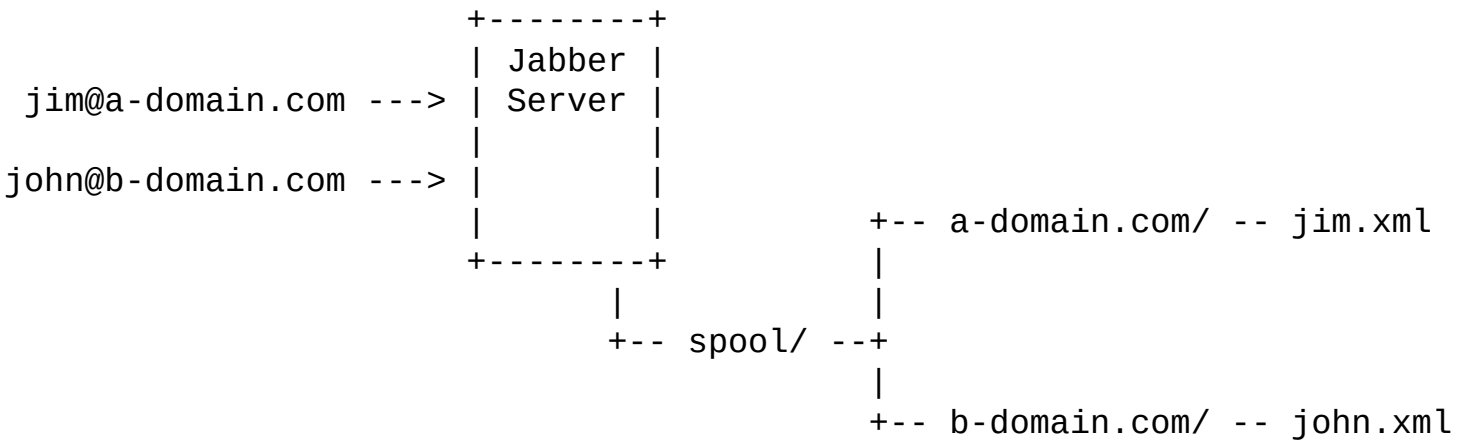
As described earlier, you can specify more than one hostname - use a `<host>...</host>` pair for each one. This will effectively give you a virtual server effect where Jabber will respond to different hostnames. This is useful in situations such as deployment in an ISP where a single host serves multiple domains. The client data stored on the server (such as rosters, offline messages, and so on) is stored by the xdb component by hostname, so that a separate directory in the spool area will be used for each specified hostname.

For example, if you specified the two hosts:

```
<host>a-domain.com</host>
<host>b-domain.com</host>
```

then the data for two users *jim@a-domain.com* and *john@b-domain.com* would be stored as shown in Figure 4-6.

Figure 4-6. Storage of server-side user data by hostname



Although specifying multiple hostnames for the session management component instance will effect a sort of virtual hosting, with separate data storage as described, the rest of the features of the component will be identical. For example, this means that the list of available services that the client can request—the *agent list* (old terminology) or *browse list* (new terminology)—and the session features such as roster management, administration functions, private data storage and so on, will be identical. If you want to offer different services for different hostnames from the same Jabber server, see the section called *Managing the Configuration* later in this chapter.

Hostname needed here
You cannot use the catchall empty `<host/>` tag for the session management component—it needs to be given an explicit host identity in order to function.

Custom configuration

In the section called *A typical component instance description* earlier, we described the elements in this order:

component type, *component identification*, *host filter*, *connection method*, *custom configuration*. Being XML, the configuration format is flexible enough to allow us to manage the ordering (but not the nesting!) of the configuration directives to suit our own layout purposes. In this instance, we come to the *custom configuration*—the *connection method* comes afterwards.

The *sessions* component (i.e. the JSM) offers a lot of facilities, which means that in order to attach an instance of the JSM into our Jabber server we have a lot of configuring to do.

Our configuration wrapper tag for the JSM instance is

```
<jsm xmlns="jabber:config:jsm">
```

The tag name "`jsm`" is simply representative of what the configuration pertains to; Once loaded, the JSM component will look for the configuration by the namespace identifier `jabber:config:jsm`. Within the wrapper tag we have different sections that approximately relate to the different *services* that the jsm is going to provide.

Services and modules

the section called *Component Connection Method* describes in detail how these jsm services are loaded. Generally, each service is represented by a tag that specifies a library to be loaded, and the tag name is what is used to refer to the service, thus giving it a name.

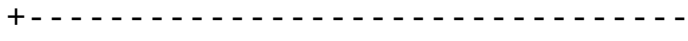
For example, the *echo* service, which is used to test the Jabber server (any message sent to a special echo address `servername/echo`, which is the servername with a resource of "echo", will be echoed back to the sender) is loaded in the `<mod_echo/>` tag and is referred to as the *`mod_echo` module*. So it can be said that the `mod_echo` *module* provides the echo *service*.

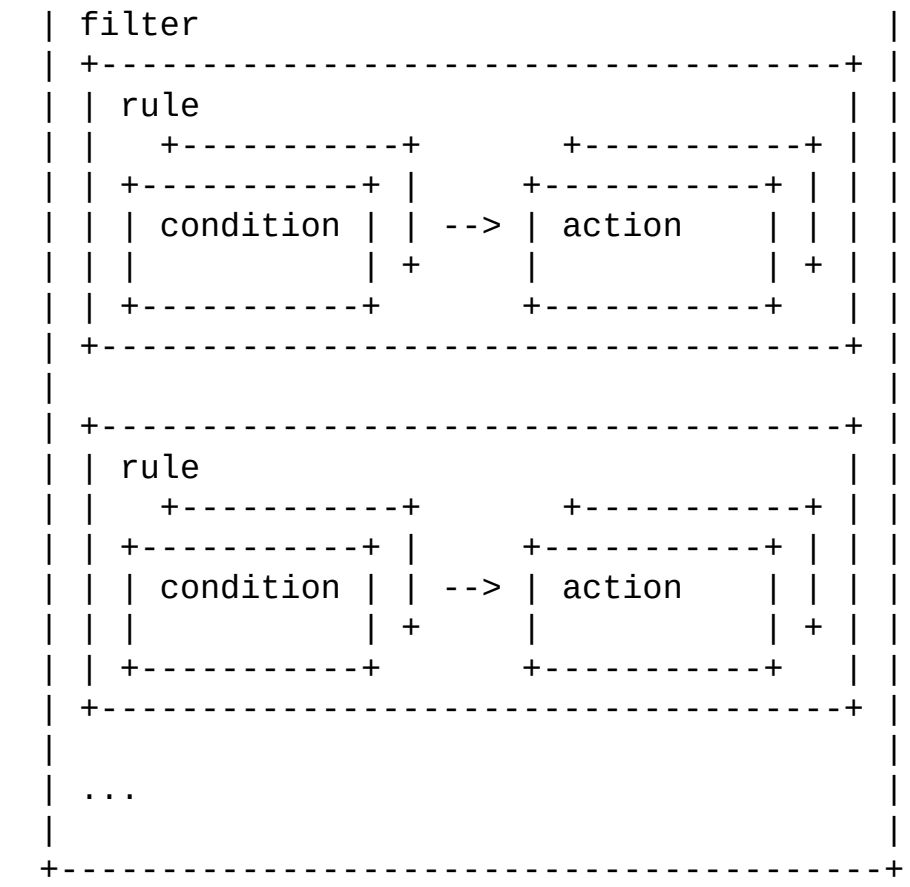
Filter Service

The message filter service, provided by the `mod_filter` module allows clients to set up mechanisms that can control and manage incoming messages as they arrive at the recipient's Jabber server—before they start on the final leg of the journey to the recipient's client.

The service allows each user to maintain their own filter, which is a collection of *rules*. A rule is a combination of conditions and actions. For each incoming message, the message filter service kicks in and goes through the rules contained in the message recipient's filter one by one, checking the characteristics of the incoming message using the conditions defined in each rule. If one of the conditions matches, then the action or actions defined in that rule are carried out and the message filter service stops going through the rules—unless the action specified is 'continue'—in which case the service goes on to the next rule. The 'continue' action makes it possible to chain together a complex series of checks and actions.

Figure 4-7. A message filter





Each user's filter is stored on the server using the *xdb* component (see later). What does a typical filter look like? Well, Example 4-9 shows a filter that contains two rules -

- Rule "holiday" checks the message recipient's presence and sends a 'holiday' notice back if the presence is set to 'Extended Away', and forwards the incoming message to a colleague.
- Rule "custreply" checks to see if the message is from someone that exists in certain groups in the recipient's roster and if so sends an auto-reply to that person, sets the incoming message type to 'normal' (in case it was a 'chat' message), and allows the message to reach its original intended destination.

  This could be useful in a customer support scenario where the support representative could handle incoming queries in a queue of 'normal' messages but have an auto-reply sent out for each query telling the customer that their request will be dealt with shortly.

Example 4-9. A message filter with two rules

```
<query xmlns="jabber:iq:filter">
  <rule name="holiday">
    <show>xa</show>
    <reply>I'm on holiday - back on the 25th!</reply>
    <forward>mycolleague@yak</forward>
  </rule>
  <rule name="custreply">
    <group>CustomersNorth</group>
    <group>CustomersSouth</group>
```

```
    <reply>Thanks - an operator will attend to you shortly</reply>
    <continue/>
  </rule>
</query>
```

Note that there is no nesting or grouping to distinguish conditions from actions. In the first rule - "holiday" - there is one condition (`<show/>`) and two actions (`<reply/>` and `<forward/>`) and in the second rule - "custreply" - there are two conditions (two `<group/>`s) and two actions (`<reply/>` and `<continue/>`).

There are a few things to note from this example:

The `<continue/>` action means that the filter checking will move on to the 'next rule' which doesn't exist, meaning that the original message will still be delivered. No `<continue/>` would have meant that the message would have been dropped (that is, it wouldn't have reached it's final original destination) - because when a rule matches the actions in that rule are carried out and a successful delivery is implied.

The conditions are ORed together, that is to say if *any* of the conditions in a rule match then the rule has matched and all of actions defined in the rule are carried out.

So with this in mind, let's examine the message filter service configuration:

```
<filter>
    <default/>
    <max_size>100</max_size>
    <allow>
        <conditions>
            <ns/>
            <unavailable/>
            <from/>
            <resource/>
            <subject/>
            <body/>
            <show/>
            <type/>
            <roster/>
            <group/>
        </conditions>
        <actions>
            <error/>
            <offline/>
            <forward/>
            <reply/>
            <continue/>
            <settype/>
        </actions>
    </allow>
</filter>
```

Within the <filter/> configuration wrapper, we have three children:

<default/>

The <default/> tag allows the server administrator to specify default filter rules that will be applied for every user registered on that Jabber server. Specifying something like this:

```
<default>
  <rule name="server wide rule">
    <from>spammer@spamcity.com</from>
    <error>No spam please, we're British!</error>
  </rule>
</default>
```

will effectively filter out all messages from our friendly spammer.

The rules specified in the <default/> tag will be *appended* to any personal rules the user may have defined himself. This is important when you consider the order in which the rules are tested, and that once a rule is matched, filter processing stops (unless the <continue/> action is used).

<maxsize/>

Filter rule matching is expensive. We don't want to allow the user to go overboard with filter rules - we can place an upper limit on the number of rules in a filter with the <maxsize/> tag. (The default is rather large - anyone who can be bothered to create 100 rules deserves to have them all checked, in my opinion!)

<allow/>

The <allow/> tag specifies the <conditions/> and <actions/> that a user is allowed to use in building rules. Table 4-1 and Table 4-2 show the possible filter conditions and actions.

Table 4-1. Filter conditions

Condition	Example	Description
<ns/>	<ns>jabber:iq:version</ns>	Matches the namespace ('ns') of an <iq/> packet. [a]
<unavailable/>	<unavailable/>	Matches when the recipient's presence type is 'unavailable'.
<from/>	<from>spammer@spamcity.com</from>	Matches the sender's Jabber ID (JID) - user@host.
<resource/>	<resource>Work</resource>	Matches the recipient's resource.
<subject/>	<subject>Work(!)</subject>	Matches the message's subject (in the <subject/> tag) - must match exactly.

`<body/>`	`<body>Are you there?</body>`	Matches the message content (in the `<body/>` tag) - must match exactly.
`<show/>`	`<show>dnd</show>`	Matches the recipient's presence 'show' - usually one of *normal* (the default), *chat*, *away*, *xa* (eXtended Away) or *dnd* (Do Not Disturb).
`<type/>`	`<type>chat</type>`	Matches the type of the incoming message (in the `type=""` attribute) - could be one of *normal*, *chat*, *headline* or *error*.
`<roster/>`	`<roster/>`	Matches if the sender is in the recipient's roster.
`<group/>`	`<group>Friends</group>`	Matches if the sender is in a particular group in the recipient's roster.
Notes: a. The name "message filter service" is slightly inaccurate as incoming `<iq/>` (Info/Query) packets can also be filtered - and the matching takes place on the namespace described in the `xmlns=""` attribute.		

Table 4-2. Filter actions

Action	Example	Description
`<error/>`	`<error>Address defunct</error>`	Sends an error reply to the sender.
`<offline/>`	`<offline/>`	Stores the incoming message offline. The recipient will receive it the next time he logs on.
`<forward/>`	`<forward>colleague@server</forward>`	The message will be forwarded to another Jabber ID (JID).
`<reply/>`	`<reply>Be right back!</reply>`	A reply will be sent to the sender.
`<settype/>`	`<settype>normal</settype>`	Changes the type of the incoming message (see `<type/>` in the Conditions table above).
`<continue/>`	`<continue/>`	Special action to continue on to the next rule.

Server vCard

Every user can maintain a virtual business card - a vCard - which is stored server-side. vCards can be retrieved at any time by any user. The `<vCard/>` tag here in the jsm configuration gives the Jabber server an identity - its vCard can be retrieved also.

You can maintain the server's vCard data in this part of the jsm configuration:

```
<vCard>
```

```
  <FN>Jabber Server on yak</FN>
  <DESC>A Jabber Server!</DESC>
  <URL>http://yak/</URL>
</vCard>
```

All the vCard elements can be used for this vCard configuration, not just the ones shown here.

Registration Instructions

Registration instructions such as those defined here:

```
<register notify="yes">
  <instructions>Choose a userid and password to register.</instructions>
  <name/>
  <email/>
</register>
```

are available to whoever asks for them; in its most formal state, the procedure for creating a new user account on a Jabber server includes a first step of asking the server what is required for the registration process.

The registration service is provided by the `mod_register` module.

In reply to such a request (which is made with an `<iq/>` get request in the `jabber:iq:register` namespace - see Part II for details) the instructions and a list of required fields are returned by `mod_register`. Note that the list of fields provided in this `<register/>` section are over and above the standard required fields required in any case for registration:

- `<username/>`
- `<password/>`

so that in this particular configuration case both `<name/>` and `<email/>` *and* `<username/>` and `<password/>` will be sent in the reply. The text inside the `<instructions/>` tag is intended for display by the client if it supports such a dynamic process. Typically the client would request the registration requirements and build a screen asking the user to enter values for the required fields, while displaying the instructions received.

The `notify="yes"` attribute of the `<register/>` tag will cause a message to be automatically created and sent to the server administrator address(es) for every new account created. See the section called *Administration* for details about specifying administration addresses.

If you want to prevent registration of new accounts on your Jabber server, comment out this `<register/>` section. `mod_register`, the only standard module that handles `<iq/>` packets in the `jabber:iq:register` namespace, will refuse to handle register requests if there is no `<register/>` section in the configuration, and so a "Not Implemented" reply will be sent to the registration details request.

Welcome Message

The welcome message defined here:

```
<welcome>
  <subject>Welcome!</subject>
  <body>Welcome to the Jabber server on yak</body>
</welcome>
```

will be sent to all new users the first time they log on. The `<subject/>` and `<body/>` contents are simply placed in a normal `<message/>` and sent off to the new Jabber ID (JID).

Administration

While the Unix user acts as the overall administrator for the Jabber server (for starting and stopping **jabberd**, for example) it is possible to specify administration rights for certain Jabber users that are local to the server. 'Local' means users that are defined as belonging to the host (or hosts) specified in the `<host/>` tag within the same jsm component instance definition - if the host tag is

```
<host>server.com</host>
```

then the JIDs `dj@server.com` and `admin@server.com` are local, but `admin@anotherserver.com` is not.

The only difference between an administration JID and a 'normal' JID is that the former is specified in tags in this section and the latter isn't. When a JID is specified between either the `<read/>` or `<write/>` tags, then it can be used to perform 'administrative' tasks.

The `<admin/>` section as delivered in the standard `jabber.xml` that comes with version 1.4.1 (see Appendix A) is commented out. Make sure that you remove the comment lines to activate the section if you want to make use of the administrative features:

```
<admin>
  <read>support@yak</read>
  <write>admin@yak</write>
  <reply>
    <subject>Auto Reply</subject>
    <body>This is a special administrative address.</body>
  </reply>
</admin>
```

If you want to specify more than one JID with administrative rights, simply repeat the tags, like this:

```
<read>admin1@yak</read>
<read>admin2@yak</read>
<read>admin3@yak</read>
```

Placing a JID inside of a `<write/>` tag implies that that JID also has `<read/>` administration rights. So there's not much point in doing something like this:

```
<read>admin@yak</read>
<write>admin@yak</write>
```

So what are the administrative features available to JIDs placed inside the `<read/>` and `<write/>` tags?

Administrative Features for `<read/>` JIDs

For JIDs appearing in a `<read/>` tag in the `<admin/>` section, these are the features available:

- *Retrieve list of users currently online*

  By sending one of two possible types of query to the server, a JID can retrieve a list of users that currently have a session on the (local) Jabber server. The results come in one of two forms, depending on the query type. The first query version is of the 'legacy' `iq:admin` type and the second is of the newer `iq:browse` type.

  The list of users in both sorts of results contain the user JID, for how long the user has been logged on (measured in seconds), how many packets have been sent from the user's session, and how many packets have been send to the user's session. The first query version also contains presence information for each user in the list.
- *Receipt of 'administrative queries'*

  Users normally send messages to other users - to other JIDs, where a JID is composed of a username and a hostname (a Jabber servername). The Jabber server itself is also a valid recipient, and the 'JID' in this case is just the servername itself - no username and no @ sign. [2]

  If a user sends a message to the server, it will be forwarded to the JIDs listed in the `<read/>` (and `<write/>`) tags in this `<admin/>` section, and the reply defined in the `<reply/>` tag will be sent back to the user as an automated response.

For JIDs appearing in a `<write/>` tag in the `<admin/>` section, these are the features available:

- *Same as `<read/>`*

  JIDs listed in `<write/>` tags automatically have access to the same features as those JIDs listed in `<read/>` tags.
- *Configuration retrieval*

  In a similar way to how a list of online users can be requested by sending a query of the `iq:admin` variety, a copy of the jsm configuration can be requested by sending an `iq:admin` query to the server. The difference is that in the former user list request, a request tag `<who/>` is sent inside the query, and in this configuration request, a `<config/>` tag is sent.

  The configuration XML, as it is defined in the jsm component instance section of the Jabber server being queried, is returned as a result.
- *Sending administrative messages*

Two types of administrative messages can be sent - an announcement to all online users, and a 'message of the day' (MOTD). The announcement goes out to all users currently online. Similarly, the MOTD goes out to all users, but not only those online - when someone logs on and starts a session the MOTD will be sent to them too, unlike the announcement, which will 'expire' as soon as it is sent. The MOTD will not expire, unless explicitly made to do so. The MOTD can also be updated - those that had already received the MOTD won't receive the updated copy during their current session, but anyone logging on after the update will receive the new version of the message.

Update Info Request

The `mod_version` module provides a simple service that, at server startup, queries a central repository of Jabber software version information at `update.jabber.org`. The `<update/>` configuration tag:

```
<update><jabberd:cmdline flag="h">yak</jabberd:cmdline></update>
```

is used to control this query.

If the `<update/>` tag is present, the query is sent. If the update tag is not present, the query is not sent.

If you do intend leaving the `<update/>` tag in, you need to make sure

1. the hostname specified as the value in the tag is resolvable and reachable as this is your Jabber server address to which the central repository will try to send back information (if there happens to be a newer version of the server software - specifically the jsm component - available)
2. your Jabber server is connected to the Internet to be able to reach `update.jabber.org`. You also need to be running instances of the *Hostname Resolution* and *Server (to Server) Connections* components so that your Jabber server can resolve the `update.jabber.org` host and send the query out.

The jsm component version releases are fortunately not so frequent that you require an automated mechanism to keep up with what's new; also you may wish to run an internal Jabber server with no connection to the outside world. So it is not uncommon for this section to be commented out. The jsm will still function without this piece of configuration.

It is worth noting here, however, that Jabber clients also use the central repository to find out about newer versions of themselves. As all Jabber client communication goes through the server you need to realise that commenting out the `<update/>` tag will not stop clients sending their queries. [3]

Auto-Update of JUD

The Jabber User Directory (JUD) is a service that provides a directory service of user names and addresses. The service comes in the form of a component - we'll be looking at the component instance definition of a JUD later in this chapter. If a Jabber server is running a JUD service, then you can connect to it with your Jabber client and enter your name and address details and query it as you would any directory service to find details of other people.

At the same time, each user has the possibility of maintaining his own vCard - we discussed vCards earlier in the section called *Server vCard*. In the same way that the server's vCard can be requested and retrieved, you can request a user's vCard, and the user whose vCard is requested does not have to be connected at that moment for the request to

be fulfilled - the vCards are stored server-side and the Jabber server handles the request (not the user's client).

So in many ways it makes sense to align the data in the user vCard with data stored in a JUD. The

`<vcard2jud/>`

configuration tag allows this alignment to happen automagically; if it appears in the configuration, it will cause any vCard updates (that would be typically performed by users changing their personal information via their Jabber clients) to be not only stored server-side in the vCard but also to be passed on to a JUD.

Which JUD? Well, the first one that's defined in the `<browse/>` section of the configuration, which is described next. Effectively it means that if you run a local JUD but also connect to the JUD running on `jabber.org`, you can choose which JUD will be the recipient of the vCard updates by placing that one before any others in the `<browse/>` list. [4]

If you're not running a JUD locally, or you simply don't want your users' vCard updates going to a JUD, you can safely comment this tag out.

Browsable Service Information

As the Jabber server administrators, *we* know what services are available on our Jabber server - what components are connected and what features they offer. We know for example that we're running a JUD locally, and have a *Conferencing* component.

But how do we let the Jabber clients know? If they're to be able to provide their users with an agreeable experience and expose them to all the server features available, we need some way to allow them to request information about what the server that they're connected to offers. Jabber has a powerful feature called 'browsing' which allows one entity to query another entity for information. Browsing defines a simple request/response exchange and with that provides a singular and uniform way to retrieve (on the requestors part) and expose (on the requestees part) feature information and availability.

Bearing that in mind, we can guess what the `<browse/>` section of the jsm custom configuration is for:

```
<browse>
  <service type="jud" jid="jud.yak" name="yak User Directory">
    <ns>jabber:iq:search</ns>
    <ns>jabber:iq:register</ns>
  </service>
  <conference type="public" jid="conference.yak" name="yak Conferencing"/>
</browse>
```

Each child of the `<browse/>` tag defines a feature - in this case a 'service' - that the Jabber server offers. Of course, these services are the ones over and above the services provided by the basic components such as *Session Management*, *Hostname Resolution* and so on.

So we have two services defined ('exposed') in the `<browse/>` configuration.

a local JUD:

```
<service type="jud" jid="jud.yak" name="yak User Directory">
  <ns>jabber:iq:search</ns>
  <ns>jabber:iq:register</ns>
</service>
```

and a conferencing service:

```
<conference type="public" jid="conference.yak" name="yak Conferencing"/>
```

The browsing features are covered in Part II, but briefly we can see here that each browsable 'item' is identified by a JID (`jid="jud.yak"` and `jid="conference.yak"`) and is classified using a category which is the combination of the item's outermost tag and the value of the tag's type attribute. So the JUD is classified as `service/jud` and has a JID of `jud.yak`, and the conferencing service is classified as `conference/public` and has a JID of `conference.yak`. The `type=""` and `jid=""` attributes are required. Each item has an optional `name=""` attribute for use when the item is displayed, for example.

Some services offer well-known facilities such as search and registration, which are commonly found across different services. These facilities can be described directly in the browse item, so that the entity requesting information about services receives information directly in the first request 'hit' as to what facilities are available for each service:

```
<ns>jabber:iq:search</ns>
<ns>jabber:iq:register</ns>
```

The 'ns' in the facility tagname (`<ns/>`) stands for namespace; it is via namespace-qualified requests to a service that features are utilised. In this case, the 'search' facility is represented by the `jabber:iq:search` namespace and the 'registration' facility is represented by the `jabber:iq:register` namespace.

Component Connection Method

Phew! Now that we've got the configuration out of the way, we can have a look how the jsm is loaded. And we can see immediately from the `<load/>` tags that it's connected using the *library load* method.

```
<load main="jsm">
  <jsm>./jsm/jsm.so</jsm>
  <mod_echo>./jsm/jsm.so</mod_echo>
  <mod_roster>./jsm/jsm.so</mod_roster>
  <mod_time>./jsm/jsm.so</mod_time>
  <mod_vcard>./jsm/jsm.so</mod_vcard>
  <mod_last>./jsm/jsm.so</mod_last>
  <mod_version>./jsm/jsm.so</mod_version>
  <mod_announce>./jsm/jsm.so</mod_announce>
  <mod_agents>./jsm/jsm.so</mod_agents>
  <mod_browse>./jsm/jsm.so</mod_browse>
  <mod_admin>./jsm/jsm.so</mod_admin>
```

```
  <mod_filter>./jsm/jsm.so</mod_filter>
  <mod_offline>./jsm/jsm.so</mod_offline>
  <mod_presence>./jsm/jsm.so</mod_presence>
  <mod_auth_plain>./jsm/jsm.so</mod_auth_plain>
  <mod_auth_digest>./jsm/jsm.so</mod_auth_digest>
  <mod_auth_0k>./jsm/jsm.so</mod_auth_0k>
  <mod_log>./jsm/jsm.so</mod_log>
  <mod_register>./jsm/jsm.so</mod_register>
  <mod_xml>./jsm/jsm.so</mod_xml>
</load>
```

It's clear that the more complex version of the method is employed here - as described in "Component Connection Methods" earlier in this Chapter - the jsm module itself is loaded through the `<jsm>...</jsm>` tag pair and this in turn pulls in the other modules that are specified with the `mod_*` module name tag pairs.

We've already become aquainted with some of the modules in this list; here's a quick summary of the ones that are being loaded here:

Modules loaded in jsm

`mod_echo`

: This module provides a simple echo service that echoes back whatever you send it.

`mod_roster`

: This module provides roster management services; the roster is stored server-side.

`mod_time`

: You can request the server send you a timestamp local to the server - this is the module that handles this request.

`mod_vcard`

: This is the module that handles requests for the Jabber server's vCard and also the user vCard management (such as submission to a JUD on change, and storing / retrieving the data from the server-side storage).

`mod_last`

: The `mod_last` provides facilities for returning 'last logout' information for users, or in the case of a query on the server itself, server uptime.

`mod_version`

: This is the module that provides the version query service described earlier in the section called *Update Info Request*.

`mod_announce`

: The server-wide announcements and MOTD facilities available to Jabber server administrators are provided by this module.

`mod_agents`

The `mod_agents` module responds to requests for 'agent' information made to the server. This is the module that returns the information in the `<browse/>` tag in the jsm configuration. It can also return a summary of the server consisting of the server's vCard and whether new user registrations are open.

When returning `<browse/>` data, it gives similar information to `mod_browse` (see the next entry) and is provided for backwards compatibility. The agent information is requested with two namespaces, `iq:agent` (for information on the server) and `iq:agents` (for information on a list of 'agents' - the old name for 'services'); these namespaces are being 'retired' in deference to the new `iq:browse` namespace.

`mod_browse`

The `mod_browse` module responds to browsing requests made on the server or on users defined on that server. The module can also be used by users to modify the information returned if a browse request is made against them.

`mod_admin`

This module provides the administrative features described in the section called *Administration*. The module itself determines which JIDs are allowed access to which features (according to the configuration in the `<admin/>` block).

`mod_filter`

The services described in the section called *Filter Service* are provided by this module.

`mod_offline`

Being offline - which in this sense means not being connected to the Jabber server and having an (online) *session* - doesn't prevent a user receiving messages. They are merely stored offline and forwarded to him when he becomes available - when he logs on and starts a session. `mod_offline` provides these storage and forwarding services. [5]

`mod_presence`

The management of presence information - whether a user is online or offline, what his presence settings currently are, who should be sent the information, and so on - these facilities are provided by the `mod_presence` module.

`mod_auth_*`

Authentication must take place when a user connects to the Jabber server and wishes to start a session. There are currently three types of authentication supported by the Jabber server - the differentiation is in how the password exchange and comparison is managed:

- **plaintext.** User passwords are stored in plaintext on the server and are transmitted from the client to the server in plaintext. [6] A simple comparison is made at the server to validate.

digest. User passwords are stored in plaintext on the server but no password is transmitted from the client to the server; instead, an SHA-1 digest is created by the client from the concatenation of the client's session id and the password and sent to the server, where the same digest operation is carried out and the results compared.

- **zero knowledge.** No user passwords are stored on the server, nor transmitted from the client to the server. A combination of hash sequencing on the client side with a final hash and comparison on the server side allows credentials to be checked in a secure way.

There are three `mod_auth_*` modules - one for each of these authentication types.

`mod_log`

`mod_log` simply records the ending of each user session.

`mod_register`

The `mod_register` module provides the services to register (create a new user) and unregister (remove a user) with the server.

`mod_xml`

Storage and retrieval of private and shared data by users is made possible by this module.

Component instance: *xdb*

The *xdb* component, described by the configuration XML shown in Example 4-10 and shown in diagram form in Figure 4-8, provides data storage for the server - it is the *X*ML *D*ata*B*ase.

Figure 4-8. Diagram view of *xdb* component instance

```
+-----+
| xdb |
+-----+-------------------------+
|                         xdb |
|--> host    *                |
|--> load    xdb_file.so      |
|--> config  spool            |
+-----------------------------+
```

All storage requirements by components connected to the Jabber backbone can be fulfilled by an xdb component. In normal configurations, there is a single instance, although it is possible to have more than one, each handling separate areas of storage, possibly using different storage mechanisms.

Example 4-10. `jabber.xml` configuration for the *xdb* component instance

```
<xdb id="xdb">

  <host/>

  <load>
    <xdb_file>./xdb_file/xdb_file.so</xdb_file>
  </load>

  <xdb_file xmlns="jabber:config:xdb_file">
    <spool><jabberd:cmdline flag='s'>./spool</jabberd:cmdline></spool>
  </xdb_file>

</xdb>
```

Component Type and Identification

The opening tag

```
<xdb id="xdb">
```

identifies this component instance to the backbone as an xdb type component, and gives it a name "xdb", in the same way that the sessions service has a name "sessions".

Host Filter

For the host filter, we have an empty tag

```
<host/>
```

specified, which signifies that this xdb component instance will answer data storage and retrieval requests for all hosts. This in turn means that all data to be stored server-side will be stored using the same data storage mechanism, in this case xdb_file, which is a simple 'lowest common denominator' storage system based upon directories containing files with XML content; these files are at a ratio of one per JID, plus a 'global' file where storage of data not tied to a JID is required. [7]

If you want to use separate data storage mechanisms for your different virtual servers, for example, you can define more than one xdb instance in your `jabber.xml` configuration and have the first use one storage system (say, xdb_file) and the second use another (say, an RDBMS-based system). [8]

You may also want to store data from different virtual hosts in different places on your system; by specifying more than one xdb instance, even if each of them uses the same storage mechanism, you can specify a different 'spool' directory in the configuration for each one.

As well as a host filter, there is another filter possible for xdb components. This is the namespace filter, represented by the `<ns/>` tag.

Every xdb storage and retrieval requests is made with a namespace definition; for example, to retrieve the last logoff time for a user, the `mod_last` module makes a data retrieval request of the xdb component and specifies the `jabber:iq:last` namespace in that request, and to check if a user is using the zero-knowledge authentication method, the `mod_auth_0k` module makes a data retrieval request and specifies the `jabber:iq:auth:0k` namespace.

If you want an xdb component instance to only handle requests qualified with certain namespaces, specify them with the `<ns/>` tag. Example 4-11 shows the initial part of an xdb component instance definition that is to handle `jabber:iq:roster` and `jabber:iq:last` qualified storage and retrieval request for the host `a-domain.com`.

Example 4-11. Host and namespace filters in an xdb definition

```
<xdb id="xdb">

  <host>a-domain.com</host>
  <ns>jabber:iq:roster</ns>
  <ns>jabber:iq:last</ns>

  ...

</xdb>
```

No namespace filter in an xdb component instance definition implies the instance is to handle requests qualified by any namespace - the equivalent of an empty tag:

```
<ns/>
```

To <ns/> or not to <ns/>
If you are going to use namespace filters for your xdb components, make sure you specify them for *every* xdb component instance definition, otherwise you will receive an error message to the effect that packet routing will be incorrect and your Jabber server will not start.

Custom Configuration

The custom configuration section in our xdb component instance definition is specific to the data storage mechanism that we're going to be using. In this case it's the xdb_file mechanism, and so we have the custom configuration wrapped by a tag qualified with a namespace to match:

```
<xdb_file xmlns="jabber:config:xdb_file">
  <spool><jabberd:cmdline flag='s'>./spool</jabberd:cmdline></spool>
</xdb_file>
```

Again, the tag name 'xdb_file' is unimportant - the part that must be correct is the namespace `jabber:config:xdb_file`.

The configuration describes a single setting - where the spool area is. This, in the context of our xdb_file mechanism, is the 'root' directory within which hostname-specific directories are created and used to store JID-specific and global XML data files. As the configuration stands here, the value (`./spool`) can be overridden at server startup time with the `-s` switch.

There is another configuration tag available for use here too. Using the configuration as it stands here, the xdb_file component would cache data indefinitely; if you were to modify data directly in the files in the spool area the modifications wouldn't have any effect for data that had already been retrieved for a JID in the course of a server's uptime. In other words, once data has been read from file, it is cached until the server is stopped. [9]

The `<timeout/>` configuration tag can be employed to control this cacheing. Used with a value that represents a number of seconds, the `<timeout/>` tag will force data in the cache to be purged (and therefore re-read from file the next time it's requested) after that number of seconds of lifetime. Table 4-3 shows the effects of various values on cacheing.

Table 4-3. `Effect of <timeout/>` tag

Tag value	Example	Effect
Less than 0	`<timeout>-1</timeout>`	No cache purge will be carried out and the cached data will live forever. This is the equivalent of having no explicit `<timeout/>` tag set.
0	`<timeout>0</timeout>`	Cache will be purged 'immediately'. This is the same as having no cache.
More than 0	`<timeout>120</timeout>`	Cached data will be purged after a certain lifetime specified (in seconds) as the value of the tag. In this example it's two minutes. Don't bother setting a positive value less than 30 - the cache purge check mechanism only runs every 30 seconds so any resolution beyond 30 is meaningless.

Component Connection Method

The component connection method is *library load*.

```
<load>
  <xdb_file>./xdb_file/xdb_file.so</xdb_file>
</load>
```

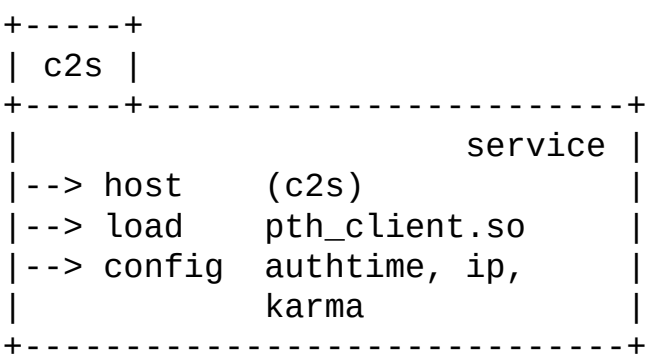
The shared library `./xdb_file/xdb_file.so` is loaded and the `xdb_file()` function is called to initialise the component.

Component Instance: *c2s*

The *c2s* component, described by the configuration XML shown in Example 4-12 and shown in diagram form in Figure 4-9, provides the *Client (to Server) Connections* service—it manages Jabber client connections to the Jabber

server.

Figure 4-9. Diagram view of *c2s* component instance



Example 4-12. `jabber.xml` configuration for the *c2s* component instance

```
<service id="c2s">

  <load>
    <pthsock_client>./pthsock/pthsock_client.so</pthsock_client>
  </load>

  <pthcsock xmlns='jabber:config:pth-csock'>
    <authtime/>
    <karma>
      <init>10</init>
      <max>10</max>
      <inc>1</inc>
      <dec>1</dec>
      <penalty>-6</penalty>
      <restore>10</restore>
    </karma>
    <ip port="5222"/>
  </pthcsock>

</service>
```

Component Type and Identification

The opening tag

```
<service id="c2s">
```

identifies this component instance to the backbone as a service type component, and gives it the name "c2s".

Host Filter

The c2s component has no explicit `<host/>` tag; the identification of the service with the `id=""` attribute is enough, and the value of the host filter will be taken as that identification. As long as the specified id is unique within the context of the whole configuration, the component will be able to function correctly. It is normally set to "c2s" by convention.

Custom Configuration

The custom configuration for our c2s component contains three main tags. The first, `<authtime/>` allows us to specify a time limit within which the connecting client has to have completed the authentication procedure. This includes sending the initial document stream identifier with the `jabber:client` namespace. Setting this to, say, ten seconds:

```
<authtime>10</authtime>
```

will allow the client up to 10 seconds to authenticate, after which c2s will drop the connection. Setting the time limit to 0 seconds, which can be accomplished with an empty tag:

```
<authtime/>
```

effectively gives the client an unlimited amount of time within which to authenticate.

The next tag we find in the c2s component instance configuration is `<karma/>`. This is a way of controlling bandwidth usage through the connections, and will be explained when we visit the "`<io/>` Section" later on in this Chapter.

Then we come to the `<ip/>` configuration tag.

```
<ip port="5222"/>
```

The default port for client connections is 5222. This is where it is specified. The `<ip/>` tag can contain an IP address - if you specify one like this:

```
<ip port="5222">192.168.0.4</ip>
```

then only socket connections will be made to that specific combination of port and IP address. Not specifying an IP address means that the c2s service will bind to the port on all (INADDR_ANY) IP addresses on the host.

You can specify more than one combination of port and IP address using multiple `<ip/>` tags:

```
<ip port="5222">
<ip port="5225">127.0.0.1</ip>
```

which here means client socket connections will be listened for on port 5222 on any IP address and port 5225 on the localhost address.

There are three other configuration tags not used here but worth identifying now.

<rate/>, like <karma/> is used to control connectivity and will be explained along with that tag in "<io/> Section" later.

<alias/> is a way of providing alias names for your Jabber server. When a Jabber client makes a connection, the opening gambit is the root of the XML document that is to be streamed over the connection:

```
<stream:stream to="furrybeast" xmlns="jabber:client"
               xmlns:stream="http://etherx.jabber.org/streams">
```

"furrybeast" may be a DNS alias for "yak", and is specified by the client here in the to="" attribute of the document's root tag (<stream/>).

With the <alias/> tag, we can 'fix' the incoming host specification by replacing it with what we as the server want it to be. If this document root tag were to be sent to our Jabber server configured as "yak" and we had an

<alias/> tag thus:

```
<alias to="yak">furrybeast</alias>
```

then the incoming host name specification "furrybeast" would be recognised and 'translated' to "yak" in the response:

```
<stream:stream xmlns:stream='http://etherx.jabber.org/streams'
               id='3AE71597' xmlns='jabber:client' from='yak'>
```

Rather than specify a hostname to translate, a default alias name can be specified like this:

```
<alias to="yak"/>
```

meaning that *any* connections to the c2s component would have their Jabber host name specification 'translated' to "yak" if necessary.

<ssl/> is the equivalent of the <ip/> tag and works in exactly the same way, with two exceptions:

- an IP address *must* be specified (that means something like

  ```
  <ssl port="5223"/>
  ```

  is not allowed).
- the connections are encrypted using SSL - for this the Jabber server must have been configured to use SSL - see Chapter 3 and the "io Section" later in this Chapter for details.

Component Connection Method

The component connection method is *library load*.

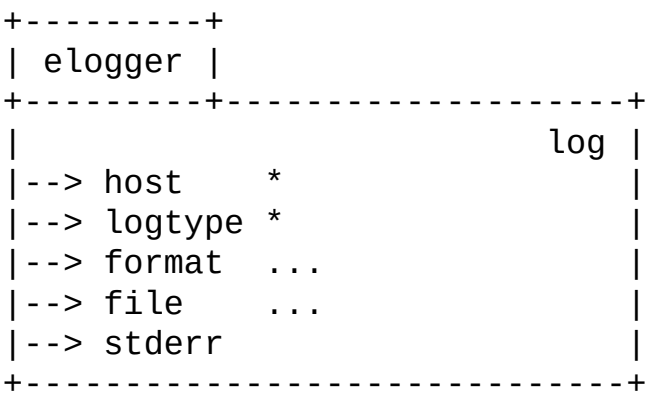
```
<load>
  <pthsock_client>./pthsock/pthsock_client.so</pthsock_client>
</load>
```

The shared library `./pthsock/pthsock_client.so` is loaded and the `pthsock_client()` function is called to initialise the component.

Logging Definition: *elogger*

It has already been intimated that the log type components are exceptions to the general pattern when it comes to defining what they are in relation to the Jabber server. In fact, the logging 'components' aren't really separate components at all - they are part of the **jabberd** backbone. Nevertheless, it is still worthwhile referring to them as components as they can be defined with different characteristics, to perform different logging tasks.

Figure 4-10. Diagram view of *elogger*



The configuration XML for *elogger* is shown in Example 4-13, and is represented in diagram form in Figure 4-10.

Example 4-13. `jabber.xml` configuration for *elogger*

```
<log id='elogger'>
  <host/>
  <logtype/>
  <format>%d: [%t] (%h): %s</format>
  <file>error.log</file>
  <stderr/>
</log>
```

Component Type and Identification

The opening tag clearly marks elogger as a log type component. The name given in the `id=""` attribute is "elogger".

Host Filter

elogger will record log records for any hosts according to the empty host filter tag specified here:

```
<host/>
```

Custom Configuration

Apart from the host filter declaration, every other tag within a `<log/>` definition can be regarded as configuration. Taking each tag in turn, we have:

- `<logtype/>`

  This tag declares which types of logging record will be handled by this logging definition. [10]

  The tag can either be empty, or contain one of the following values: `alert`, `notice`, `record` or `warn`. You can specify more than one `<logtype/>` tag to capture more than one log type. If you specify an empty tag (as is the case with the elogger component here) then all log types will be captured and handled, *apart from any log types that are explicitly declared elsewhere in other logging components*. What does this mean? Well, in our case, since we have a second log type component "rlogger" (described in the next section) that has an explicit

  ```
  <logtype>record</logtype>
  ```

  this elogger component won't receive log records of type '`record`' to handle.

- `<format/>`

  A logging component will typically write out the data it receives in a human readable format. With the `<format/>` tag, we can specify how the data appears. There are four variables which we can embellish with whatever text we like to form something that will be meaningful to us (and perhaps easily parseable for our scripts). The four variables are shown in table 4-4.

  In elogger's `<format/>` tag, we have

  ```
  %d: [%t] (%h): %s
  ```

  so a typical log record written by elogger might look like this:

  ```
  20010420T21:38:30: [warn] (yak): dropping a packet to yak
    from jsm@update.jabber.org/1.4.1: Unable to deliver, destination unknown
  ```

- `<file/>`

Typically the output from a logging component goes to a file. You can specify the name of the file with the `<file/>` tag.

- `<stderr/>`

  Additionally, it's possible to have the output from a logging component written to STDERR - place the empty `<stderr/>` tag in the logging component's definition to have this happen.

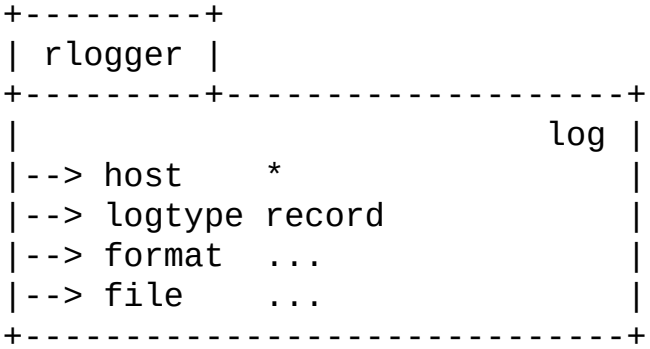
Table 4-4. Logging component variables for `<format/>`

Variable	Description
%d	Timestamp
%h	Host
%s	The actual log message
%t	Log type

Logging Definition: *rlogger*

Logging definition *elogger* is a general catch-all component that serves to direct all unhandled log records to a log file `error.log`. The logging definition *rlogger* on the other hand has been defined specifically to capture and store (to a file—`record.log`) `record` type log records.

Figure 4-11. Diagram view of *rlogger*



The configuration XML for *rlogger* is shown in Example 4-14, and is represented in diagram form in Figure 4-11.

Example 4-14. `jabber.xml` configuration for *rlogger*

```
<log id='rlogger'>
  <host/>
  <logtype>record</logtype>
  <format>%d %h %s</format>
```

```
  <file>record.log</file>
</log>
```

Component Type and Identification

Like elogger, rlogger is identified as a log type component. It takes its name from the `id=""` attribute.

Host Filter

Again, like elogger, this logging definition will handle log records for any hosts.

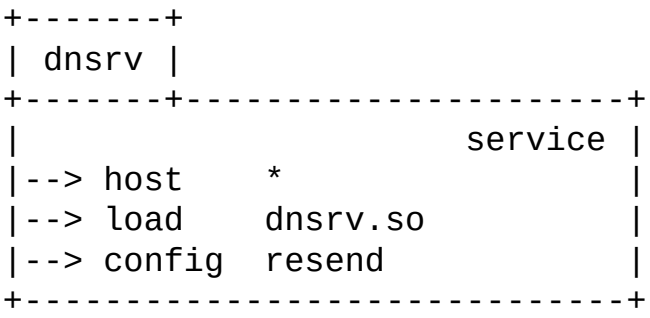
Custom Configuration

The custom configuration of rlogger is very similar to that of elogger, except that the target file is called `record.log` (the `<file/>` tag), the output format is slightly different (the `<format/>` tag) and no output to STDERR is desired.

Component Instance: *dnsrv*

The *dnsrv* component, described by the configuration XML shown in Example 4-15 and shown in diagram form in Figure 4-12, provides routing logic and name resolution for packets that are destined for a non-local component; in other words for a component that is running on another Jabber server. [11]

Figure 4-12. Diagram view of *dnsrv* component instance



Example 4-15. `jabber.xml` configuration for the *dnsrv* component instance

```
<service id="dnsrv">

  <host/>

  <load>
    <dnsrv>./dnsrv/dnsrv.so</dnsrv>
  </load>

  <dnsrv xmlns="jabber:config:dnsrv">
```

```
        <resend service="_jabber._tcp">s2s</resend>
        <resend>s2s</resend>
  </dnsrv>

</service>
```

The component, once started, forks to spawn a child process that services the actual resolution requests and the route determination. The component and its child communicate with a simple XML stream within which hostnames are passed to the child process in a 'query' tag:

```
<host>update.jabber.org</host>
```

and answers are passed back in the form of attribute additions to the original query tag:

```
<host ip='208.245.212.100' to='s2s'>update.jabber.org</host>
```

Component Type and Identification

The component is a service, and is identified with the name "dnsrv":

```
<service id="dnsrv">
```

Host Filter

The dnsrv is to provide hostname resolution and routing for all component activity within the Jabber server. For this reason, it needs to be open to all comers and has an empty host filter tag:

```
<host/>
```

Custom Configuration

The dnsrv component provides hostname lookup and dynamic routing services. To this end, the configuration concerns itself with defining how the routing is to be determined.

The configuration, identified with the `jabberd:config:dnsrv` namespace, contains a list of entries that describe service-types and next-delivery-point data. What does this mean? In hostname resolution, a lookup request can be made where a service and protocol are specified as well as a domain name. In the case where more than one server shares the same domain name, and services (such as Jabber) are managed across the servers, different host addresses can be returned, depending on the service requested. So far so good - this is our resolution part of the deal covered. But what happens once an IP address has been returned? The packet destined for the non-local component must be sent on its way - but via where? This is what the next-delivery-point data specifies.

If we examine the configuration, we see this:

```
<dnsrv xmlns="jabber:config:dnsrv">
  <resend service="_jabber._tcp">s2s</resend>
```

```
  <resend>s2s</resend>
</dnsrv>
```

The configuration is a list of services to try for during the resolution request. This particular list has two items. The first has the `service="_jabber._tcp"` attribute that says "try for the Jabber (via TCP) service when trying to resolve a name (using a SRV record lookup request) and if successful, send the packet on to the s2s component". The second is the default which says "If you've reached here in the list and haven't managed to get a resolution for a particular service, just resolve it normally (using a standard resolver call) and send it on to the s2s component". [12]

The `_jabber._tcp` is not Jabber configuration syntax, it is the prefix format to use with a domain name when making DNS SRV lookups. [13]

Let's look at what happens in a typical request of dnsrv; to set the scene, a client connecting to the Jabber server to which this dnsrv component instance is connected has requested software update information (see the section called *Update Info Request*) and as no local component with the identification or filter that fits the hostname `update.jabber.org` was found in the configuration, a request to resolve and route to this hostname is passed on to the dnsrv component.

1. The component receives a request for `update.jabber.org`
2. Resolution attempt for first `<resend/>` tag in the list - this is the one that implies a SRV lookup by specifying the `_jabber._tcp` service definition.
3. The SRV lookup fails - there are no SRV records maintained for the Jabber service for `update.jabber.org`.
4. Resolution attempt for second (and last) `<resend/>` tag in the list. No service is specified in this tag, so a normal resolution lookup is made.
5. The lookup is successful, and returns an IP address.
6. The success means that we've got a match, and the packet destined for `update.jabber.org` can be passed on to the component specified as next in the chain, which is s2s, as specified in the tag:

   ```
   <resend>s2s</resend>
   ```

Component Connection Method

The dnsrv component is loaded as a shared library with the *library load* method:

```
<load>
  <dnsrv>./dnsrv/dnsrv.so</dnsrv>
</load>
```

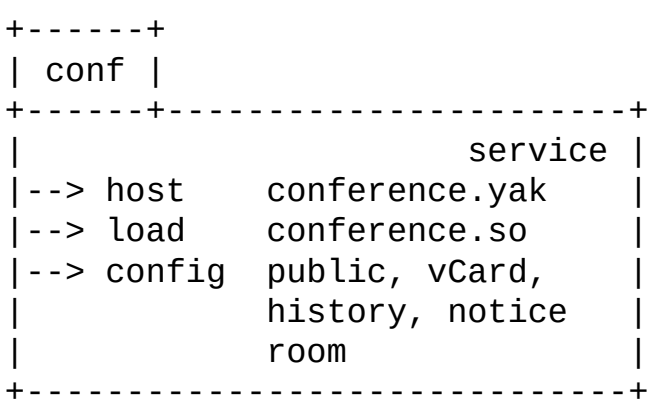
and the `dnsrv()` function is called when loading is complete to initialise the service.

Component Instance: *conf*

The *Conferencing* component is a service that provides group chat facilities in a Jabber environment. Rooms can be created and people can join and chat, in a similar way to how they do in IRC (Internet Relay Chat) channels. The

component instance described by the configuration XML in Example 4-16 is shown in diagram form in Figure 4-13.

Figure 4-13. Diagram view of *conf* component instance



Example 4-16. `jabber.xml` configuration for the *conf* component instance

```
<service id='conf'>

  <host>conference.yak</host>

  <load>
    <conference>./conference-0.4.1/conference.so</conference>
  </load>

  <conference xmlns="jabber:config:conference">
    <public/>
    <vCard>
      <FN>yak Chatrooms</FN>
      <DESC>This service is for public chatrooms.</DESC>
      <URL>http://yak/chat</URL>
    </vCard>
    <history>20</history>
    <notice>
      <join> has become available</join>
      <leave> has left</leave>
      <rename> is now known as </rename>
    </notice>
    <room jid="kitchen@conference.yak">
      <name>The Kitchen</name>
      <notice>
        <join> has entered the cooking melee</join>
        <leave> can't stand the heat</leave>
        <rename> now answers to </rename>
      </notice>
    </room>
  </conference>
```

```
</service>
```

The component acts as a sort of third party, and all interaction between room participants are made through this third party. This makes it possible to support privacy (such as using nicknames to hide users' real JIDs) and other features.

Component Type and Identification

The *Conferencing* component is identified to the backbone as a service:

```
<service id="conf">
```

and is given the identity "conf", with which the component instance will register itself when loaded.

Host Filter

By convention, it is often the case that the *Conferencing* component is addressed (by clients) as 'conference.<hostname>'; we see that convention has been followed in that the host filter for this instance is

```
<host>conference.yak</host>
```

which means that all packets destined to any id at the hostname `conference.yak` will be sent to the conf component instance. This matches the identification of this service in the `<browse/>` section of the jsm custom configuration.

Custom Configuration

Configuration of the *Conferencing* component is straightforward and is identified with the `jabber:config:conference` namespace:

```
<conference xmlns="jabber:config:conference">
  ...
</conference>
```

We can see from the contents of the custom configuration that there are a number of elements:

- *Public or private service*

  ```
  <public/>
  ```

  Specifying (with the `<public/>` tag) that a conference service is public means that users are allowed to browse the elements that the service is controlling, namely the rooms. Rooms are either pre-created or created on the fly by the first user to specify a new name when requesting to join a room. Specifying (with the `<private/>` tag) that a conference service is private means that users are only allowed to browse rooms that they already know about, meaning rooms in which they're already present.
- *vCard*

```
<vCard>
  <FN>yak Chatrooms</FN>
  <DESC>This service is for public chatrooms.</DESC>
  <URL>http://yak/chat</URL>
</vCard>
```

The conference service component can have its own vCard information which can be requested at any time. Here is where that vCard information can be maintained. Like the vCard for the jsm service, this particular definition only uses a few of the many possible vCard fields.

- *Message history*

```
<history>20</history>
```

When you join a room, it is sometimes useful to see some of the most recent messages from the room's conversation(s). With the `<history/>` tag you can specify how many previous messages are relayed to new room joiners.

If you don't specify a `<history/>` tag a default value of 10 will be used.

- *Action Notices*

```
<notice>
  <join> has become available</join>
  <leave> has left</leave>
  <rename> is now known as </rename>
</notice>
```

There are three main 'events' that happen with users and rooms. A user enters a room; a user leaves a room; a user changes his nickname. When any of these events occurs, the conference service component sends some text to the room to notify the participants. You can modify the text that gets sent with the tags in the `<notice/>` configuration element.

- *Rooms*

```
<room jid="kitchen@conference.yak">
  <name>The Kitchen</name>
  <notice>
    <join> has entered the cooking melee</join>
    <leave> can't stand the heat</leave>
    <rename> now answers to </rename>
  </notice>
</room>
```

A room is normally created when a user requests to join a room that doesn't already exist - the requesting user is determined to be the room's owner. It is also possible to pre-create rooms when the service starts up; you can define rooms to be pre-created with the `<room/>` tag. Each room has a JID. Optionally you can give the room a name (which may be displayed by clients as the room's title), and its own action notices.

There are other settings for rooms that you can specify within a `<room/>` tag, and these are shown in Table 4-5.

Table 4-5. Conference room settings

Setting	Description
`<nick/>`	This signifies that a nickname is required for entry to the room. If one is not specified (in the join request) a nick will be constructed for the user dynamically, usually the JID with a numeric suffix to make it unique in the room.
`<secret/>`	Rooms can be protected from unauthorized entry by specifying a secret which will be required on entry.
`<privacy/>`	This tag signifies that a privacy mode is supported, which means that users' real JIDs will be hidden from browsing requests.

Component Connection Method

The *Conferencing* component is compiled into a shared object library (`./conference-0.4.1/conference.so`) and is connected to the Jabber backbone with the *library load* method:

```
<load>
  <conference>./conference-0.4.1/conference.so</conference>
</load>
```

Once loaded, the function `conference()` is called to initialize the component and perform setup tasks, such as creating rooms specified in the configuration.

Component Instance: *jud*

As mentioned already, the JUD is a user directory service that provides storage and query facilities for user name and address data. Figure 4-14 shows, in diagram form, the XML configuration for the *jud* component instance in Example 4-17.

Figure 4-14. Diagram view of *jud* component instance

```
+-----+
| jud |
+-----+-------------------------+
|                       service |
|--> host    jud.yak            |
|--> load    jud.so             |
|--> config  vCard              |
+-------------------------------+
```

The JUD that is defined here relies upon the xdb component for data storage and retrieval services, which in turn

means that in this case, the data is stored in XML format in a file under the directory defined in the `<spool/>` tag in the xdb component instance's definition. All the data managed by the JUD is stored in one lump, with no specific JID associated with it; this means that xdb's engine, xdb_file, will store it as a single file called `global.xml` under the directory named after the JUD 'hostname' `jud`.

Example 4-17. `jabber.xml` configuration for the *jud* component instance

```
<service id="jud">

  <host>jud.yak</host>

  <load>
    <jud>./jud-0.4/jud.so</jud>
  </load>

  <jud xmlns="jabber:config:jud">
    <vCard>
      <FN>JUD on yak</FN>
      <DESC>yak User Directory Services</DESC>
      <URL>http://yak/</URL>
    </vCard>
  </jud>

</service>
```

Component Type and Identification

JUD is clearly a service component, and is identified as such with this tag:

```
<service id="jud">
```

The name given to the component instance is "jud".

Host Filter

Requests of the JUD, such as searches or registrations (a user 'registers' with the JUD and thereby causes his name and address details to be stored by the JUD) must be directed specifically at the JUD, which we have identified in the `<browse/>` area of our jsm configuration (see the section called *Custom configuration*) as `jud.yak`.

As we have identified the JUD in this way, requests will reach the JUD by way of this hostname, which is therefore what we want to filter on:

```
<host>jud.yak</host>
```

Requests to any other hostnames are not appropriate for the JUD to handle and will therefore be filtered out.

Custom Configuration

There is not much to configure in the JUD; it is a simple user directory service and many of the features are currently hardcoded - where the data is stored, what data fields are stored per JID, and so on. The only configuration we can maintain is the JUD's vCard information. Just as the Jabber server itself and each user can have a vCard, components can have vCards too. These component vCards can be requested in the same way. [14]

```
<jud xmlns="jabber:config:jud">
  <vCard>
    <FN>JUD on yak</FN>
    <DESC>yak User Directory Services</DESC>
    <URL>http://yak/jud</URL>
  </vCard>
</jud>
```

The namespace that declares the JUD configuration is `jabber:config:jud`.

Component Connection Method

The JUD defined here is implemented as a set of C programs compiled into a shared object (`./jud-0.4/jud.so`) library. It is connected to the backbone with the *library load* connection method:

```
<load>
  <jud>./jud-0.4/jud.so</jud>
</load>
```

and the function `jud()` is called to initialise the component.

Component Instance: *s2s*

Just as the c2s component provides the *Client (to Server) Connections* service, so the s2s component provides the *Server (to Server) Connections* service. The XML configuration that describes the *s2s* component is shown in Example 4-18, and is represented in diagram form in Figure 4-15.

Figure 4-15. Diagram view of *s2s* component instance

```
+-----+
| s2s |
+-----+-------------------------+
|                       service |
|--> host    (s2s)              |
|--> load    dialback.so        |
|--> config  legacy, ip, karma  |
+-------------------------------+
```

Example 4-18. `jabber.xml` configuration for the *s2s* component instance

```
<service id="s2s">

  <load>
    <dialback>./dialback/dialback.so</dialback>
  </load>

  <dialback xmlns='jabber:config:dialback'>
    <legacy/>
    <ip port="5269"/>
    <karma>
      <init>50</init>
      <max>50</max>
      <inc>4</inc>
      <dec>1</dec>
      <penalty>-5</penalty>
      <restore>50</restore>
    </karma>
  </dialback>

</service>
```

Component Type and Identification

The component type is service, and the instance here is identified as "s2s":

```
<service id="s2s">
```

Host Filter

As with the c2s component instance definition, no explicit host filter is set here for the s2s. The identification of the component instance as "s2s" acts as an ersatz host filter.

Custom Configuration

The configuration for the s2s is similar to that of the c2s; after all, it is about managing connections to other hosts. The configuration namespace is, however, a little odd:

```
<dialback xmlns="jabber:config:dialback">
```

'Dialback'? Well, in order to prevent spoofing on a connecting server's part, the s2s component implements a identity verification mechanism that is used to check that a connecting server is who it says it is. See the sidebar entitled "Dialback" for more details.

Dialback

When an external Jabber server requests a connection to your Jabber server, the dialback mechanism ensures that the external Jabber server's identification can be verified. To do this, your Jabber server 'dials back' the external Jabber server to check credentials.

The sequence of events in a dialback conversation involves three (logical) parties and is roughly as follows:

1. The external Jabber server makes a connection to your Jabber server and in the opening document stream root tag declares support for the dialback mechanism.

   (*External server: "Hello, I support dialback."*).
2. After receiving your Jabber server's document stream root tag as response, the external Jabber server sends a key to your Jabber server to be verified.

   (*Your server: "Oooh, hello, so do I." External server: "Verify this key which will prove my identify, please."*).
3. Your server performs a DNS lookup on the name with which the external server identified itself, and makes a connection to the address given (this should normally be the external server itself, but in the case of a complex server farm environment, might not be), sending it a (second) opening document stream root tag, declaring support for dialback. This resolved address is the address of what is called the 'authoritative' server.

   (*Your server: "Hello, I support dialback."*)
4. The authoritative server responds by sending its document stream root tag.

   (*Authoritative server: "Hello, I do too."*)
5. Now your Jabber server asks the authoritative server to verify the key previously sent by the external server.

   (*Your server: "Can you verify this key for me please?"*)
6. The authoritative server can determine whether the key is valid or not (after all, it should be either the same server that originated the key, or share the key with the authoritative server within a server farm environment).

   (*Authoritative server: "Yes, the key is valid." or "No, the key is not valid - imposter alert!"*)

Based on the validity of the key, your Jabber server can accept or refuse the incoming connection from the external Jabber server.

And of course, the tables are turned and the boot is on the other foot, to mix metaphors, if your Jabber server is the one trying to make a connection.

Server-to-server namespace: As the namespace for the exchange of document streams in a client-to-

server connection is `jabber:client`, so the namespace for the exchange of document streams in a server-to-server connection is `jabber:server`.

There are three immediate child tags in the configuration wrapper tag:

- `<legacy/>`

  This acts as a flag that allows (or not, if it is absent) 'legacy' Jabber servers to connect. A 'legacy' Jabber server is one that has version 1.0, and, of more relevance, no support for the dialback mechanism. Without the tag, an incoming connection from a version 1.0 Jabber server that didn't support dialback would be refused.
- `<ip/>`

  While a normal Jabber server listens for client connections on 5222, it listens for connections from other Jabber servers on port 5269. This is specified with the `<ip/>` tag, which has the same characteristics as the `<ip/>` tag in the c2s configuration settings (more than one tag allowed, specific IP address optional).
- `<karma/>`

  Karma is used in the s2s component to control connection traffic, just as it is used in c2s. See the "`<io/>` Section" below for more details.

Component Connection Method

The *library load* method is used to connect the s2s component to the backbone:

```
<load>
  <dialback>./dialback/dialback.so</dialback>
</load>
```

The `dialback()` is called in the shared library after it has been loaded.

io Section

The io section of the `jabber.xml` configuration file, shown in Example 4-19 and represented in diagram form in Figure 4-16, is where a number of settings relating to socket communication with the Jabber server are set.

Figure 4-16. Diagram view of *io* section

```
+----+
| io |
+----+-----------------+
|--> rate   ...        |
|--> karma  ...        |
|--> ssl    ...        |
|--> allow  ...        |
|--> deny   ...        |
```

```
+----------------------+
```

Example 4-19. `jabber.xml` configuration for the *io* section

```
<io>

  <karma>
    <heartbeat>2</heartbeat>
    <init>64</init>
    <max>64</max>
    <inc>6</inc>
    <dec>1</dec>
    <penalty>-3</penalty>
    <restore>64</restore>
  </karma>

  <rate points="5" time="25"/>

</io>
```

Although a distinct section, *io* does not describe a component with custom configuration or a connection method - the contents are merely settings. Let's examine each of these settings here.

`<rate/>`

The `<rate/>` tag afford us a sort of connection throttle by allowing us to monitor the rate at which incoming connections are made, and to put a hold on further connections if the rate is reached.

The rate is calculated to be a number of connection attempts - from a single IP address - within a certain amount of time. We can see these two components of the rate formula as attributes of the `<rate/>` tag itself:

```
<rate points="5" time="25"/>
```

This means acceptance of incoming connections from an individual IP address will be stopped if more than 5 connection attempts (`points`) are made in the space of 25 seconds (`time`).

The 'rating' (the throttling of connection attempts) will be restored at the end of the period defined (25 seconds in this case).

The effect of a `<rate/>` tag in this io section is server-wide; all socket connections (for example c2s) can be rate-limited. If there is no specific `<rate/>` specification in a particular service that listens on a socket for connections, then the specification in this io section is used. If no `<rate/>` tag is specified in this io section, then the server defaults are used - these are actually the same as what's explicitly specified here.

`<karma/>`

Like the `<rate/>` tag, `<karma/>` is used to control connectivity. Whereas rating helps control the number of

connections, karma allows us to control the data flow rate per connection once a connection has been made.

The concept of karma is straightforward; each socket has a karma value associated with it. We can understand it better if we think of it as each entity (connecting through a socket) having a karma value. The higher the value - the more karma - an entity has, the more data it is allowed to send through the socket. So as rating is a throttle for connections, so karma is a throttle for data throughput.

There are certain settings that allow us to fine-tune our throughput throttle. Table 4-5 lists these settings, along with the values explicitly set in each of the c2s and s2s component sections in our `jabber.xml` file. Notice how the settings for the *Server (to Server) Connections* component are considerably higher than those for the *Client (to Server) Connections* - this is based on the assumption that server-to-server traffic will be greater than client-to-server *on a socket by socket basis*.

The relationship between an entity's karma and how much data it is allowed to write to the socket is linear - in fact the amount is

```
(karma value * 100)
```

and this every 2 seconds. The multiplier (100) and the karma period (2) are hardcoded into the server - a recompilation would be required to change these values.

Over time, an entity's karma value will increase, up to a maximum value (we need a ceiling on how much we're going to allow an entity to send!) every karma period (2 seconds).

The same karma formula is used to penalize an entity for sending too much data. If more than `(karma * 100)` bytes are sent within a certain period, the entity's karma value is decreased. Once the value reaches zero, it is plunged to a negative number meaning that the entity must take a breather until the value grows back to zero (over time, it will). At this point, the value will be 'restored' to a value that gives the entity chance to start sending data again.

Table 4-6. Settings for karma control, with c2s and s2s values

Setting	c2s values	s2s values	Description
`<init/>`	10	50	The initial value for karma on a new socket
`<max/>`	10	50	The maximum karma value that can be attained by a socket
`<inc/>`	1	4	By how much the karma value is incremented (over time)
`<dec/>`	1	1	By how much the karma value is decremented in a penalty situation
`<penalty/>`	-6	-5	The karma value is plunged to this level once it falls to zero
`<restore/>`	10	50	The karma value is boosted to this level once it rises (after a penalty) to zero.

`<ssl/>`

If you have compiled your Jabber server with SSL (see Chapter 3) and want to use SSL-encrypted connections, you will have to have specified the `<ssl/>` tags in the configuration of the c2s component instance. Furthermore, you

must specify the location of your SSL certificate and key file. There is an <ssl/> tag in this io section for this purpose.

You can have separate files for each IP address specified in the c2s component instance configuration's <ssl/> tag. Example 4-20 shows the specification of two .pem files - one for each of two IP addresses.

Example 4-20. Specifying SSL certificate & key files per IP address

```
<ssl>
  <key ip="192.168.0.4">/usr/local/ssl/certs/ks1.pem</key>
  <key ip="192.168.9.1">/usr/local/ssl/certs/ks2.pem</key>
</ssl>
```

<allow/> and <deny/>

You can control at the IP address and network level who can connect to your Jabber server with the <allow/> and <deny/> tags.

The default (when no tags are specified) is to allow connections from everywhere. If you use <allow/> tags, then connections will be allowed *only* from the addresses or networks specified. If you use <deny/> tags, then connections will be denied from those addresses or networks specified. If you have both <allow/> and <deny/> tags, the intersection of addresses between the two tag sets will be *denied* - in other words, <deny/> overrides <allow/>.

The tags wrap individual ip addresses, which are specified using the <ip/> tag, or network addresses, which are specified using the <ip/> tag in combination with the <mask/> netmask tag. Example 4-21 shows connections to a Jabber server being limited to hosts from two internal networks with the exception of one particular IP addresses, and a specific host on the Internet.

Example 4-21. Using <allow/> and <deny/> to control connections

```
<allow>
  <ip>192.168.10.0</ip>
  <mask>255.255.255.0</mask>
</allow>

<allow>
  <ip>192.168.11.0</ip>
  <mask>255.255.255.0</mask>
</allow>

<allow>
  <ip>195.82.105.244</ip>
</allow>

<deny>
  <ip>192.168.11.131</ip>
```

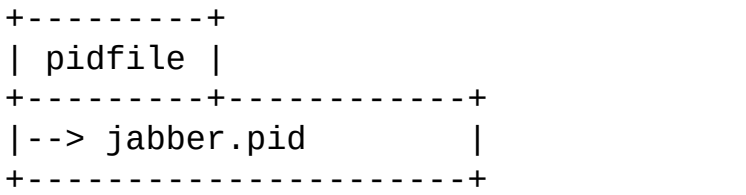
```
</deny>
```

pidfile Section

The pidfile section simply specifies the name of the file where the process ID (PID) of the Jabber server will be written at startup. In this case the name of the file is `./jabber.pid`.

The XML describing this section is shown in Example 4-22, and is represented in diagram form in Figure 4-17,

Figure 4-17. Diagram view of *pidfile* section



Example 4-22. `jabber.xml` configuration for the *pidfile* section

```
<pidfile>./jabber.pid</pidfile>
```

Notes

[1] Remember that if no `<host/>` is specified, the instance id - in this case "sessions" is used instead.

[2] The queries to retrieve a list of online users are addressed to the server directly - here the recipient JID is the servername; the `iq:admin` query version actually requires a resource of 'admin' to be suffixed to the server name too. For more information on the makeup of JIDs, see Chapter 5.

[3] Actually, that's not true - it was just to catch those of you who were falling asleep - Jabber clients can use the Out Of Band (OOB) namespace to negotiate file transfers between themselves. See Part II for more details.

[4] The JUD is referred to here as a 'local' JUD because there's a central JUD running on `jabber.org` that every Jabber user can connect to and use (even if they're on a different Jabber server); the word 'local' makes the distinction that the JUD is running locally to the Jabber server to which the Jabber user is connected.

[5] in conjunction with the xdb component - see later.

[6] If the connection between the client and the server is encoded using SSL, then the plaintext password travels through an encrypted connection.

[7] An example of this would be JUD's usage of xdb (and implicitly xdb_file in our configuration) - a file called `global.xml` is used to store the user directory information that JUD manages.

[8] Some of these are being developed and are available right now

[9] Only do this if you're sure you know what you're doing.

[10] Actually, the `<logtype/>` tag is more of a filter (like `<host/>`) than configuration. But that's splitting hairs.

[11] Or on a server offering another IM service, for example - this is planned for the future.

[12] `gethostbyname()`

[13] See RFC 2782 for more details.

[14] Ok, the vCard in this case is actually for the jsm, which is the heart of the Jabber server...

Prev Server Configuration | Home | Up | Next Managing the Configuration

Managing the Configuration

Now that we've had a tour of the components, and have an idea of what sorts of configurations are possible, you may be wondering whether there's a way to retain some sort of overview of the actual XML. Dropping component instance definitions in and out of the configuration file is somewhat tedious, and certainly when editing such a large file, it's not difficult to lose sense of direction and comment out or edit the wrong section.

<jabberd:include/>

Help is at hand, in the form of the `<jabberd:include/>` tag.

This tag comes from the same stable as `<jabberd:cmdline/>` and provides the Jabber server administrator with ways to better manage the XML configuration.

The contents of a file specified with the `<jabberd:include/>` tag are imported (included) in the position that the `<jabberd:include/>` tag occupies. Depending on what the root tag in the file to be included is, the import is done in one of two ways:

- *root tag matches the parent tag of* `<jabberd:include/>`

  The contents of the file *minus the root tag* are included
- *root tag does not match the parent tag of* `<jabberd:include/>`

  The entire contents of the file are included

For example, if we have a section like this in the `jabber.xml` file:

```
...
<conference xmlns="jabber:config:conference">
  <public/>
  <vCard>
    <FN>yak Chatrooms</FN>
    <DESC>This is a public chatroom service.</DESC>
    <URL>http://yak/chat</URL>
  </vCard>
```

```
  ...
  <jabberd:include>./rooms.xml</jabberd:include>
</conference>
...
```

and the content of `./rooms.xml` looks like this:

```
<room jid="kitchen@conference.yak">
  <name>The Kitchen</name>
  <notice>
    <join> comes to add to the broth-spoiling</join>
    <leave> can't stand the heat</leave>
    <rename> is now known as </rename>
  </notice>
</room>
<room jid="cellar@conference.yak">
  <name>The Cellar</name>
  <secret>cellarsecret</secret>
</room>
```

then these rooms will be defined to the *Conferencing* component as if the configuration XML had appeared directly inside of the `<conference/>` configuration wrapper tag.

We can put the `<jabberd:include/>` tag to good use and organise our configuration component instances as shown in Example 4-23.

Example 4-23. Configuration XML organised with `<jabberd:include/>`

```
<jabber>

  <!-- Core components -->

  <jabberd:include>./sessions.xml</jabberd:include>
  <jabberd:include>./config/standard/xdb.xml</jabberd:include>
  <jabberd:include>./config/standard/c2s.xml</jabberd:include>

  <!-- Testing -->

  <!--
  <jabberd:include>./config/local/conference.xml</jabberd:include>
  <jabberd:include>./config/test/test.service.xml</jabberd:include>
  -->
```

```
  <!-- Logging -->

  <jabberd:include>./config/standard/elogger.xml</jabberd:include>
  <jabberd:include>./config/standard/rlogger.xml</jabberd:include>

  <!--
  Internal-only server right now

  <jabberd:include>./config/standard/dnsrv.xml</jabberd:include>
  <jabberd:include>./config/standard/s2s.xml</jabberd:include>
  -->

  <!-- Misc -->

  <jabberd:include>./config/standard/jud.xml</jabberd:include>

  <!-- IO (incl. karma), PIDfile -->

  <jabberd:include>./config/standard/io.xml</jabberd:include>
  <jabberd:include>./config/standard/pidfile.xml</jabberd:include>

</jabber>
```

The XML in Example 4-23 gives us a great overview of which components are included in our Jabber server; we have the core components providing the *Session Management*, *Client (to Server) Connections* and *Data Storage* services; a couple of components under test (*Conferencing* and a custom component we're calling 'test.service') that are currently deactivated; the *Logging* services in their standard configuration; the components providing facilities for connecting to other Jabber servers - *Server (to Server) Connections* and *Hostname Resolution* - are currently inactive, meaning that as configured, the Jabber server will be purely internal; there's also a local JUD defined too, and finally we have the IO and PIDfile specifications - also abstracted out into separate XML chunks.

This works well especially if there are certain parts of the configuration - for example certain component instance definitions - that don't normally change; you can see that many of the component configuration files are in a 'standard' directory, which by convention could signify that they're the same as the XML configuration as-delivered and are not likely to change.

<jabberd:cmdline/>

The `<jabberd:cmdline/>` tag was mentioned in Chapter 3 as a way of providing a command line hook into the configuration - values stored in the XML could be overridden by command line switches used when invoking **jabberd**.

The tag is used in the standard XML configuration (see Figure 4-3) to allow replacement of the hostname

and spool directory:

```
<host><jabberd:cmdline flag="h">yak</jabberd:cmdline></host>
```

and

```
<spool><jabberd:cmdline flag='s'>./spool</jabberd:cmdline></spool>
```

In fact, this tag can be used in most places in the XML - so if for example you have a requirement to modify (re-specify) the error and record log files for each **jabberd** invocation, you can do something like this:

```
<log id='elogger'>
  <host/>
  <logtype/>
  <format>%d: [%t] (%h): %s</format>
  <file><jabberd:cmdline flag="e">error.log</jabberd:cmdline></file>
  <stderr/>
</log>
```

and then override the value `error.log` with something else at invocation time:

```
yak:~/jabber-1.4.1$ ./jabberd/jabberd -e error_log.txt &
```

Server Constellations

Throughout the discussion of components in this chapter, and how they're arranged to form a 'complete' Jabber server, we've only really considered a monolithic server, running in a single process. [1]

However, there may be good reasons (performance, administration and manageability) to run the Jabber server in different configurations, or 'constellations'. [2]

In this concluding section of Chapter 4 we take a look at some of the possible constellations, and how they're constructed.

Multiple Servers on one Host

Although it's unlikely that this constellation would be of much use, it is possible to run more than one Jabber server on one host simply by creating multiple installations, maintaining each server's `jabber.xml` configuration file separately, and starting them up to listen on different ports to each other. Note that some Jabber clients don't support connections to anything other than port 5222, however.

As we have seen from examining the instance configuration for the *Client (to Server) Connections* and the *Server (to Server) Connections* components, the 'standard' Jabber ports for client and server connectivity are 5222 and 5269 respectively. To run a second Jabber server on the same host, just ensure that its *Connections* component instances are configured to listen on different ports.

'Real' Virtual Jabber Servers

While looking at the section called *Host filter* we saw how to use multiple `<host/>` tags to allow connection to the Jabber server under multiple hostnames. Although this simple feature might be useful in some circumstances, a better distinction of *Session Management* functionality might be more appropriate.

Taking our `a-domain.com` and `b-domain.com` hostname examples again, we might want to offer different 'welcome' messages to new users; we may want to limit the authentication possibilities for the `b-domain.com` host to zero-knowledge only; and disable the message filtering service for the `a-domain.com` host; furthermore it's likely that we'd want to offer - in the `<browse/>` list - a different set of services for each of the hosts.

Let's have a look how this can be done. Using the `<jabberd:include/>` tag to organise our configuration XML by component instance definitions, we might have a `jabber.xml` configuration file that looks like Example 4-24.

Example 4-24. Virtual server `jabber.xml` configuration

```
<jabber>

  <!-- Common components -->

  <jabberd:include>./config/common/xdb.xml</jabberd:include>
  <jabberd:include>./config/common/c2s.xml</jabberd:include>
  <jabberd:include>./config/common/elogger.xml</jabberd:include>
  <jabberd:include>./config/common/rlogger.xml</jabberd:include>
  <jabberd:include>./config/common/dnsrv.xml</jabberd:include>
  <jabberd:include>./config/common/s2s.xml</jabberd:include>


  <!-- a-domain.com -->

  <jabberd:include>./config/a-domain/sessions.xml</jabberd:include>
  <jabberd:include>./config/a-domain/conference.xml</jabberd:include>


  <!-- b-domain.com -->

  <jabberd:include>./config/b-domain/sessions.xml</jabberd:include>
  <jabberd:include>./config/b-domain/conference.xml</jabberd:include>
  <jabberd:include>./config/b-domain/jud.xml</jabberd:include>


  <!-- IO, PIDfile -->

  <jabberd:include>./config/common/io.xml</jabberd:include>
  <jabberd:include>./config/common/pidfile.xml</jabberd:include>

</jabber>
```

What can we see here?

First, a-domain.com and b-domain.com Jabber users will share the common facilities such as *Data Storage*, remembering that data will be stored by hostname within the spool area (and while you're remembering, remember also that you *can* use two xdb component instances, specifying a different host filter in each, to store data in separate places - see the section called *Component instance: xdb* earlier in this Chapter), *Client (to Server) Connections*, *Logging* and so on. They also share the same IO settings and PIDfile definition - after all, there is still only one Jabber server that is hosting these two virtual servers, so we only need one PIDfile.

But we also see that there are two `sessions.xml` files included - one for the `a-domain.com` host and another for the `b-domain.com` host. And with each of the `sessions.xml` files included, we have one or two other components - for *Conferencing* and JUD services.

Configuration for a-domain.com

The layout in the `jabber.xml` file indicates that there are separate definitions for each of the two hosts. Let's examine the contents of `./config/a-domain/sessions.xml`:

```
<service id="sessions.a-domain">

  <host>a-domain.com</host>

  <jsm xmlns="jabber:config:jsm">

    <!-- no filter config necessary -->

    <vCard>
      <FN>a-domain.com Jabber Services</FN>
      <DESC>Jabber 1.4.1 on a-domain.com</DESC>
      <URL>http://www.a-domain.com</URL>
    </vCard>

    <browse>
      <conference type="public" jid="conference.a-domain.com"
                  name="a-domain Conferencing"/>
    </browse>

    <!--

    a-domain.com not open for self-service new user accounts

    <register notify="yes">
      <instructions/>
      <name/>
      <email/>
    </register>

    -->

    <welcome>
      <subject>Welcome!</subject>
      <body>Welcome to the Jabber server at a-domain.com</body>
    </welcome>

    <admin>
      <write>admin@a-domain.com</write>
      <reply>
        <subject>Auto Reply</subject>
        <body>This is a special administrative address.</body>
      </reply>
    </admin>
```

```
  </jsm>

  <load main="jsm">
    <jsm>./jsm/jsm.so</jsm>
    <mod_echo>./jsm/jsm.so</mod_echo>
    <mod_roster>./jsm/jsm.so</mod_roster>
    <mod_time>./jsm/jsm.so</mod_time>
    <mod_vcard>./jsm/jsm.so</mod_vcard>
    <mod_last>./jsm/jsm.so</mod_last>
    <mod_version>./jsm/jsm.so</mod_version>
    <mod_announce>./jsm/jsm.so</mod_announce>
    <mod_agents>./jsm/jsm.so</mod_agents>
    <mod_browse>./jsm/jsm.so</mod_browse>
    <mod_admin>./jsm/jsm.so</mod_admin>

    <!-- No filter service for a-domain.com
    <mod_filter>./jsm/jsm.so</mod_filter>
    -->

    <mod_offline>./jsm/jsm.so</mod_offline>
    <mod_presence>./jsm/jsm.so</mod_presence>
    <mod_auth_plain>./jsm/jsm.so</mod_auth_plain>
    <mod_auth_digest>./jsm/jsm.so</mod_auth_digest>
    <mod_auth_0k>./jsm/jsm.so</mod_auth_0k>
    <mod_log>./jsm/jsm.so</mod_log>
    <mod_register>./jsm/jsm.so</mod_register>
    <mod_xml>./jsm/jsm.so</mod_xml>
  </load>

</service>
```

We can see that this configuration file contains the definition of a jsm component instance. The instance is identified with the name `sessions.a-domain` and the host `a-domain.com` has been registered as what the jsm listens for - its 'external identification'.

We can also see that the literal texts in the descriptions and welcome message are specific to a-domain.com, the administration section in the configuration describes a local user at a-domain.com as the administrator, the new user registration facility has been disabled, and that the `mod_filter` service has been commented out from the list of loaded modules in the component connection definition.

There is one service listed in the browse section - the *Conferencing* service, with the JID `conference.a-domain.com`; this is the service that's defined in the file `./config/a-domain/conference.xml`, which itself is specified in a `<jabberd:include/>` tag in the main `jabber.xml` alongside this `sessions.xml` file.

Taking a look at this *Conferencing* service definition for a-domain.com in the `./config/a-`

`domain/conference.xml` file, we see:

```
<service id='conf.a-domain'>
  <host>conference.a-domain.com</host>
  <load><conference>./conference-0.4.1/conference.so</conference></load>
  <conference xmlns="jabber:config:conference">
    <public/>
    <vCard>
      <FN>a-domain Chatrooms</FN>
      <DESC>This service is for public chatrooms.</DESC>
      <URL>http://www.a-domain.com/chatrooms</URL>
    </vCard>
    <history>10</history>
    <notice>
      <join> is here</join>
      <leave> has left</leave>
      <rename> is now known as </rename>
    </notice>
    <room jid="bar@conference.a-domain.com">
      <name>The Bar</name>
    </room>
  </conference>
</service>
```

Similar to the `./config/a-domain/sessions.xml` content, here we see a-domain.com specific definitions - crucially the service identification as `conf.a-domain` and the `<host/>` tag declaring the hostname that this service serves under.

Configuration for b-domain.com

Now that we've seen the a-domain specific XML, let's have a look at the b-domain specific XML:

```
<service id="sessions.b-domain">
  <host>b-domain.com</host>
  <jsm xmlns="jabber:config:jsm">

    ...

    <vCard>
      <FN>b-domain Jabber Server</FN>
      <DESC>Jabber 1.4.1 on b-domain.com</DESC>
      <URL>http://www.b-domain.com/</URL>
    </vCard>

    <browse>
      <conference type="public" jid="conference.b-domain"
```

```
                name="b-domain Conferencing"/>
    <service type="jud" jid="jud.b-domain" name="b-domain JUD">
      <ns>jabber:iq:search</ns>
      <ns>jabber:iq:register</ns>
    </service>
  </browse>

  <register notify="yes">
    <instructions>
       Choose a username and password to register with this server.
    </instructions>
    <name/>
  </register>

  <welcome>
    <subject>Welcome!</subject>
    <body>Welcome to the Jabber server at b-domain</body>
  </welcome>

  <admin>
    <read>info@b-domain</read>
    <write>service@b-domain</write>
    <reply>
      <subject>Auto Reply</subject>
      <body>This is a special administrative address.</body>
    </reply>
  </admin>

</jsm>

<load main="jsm">
  <jsm>./jsm/jsm.so</jsm>
  <mod_echo>./jsm/jsm.so</mod_echo>
  <mod_roster>./jsm/jsm.so</mod_roster>
  <mod_time>./jsm/jsm.so</mod_time>
  <mod_vcard>./jsm/jsm.so</mod_vcard>
  <mod_last>./jsm/jsm.so</mod_last>
  <mod_version>./jsm/jsm.so</mod_version>
  <mod_announce>./jsm/jsm.so</mod_announce>
  <mod_agents>./jsm/jsm.so</mod_agents>
  <mod_browse>./jsm/jsm.so</mod_browse>
  <mod_admin>./jsm/jsm.so</mod_admin>
  <mod_filter>./jsm/jsm.so</mod_filter>
  <mod_offline>./jsm/jsm.so</mod_offline>
  <mod_presence>./jsm/jsm.so</mod_presence>
  <!--

  zero-knowledge authentication only
```

```
    <mod_auth_plain>./jsm/jsm.so</mod_auth_plain>
    <mod_auth_digest>./jsm/jsm.so</mod_auth_digest>
    -->

    <mod_auth_0k>./jsm/jsm.so</mod_auth_0k>

    <mod_log>./jsm/jsm.so</mod_log>
    <mod_register>./jsm/jsm.so</mod_register>
    <mod_xml>./jsm/jsm.so</mod_xml>
  </load>

</service>
```

We can see here that we've fulfilled our requirements of the virtual server for b-domain - registration is open but authentication is limited to zero-knowledge, and the services offered in the `<browse/>` list are unique to b-domain. [3]

The *Conferencing* and JUD services associated with the b-domain.com hostname will be configured in a similar way to how the *Conferencing* service was configured in `./config/a-domain/conference.xml` for a-domain - crucially again the service ids will be unique and the `<host/>` tags will be specific to b-domain.com.

As long as each component instance is uniquely identified and you have used separate hostname definitions, 'real' virtual Jabber servers *all listening to the same Jabber standard client port of 5222 on a single host* can be, er, a reality.

Splitting up the Jabber Server Processes

As well as being possible to lump multiple Jabber server identities in the form of virtual hosting onto a single Jabber server and its corresponding monolithic process, it is also possible to go in the opposite direction and split up a single Jabber server into multiple processes. These processes interact through TCP socket connections and so it's possible for them to run on the same or different physical hosts.

How is this achieved? Well, revisiting the ideas from the start of this chapter, we consider that a Jabber server is a daemon (**jabberd**) and a set of components that provide the services. Taking one step away from the 'classic' Jabber server model which contains components such as the ones described in the section called *An Overview of the Server Architecture* we can imagine a Jabber server where **jabberd** controls just one component, say the *Conferencing* component.

How much use is a Jabber server with a single *Conferencing* component? Not much. But *linked together with another* Jabber server, we can see that this is a way to split off components and run them independently.

Taking the *Conferencing* component as an example candidate for ostracism, let's have a look at what we need to do.

Define the Configuration for the Satellite Server

This is very straightforward - we've seen *Conferencing* configuration before so we'll shorten it a bit here:

```
<jabber>

  <service id='conf.yak'>
    <host>conference.yak</host>
    <load><conference>./conference-0.4.1/conference.so</conference></load>
    <conference xmlns="jabber:config:conference">

      ...

    </conference>
  </service>

</jabber>
```

This is the entirety of the configuration file so far for the satellite server - there's only one component instance - with an id of "conf.yak". Notice that the only other tag pair is the file-wide `<jabber> ... </jabber>`. Let's call it `jabber_conf.xml`.

Open a Connection Point in the Main Server

We've already seen a mechanism earlier in this chapter in the section called *Component Connection Methods* that allows external components to connect into the Jabber server backbone by exchanging XML streams in the `jabber:component:accept` namespace. This is the TCP socket connection method.

We can prepare a connection point to the main Jabber server by specifying a component connection like this:

```
<service id="conflinker">
  <host>conference.merlix</host>
  <accept>
    <ip>127.0.0.1</ip>
    <port>9001</port>
    <secret>confsecret</secret>
  </accept>
</service>
```

in the configuration for the main Jabber server.

There's no real difference between this XML and the XML shown in the `<accept/>` example earlier in this Chapter. The clue lies in the service id, which has been defined as "conflinker". There's nothing special about the name; it simply gives the administrator a hint that there's some sort of link to a conference service from this point.

We're specifying acceptance of connections on IP address 127.0.0.1 - the same host as this main server, but it could just as easily be the IP address assigned to a network card, so that the connection could be made from a satellite server on a separate host.

List the Service Definition in <browse/>

While we're editing the main server's XML, we should add an entry for our satellite conference service:

```
<browse>

  ...

    <conference type="public" jid="conference.yak" name="yak Conferencing"/>

  ...

</browse>
```

The jid defined here must match the host defined in the *Conferencing* component instance definition in the satellite server configuration.

Add a Connector Mechanism to the Satellite Server

Now we've opened up a connection point in the main server, we need to add some corresponding configuration to the satellite server's XML - the 'plug' that will attach to the connection point on the main server:

```
<jabber>

  <service id="conflinker">
    <uplink/>
    <connect>
      <ip>127.0.0.1</ip>
      <port>9001</port>
      <secret>confsecret</secret>
    </connect>
  </service>

  <service id='conf.yak'>
    <host>conference.yak</host>
    <load><conference>./conference-0.4.1/conference.so</conference></load>
    <conference xmlns="jabber:config:conference">

      ...

    </conference>
  </service>
```

```
</jabber>
```

This new service - the 'plug' - with an id of "conflinker" (to match the id of the corresponding 'socket' in the main server) contains two elements - the `<connect/>` tag which corresponds to the `<accept/>` tag in the main server's configuration, and the `<uplink/>` tag, which serves as a conduit for all types of packets - those handled by each of the three delivery trees *log*, *xdb* and *service*. [4]

While we're looking at the satellite server's configuration again, it's worth pointing out that even in a situation where the satellite server process would be running on a separate host (we're running it here on the same host - hence the localhost IP address of 127.0.0.1 in the `<accept/>` tag), the value of the conference service's host filter is still `conference.yak` - that is the name of the host where the satellite server runs is actually irrelevant. This is because the conference service is still seen by the main Jabber server's **jabberd** as 'local', through the accept/connect binding.

Specify a Different PIDfile location

If the satellite server is going to be running on the same host as the main server, and from the same directory (indeed, in this example, we've named the satellite server's configuration file '`jabber_conf.xml`' to distinguish it from the main server's '`jabber.xml`'), make sure a different location for storing the PIDfile is specified:

```
<jabber>

  <service id="conflinker">

    ...

  </service>

  <service id='conf.yak'>

    ...

  </service>

  <pidfile>jabber_conf.pid</pidfile>

</jabber>
```

Start up the Main Server

Once everything is configured, start up the main server:

```
yak:~/jabber-1.4.1$ ./jabberd/jabberd -c jabber.xml &
```

The `<accept/>` section should start listening on port 9001 for a connection.

```
yak:~/jabber-1.4.1$ netstat -an | grep 9001
tcp        0      0 127.0.0.1:9001          0.0.0.0:*               LISTEN
```

Start up the Satellite Server

Now it's time to start up the satellite server. From the same directory in this example:

```
yak:~/jabber-1.4.1$ ./jabberd/jabberd -c jabber_conf.xml &
```

The satellite server should make a connection to the socket listening on 127.0.0.1:9001.

At this stage, you should have Jabber server services split between a main process and a separate process that runs a *Conferencing* component.

At the risk of stating the obvious, it is worth pointing out that this example shows that simply starting **jabberd** does not mean that any process will bind to and start listening on port 5222. It is the *jsm* component that makes this happen. So starting a second **jabberd** on the same host did not cause any socket listening problems because this second **jabberd** doesn't have a jsm component and so doesn't try to bind to port 5222.

Using Services on Other Jabber Servers

This section describes a technique that we've already seen used implicitly in the section called *Splitting up the Jabber Server Processes*. That is the use of services on *other* Jabber servers. In reality, the example of running a *Conferencing* module in a satellite Jabber server showed the technique in the context of local administrative control - we control the main and and satellite servers, and the module in the satellite server may rely on services in the main server for support.

But consider the `<browse/>` section in the `jabber.xml` configuration file that comes out of the box with Jabber server 1.4.1:

```
<browse>

  ...

  <service type="jud" jid="users.jabber.org" name="Jabber User Directory">
    <ns>jabber:iq:search</ns>
    <ns>jabber:iq:register</ns>
  </service>

  ...

</browse>
```

What's this? A JID of `users.jabber.org`? How many Jabber server installations will be running with the `jabber.org` domain name? Yes, just one. So what this means is that this `<browse/>` section is pointing to a *JUD* component running at `jabber.org` as `users.jabber.org`. And if *our* Jabber server is running the *Server (to Server) Connections* and *Hostname Resolution* components, clients connecting to our server can transparently jump across the wire and avail themselves of the JUD services at `users.jabber.org`.

Indeed, the entry doesn't even have to be in the `<browse/>` section - this is more for convenience, so that the clients can build a dynamic list of services from which the user may choose. The client may of course offer a facility for the user to directly enter the name (hostname, address) of the service they require.

How does this procedure compare to the 'satellite server' procedure? In this case, the packets that originate from a Jabber client connected to our Jabber server make their way to the JUD service on `users.jabber.org` by means of the s2s service. That is, they travel through a connection described by the `jabber:server` namespace. On the other hand, packets on our server destined for a satellite conference service travel through a connection described by the `jabber:component:accept` namespace.

Notes

[1] If the *Hostname Resolution* component is configured to run, then you will see two processes at startup, as the component - `dnsrv` - forks to have the resolver functionality run in a separate child process.

[2] 'Constellation' is used in German for the word 'configuration' when referring to system arrangement in IT. It's such a wonderfully evocative and appropriate word - much more interesting than 'configuration' - that I've decided to use it here.

[3] That said, it *is* possible for someone registered to b-domain to connect to and use, say, the *Conferencing* service listening on `conferencing.a-domain.com` - see the section called *Using Services on Other Jabber Servers* later in this Chapter.

[4] Although the service ids do not *need* to match.

II. Putting Jabber's Concepts to Work

Table of Contents

Chapter 5. Jabber Technology Basics

Table of Contents

One of Jabber's strengths is its simplicity. Neither the technology employed to build Jabber networks nor the protocol used to facilitate conversations within those networks is complicated.

The aim of this Chapter is to give you a good grounding in the technology and the protocol. In Chapter 1 we likened Jabber to chess: a small set of rules but boundless possibilities. And indeed it is the case. In this Chapter we cover identification within Jabber—how entities are addressed. Related to identity is the concept of *resources*; we look at how that relates to addressing, as well as it's relationship to *presence* and *priority*.

The Jabber protocol is constructed in XML, which is streamed between endpoints. We look at the details of these XML streams, and see how they're constructed.

And then there's the protocol itself. Comprising of surprisingly few fundamental elements, the Jabber protocol is small but perfectly formed. We'll review each element in detail.

With this Chapter under your belt, there's nothing else that's fundamental to Jabber that you must learn. Everything else is strategy, planning, and end-games.

Jabber Identifiers

An entity is anything that can be addressed in Jabber. A server, a component, a user connected with a client—these are all addressable entities. Every entity is identifiable by a *Jabber ID*, or JID. These JIDs give these entities their *addressability*. This is what a typical JID looks like:

```
qmacro@jabber.org/Laptop
```

This JID represents a user, connected to Jabber on a particular client. We can look at this JID in a more abstract way, by identifying its component parts:

```
username@hostname/resource
```

The `username` is separated from the `hostname` with an `@` symbol, and the `resource` is separated from the `hostname` with a slash (`/`).

It's quite likely that the JIDs that you may have encountered so far are those representing users' connections, such as the `qmacro@jabber.org/Laptop` example above. This is not the only sort of entity that JIDs are used to represent. As a Uniform Resource Locator (URL) is fundamental to the HyperText Transport Protocol (HTTP), so a JID is fundamental in Jabber. JIDs are used to represent not only users connected to Jabber via their clients, but every single entity in the Jabber universe that is to be addressed—in other words, that is to be the potential recipient of a message. Before looking at the restrictions that govern how a JID might be constructed (these restrictions are described in the section called *Rules and Regulations*), let's first have a look at some examples where a JID is employed to give entities their addressability:

A Jabber server

> A Jabber server is identified by a JID which doesn't contain a `username`. For basic addressing, the JID is simply the `hostname`:
>
> ```
> jabber.org
> ```
>
> To address specific features of the server, a `resource` is often specified and reflects the feature being addressed:

```
jabber.org/admin
```

The JID `jabber.org/admin` is used by server administrators at *jabber.org* to obtain a list of online users.

Administrators can send an announcement to all online users on the Jabber server `yak` by sending a message to the JID `yak/announce/online`.

```
yak/announce/online
```

In this case, the resource is `announce/online`. The first slash in the JID is interpreted as the *separator*; the second slash is simply part of the *resource name*.

Unique identification of Jabber software

Jabber clients can make a request for information on new versions of themselves by sending a special packet to an update server which manages a software version database. The packet they send is a presence packet (see later in this Chapter for an explanation of packet types) to a JID that takes this form:

```
959967024@update.jabber.org/1.6.0.3
```

In this case, the important part of the JID is the hostname—`update.jabber.org`—which is the Jabber server where the presence packet is destined. The username (`95996702`) is used to represent the unique identification of the client software requesting version information, and the resource (`1.6.0.3`) is set to be the current version of the client software.

A conference room

Jabber has a conferencing component that provides group chat facilities akin to IRC. Where IRC has channels, the conferencing component offers *rooms*. These rooms are addressed with JIDs in this form:

```
jdev@conference.jabber.org
```

The room name is specified in the `username` portion of the JID, and the `hostname` reflects the address of the conference component.

Browsing entities

Browsing is a powerful *hierarchical navigation* feature in Jabber. When a browse request is sent to an entity, that entity may return various pieces of information which reflects its component parts—how it's made up, what services it offers, what features it has, and so on.

The browse request is addressed to the entity via its JID, and the component parts that are

returned in response are all identified with JIDs too. If we address a browse request to the JID `yak/admin`, we receive a list of online users. This is shown in Example 5-1.

Example 5-1. Querying the server `yak` for online users

```
SEND: <iq type='get' to='yak/admin'>
        <query xmlns='jabber:iq:browse'/>
      </iq>

RECV: <iq type='result' to='dj@yak/console' from='yak/admin'>
        <item name='Online Users (seconds, sent, received)'
              xmlns='jabber:iq:browse' jid='yak/admin'>
          <user name='dj (548, 18, 15)' jid='dj@yak'/>
          <user name='john (535, 11, 13)' jid='john@yak'/>
          <user name='jim (488, 15, 17)' jid='jim@yak'/>
        </item>
      </iq>
```

The JID `yak/admin` represents an administrative function in the Jabber server `yak`; it *identifies* the place—the service entry point—by Jabber address, where this information can be retrieved.

Example 5-1 shows how pervasive the JID is as a mechanism for identification within Jabber. How we might use the information returned to us is not relevant at this point; the key thing to note is that the hooks used in conversations to jump from one point to another, to refer to other entities—services, users, transports, call-hooks into a server to obtain specific information—take the form of JIDs. Each one of the highlighted attribute values in the example is a JID. [1]

Taking another example from the conferencing area, JIDs are used to represent those present in a room in an abstracted way. Each room participant has an identity specific to that room:

```
jdev@conference.jabber.org/bd9505f766f98bd559d4c2d8a9d5ae78e3a7bbf5
```

As before, the room itself is represented by the `username@hostname` parts of the JID—in this case it's the Jabber Developers room hosted on `conference.jabber.org`. The `resource` is the long hex number which represents an individual room participant. It's a hexadecimal SHA1 message digest of their JID, designed to be unique, and calculated and assigned by the conferencing component as a user enters the room. [2]

Components, Hostnames, and Users

In the *Client software identification* example of a JID being used to carry software version information, we have a presence packet addressed to a JID using the following form:

```
959967024@update.jabber.org/1.6.0.3
```

But why doesn't the presence packet end up getting sent to a user called `959967024`? The short answer is because the Jabber Session Manager (JSM) component isn't running at `update.jabber.org`.

Instead, the server is running a special component that provides a version information service, and has no concept of user sessions as such. This component receives the presence packet—which doesn't go any further (i.e., it isn't punted on to somewhere else)—and then inspects the username and resource before performing the database lookup to see if their software needs to be updated.

So we see that just because a JID might have something defined for the `username@` part, it doesn't necessarily mean there's a user at the end of the line. It just serves as a carrier of unique information embedded in the JID to whichever component is listening for packets to the `hostname`.

As you can see, JIDs are flexible identifiers used throughout Jabber to give addressability to various entities. In the context of the JSM and user management, the address structure `username@hostname` has many parallels with email addressing, and indeed not without reason. In the context of individual users, an email address represents a user on a specific email server. This server is their "home", their mailbox to which everything addressed to their email address is routed. Different email users have different home mailboxes. In the same way, the JIDs of different Jabber users reflect each user's home Jabber server, to which everything addressed to their JID is routed. A message addressed by a user based on one Jabber server to a user based on another Jabber server is automatically routed from the one server to the other.

Rules and Regulations

A JID *must* contain a `hostname` part to be valid. The `username` and `resource` parts are optional; circumstance and usage dictates where either of these parts are necessary. A `username` is specific to the hostname that it's paired up with. For example:

```
qmacro@jabber.org
```

is not the same as:

```
qmacro@jabber.com
```

There are some restrictions on how each JID part is composed; Table 5-1 details these restrictions. Although you can be particular about the case of letters in a `username`, any operations (such as comparisons) at the Jabber server are case-insensitive. For example, if a user has registered `dj` as their username, then another user cannot register with the username `DJ`. However, the person who registered as `dj` can connect and send `DJ` when he authenticates, and for the duration of that session will be known as `DJ` not `dj`.

On the other hand, resources *are* case-sensitive.

Table 5-1. JID restrictions

JID part	Restrictions
`username`	A `username` can be up to 255 characters in length, and may not contain any ASCII character under 33 (decimal), [a] nor can it contain any of the characters `:`, `@`, `"` or `'`; also whitespace (tabs, newlines, and carriage-returns) and control characters are forbidden.
`hostname`	The same restrictions apply here as for normal DNS hostnames.
`resource`	There are no restrictions for the `resource` part of a JID.
Notes: a. that is, it may not contain spaces or those considered to be control characters	

Notes

[1] This administrative information about online users on a Jabber server can be retrieved by sending the IQ-get element shown in the example. However, the information is only forthcoming if the user making the request—sending the IQ-get element—is *administrative* user. See the the section called *Administration* in Chapter 4 for details on administrative users.

[2] This is to shield the participant's real identity, which is the default setting for a conference room.

Resources and Priority

In the previous section, we saw how the `resource` is used to “qualify” certain queries to a servername, to hold information such as version numbers, and to represent users in a conference room. However, the resource is traditionally seen as a way of making a distinction between simultaneous connections by a user to the same Jabber server. For example, if you connect to a Jabber server using the same username and password on three different machines (or "resources"), the Jabber server will look at the resource part of the JID to determine which client to route messages to.

For the purpose of this example, let's say that the three resources are a workstation, laptop, and a PDA. Each client is connected to the same Jabber server, so the `resource` part of the JID can be used to distinguish between the three connections. They could also be used to differentiate between the three connections coming from the same client host.

The classic explanation serves us well here: In a work situation, I might be connected to my Jabber server using a client on my workstation. I might also be connected, with the same `username`, to my Jabber server on my laptop that's sitting next to my workstation. Furthermore, I might have a handheld device that runs a small Jabber client, that I'm connected with too.

On each client machine, I'm connecting using the same credentials (username and password) to the same Jabber server. So the `resource` part of a JID can be used to distinguish between my three connections. The three 'resources' are my workstation, laptop and handheld, in this example. In another example, they might be used to differentiate between three different connections coming from the same client host (say, my desktop).

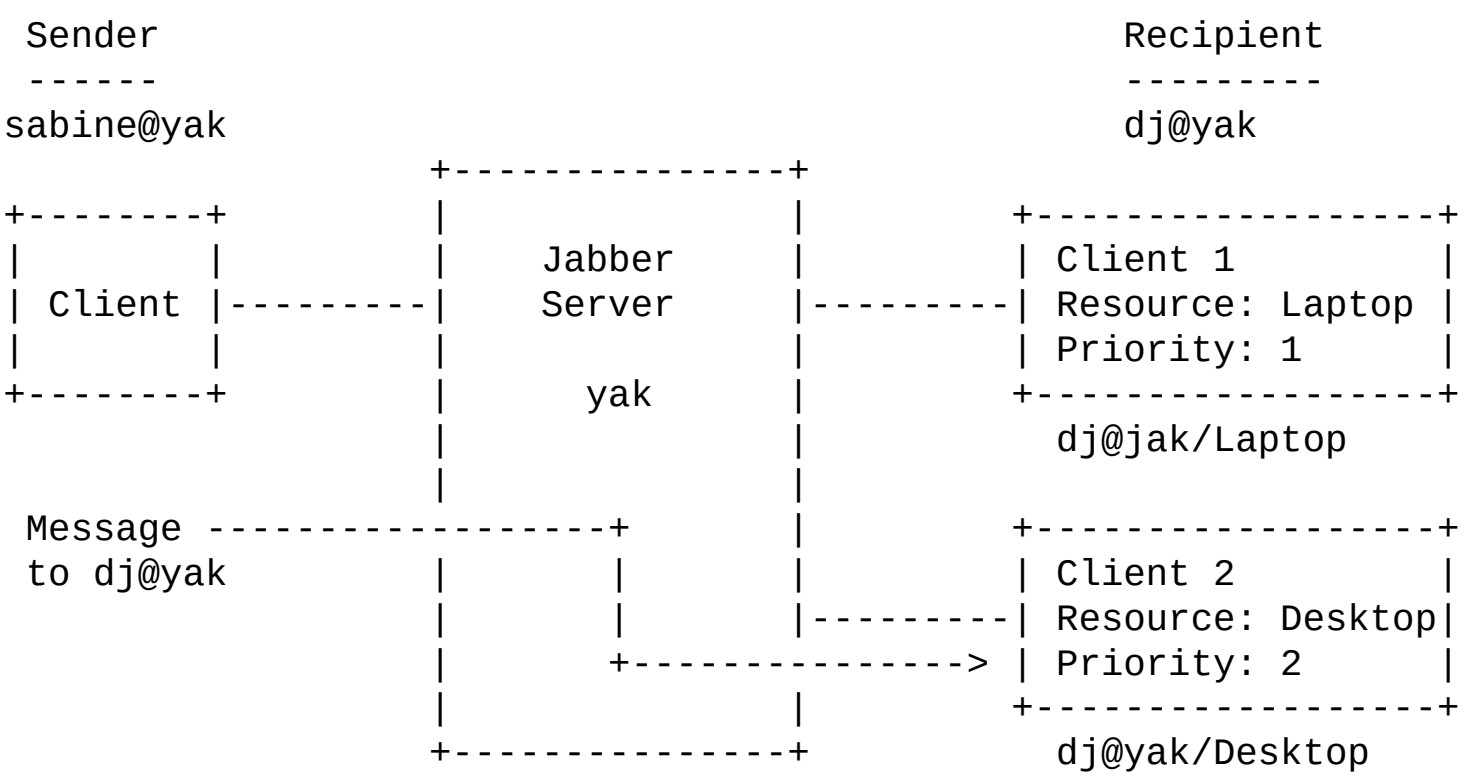
The `resource` part of a JID allows a user to be connected to Jabber (specifically the JSM, which manages users and sees user sessions as separate entities) multiple times.

"But", I hear you say, "what happens when someone sends you a message? To which client is the message sent?"

This is where the concept of connection *priority* comes to our aid. Each Jabber client connection can be given a priority. When a user has more than one concurrent connection to a Jabber server, the priority is used to determine to which connection any messages intended for that user should be sent. [1] The connection with the highest priority value is the connection to which the messages are sent. [2]

Figure 5-1 shows priority in action. In this example, Sabine's message is sent to the Jabber client on the Desktop, as it has a higher priority. Note that with Jabber priority, 1 has a lower priority than, say, 5. The higher the number, the higher the priority.

Figure 5-1. Resources, Priority, and Message Delivery



If there is a priority tie, for example if I have connected to my Jabber server with a client on my laptop (where I specified the resource “Laptop”) and also with a client on my desktop (where I specified the resource “Desktop”), and the settings in *both* clients have their priorities set to the value 1, the *most recent* connection to be made wins and receives the messages.

It is also possible to direct messages to a particular client. Taking the example from Figure 5-1, if `sabine@yak` were to specify `dj@yak/Laptop` instead of `dj@yak` as the recipient for a message, her message would go to Client 1 (`Laptop`), not Client 2 (`Desktop`), despite Client 1's lower priority value.

Priorities are specified when a user sends presence information. We will see this later in the section called *The Jabber Building Blocks*. It makes sense for the priority to be associated with a user's presence, rather than a user's client connection. For example, if the priority was specified at connection time, the user would have to disconnect and reconnect if she wanted to change priority. As it stands, she just has to send presence information containing a new priority value to change it. Figure 5-2 shows a WinJab client popup window used to change presence information. The value of the current priority can be changed.

Figure 5-2. Changing presence and priority in the WinJab client

Notes

[1] "Messages" is used in the generic sense; actually any packet destined to the user is implied. See the section called *The Jabber Building Blocks* in this Chapter for information on the types of packets that are send and received.

[2] Priority values must be a positive integer, and cannot be zero or less.

XML Streams

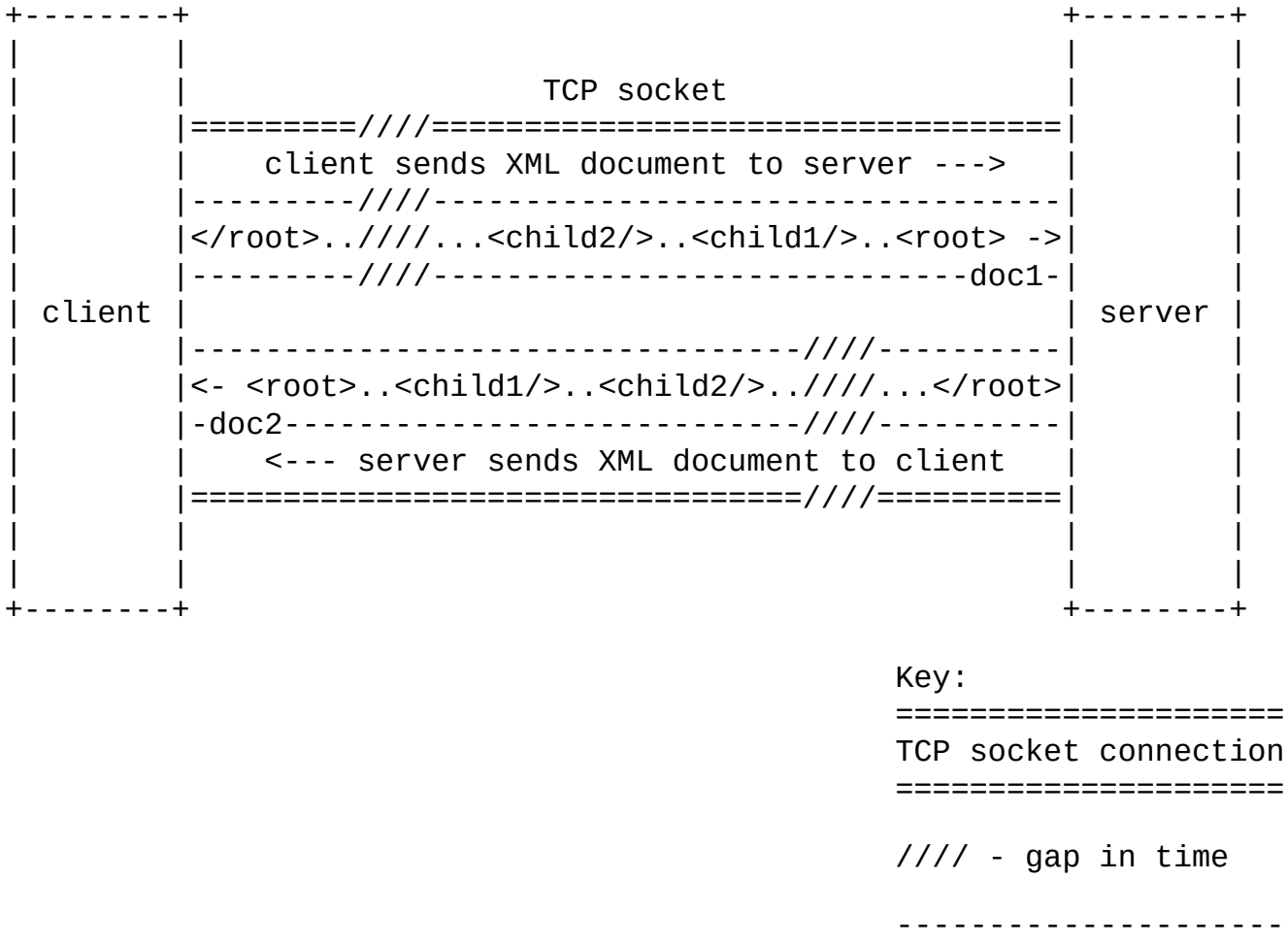
By now, you should already know that Jabber relies heavily on XML. XML courses through Jabber's veins; data sent and received between entities, and internally within the server itself, is formatted in XML *packets*.

However, the XML philosophy goes further than this. A connection between two Jabber endpoints, say, a client and a server, is made via a TCP socket, and XML is transferred between these endpoints. However, it's not just random fragments of XML flowing back and forth. There is a structure, a choreography, imposed upon that flow. The entire conversation that takes place between these two endpoints is embodied in a pair of XML documents.

The Conversation as XML Documents

The conversation is two-way, duplexed across a socket connection. One one side, the client sends an XML document to the server. On the other side, the server responds by sending an XML document to the client. Figure 5-3 shows the pair of XML documents being streamed across the TCP socket connection between client and server, over time.

Figure 5-3. Client<->Server conversation as a pair of streamed XML documents



```
                                        streamed XML document
                                        ---------------------
```

But what do we mean when we say that the conversation is an XML document? To answer this, consider this simple XML document:

```
<?xml version="1.0"?>
<roottag>
  <fragment1/>
  <fragment2/>
  <fragment3/>
  ...
  <fragmentN/>
</roottag>
```

The document starts with a document type declaration:

```
<?xml version="1.0"?>
```

which is immediately followed by the opening root tag. This root tag is significant because there can be only one (and, of course, its corresponding closing tag) in the whole document. In effect, it wraps and contextualizes the content of the document.

```
<roottag>
  ...
</roottag>
```

The real content of the document is made up of the XML fragments that come after the opening root tag:

```
<fragment1/>
<fragment2/>
<fragment3/>
...
<fragmentN/>
```

So, taking a connection between a Jabber client and a Jabber server as an example, this is exactly what we have. The server is listening on port 5222 for incoming client-initiated connections. Once a client has successfully connected to the Jabber server, it sends an XML document type declaration and the opening root tag to announce its intentions to the server, which in turn responds by sending an XML document type declaration and opening root tag of its own.

From then on, every subsequent piece of data that the client sends to the server over the lifetime of the connection is an XML fragment (`<fragmentN/>`). The connection can be closed by the client by sending the matching closing root tag. Of course, the connection can be also closed by the server by sending the closing root tag of *its* XML document.

The fragments sent within the body of the XML document are the XML building blocks on which Jabber solutions are based. These XML building blocks are introduced and examined in the next section.

Suffice it to say here that these fragments can come in any order within the body of the XML document, precisely because they're *in* the body. As long as an XML document has a root tag, and the fragments themselves are well-defined, then it

doesn't matter *what* the content is. Because of the way the document is parsed—in chunks, as it appears—it doesn't matter if the fragments appear over a long period, which is the case in a client/server connection where messages and data are passed back and forth over time.

It should be fairly easy now to guess why this section (and the technique) is called “XML Streams”. XML is *streamed* over a connection in the form of a document, and is parsed and acted upon by the recipient in fragments, as they appear.

The Opening Tag

Earlier, we said that the opening document tag was used by the client to “announce its intentions.” The following is a typical opening document tag from a Jabber client which has made a socket connection to port 5222 on the Jabber server `jabber.org`.

```
<stream:stream
    xmlns:stream="http://etherx.jabber.org/streams"
    to="jabber.org"
    xmlns="jabber:client">
```

There are four parts to this opening tag.

The `<stream:stream>` tag itself

```
<stream:stream>
```

Every streaming Jabber XML document must start, and end, with a tag named `stream`, qualified with the `stream` namespace.

The `stream` namespace declaration

```
xmlns:stream="http://etherx.jabber.org/streams"
```

The declaration of the `stream` namespace also comes in the opening `stream` tag. It refers to a URL (`http://etherx.jabber.org/streams`) which is a fixed value, and serves to uniquely identify the `stream` namespace used in the XML document, rooted with `<stream/>`, that is streamed over a Jabber connection.

The namespace qualifies only the tags that are prefixed "`stream:`". Apart from `stream`, there is one other tag name used in these documents that is qualified by this namespace, and that is `error`. The `<stream:error/>` tag is used to convey Jabber XML stream connection errors, such as premature disconnection, invalid namespace specifications, incomplete root tag definitions, a timeout while waiting for authentication to follow the root tag exchange, and so on.

The `to` attribute

```
to="jabber.org"
```

There is a `to` attribute that specifies to which Jabber server the connection is to be made, and where the user session is to be started and maintained.

We've already specified the `jabber.org` hostname, representing our Jabber server, when defining the socket

connection (`jabber.org:5222`), so why do we need to define it again here? As indicated by the `to` attribute, you can see that we've made a *physical* connection to the `jabber.org` host. However, there may be a choice of *logical* hosts running within the Jabber server to which our client could connect. When making the *physical* connection from our client to the Jabber server, we defined the hostname `jabber.org` for our socket connection (to `jabber.org:5222`). Now we're connected, we're specifying `jabber.org` again as the `logical` host to which we want to connect *inside* Jabber. This is the logical host identity within the Jabber server running on the `jabber.org` host.

This "repeat specification" is required, because there's a difference between a *physical* Jabber host, and a *logical* Jabber host. In the section called *A Tour of `jabber.xml`* in Chapter 4 we see how a single Jabber server can be set up to service user sessions (with one or more JSMs) that are each identified with different logical hostnames. This is where the *physical*/*logical* hostname distinction comes from, and why it's necessary to specify a name in the root `<stream:stream>` tag's `to` attribute.

It just so happens that in the example of an opening tag that we've used, that the *logical* hostname is the same as the *physical* one—`jabber.org`. In many cases, this will be the most commonplace. However, an Internet Service Provider (ISP), for example, may wish to offer Jabber services to its customers and dedicate a single host for that purpose. That host has various DNS names, which all resolve to that same host IP address. Only one Jabber server is run on that host. (If a second server were to be installed, it would have to listen on different—non-standard— ports, which would be less than ideal.) To reflect the different names under which it would want to offer Jabber services, it would run multiple JSMs under different *logical* names (using different values for each `<host/>` configuration tag, as explained in the section called *A Tour of `jabber.xml`* in Chapter 4). When connecting to that Jabber server, it may well be that the *logical* name specified in the opening tag's `to` attribute would be different to the *physical* name used to reach the host in the first place.

The namespace of the conversation

```
xmlns="jabber:client"
```

In addition to the namespace that qualifies the `stream` and `error` tag names—which could be seen as representing the "outer shell" of the document, the `xmlns` attribute specifies a namespace which will qualify the tags in the body of the document; the conversation fragments of XML which will appear over time. This namespace is `jabber:client` and signifies that the *type* of conversation that is about to ensue over this document connection is a *client (to server)* conversation.

This namespace specification is required because a client connection is just one type of connection that can be made with a Jabber server, and different connections carry conversations with different content. Table 5-1 lists the conversation namespaces currently defined in the Jabber protocol.

Table 5-2. Conversation Namespaces

Namespace	Description
`jabber:client`	This is the namespace that qualifies a connection between a Jabber client and a Jabber server.
`jabber:server`	This namespace qualifies a connection between two Jabber servers. *Dialback* (host verification mechanism) conversations take place within the `jabber:server` namespace.

`jabber:component:accept`	When an external program connects to a Jabber server via a TCP sockets connection, this namespace is used to qualify the pair of XML documents exchanged over the connection.
`jabber:component:exec`	When an external program connects to a Jabber server via a STDIO connection, this namespace is used to qualify the pair of XML documents exchanged over such the connection. [a]
Notes: a. For more details on external program connections to Jabber, see Chapter 4.	

The Response

To complete our initial look at XML streams in a Jabber client-server conversation, let's have a look at what the Jabber server might send in response to the opening tag from the client:

```
<stream:stream
    xmlns:stream='http://etherx.jabber.org/streams'
    id='3AFD6862'
    xmlns='jabber:client'
    from='jabber.org'>
```

There are a couple of differences between this opening tag from the server and the opening tag from the client. That is, above and beyond the fact that this response's opening tag is for a document that is going to be streamed along the socket in the opposite direction to that of document to which the the request's opening tag belongs. The first difference is that there's a `from` attribute instead of a `to` attribute. The second difference is that there's an extra attribute—`id`. Let's look at these in turn.

The `from` attribute

```
from="jabber.org"
```

The `from` attribute is fairly straightforward; it normally serves to confirm to the client that the requested logical host is available. If the host is available, the value of the `from` attribute from the server will match the value of the `to` attribute from the client. However, in some circumstances the value can be different. The value sent in the `from` attribute is a *redirection*, or *respecification*, of the logical host by which the Jabber server (or more specifically the JSM component within the Jabber server) is actually known.

Logical host aliases can be defined in the Jabber server's configuration to “convert” a hostname specified in the incoming `to` attribute. The `<alias/>` tag, which is used to define these logical host aliases, is described in the section called *Component Instance: c2s* in Chapter 4. But how are these hostname conversions used? Here's an example…

Let's say that you're running a Jabber server on an internal network that doesn't have an available DNS server. The host where the Jabber server runs is called `apollo`, and its IP address is `192.168.1.4`. Some people will connect to the host via the hostname because they have it defined in a local `/etc/hosts` file; others will connect via the IP address. Normally, the hostname (or IP address) specified in the connection parameters given to a Jabber client will be:

- Used to build the socket connection to the Jabber server.
- Specified in the `to` attribute in the opening XML stream to specify the logical host.

If the JSM section of the Jabber server is defined to have a hostname of `apollo`:

```
<host><jabberd:cmdline flag='h'>apollo</jabberd:cmdline></host>
```

then we need to make sure that the Jabber client uses that name when forming any JIDs for that Jabber server (e.g., the JID `apollo` used as an addressee for an IQ browse request). Having this:

```
<alias to='apollo'>192.168.1.4</alias>
```

in our `c2s` instance configuration would mean that any incoming XML stream header with a value of `192.168.1.4` in the `to` attribute:

```
<stream:stream
    to="192.168.1.4"
    xmlns="jabber:client"
    xmlns:stream="http://etherx.jabber.org/streams">
```

would elicit the following response:

```
<stream:stream
    from='apollo'
    id='1830EF6A'
    xmlns='jabber:client'
    xmlns:stream='http://etherx.jabber.org/streams'>
```

which effectively says: "*Okay, you requested `192.168.0.4`, but please use `apollo` instead*." The client should use the value *confirmed* in the `from` attribute when referring to that Jabber server in all subsequent stream fragments.

Not specifying an `<alias/>` tag in this example would result in problems for the client. Without any way of checking and converting incoming hostnames, the `c2s` component will by default simply transfer the value from the `to` attribute to the `from` attribute in its stream header reply.

Following this thread to its natural conclusion, it's worth pointing out that if we have an `alias` specification like this:

```
<alias to='apollo'/>
```

then the value of the `from` attribute in the reply will *always* be set to `apollo` regardless of what's specified in the `to` attribute. This means that the `to` attribute could be left out of the opening stream tag. Although this serves well to illustrate the point, it is not good practise.

The `id` attribute

```
id='3AFD6862'
```

The `id` attribute is the ID of the XML stream, and is used in the subsequent authorization steps, which are described

in Chapter 6. The value is a random hexadecimal string generated by the server, and is not important per se. What *is* important is that it's a value that is random, and shared between server and client. The server knows what it is because it generated it, and the client knows what it is because the server sends it in the opening tag of the response.

The Simplest Jabber Client

Now that we know how a conversation with a Jabber server is started, let's try it ourselves. At a stretch, one could say that the simplest Jabber client, just like the simplest HTTP client, or the simplest client that has to interact with *any* server that employs a *text-based* protocol over a socket connection, is **telnet**.

Simply point **telnet** to Jabber server, specifying port 5222, and send an opening tag. You will receive an opening tag, from the server, in response:

```
yak:~$ telnet localhost 5222
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
<?xml version='1.0'?>
<stream:stream xmlns:stream='http://etherx.jabber.org/streams' to='yak'
xmlns='jabber:client'>
<?xml version='1.0'?><stream:stream xmlns:stream='http://etherx.jabber.org/streams'
id='3AFD839E' xmlns='jabber:client' from='yak'>
```

(If you haven't got a Jabber server to experiment with, see Chapter 3 on how to set one up.)

Using **telnet** is a great way to find out more about the way the Jabber protocol works. Perhaps the next thing to do is try out the user registration and authentication steps described in Chapter 6. But watch out—send some invalid XML the server will close the connection on you!

The Jabber Building Blocks

At this stage we've got a good impression of the structure of Jabber; what different elements make up a Jabber system, how entities in Jabber are addressed, and how communication between these entities is carried.

Now it's time to look at *what* gets carried—the fragments that we touched upon in the previous section. These fragments are the heart and soul of Jabber—the *lifeblood* that courses through Jabber's veins carrying information back and forth—these fragments in many ways define what Jabber is, what it stands for.

Surprisingly, when we look closely at these fragments, with Jabber's capabilities as a messaging platform in mind, we see that there are only three basic elements involved—`<message/>`, `<presence/>`, and `<iq/>`. Three different types of XML fragment, each with a different purpose. But with these three fragment types, all that Jabber promises, and more, can be achieved.

Now let's look at each of these Jabber elements in greater detail. But before we do, let's dive into the XML stream and pull out a handful of XML fragments to get us in the mood. Example 5-2 shows a chunk of conversation between a Jabber client and a Jabber server which occurred immediately after the connection and authentication stages.

Although any conversation between two Jabber entities is contained within two XML documents exchanged in streams, the traditional way to represent both documents at the same time is to use prefixes to show whether a fragment is being sent (`SEND:`) or received (`RECV:`), by one of the two entities. When appropriate, the perspective is taken from the viewpoint of the entity that's *not* the Jabber server; in the case of Example 5-2, the viewpoint is of the Jabber client.

Example 5-2. A Chunk of Conversation between a Jabber client and a Jabber server

```
SEND: <iq id='roster_0' type='get'>(1)
        <query xmlns='jabber:iq:roster'/>
      </iq>

RECV: <iq id='roster_0' type='result' from='dj@yak/Work'>
        <query xmlns='jabber:iq:roster'>
          <item jid='sabine@yak' name='sabine' subscription='both'>
            <group>Family</group>
          </item>
        </query>
      </iq>

SEND: <presence><status>Online</status></presence>(2)

      ... time passes ...
```

```
RECV: <message id='1' to='dj@yak' from='sabine@yak/winjab' type='chat'>(3)
        <thread>3FE7392DDCA919CB49C73A2FFCE9901D</thread>
        <body>Hello</body>
      </message>
```

Example 5-2 shows three different elements in action, described as follows:

(1)
: The user's (`dj@yak`'s) contact list is requested and sent back by the server (the `<iq/>` elements)

(2)
: The client broadcasts the user's availability (the `<presence/>` element)

(3)
: The user receives a message from `sabine@yak` (the `<message/>` element)

So, let's have a look at each of these elements, starting with arguably the most commonly occurring—`<message/>`.

The Message Element

It's obvious that in a messaging architecture such as Jabber, sending messages is fundamental. The `<message/>` element provides us with this facility. Any data, other than availability information or structured requests and responses (which are handled by the other two element types) sent from one Jabber entity to another, is sent in a `<message/>` element.

All things considered, the keyword in that last sentence is *in*. It's a good idea to regard Jabber elements as *containers*; the simile fits well as the elements themselves remain relatively static (save for the attributes) but the *content* can change to reflect the circumstances.

Message Attributes

The `<message/>` element is a container, or envelope, which appropriately requires some form or addressing. The attributes of the `<message/>` element serve this purpose.

type

: *Optional*

: Example: `<message type='chat'>`

: The Jabber protocol defines five different message *types*. The message type gives an indication to the recipient as to what sort of content is expected; the client software is then able, if it wishes, to handle the incoming message appropriately.

: **`type='normal'`**

: The `normal` message type is used for simple messages that are often one-time in nature, similar to an email

message. If I send you a message and I'm not particularly expecting a response, or a discussion to ensue, then the appropriate message type is `normal`.

Some clients handle `normal` message types by placing them in a sort of message inbox, to be viewed by the user when he so chooses. This is in contrast to a `chat` type message.

Note that the `normal` message type is the default. So if a message is received without an explicit `type` attribute, it is interpreted as being `normal`.

type='chat'

The `chat` message type differs from the `normal` message type in that it carries a message that is usually part of a live conversation, which is best handled in real-time with immediate responses—a `chat` session.

The handling of `chat` messages in many clients is done with a single window that displays all the `chat` messages both sent and received between the two parties involved. All the `chat` messages that belong to the same thread of conversation, that is. There's a subelement of the `<message/>` element that allows the identification of conversational threads so that the right messages can be grouped together; see the reference to `<thread/>` later in this section.

type='groupchat'

The `groupchat` message type is to alert the receiving client that the message being carried is one from a conference (`groupchat`) room. The user can participate in many conference rooms, and receive messages sent by other participants in those rooms. The `groupchat` type signifies to the receiving client that the addressee specified in the `to` attribute (see later in this section) is *not* the user's JID but the JID of the conference room whence the `groupchat` message originates.

type='headline'

This is a special message type designed to carry news style information, often accompanied by a URL and description in an attachment qualified by the `jabber:x:oob` namespace. Messages with their `type` set to `headline` can be handled by clients in such a way that their content is placed in a growing list of entries that can be used as reference by the user.

type='error'

The `error` message type signifies that the message is conveying error information to the client. Errors can originate in many places and under many circumstances. Refer to the description of the `<error/>` subelement later in this section for more details.

from

Set by server

Example: `<message` **`from='dj@yak/Desktop'`**`>`

The `from` attribute of the `<message/>` element shows the message originator's JID. In many cases this is the

JID of a user, but with the message type `groupchat` for example, it can be the JID of the conference room where the message was originally sent.

Note that the `from` attribute should not be set by the client. It is the Jabber server, to which the client where the message was originated is connected, that sets the attribute value. This is to prevent spoofing of JIDs. If a `from` attribute *is* set, it will be overriden by the server.

to

Optional

Example: `<message to='qmacro@jabber.org'>`

The `to` attribute is used to specify the intended recipient of the message, and is a JID. The recipient may be another Jabber user, in which case the JID will usually be in the form `username@hostname` (with an optional `/resource` if a message should be sent to a specific client connection), or it could be a Jabber server identity, in which case the JID will be in the form `hostname` with an optional `/resource` depending on the situation.

If no `to` attribute is specified, then the message will be directed back to the *sender*, or the server, depending on the circumstances. This may or may not be what you want.

(This is also the case with the `to` attribute for the `<iq/>` element, however it is not the case with the `<presence/>` element. [1] See the sidebar *Element Handling by the Jabber Server* for an explanation.)

Element Handling by the Jabber Server

When elements (packets) make their way over the `jabber:client` XML stream and arrive at the Jabber server, they're delivered to the JSM which provides many of the services associated with Jabber's IM features, such as roster management, presence subscription, offline storage and so on. Each packet received runs a gauntlet of handlers before being delivered to its ultimate destination specified by the value of the `to` attribute.

In some cases, a packet has no 'ultimate destination' and is deemed to have been 'handled' without reaching a final delivery point.

In the case of a simple `<message/>` packet with a JID specified in the `to` attribute, the packet will not be swallowed by a handler, but be delivered to that JID destination. In the case of a simple `<presence/>` packet without a `to` attribute (a normal notification of availability)

id

Optional

Example: `<message id='JCOM_12'>`

When a message is sent, and a reply is expected, it is often useful to give the outbound message an identifier.

When the recipient responds, the identifier is included in the response. In this way, the originator of the message can work out which reply corresponds to which original message.

At the Jabber server, this works because a reply is usually built from a copy of the original message, with the `from` and `to` attributes switched around. So the `id` attribute remains untouched and in place.

Message Subelements

While the `<message/>` element itself is a container for the information being carried, the subelements are used to hold and describe the information being carried. Depending on the circumstances and the message type, different subelements can be used.

subject

Optional

Example:

```
<message to='qmacro@jabber.org' from='piers@jabber.org/Home'>
  <subject>Time to meet?</subject>
  <body>What time to you want to meet this afternoon?</body>
</message>
```

The `<subject/>` subelement is used to set a message subject. Message subjects are not that common in `chat` type messages, but are more appropriate in `normal` type messages where the subject can be displayed in the style of a list of inbox items. This subelement is also used in `groupchat` type messages to set the subject (or `topic`) of a conference room.

body

Optional

Example:

```
<message to='qmacro@yak' from='john@yak' type='chat'>
  <body>Hey - got a minute?</body>
</message>
```

Naturally, the `<body/>` subelement carries the body of the message.

error

Optional

Example:

```
<message to='piers@pipetree.com/Home' from='qmacro@jaber.org' type='error'>
  <body>Are you there?</body>
  <error code='502'>Unable to resolve hostname.</error>
```

```
</message>
```

The <error/> subelement is for carrying error information in a problem situation. In this example, the original message sent by piers@pipetree.com was a simple “Are you there?” to what he thought was qmacro's id on the Jabber server at jabber.org. However, the to attribute was specified incorrectly (jaber.org) and the Jabber server on pipetree.com wasn't able to resolve the hostname. So Piers receives his message back with an additional <error/> subelement, and the message type has been switched to error (the type='error' attribute).

The <error/> subelement carries two pieces of related information: an error number, specified in the code attribute, and the error text. Table 5-3 shows a list of the standard error codes and texts. The entity generating the error can specify a custom error text to go with the error code; if none is specified, the standard text as shown is used.

html

Optional

Example:

```
<message id="3" to="dj@yak" type="chat">
  <html xmlns="http://www.w3.org/1999/xhtml">
    <body>
      <span style="font-family: Arial; font-size: 10pt">
        This is really <em>nice!</em>
      </span>
      <br/>
    </body>
  </html>
  <body>This is really nice!</body>
</message>
```

The <html/> tag is for support of XHTML-formatted messages. The normal <body/> tag carries plain text; text formatted with XHTML markup can be carried in <message/> elements inside the <html/> subelement.

The markup must be qualified by the XHTML namespace http://www.w3.org/1999/xhtml (as shown in the example) and conform to the markup described in the XHTML-Basic specification defined at http://www.w3.org/TR/xhtml-basic.

Note that the content of the message must also be repeated in a normal <body/> subelement without formatting, to comply with the “lowest common denominator” support for different Jabber clients—not all of them will be able to interpret the XHTML formatting so will need to receive the message content in a way that they can understand.

The <html/> subelement effectively is a wrapper around a second, alternative, <body/> subelement.

thread

Optional

Example:

```
<message to='qmacro@jabber.org' type='chat'>
  <thread>B19217AFEEBDC2611971DD1E8B23AAE4</thread>
  <body>Yes, they're at http://docs.jabber.org</body>
</message>
```

The `<thread/>` subelement is used by clients to group together snippets of conversations (between users) so that the whole conversation can be visually presented in a meaningful way. Typically a conversation on a particular topic—a *thread*—will be displayed in a single window. Giving each conversation thread an identity enables a distinction to be made when more than one conversation is being held at once and `chat` type messages which are component parts of these conversations are being received (possibly from the same correspondent) in an unpredictable sequence.

Only when a new topic or branch of conversation is initiated must a client generate a thread value. At all other times, the correspondent client must simply include the `<thread/>` tag in the response. Here thread value is generated from a hash of the message originator's JID and the current time.

x

Optional

Example:

```
<message to='dj@yak' type='chat' from='sabine@yak/laptop'>
  <body>Hi - let me know when you get back. Thanks.</body>
  <x xmlns='jabber:x:delay' from='dj@yak' stamp='20010514T14:44:09'>
    Offline Storage
  </x>
</message>
```

The `<x/>` subelement is special. While the other subelements like `<body/>` and `<thread/>` are fixed into the Jabber building blocks design, the `<x/>` subelement allows `<message/>` elements to be extended to suit requirements. What the `<x/>` subelement does is provide an anchor-point for further information to be attached to messages in a structured way.

The information attached to a message is often called the *payload*. Multiple anchor-points can be used (as in the example) to convey multiple payloads, and each one must be *qualified* using a namespace.

Just as the content of XML streams is qualified by a namespace (one from the list in Table 5-2 earlier in this Chapter), so the content of the `<x/>` *attachment* must be qualified. There are a number of Jabber-standard namespaces that are defined for various purposes. One of these, `jabber:x:delay`, is used in the example. They are described in Chapter 5a. But there's nothing to stop you defining your own namespace to describe (and qualify) the data that you wish to transport in a `<message/>`. Namespaces beginning `jabber:` are reserved, anything else is OK.

Briefly, you can see how payloads are attached from the example. For every `<x/>` subelement, there's an `xmlns` attribute that qualifies it, and the data contained within the `<x/>` tag is formatted depending on the namespace.

In the example, the payload is carried in addition to the `<body/>` subelement. However, as the `<body/>` is actually *optional* in a message, it is possible to transmit structured payloads between Jabber entities without the need for “conventional” message content.

Table 5-3. Standard Error Codes and Texts

Code	Text
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Not Allowed
406	Not Acceptable
407	Registration Required
408	Request Timeout
409	Conflict
500	Internal Server Error
501	Not Implemented
502	Remove Server Error
503	Service Unavailable
504	Remove Server Timeout
510	Disconnected

The Presence Element

The `<presence/>` element is that which is used to convey a Jabber entity's availability. An entity can be *available*, which means that it's connected and any packets sent to it will be delivered immediately, or it can be *unavailable*, which means that it's not connected, and any packets sent to it will be stored and delivered the next time a connection is made.

For the large part, it is the entity itself, not the Jabber server to which it connects, that controls the availability information. The Jabber server will communicate an entity's *unavailability* if that entity disconnects from the server, but will only do that if the entity has communicated its *availability* beforehand.

Availability information isn't a free-for-all. Presence in Jabber is usually exchanged within a subscription mechanism. See the section called *Presence Subscription* for an explanation.

Presence is a `jabber:client` thing

It's worth noting that the *entities* referred to here are *client* entities. That is, clients (and therefore the users using those clients) connected to the Jabber server over an XML stream qualified by the `jabber:client` namespace (see the section called *XML Streams*, earlier). Presence is a feature made available within the Jabber Session Manager (JSM). External components that connect to the Jabber server backbone are separate from the JSM and therefore don't have any concept of presence. They are either connected to the backbone, or they're not.

Presence Attributes

The attributes of the `<presence/>` element are similar to those of the `<message/>` element.

type

Optional

Example: `<presence type='unavailable'>`

The `type` attribute of the `<presence/>` element is used for many purposes. The basic usage is to convey availability. Two values are used: `available` and `unavailable`. Another value is to signify that `<presence/>` packet is being used to query the packet recipient's presence (value is `probe`). The rest of the values (`subscribe`, `unsubscribe`, `subscribed`, `unsubscribed`) are used in the subscription structure which is described in the section called *Presence Subscription*.

`type='available'`

The `available` presence type is used by entities to announce their availability. This announcement is usually made to the Jabber server which manages the presence subscription mechanism (see the section called *Presence Subscription* for more details). However it can also be directed to a particular JID if the entity wants to control presence information itself.

The `available` presence isn't a simple binary “on/off”, there are varying degrees of availability which are specified using subelements of the `<presence/>` packet. These include `<show/>` and `<status/>`. These subelements are described next.

If no `type` attribute is specified, then this value of `available` is assumed. It makes sense, as the most common type of `<presence/>` packet sent by entities is usually the `available` type, optionally qualified with the `<show/>` and `<status/>` subelements, as the user of the connected client changes his circumstances over time (off for a break, back, out to lunch, and so on).

`type='unavailable'`

The `unavailable` presence type is the antithesis of the `available` presence type. It is used to qualify an

entity's unavailability. An entity is unavailable when its client has disconnected from the Jabber server.

So how can the client send an `unavailable` presence packet if it's disconnected? Furthermore, how can we make sure that clients actually *send* such a packet when they disconnect (to keep the presence information equilibrium)? Well, it can't, and we don't, respectively. It is the Jabber server that sends out `unavailable` presence packets on behalf of the entity once it disconnects. This is part of the presence service of the JSM and closely related to the presence subscription mechanism. See the sidebar *Availability Tracker* for more details.

While the `<show/>` and `<status/>` subelements qualify the `available` presence packet, there's no point in any embellishment of the fact that the entity is *unavailable*, so no subelements are used when the packet is of the `unavailable` type.

When a user is `unavailable`, any packets (`<message/>`, `<presence/>` or `<iq/>`) sent to that user are stored offline and delivered when he comes online and establishes his availability.

`type='probe'`

The `probe` presence type is a query, or probe, on another entity's availability. This probe is used by the Jabber server to determine the presence of entities in its management of the presence subscription mechanism. Under normal circumstances this presence probe should not be used directly by a client—availability information is always pushed to the client by the server. Regardless, if a client insists on using a probe, there are two things to bear in mind:

- Information will only be returned in response to an availability probe if the probing entity already has a subscription to the entity being probed. This means that you can't bypass the subscription model and probe random entities for availability information—only those who have previously given you permission to be informed of their availability. See the section called *Presence Subscription* for more details.
- The `<presence/>` packet must be specified with a `from` attribute specifying the sender's JID in the form `username@hostname` before it is sent. The Jabber server does not add this attribute. The presence mechanism will use the full JID (including any `resource`) when working out whether the prober has permission. This will ultimately fail because permission is determined on a `username@hostname` basis, not a `username@hostname/resource` basis.

`type='subscribe'`

This presence type is a request to subscribe to an entity's presence. *("Will you allow me to be sent your presence information by the server?")* See the section called *Presence Subscription* for details.

`type='unsubscribe'`

This presence type is a request to unsubscribe from an entity's presence. *("I don't want to be sent your presence information anymore, please have the server stop sending it to me.")* See the section called *Presence Subscription* for details.

`type='subscribed'`

This presence type is sent in reply to a presence subscription request, used to accept the request. *("Okay, I accept your request; the server will send you my presence information.")* See the section called *Presence Subscription* for details.

`type='unsubscribed'`

This presence type is sent in reply to a presence unsubscription request, used to accept the request. *("Okay, I accept your unsubscription request; the server will stop sending you my presence information.")*

It is also used to deny a presence subscription request. *("No, I don't accept your subscription request; I don't want the server to send you my presence information.")* See the section called *Presence Subscription* for details.

from

Set by server

Example: `<presence **from='dj@yak'**/>`

Similar to the attribute of the same name in the `<message/>` element, here the `from` attribute is set by the server and represents the JID where the availability information originates.

If you are sending a presence probe `type='probe'` you must set the `from` attribute yourself, as mentioned earlier.

to

Optional

Example: `<presence **to='sabine@yak'**/>`

The `to` attribute is optional; if you are just announcing availability (with the intention of having that announcement reflected to the appropriate members of your roster), then specifying a `to` attribute is not appropriate. [2]

If you want to send your availability to a specific entity, then do so using this `to` attribute, specifying that entity's JID. Why might you want to do this? See the sidebar *Availability Tracker* for an answer.

id

Optional

Example: `<presence **id='p1'**/>`

All Jabber elements support an `id` attribute for tracking purposes. So, the `<presence/>` packet is no different from the `<message/>` packet in this respect. As presence notification is usually a one-way thing, it is very uncommon to see `<presence/>` packets qualified with an `id` attribute.

Availability Tracker

The Jabber server (specifically, the presence handler within the JSM) has a mechanism called the *Availability Tracker*. As its name implies, its job is to track the availability of entities that have previously made an availability announcement (in a `<presence/>` element).

the section called *Presence Subscription* introduces the concept of exchange of availability information via an exchange agreement recorded in the roster. This mechanism covers the automatic distribution of availability notification based upon pre-arranged presence subscriptions.

However, Jabber services (which are connected to the **jabberd** backbone; see the section called *An Overview of the Server Architecture* in Chapter 4) can be connected to the Jabber backbone that need to know an entity's availability, or more importantly, when they suddenly become *unavailable*. These Jabber services usually won't have a prior presence subscription agreement recorded in anyone's roster.

The *Conferencing* service, which provides group chat facilities, allows users to join discussion “rooms” and chat, is one of these services. The service maintains data for each room's participants, and, so that it can manage its memory usage effectively, needs to know when a user ends their connection with the Jabber server—in other words when they become unavailable—so it can free that user's data. Normally, a user leaving a room is information enough for the service to know that data can be freed. But what if the user disconnects (or is disconnected) from their Jabber server without first leaving the room?

The availability tracker mechanism comes to the rescue. It maintains a list of JIDs to which an entity has sent their availability in a `<presence/>` packet containing a `to` attribute (i.e., a *directed* `<presence/>` packet. When the JSM notices that a user has ended their session by disconnecting, the presence handler invokes the availability tracker to send an `unavailable` `<presence/>` packet (with the `type='unavailable'` attribute) to all the JIDs to which the entity had sent directed availability information during the lifetime of that session.

How does this help in the *Conferencing* service case? Well, one of the requirements to enter a room is that presence must be sent to that room. Each room has its own JID, so a typical presence packet in room entry negotiation might look like this:

```
SEND: <presence to='jdev@conference.jabber.org'/>
```

which would be for the `jdev` room running at the conferencing service at `conference.jabber.org`. [3]

So, the availability tracker would have recorded this directed presence, and will send an unavailable presence to the same JID if the user's session ends.

Presence Subelements

show

Optional

Example:

```
<presence>
  <show>xa</show>
  <status>Gone home for the evening</status>
</presence>
```

When an available presence is sent, it can be qualified with more detail. The detail comes in two parts, and is represented by two subelements of the `<presence/>` element. The first part of the detail is in the form of a `<show/>` tag, which by convention contains one of five possible values. Table 5-4 lists these values, and their meaning.

Table 5-4. Presence 'show' values

Value	Meaning
away	The user is available, but temporarily away from the client.
chat	This is similar to the `normal` value, but suggests that the user is open to conversation.
dnd	"Do Not Disturb." Although online and *available*, the user doesn't want to be disturbed by anyone. Don't forget, unless the user is actually offline (unavailable, or disconnected from the Jabber server), messages to that user will be sent immediately and retrieved by the user when they change their status to available.
normal	This is the normal availability; there's nothing really special about this qualification—the user is simply available. If no `<show/>` tag is specified in an available `<presence/>` element, a value of normal is assumed.
xa	This is an extreme form of the `away` value—`xa` stands for *Extended Away*, and is probably as near to an unavailable presence as you can get.

status

Optional

Example:

```
<presence>
  <show>dnd</show>
  <status>working on my book!</status>
</presence>
```

The other part of the detail that qualifies a user's availability is the `<status/>` subelement. It allows for a more descriptive remark that embellishes the `<show/>` data.

The examples for this subelement and the `<show/>` subelement show how the `<status/>` value is used as a textual description to explain the `<show/>` value's "short-code", or mnemonic.

priority

Optional

Example:

```
<presence>
  <show>chat</show>
  <status>coffee break</status>
  <priority>5</priority>
</presence>
```

Earlier in this chapter, the section called *Resources and Priority* describes how a user's *priority* is used to determine the primary session to which messages should be sent.

As we see here, the priority is set using the `<presence/>` element. In this example, we see that the user has set the priority high to make sure that messages are routed to him on the Jabber client running on this machine.

x

Optional

Example:

```
<presence from='dj@yak/Work' to='sabine@yak'>
  <status>Online</status>
  <priority>1</priority>
  <x xmlns='jabber:x:delay' from='dj@yak/Work'
     stamp='20011005T10:58:28'/>
</presence>
```

Just as with the `<message/>` element, extra information can be attached to the `<presence/>` element by means of the `<x/>` tag. In the same way, each `<x/>` tag must be qualified with a namespace.

While there aren't many external uses for payloads in a `<presence/>` packet, the Jabber server uses this facility to add information. In this example, we see that `dj@yak`'s notification of availability (remember, `type='available'` is assumed for `<presence/>` packets without an explicit `type` attribute) is being sent to `sabine@yak`. While `dj@yak` connected to the Jabber server and send his availability (which was stamped on receipt by the Jabber server) just before 11 a.m., `sabine@yak` is just logging on now (say, 30 minutes later). When she receives `dj@yak`'s presence, she knows how long that presence status has been valid for.

See the section called *The X Namespaces* in Chapter 5a to find out what namespaces are available to qualify `<x/>`-included payloads.

Presence Subscription

Presence subscription is the name given to the mechanism that allows control over how entity presence information is made available to other entities. By default, the availability of an entity is unknown to other entities.

Let's put this into more concrete terms. For example, let's assume that you and I are both Jabber users. I'm registered with the Jabber server running at `jabber.org`, my JID is `qmacro@jabber.org`, and you are registered with a Jabber server running at your company, and your JID is `you@yourserver.com`.

If you want to know whether I'm available, you have to *subscribe* to my `presence`. This is done by sending a `<presence/>` packet to me with the `type` attribute set to `subscribe`. In the example that follows, the XML fragments are sent and received from *your* perspective):

```
SEND: <presence type='subscribe' to='qmacro@jabber.org'/>
```

I receive the `<presence/>` packet, and when I receive it, it's been stamped (by your Jabber server) with a `from` attribute with the value `you@yourserver.com`. So, based upon who it is, I decide to accept the subscription request and send back a reply:

```
RECV: <presence type='subscribed' to='you@yourserver.com'/>
```

This lets you know that I've accepted your subscription request. From now on, every time my availability changes (when I send a `<presence/>` packet or when I disconnect (and the server generates an `unavailable` `<presence/>` packet on my behalf), that availability information will be relayed to you.

But how does this work? How does the Jabber server know that you've subscribed to my presence and I've accepted that subscription?

Enter the *roster*, stage right. The roster is a list of JIDs maintained for each user, stored on the server-side. A roster is similar to an AOL's Buddy List; one could say that it's a sort of personal address book, but it's more than that. The presence subscription and roster mechanisms are tightly intertwined. We'll be examining the roster in more detail in the section called *`jabber:iq:roster`* in Chapter 5a. Here, we'll just look at the characteristics of the roster that are relevant for the presence subscription mechanism. The roster is managed using the third basic Jabber element—`<iq/>`—which will be explained in more detail later in this section. Ignore the tags that you aren't yet familiar with; it's just important to get the basic drift of what's going on.

While the roster is stored and maintained on the server-side, any changes to the roster are made by the server are reflected (pushed) in the client so it can be synchronized with a local copy. [4]

Let's expand the simple exchange of `<presence/>` packets from above and look how the roster is used to record presence subscription information.

If you wish to subscribe to my presence and add my JID to your roster at the same time, these two actions are linked for obvious and practical reasons. Many Jabber clients use the roster as a basis for displaying availability information, and with the exception of an entity sending presence information directly to another entity irrespective of roster membership, presence subscription information is stored by-user in the roster. Here's the order in which the subscription would take place:

1. A request is sent to the server to update your roster, adding my JID to it:

```
SEND: <iq id="adduser1" type="set">
        <query xmlns="jabber:iq:roster">
```

```
          <item jid="qmacro@jabber.org" name="DJ Adams"/>
        </query>
      </iq>
```

You add an `id` attribute to be able to track the request and match up the response when it comes.

2. The server responds with a push of the updated (new) roster item:

```
RECV: <iq type='set'>
        <query xmlns='jabber:iq:roster'>
          <item jid='qmacro@jabber.org' name='DJ Adams'
                subscription='none'/>
        </query>
      </iq>
```

Note that in the update, an additional attribute `subscription='none'` is sent, reflecting the presence subscription relationship between you and me. At this stage, the relationship is that I don't have a subscription to your presence and you don't have a subscription to my presence, hence the value `none`.

3. It also acknowledges the original update request, confirming its success:

```
RECV: <iq id='adduser1' type='result'
          from='you@yourserver.com/Work'
          to='you@yourserver.com/Work'/>
```

Note the `id='adduser1'` identity is passed back so we can track the original request and from where this response is being made.

4. Meanwhile, you send the subscription request:

```
SEND: <presence to="qmacro@jabber.org" type="subscribe"/>
```

5. The server notes the subscription request going through and once more updates your roster and pushes the item out to you:

```
RECV: <iq type='set'>
        <query xmlns='jabber:iq:roster'>
          <item jid='qmacro@jabber.org' name='DJ Adams'
                subscription='none' ask='subscribe'/>
        </query>
      </iq>
```

The current subscription relationship is reflected with the `subscription='none'` attribute. In addition, we have a subscription request status, with `ask='subscribe'`. This request status shows that there is an outstanding presence subscription request to the JID in that roster item. If you've ever seen the word “Pending” next to a username in a Jabber roster, this is where that comes from. Don't forget that a subscription request might not get an immediate response, so we need to remember that the request is still outstanding.

6. Your subscription request is received and accepted, and a `subscribed` type is sent back to you as part of a `<presence/>` packet:

```
RECV: <presence to='you@yourserver.com'
```

```
                type='subscribed' from='qmacro@jabber.org'/>
```

7. The server also notices the subscription request acceptance, and yet again updates your roster to keep track of the presence subscription. Again, it pushes the subscription information out to you so your client can keep its copy up to date:

   ```
   RECV: <iq type='set'>
           <query xmlns='jabber:iq:roster'>
             <item jid='qmacro@jabber.org' name='DJ Adams'
                   subscription='to'/>
           </query>
         </iq>
   ```

   This time, the `subscription` attribute in the roster item has been set to `to`. This means that the roster owner (you) has a presence subscription *to* the JID in the roster item (i.e., me).
8. The server knows you've just subscribed to my presence, it generates a presence probe on your behalf which causes my presence information to be retrieved and sent to you.

   ```
   RECV: <presence from='qmacro@jabber.org/Work'
                   to='you@yourserver.com'>
           <status>Available</status>
           <priority>1</priority>
           <x xmlns='jabber:x:delay'
              from='qmacro@jabber.org/Work'
              stamp='20010515T11:37:40'/>
         </presence>
   ```

Of course, at this stage, our relationship is a little unbalanced, in that you have a subscription request to me, but I don't have a subscription request to you. So you are aware of my availability, but not the other way around. In order to rectify this situation, I can repeat the process in the opposite direction, asking for a subscription to your presence information.

The only difference to the sequence that we've just seen is that you will already exist on my roster because the server will have maintained an item for your JID to record the presence subscription relationship. While the item in your roster that represents my JID has a `subscription` attribute value of `to` (the roster owner has a presence subscription *to* this JID)—we've seen this in Step 7—the item in my roster that represents your JID has a `subscription` attribute value of `from` (the roster owner has a presence subscription *from* this JID).

Once I repeat this sequence to subscribe to your presence (and you accept the request), the value for the `subscription` attribute in the items in each of our rosters will be set to `both`.

The upshot of all this is that when an entity announces its presence, it does so using a single `<presence/>` packet, with no `to` attribute specified. All the members in that entity's roster who have a subscription to that entity's presence will receive a copy of that `<presence/>` packet and thereby be informed. [5]

The IQ Element

The third and final element in the Jabber building block set is the `<iq/>` element ("iq" stands for "Info/Query"), which represents a mechanism for sending and receiving information. What the `<iq/>` element has over the `<message/>` element for this purpose is *structure* and *inherent meaning*. It is useful to liken the info/query mechanism to the request/response model of the HyperText Transfer Protocol (HTTP) using GET and POST.

The `<iq/>` element allows a structured conversation between two Jabber entities. The conversation exists to exchange data, to retrieve or set it, and to notify the other party as to the success (or not) of that retrieve or set action. There are four *states* that an `<iq/>` element can be in, each reflecting one of the activities in this conversation:

get

: Get information.

set

: Set information.

result

: The result, in the case where the get or set was successful.

error

: An error, in the case where the get or set was not successful.

At the beginning of this section, we saw various elements in action in Example 5-2. The first two were `<iq/>` elements, and show a retrieval request and response for roster information.

First comes the request:

```
SEND: <iq id='roster_0' type='get'>
        <query xmlns='jabber:iq:roster'/>
      </iq>
```

Then the response:

```
RECV: <iq id='roster_0' type='result' from='dj@yak/Work'>
        <query xmlns='jabber:iq:roster'>
          <item jid='sabine@yak' name='sabine' subscription='both'>
            <group>Family</group>
          </item>
        </query>
      </iq>
```

There's a number of things we can see in this snippet:

- The type of each info/query activity is identified by the `type` attribute.
- Each info/query activity contains a subelement `<query/>` which is qualified by a namespace.
- The subelement is used to carry the information being retrieved.
- The response (`type='result'`) can be matched up to the request (`type='get'`) via the `id` tracking

attribute.

So, if we look at the first `<iq/>` element:

```
<iq id='roster_0' type='get'>
```

We can see that this "request" `<iq/>` doesn't contain a `to` attribute. This is because the request is being made *of* the Jabber server (specifically the JSM), instead of a particular user. Next we see the response from the server:

```
<iq id='roster_0' type='result' from='dj@yak/Work'>
```

This "response" `<iq/>` contains a `from` attribute stating that the result is coming back from the original requester! This is simply because the `from` attribute is a hang-over from the original request to the Jabber server, which is stamped with its origin (`dj@yak/Work`) in the form of the `from` attribute. Here, as in many other places in the Jabber server, the response is simply built by turning the incoming request packet around and adding whatever was required to it before sending it back.

Okay, let's examine the details of the `<iq/>` element.

IQ Attributes

The attributes of the `<iq/>` element are the same as those of the `<presence/>` and `<message/>` elements, and used pretty much in the same way.

type

Required

Example: `<iq type='get'/>`

As mentioned already, the `from` attribute is used to specify the activity.

type='get'

This is used to specify that the `<iq/>` element is being used in *request* mode, to retrieve information. The actual subject of the request is specified using the namespace qualification of the `<query/>` subelement; see later in this section for details.

Using the HTTP parallel, this is the equivalent of the GET verb.

type='set'

While the `get` `<iq/>` type is used to retrieve data, the corresponding `set` type is used to send data, and is the equivalent of the POST verb in the HTTP parallel.

Very often, a `get` request will be made of an entity, to discover fields that are to be completed to register with

that entity. The Jabber User Directory (JUD) is a component that plugs into the **jabberd** backbone and provides simple directory services; users can register an entry in the JUD address book, on which searches can be performed.

Let's have a look how IQ elements are used to interact with the JUD.

The registration conversation with the JUD starts with an `<iq/>` `get` to discover the fields that can be used for registration, followed by an `<iq/>` `set` filling those fields in the act of registration. Note how each time, the JUD responds with an `<iq/>` `result` to confirm each action's success.

Here we are requesting registration information from the JUD. Note the namespace that qualifies the `<query/>` subelement (and hence the `<iq/>`).

```
SEND: <iq type='get' to='jud.yak'
                  id='judreg_ask'>
        <query xmlns='jabber:iq:register'/>
      </iq>
```

The JUD responds with the fields to fill in. The response is basically a copy of the request, with new attributes and tags.

```
RECV: <iq type='result' to='dj@yak/Work'
          from='jud.yak' id='judreg_ask'>
        <query xmlns='jabber:iq:register'>
          <instructions>
            Complete the form to submit your details
            to the User Directory
          </instructions>
          <name/>
          <first/>
          <last/>
          <nick/>
          <email/>
        </query>
      </iq>
```

Now we know what to send:

```
SEND: <iq type='set' to='jud.yak' id='judreg_do'>
        <query xmlns='jabber:iq:register'>
          <name>DJ Adams</name>
          <first>DJ</first>
          <last>Adams</last>
          <nick>qmacro</nick>
          <email>dj@mailserver.org</email>
        </query>
      </iq>
```

And the JUD responds saying our *set* request was successful.

```
RECV: <iq type='result' to='dj@yak/Work'
                  from='jud.yak' id='judreg_do'/>
```

type='result'

As shown in the JUD conversation, the 'result' type `<iq/>` packet is used to convey a result. Whether that result is boolean (*it worked*, as opposed to *it didn't work*) or conveys information (such as the registration fields that were requested), each *get* or *set* request is followed by a `result` response, if successful.

type='error'

If not successful, the *get* or *set* request is not followed by a `result` response, but an `error` response. In the same way that a subelement `<error/>` carries information about what went wrong in a `<message type='error'/>` element, so it also provides the same service for `<iq type='error'/>` elements. [6]

Let's have a look at an `<iq type='error'>` in action. A user, who is trying to join a conference room, is notified that his entrance is barred because he hasn't supplied a required password.

First, the user requests information on the room he wishes to join.

```
SEND: <iq type="get" id="conf1" to="cellar@conference.yak">
        <query xmlns="jabber:iq:conference"/>
      </iq>
```

The conference component instance, to which the `<iq/>` was addressed (with the `to='cellar@conference.yak'` attribute), responds with information about the 'cellar' room, including the fact that a *nickname* and *password* must be specified to gain entrance.

```
RECV: <iq type='result' id='conf1' to='dj@yak/winjab'
                           from='cellar@conference.yak'>
        <query xmlns='jabber:iq:conference'>
          <name>Dingy Cellar</name>
          <nick/>
          <secret/>
        </query>
      </iq>
```

After sending availability to the room, to have the availability tracker kick in for that room's JID (see the sidebar *Availability Tracker* earlier in this section)…

```
SEND: <presence to="cellar@conference.yak"/>
```

…entrance to the room is attempted with a nickname but no password is specified.

```
SEND: <iq to="cellar@conference.yak" type="set" id="conf2">
```

```
        <query xmlns="jabber:iq:conference">
          <nick>dj</nick>
        </query>
      </iq>
```

The entrance attempt was unsuccessful. An `error` response is given with an `<error/>` subelement explaining what the problem was.

```
RECV: <iq to='dj@yak/winjab' type='error' id='conf2'
           from='cellar@conference.yak'>
        <query xmlns='jabber:iq:conference'>
          <nick>dj</nick>
        </query>
        <error code='401'>Unauthorized</error>
      </iq>
```

Again, the response is simply the request with modified attributes and data (the `<error/>` tag) added.

from

Set by server

Example: `<iq from='dj@yak/Work'/>`

Similar to the `from` attribute in the `<message/>` and `<presence/>` elements, this is set by the server and represents the JID where the `<iq/>` originated.

to

Optional

Example: `<iq to='jdev@conference.jabber.org'/>`

This attribute is used to specify the intended recipient of the info/query action or response. If no `to` attribute is specified, the delivery of the packet is set to the sender, as is the case for `<message/>` packets. However, unlike the case for `<message/>` packets, `<iq/>` packets are usually dealt with enroute and handled by the JSM.

What does that mean? Packets sent from a client travel over a `jabber:client` XML stream and reach the Jabber server, where they're routed to the JSM. [7]

A large part of the JSM consists of a series of packet handlers whose job it is to review packets as they pass through and act upon them as appropriate; some of these actions may cause the packet to be deemed to have been “delivered” to its intended destination (thus causing the packet routing to end for that packet) before it gets there.

So in the case of `<iq/>` packets without a `to` attribute, the default destination is the sender's JID, as we've already seen with the `<message/>` element. But because JSM handlers that receive the packet may perform some action to handle it and cause that packet's delivery to be terminated (marked complete) prematurely, the

effect is that something sensible will happen to the `<iq/>` packet that doesn't have a `to` attribute and it won't appear to act like a boomerang.

Here's an example:

The namespace `jabber:iq:browse` represents a powerful *browsing* mechanism that pervades much of the Jabber server's services and components. Sending a simple browse request without specifying a destination (no `to` attribute):

```
SEND: <iq type='get'>
        <query xmlns='jabber:iq:browse'/>
      </iq>
```

will technically be determined to have a destination of the *sender's* JID. However, a JSM handler called `mod_browse` which performs browsing services gets a look-in at the packet before it reaches the sender and *handles* the packet to the extent that the query is deemed to have been answered and so the delivery completed. The packet stops travelling in the sender's direction, having been responded to by `mod_browse`:

```
RECV: <iq type='result' to='dj@yak/sjabber' from='dj@yak'>
        <user name='DJ Adams' xmlns='jabber:iq:browse' jid='dj@yak'/>
      </iq>
```

And while we're digressing, here's a meta-digression: We see from this example that a `browse` to a particular JID is handled at the server. The client doesn't even get a chance to respond. So, as one of browsing's remits is to facilitate resource discovery (the idea is that you can query someone's client to find out what that client supports—whiteboarding or XHTML text display, for example), how is this going to work if the client doesn't see the request and can't respond? [8]

The answer lies in the distinction of specifying the recipient JID with or without *resource*. As a resource is per-client connection and often *represents* that client, it makes sense to send a browse request to a JID including a specific resource:

```
SEND: <iq type='get' to='qmacro@jabber.org/sjabber'>
        <query xmlns='jabber:iq:browse'/>
      </iq>
```

This time the destination JID is resource-specific and the packet passes by the `mod_browse` handler to reach the client (sjabber), where a response can be returned: [9]

```
RECV: <iq type='result' to='piers@jabber.org/WinJab
                    from='qmacro@jabber.org/sjabber'>
        <user type='client' xmlns='jabber:iq:browse'
                        jid='qmacro@jabber.org/sjabber'>
          <whiteboard/>
          <videochat/>
          <PGP/>
        </user>
```

```
</iq>
```

id

Optional

Example: `<iq type='get' id='roster1'>`

If we're going to rank the elements in terms of tracking importance, `<iq/>` would arguably come out on top, as it inherently describes a request-response mechanism. So this element also has an `id` attribute for tracking purposes.

Don't forget that the pair of XML streams that represent the two-way traffic between Jabber client and server are independent, and any related packets such as a request (travelling in one XML stream) and the corresponding response (travelling in the other) are asynchronous. So a tracking mechanism like the `id` attribute is essential to be able to match packets up.

IQ Subelements

We've seen two of these three subelements of the `<iq/>` element already in earlier examples - `<query/>` and `<error/>`. The other one is `<key/>`. Here's a review of them.

query

Required

Example:

```
<iq type='get' to='yak'>
<query xmlns='jabber:iq:version'/>
</iq>
```

We've already seen the `<query/>` subelement performing the task of container for the info/query activity.

- For a *get* activity, the subelement usually just contains a qualifying namespace that in turn defines the essence of the *get* activity. This is evident in the example here, where the `<iq/>` element is a retrieval of the server (`yak`) version information.
- For a *set* activity, it contains the qualifying namespace and also child tags that hold the data to be *set*, as in this example where a vCard (an electronic “business card”) is being updated:

```
SEND: <iq type='set'>
        <vCard xmlns='vcard-temp' version='3.0'>
        ... [vCard information] ...
        </vCard>
      </iq>
```

- When `result` information is returned, it is enclosed within a `<query/>` subelement qualified with the

appropriate namespace, like in this response to the earlier request for server version information:

```
RECV: <iq type='result' to='dj@yak/Work' from='yak'>
        <query xmlns='jabber:iq:version'>
          <name>jsm</name>
          <version>1.4.1</version>
          <os>Linux 2.2.12-45SAP</os>
        </query>
      </iq>
```

Of course, there are some results that don't carry any further information—the so-called *boolean* results. When there's no information to return in a result, the `<query/>` subelement isn't necessary. A typical case where a boolean result is returned is on successfully authenticating to the Jabber server (where the credentials are sent in a *set* request in the `jabber:iq:auth` namespace); the result `<iq/>` element would look like this:

```
RECV: <iq type='result' id='auth_0'/>
```

- And for an `error` situation, while the actual error information is carried in an `<error/>` subelement, any context in which the error occurred is returned too in a `<query/>` subelement. This is usually because the service returning the error just turns around the *set* `<iq/>` packet—which already contains the context as the data being *set*—and *adds* the `<error/>` subelement before returning it.

  Here we see that the authentication step of connecting to the Jabber server failed because Sabine mistyped her password:

```
RECV: <iq type='error' id='auth_0'>
        <query xmlns='jabber:iq:auth'>
          <username>sabine</username>
          <password>geheimnix</password>
          <resource>pavilion</resource>
        </query>
        <error code='401'>Unauthorized</error>
      </iq>
```

Whoa! Hold on a minute, what's that **`<vCard`** `xmlns='vcard-temp' version='3.0'>` doing up there? Shouldn't it be **`<query`** `xmlns='vcard-temp' version='3.0'>`?

Actually, no. The *name* of the subelement doesn't really matter, in fact. By convention it's `<query/>`. [10] But it can be anything. When you're interacting with a Jabber server via telnet and are writing the stream fragments by hand, you'll soon appreciate the fact that you can write something like

```
<q xmlns='.....'>
```

instead of

```
<query xmlns='.....'>
```

It doesn't seem much, but after the twentieth `<iq/>` packet, you'll think otherwise!

The critical part of the subelement is the namespace specification with the `xmlns` attribute. And we've seen this somewhere before—in the definition of component instance configuration in the section called *Server Configuration* in Chapter 4 we learn that the tag wrapping the component instance's configuration, like that for the c2s service:

```
<service id="c2s">
  ...
  <pthcsock xmlns='jabber:config:pth-csock'>
    ... [configuration here] ...
  </pthcsock>
</service>
```

which is `pthcsock` here, is irrelevant, while the namespace defining that tag (`jabber:config:pth-csock`) is important because it's what is used by the component to retrieve the configuration.

We've seen this feature in this chapter too; remember the `<iq/>` examples in the `jabber:iq:browse` namespace? The result of a browse request that returned user information looked like this:

```
RECV: <iq type='result' to='dj@yak/sjabber' from='dj@yak'>
        <user name='DJ Adams' xmlns='jabber:iq:browse' jid='dj@yak'/>
      </iq>
```

Again, the `query` tag is actually `<user/>`. In fact, in browsing, the situation is extreme, as the `<iq/>` response's subelement tag name will be different depending on what was being browsed. But what is always consistent is the namespace qualifying the subelement; in this example it's `jabber:iq:browse`.

error

Optional

Example:

```
<iq type='error' from='dj@yak/Work' to='dj@yak/Work'>
  <query xmlns='jabber:iq:browse'/>
  <error code='406'>Not Acceptable</error>
</iq>
```

The `error` subelement carries error information back in the response to a request that could not be fulfilled. Table 5-3 shows the standard error codes and default accompanying texts.

The example here shows the response to a browse request, but why might the request have been erroneous? Because the `<iq/>` `type` attribute had been specified as `set` instead of `get`. Browsing is a read-only mechanism.

key

Optional

Example:

```
<iq type='result' id='callback_13' to='dj@yak/Work' from='callback.yak'>
  <query xmlns='jabber:iq:register'>
    <password/>
    <instructions>
      Please specify a name and password to register for this service.
    </instructions>
    <key>cff28e89afa94e734aabfb11ec1099780450d80e</key>
  </query>
</iq>
```

The `<key/>` tag is used in registration sequences to add a simple form of security between the service and the entity registering.

The example here shows the result of an `<iq/>` *get* request, qualified by the `jabber:iq:register` namespace, made to an imaginary alarm service (*callback*) running on the Jabber server `yak`. What must happen for a successful registration is that the key enclosed in the `<key/>` tag must be sent as-is in the `<iq/>` *set* request along with the rest of the information required (name and password). This is so that the service can verify that the requester is known to the service as having previously asked for procedural instructions.

The value of the key is a hash value randomly generated by the server.

Notes

[1] Actually, it is, internally, but the effect is that it isn't. The packet is swallowed on its final delivery stage by the presence handler.

[2] In fact, as in the cases for the other two elements `<message/>` and `<iq/>`, not specifying a `to` attribute will cause the `<presence/>` packet to be sent to the sender. However, in the case of the presence handler mechanism, the packet is swallowed before it can reach its destination to prevent reflective presence problems.

[3] The example here contains a room JID with no resource specified; this is taken from the 0.4 version of the Conferencing protocol. An earlier version of the protocol (Groupchat 1.0) required that the nickname for the person entering the room be specified as a resource to the room's JID; for example: `jdev@conference.jabber.org/dj`.

[4] The local copy would only exist for the duration of the user's session and should always be regarded as a *slave copy*.

[5] That is, where there's a value of `to` or `both` in the roster item's `subscription` attribute.

[6] Table 5-3 lists the standard Jabber error codes and their default descriptions.

[7] with the internal `<route/>` packet; see the section called *Component types* in Chapter 4 for more details.

[8] Whiteboarding is collaborative sketching, not a form of surfing atop wave crests.

[9] Well, I can dream, can't I?

[10] There is, of course an exception to this rule. The name of the tag *must* be `query` in the user registration and authentication steps; see the description for the `jabber:iq:auth` namespace in the next section, and Chapter 6.

Prev | Home | Next
XML Streams | Up | Jabber Namespaces

Chapter 5a. Jabber Namespaces

Table of Contents

While the building blocks of the Jabber protocol, described in Chapter 5, provide the groundwork for our solutions, for our chess rules, there's still something missing.

There's a purity and elegance that's to be had with usage of the three core elements `<message/>`, `<presence/>`, and `<iq/>`, but there's a depth of meaning that's missing. While the core elements define the moves we can make, it's the Jabber *namespaces* that provide us with the contextual set-moves that allow us to relate Jabber to the real world.

Namespaces provide a level of meaning, an environmental layer, above the basic "packet shunting" world that would otherwise exist if our elements were to be passed back and forth bereft of context and application.

Basic activities like user registration, authentication, roster management, and time stamping are all made possible using meaning brought about by the application of standard Jabber namespaces to our elements.

This Chapter describes those namespaces.

Namespace usage

In the previous chapter, we made quite a few references to *namespaces*. Jabber's namespaces are used within the message elements to qualify payloads (distinct content) within these elements. For example:

```
RECV: <iq id='roster_0' type='result' from='dj@yak/Work'>
        <query xmlns='jabber:iq:roster'>
          <item jid='sabine@yak' name='sabine' subscription='both'>
            <group>Family</group>
          </item>
        </query>
      </iq>
```

Here the `jabber:iq:roster` namespace is being used to qualify a chunk of XML that contains roster information embedded in an `<iq/>` element. A payload exists as a subelement of the main element (that is, a child tag of the parent `<message/>`, `<presence/>` or `<iq/>` tag) and, in XML terms, belongs to a different namespace than the main element.

The namespace of the *main* elements in the XML document that is streamed across the connection—`<message/>`, `<presence/>`, and `<iq/>`, and indeed their "standard" subelements, such as `<message/>`'s `<subject/>` tag—is defined in the root tag of the XML document, and in this case is `jabber:client`. Namespaces like `jabber:client` that are used to qualify such XML document body fragments are described in the section called *The Opening Tag* in Chapter 5. While the main elements in our client to server connection are qualified by `jabber:client`, each distinct payload (“attachment” is also a good way to think of these additional chunks of XML) is qualified by one of the specific namespaces listed in this Chapter.

Jabber namespace naming rules

Standard Jabber namespaces begin `jabber:`; however, there are a few exceptions. It could be argued that the exceptions aren't really Jabber standard since these are the namespaces that describe things like vCards and XHTML payloads. There's nothing to stop you from defining your own namespaces to qualify any sort of XML you'd like to attach to a Jabber element. The only rule is that if you do, it *shouldn't* begin with `jabber:`.

Further to the rule that Jabber standard namespaces begin with `jabber:`, the categorization can be seen

as falling into two distinct spaces. The first, the `iq` space, contains namespaces that qualify content within `<iq/>`-based conversations. The second, the `x` space, contains namespaces that qualify extensions within all the elements (`<message/>`, `<iq/>`, and `<presence/>`).

Prev | Home | Next
Jabber Namespaces | Up | The IQ Namespaces

The IQ Namespaces

The namespaces that qualify attachments to `<iq/>` elements are many and varied. After all, you could say that the raison d'etre of this request/response mechanism is to exchange structured information—and what better way to define that information than with namespaces?

This section looks briefly at each of the IQ namespaces in turn. Some of them will be covered in more detail in later Chapters, as they will be used in examples that appear later in the book.

`jabber:iq:agent`

The `jabber:iq:agent` namespace is used to request and return information on an *agent*. An agent is a service running on a Jabber server, and it has a JID. To find out what features the particular agent offers, an info-request (an `<iq/>` `get` request) can be made using this namespace:

```
SEND: <iq type='get' to='yak/groups'>
        <query xmlns='jabber:iq:agent'/>
      </iq>
```

Here, a request for features is being made of the agent with the JID `yak/groups`, which is the standard name for the “Shared Groups” service. The JID here is composed of a hostname (`yak`) and a resource (`groups`).

The response looks like this:

```
RECV: <iq type='result' to='dj@yak/Work' from='yak/groups'>
        <query xmlns='jabber:iq:agent'>
          <name>Jabber Server at yak</name>
          <url>http://yak</url>
          <service>jabber</service>
          <register/>
        </query>
      </iq>
```

In reality, although the agent or service itself is specified as the recipient of the query, it is often a centralized mechanism that responds on behalf of the agent if the agent itself doesn't or can't respond. [1] This means that the results of the query might not be as helpful as you might expect. The only detail in the response shown here that might be of some use is the `<register/>` tag, but that's actually misleading as it's picked up from the general registration capabilities configuration and not anything particular to the JUD.

The main reason for this is actually also the answer to a question you might have right now: "How do I know which agent JIDs I can query on a particular Jabber server?" Indeed. It's very hit and miss to pick agent JIDs at random. The `jabber:iq:agents` (plural) namespace defines a *list* of agents. Usually what happens is that a query is made using the `jabber:iq:agents` namespace, and then further detail is requested with the `jabber:iq:agent` for a particular agent. However:

- The general information for both queries comes from the same place in the Jabber server configuration.
- That place is the `<agents/>` tag inside the JSM custom configuration…and is deprecated in favor of the `<browse/>` tag.

The `jabber:iq:agent`-based agent facility query is slowly but surely being replaced by the more generic but more powerful `jabber:iq:browse` mechanism (which is directly related to the `<browse/>` configuration area of the JSM). That said, it is still supported for compatibility reasons; many Jabber clients still use the `jabber:iq:agent` and `jabber:iq:agents` namespaces in calls to discover services on the server. See the section called *`jabber:iq:browse`* for more details on the `jabber:iq:browse` mechanism.

`jabber:iq:agents`

The `jabber:iq:agent` namespace is used in a query of an individual Jabber agent, or service. Likewise, the `jabber:iq:agents` namespace is used in a query to retrieve a *list* of these agents.

As mentioned in the description for the `jabber:iq:agent` namespace, the Jabber server configuration (in the JSM custom configuration section) in earlier releases contained an `<agents/>` tag, which was used to list the agents that were available on the Jabber server. The listing looked like this:

```
<agents>

  <!-- Note: this <agents/> listing is not used in 1.4.1 -->

  <agent jid='users.jabber.org'>
    <name>Jabber User Directory</name>
    <description>
      You may register and create a public searchable profile,
      and search for other registered Jabber users.
    </description>
    <service>jud</service>
    <register/>
    <search/>
  </agent>

  <agent jid='...'>
    ...
  </agent>

  ...

</agents>
```

The `<agents/>` listing has now been superseded by the `<browse/>` tag. In fact, when responding to `jabber:iq:agents` *and* `jabber:iq:browse` queries, the Jabber server itself will refer to the same `<browse/>` listing in both cases. Here's an example of a response to a `jabber:iq:agents` query:

```
RECV: <iq type='result' to='dj@yak/laptop' from='yak' id='agents'>
        <query xmlns='jabber:iq:agents'>
          <agent jid='users.jabber.org'>
            <name>Jabber User Directory</name>
            <service>jud</service>
```

```
          <register/>
          <search/>
        </agent>
      </query>
    </iq>
```

We can see that this response pretty much reflects the information in the `<agents/>` configuration.

For more details on how this differs in response to a `jabber:iq:browse` query, see the section called *`jabber:iq:browse`*.

Further examples of `jabber:iq:agents` usage can be found in the section called *RSS punter* in Chapter 8.

jabber:iq:auth

The `jabber:iq:auth` namespace is used to qualify a structured authentication procedure between client and server.

Details of authentication are covered in Chapter 6; however, here we will look at the simplest authentication conversation between client and server. In this example, the client sends a username and password, and the server responds by sending a “successful” response, acknowledging the user's credentials, thus creating a session for that user.

```
SEND: <iq type='set' id='auth0'>
        <query xmlns='jabber:iq:auth'>
          <username>sabine</username>
          <password>geheimnis</password>
          <resource>WinJab</resource>
        </query>
      </iq>

RECV: <iq type='result' id='auth0'/>
```

In this authentication procedure, the name of the query subelement containing the `xmlns` attribute *must* be `query`. This is an exception to the rule mentioned in the previous section, which stated that the name of the tag was irrelevant.

jabber:iq:autoupdate

The “Update Info Request” configuration description in the section called *Update Info Request* in Chapter 4 describes a mechanism for Jabber servers to query a software version information repository to find out about new versions of the server. [2] This version information repository that responds to queries is also known as the “Auto-Update” service.

Not only can servers request software update information, but clients can too. The procedure is the same in both cases, and involves the `jabber:iq:autoupdate` and `jabber:x:autoupdate` namespaces. If clients support this software update information request, it will be usually in the form of a "silent" request that it sends out at startup. The sending out of this request can be often switched on and off in the client's configuration.

The conversation starts with the requester sending a special availability packet to the information repository. Currently, there are two such public repositories: one running at `jabber.org` covering a wide range of Jabber software, and the other running at `jabber.com` covering certain clients including *Jabber Instant Messenger* (JIM).

This special availability packet looks like this:

```
SEND: <presence to='959967024@update.jabber.org/1.6.0.3'/>
```

This is a *directed* `<presence/>` packet because it has a `to` attribute. What's even more interesting is that if we breakdown the JID, we're left with `959967024` as the username, `update.jabber.org` as the hostname, and `1.6.0.3` as the resource. This doesn't mean that the availability is destined for a user called `959967024` registered on the `update.jabber.org` Jabber server. While most presence packets are destined for users (within the presence subscription model), this one is destined for a service.

The service is running with the identification `update.jabber.org`—a component connected to the **jabberd** server backbone running at `jabber.org`. Therefore, the `<presence/>` packet will be routed to that service. Unlike the JSM, the `update.jabber.org` service has no concept of users or sessions. Instead, it receives the complete `<presence/>` packet and disassembles the JID in the destination address and interpret component parts as it sees fit.

The service uses the `username` portion of the JID to identify the piece of software for which new version information is being requested. In our example, this is `959967024`. This value actually represents the JIM client, and is the key to the client database kept on http://www.jabbercentral.org. Using a unique client database key to represent the piece of software allows the client's name to be changed without causing problems in the retrieval of version information by the Auto-Update service.

The version information stored in the repository is compared to the current version of the requesting piece of software; in this case our JIM version 1.6.0.3. If a new version isn't available, nothing will happen. Because the initial part of the request was a `<presence/>` packet, no official response is expected (unlike a situation where the initial part of the request was an `<iq/>` `get` packet).

If there *is*, however, information stored in the repository about newer versions of the software, the query is replied to using a `<message/>` element, with an autoupdate *attachment*:

```
RECV: <message to='qmacro@jabber.org/Work' from='959967024@update.jabber.org'>
        <subject>Upgrade available for Jabber Instant Messenger</subject>
        <body>
          There is an update available for Jabber Instant Messenger.
          If your client supports the iq:autoupdate namespace then
          you should see something in the client that will list the
          available files. If not, then goto http://www.jabbercentral.com
          and grab the new version.
        </body>
        <x xmlns='jabber:x:autoupdate'>959967024@update.jabber.org</x>
      </message>
```

The reply contains some text (in the `<subject/>` and `<body/>` tags) that could be displayed to the user.

Furthermore, the autoupdate *attachment*—an `<x/>` subelement qualified by the `jabber:x:autoupdate` namespace—contains information on where further information can be obtained in a programmatic way. [3]

This "programmatic way" sends an empty `<iq/>` query, qualified by the `jabber:iq:autoupdate` namespace, to the address given in the `jabber:x:autoupdate` `<message/>` attachment:

```
SEND: <iq type="get" id="id_3" to="959967024@update.jabber.org">
        <query xmlns="jabber:iq:autoupdate"/>
```

```
      </iq>
```

We're back on familiar ground; the Auto-Update service responds to the request by sending version information for that piece of software:

```
RECV: <iq type='result' to='qmacro@jabber.org/Work'
                        from='959967024@update.jabber.org' id='id_3'>
        <query xmlns='jabber:iq:autoupdate'>
          <release priority='optional'>
            <url>http://www.jabber.com/download/jabbersetup.exe</url>
            <version>1.7.0.14</version>
            <desc/>
          </release>
        </query>
      </iq>
```

The response contains information about the latest software release that prompted the version request. The release is either *required* or *optional* (as in this example). The tags within the `jabber:iq:autoupdate` qualified query are fairly self-explanatory; note that the version description is empty in this example.

`jabber:iq:browse`

The `jabber:iq:browse` namespace is relatively new and could almost be seen as a departure from the traditional namespaces found elsewhere in Jabber. While namespaces such as `jabber:iq:agents` and `jabber:iq:register` define very strict content using specific tag names, `jabber:iq:browse` allows a more free-form containment of information. Both forms of *tight* and *loose* namespaces have a place in Jabber.

The real world contains countless types and classifications of information far more than you could ever reasonably cover with a finite collection of namespaces. And even if you did, that coverage would be out of date as soon as it was completed. The Jabber concept of *browsing*, introduced in Chapter 2 is an approach to being able to classify and exchange information of all kinds without the definitions being previously cast in stone.

More or less any hierarchical information can be represented in the `jabber:iq:browse` namespace. It can be seen as an open-ended way of describing structures in an almost ad-hoc way. That said, the namespace comes with some general rules and some predefined classifications.

Information represented and described in a `jabber:iq:browse` extension is subject to classification. This classification is in two levels: *categories* and *subtypes*. The *category* is used to define the general area or type of information being represented. The *subtype* is to give a more specific definition of that category. Table 5a-1 shows a list of initial categories.

Table 5a-1. `jabber:iq:browse` Categories

Category	Description
application	Applications addressable via a JID can be described in this category. Initial suggestions for such application subtypes include `calendar` (calendar/schedule services), `whiteboard` (collaborative whiteboard tools), and `game` (multiplayer games).

conference	Used to describe elements in the conferencing (talk between three or more entities) world, such as private and public rooms. Subtypes of this category include `private` (private chat rooms), `irc` (IRC rooms), and `url` (for web-based conferences).
headline	Stock-ticker-style notification systems can be described using this category. Subtypes already defined include `rss`, `logger`, and `notice`.
item	A category placeholder, to effect hierarchies and lists in a `jabber:iq:browse` structure. You can fall back to the this category for representation of pretty much any type of information in a navigable drill-down fashion.
keyword	IRC-style utilities that are invoked from a chat-input command line; so-called *keyword* services such as dictionary lookups (subtype `dictionary`), DNS resolution (subtype `dns`), and FAQ answers (subtype `faq`) have their category in the `jabber:iq:browse` world.
render	Translation services such as English to French (subtype `en2fr`), or spelling tools (subtype `spell`) are defined in this category.
service	Maps to traditional Jabber services, such as IM transports and gateways to other systems, user directories, and so on. Typical subtypes within this category are `irc` (IRC gateway), `aim` (AIM transport), and `jud` (Jabber User Directory).
user	Various addressable elements of users, such as their clients (subtype `client`), inbox mechanisms (subtype `inbox`) and so on, find themselves in this category.

The categories listed in Table 5a-1 are not exhaustive; the `jabber:iq:browse` namespace and the browsing idea was introduced with version 1.4 of the Jabber server and is still evolving. The same goes for the category subtypes.

Any particular browsable entity can be described using the combination of the category and subtype. For example, `user/client` can be used much in the same way as MIME-types. Following the MIME meme further, we can define our own subtypes on the fly and specify them with an `x-` prefix, such as `user/x-schedule`.

Indeed, the browsing description model of category/subtype follows the MIME model; in places the category is often referred to in Jabber documentation as the *JID-type*. The JID is critical to browsing; it is a required attribute of all entities described in a `jabber:iq:browse`-based hierarchy. The JID is the key to navigating the hierarchy structure.

Earlier in this section we saw the results of making a query in the `jabber:iq:agents` namespace to retrieve information on the services available on a Jabber server. Now let's have a look at the a similar query using the `jabber:iq:browse` namespace:

```
SEND: <iq type='get' to='yak'>
        <query xmlns='jabber:iq:browse'/>
      </iq>

RECV: <iq type='result' to='dj@yak/home' from='yak'>
        <service name='Jabber Server' type='jabber'
                                 xmlns='jabber:iq:browse' jid='yak'>
          <conference name='yak Conferencing'
                      type='public' jid='conference.yak'/>
          <service name='yak User Directory' type='jud' jid='jud.yak'>
            <ns>jabber:iq:search</ns>
            <ns>jabber:iq:register</ns>
          </service>
          <service name='User Directory (Browsable)'
                           type='jud' jid='jud.merlix/users'/>
        </service>
```

```
      </iq>
```

Notice how the information returned forms a hierarchy. The outermost item in the browse results represents the Jabber server as a whole (with the JID `yak`) and contains subitems that are services of that Jabber server (the `yak Conferencing` service, and the two forms of the JUD).

How many levels of hierarchy can we expect to receive (as a browsing information consumer) or provide (as a browsing information provider) in any given situation? It really depends on the application situation. and the balance you want to achieve between shallow hierarchy responses and many IQ calls for navigational descent (light extensions but more traffic) and deeper hierarchy responses and few IQ calls for navigational descent (heavier extensions but less traffic).

Descending the browse hierarchy from an LDAP reflector

As illustration, let's have a look how we might perform a hierarchy descent in the navigation of LDAP information provided by a custom LDAP reflector in a `jabber:iq:browse` context. Each time, the link to the next level is via the item's JID, which is the target of the browse query.

First, we send an initial query:

```
SEND: <iq type="get" id="browser_JCOM_15" to="ldap.yak">
        <query xmlns="jabber:iq:browse"></query>
      </iq>
```

In answer to the initial query to what is effectively the LDAP root represented by the JID of the LDAP component itself (`ldap.yak`, no username prefix), the initial hierarchy level containing *People* and *Groups* is returned, wrapped in a pseudo root:

```
RECV: <iq type='result' to='dj@yak/winjab' from='ldap.yak' id='browser_JCOM_15'>
        <item name='root entry' xmlns='jabber:iq:browse' jid='ldap.yak'>
          <item name='ou=People' jid='ou=People@ldap.yak'/>
          <item name='ou=Groups' jid='ou=Groups@ldap.yak'/>
        </item>
      </iq>
```

We see the items presented to us and choose to descend the path marked *Groups*; our second browse request is made to the JID that represents that item, `ou=Groups@ldap.yak`.

```
SEND: <iq type="get" id="browser_JCOM_17" to="ou=People@ldap.yak">
        <query xmlns="jabber:iq:browse"></query>
      </iq>
```

The LDAP reflector component receives the IQ packet addressed to the JID `ou=People@ldap.yak` and interprets the username part of the JID (`ou=People`) as an LDAP RDN (*relative distinguished name*, a form of key within an LDAP structure that's further qualified by a common suffix), which returns the appropriate information from the next level in the LDAP hierarchy: the countries.

```
RECV: <iq type='result' to='dj@yak/winjab' from='ou=People@ldap.yak'
                                                  id='browser_JCOM_17'>
        <item name='ou=People' xmlns='jabber:iq:browse'
                                  jid='ou=People@ldap.yak'>
```

```
        <item name='ou=UK' jid='ou=UK,ou=People@ldap.yak'/>
        <item name='ou=France' jid='ou=France,ou=People@ldap.yak'/>
        <item name='ou=Germany' jid='ou=Germany,ou=People@ldap.yak'/>
      </item>
    </iq>
```

And so the descent continues, via the JID `ou=UK,ou=People@ldap.yak` that was specified as the unique identifier for that item (the country `UK`).

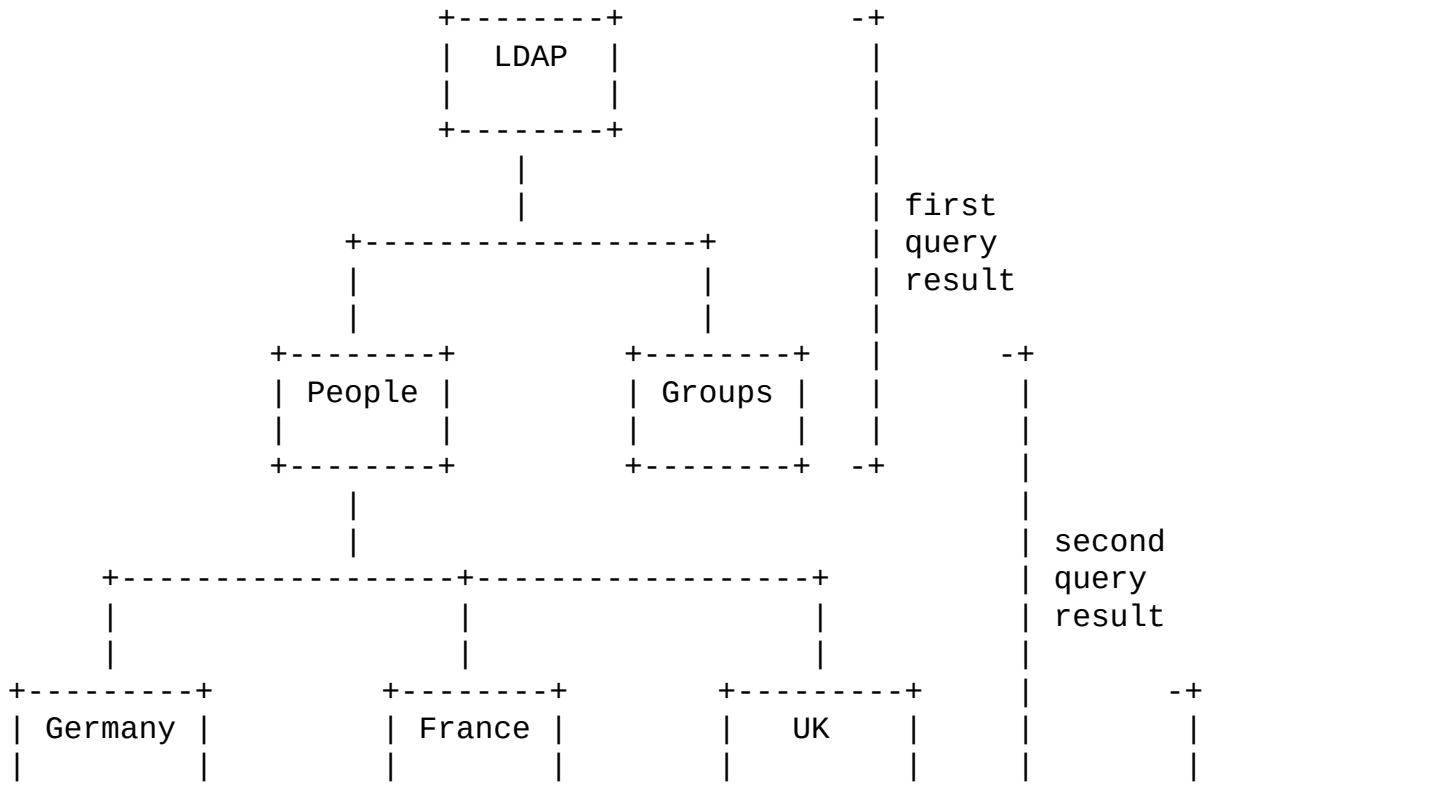
```
SEND: <iq type="get" id="browser_JCOM_18" to="ou=UK,ou=People@ldap.yak">
        <query xmlns="jabber:iq:browse"></query>
      </iq>
```

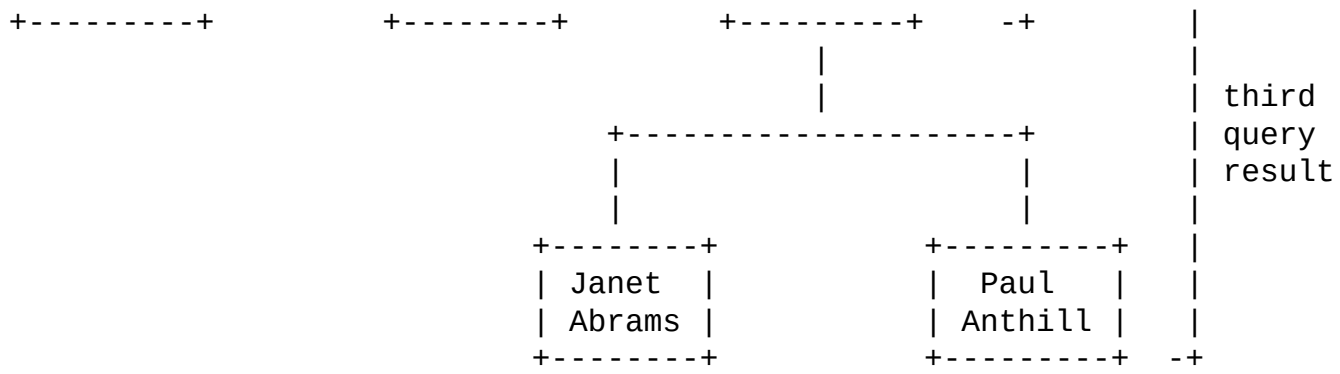
which continues:

```
RECV: <iq type='result' to='dj@yak/winjab' from='ou=UK,ou=People@ldap.yak'
                                          id='browser_JCOM_18'>
        <item name='ou=UK,ou=People' xmlns='jabber:iq:browse'
                                     jid='ou=UK,ou=People@ldap.yak'>
          <user name='cn=JanetAbrams' jid='JanetAbrams@yak'/>
          <user name='cn=PaulAnthill' jid='PaulAnthill@yak'/>
          ...
        </item>
      </iq>
```

The section of the actual LDAP hierarchy browsed is shown in Figure 5a-1.

Figure 5a-1. The LDAP hierarchy browsed in the section called *Descending the browse hierarchy from an LDAP reflector*





Browse data isn't just something that can be retrieved; like presence, it can be pushed to an entity when required. In the same way that an alert in the form of a `<message/>` element might arrive at a client unannounced, so might browse information also appear. This is referred to as *live browsing*, as the information that is pushed is effectively *live*.

The Conferencing service uses this mechanism to push information on room participants to a new joiner. As the browse information is enveloped in an IQ element, it makes the most sense to use a `type='set'` (it might help to consider the parallel with the HTTP verb 'POST' as introduced in Chapter 2) to push this information. And this is what happens, as seen in this excerpt from information sent to a client as a conference room is joined:

```
RECV: <iq type='set' to='qmacro@jabber.org/winjab'
                     from='jdev@conference.jabber.org'>
        <conference xmlns='jabber:iq:browse' name='Development Room'
                                             type='public'>
          <user name='piers'

jid='jdev@conference.jabber.org/445d4b864bd6988b52c5244e7aa50536125e373a'/>
          <user name='pgmillard'

jid='jdev@conference.jabber.org/1cffcbf43c75a007677246edc4c6f8d8c5e65b46'/>
          <user name='reatmon'

jid='jdev@conference.jabber.org/b3f3c19859de4d25c1362711b81e42e36c417309'/>
          ...
        </conference>
      </iq>
```

An example of a simple `jabber:iq:browse` implementation can be found in the section called *RSS punter* in Chapter 8.

`jabber:iq:conference`

The conferencing service provides facilities for entities to join rooms and chat to each other. The entry negotiations that take place between a room (via the service) and a potential participant are made using the `jabber:iq:conference` namespace. With this namespace, information on rooms can be requested, and attempts to enter rooms can be made.

The `jabber:iq:conference` namespace at work

Here we see a typical sequence of IQ elements that ensue in the entry negotiations for the JDEV room hosted by the conferencing service on `jabber.org`'s Jabber server.

Information on the JDEV room is requested:

```
SEND: <iq type="get" id="c2" to="jdev@conference.jabber.org">
        <query xmlns="jabber:iq:conference"/>
      </iq>
```

The conferencing service replies with the relevant information.

> : The JID to which the query was sent—`jdev@conference.jabber.org`—works in a similar way to the LDAP reflector in the section called *Descending the browse hierarchy from an LDAP reflector*. There's no real distinction between conferencing service *usernames* in the same way that there's a distinction in the JSM service, but that part of the JID is used to identity each room hosted by that service.

We see that the “friendly” name of the JDEV room is “Development Room” and that we need to specify a nickname in order to gain entry. There are no other requirements (such as a secret password) that would have been identified with an extra `<secret/>` tag in the results.

```
RECV: <iq type='result' id='c2' to='qmacro@jabber.org/hailsham'
                                from='jdev@conference.jabber.org'>
        <query xmlns='jabber:iq:conference'>
          <name>Development Room</name>
          <nick/>
        </query>
      </iq>
```

We choose a nickname, and send this back in an IQ set request.

Before doing this, we must send our presence to the room to invoke the Availability Tracker, which is described in the sidebar *Availability Tracker*. More information on joining and interacting with conference rooms can be found in the section called *Keyword assistant* in Chapter 8.

```
SEND: <presence to="jdev@conference.jabber.org"/>

SEND: <iq to="jdev@conference.jabber.org" type="set" id="c3">
        <query xmlns="jabber:iq:conference">
          <nick>qmacro</nick>
        </query>
      </iq>
```

The Conferencing service acknowledges our entry to the room with our chosen nickname, having assigned us an anonymous handle in the `<id/>` tag:

```
RECV: <iq to='qmacro@jabber.org/winjab' type='result' id='c3'
                              from='jdev@conference.jabber.org'>
        <query xmlns='jabber:iq:conference'>
          <nick>qmacro</nick>
          <name>Development Room</name>

<id>jdev@conference.jabber.org/650e81de0fcc265d43357ec09ded0ce421ecdff5</id>
        </query>
      </iq>
```

Closely linked with the `jabber:iq:conference` namespace is the `jabber:iq:browse` namespace, which is also used as a conduit for room-specific information and activity; see the section called *`jabber:iq:browse`*.

`jabber:iq:gateway`

The `jabber:iq:gateway` namespace is used to envelope a utility mechanism for converting external system identifiers (usernames, and so on) to a JID equivalent. The requirement for this grew out of the transport services to other IM systems (AIM, Yahoo!, and so on), which have their own formats for user identification.

First, we know whether a service offers this utility from the namespace list that is returned if we *browse* that service. The next section shows how this might be done with the AIM Transport service.

Discovering and using the AIM Transport's `jabber:iq:gateway` utility

By browsing a service, we can tell whether it supports the `jabber:iq:gateway` utility or not:

```
SEND: <iq type="get" id="aim1" to='aim.jabber.org'>
        <query xmlns="jabber:iq:browse"/>
      </iq>

RECV: <iq type='result' id='aim1' to='qmacro@jabber.org/winjab'
                                          from='aim.jabber.org'>
        <service xmlns='jabber:iq:browse' type='jabber'
                          jid='aim.jabber.org' name='AIM Transport'>
          <ns>jabber:iq:register</ns>
          <ns>jabber:iq:gateway</ns>
        </service>
      </iq>
```

We can now avail ourselves of this utility, to convert an AIM screenname *test ScreenName* to the equivalent JID to be used (in relation to the AIM transport service) in a Jabber context:

```
SEND: <iq type='get' to='aim.jabber.org' id='conv5'>
        <query xmlns='jabber:iq:gateway'/>
      </iq>

RECV: <iq type='result' to='qmacro@jabber.org/hailsham' id='conv5'
                                                 from='aim.jabber.org'>
        <query xmlns='jabber:iq:gateway'>
          <desc>Enter the user's screenname</desc>
          <prompt/>
        </query>
      </iq>
```

We can reply, with an IQ set, with our screen name:

```
SEND: <iq type='set' to='aim.jabber.org' id='conf6'>
        <query xmlns='jabber:iq:gateway'>
          <prompt>test ScreenName</prompt>
```

```
      </query>
    </iq>
```

and receive the result of the transport-specific to JID conversion:

```
RECV: <iq type='result' to='qmacro@jabber.org/Work' id='conf6'
                                                 from='aim.jabber.org'>
        <query xmlns='jabber:iq:gateway'>
          <prompt>testScreenName@aim.jabber.org</prompt>
        </query>
      </iq>
```

jabber:iq:last

Like the `jabber:iq:time` and `jabber:iq:version` namespaces, the `jabber:iq:last` namespace allows a simple query on uptime or idletime to be made on clients and servers alike.

Elapsed time information, in seconds, is returned in response to queries in the `jabber:iq:last` namespace. If the query is made of a server element (the Jabber server itself, or a component connected to that server), then the information returned represents the time since that element started, that is, the uptime:

```
SEND: <iq type='get' to='yak'>
        <query xmlns='jabber:iq:last'></query>
      </iq>

RECV: <iq type='result' to='dj@yak/Work' from='yak'>
        <query xmlns='jabber:iq:last' seconds='2339811'/>
      </iq>
```

Not all components support the `jabber:iq:last` namespace; then again, in many cases the components—certainly those that are connected with the *library load* mechanism (see Chapter 4)—will have the same uptime as the Jabber server they're connected to. In other cases, for STDIO and TCP socket connected components that can be attached while the Jabber server is running, the uptime may be less. [4]

When a *client* disconnects, the last (un)availability information in the closing `<presence/>` element is stored for that user, along with the current time:

```
SEND: <presence type='unavailable'>
        <status>Gone home for the evening!</status>
      </presence>
```

Making a `jabber:iq:last` based query on a user's JID will return the information that was stored from the `<status/>` tag as well as the number of seconds representing the elapsed time since that disconnection (as a difference between the time that the query was made and the time stored for that user):

```
SEND: <iq type='get' to='dj@yak' id='lastq'>
        <query xmlns='jabber:iq:last'/>
      </iq>

RECV: <iq type='result' to='sabine@yak/Work' id='lastq' from='dj@yak'>
```

```
        <query xmlns='jabber:iq:last' seconds='4521'>
          Gone home for the evening!
        </query>
      </iq>
```

Notice that the JID of the user being queried is `dj@yak` and not, for example, `dj@yak/Work`. This, of course, is because the user was still disconnected. The query was addressed to the user with no resource specified, and was answered on behalf of the user by the server (by the `mod_last` module—the same module that looks after storing this information). In a disconnected context, a resource is not appropriate for a user's JID (in the JSM); they are only found in a connected context.

The `jabber:iq:last` is also designed to support a similar client-targeted query, this time requesting information on how long it has been since the user of that client was active (sent a message, changed their presence, and so on). In contrast to the previous `jabber:iq:last` query type, this query is designed to be made to a connected user.

```
SEND: <iq type="get" to="dj@yak/Work">
        <query xmlns='jabber:iq:last'/>
      </iq>

RECV: <iq type='result' from='dj@yak/Work'>
        <query xmlns='jabber:iq:last' seconds='19'/>
      </iq>
```

Here we see that the user is using a client that supports this type of `jabber:iq:last` query and was last active 19 seconds ago.

`jabber:iq:oob`

We've already seen a form of the `oob`—"Out Of Band" [5] —namespace in action, in the imaginary conversation in Chapter 1, where `jabber:x:oob` was used to pass information about a third-party file location (in the form of a Uniform Resource Locator (URL).

The `jabber:iq:oob` namespace is used for pretty much the same thing, except that its usage describes a very simple handshake between two Jabber clients to exchange a file between themselves. (Yes, *real* peer-to-peer for the purists.) Typically, the client sending the file will only start listening for HTTP requests at the beginning of the transfer process, and stop listening at the end of the transfer process. The handshake is used to coordinate the process.

The sender initiates the process by making the file available via HTTP on a specific (non-standard) port, and notifying the recipient of the URL:

```
SEND: <iq type='set' to='sabine@yak/winjab' id='file_2'>
        <query xmlns='jabber:iq:oob'>
          <url>http://192.168.0.7:5600/meetingnotes.txt</url>
          <desc>Meeting Notes</desc>
        </query>
      </iq>
```

The recipient retrieves the file, and notifies the sender when the transfer is complete.

```
RECV: <iq type='result' to='dj@yak/Work' id='file_2' from='sabine@yak/winjab'/>
```

jabber:iq:private

The jabber:iq:private namespace is traditionally a way of storing user-defined data that should be kept private. Persistency across sessions is achieved by storing the data in the user's records on the server. The data is, of course, formatted in XML.

> **How private is private?:** Private data stored by a user is only accessible to that user. Remember, however, that the private data is stored on the server. Unencrypted. If you're paranoid, encrypt it before storing it.

Example 5a-1 shows a typical use. The JIM client stores countless user preferences on a per-user basis using this namespace. Once a user has connected and authenticated with a Jabber server, those user preferences are retrieved and used by the client to customize the settings.

Example 5a-1. JIM retrieves user preferences stored in a jabber:iq:private namespace

```
SEND: <iq id="jabberim:prefs3860" type="get">
        <query xmlns="jabber:iq:private">
          <jabberIM xmlns="jabberim:prefs"/>
        </query>
      </iq>

RECV: <iq id='jabberim:prefs3860' type='result' from='dj@yak/Work'>
        <query xmlns='jabber:iq:private'>
          <jabberim xmlns='jabberim:prefs'
                    UseAutoAway='true'
                    AwayTime='5'
                    XATime='30'
                    AwayStatus='Away (auto)'
                    XAStatus='Ext. Away (auto)'
                    WizardShown='false'
                    ... >
          </jabberim>
        </query>
      </iq>
```

In this example, you can see that a private namespace is used to qualify the particular chunk of stored data, jabberim:prefs. Also of interest is the difference between the tags—<jabberIM/> in the retrieval request and <jabberim/> in the response. Again we see evidence of an XML usage convention previously seen (for example, in the Jabber server component configuration stanzas; see Chapter 4 for more details). The namespace itself is critical, not the enclosing tag name. If the preferences were originally stored using a tag name of <jabberim/> then that's how it will be stored, and returned.

To add (or change) private data, use the namespace in an IQ set context:

```
SEND: <iq id="private-s3" type="set">
        <query xmlns="jabber:iq:private">
          <reminders xmlns="cal:events">
            <event date='20010617'>Father's Day</event>
          </reminders>
       </query>
```

```
    </iq>
```

Due to the way the `jabber:iq:private` storage mechanism is currently implemented, you can only interact with *one* private namespace-qualified chunk. In other words, a private store request like this:

```
SEND: <iq id="private-s4" type="set">
        <query xmlns="jabber:iq:private">
          <reminders xmlns="cal:events">
            <event date='20010617'>Father's Day</event>
          </reminders>
          <favorites xmlns='url:favourites'>
            <fav url='http://dev.jabber.org'>Jabber DevZone</fav>
            <fav url='http://www.scripting.com'>Scripting News</fav>
          </favorites>
        </query>
    </iq>
```

would only result in the storage of the `cal:events` chunk. The `url:favorites` chunk would be ignored.

In the 1.4.1 release of the Jabber server, the `mod_xml` JSM module that services the `jabber:iq:private` namespace has been extended to allow this server-side storage to encompass nonprivate (i.e., publically accessible) user data. The namespace in this case is, fittingly *not* `jabber:iq:private`. It can be anything you wish, provided that it doesn't encroach on the standard Jabber namespace names—`jabber:*` and `vcard-temp` are not allowed—however, anything else goes. [6]

The idea of publically accessible data is just that; you can make information available to your fellow Jabber users (share URLs, contact lists, and so on). Of course, this sharing is only one way; you write and others can only read. But how do they find out *what* you've made available for them to read? The namespaces of any data stored publically (i.e., any namespace except for `jabber:iq:private`) are returned by the Jabber server acting on behalf of the user in response to a `jabber:iq:browse` request to that user's JID. [7]

Let's have a look at this in action. We'll also have a peek at how the storage of the public and private information is structured in the user's spool file on the server to understand how this works. [8] In addition to the Father's Day event that was stored privately in the previous example, we can also set some favorite URLs in a publically accessible namespace and receive an acknowledgement of successful storage from the server:

```
SEND: <iq type='set' id='setfavs'>
        <query xmlns='dj:public:favorites'>
          <item url='http://dev.jabber.org'>Jabber DevZone</item>
          <item url='http://www.scripting.com'>Scripting News</item>
        </query>
      </iq>

RECV: <iq type='result' from='dj@yak/Work' to='a1@yak/Work' id='setfavs'/>
```

Now, the relevant section of `dj@yak`'s spool file on the server looks something like that shown in Example 5a-2.

Example 5a-2. Section of user's spool storage showing public and private data

```
...

<foo xmlns='jabber:xdb:nslist' xdbns='jabber:xdb:nslist'>
```

```
  <ns type='private'>cal:events</ns>
  <ns>dj:public:favourites</ns>
</foo>

<reminders xmlns='cal:events' j_private_flag='1' xdbns='cal:events'>
  <event date='20010617'>Father's Day</event>
</reminders>

<query xmlns='dj:public:favorites' xdbns='dj:public:favourites'>
  <item url='http://dev.jabber.org'>Jabber DevZone</item>
  <item url='http://www.scripting.com'>Scripting News</item>
</query>

...
```

There are a few things to note in this example:

- The `jabber:xdb:nslist` namespace maintains a list of namespaces containing information stored for private and public reference.
- The `private` namespaces are marked in this list with a `type='private'` attribute.
- There is an additional flag (`j_private_flag='1'`) which is held as an attribute of each of the privately stored fragments.
- Otherwise the information is stored exactly as it was set (additional `xdbns` attributes related to the XDB storage mechanisms notwithstanding).

The namespaces (`<ns/>` tags) in the `jabber:xdb:nslist` qualified list are returned in any browse request to that user:

```
SEND: <iq type='get' to='dj@yak'>
        <query xmlns='jabber:iq:browse'/>
      </iq>

RECV: <iq type='result' to='sabine@yak/Work' from='dj@yak'>
        <user name='DJ Adams' xmlns='jabber:iq:browse' jid='dj@yak'>
          <ns>dj:public:favorites</ns>
        </user>
      </iq>
```

and can be subsequently retrieved by anyone:

```
SEND: <iq type='get' to='dj@yak'>
        <query xmlns='dj:public:favorites'/>
      </iq>

RECV: <iq type='result' to='sabine@yak/Work' from='dj@yak'>
        <query xmlns='dj:public:favorites'>
          <item url='http://dev.jabber.org'>Jabber DevZone</item>
          <item url='http://www.scripting.com'>Scripting News</item>
        </query>
      </iq>
```

Publically stored data can contain multiple fragments qualified by different namespaces, such as:

```
SEND: <iq type='set'>
        <query xmlns='my:resume'>
          <education xmlns='resume:education'>
            <degree type='BA'>Classics</degree>
          </education>
          <employment xmlns='work:clients'>
            <client from='2001'>Author, O'Reilly &amp;amp; Associates, Inc.</client>
            <client from='1999'>Deluxe Video Services</client>
            <client from='1996'>Andersen Consulting</client>
            ...
          </employment>
        </query>
      </iq>
```

However, the retrieval *resolution* is still limited to all of the fragment defined by the top-level namespace (`my:resume` in this case).

`jabber:iq:register`

As the name suggests, the `jabber:iq:register` namespace is used to conduct registration exchanges between the client and server. The most obvious example of this is to create (*register*) a new user on the Jabber server. Chapter 6 covers user registration in detail, so here we'll look at how to use the namespace to add or change an entry in the JUD.

First we request the fields for registration with an IQ `get`:

```
SEND: <iq type='get' to='jud.yak' id='jud-2'>
        <query xmlns='jabber:iq:register'/>
      </iq>

RECV: <iq type='result' to='dj@yak/Work' from='jud.yak' id='jud-2'>
        <query xmlns='jabber:iq:register'>
          <instructions>
            Complete the form to submit your searchable attributes
            in the Jabber User Directory
          </instructions>
         <name/>
         <first/>
         <last/>
         <nick/>
         <email/>
       </query>
     </iq>
```

and then send an IQ `set` to set our information:

```
SENT: <iq type='set' to='jud.yak' id='jud-3'>
        <query xmlns='jabber:iq:register'>
          <name>DJ Adams</name>
          <first>DJ</first>
          <last>Adams</last>
```

```
          <nick>qmacro</nick>
          <email>dj.adams@pobox.com</email>
        </query>
      </iq>

RECV: <iq type='result' to='dj@yak/Work' from='jud.yak'/>
```

Making requests for form fields: This idiom—making a request to a service to return the fields appropriate for completion—is common in Jabber and is worth bearing in mind if you're intending to build a Jabber client. The nature of the form field requests means that the client application has to be flexible and accommodating, to bend itself around the dynamic server.

Services offering a registration mechanism are identifiable in the list returned from a `jabber:iq:agents` or a `jabber:iq:browse` query, as shown in Example 5a-3.

Example 5a-3. An `agents` or `browse` query reveals registration and search mechanisms

```
RECV: <iq to='dj@yak/Work' type='result' from='yak'>
        <query xmlns='jabber:iq:agents'>
          <agent jid='jud.yak'>
            <name>yak JUD (0.4)</name>
            <service>jud</service>
            <search/>
            <register/>
          </agent>
          ...
        </query>
      </iq>

      ...

RECV: <iq type='result' to='dj@yak/Work' from='yak'>
        <service xmlns='jabber:iq:browse' type='jabber' jid='yak'
                                                              name='Jabber
Server'>
          <service type='jud' jid='jud.yak' name='yak JUD (0.4)'>
            <ns>jabber:iq:search</ns>
            <ns>jabber:iq:register</ns>
          </service>
          ...
        </service>
      </iq>
```

There are a couple of extra elements that are fairly common across different implementations of the `jabber:iq:register` namespace:

```
<remove/>
```

When sent with an IQ set request, the `<remove/>` tag requests that the registration be cancelled, revoked, or reversed.

```
<registered/>
```

When received in an IQ result, the `<registered/>` tag signifies that registration has already been made with the service, and any further registration IQ sets will serve to modify the current registration details.

Another example of registration using the `jabber:iq:register` namespace is shown in the section called *RSS punter* in Chapter 8.

jabber:iq:roster

In the section called *Presence Subscription* in Chapter 5, we looked at the presence subscription mechanism used to coordinate and record information about the relationships between users and how they exchange availability information. This mechanism revolves around certain types of `<presence/>` packets and storage of information in the users' *rosters*.

The roster structure is managed within the `jabber:iq:roster` namespace. Clients make roster requests when they connect to the Jabber server, to pull down the roster which is stored server side. They also update the roster to add, change, or remove entries. However, roster updates aren't limited to just the client; there are certain attributes within each roster item that are maintained by the server, in response to presence subscription activity.

The roster in Example 5a-4 contains five items. Three are friends, grouped together using `<group>Friends</group>`, which is used by clients to build the roster item display in a structured (hierarchical) way.

Example 5a-4. A typical roster

```
<query xmlns='jabber:iq:roster'>
  <item jid='shiels@jabber.org' subscription='both' name='Robert'>
    <group>Friends</group>
  </item>
  <item jid='piers@jabber.org' subscription='both' name='Piers'>
    <group>Friends</group>
  </item>
  <item jid='sabine@pipetree.com' subscription='to' name='Sabine'>
    <group>Friends</group>
  </item>
  <item jid='jim@company-a.com' subscription='from' name='Jim'>
    <group>Work</group>
  </item>
  <item jid='jim@company-b.com' subscription='none'
                                        ask='subscribe' name='John'>
    <group>Work</group>
  </item>
</query>
```

The `subscription` attribute is used to store the presence subscription state between the roster owner and the particular item that holds that attribute. With two of the Friends, Robert and Piers, the roster owner is subscribed to each of their presences, and they are each subscribed to the presence of the roster owner. This is denoted by the `both` value, which means that the presence subscription flows both ways. Where the attribute has the value `to` (as in Sabine's case), or `from` (Jim's case), the subscription flows in only one direction. Here, the roster owner is subscribed *to* Sabine's presence (but Sabine is not subscribed to the roster owner's presence), and Jim is subscribed to the roster owner's presence (i.e., the roster owner has a presence subscription *from* Jim). [9]

Where the value of the `subscription` attribute is `none`, neither party has a subscription to the other. In this case, a further

attribute `ask` may be used to reflect that a presence subscription request is in progress. [10] The `ask` attribute can have one of two values:

`subscribe`

This attribute sends a request to subscribe to a user's presence.

`unsubscribe`

Sends a request to unsubscribe from a user's presence.

In both cases, these requests are sent using a `<presence/>` element with an appropriate value for the `type` attribute.

The server, is responsible for maintaining the `subscription` and `ask` attributes; the client may maintain all the other elements. If an item is updated by the server—for example, as a result of a correspondent accepting a previous subscription request—the server will push the updated item to the client with an IQ set:

```
SEND: <iq type='set'>
        <query xmlns='jabber:iq:roster'>
          <item jid='john@company-b.com' subscription='to' name='John'/>
        </query>
      </iq>
```

Here, John has accepted the roster owner's subscription request by sending the following:

```
<presence to='dj@yak/Work' type='subscribed'/>
```

The server will update the roster item accordingly by removing the `ask='subscribe'` and setting the value of the `subscription` attribute to `to`.

jabber:iq:search

The `jabber:iq:search` is closely related to the `jabber:iq:register` namespace, in that the dance steps are pretty much the same. As with `jabber:iq:register`, you can discover which entities (usually server components) support search features from the results of an agents or browse query, as shown in Example 5a-3.

Also, as with the `jabber:iq:register` namespace, the fields to be used in the interaction are first retrieved with an IQ get qualified by the `jabber:iq:search` namespace. Here we see an example of that with the JUD running on the `jabber.org` server:

```
SEND: <iq type='get' to='users.jabber.org' id='800'>
        <query xmlns='jabber:iq:search'/>
      </iq>

RECV: <iq type='result' from='users.jabber.org'
          to='qmacro@jabber.org/laptop' id='800'>
        <query xmlns='jabber:iq:search'>
          <instructions>
            Fill in a field to search for any matching Jabber User
          </instructions>
```

```
          <first/>
          <last/>
          <nick/>
          <email/>
        </query>
      </iq>
```

To continue the similarity theme, an IQ set is used to submit the search, sending back a value (or values) in the fields like this: `<email>pipetree.com</email>`.

The only exciting feature of the `jabber:iq:search` namespace is perhaps the way it can return results in response to an IQ set. This depends on the component, and how the feature is implemented.

While the JUD component will return all results in one IQ element:

```
RECV: <iq type='result' from='users.jabber.org' to='qmacro@jabber.org/laptop'>
        <query xmlns='jabber:iq:search'>
          <item jid='qmacro@jabber.org'>
            <name>DJ Adams</name>
            <first>DJ</first>
            <last>Adams</last>
            <nick>qmacro</nick>
            <email>dj@pipetree.com</email>
          </item>
          <item jid='piers@jabber.org'>
            <name>Piers Harding</name>
            <first>Piers</first>
            <last>Harding</last>
            <nick>pxh</nick>
            <email>piers@pipetree.com</email>
          </item>
          ...
        </query>
      </iq>
```

The component providing transport services to the ICQ IM system returns the results item-by-item:

```
RECV: <iq type='set' from='icq.jabber.org' id='icqs8'
          to='qmacro@jabber.org/laptop'>
        <query xmlns='jabber:iq:search'>
          <item jid='4711471@icq.jabber.org'>
            <given>DJ</given>
            <family>Adams</family>
            <nick>qmacro</nick>
            <email>dj@pipetree.com</email>
          </item>
        </query>
      </iq>

RECV: <iq type='set' from='icq.jabber.org' id='icqs8'
          to='qmacro@jabber.org/laptop'>
```

```
        <query xmlns='jabber:iq:search'>
          <item jid='1234567@icq.jabber.org'>
            <given>Piers</given>
            <family>Harding</family>
            <nick>pxh</nick>
            <email>piers@pipetree.com</email>
          </item>
        </query>
      </iq>
```

The component signals the end of the search results with an empty IQ result element:

```
RECV: <iq type='result' from='icq.jabber.org' id='icqs8'
           to='qmacro@jabber.org/laptop'>
        <query xmlns='jabber:iq:search'/>
      </iq>
```

jabber:iq:time

The `jabber:iq:time` namespace qualifies an info/query-based conversation to make or respond to a query on time information.

To query the time at a particular entity, an `<iq/>` `get` request is sent like this:

```
SEND: <iq type='get' id='time_19' to='conference.yak'>
        <query xmlns='jabber:iq:time'/>
      </iq>
```

There are three pieces of information returned in response to such a query, the time in UTC (coordinated universal time) format, the local timezone, and a nice display version of the local time:

```
RECV: <iq type='result' id='time_19' to='sabine@yak/Work'
          from='conference.yak'>
        <query xmlns='jabber:iq:time'>
          <utc>20010520T08:55:38</utc>
          <tz>GMT</tz>
          <display>Sun May 20 09:55:38 2001</display>
        </query>
      </iq>
```

The format of the `<tz/>` and `<display>` tags is not fixed. While this is what the *Conferencing* service returns, a response from the JIM client would give “GMT Standard Time” and “20/05/01 09:55:38,” respectively.

Incidentally, if you've tried to send a `jabber:iq:time` query to a client and received an error in response, check the description in the sidebar *Specifying Clients as Query Targets*.

> **Different components running in different timezones:** If you consider that certain components can be connected to the Jabber backbone but be running on different hosts, communicating over TCP socket connections, as described in the section called *Server Constellations* in Chapter 4, this may be more useful than you initially think.

Specifying Clients as Query Targets

Many of these namespaces qualify queries that make just as much sense sent to a Jabber client as sent to a Jabber server or service. But if you don't compose the recipient JID correctly, you could end up with an unexpected response.

Let's say you want to query the local time on Piers' client, which is connected to the Jabber server at `jabber.org`:

```
SEND: <iq type='get' id='time_21' to='piers@jabber.org'>
        <query xmlns='jabber:iq:time'/>
      </iq>
```

You'll likely get a response like this:

```
RECV: <iq type='error' id='time_21' from='piers@jabber.org'
          to='qmacro@jabber.org/home'>
        <query xmlns='jabber:iq:time'/>
        <error code='503'>Service Unavailable</error>
     </iq>
```

And why? Because this is a query that's addressed not to a particular client or session. While `<message/>` packets can be addressed to a user JID in the form `username@hostname` and will be sent to the "primary" session according to presence priority, the recipients of `<iq/>` packets must be specified exactly. As we want to find out the time at Piers' client, which has the associated resource `desktop`, we must specify that resource in the JID:

```
SEND: <iq type='get' id='time_21' to='piers@jabber.org/desktop'>
        <query xmlns='jabber:iq:time'/>
      </iq>
```

This will give us the response we're looking for:

```
RECV: <iq type='result' id='time_21' to='qmacro@jabber/home'
        from='piers@jabber.org/desktop'>
        <query xmlns='jabber:iq:time'/>
          <utc>20010520T11:25:58</utc>
          <tz>CET</tz>
          <display>Sun May 20 13:25:58 2001</display>
        </query>
      </iq>
```

jabber:iq:version

Similar to the `jabber:iq:time` namespace, the `:version` namespace is used to make and respond to queries regarding the version of the particular piece of software being addressed. The query is formulated like this:

```
SEND: <iq type='get' id='ver-a' to='JID'>
        <query xmlns='jabber:iq:version'/>
```

```
</iq>
```

Responses depend on the entity being queried. Here are responses from three different entities:

- a client (**sjabber**):

```
RECV: <iq type='result' to='dj@yak/Work' from='sabine@yak/sjabber'>
        <query xmlns='jabber:iq:version'>
          <name>sjabber</name>
          <version>0.4</version>
          <os>linux</os>
        </query>
      </iq>
```

- The Jabber server itself (well, the JSM):

```
RECV: <iq type='result' to='dj@yak/Work' from='sabine@yak/sjabber'>
<query xmlns='jabber:iq:version'>
        <name>jsm</name>
        <version>1.4.1</version>
        <os>linux 2.2.12-45SAP</os>
</query>
</iq>
```

- And the JUD component:

```
RECV: <iq type='result' to='dj@yak/Work' from='sabine@yak/sjabber'>
<query xmlns='jabber:iq:version'>
        <name>jud</name>
        <version>0.4</version>
        <os>linux 2.2.12-45SAP</os>
</query>
</iq>
```

Another example of registration using the `jabber:iq:version` namespace is shown in the section called *RSS punter* in Chapter 8.

Notes

[1] The `mod_agents` module within the JSM.

[2] By “Jabber server,” we're referring to the JSM.

[3] Remember that the Jabber namespaces used to qualify `<iq/>` queries begin `jabber:iq`, while Jabber namespaces used to qualify general payloads to any of `<message/>`, `<iq/>`, and `<presence/>` begin `jabber:x`.

[4] There is a feature in the Jabber server version 1.4.1 that allows dynamic starting and stopping of *library load* components, but it is not completely developed at the moment.

[5] The word "band" here refers to the *bandwidth*, or connection, between the client and the server. The point of an *out of band* connection is that it's independent of that client-to-server connection (it typically is a connection from one client directly to another), and so doesn't impact the traffic, or bandwidth on that connection. This makes sense when you consider that out of band connections are typically used for exchanging large volumes of data, such as binary files.

[6] The reason for the `vcard-temp` namespace name is that there is an emerging but nevertheless not-yet-established standard for vCard data. Until that standard is established, the Jabber server developers have decided to handle this format in a temporary way.

[7] That is, the JID without a specified resource; otherwise it would be passed on by the server to be handled by the client connection with that resource.

[8] The location of the spool files is defined in the *xdb* component instances configuration—see the section called *Component instance: xdb* in Chapter 4.

[9] In colloquial Jabber terms, Jim is known as a *lurker*, as he knows about the roster owner's availability, without the roster owner knowing about his.

[10] “Pending,” in Jabber client parlance.

The X Namespaces

While the IQ namespaces are used in exchanging structured information in semi-formalized conversations, the X namespaces are more ad-hoc extensions that add value, context, and information to any type of packet.

jabber:x:autoupdate

The `jabber:x:autoupdate` namespace is used to carry information on where new version information can be found. Details and an example of this namespace's usage can be found in the description for the IQ version; `jabber:iq:autoupdate` in the section called *jabber:iq:autoupdate*.

jabber:x:conference

Just as `jabber:x:autoupdate` is related to its big brother `jabber:iq:autoupdate`, so too is the `jabber:x:conference` namespace related to `jabber:iq:conference`. The `<x/>` version of the IQ conference namespace is used to convey information about a conferencing room, usually attached to a message:

```
SEND: <message id='2113' to='robert@company-a.com'>
        <subject>Design Meeting</subject>
        <body>Robert - you're supposed to be at the meeting now!</body>
        <x xmlns='jabber:x:conference' jid='meeting1@meetings.company-a.com'/>
      </message>
```

If supported by the receiving client, this will be interpreted as an invitation to the room and the procedure for joining the room (in this case identified with the JID `meeting1@conf.company-a.com`) can be automatically initiated:

```
SEND: <iq type='get' id='c4' to='meeting1@meetings.company-a.com'>
        <query xmlns='jabber:iq:conference'/>
      </iq>
```

jabber:x:delay

Packets—those conveying messages and presence—are sometimes sent to entities that aren't available at that particular moment. If they are stored offline, they are timestamped, in the `jabber:x:delay` namespace, so that when they are finally received, the recipient can use this information to determine when the packets were originally sent.

```
RECV: <message to='sabine@yak/Work' from='yak'>
        <subject>Weekend at last!</subject>
```

```
        <body>Don't forget Father's Day on Sunday!</body>
        <x xmlns='jabber:x:delay' from='yak/announce/motd'
           stamp='20010615T09:00:01'>Announced</x>
        </message>
```

In this Message Of The Day (MOTD) announcement, we see that as well as the `stamp` attribute showing that the announcement was sent out on the Friday morning before Father's Day, a short text description *Announced* is included.

The namespace is also used by the `xdb` component to timestamp various fragments of data. Here, we see that Sabine updated her user registration (using a `jabber:iq:register` query during her session) at the beginning of March.

```
<query xmlns='jabber:iq:register' xdbns='jabber:iq:register'>
  <name>S. Reitz-Adams</name>
  <email>sabine@reitz-adams.org</email>
  <x xmlns='jabber:x:delay' stamp='20010302T12:15:42'>updated</x>
</query>
```

jabber:x:encrypted

The relatively new `jabber:x:encrypted` namespace can be used to implement message-level security. It allows for the attachment of encrypted data Public Key Infrastructure (PKI) techniques, meaning that the data is encrypted using the message sender's private key, and decrypted by the recipient using the sender's public key.

```
SEND: <message to='john@company-a.com' id='m221'>
        <subject>Top Secret!</subject>
        <x xmlns='jabber:x:encrypted'>
hQEOA7ucqu53AhlPEAP/ZbU6oPnRAbIcUxMK1XRVnkgZ/Agtq1tcTQuEZxbpZLl4
lkKJlkjkJjghyri(*8ygHbkjaja09Ja09sajkls8aj1h0J/a=ajhsa9A1ska191l
        </x>
      </message>
```

The Jabber server itself does not currently provide any mechanisms for key management or exchange; the namespace is for the time being purely a marked container to hold encrypted data.

There are no components in the Jabber server that can encrypt or decrypt using PKI tools (such as PGP or GPG); this namespace can be seen as enabling *pure* P2P-encrypted communication in an Out Of Band (OOB) style, in the same way that the `jabber:iq:oob` namespace enables P2P file exchange.

jabber:x:envelope

The `jabber:x:envelope` namespace is used to describe more complex message addressing details than the simple `from` and `to` attributes in the `<message/>` packets.

The first area where this namespace is used is in server-side filtering, a service provided by the JSM's `mod_filter`

module. For example, when a user sets a filter rule to forward all messages to someone else while he's not around: [1]

```
<rule name="absent">
  <show>xa</show>
  <forward>john@company-b.com</forward>
</rule>
```

a message such as this:

```
<message id='284' to='janet@company-b.com'>
  <body>Can you give me the sales figures for last quarter?</body>
</message>
```

will be passed on to `john@company-b.com` in this form:

```
<message id='284' to='janet@company-b.com'>
  <body>Can you give me the sales figures for last quarter?</body>
  <x xmlns='jabber:x:envelope'>
    <forwardedby jid='janet@company-b.com'>
    <from jid='mark@company-b.com'>
    <cc jid='john@company-b.com'>
  </x>
</message>
```

to add context information on where the message has come from.

`jabber:x:event`

Message *events* allow clients and servers alike to add information about the receipt and handling of messages at various stages of delivery. There are currently four types of events supported in this namespace:

Composing

: The composing event, represented by the `<composing/>` tag within the `<x/>` extension qualified by the `jabber:x:event` namespace, can be set by clients and signifies that the user is composing a reply to the message just sent.

Delivered

: When a message is received by a client, it can set the `<delivered/>` flag to signify that the message has been received.

Displayed

: The displayed event is used to indicate that the message sent has been displayed to the user. This event is set, using the `<displayed/>` tag, by clients.

Offline

When a message recipient is not connected, the JSM module `mod_offline` will store the message and send it to the recipient when he is next available. This offline storage event can be set by the server, using the `<offline/>` tag, to notify the sender that the message has been stored offline.

The `<composing/>`, `<delivered/>` and `<displayed/>` events are client events and are only appropriate to be set by the client. The `<composing/>` event is a server event and only appropriate to be set by the server. In all cases, the events are only set if the message originator requests that they are. Adding a `jabber:x:event` extension to a message like this:

```
SEND: <message to='sabine@yak' id='M31'>
        <subject>Where are you?</subject>
        <body>Let me know when you get back</body>
        <x xmlns='jabber:x:event'>
          <displayed/>
          <offline/>
        </x>
      </message>
```

is the way to request that we get notified:

- If and when the message is stored offline by the server in the eventuality that `sabine` is not connected.
- When the message is eventually displayed to Sabine.

The former event will be set by the server, the latter by Sabine's client.

Setting an event is similar to requesting one, and uses the `jabber:x:event` namespace. Here is what we would receive if the server did store our message to Sabine offline:

```
RECV: <message to='dj@yak/Work' id='M31' from='sabine@yak'>
        <x xmlns='jabber:x:event'>
          <offline/>
          <id>M31</id>
        </x>
      </message>
```

That is, the `<offline/>` tag is sent back to the originator, along with an `<id/>` tag which contains the `id` of the message that was stored offline.

Example 5a-5 shows the receipt of a chat message and the `<composing/>` event being raised as Sabine starts to type her reply.

Example 5a-5. Raising and cancelling the `<composing/>` evevent

DJ sends a quick chat message to Sabine, and requests that his client be notified when she starts typing her response.

```
RECV: <message to='sabine@yak/Work' from='dj@yak/home' id='122' type='chat'>
        <body>hey, want a coffee?</body>
```

```
        <thread>ABAF6FC6521546A2B65B19EA391CB72A</thread>
        <x xmlns='jabber:x:event'>
          <composing/>
        </x>
      </message>
```

Sabine starts to type, which fires the `<composing/>` event:

```
SEND: <message from='sabine@yak/Work' to='dj@yak/home'>
        <x xmlns='jabber:x:event'>
          <composing/>
          <id>122</id>
        </x>
      </message>
```

Sabine is distracted, and her client decides she's abandoned the reply, and sends a cancellation of the `<composing/>` event, containing only the message `id` included when the event was originally raised.

```
SEND: <message from='sabine@yak/Work' to='dj@yak/home'>
        <x xmlns='jabber:x:event'>
          <id>122</id>
        </x>
      </message>
```

jabber:x:expire

The `jabber:x:expire` is a simple namespace to add an 'use by' or 'read by' stamp to a message. If you wish to send a message impose a finite lifetime upon it, attach an expiry extension thus:

```
SEND: <message to='piers@pipetree.com' id='M24'>
        <subject>Eccles cakes!</subject>
        <body>
          I've got some fresh Eccles cakes here, pop
          round for one before they all disappear!
        </body>
        <x xmlns='jabber:x:expire' seconds='1800'/>
      </message>
```

If John was not connected when the message was sent, the `mod_offline` module would hold the message ready for when he reconnects. But before storing it, an extra attribute (`stored`) is added with the current time. Example 5a-6 shows what the relevant section of John's spool file would look like.

Example 5a-6. Storage of an offline message with the `jabber:x:expire` extension

```
<foo xmlns='jabber:x:offline' xdbns='jabber:x:offline'>
  <message to='piers@pipetree.com' id='M24' from='dj@pipetree.com/kitchen'>
```

```
    <subject>Eccles cakes!</subject>
    <body>
      I've got some fresh Eccles cakes here, pop
      round for one before they all disappear!
    </body>
    <x xmlns='jabber:x:expire' seconds='600' stored='993038415'/>
    <x xmlns='jabber:x:delay' from='dj@pipetree.com' stamp='20010620T12:00:15'>
      Offline Storage
    </x>
  </message>
</foo>
```

When John reconnects, `mod_offline` retrieves the message and compares the current time with the value in the `stored` attribute. If the difference exceeds the desired lifetime of the message, as specified in the `seconds` attribute, the message is discarded. Otherwise, the `seconds` attribute value is reduced to reflect the amount of time the message sat in storage, the `stored` attribute is removed, and the message is sent to John.

Furthermore, if John's client supports it, a further check of the message's lifetime can be made before display, in case the message was stored in an inbox-style mechanism. (With John's luck, he probably missed out on the Eccles cake.)

jabber:x:oob

We've already seen the `jabber:x:oob` in action earlier in the book. It is used in a similar way to its big brother the `jabber:iq:oob` namespace. Attaching URLs to messages, typically done by mechanisms that delivery news and alert style headines, is done like this:

```
SEND: <message type='headline' to='qmacro@jabber.org/laptop' id='h12'>
        <subject>Jabber Foundation Public Conference</subject>
        <x xmlns='jabber:x:oob'>
          <url>
            http://www.jabbercentral.com/news/view.php?news_id=989358658
          </url>
          <desc>
            Tomorrow, May 9th, a meeting regarding the Jabber
            Foundation will be held.
          </desc>
        </x>
      </message>
```

Multiple attachments can be made to a message.

The RSS Punter, described in the section called *RSS punter* in Chapter 8 and the Headline Viewer, described in the section called *Headline viewer* in Chapter 8 both use the `jabber:x:oob` namespace.

jabber:x:roster

The `jabber:x:roster` namespace is related to its big brother `jabber:iq:roster`; it is used to carry roster information as message attachments. This makes it straightforward for users to exchange contact information between themselves:

```
SEND: <message id='M91' to='shiels@jabber.org'>
        <body>Hi Robert - this is that fool I was telling you about...</body>
        <x xmlns='jabber:x:roster'>
          <item jid='qmacro@jabber.org' name='DJ Adams'>
            <group>Fools</group>
          </item>
        </x>
      </message>
```

Note that it is inappropriate to send the subscription related attributes (`subscription` and `ask`, described in the section called *`jabber:iq:roster`*). Instead, it is up to the recipient to negotiate their own presence subscription arrangements with the contact, or contacts (more than one item can be sent in such an attachment) listed.

`jabber:x:signed`

The `jabber:x:signed` namespace is related to the `jabber:x:encrypted` namespace and is used to stamp `<presence/>` and `<message/>` packets with a PKI-based signature, thus providing reliable identification of the packet originator.

In generating the signature block, some relevant data must be used to pass into the signing algorithm so that an electronic signature is produced. [2] In the case of `<presence/>` packets, the contents of the `<status/>` tag are used, and in the case of `<message/>` packets, the contents of the `<body/>` tag are used.

The presence of a `jabber:x:signed` signature in a `<presence/>` packet is intended to signify that the client sending the packet supports such PKI infrastucture and, for example, is able to decrypt messages encrypted in the `jabber:x:encrypted` namespace.

Here's a `<presence/>` packet containing a signature; the data *All present and correct* is what is fed into the algorithm.

```
SEND: <presence from='piers@jabber.org' to='qmacro@pipetree.com'>
  <status>All present and correct</status>
  <x xmlns='jabber:x:signed'>
    aslkjlksjdf jsfkjk23jskdfskjdfksjdf
  </x>
</presence>
```

Notes

[1] The filter service is described in the section called *Filter Service* in Chapter 4.

[2] Typically, when signing an email message electronically, the body of the email is passed into the signing algorithm to generate the signature.

Prev | Home | Next
The IQ Namespaces | Up | The X::IQ relationship

The X::IQ relationship

As has been noted, some of the X namespaces have cousins in the IQ space—*autoupdate*, *conference*, *roster*, and *oob*. If you're still confused about which to use where, there's a rule of thumb about context: The IQ namespaces generally are used to qualify a *conversation* that revolves around whatever the namespace represents, while the X namespaces apply more to one-off, ad-hoc, information-laden messages.

For example, the `jabber:iq:conference` namespace qualifies the much of the content of a conversation between a user and the conferencing service regarding entry to a specific room. The `jabber:x:conference` namespace is used to provide context and meaning to a pointer to a room.

Likewise, the `jabber:x:oob` namespace qualifies a pointer to some piece of information that is out of band, where as the `jabber:iq:oob` namespace provides context to a negotiation that leads to the usage of that external bandwidth.

Chapter 6. User Registration and Authorization

Table of Contents

With all the Jabber building blocks at our fingertips at this stage, we can now take a look at two fundamental processes that revolve around Jabber users:

- the registration, or creation, of Jabber user accounts
- the authentication of a Jabber user account, including the different authentication methods

These processes involve some of the things we learned about in the previous chapter, so we'll take a look at them again along the way:

- the XML stream header
- the `jabber:iq:register` namespace
- the `jabber:iq:auth` namespace

At the end of this chapter we will create a small utility program with which we can register users on a server. We can then use this utility program to create users as required for further recipes in this book.

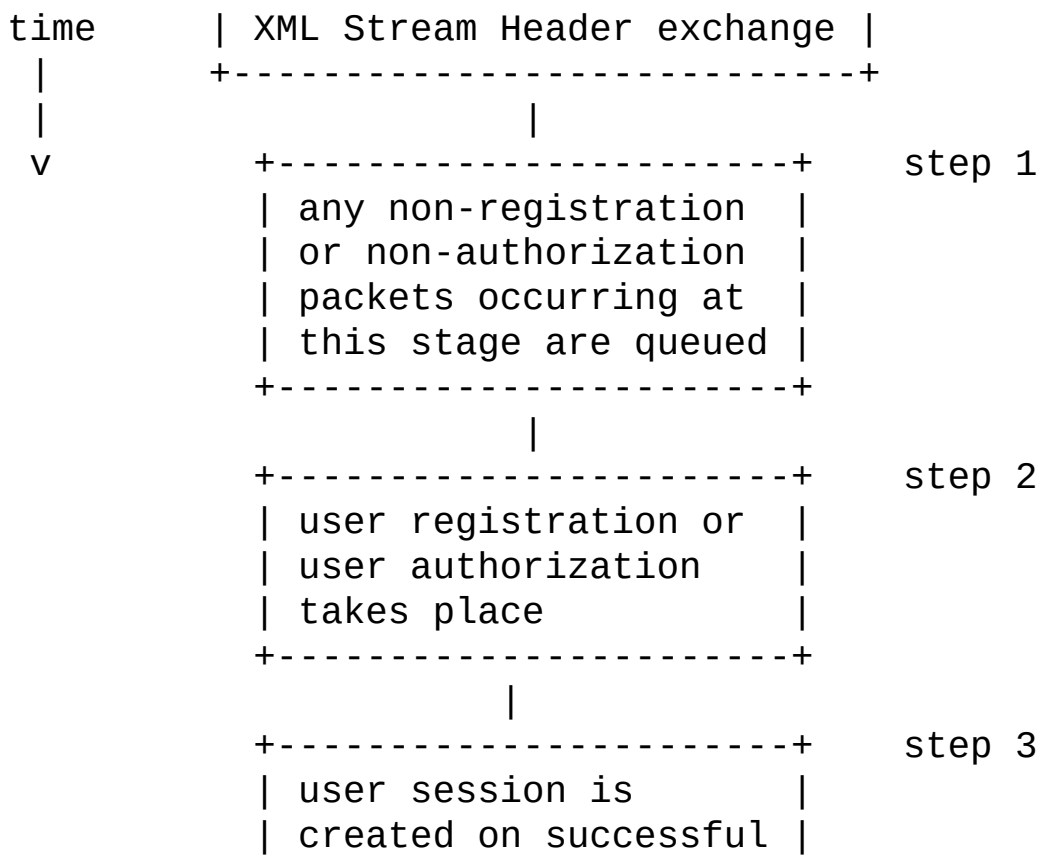
XML Stream Flow

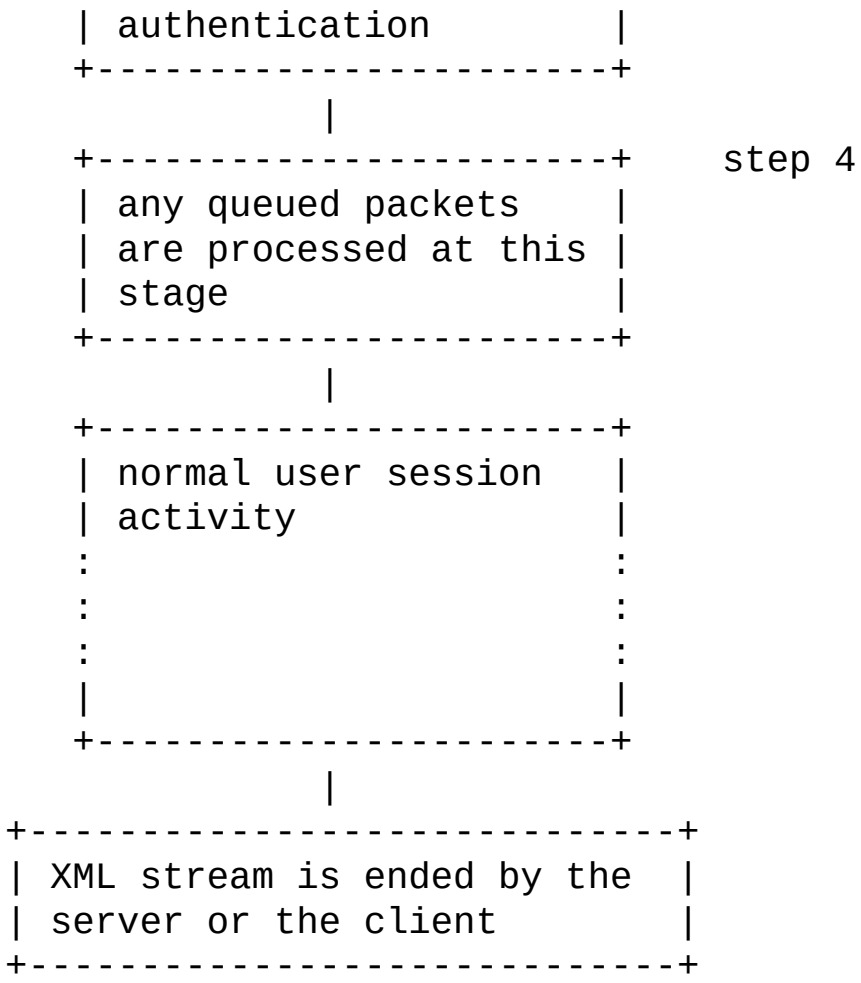
User registration and user authorization are two processes that take place at the start of an XML stream, immediately following the XML stream header exchange. The Jabber server is designed to check for and process any user registration request or authentication request *before* anything else.

This makes sense, as outside of a session (a session is created for a user when the authentication completes successfully) any other element - a `<message/>` packet, a `<presence/>` packet or an `<iq/>` packet that's not in the `jabber:iq:register` or `jabber:iq:auth` namespaces - is invalid. So if any of these other elements are sent *before* authentication has taken place, that is, before a session has been created for that connection, they are queued, and processed *after* authentication.

Figure 6-1 shows the general flow within an XML stream, with regard to where the registration and / or authentication steps happen.

Figure 6-1. XML Stream flow showing registration and authentication




authentication
any queued packets are processed at this stage
step 4
normal user session activity
XML stream is ended by the server or the client

User Registration

User registration with a Jabber *server* specifically means the creation (or modification) of a user account for the JSM component - the component that provides the basic IM services and has a notion of users and user sessions. It takes place at the start of a connection to a Jabber server, as does user authentication, as shown in Figure 6-1.

Let's take a look at the XML fragments involved in a typical user registration process. Example 6-1 shows the XML stream header exchange and the IQ packets in the `jabber:iq:register` namespace.

Example 6-1. A typical user registration process

First, the XML stream header exchange

```
SEND: <?xml version='1.0'?>
      <stream:stream to='yak' xmlns='jabber:client'
                     xmlns:stream='http://etherx.jabber.org/streams'>

RECV: <?xml version='1.0'?>
      <stream:stream xmlns:stream='http://etherx.jabber.org/streams'
                     id='3B2DB1A7' xmlns='jabber:client' from='yak'>
```

Then the client sends a request to discover what information must be passed to the Jabber server to register a new user:

```
SEND: <iq type='get'>
        <query xmlns='jabber:iq:register'/>
      </iq>

RECV: <iq type='result'>
        <query xmlns='jabber:iq:register'>
          <instructions>
            Choose a username and password to register with this server.
          </instructions>
          <name/>
          <email/>
          <username/>
          <password/>
```

```
        </query>
      </iq>
```

The client does as asked, and sends the required information, which results in a successful new user registration:

```
SEND: <iq type='set'>
        <query xmlns='jabber:iq:register'>
          <username>leslie</username>
          <password>secret</password>
          <email>lel@plevna.com</email>
          <name>Leslie Hawke</name>
        </query>
      </iq>

RECV: <iq type='result'/>
```

Configuration and Module Load Directives

Lets start by reviewing the relevant configuration in `jabber.xml`. User registration is a JSM feature, and we find two places of interest in that component instance definition. The first is in that instance's configuration, in the section qualified by the `jabber:config:jsm` namespace:

```
<register notify="yes">
  <instructions>
    Choose a username and password to register with this server.
  </instructions>
  <name/>
  <email/>
</register>
```

The second is in that instance's connection method, showing the `mod_register` module being loaded. The module plays a major part in handling the registration process, but there are others too, as we will see.

```
<load main="jsm">
  <jsm>./jsm/jsm.so</jsm>
  <mod_echo>./jsm/jsm.so</mod_echo>
  <mod_roster>./jsm/jsm.so</mod_roster>
  <mod_time>./jsm/jsm.so</mod_time>
  <mod_vcard>./jsm/jsm.so</mod_vcard>
  <mod_last>./jsm/jsm.so</mod_last>
  <mod_version>./jsm/jsm.so</mod_version>
  <mod_announce>./jsm/jsm.so</mod_announce>
```

```
  <mod_agents>./jsm/jsm.so</mod_agents>
  <mod_browse>./jsm/jsm.so</mod_browse>
  <mod_admin>./jsm/jsm.so</mod_admin>
  <mod_filter>./jsm/jsm.so</mod_filter>
  <mod_offline>./jsm/jsm.so</mod_offline>
  <mod_presence>./jsm/jsm.so</mod_presence>
  <mod_auth_plain>./jsm/jsm.so</mod_auth_plain>
  <mod_auth_digest>./jsm/jsm.so</mod_auth_digest>
  <mod_auth_0k>./jsm/jsm.so</mod_auth_0k>
  <mod_log>./jsm/jsm.so</mod_log>
  <mod_register>./jsm/jsm.so</mod_register>
  <mod_xml>./jsm/jsm.so</mod_xml>
</load>
```

The `<register/>` configuration section looks familiar - the contents are very similar to the IQ result returned in response to the IQ get in the `jabber:iq:register` namespace in Example 6-1. Indeed, to respond to an IQ get, the `mod_register` module looks for this `<register/>` section and formulates the contents into a reply, making a simple modification as it goes - it appends tags for the the two fields

```
<username/>
```

and

```
<password/>
```

as these two are *always* required, regardless of configuration.

Removing the `<register/>` section from the configuration effectively blocks the registration process, and returns an appropriate error to the user:

```
SEND: <iq type='get'>
        <query xmlns='jabber:iq:register'/>
      </iq>

RECV: <iq type='error'>
        <query xmlns='jabber:iq:register'/>
        <error code='501'>Not Implemented</error>
      </iq>
```

Removing the reference to the `mod_register` module from the component instance's connection method:

```
<!--
```

```
  <mod_register>./jsm/jsm.so</mod_register>
-->
```

has the same effect.

Step by Step

Taking the stream contents in Example 6-1 step by step, what do we see?

The XML Stream Header Exchange

The XML declaration that is send immediately preceding the opening `<stream:stream/>` root tag is optional. The Jabber server does not enforce its presence, and so you can leave it out if you wish. [1] The Jabber server will always send one in response, however. In both cases - both streamed XML documents - the encoding is assumed to be UTF-8.

For the most part, the rest of the `<stream:stream/>` root tag is static. The namespace qualifying the stream content is `jabber:client` (which is the only namespace acceptable when making such an XML stream connection to the `c2s` (Client to Server) component listening on port 5222), and the namespace qualifying the stream itself is fixed at `http://etherx.jabber.org/streams`. [2]

The only think that is going to be dynamic is the `to` attribute, which is used to specify the Jabber server name. Note that this is the *internal name* of the Jabber server. In our example we've already resolved the physical hostname '`yak`' and connected to port 5222; the `to` attribute is to specify the virtual Jabber host, which in many cases - including an out-of-the-box `jabber.xml` configuration - is the same as the physical host.

If our Jabber server has just a single virtual host, we can use an `<alias/>` configuration tag in the `c2s` component instance configuration, as described in Chapter 5, to remove the requirement of specifying the `to` attribute:

```
<alias to='yak'/>
```

in the configuration section for the `c2s instance` will set the *default* alias to 'yak' and indeed override any value specified in the `to` attribute.

The Query

While all the examples in this book follow the convention that the opening tag name used in an IQ extension is '`query`', we know from the section called *IQ Subelements* in Chapter 5 that in this case, the tag name *must* be '`query`' and nothing else. This is due to the way that the XML is parsed by the Jabber

server at this early stage in the connection. The same is true for the user authorization case too.

'Required' Fields?

The list of fields returned in the response to the IQ get in the `jabber:iq:register namespace`:

```
<name/>
<email/>
<username/>
<password/>
```

is actually supposed to be a list of *mandatory* fields. However, with the current version of the Jabber server (1.4.1, which is our reference version for this book), this is not the case.

The two fields `<username/>` and `<password/>` *are* mandatory; not supplying, or supplying an invalid username or password will result in an error:

```
SEND: <iq type='set'>
        <query xmlns='jabber:iq:register'>
          <username>leslie</username>
          <email>lel@plevna.com</email>
          <name>Leslie Hawke</name>
        </query>
      </iq>

RECV: <iq type='error'>
        <query xmlns='jabber:iq:register'>
          <username>leslie</username>
          <email>lel@plevna.com</email>
          <name>Leslie Hawke</name>
        </query>
        <error code='406'>Not Acceptable</error>
      </iq>
```

However, currently, not supplying any of the other fields - the fields that are specified in the `<register/>` section of the JSM instance's configuration - will not result in an error. This may be fixed in a later release. In any case, it's no great loss; the details are simply stored in the user's spool file on the server, and right now there's only one situation where this information is subsequently used - in answering a browse request made to a user's JID, the server looks up the `<name/>` tag from the registration data stored, and uses the value there in the browse response:

```
SEND: <iq type='get' to='dj@yak'>
        <query xmlns='jabber:iq:browse'/>
```

```
      </iq>

RECV: <iq type='result' from='dj@yak' to='sabine@yak/Work'>
        <user name='DJ Adams' xmlns='jabber:iq:browse' jid='dj@yak'/>
      </iq>
```

Set Without Get

In many of the examples of IQ throughout the book, we've seen a standard pattern: IQ get -> IQ result -> IQ set -> IQ result. This pattern isn't any different here, but it's not essential. Bearing in mind that registration field requirements aren't going to change that often, and even if they do, the only ones that are enforced are `<username/>` and `<password/>`, you can get away with forgoing the IQ get and cutting straight to the chase with an IQ set. This isn't a recommendation to do that, merely an observation, as it's always good practise to "ask first".

Still No Connection

After registering a new user, note that there's still no session. Only after successful authorization (see later in this Chapter) is a session created. Although used in authentication (and so implicitly in session creation), the value of the `id` attribute in the XML stream header returned by the server (which has the value '3B2DB1A7' in Example 6-1) is a *connection* id, not a *session* id.

So, what can we do at this stage? Well, one of two things. Register another user (yes!) or proceed to the authentication stage. Basically, reaching the end of the registration process, we're back where we started - a 'raw' connection where only one of two sequences are valid - the `jabber:iq:register` or `jabber:iq:auth` sequences.

Passwords

You may be wondering about the plaintext nature of the password sent in the registration process. Although the Jabber server offers different types of password-based authentication, there's a 'bootstrap' process required to get the password to the server in the first place. There's no way round the fact that the server must at one time receive the password in all its plaintext glory. After receiving it, there are authentication processes that don't use the plaintext password again.

So if you're concerned about the security of this registration phase, consider doing it over a secure (SSL) connection to the server.

We will look at the detail of the different authentication mechanisms later in this Chapter; however it is worth noting here, in the context of the registration process, that the JSM modules that implement the mechanism are responsible for storing the password when it's received. The `mod_register` module actually 'registers' the user, but it is the `mod_auth_plain` and `mod_auth_0k` modules that actually

store the password when received. [3]

There's another occasion where passwords are stored, and that is when a user wishes to *change* their password. This procedure is also covered by the `jabber:iq:register` namespace, albeit in a different context - the context of a *session*. While a `jabber:iq:register`-based IQ conversation outside the context of a session is for registering a user, a similar conversation *within* the context of a session, that is, after a user has authenticated, is used to change the user's password. Among other reasons, this is for security - a session context implies the user has identified and authenticated himself, and so has the authority to change the password.

Example 6-2 shows a typical IQ set to change a password.

Example 6-2. Changing a password with `jabber:iq:register`

```
SEND: <iq type='set' id='pass_4' to='yak'>
        <query xmlns='jabber:iq:register'>
          <password>newsecret</password>
        </query>
      </iq>

RECV: <iq type='result' id='pass_4' from='yak' to='dj@yak/Work'/>
```

The `to` attribute is required here, to make sure the query is handled by the server itself. We can also see evidence that the context of this exchange is within a session in the value of the `to` attribute on the IQ result packet - the JID '`dj@yak/Work`' includes a resource suffix, which implies a session (a resource must be specified in the authentication process - see later in this Chapter). And the specification of a `<username/>` is not necessary, as the server will stamp the incoming IQ set anyway with the JID associated with the user's session.

If you had made an IQ get, as recommended above, before doing the IQ set to change the password, the result would have looked like this:

```
RECV: <iq type='result' to='dj@yak/Work' id='pass_2' from='yak'>
        <query xmlns='jabber:iq:register'>
          <password/>
          <instructions>
            Choose a username and password to register with this server.
          </instructions>
          <name/>
          <email/>
          <key>9a6957b7f69535274afa5c134fb4d916c5d5c20b</key>
          <registered/>
        </query>
```

```
      </iq>
```

We see that, as in the registration IQ get outside the session context, the contents of the `<register/>` section of the JSM instance's configuration have been inserted (the `<instructions/>`, `<name/>` and `<email/>` tags). Additionally, we have a `<key/>` tag, as a simple security token as described in the section called *IQ Subelements* in Chapter 5, and a `<registered/>` tag. The `<key/>` is not actually checked in the current implementation, and is therefore not necessary to supply in the return IQ set packet. And the `<register/>` tag is merely a flag telling us that the user is already registered.

Use of the `jabber:iq:register` namespace in conversation with the JSM in a session context is not limited to changing passwords; you can modify the rest of the registration details supplied when the user was created - in this case, the `<name/>` and `<email/>` information. In fact, with the current implementation, because of the lack of checks, you can specify your own fields in the `jabber:iq:register` IQ set, in both contexts. But don't do it; it's a habit that will probably be impossible to keep up in later releases of the server. [4]

It almost goes without saying that because IQs in the `jabber:iq:register` namespace are handled differently in a session context, you can't register a new user once your session has started; you must end it. To end it, the XML stream must be closed with a `</stream:stream>`, and a new connection and stream must then be created.

Reversing a User Registration

The opposite of registering a user is *un*registering a user. This is not the same as removing that user altogether. When the `<remove/>` tag, described in the section called *`jabber:iq:register`* in Chapter 5a, is used in a `jabber:iq:register` qualified IQ set during a user session, the user is *unregistered*. That is, all the information held in the user's spool file is removed. But the spool file itself is not removed until the Jabber server is shut down. This means that even if you `<remove/>` a user, the username will still exist until the server is cycled, causing an error if the same username is used in a new registration attempt:

```
RECV: <iq type='error'>
        <query xmlns='jabber:iq:register'>
          <username>dj</username>
          <password>secret</password>
        </query>
        <error code='409'>Username Not Available</error>
      </iq>
```

This may well be fixed in a later release of the Jabber server.

A Note On Error Messages

There are various errors that can occur during user registration. They are on the whole fairly plain and easy to understand. But because of the way the server has been written, you might be surprised at *what* error message you receive in certain circumstances.

Because the required fields `<username/>` and `<password/>` are checked *before* looking to see whether or not there is a `<register/>` section in the JSM instance configuration, you will always receive a *406 'Not Acceptable'* instead of a *501 'Not Implemented'* if you don't supply those fields.

Likewise, if you specify a username that already exists, you will receive a *409 'Username Not Available'* instead of a *501 'Not Implemented'*.

Of course, if you do an IQ get with the `jabber:iq:register` namespace beforehand, you *will* receive the 'correct' error - good practise pays!

Notes

[1] This may not seem like a big deal, but when you're testing against a Jabber server using **telnet**, it's 20 less characters that you have to type every time you create a new connection ;-)

[2] The name 'etherx' comes from an old library that implemented XML streams.

[3] The `mod_auth_digest` module doesn't play a password-storing role, as the mechanism it provides uses the plaintext password that is stored by `mod_auth_plain`

[4] And no, you can't spoof someone else by specifying the `<username/>` tag in a session context `jabber:iq:register` IQ set; it is ignored, the correct JID being taken from the `from` attribute stamp made as the packet hits the server.

User Authentication

There are similarities between user registration and user authentication:

- authentication must take place outside of a session context (it doesn't really makes sense inside a session context, anyway). It is perfectly possible to perform a user registration step followed by a user authentication step (for any user) in the same XML stream.
- the `<query/>` tag must have the name 'query'.
- any packets sent before the authentication step (apart from user registration packets) are queued until after the authentication step has been completed.
- the IQ get in the `jabber:iq:auth` namespace is not mandatory, but recommended (even more strongly than the recommendation for the IQ get in the `jabber:iq:register` namespace.)

Example 6-3 shows a typical authentication process, including the XML stream header exchange.

Example 6-3. A typical user authentication process

Here the authentication process immediately follows the initial XML stream header exchange:

```
SEND: <?xml version='1.0'?>
      <stream:stream to='yak' xmlns='jabber:client'
                     xmlns:stream='http://etherx.jabber.org/streams'>

RECV: <?xml version='1.0'?>
      <stream:stream xmlns:stream='http://etherx.jabber.org/streams'
                     id='1ED34A55' xmlns='jabber:client' from='yak'>
```

We ask the server about the authentication methods available for our specific user:

```
SEND: <iq type='get'>
        <query xmlns='jabber:iq:auth'>
          <username>dj</username>
        </query>
      </iq>

RECV: <iq type='result'>
        <query xmlns='jabber:iq:auth'>
```

```
          <username>dj</username>
          <password/>
          <digest/>
          <sequence>496</sequence>
          <token>3B2DEEC0</token>
          <resource/>
        </query>
      </iq>
```

Because we're connecting here to the server with **telnet** and don't have any digest utilities handy, we decide to use the simplest authentication method, and send our password in plaintext. The server checks the credentials, and gives us the thumbs up.

```
SEND: <iq type='set'>
        <query xmlns='jabber:iq:auth'>
          <username>dj</username>
          <password>secret</password>
          <resource>laptop</resource>
        </query>
      </iq>

RECV: <iq type='result' id='pthsock_client_auth_ID'/>
```

At this stage, we have a session. [1]

Configuration and Module Load Directives

The `c2s` component contains a configuration directive related to the authorization process:

```
<service id="c2s">
  <load>
    <pthsock_client>./pthsock/pthsock_client.so</pthsock_client>
  </load>
  <pthcsock xmlns='jabber:config:pth-csock'>
    <alias to='yak'/>
    <authtime/>
    ...
  </pthcsock>
</service>
```

This `<authtime/>` tag is used to set the time limit, in seconds, within which authentication should be

completed, starting to measure at the time the connection was made. See the section called *Custom Configuration* in Chapter 4 for more details.

There is also an undocumented tag `<auth/>` which can be specified in the JSM instance configuration (for example after the `<register/>` section) with which you can specify an external component that is to handle authentication in place of the standard JSM modules (`mod_auth_*`).

The JSM module load directives specify the modules which are to handle authentication:

```
<load main="jsm">
  <jsm>./jsm/jsm.so</jsm>
  <mod_echo>./jsm/jsm.so</mod_echo>
  <mod_roster>./jsm/jsm.so</mod_roster>
  <mod_time>./jsm/jsm.so</mod_time>
  <mod_vcard>./jsm/jsm.so</mod_vcard>
  <mod_last>./jsm/jsm.so</mod_last>
  <mod_version>./jsm/jsm.so</mod_version>
  <mod_announce>./jsm/jsm.so</mod_announce>
  <mod_agents>./jsm/jsm.so</mod_agents>
  <mod_browse>./jsm/jsm.so</mod_browse>
  <mod_admin>./jsm/jsm.so</mod_admin>
  <mod_filter>./jsm/jsm.so</mod_filter>
  <mod_offline>./jsm/jsm.so</mod_offline>
  <mod_presence>./jsm/jsm.so</mod_presence>
  <mod_auth_plain>./jsm/jsm.so</mod_auth_plain>
  <mod_auth_digest>./jsm/jsm.so</mod_auth_digest>
  <mod_auth_0k>./jsm/jsm.so</mod_auth_0k>
  <mod_log>./jsm/jsm.so</mod_log>
  <mod_register>./jsm/jsm.so</mod_register>
  <mod_xml>./jsm/jsm.so</mod_xml>
</load>
```

Each of these modules - `mod_auth_plain`, `mod_auth_digest` and `mod_auth_0k` - can play a role in the authentication process. As mentioned in the section called *Component Connection Method* in Chapter 4, they provide different authentication methods, these methods being reflected in their names:

- `mod_auth_plain`: plaintext
- `mod_auth_digest`: digest
- `mod_auth_0k`: zero-knowledge

There's a certain amount of flexibility you have as administrator to determine what methods are made available on your Jabber server. If you want to offer all three, do nothing. If you only want to offer the

zero-knowledge method, comment out or otherwise remove the other two definitions

```
<!--
  <mod_auth_plain>./jsm/jsm.so</mod_auth_plain>
  <mod_auth_digest>./jsm/jsm.so</mod_auth_digest>
-->
```

from the list of module load directives. If you want to offer the *digest* method, you must include the module load directives for both `mod_auth_plain` and `mod_auth_digest`, as the latter is merely an extension - a 'parasite' - upon the other.

Let's look at each of these authentication methods in turn.

Plaintext Authentication Method

The plaintext authentication method works as you would expect, and is the default 'lowest common denominator' method supplied with the Jabber server. It is provided by the `mod_auth_plain` module.

Method

> The password is transmitted in the XML stream, inside the `<password/>` tag in the `jabber:iq:auth` IQ set packet, from the client to the server in plaintext, where it is compared to the password stored, also in plaintext, on the server.
>
> When a password is changed, using a `jabber:iq:register` IQ set as described earlier in this Chapter, `mod_auth_plain` stores the password, as received, in the user's spool file.

Advantages

> This method is by far the simplest to implement on the client side. It is also useful for debugging and testing purposes as it can be used in a connection 'by hand' via **telnet**, not requiring any extra computation such as the digest and zero-knowledge methods do.

Disadvantages

> It's insecure, on two levels. First, the password is transmitted in plaintext across the wire from client to server. The risk can be minimized by encrypting the whole connection using SSL. Second, the password is stored in plaintext on the server, which may be compromised.

Digest Authentication Method

The module that provides the digest authentication method, `mod_auth_digest`, works in conjunction with the plaintext module `mod_auth_plain`. It provides a way of avoiding having to send the

plaintext password across the wire.

Method

The digest method is similar to the plaintext method, in that the password sent by the client is compared to the password stored on the server. However in this case, the password is first encoded using a hashing algorithm. It is encoded by the client before being sent across the wire, and it is encoded by the server (having retrieved it in plaintext) before being making the comparison.

The algorithm is the NIST SHA-1 message digest algorithm. [2] This algorithm takes arbitrary input and produces a 'fingerprint' or message digest of it. [3] A random string, shared between the client and the server, is appended to the password before being passed to the hashing algorithm. This random string is the value of the `id` attribute in the server's XML stream header response that we saw in Example 6-3:

```
RECV: <?xml version='1.0'?>
<stream:stream xmlns:stream='http://etherx.jabber.org/streams'
  id='1ED34A55' xmlns='jabber:client' from='yak'>
```

which means, in this case, the string that will be hashed is

```
secret1ED34A55
```

which is

```
03ea09f012493415908d63dcb1f6dbdb9bfc09ba
```

The digested password is transmitted to the server inside the `<digest/>` tag.

`mod_auth_digest` is unlike the other two modules in that it doesn't take any responsibility for storing passwords; it leaves that to mod_auth_plain, as the plaintext password is needed to be re-hashed with a new random suffix each time.

Advantages

No plaintext password is transmitted across the wire from client to server, and only a small amount of computation is required - a single SHA1 hash process.

Disadvantages

The password is still stored in plaintext on the server.

Zero-Knowledge Authentication Method

The zero-knowledge authentication method is so called as the server requires no knowledge of the password in order to check the credentials. It makes use of the same hashing algorithm used in the digest authentication method.

Method

Just as `mod_auth_plain` is responsible for storing the password (in plaintext) when a user is created or when the password is changed, so `mod_auth_0k` is responsible for storing *its version* of the password, or in fact the information it needs (originally *based* on the password) to check the client's credentials in a zero-knowledge authentication process.

As we know from the user registration and password change processes, any new password is supplied to the server in plaintext. This is where a secure (SSL) connection is critical for complete security. While the `mod_auth_plain` module just stores that password as-is, the `mod_auth_0k` module stores a sequenced hash of the password instead.

What does this mean?

The idea is this: the server stores a value which is the password hashed with an arbitrary string token multiple (N) times, recursively. It doesn't store the password itself. It remembers how many times it has been hashed (N).

Whenever a client wants to authenticate, the server sends the client the string token, and the value of N. The client, having obtained the password from the user, performs the same iterative hashing sequence that the server performed when it was originally given the password, but performs the sequence N-1 times. It passes the result of this sequence to the server, which does one more hash to go from N-1 to N, and compares what it gets with what it has stored. If they match, authentication is successful, and the server stores the N-1 hash passed from the client and decrements N, ready for next time.

The N-1 hash value is passed by the client to the server in the `<hash/>` tag.

By way of illustration, Figure 6-2 shows a small Perl script that a **telnet** user can use to make the necessary hashing computations when wishing to authenticate with a Jabber server using the zero-knowledge method.

Figure 6-2. A script implementing the client-side zero-knowledge process

```
#!/usr/bin/perl -w
```

```
use strict;
use Digest::SHA1 qw(sha1_hex);
use Getopt::Std;

our($opt_p, $opt_t, $opt_s);
getopt('pts');

unless ($opt_p and $opt_t and $opt_s)
{
  print "Usage: $0 -p <password> -t <token> -s <sequence>\n";
  exit;
}

# Initial hash of password
my $hash = sha1_hex($opt_p);

# Sequence 0: hash of hashed-password and token
$hash = sha1_hex($hash.$opt_t);

# Repeat N-1 times
$hash = sha1_hex($hash) while $opt_s--;

print "$hash\n";
```

Pass the data to the script via the parameters, using the values obtained from the response to the initial IQ get. Use the value that the script produces for the value to send in the `<hash/>` tag.

Advantages

No password is stored on the server. No (plaintext) password is transmitted from client to server.

Disadvantages

This method is slightly more compute-intensive than the other methods. There is still a security weak spot in the procedure, when the password is set or re-set (required when the counter N reaches zero), as it must be passed in plaintext.

Choosing the Authentication Method

Now we know a little bit about the authentication methods, let's jump back to the initial IQ get query in Example 6-3:

```
SEND: <iq type='get'>
```

```
        <query xmlns='jabber:iq:auth'>
          <username>dj</username>
        </query>
      </iq>

RECV: <iq type='result'>
        <query xmlns='jabber:iq:auth'>
          <username>dj</username>
          <password/>
          <digest/>
          <sequence>496</sequence>
          <token>3B2DEEC0</token>
          <resource/>
        </query>
      </iq>
```

What we're actually seeing here is the result of the authentication modules announcing their readiness to authenticate the user `dj`. The query is passed to each of the modules. `mod_auth_plain` announces its readiness by inserting the `<password/>` flag, `mod_auth_digest` with the `<digest/>` flag, and `mod_auth_0k` inserts the `<sequence/>` and `<token/>` tags and values, which is what the client will need it if wishes to authenticate using the zero-knowledge method. The `<resource/>` tag, which is required in any authentication, is finally added before the result is returned.

This way, the IQ result can convey which authentication methods are available - if the `mod_auth_plain` and `mod_auth_digest` modules were to be commented out in the module load directive list, as we saw earlier, then the IQ result would look like this:

```
RECV: <iq type='result'>
        <query xmlns='jabber:iq:auth'>
          <username>dj</username>
          <sequence>496</sequence>
          <token>3B2DEEC0</token>
          <resource/>
        </query>
      </iq>
```

that is, without the `<password/>` or `<digest/>` tags.

At the beginning of this section on User Authorization, we noted that the IQ get in the `jabber:iq:auth` namespace was strongly recommended, before proceeding with the IQ set. This was primarily to be able to check what authentication methods were supported. However, as you can see here, it is in fact *essential* in the case of the zero-knowledge authentication method, because without the

<sequence/> and <token/> information, we cannot begin our hashing sequence.

Password Errors and Retries

If you don't supply all the required parameters, you are notified with a 406 "Not Acceptable" error:

```
SEND: <iq type='set'>
        <query xmlns='jabber:iq:auth'>
          <username>dj</username>
          <digest>03ea09f012493415908d63dcb1f6dbdb9bfc09ba</digest>
        </query>
      </iq>

RECV: <iq type='result' id='pthsock_client_auth_ID'>
        <query xmlns='jabber:iq:auth'>
          <username>dj</username>
          <digest>03ea09f012493415908d63dcb1f6dbdb9bfc09ba</digest>
        </query>
        <error code='406'>Not Acceptable</error>
      </iq>
```

In this case, no <resource/> value was supplied. The <resource/> value is required, as it is the key part of the JID that is used to distinguish between multiple connections as the same user on the same Jabber server.

If you get the password wrong, you receive a 401 "Not Authorized" error. There is currently no limit on the number of times an authorization attempt can be made, but the <karma/> limits for the c2s connections (see Chapter 5) may slow the attempts down.

Notes

[1] The 'pthsock_client_auth_ID' value for the id attribute is placed by the JSM as it processes the IQ set request, in cases where a value has not been specified.

[2] More information on SHA-1 - Secure Hash Algorithm 1, can be obtained at http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[3] There are different types of message digest that can be produced with this algorithm - binary, hexadecimal and base64. The hexadecimal format is used here and elsewhere in Jabber.

User Registration Script

Currently, the Jabber technology has no concept of 'anonymous' users - users that can connect to the server with no previous registration requirement. Until such time as this is possible, we will have to make do with creating specific users for specific scenarios.

To this end, it would be useful to be able to run a quick script to register a new user, rather than grab an existing client, start it up, and go through the process of registering a new user with the Jabber server, with whatever window navigation and mouse-clicking that might entail. All the script must do is interact with the Jabber server in the context of the `jabber:iq:register` namespace, specifically pre-session. It must be able to make a registration *enquiry* by sending an IQ get and returning the fields listed in the result, and to make a registration *attempt*, when supplied with values for the registration fields.

We should be able to invoke our script, **reguser**, in one of two ways. The first, specifying merely a hostname (and optional port, which will default to 5222 if not specified), implies a registration *enquiry* - where we wish to make an enquiry of the Jabber server as to (a) whether registration is possible and (b) if so, what fields are 'required'. The second way, specifying not only the host and optional port, but also a list of fieldname and value pairs to send in a user registration *attempt*. Figure 6-3 shows both these ways in action.

Figure 6-3. Uses of the reguser script

```
$ ./reguser yak:5222
[Enquiry] Fields: username password email name

$ ./reguser yak username=joseph password=spinach 'name=Joseph Adams' email=joseph@yak
[Attempt] (joseph) Successful registration

$ ./reguser yak username=dj password=secret 'name=DJ Adams' email=dj@yak
[Attempt] (dj) Error: 409 (Username Not Available)
```

As it's our first substantial script, let's take it step by step. It's written in Perl.

```
#!/usr/bin/perl

use strict;
use Net::Jabber 1.0022 qw(Client);

use constant NS_REGISTER => 'jabber:iq:register';

unless (@ARGV) {
  print usage();
  exit;
}
```

We start out with some basic start-of-script housekeeping - delaring our usage of the Net::Jabber module, setting a constant for the `jabber:iq:register` namespace, and handling the case of being 'wrongly' invoked by giving some help text from the

`usage()` subroutine. The specification of "`Client`" in the

```
use Net::Jabber 1.0022 qw(Client);
```

means that the connection is going to be client-based; in other words, the namespace that will be used to qualify the XML stream header that Net::Jabber will produce is `jabber:client`.

The Net::Jabber module has changed as it has matured over the recent versions (1.0020, 1.0021 and 1.0022) and these changes do sometimes affect how the scripts that use Net::Jabber should be written. So we explicitly specify in the `use` statement which version of Net::Jabber we require, to avoid confusion.

```
my ($host, $port) = split(":", shift @ARGV);
$port ||= 5222;

my $c = Net::Jabber::Client->new();

defined($c->Connect(
            hostname => $host,
            port     => $port,
)) or die "Cannot reach Jabber server at $host:$port\n";

my ($iq, $query, $result);
```

We parse the hostname and port, defaulting the latter to 5222 if it wasn't specified. Then we create a new instance of the Net::Jabber::Client object. The Net::Jabber family of modules presents its collective functions in an object-oriented way. The scalar `$c` represents the 'client' mechanism with which we connect to the Jabber server.

With the `Connect()` method, we make a connection to the Jabber server; the namespace of the XML stream for this connection, sent to the Jabber server in the stream header, is `jabber:client`.

```
# Registration attempt or enquiry?

if (scalar @ARGV) {

  # Attempt:
  # Send <iq type='set'>
  #        <query xmlns='jabber:iq:register'>
  #          <username>...</username>
  #          <password>...</password>
  #          ...
  #        </query>
  #      </iq>

  print "[Attempt] ";

  $iq = Net::Jabber::IQ->new();
  $iq->SetType('set');
  $query = $iq->NewQuery(NS_REGISTER);
```

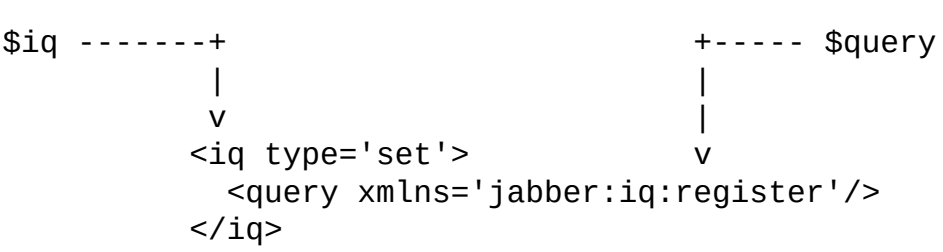
We work out what we have to do by looking to see if any extra parameters beyond the hostname and port were specified. If there were, we need to build an IQ set in the `jabber:iq:register` namespace to make a registration attempt.

The Net::Jabber::IQ module represents the IQ model and provides methods to manage IQ packets. With the `new()` constructor we create a new, empty IQ packet in `$iq`, and set its `type` attribute to 'set'.

As we know, the `<query/>` part of an IQ packet is contained *within* the `<iq/>` tag. The `NewQuery()` method, called on an IQ packet, creates a `<query/>` tag as a child of that IQ packet, and delivers us a handle on that `<query/>` tag - which we store in `$query` - so that we can manipulate it independently of the IQ packet that wraps around it. The `jabber:iq:register` namespace value is passed as a parameter to the `NewQuery()` call to set the correct `xmlns` namespace attribute.

Figure 6-4 shows what the packet looks like at this stage, and how the scalar object references `$iq` and `$query` relate to it.

Figure 6-4. An IQ packet under construction by Net::Jabber::IQ



In our `foreach` loop we run through the list of parameters, in the form *fieldname=value*, and call a `Set` method on the `$query` object (the `<query/>` packet) to add child tags:

```
foreach my $arg (@ARGV) {
  my ($field, $value) = split('=', $arg);
  print "($value) " if $field eq 'username';
  eval '$query->Set'.ucfirst($field).'($value)';
}

$result = $c->SendAndReceiveWithID($iq);
```

Net::Jabber::Query provides a number of `SetXXXX` methods which are available according to namespace. These 'set' methods available for the `jabber:iq:register` namespace are plentiful and include `SetName`, `SetEmail`, `SetPhone` and so on. Each method will create a child tag named after the method (e.g. `SetName` will create a `<name/>` tag and `SetPhone` will create a `<phone/>` tag) and insert the value passed to the method into the tag.

For example,

```
$query->SetName('DJ Adams');
```

will insert (or amend) a tag in the `<query/>` thus:

```
<iq type='set'>
  <query xmlns='jabber:iq:register'>
    <name>DJ Adams</name>
  </query>
</iq>
```

We use `eval` to allow us to make our `SetXXXX` method calls dynamically, according to each fieldname specified. The `ucfirst()` function is used to change the first character of the fieldname to upper case, to suit the `SetXXXX` method naming conventions.

Once we've added all the fields, we send the complete packet (`$iq`) to the server using the `SendAndReceiveWithID()` method on the connection (`$c`) object. This method is extremely powerful and does many things for us. It keeps the process of writing small scripts like this very simple. What is does is: adds a unique `id` attribute to the `<iq/>` packet, transmits the packet over the XML stream, *and waits for a response*.

"Hey, what about the event-model that we read about?" you might ask. Of course, Net::Jabber supports an event-model programming style, but for now we can get away with keeping our code 'procedural' - and short - using this high-level method which does everything we want. After all, in any one execution of the script, we only wish to send one packet to the Jabber server and receive one back. Nothing more complicated than that.

Later scripts will demonstrate the event-model.

The response is stored in `$result`, and is itself an IQ packet, as we expect. So `$result` is a handle on a Net::Jabber::IQ object that we can now manipulate.

```
  # Success
  if ($result->GetType() eq 'result') {
    print "Successful registration\n";
  }

  # Failure
  else {
    print "Error: ",
          $result->GetErrorCode(),
          " (",
          $result->GetError(),
          ")\n";
  }

}
```

We check the type of the IQ returned from the server. If it's a 'result':

```
RECV: <iq type='result' id='1'>
        <query xmlns='jabber:iq:register'/>
      </iq>
```

then great - the registration was successful. Otherwise, we can grab the error code and description from the `<error/>` element:

```
RECV: <iq type='error'>
        <query xmlns='jabber:iq:register'>
          <username>dj</username>
          <password>secret</password>
        </query>
        <error code='409'>Username Not Available</error>
      </iq>
```

using the `GetError()` and `GetErrorCode()` methods on the IQ object.

We go through a similar process if there are no further parameters following the host[:port] specification:

```
else {

  # Enquiry:
  # Send <iq type='get'><query xmlns='jabber:iq:register'/></iq>

  print "[Enquiry] ";

  $iq = Net::Jabber::IQ->new();
  $iq->SetType('get');
  $query = $iq->NewQuery(NS_REGISTER);

  $result = $c->SendAndReceiveWithID($iq);
```

The only difference here is that we set the IQ type to 'get', not 'set', and we don't insert any tags into the `$query` object, before sending the packet off and waiting for a response.

```
  # Success
  if ($result->GetType() eq 'result') {
    $query = $result->GetQuery();

    my %contents = $query->GetRegister();
    delete $contents{'instructions'};
    print "Fields: ", join(', ', keys %contents), "\n";
  }
```

If we receive a 'result' type, like this:

```
RECV: <iq type='result'>
        <query xmlns='jabber:iq:register'>
          <instructions>
            Choose a username and password to register with this server.
          </instructions>
          <name/>
          <email/>
          <username/>
          <password/>
        </query>
      </iq>
```

then we need to extract the fields listed in the `<query/>` tag and return them to the user. While the `NewQuery()` method creates a new `<query/>` tag inside an IQ object, the `GetQuery()` method retrieves an existing one, in the form of a Net::Jabber::Query object whose handle we store in `$query`. We can call the `GetRegister()` method on this query object, which returns a hash of the contents:

```
(
  'instructions' => 'Choose a username and password ...',
  'name'         => undef,
  'email'        => undef,
  'username'     => undef,
  'password'     => undef
```

```
)
```

And, after removing the 'instructions', we can display them as the result.

In the case where an error is returned in response to the IQ get (perhaps no registrations are allowed), we display the error in the same was as before:

```
  # Failure
  else {
    $query = $result->GetQuery();
    print "Error: ",
          $result->GetErrorCode(),
          " (",
          $result->GetError(),
          ")\n";
  }

}
```

When we've finished, we close the connection and exit. Here we also have the `usage()` subroutine defined.

```
$c->Disconnect;

exit;


sub usage {

<<EOF
Usage:
Enquiry: reguser host[:port]
Attempt: reguser host[:port] field1=value1 [fieldN = valueN] ...
EOF

}
```

Using the Script

The script is very basic, but it gets the job done. It is suitable for calling from another script, for mass user generation, although you may wish to modify it so that a connection is not created and destroyed for every username that needs to be registered.

It also illustrates how simple a Jabber client can be. In this case, the Net::Jabber libraries mask the bulk of the effort (socket connection, XML stream negotiation, XML fragment traffic management, and so on). We'll be making use of this script to create users for our recipes later on in the book.

Chapter 7. Messages, Presence, and Presence Subscription

Table of Contents

Now that we have a decent grounding in the Jabber protocol and technology, let's put it to work for us. This chapter fits Jabber into solutions for two or three common problems, and shows how the technology and features lend themselves very well to application-to-person (A2P) scenarios.

By way of introduction, we'll have a look at constructing and sending simple Jabber messages, to effect an "in your face" notification mechanism for a version control system. We'll also introduce a usage of the `<presence/>` element as an availability indicator for connecting systems. Finally we'll combine the two features (`<message/>` and the concept of *availability*) to make the notification mechanism "sensitive" to the presence of the person being notified.

CVS notification

CVS—the Concurrent Versions System [1] —allows you to comfortably create and manage versions of the sources of your project. The most common use for CVS is to create and manage versions of program source code, but it can be readily used for any text files. For example, this book was written using the DocBook markup language [2] and CVS was used to take versions of the manuscript at certain points in the writing's progress. The versions so taken could be compared, and old versions could be retrieved.

That's the *Versions System* part of the name. The *Concurrent* part means that this facility is given an extra dimension in the form of group collaboration. With CVS, more than one person can share work on a project, and the various chunks of work carried out by each participant are coordinated—automatically, to a large extent—by CVS. Multiple changes by different people to the same file can be merged by CVS; any unresolvable conflicts (which may for example arise when more than one person changes exactly the same line of source code) are flagged and must be resolved by the participants involved.

The general idea is that you can create a project containing files and directories and have it stored centrally in a CVS *repository*. Depending on what sort of access is granted to this repository, other project participants can pull down a copy of the project—those files and directories—and work on it independently. In this way, each participant's work is isolated (in time and space) from the others. When the work is done, the work can be sent back to the repository and the changes will be merged into the central copy. After that, those merged changes are available to the rest of the participants.

CVS watches and notification

While CVS automatically handles most of the tedious merging process that comes about when more than one person works on a project, it also offers a facility which allows you to set a "watch" on one or more files in the project, and be alerted when someone else starts to work on those watched files. This is useful if you wish to preempt any automatic merging process by contacting the other participant and coordinating your editing efforts with them.

There are two CVS commands involved in setting up watches and notifications. There are also a couple of CVS administrative files that determine how the notifications are carried out. Let's look at these commands and files in turn.

CVS commands

The CVS commands **cvs watch** and **cvs notify** are used, usually in combination, by project participants to set up the notification mechanism.

cvs watch on|off

Assuming we have a CVS-controlled project called '*proj1*', and we're currently inside our local checked-out copy of the project's files, we first use **cvs watch** to tell CVS to watch a file ("turn a watch *on*") that we're interested in, which is `file4` in this example:

```
yak:~/projects/proj1$ cvs watch on file4
```

This causes CVS to mark `file4` as "watched", which means that any time a project participant checks out the file from the central repository, the checked-out working copy is created with read-only attributes. This means that the participant is (initially) prevented from saving any changes to that working copy. It is, in effect, a reminder to that participant to use the CVS command **cvs edit**, specifying `file4`, before commencing the edit session. Using **cvs edit** will cause CVS to:

1. remove the read-only attribute for the file
2. send out a notification that the participant has commenced editing it.

cvs watch add|remove

While running **cvs watch on** against a file will set a marker causing the file to be replicated with the read-only attribute when checked out (which has the effect of "suggesting" to the participant editing the file that he use the **cvs edit** command to signal that he's to start editing), the actual determination of the notification recipients is set up using the **cvs watch add** command.

Running the command:

```
yak:~/projects/proj1$ cvs watch add file4
```

will arrange for the CVS notification to be sent to *us* when someone else signals their intention (via **cvs edit**) to edit `file4`.

CVS administrative files

Kept in the central CVS repository are a number of administrative files that are used to control how CVS works. Two of these files, `notify` and `users`, are used to manage the watch-based notification process.

`notify`

The standard `notify` file contains a line like this:

```
ALL mail %s -s "CVS notification"
```

The '`ALL`' causes the formula described here to be used for any notification requirements (an alternative to `ALL` is a regular expression to match the directory name in which the edit causing the notification is being carried out).

The rest of the line is the formula to use to send the notification. It is a simple invocation of the **mail** command, specifying a subject line (`-s "CVS notification"`). The `%s` is a placeholder that CVS replaces with the address of the notification's intended recipient. The actual notification text, generated by CVS, is piped into the **mail** command via STDIN.

`users`

The `users` file contains a list of notification recipient addresses:

```
dj:dj.adams@pobox.com
piers:pxharding@ompa.net
robert:robert@shiels.com
...
```

This is a mapping from the user IDs (`dj`, `piers`, and `robert`) of the CVS participants, local to the host where the CVS repository is stored, to the actual addresses (`dj.adams@pobox.com`, `pxharding@ompa.net`, and `robert@shiels.com`) that are used to replace the `%s` in the formula described in the `notify` file.

The Notification

If the contents of the `notify` and `users` files have been set up correctly, a typical notification, set up by *dj* using the **cvs watch on file4** and **cvs watch add file4** commands, and triggered by *piers* using the **cvs edit file4** command, will be received in *dj*'s inbox looking like the one shown in Figure 7-1.

Figure 7-1. A typical email CVS notification

```
Date: Fri, 8 Jun 2001 13:10:55 +0100
From: piers@ompa.net
To: dj.adams@pobox.com
Subject: CVS notification

testproject file4
---
Triggered edit watch on /usr/local/cvsroot/testproject
```

By piers

CVS notifications via Jabber

While email-based notifications are useful, we can add value to this process by using a more immediate (and penetrating) form of communication: Jabber. While mail clients can be configured to check for mail automatically on a regular basis, using an IM-style client has a number of immediately obvious advantages:

- it's likely to take up less screen real-estate
- no amount of tweaking of the mail client's auto-check frequency, if available (which will log in, check for, and *pull* emails from the mail server) will match the immediacy of IM-style message *push*
- in extreme cases, the higher the auto-check frequency, the higher effect on overall system performance
- depending on the configuration, an incoming Jabber message can be made to pop up, with greater effect
- a Jabber user is more likely to have a Jabber client running permanently than an email client
- it's more fun!

The design of CVS's notification mechanism is simple and abstract enough for us to put an alternative notification system in place. If we substitute the formula in the `notify` configuration file with something that will call a Jabber script, we might end up with something like Example 7-1.

Example 7-1. A Jabber notification formula in the `notify` file

```
ALL python cvsmsg %s
```

Like the previous formula, it will be invoked by CVS to send the notification, and the `%s` will be substituted by the recipient's address determined from the `users` file. In this case, the Python script *cvsmsg* is called.

But now that we're sending a notification via Jabber, we need a Jabber address - a JID - instead of an email address. No problem; just edit the `users` file to reflect the new addresses. Example 7-2 shows what the `users` file might contain if we were to use JIDs instead of email addresses.

Example 7-2. Matching users to JIDs in the `notify` file

```
dj:dj@gnu.pipetree.com
piers:piers@jabber.org
robert:shiels@jabber.org
```

As Jabber user JIDs in their most basic form (i.e., without a *resource* suffix) resemble email IDs, there doesn't appear to be that much difference. In any case, CVS doesn't really care, and simply passes the portion following the colon separator to the formula in the `notify` file.

The `cvsmsg` notification script

Let's now have a look at the `cvsmsg` script. It has to send a notification message, which it receives on STDIN, to a JID, which it receives as an argument passed to the script.

```
import Jabber, XMLStream
import sys
```

We're going to use the *JabberPy* Python libraries for Jabber [3] so we import them here. As is the way with many implementations for Jabber, we find there is a module to handle the XML stream-based connection with the Jabber server (the `XMLStream` module) and a module to handle the various Jabber connection mechanisms such as authentication, the sending of Jabber elements (`<message/>`, `<presence/>`, and `<iq/>`), and the dispatching of Jabber elements received (the `Jabber` module). We also import the `sys` module for reading from STDIN.

As the usage of the script will be fairly static, we can get away here with hardcoding a few parameters:

```
Server   = 'gnu.pipetree.com'
Username = 'cvsmsg'
Password = 'secret'
Resource = 'cvsmsg'
```

Specified here are the connection and authentication details for the `cvsmsg` script itself. If it's to send a message via Jabber, it must itself connect to Jabber. The `Server` variable specifies which Jabber server to connect to, and the `Username`, `Password`, and `Resource` variables contain the rest of the information for the script's own JID (`cvsmsg@gnu.pipetree.com/cvsmsg`) and password.

```
cvsuser  = sys.argv[1]
message  = ''

for line in sys.stdin.readlines(): message = message + line
```

The `sys.argv[1]` refers to the notification recipient's JID, which will be specified by the CVS notification mechanism, as it is substituted for the `%s` in the `notify` file's formula. This is saved in the `cvsuser` variable. We then build up the content of our message body we're going to send via Jabber by reading what's available on STDIN. Typically this will look like what we saw in the email message body

in Figure 7-1:

```
testproject file4
---
Triggered edit watch on /usr/local/cvsroot/testproject
By piers
```

```
con = Jabber.Connection(host=Server)
```

Although it is the `XMLStream` module that handles the connection to the Jabber server, we are shielded from the details of this by the `Jabber` module that wraps and uses `XMLStream`. Hence the call to instantiate a new `Jabber.Connection` object into `con`, to lay the way for our connection to the host specified in our Server variable: `gnu.pipetree.com`. No port is explicitly specified here, and the method assumes a default port of 5222, on which the *c2s* service listens.

The instantiation causes a number of parameters and variables to be initialized, and an `XMLStream.Client` object is instantiated; various parameters are passed through from the `Jabber.Connection` object (for example for logging and debugging purposes) and an XML parser object is instantiated. This will be used to parse fragments of XML that come in over the XML stream.

```
try:
    con.connect()
except XMLStream.error, e:
    print "Couldn't connect: %s" % e
    sys.exit(0)
```

A connection is attempted with the `connect()` method of the connection object in `con`. This is serviced by the `XMLStream.Client` object and a stream header, as described in the section called *XML Streams* in Chapter 5, is sent to `gnu.pipetree.com:5222` in an a attempt to establish a client connection.

XMLStream will raise an error if the connection cannot be established, we trap this, after a fashion, with the `try: ... except` shown here.

```
con.auth(Username,Password,Resource)
```

Once we're connected—our client has successfully exchanged XML stream headers with the server—we need to authenticate. The `auth` method of the `Jabber.Connection` object provides us with a simple way of carrying out the authentication negotiation, qualified with the `jabber:iq:auth` namespace and described in detail in the section called *User Authentication* in Chapter 6. Although we supply our password here in the script in plaintext ('`secret`'), the `auth` method will use the IQ-get (`<iq`

`type='get'...>`) to retrieve a list of authentication methods supported by the server. It will try and use the most secure, "gracefully degrading" to the least, until it finds one that is supported. [4]

Note the presence of the `resource` in the call. This is required for a successful client authentication regardless of the authentication method. Sending an IQ-set (`<iq type='set'...>`) in the `jabber:iq:auth` namespace without specifying a value in a`<resource/>` tag results in a "Not Acceptable" error 406; see Table 5-3 for a list of standard error codes and texts.

We're connected, and authenticated. The world is now our lobster, as an old friend used to say. We're not necessarily *expecting* to receive anything at this stage, and even if we did, we wouldn't really want to do anything with what we received anyway. At least at this stage. So we don't bother setting up any mechanism for handling elements that might appear on the stream.

```
con.send(Jabber.Message(cvsuser, message, subject="CVS Watch Alarm"))
```

All we do is send the notification message, in the `message` variable, to the user that was specified as an argument when the script was invoked, stored in the `cvsuser` variable. There are actually two calls here. The innermost—`Jabber.Message()`—creates a simple message element that looks like this:

```
<message to='[value in cvsuser variable]'>
  <subject>CVS Watch Alarm</subject>
  <body>[value in message variable]</body>
</message>
```

It takes two positional (and required) parameters; any other information to be passed (such as the `subject` in this example) must be supplied as *key=value* pairs. The outermost—`con.send()`—sends whatever it is given over the XML stream that the `Jabber.Connection` object `con` represents. In the case of the `Jabber.Message` call, this is the string representation of the object so created—i.e. our `<message/>` element.

Once the notification message has been sent, the script's work is done. We can therefore disconnect from the server before exiting the script.

```
con.disconnect()
```

Calling the `disconnect()` method of the `Jabber.Connection` sends an *unavailable* presence element to the server on behalf of the user that is connected:

```
<presence type='unavailable'/>
```

This is sent regardless of whether a `<presence/>` element was sent during the conversation, but does

no harm if one wasn't.

After sending the unavailable presence information, theXML stream is closed by sending the stream's closing tag:

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

This signifes to the server that the client wishes to end the conversation. Finally, the socket is closed.

The script in its entirety

Here's the script in its entirety.

```
import Jabber, XMLStream
import sys

Server   = 'gnu.pipetree.com'
Username = 'cvsmsg'
Password = 'secret'
Resource = 'cvsmsg'

cvsuser  = sys.argv[1]
message  = ''

for line in sys.stdin.readlines(): message = message + line

con = Jabber.Connection(host=Server)

try:
    con.connect()
except XMLStream.error, e:
    print "Couldn't connect: %s" % e
    sys.exit(0)

con.auth(Username,Password,Resource)
con.send(Jabber.Message(cvsuser, message, subject="CVS Watch Alarm"))
con.disconnect()
```

Notes

[1] You can find out more about CVS at http://www.cvshome.org.

[2] http://www.docbook.org.

[3] http://sourceforge.net/projects/jabberpy/ available at http://sourceforge.net/projects/jabberpy/

[4] This degradation typically will follow the pattern "*zero-knowledge* supported? No. Ok. How about *digest*? No? Ok. Then how about *plaintext*? No? Oh dear. This isn't going to work."

Prev | Home | Next

Messages, Presence, and Presence Subscription | Up | Dialup system watch

Dialup system watch

When working from home, my Linux server, which acts as a file server and router for the rest of the house, is connected to the Internet for pretty much most of the time. To my wife, I can justify the pay-per-minute dialup expense with work-related reasons. But when I'm away, those reasons aren't relevant. So for most of the time, the connection is down. Now and again though, I do need access to things on the server at home. After my wife grew tired of answering the phone and have me talk her through going downstairs into the cellar, getting the server to dial up, and finding out the IP address, I decided there was a better way. I could take one step in the right direction by getting the server to dial up at specified intervals and I would hop on at the pre-ordained times if I needed to. But there was a problem with this, which I'll describe in a second.

Most of the time at customer sites, I'm running a Jabber client of some sort. Whether it's WinJab on Windows, or Jarl in *cli* (command line) mode on a remote server over an **ssh** connection, I can get my fix. This platform—my Jabber client—turns out to be an ideal ready-made component for my solution.

Here's how it would work:

- Get the server to dial up and connect to the Internet regularly
- On connection, start a script that sends Jabber presence to me
- On disconnection, get the script to end

While the Jabber client sits on my desktop screen, I am aware of things going on in the Jabber-based conversation world; people would send me messages, my email alert mechanism would punt the subject line of important incoming mails to me, and I could see who (and what) was coming and going in my roster—the collection of people and services I have in my contact list. If I could somehow add to my roster a JID that represented my server at home, I could subscribe to the server's presence and know when it was available—connected to the Internet—and when it wasn't.

This would solve the problem I'd previously had with the timing of the automated dialup: I'd set the server to dial up every hour, or every two hours, and to stay online for ten minutes or so each time, but invariably due to timekeeping discrepancies, eddies in the space-time continuum, and the fact that I don't wear a watch, the poor timing between me and the server meant that I often missed the online window.

The preparation

Before diving into the script, it's necessary to do a bit of preparation. We're going to be using the *presence subscription* concept, which is is described in Chapter 5, and covered in more detail in the next section in this chapter. We're also going to have to get the script to run, and stay running, when the dialup connection is made, and have it stop when the dialup connection is ended.

Presence

Rather than get involved in the nitty gritty of presence subcriptions right now, let's use the tools that are around us to get things set up. In order for this to work, we need to be subscribed to the presence of the script that will be invoked when the server dials up and connects to the Internet. The script will connect to the Jabber server using a JID with a username that represents the Linux server: `myserver@gnu.pipetree.com`. My JID in this case is `dj@gnu.pipetree.com`, so we

just use whatever Jabber client happens to be at hand, say Jabber Instant Messenger (JIM), to effect both sides of the subscription.

Step 1: Create JID `myserver@gnu.pipetree.com`

We need to create the script's JID if it doesn't already exist. We can use the **reguser** script we wrote in the section called *User Registration Script* in Chapter 6 to do this:

```
[dj@yak dj]$ ./reguser gnu.pipetree.com username=myserver password=secret
[Attempt] (myserver) Successful registration
[dj@yak dj]$
```

Step 2: Subscribe to *myserver*'s presence

We start JIM with the JID `dj@gnu.pipetree.com`, and add `myserver@gnu.pipetree.com` to the roster. This should automatically send a presence subscription request to the JID. Adding the JID to the roster using JIM is shown in Figure 7-2.

Step 3: Accept presence subscription as *myserver*

Now connecting with the *myserver* JID, again, using the JIM client, we accept the presence subscription request from Step 2, so that `dj@gnu.pipetree.com` will automatically receive `myserver@gnu.pipetree.com`'s availability information. Whether or not *myserver* subscribes to *dj*'s presence is irrelevant in this case, as the script itself is not interested in the availability of anyone at all.

Figure 7-2. Adding `myserver@gnu.pipetree.com` to the roster

At this stage, the entry in `dj@gnu.pipetree.com`'s roster that represents the Linux server will indicate whether the script run at dialup time is active or not. If we continue to use the JIM client, we will see that this is shown by a yellow bulb, or no icon at all, respectively.

Starting and stopping the script

The dialup connection is set up using the Point to Point Protocol daemon **pppd**. This uses a program such as **chat** to talk to the modem and get it to dial the ISP. The **pppd** mechanism affords us an idea way to start and stop a script on the respective connection and disconnection of the line. When the connection has been made, the script `/etc/ppp/ip-up` is invoked and passed a number of connection-related parameters. Similarly `/etc/ppp/ip-down` is invoked when the connection is closed.

Some implementations of **pppd** also offer `/etc/ppp/ip-up.local` and `/etc/ppp/ip-down.local` which should be used in place of the `ip-up` and `ip-down` scripts if they exist.

So what we want to do is start our script with `ip-up[.local]` and stop it with `ip-down[.local]`. What these starter and stopper scripts might look like are shown in Example 7-3 and Example 7-4. They are simply shell scripts that share the process id (PID) of the Jabber script via a temporary file. The starter starts the Jabber script and writes the PID of that script to a file. The stopper kills the script using the PID.

Example 7-3. An `ip-up` starter script

```
#!/bin/sh

# Change to working directory
cd /jabber/java/

# Call the Jabber script and put to background
/usr/java/jdk1.3.1/bin/java -classpath jabberbeans.jar:. HostAlive $5 &

# Write the running script's PID
echo $! > /tmp/HostAlive.pid
```

Example 7-4. An `ip-down` stopper script

```
#!/bin/sh

# Simply kill the process using the
# JID written by the starter script
/bin/kill `cat /tmp/HostAlive.pid`

# Remove the PID file
/bin/rm /tmp/HostAlive.pid
```

Example 7-3 shows that we're passing through one of the parameters that **pppd** gives to the `ip-up` script: the remote IP address—that by which the server is known during its temporary connection to the Internet—in the `$5` variable. [1] This IP address can be passed along as part of the availability information in the `<presence/>` element, so that the recipient (*dj*) can see what IP address has been assigned to the server.

The Jabber script

As you might have guessed from looking at Example 7-3, we're going to write this script in Java. We'll use the Jabberbeans library; see the section called *The Software* in Chapter 1 for details of where to get this library and what the requirements are.

Let's start by importing the libraries (the classes) we would like to use:

```
import org.jabber.jabberbeans.*;
import org.jabber.jabberbeans.Extension.*;
import java.net.InetAddress;
```

The Jabberbeans Jabber library is highly modular and designed so we can pick only the features that we need; in this case, however, we're just going to import the whole set of classes within the `org.jabber.jabberbeans` and `org.jabber.jabberbeans.Extension` packages, for simplicity.

We're also going to be manipulating the Jabber server's hostname, so we pull in the `InetAddress` class for convenience.

The script must connect to the Jabber server on `gnu.pipetree.com` as the *myserver* user. We define some constants for this:

```
public class HostAlive
{
  public static final String SERVER   = "gnu.pipetree.com";
  public static final String USER     = "myserver";
  public static final String PASSWORD = "secret";
  public static final String RESOURCE = "alive";
```

In the same way as our Python-based CVS notification script earlier in this chapter, we also start off by building a connection to the Jabber server. As before, it's a two-stage process. The first stage is to create the connection object:

```
  public static void main(String argv[])
  {

    ConnectionBean cb=new ConnectionBean();
```

A `ConnectionBean` object represents the connection between your script and the Jabber server. All XML fragments;—Jabber elements—pass through this object.

Then it's time to attempt the socket connection and the exchange of XML stream headers:

```
    InetAddress addr;

    try
    {
      cb.connect(addr=InetAddress.getByName(SERVER));
    }
    catch (java.net.UnknownHostException e)
    {
      //from getByName()
      System.out.println("Cannot resolve " + SERVER + ":" + e.toString());
      return;
```

```
}
catch (java.io.IOException e)
{
  //from connect()
   System.out.println("Cannot connect to " + SERVER);
   return;
}
```

We create an Internet address object in `addr` from the hostname assigned to the `SERVER` constant. As the creation of the `addr` instance may throw an exception (*unknown host*), we combine the instantiation with the `connection()` call on our `ConnectionBean` object which may also throw an exception of its own—if there is a problem connecting.

At this stage, we're connected and have successfully exchanged the XML stream headers with the Jabber server. So now we must authenticate:

```
InfoQueryBuilder iqb=new InfoQueryBuilder();
InfoQuery iq;
IQAuthBuilder iqAuthb=new IQAuthBuilder();

iqb.setType("set");


iqAuthb.setUsername(USER);
iqAuthb.setPassword(PASSWORD);
iqAuthb.setResource(RESOURCE);

try
{
  iqb.addExtension(iqAuthb.build());
}
catch (InstantiationException e)
{
  //building failed ?
  System.out.println("Fatal Error on Auth object build:");
  System.out.println(e.toString());
  System.exit(0);
}


try
{
  //build the full InfoQuery packet
  iq=(InfoQuery)iqb.build();
}
catch (InstantiationException e)
{
  //building failed ?
  System.out.println("Fatal Error on IQ object build:");
  System.out.println(e.toString());
  return;
}
```

```
    cb.send(iq);
```

Yes, that's an awful lot. Let's take it piece by piece.

Figure 7-3 shows how the objects in this section of code interrelate and represent various parts of what we're trying to do—which is to construct an authorization element. This takes the form of an IQ-set containing a `<query/>` tag qualified by the `jabber:iq:auth` namespace like this: [2]

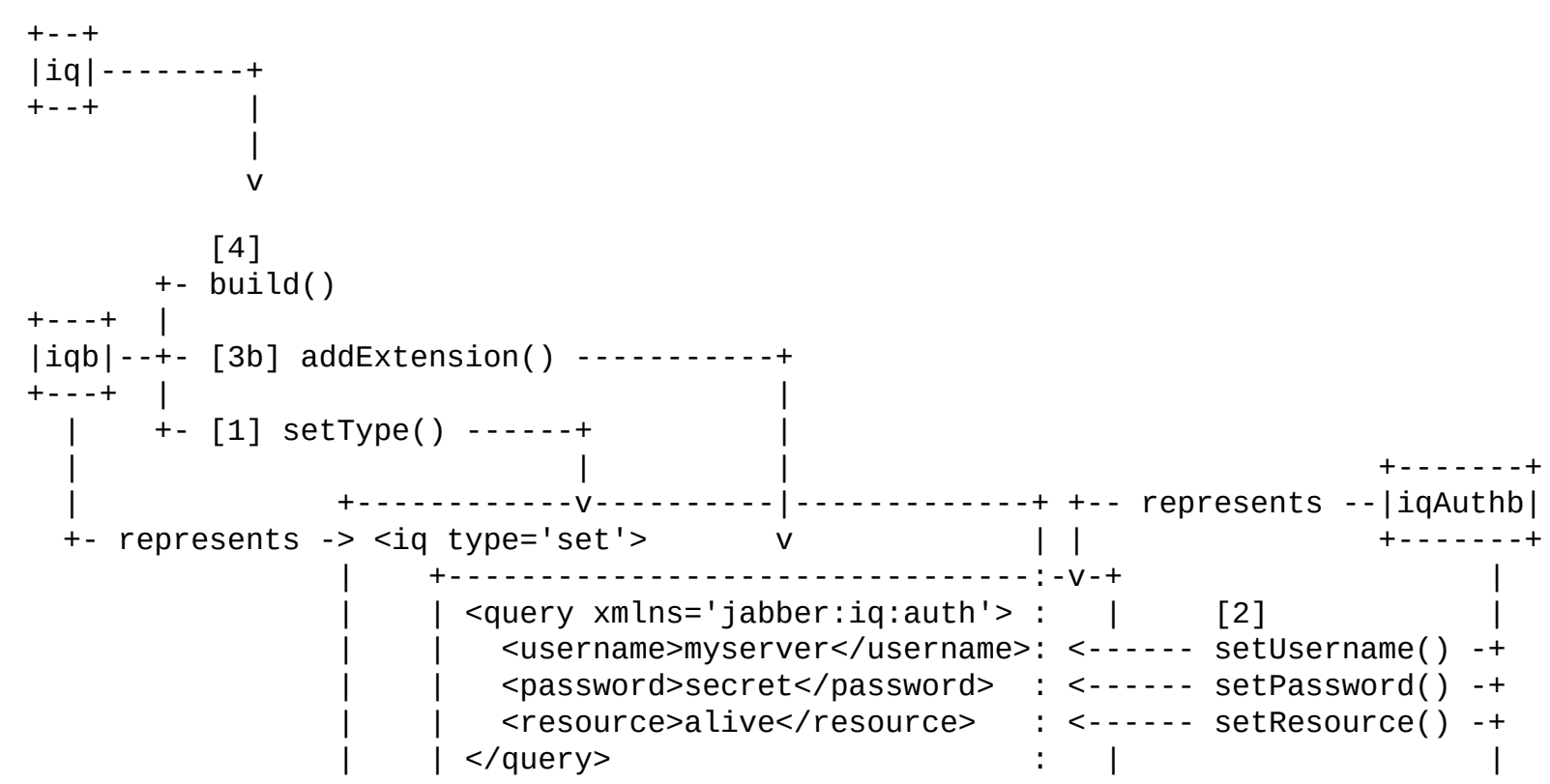
```
<iq type='set'>
  <query xmlns='jabber:iq:auth'>
    <username>myserver</username>
    <password>secret</password>
    <resource>alive</resource>
  </query>
</iq>
```

Constructing Jabber elements with Jabberbeans uses so-called *builders* that allow individual element components to be created separately and then fused together into a final structure. In our code, we use two builders: an `InfoQueryBuilder` to construct the `<iq/>` envelope and an `IQAuthBuilder` to construct the `<query/>` content.

Taking the code step-by-step, we create or declare each of the three things *iqb*, *iq*, and *iqAuthb* in Figure 7-3:

```
    InfoQueryBuilder iqb=new InfoQueryBuilder();
    InfoQuery iq;
    IQAuthBuilder iqAuthb=new IQAuthBuilder();
```

Figure 7-3. Creating the authorization packet



```
|    +--------------------------------:---            build() -+
| </iq>                                |               [3a]
+--------------------------------------+
```

There are numbered steps in Figure 7-3; these follow what happens in the rest of the authentication preparation:

Step [1] Set the `type` attribute of the IQ

We call the `setType()` method on our *iqb* object that represents our outer IQ envelope to set the value of the `type` attribute:

```
iqb.setType("set");
```

Step [2] Set the values for the parts of the authorization section of the element

Having constructed our *iqAuthb* object, which represents the `<query/>` portion of our element, we fill the values with these calls:

```
iqAuthb.setUsername(USER);
iqAuthb.setPassword(PASSWORD);
iqAuthb.setResource(RESOURCE);
```

Step [3] Crystallize `iqAuthb` and add it to the IQ object

Once the values inside the authorization `<query/>` tag are set, we can call the `build()` method on the object representing that tag (*iqAuthb*) to generate an extension object—to *crystallize* the tag— that can then be attached as an extension to the *iqb* object using the `addExtension()` method:

```
try
{
  iqb.addExtension(iqAuthb.build());
}
...
```

Step [4] Crystallize *iqb* and assign it to the *iq* object

In the same way that we crystallized the authorization `<query/>` tag, we can crystallize the whole element and assign it to *iq*:

```
try
{
  //build the full InfoQuery packet
  iq=(InfoQuery)iqb.build();
}
...
```

Once we've constructed our authorization element, now held as the *iq* object, we can send it down the stream to the Jabber server with the `send()` method of the ConnectionBean object `cb`:

```
cb.send(iq);
```

Finally, once we've authenticated, we can construct our presence packet and send it. [3] This is achieved using the same technique as before. We construct a new object to represent the presence packet denoting general availability—`<presence/>`:

```
PresenceBuilder pb=new PresenceBuilder();
```

In this case, there are no namespace-qualified extensions to add to our `<presence/>` element, but we do want to add the IP address that was passed into the script and available in `argv[0]`. We can use the `setStatus()` method on our presence object to set the optional `<status/>` to contain that IP address:

```
pb.setStatus(argv[0]);
```

After this, we can go ahead and crystallize the element, which will look like this:

```
<presence>
  <status>123.45.67.89</status>
</presence>
```

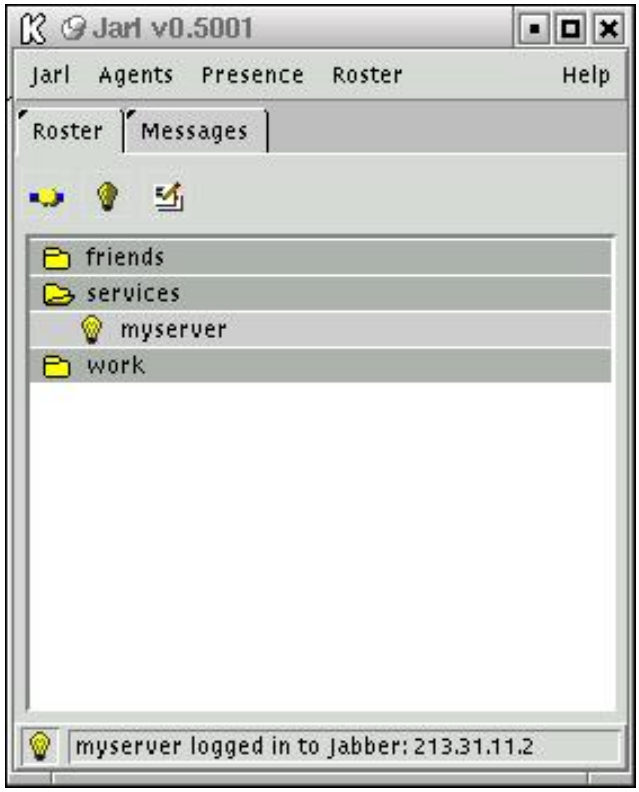
After the crystallization with the `build()` call, we send it down the stream in the same way that as the authorization `<iq/>` element:

```
try
{
  cb.send(pb.build());
}
catch (InstantiationException e)
{
  System.out.println("Fatal Error on Presence object build:");
  System.out.println(e.toString());
  return;
}
```

As for each of the `build()` calls, we must trap a possible exception that `build()` throws if it can't complete (for example, due to lack of information). This is the *InstantiationException*.

We can see the results of *myserver* sending such an information-laden `<presence/>` element to *dj* in Figure 7-4. As the server connects to the Internet, the Java script is started via the `ip-up` script and relays the assigned IP address which is shown in Jarl's status bar as the availability information reaches *dj*'s client.

Figure 7-4. *myserver* becoming available and relaying its IP address



All that remains for the script to do now is to hang around. While the XML stream to the Jabber server remains, and the connection is not broken, our availabiliy will remain as it was as described by the simple `<presence/>` element we sent. So we simply go into a sort of hibernation. We have no hope of escaping, but it should be taken care of by our `ip-down` script as described earlier.

```
while (true) {
  try {
    Thread.sleep(9999);
  }
  catch (InterruptedException e)
  {
    System.out.println("timeout!");
  }
}
```

In fact, when the `ip-down` script kills our script, the socket connection will be closed, but there was no clean disconnect—no `<presence type='unavailable/>` was sent by the script to the Jabber server. In this case, the Jabber server will notice that the socket was closed, and generate an unavailable presence element on behalf of the client.

The script in its entirety

Here's the script in its entirety.

```
import org.jabber.jabberbeans.*;
import org.jabber.jabberbeans.Extension.*;
import java.net.InetAddress;

public class HostAlive
```

```
{
  public static final String SERVER   = "gnu.pipetree.com";
  public static final String USER     = "myserver";
  public static final String PASSWORD = "secret";
  public static final String RESOURCE = "alive";

  public static void main(String argv[])
  {

    ConnectionBean cb=new ConnectionBean();

    InetAddress addr;

    try
    {
      cb.connect(addr=InetAddress.getByName(SERVER));
    }
    catch (java.net.UnknownHostException e)
    {
      //from getByName()
      System.out.println("Cannot resolve " + SERVER + ":" + e.toString());
      return;
    }
    catch (java.io.IOException e)
    {
      //from connect()
       System.out.println("Cannot connect to " + SERVER);
       return;
    }

    InfoQueryBuilder iqb=new InfoQueryBuilder();
    InfoQuery iq;
    IQAuthBuilder iqAuthb=new IQAuthBuilder();

    iqb.setType("set");


    iqAuthb.setUsername(USER);
    iqAuthb.setPassword(PASSWORD);
    iqAuthb.setResource(RESOURCE);

    try
    {
      iqb.addExtension(iqAuthb.build());
    }
    catch (InstantiationException e)
    {
      //building failed ?
      System.out.println("Fatal Error on Auth object build:");
      System.out.println(e.toString());
      System.exit(0);
    }
```

```
try
{
  //build the full InfoQuery packet
  iq=(InfoQuery)iqb.build();
}
catch (InstantiationException e)
{
  //building failed ?
  System.out.println("Fatal Error on IQ object build:");
  System.out.println(e.toString());
  return;
}

cb.send(iq);

PresenceBuilder pb=new PresenceBuilder();
pb.setStatus(argv[0]);

try
{
  cb.send(pb.build());
}
catch (InstantiationException e)
{
  System.out.println("Fatal Error on Presence object build:");
  System.out.println(e.toString());
  return;
}

while (true) {
  try {
    Thread.sleep(9999);
  }
  catch (InterruptedException e)
  {
    System.out.println("timeout!");
  }
}
```

Notes

[1] The parameters that are passed from **pppd** to the `ip-up` script are: *interface-name*, *tty-device*, *speed*, *local-link-local-address*, *remote-link-local-address*, and *ipparam*. It's the *remote-link-local-address* that we're interested in here.

[2] We're not bothering in this example to ask the server for the authentication methods it supports and are just going ahead with a plaintext attempt.

[3] We're assuming our password is correct here, to keep this example straightforward. If it isn't correct, there's not much we can do at this point anyway.

Presence-sensitive CVS notification

In the section called *CVS notification* earlier in this chapter we replaced the email-based CVS notification mechanism with a Jabber-based one. The script used was extremely simple—it connected to the Jabber server specified, authenticated, and sent off the notification message to the recipient JID.

What if we wanted to make the script "sensitive"? Jabber's presence concept could help us here; if we extended the mechanism to allow for the building of presence-based relationships between the notification script and the notification recipients, we can make the sending of the notification message dependent on the recipient's availability. "Presence-based relationships" refers to the *presence subscription* mechanism described in the sidebar in the section called *Presence Attributes* in Chapter 5.

Here's how it would work:

- Each potential recipient adds the JID used by the CVS notification script to their roster, and sends a subscription request to it. [1]
- The notification script, on receipt of the presence subscription from a recipient, accepts the request and reciprocates by sending a subscription request back to that recipient.
- On receipt of the presence subscription from the notification script, the recipient accepts the request.
- When the notification script starts up to send a message, it announces its own availability with a `<presence/>` element, which causes the availability of the JIDs to which it has a presence subscription to be sent to it. Based on these `<presence/>` packets received, it can make a decision as to whether to send the notification message or not.

  The decision we're going to use here is an arbitrary one: If the recipient is online, we'll send the message, unless they've specified that they don't want to be disturbed, with the `<show>dnd</show>` element.

Subscription relationships

This method will result in "balanced" subscription relationships between script and recipients. In other words, the script is subscribed to a recipient's presence, and vice versa.

Of the two presence subscription "directions", the one where the *notification script* subscribes to the *recipient's* presence (as opposed to the one where the *recipient* subscribes to the *notification script's* presence) is by far the most important. While it's not critical that the recipients know when the notification script is connected and active, it's essential that the notification script know about a recipient's availability at the time it wants to send a message.

So would it be more appropriate to create "unbalanced" subscription relationships?

An unbalanced relationship is one where one party knows about the other party's availability, but *not* vice versa. The idea for sensitizing our notification script will work as long as the script can know about the availability of the recipients. Whether or not the opposite is true is largely irrelevant.

Nevertheless, it's worth basing the interaction on balanced, or reciprocal, presence subscriptions, primarily for simplicity's sake, and also for the fact that most Jabber clients (and most *users* of these clients) tend to cope well and consistently with "balanced" subscriptions, whereby the representation and interpretation of unbalanced relationships is dealt with and understood in different manners. Some clients use a "*Lurker*" group to classify one-way presence subscriptions from other JIDs: a "lurker" being one

that can see you while you can't see it.

Far from being nebulous concepts, balanced and unbalanced subscription relationships are characterized technically by values of a certain attribute specified in each item—each JID—in a roster: the `subscription` attribute of the `<item/>` tags within the roster. As we progress through the extensions to our CVS notification script, we'll pause at various stages to examine these values.

> **Anthropomorphism:** It's worth pointing out at this stage that adding a JID that's used by a *script* to connect to Jabber is slightly symbolic of the extension of the instant messaging world into the wider arena of A2P messaging. Adding a *service* JID to your roster and sharing presence information with that service is such a powerful concept and one which can be used to base A2P solutions upon.

Extending the script

The script, as it stands in the section called *The script in its entirety*, is what we want to extend and make sensitive to presence. We'll walk through the additions, and then present the extended script at the end of this section.

We bring in a string function that we'll be needing later in the script to chop up JIDs into their component parts (*username*, *hostname*, and *resource*):

```
import Jabber, XMLStream
import sys
from string import split

cvsuser  = sys.argv[1]
message  = ''
```

Callbacks and handlers

The next addition to the script is a callback to handle `<presence/>` elements. The callback in this script takes the form of a subroutine called `presenceCB()` ("presence CallBack").

Jabber programming and *callbacks*

When programming all but the simplest Jabber scripts, you're going to be using *callbacks*. Callbacks are also known as *handlers*.

Rather than purely procedural programming ("do this, then do that, then do the other"), we need a different model to cope with the *event-based* nature of Jabber, or more precisely, the event-based nature of how we converse using the Jabber protocol over an XML stream. Although we control what we *send* over the XML stream connection that we've established with the Jabber server, we can't control what we *receive*, and more importantly, we can't control *when* we receive it. We need an event-based programming model to be able to *handle* the protocol elements as they arrive.

The libraries available for programming with Jabber offer *callback* mechanisms. With these callback mechanisms we can register subroutines with the mechanism that's handling the reception of XML stream fragments, so that when an element appears on the incoming stream (a fragment in the stream document that the Jabber server is sending to us—see the section called *XML Streams* in Chapter 5), the Jabber library can pass it to the appropriate subroutine in our script for us to act upon.

Figure 7-5 shows the relationship between the library and script, and the sequence of events surrounding registering a handler

and having it called. Here are the steps shown:

[1] Register handler

The script uses a library function to register a subroutine —in this case it's `presenceCB()`—as a handler with the library. In the registration, the subroutine is assigned as a handler for `<presence/>` elements.

[2] `<presence/> element arrives`

An XML fragment arrives on the stream, sent by the Jabber server.

[3] Parse, and create node

The fragment is parsed into its component parts by an XML parser, and a node is created. A "node" is simply a term used to describe a succinct XML fragment—containing attributes, data, and child tags— that usually is in the form of an object that is programmatically accessible. The node creation step is theoretically optional; we could have the library pass on the fragment in a simple string representation form, but that would put the onus on the script to parse the string before being able to manipulate the fragment that the string represented.

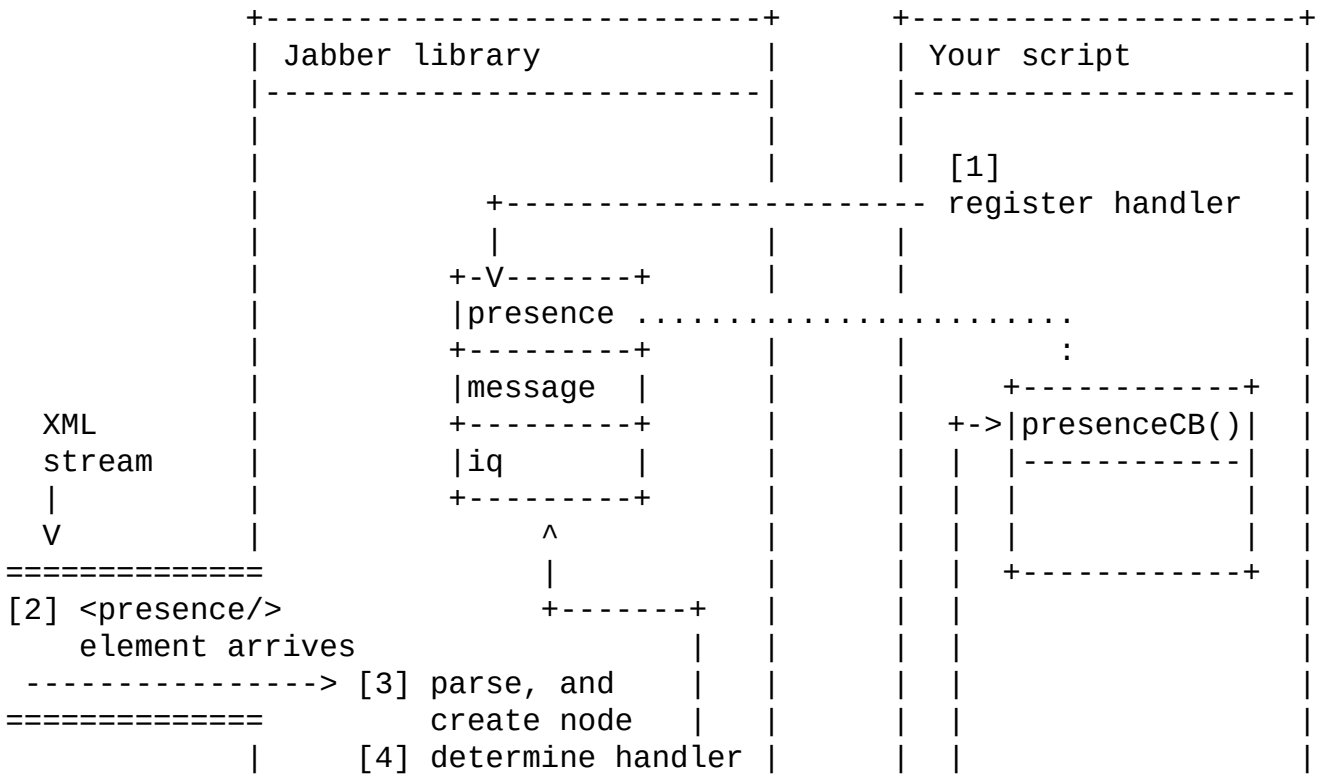
[4] Determine handler

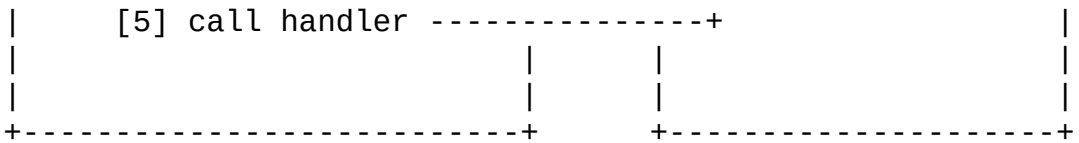
Once parsed, the library looks at what sort of element, or node, the fragment is, and determines what (if any) handler has been registered. In this case, it's a `<presence/>` element, and finds that the subroutine `presenceCB()` has been registered as a handler for `<presence/>` elements.

[5] Call handler

The library calls the handler `presenceCB()` in the script, passing in the node. It may pass in other information too (for example, the Python library `Jabber` also passes in the stream connection object, as we'll see in a moment), and the Perl library `Net::Jabber` also passes in a session ID relating to the stream. [2]

Figure 7-5. Handlers and the relationship between the Jabber library and your script





This is what the callback for handling `<presence/>` elements looks like:

```
def presenceCB(con, prs):

    type = prs.getType()
    parts = split(prs.getFrom(), '/')
    who = parts[0]

    if type == None: type = 'available'

    # subscription request:
    # - accept their subscription
    # - send request for subscription to their presence
    if type == 'subscribe':
        print "subscribe request from %s" % (who)
        con.send(Jabber.Presence(to=who, type='subscribed'))
        con.send(Jabber.Presence(to=who, type='subscribe'))

    # unsubscription request:
    # - accept their unsubscription
    # - send request for unsubscription to their presence
    elif type == 'unsubscribe':
        print "unsubscribe request from %s" % (who)
        con.send(Jabber.Presence(to=who, type='unsubscribed'))
        con.send(Jabber.Presence(to=who, type='unsubscribe'))

    elif type == 'subscribed':
        print "we are now subscribed to %s" % (who)

    elif type == 'unsubscribed':
        print "we are now unsubscribed to %s" % (who)

    elif type == 'available':
        print "%s is available (%s/%s)" % (who, prs.getShow(), prs.getStatus())
        if prs.getShow() != 'dnd' and who == cvsuser:
            con.send(Jabber.Message(cvsuser, message, subject="CVS Watch Alarm"))

    elif type == 'unavailable':
        print "%s is unavailable" % (who)
```

Phew! Let's take it a bit at at time. The first thing to note is what's specified in the subroutine declaration:

```
def presenceCB(con, prs):
```

As a handler, the subroutine `presenceCB()` will be passed the *connection object* in `con` and the *presence node* in `prs`. `con` is the same connection object that is created later in the script (`con = Jabber.Connection(host=Server)`) and is passed in for convenience, as it's quite likely we're going to want to use it, say, to send something back over the stream.

Presence nodes

The presence node in `prs` is an object representation of the XML fragment that came in over the stream and was parsed into its component parts. The object is an instance of the `Jabber.Presence` class, which is simply a specialization of the more generic `Jabber.Protocol` class, as are the other classes that represent the other two Jabber protocol elements that are to be expected: `Jabber.Message` and `Jabber.Iq`. The `Jabber.Protocol` class represents protocol elements in general.

As such, there are a number of `<presence/>` element-specific methods we can call on the `prs` object, such as `getShow()` and `getStatus()` (which return the values of the `<show/>` and `<status/>` tags—children of the `<presence/>` element—respectively), and general element methods such as `getID`, which returns the value of any `id` attribute assigned to the element, and `setTo()` which can be used to address the element—to set the value of the `to` attribute.

The first thing the handler does is to call a few of these element methods to determine the *type* of `<presence/>` element (presence types are described in the section called *The Presence Element* in Chapter 5), and who it's coming from:

```
type = prs.getType()
parts = split(prs.getFrom(), '/')
who = parts[0]
```

When the notification script is called, the JID found in the CVS `users` file is substituted for the `%s` in the formula contained in the CVS `notify` file. So if the user *dj* were to be notified, the JID passed to the script would be `dj@gnu.pipetree.com`.

The way JIDs are passed around *independently* of the context of a Jabber session is usually in the simpler form: `username@hostname`. That is, without the resource suffix: `username@hostname/resource`. As described in Chapter 5, the resource is primarily used to distinguish individual sessions belonging to one Jabber user.

But when our Jabber library—and subsequently a handler subroutine in our script—receives an element, it contains a `from` attribute whose value has been stamped by the Jabber server as it passes through. [3] The value represents the session, the *connection*, from which the `<presence/>` element was sent, and as such, includes a resource suffix. So in order to properly match up the source JID for any incoming `<presence/>` element with the JID specified when the script was invoked (contained in the `cvsuser` variable), we need to strip off this resource suffix. The remaining `username@hostname` part is captured in the `who` variable.

There's one more step to determine the presence type. The `type` attribute is *optional*, and its absence signifies the default presence type, which is "available". So we effect this default substitution here to make the subsequent code clearer:

```
if type == None: type = 'available'
```

Presence subscription

At this stage, we want to take different actions depending on what sort of presence information has arrived. Recalling the sequence of events in the reciprocal presence subscription exchange described earlier in this chapter, one of the activities is for a potential notification recipient to subscribe to the presence of the script's JID.

Request to subscribe

This subscription request is carried in a `<presence/>` element, with a type of "*subscribe*". Example 7-5 shows what a typical subscription request would look like.

Example 7-5. A presence subscription request from *dj@gnu.pipetree.com*

```
<presence type='subscribe' to='cvsmsg@gnu.pipetree.com'
                           from='dj@gnu.pipetree.com/work'/>
```

Pause:

> At this stage, *dj@gnu.pipetree.com* has just sent a request to subscribe to the script's presence. The subscription relationship between the two parties is nondescript, and this is reflected in the details of the item in *dj*'s roster that relates to the script's JID:
>
> ```
> <item jid='cvsmsg@gnu.pipetree.com' subscription='none' ask='subscribe'/>
> ```
>
> The relationship itself is reflected in the `subscription` attribute, and the current state of the relationship is reflected in the `ask` attribute.

If one of these is received, we want the script to respond by accepting their subscription request and requesting a subscription to *their* presence in return.

This incoming subscription request is handled here:

```
    # subscription request:
    # - accept their subscription
    # - send request for subscription to their presence
    if type == 'subscribe':
        print "subscribe request from %s" % (who)
        con.send(Jabber.Presence(to=who, type='subscribed'))
        con.send(Jabber.Presence(to=who, type='subscribe'))
```

Each call to the `Jabber.Presence` class constructor creates a node representing a `<presence/>` element. The two parameters passed in the call are fairly self-explanatory: we specify *to whom* the `<presence/>` element should be sent, and the *type*.

If the presence subscription request came in from the JID *dj@gnu.pipetree.com*, then the XML represented by the node created in the first call here (specifying a presence type of "subscribed") would look something like that in Example 7-6.

Example 7-6. Acceptance of a presence subscription request from *dj@gnu.pipetree.com*

```
<presence type='subscribed' to='dj@gnu.pipetree.com'/>
```

Addressing `<presence/>` elements

It's worth pointing out here that there's a subtle difference between sending `<presence/>` elements in a presence subscription conversation, and sending general "availability" `<presence/>` elements.

In the first case, we use a `to` attribute, because our conversation is one-to-one. In the second, we don't; Our unaddressed availability information is caught by the server and in turn sent on to those entities that are subscribed to your presence.

Although you *can* send `<presence/>` elements that convey availability information directly *to* a JID, it's not normal. However, explicitly addressing the elements in a subscription scenario is essential.

> There's another situation in which such "directed" (explicitly addressed) `<presence/>` elements are used - to partake of the services of the Availability Tracker. This is described in the "*Availability Tracker*" sidebar in the section called *Presence Attributes* in Chapter 5.

Once constructed, each of the `Jabber.Presence` nodes are sent back along the stream with the `con.send()` calls.

Pause:

> Now the script has accepted *dj*'s subscription request, *dj*'s roster item for the script now reflects the new relationship:
>
> ```
> <item jid='cvsmsg@gnu.pipetree.com' subscription='to'/>
> ```
>
> `subscription='to'` denotes that the subscription relationship is currently one way—*dj* has a subscription to the script. There's no `ask` attribute as there's no current request going from *dj* to the script.
>
> While *dj*'s roster item for the script shows a `subscription` value of "to", the *script*'s roster item for *dj* shows a `subscription` value of "from":
>
> ```
> <item jid='dj@gnu.pipetree.com' subscription='from' ask='subscribe'/>
> ```
>
> which shows that the script has a subscription *from dj*.
>
> Furthermore, remember that the script not only accepts *dj*'s subscription request, but sends a reciprocal one of its own. (Hence the `ask='subscribe'` status in the item.) When *dj* accepts this request, the roster item changes yet again to reflect the balanced relationship:
>
> ```
> <item jid='cvsmsg@gnu.pipetree.com' subscription='both'/>
> ```

Request to unsubscribe

We want the script to handle requests to *unsubscribe* from its presence in the same way:

```
    # unsubscription request:
    # - accept their unsubscription
    # - send request for unsubscription to their presence
    elif type == 'unsubscribe':
        print "unsubscribe request from %s" % (who)
        con.send(Jabber.Presence(to=who, type='unsubscribed'))
        con.send(Jabber.Presence(to=who, type='unsubscribe'))
```

The only difference between this section and the previous one is that it deals with requests to *unsubscribe* as opposed to *subscribe* to presence. Otherwise it works in exactly the same way. A sequence of `<presence/>` elements used in an "unsubscription conversation" between *dj* and the script, and the changes to the roster `<item/>` tags on each side, is shown in Figure 7-6.

Figure 7-6. `<presence/>` elements and roster `<item/>`s in an "unsubscription conversation"

`NOTE: This diagram needs some design explanation and help!`


Elements from dj
Elements from cvsmsg
dj's roster items
cvsmsg's roster items
<item jid='cvsmsg@gnu.pipetree.com' subscription='both'/>
<item jid='dj@gnu.pipetree.com' subscription='both'/>
<presence type='unsubscribe' to='cvsmsg@gnu.pipetree.com'/>
<item jid='cvsmsg@gnu.pipetree.com' subscription='both' ask='unsubscribe'/>
<item jid='dj@gnu.pipetree.com' subscription='both'/>
<presence type='unsubscribed' to='dj@gnu.pipetree.com'/>
<item jid='cvsmsg@gnu.pipetree.com' subscription='from'/>
<item jid='dj@gnu.pipetree.com' subscription='to'/>

Accepted subscription or unsubscription

While we must take action on presence types *subscribe* and *unsubscribe* we don't really need to do anything for their *acknowledgement* counterparts. We can see these counterparts - *subscribed* and *unsubscribed* ("I have accepted your request and you are now (un)subscribed to my presence.")

Just for illustration purposes, we'll include a couple of conditions to let us know what's going on when the script runs:

```
    elif type == 'subscribed':
        print "we are now subscribed to %s" % (who)
```

```
    elif type == 'unsubscribed':
        print "we are now unsubscribed to %s" % (who)
```

Availability information

Apart from the types of `<presence/>` element covering the presence subscription process, we should also expect the basic availability elements:

```
<presence>...</presence>
```

and

```
<presence type='unavailable'/>
```

Availability

It is an available `<presence/>` element on which the functionality of the script hinges.

```
    elif type == 'available':
        print "%s is available (%s/%s)" % (who, prs.getShow(), prs.getStatus())
        if prs.getShow() != 'dnd' and who == cvsuser:
            con.send(Jabber.Message(cvsuser, message, subject="CVS Watch Alarm"))
```

This `presenceCB()` subroutine is set up to handle `<presence/>` elements. In a typical execution scenario, where the script is subscribed to the presence of many potential CVS notification recipients, the subroutine is going to be called to handle the availability information of all those recipients that happen to be connected to Jabber at the moment of notification. We're only interested in the availability information of one particular recipient (`who == cvsuser`) as well as checking on the contents of the `<show/>` tag.

If we get a match, we can send the notification message by creating a `Jabber.Message` node that will look like this:

```
<message to='dj@gnu.pipetree.com'>
  <subject>CVS Watch Alarm</subject>
  <body>
    testproject file4
    ---
    Triggered edit watch on /usr/local/cvsroot/testproject
    By piers
  </body>
</message>
```

As before, once created, the node can be sent with the `con.send()` method call.

Unavailability

Like the conditions for the presence subscription and unsubscription acknowledgements, we're including a final condition to deal with the case where a recipient disconnects from the Jabber server during the execution of our script: an *unavailable* `<presence/>` element will be sent:

```
    elif type == 'unavailable':
```

```
        print "%s is unavailable" % (who)
```

We're simply logging such an event for illustration purposes.

Connection and authentication

Most of the main part of the script is the same as the non-sensitive version from the section called *CVS notification*; reading in the notification message, preparing a connection to the Jabber server, and trying to connect.

```
for line in sys.stdin.readlines(): message = message + line

con = Jabber.Connection(host=Server)

try:
    con.connect()
except XMLStream.error, e:
    print "Couldn't connect: %s" % e
    sys.exit(0)

con.auth(Username,Password,Resource)
```

Registration of <presence/> handler

While we've *defined* our `presenceCB()` subroutine to handle `<presence/>` packets, we haven't actually told the Jabber library about it. The call to the `setPresenceHandler()` method of the connection object does this for us, performing the "Register handler" step shown in Figure 7-5:

```
con.setPresenceHandler(presenceCB)
```

Request for roster

It's easy to guess what the next method call does:

```
con.requestRoster()
```

It makes a request for the roster by sending an IQ-get with a query qualified by the `jabber:iq:roster` namespace:

```
<iq type='get' id='3'>
  <query xmlns='jabber:iq:roster'/>
</iq>
```

to which the server responds with an IQ-result:

```
<iq type='result' id='3'>
  <query xmlns='jabber:iq:roster'>
    <item jid='dj@gnu.pipetree.com' subscription='both'/>
    <item jid='piers@jabber.org' subscription='both'/>
    <item jid='shiels@jabber.org' subscription='both'/>
    ...
  </query>
```

```
</iq>
```

However, as there are no explicit references to the roster anywhere in the script, it's not as easy to guess why we request the roster in the first place. We know that our client-side copy is merely a "slave" copy, and even more relevant here, we know that *subscription* information in the roster `<item/>` tags is managed by the server—we as a client don't need to (in fact, *shouldn't*) do anything to maintain the `subscription` and `ask` attributes and keep them up to date.

So why do we request it?

Basically, it's because there's a fundamental difference between `<presence/>` elements used to convey availability information, and `<presence/>` elements used to convey presence subscription information. If John sends Jim availability information in a `<presence/>` element, whether directly (with a `to` attribute) or indirectly (through the distribution of that element by the server to Jim as a subscriber to John's presence), and Jim's offline, on holiday, it doesn't make much sense to store and forward it to him when he next connects:

[*Jim connects to Jabber after being away for two weeks*]

Jabber server: "*Here's some availability information for John, dated 9 days ago.*"

Jim: "*Who cares?*"

`<presence/>` elements conveying availability information are *not* stored and forwarded if they can't be delivered because the intended recipient is offline. What would be the point?

However, `<presence/>` elements that convey subscription information are a different kettle of fish. While it's not important that I'm punted out of date availability information when I next connect to my Jabber client, any subscription (or unsubscription) requests or confirmations that were sent to me *are* important. So they need to be stored and forwarded.

As we've already seen, the presence subscription mechanism and rosters are inextricably linked. And if we look briefly under the covers, we see how this is so. When a presence subscription request is sent to a user, it runs the gauntlet of modules in the JSM (see the section called *Component Connection Method* in Chapter 4 for details on what these modules are). The roster-handling module `mod_roster` grabs this request, and, just in case the recipient turns out not to be connected, *stores* it.

And here's how intertwined the presence subscription mechanism and rosters really are: *The request is stored as a cluster of attribute details within an `<item/>` tag in the roster belonging to the* recipient *of the presence subscription request.* It looks like this:

```
<item jid='user@hostname' subscription='none' subscribe='' hidden=''/>
```

On receipt of a presence subscription request, the `mod_roster` module will create the roster item if it doesn't exist already, and then assign the attributes related to presence subscription—`subscription='none'` and `subscribe=''`—to it. There's no `ask` attribute, as this is only assigned to the item on the roster belonging to the *sender*, not the *receiver*, of the subscription request.

The `subscribe` attribute is used to store the reason for the request, that, if specified, is carried in the `<status/>` tag of the `<presence/>` element that conveys the request. If no reason is given, the value for the attribute is empty, as shown here. Otherwise it will contain what was stored in the `<status/>` tag. Example 7-7 shows a presence subscription request that carries a reason.

Example 7-7. A presence subscription request with a reason

```
<presence to='dj@gnu.pipetree.com'>
  <status>I'd like to keep my eye on you!</status>
</presence>
```

The `hidden` attribute is used internally by `mod_roster` to mark the item as non-displayable; it effectively is a pseudo `<item/>`, that, when brought to life, actually turns out to be a `<presence/>` element. So when a request for the roster is made, `mod_roster` makes sure that it doesn't send these "hidden" items. The `hidden` attribute always has an empty value, as shown here.

After storing the subscription request, `mod_roster` will actually send the original `<presence/>` element that conveyed that request. If the recipient is online, *and* if the recipient has already made a request for their roster. As sending an availability presence packet:

```
<presence/>
```

is a kick to the `mod_offline` module to punt any messages stored offline in that user's absence, so requesting the roster:

```
<iq type='get'><query xmlns='jabber:iq:roster'/></iq>
```

is a kick to the `mod_roster` module to punt any subscription requests stored offline in that user's absence.

Sending of availability information

Ok. We've connected, authenticated, defined and registered our `<presence/>` callback, and kicked `mod_roster`. Now we need to announce our own availability in the form of a simple `<presence/>` element:

```
<presence/>
```

We can do this by calling the `sendInitPresence()` method on our connection object:

```
con.sendInitPresence()
```

This availability information will be distributed to all the entities that are subscribed to the script's presence and are online at that moment. It will also signify to the Jabber server that we are properly online—in which case it can forward to us any messages that had been stored up in our absence.

We're not really expecting any `<message/>` elements; indeed, we haven't set up any subroutine to handle them so they'd just be thrown away by the library anyway. The real reason for sending presence is so that the server will actively go and probe those in a presence subscription relationship with the script and report back on those who are available (who have themselves sent their presence during their current session). This causes `<presence/>` elements to arrive on the stream, which make their way to our `presenceCB()` handler.

Waiting for packets

Once eveything is set up, and the script has announced its presence, it really just needs to sit back and listen to the `<presence/>` elements that come in. If one of these is from our intended notification recipient, and the availability state is right (i.e., not in "Do Not Disturb" mode), bingo.

But the elements being sent over the stream from the server don't spontaneously get received, parsed, and dispatched; we can control when that happens from our script. This is the nub of the symbiosis between the element events and our procedural

routines, and it's name is `process()`.

Calling `process()` will check on the stream to see if any XML fragments have arrived and are waiting to be picked up. If there are any, steps 3 through 5, shown in Figure 7-5, are executed. The numeric value specified in the call to `process()` is the number of seconds to wait for incoming fragments if none are currently waiting to be picked up. Specifying no value (or zero) means that the method won't hang around if nothing has arrived. Specifying a value of 30 means that it will wait up to half a minute. We really want something in between, and it turns out that waiting for up to a second for fragments in a finite loop like this:

```
for i in range(5):
  con.process(1)
```

will allow for a slightly stuttered arrival of the `<presence/>` elements that are punted to the script as a result of the server-initiated probes.

Finishing up

We're just about done. The `<presence/>` elements that arrive and find their way to our callback are examined, and the CVS notification message is sent off if appropriate. Once the `process()` calls have finished, and, implicitly, the (potentially) multiple calls to `presenceCB`, there's nothing left to do. So we simply disconnect from the Jabber server, as before:

```
con.disconnect()
```

The script in its entirety

As that was probably a little disjointed, here's the presence-sensitive version of the CVS notification script in its entirety.

```
import Jabber, XMLStream
import sys
from string import split

cvsuser  = sys.argv[1]
message  = ''

def presenceCB(con, prs):

    type = prs.getType()
    parts = split(prs.getFrom(), '/')
    who = parts[0]

    if type == None: type = 'available'

    # subscription request:
    # - accept their subscription
    # - send request for subscription to their presence
    if type == 'subscribe':
        print "subscribe request from %s" % (who)
        con.send(Jabber.Presence(to=who, type='subscribed'))
        con.send(Jabber.Presence(to=who, type='subscribe'))

    # unsubscription request:
```

```
    # - accept their unsubscription
    # - send request for unsubscription to their presence
    elif type == 'unsubscribe':
        print "unsubscribe request from %s" % (who)
        con.send(Jabber.Presence(to=who, type='unsubscribed'))
        con.send(Jabber.Presence(to=who, type='unsubscribe'))

    elif type == 'subscribed':
        print "we are now subscribed to %s" % (who)

    elif type == 'unsubscribed':
        print "we are now unsubscribed to %s" % (who)

    elif type == 'available':
        print "%s is available (%s/%s)" % (who, prs.getShow(), prs.getStatus())
        if prs.getShow() != 'dnd' and who == cvsuser:
            con.send(Jabber.Message(cvsuser, message, subject="CVS Watch Alarm"))

    elif type == 'unavailable':
        print "%s is unavailable" % (who)

for line in sys.stdin.readlines(): message = message + line

con = Jabber.Connection(host=Server)

try:
    con.connect()
except XMLStream.error, e:
    print "Couldn't connect: %s" % e
    sys.exit(0)

con.auth(Username,Password,Resource)

con.setPresenceHandler(presenceCB)
con.requestRoster()
con.sendInitPresence()

for i in range(5):
  con.process(1)

con.disconnect()
```

Notes

[1] The process of adding a JID to the roster and making a subscription request is an atomic action in many clients. It is generally assumed that adding a JID to your roster means that you're going to want to know when that entity is available, so the action of adding a JID to the roster will often generate a presence subscription request automatically.

[2] This session ID is currently part of a new development within the `Net::Jabber` library and is currently not used for anything.

[3] This is to reduce the possibility of JID spoofing.

Dialup system watch | Up | Extending Messages, Groupchat, Components, and Event Models

Chapter 8. Extending Messages, Groupchat, Components, and Event Models

Table of Contents

At this stage, we've got a good idea of how scripts interact with Jabber, and how the core elements such as `<message/>` and `<presence/>` can be constructed and handled.

This Chapter builds upon what we've already seen in the previous one, and introduces new concepts. We build a nosey assistant that joins a conference room and alerts us to words and phrases that we want it to listen for. There are two popular conference protocols; the assistant recipe, a foray into the world of 'bots, takes a look at the original presence-based one. [1]

Programming within Jabber's event model is, as we've seen, straightforward. What happens when you want to meld other components with event models of their own? We look at a couple of typical scenarios where this melding needs to happen. The first is a homage to the Trojan Room Coffee Machine, [2] where we give life, or at least presence, to a coffee pot, using LEGO Mindstorms. The second is an RSS headline viewer that uses Tk, which has an event loop of its own.

We also look at extending messages, and build a mechanism that "punts" RSS headlines to clients who register with that mechanism. These headlines are carried using an extended message type.

In fact, the RSS punter mechanism is a *component*. The three recipes in the previous Chapter were Jabber *clients*, in that they connected to the Jabber network via the JSM (Jabber Session Manager) service. We look at the differences between programming a client and programming a component in this Chapter, and build a complete component that can be queried and interacted with the third of Jabber's building blocks—the `<iq/>` element.

Happy coding!

Notes

[1] The other uses the `jabber:iq:conference` namespace described in the section called *`jabber:iq:conference`* in Chapter 5a.

[2] Famously the first "device" to be reachable over the Web, at http://www.cl.cam.ac.uk/coffee/coffee.html.

Prev: Presence-sensitive CVS notification
Home
Up
Next: Keyword assistant

Keyword assistant

Many of the Jabber core and peripheral developers hang out in a conference room called "jdev" hosted by the *Conferencing* component on the Jabber server running at `jabber.org`. While a lot of useful information was to be gleaned from listening to what went on in "jdev", it wasn't possible for me to be there all the time. The conversations in "jdev" were logged to web pages, which I used to visit after the fact to try and catch up with things. This was mostly a hopeless task, so I decided to try and improve the situation slightly by looking for keywords and Uniform Resource Locators (URLs) in those logs.

This recipe is an automated version of what I did manually. It is a script that connects to a Jabber server, enters a conference room, and sits quite still, listening to the conversation and letting me know when certain words or phrases were uttered. I've given the script a bit of "intelligence" in that it can be interacted with, and told, while running, to watch for, or stop watching for, words and phrases of our choosing.

The script introduces us to programmatic interaction with the *Conferencing* component. Before looking at the script, let's have a brief overview of *Conferencing* in general.

Conferencing

The *Conferencing* component at `jabber.org` has the name `conference.jabber.org`. Details of the component instance configuration for such a *Conferencing* component can be found in the section called *Component Instance: conf* in Chapter 4, where we see that the component exists as a shared object library connected with the *library load* component connection method. This component provides general conferencing facilities, orientated around a conference *room* and conference *user* model.

A Jabber user can "enter" (or "join") a conference *room* and thereby becomes a conference *user* identified by a nickname that is chosen upon entering that room. Nicknames are generally used in conference rooms to provide a modicum of privacy—it is assumed that by default you don't want to let the other conference room members know your real JID.

The *Conferencing* component that is currently available for use with the Jabber 1.4.1 server supports two protocols for user and room interaction; a simple one that provides basic features, and a more complex one that provides the basic features plus facilities such as password-protected rooms, and room descriptions: [1]

Groupchat

> The *Groupchat* protocol is the simpler of the two and provides basic functions for entering and exiting conference rooms, and choosing nicknames.
>
> This *Groupchat* protocol has a nominal protocol version number of 1.0. It is known as the "presence-based" protocol, because the protocol is based upon `<presence/>` elements that are used for room entry, exit, and

nickname determination.

Conference

The *Conference* protocol offers more advanced features than the *Groupchat* protocol and makes use of two IQ namespaces—`jabber:iq:conference` and `jabber:iq:browse`. It has a nominal protocol version number of 1.4 which reflects the version of the Jabber server with which it is delivered. Sometimes this version number is referred to as 0.4, such as in the downloadable tarball, and in the value returned in response to a "version query" on the component itself, as shown in Example 8-1.

The version number isn't that important. The main thing to note is that the component that is called "`conference.so`" (see the reference to the shared object library in the section called *Component Connection Method* in Chapter 4) supports both the *Groupchat* protocol and the *Conference* protocol. If you come across a shared object library called `groupchat.so`, this is the original *Conferencing* component that was made available with Jabber server version 1.0. This library only supports the *Groupchat* protocol.

Example 8-1. Querying the *Conferencing* component's version

```
SEND: <iq type='get' to='conference.gnu.mine.nu'>
        <query xmlns='jabber:iq:version'/>
      </iq>

RECV: <iq to='dj@gnu.mine.nu/jarl' from='conference.gnu.mine.nu'
        type='result'>
        <query xmlns='jabber:iq:version'>
          <name>conference</name>
          <version>0.4</version>
          <os>Linux 2.4.2-2</os>
        </query>
      </iq>
```

In this recipe we'll be using the simpler "Groupchat" protocol. It's widely used, and easy to understand. Example 8-2 shows a typical element log from Groupchat-based activity. It shows a user, with the JID `qmacro@jabber.com`, entering a room called "cellar", hosted on the conference component at `conf.merlix.dyndns.org`, a room which currently has two other occupants who go by the nicknames "flash" and "roscoe". The elements are from *qmacro*'s perspective.

Example 8-2. The Groupchat protocol in action

qmacro tries to enter the room with the nickname "flash", and fails:

```
SEND: <presence to='cellar@conf.merlix.dyndns.org/flash'/>

RECV: <presence to='qmacro@jabber.com/jarltk'
                from='cellar@conf.merlix.dyndns.org/flash'
                type='error'>
        <error code='409'>Conflict</error>
      </presence>
```

He tries again, this time with a different nickname "deejay". Success:

```
SEND: <presence to='cellar@conf.merlix.dyndns.org/deejay'/>

RECV: <presence to='qmacro@jabber.com/jarltk'
              from='cellar@conf.merlix.dyndns.org/flash'/>

RECV: <presence to='qmacro@jabber.com/jarltk'
              from='cellar@conf.merlix.dyndns.org/roscoe'/>

RECV: <presence to='qmacro@jabber.com/jarltk'
              from='cellar@conf.merlix.dyndns.org/deejay'/>

RECV: <message to='qmacro@jabber.com/jarltk'
             type='groupchat' from='cellar@conf.merlix.dyndns.org'>
        <body>qmacro has become available</body>
      </message>
```

"roscoe" says hi, and *qmacro* waves back:

```
RECV: <message to='qmacro@jabber.com/jarltk'
             from='cellar@conf.merlix.dyndns.org/roscoe'
             type='groupchat' cnu=''>
        <body>hi qmacro</body>
      </message>

SEND: <message to='cellar@conf.merlix.dyndns.org' type='groupchat'>
        <body>/me waves to everyone</body>
      </message>
```

"flash" sends *qmacro* a private message:

```
RECV: <message to='qmacro@jabber.com/jarltk'
             from='cellar@conf.merlix.dyndns.org/flash'
             type='chat'>
        <body>psst - want to come out for a beer or two?</body>
        <thread>jarl1998911094</thread>
      </message>
```

"roscoe" leaves the room:

```
RECV: <presence to='qmacro@jabber.com/jarltk' type='unavailable'
              from='cellar@conf.merlix.dyndns.org/roscoe'/>

RECV: <message to='qmacro@jabber.com/jarltk' type='groupchat'
             from='cellar@conf.merlix.dyndns.org'>
        <body>roscoe has left</body>
      </message>
```

Let's take the elements in Example 8-2 bit by bit.

Failed attempt to enter room:

qmacro makes an attempt to enter the room using the *Groupchat* protocol. This is done by sending a directed `<presence/>` element to a particular JID that represents the room and the chosen nickname. This JID is constructed as follows:

```
[room name]@[conference component]/[nickname]
```

In this example, the conferencing component is identified with the hostname `conf.merlix.dyndns.org`. *qmacro*'s choice of nickname is "flash", determining that the JID, to which available presence should be sent, be:

```
cellar@conf.merlix.dyndns.org/flash
```

Thus the following element is sent:

```
SEND: <presence to='cellar@conf.merlix.dyndns.org/flash'/>
```

The conference component determines that there is already someone present in the room `cellar@conf.merlix.dyndns.org` with the nickname "flash", and so notifies *qmacro* by sending back a directed presence with an `<error/>` tag:

```
RECV: <presence to='qmacro@jabber.com/jarltk'
              from='cellar@conf.merlix.dyndns.org/flash'
              type='error'>
        <error code='409'>Conflict</error>
      </presence>
```

Note that the `<presence/>` element has the type "`error`", and comes `from` the artificial JID that we constructed in our room entry attempt. The element is addressed to our real JID, of course—`qmacro@jabber.com/jarltk`—as otherwise it wouldn't reach us.

The error code 409 and text "Conflict" tells *qmacro* that the nickname conflicted with one already in the room. This is a standard error code/text pair; Table 5-3 shows a complete set of code/text pairs.

At this stage, *qmacro* is not yet in the room.

Successful attempt to enter room:

qmacro tries again, this time with a different nickname "deejay": [2]

```
SEND: <presence to='cellar@conf.merlix.dyndns.org/deejay'/>
```

This time, there is no conflict—no other user is in the room "cellar" with that nickname—and the conference component registers the entry. It does this by sending *qmacro* the presence of all the room occupants, including

that of himself as "deejay":

```
RECV: <presence to='qmacro@jabber.com/jarltk'
              from='cellar@conf.merlix.dyndns.org/flash'/>

RECV: <presence to='qmacro@jabber.com/jarltk'
              from='cellar@conf.merlix.dyndns.org/roscoe'/>

RECV: <presence to='qmacro@jabber.com/jarltk'
              from='cellar@conf.merlix.dyndns.org/deejay'/>
```

These presence elements are also sent to the other room occupants.

Conference component-generated notification:

In addition to the presence elements sent for each room occupant, a general room-wide message noting that someone with the nickname "deejay" just entered the room, is sent out by the component as a `type='groupchat'` message to all the room occupants. Like all the other people, *qmacro* receives it:

```
RECV: <message to='qmacro@jabber.com/jarltk'
            type='groupchat'
            from='cellar@conf.merlix.dyndns.org'>
        <body>qmacro has become available</body>
      </message>
```

The text "has become available" used in the body of the message is taken directly from the *Action Notices* definitions, part of the *Conferencing* component instance configuration described in the section called *Custom Configuration* in Chapter 4. Note that the identity of the room itself is simply a generic version of the JID that the room occupants use to enter:

```
cellar@conf.merlix.dyndns.org
```

Room-wide chat:

Once the user with the nickname "roscoe" sees that *qmacro* has entered the room (he knows that *qmacro* sometimes goes by the nickname "deejay" and so recognises him), he sends his greetings, and *qmacro* waves back.

```
RECV: <message to='qmacro@jabber.com/jarltk'
             from='cellar@conf.merlix.dyndns.org/roscoe'
             type='groupchat' cnu=''>
        <body>hi qmacro</body>
      </message>

SEND: <message to='cellar@conf.merlix.dyndns.org'
        type='groupchat'>
        <body>/me waves to everyone</body>
      </message>
```

As with the notification message, each message is a "groupchat" type message. The one received appears to come from cellar@conf.merlix.dyndns.org/roscoe, which is the JID representing the user in the room with the nickname "roscoe". This way, the "roscoe"'s real JID is never sent to *qmacro*. [3] The one sent is addressed to the room's identity cellar@conf.merlix.dyndns.org, and contains a message that starts with "*/me*". This is simply a convention that is understood by clients that support conferencing, meant to represent an action and displayed thus:

```
* deejay waves to everyone
```

One-on-one chat:

The *Conferencing* component also supports a one-on-one chat mode, which is just like normal chat mode (where <message/>s with the type "chat" are exchanged) except that the routing goes through the component. The intended recipient of a conference-routed chat message is identified by his room JID. So in this example:

```
RECV: <message to='qmacro@jabber.com/jarltk'
        from='cellar@conf.merlix.dyndns.org/flash'
        type='chat'>
        <body>psst - want to come out for a beer or two?</body>
        <thread>jarl1998911094</thread>
      </message>
```

The user with the nickname "flash" actually addressed the chat message to the JID:

```
cellar@conf.merlix.dyndns.org/deejay
```

which arrived at the *Conferencing* component (because of the hostname "conf.merlix.dyndns.org" which caused the <message/> element to be routed there) which then looked up internally who "deejay" really was (qmacro@jabber.com/jarlrk) and sent it on. This way, the recipient of a conference-routed message never discovers the real JID of the sender. In all other ways, the actual <message/> element is like any other <message/> element—in this case it contains a message <body/> and a chat <thread/>. [4]

Leaving the room:

In the same way that room entrance is effected by sending an *available* presence (remember, a <presence/> element without an explicit type attribute is understood to represent type='available'), leaving a room is achieved by doing the opposite—sending an *unavailable* presence to the room, which is relayed to all the remaining room occupants:

```
RECV: <presence to='qmacro@jabber.com/jarltk' type='unavailable'
                from='cellar@conf.merlix.dyndns.org/roscoe'/>
```

The fact that "roscoe" left the room is conveyed by the unavailable presence packet. This is by and large for the benefit of each user's client, so that the room occupant list can be updated. The component also sends out a notification, in the same way as it sends a notification out when someone joins:

```
RECV: <message to='qmacro@jabber.com/jarltk' type='groupchat'
```

```
            from='cellar@conf.merlix.dyndns.org'>
      <body>roscoe has left</body>
    </message>
```

Like the join notification, the text for the leave notification ("has left") comes directly from the component instance configuration described in the section called *Custom Configuration* in Chapter 4.

The script's scope

We're going to write the script in Python, using the JabberPy library. What we want the script to do is this:

- Connect to a pre-determined Jabber server.
- Join a pre-determined conference room.
- Sit there quietly, listening to the conversation.
- Take simple commands from people to watch for, or stop watching for, particular words or phrases uttered in the room.
- Relay the context of those words or phrases to whoever requested them, if heard.

While we're going to set the identity of the Jabber server and the conference room into variables in the script, to keep things simple, we'll need to keep track of which users ask our assistant for what words and phrases. We'll use a hash ('dictionary' in Python terms) to do this. Having a look at what this hash will look like during the lifetime of this script will help us to visualize what we're trying to achieve. Example 8-3 shows what the contents of the hash might look like at any given time.

Example 8-3. Typical contents of the Keyword assistant's hash

```
{
  'dj@gnu.pipetree.com/home':             {
                                            'http:': 1,
                                            'ftp:': 1
                                          },

  'piers@jabber.org/work':                {
                                            'Perl': 1,
                                            'Java': 1,
                                            'SAP R/3': 1
                                          },

  'cellar@conf.merlix.dyndns.org/roscoe': {
                                            'Hazzard': 1
                                          }
}
```

We can see from the contents of the hash in Example 8-3 that three people have asked the script to look out for words and phrases. Two of those people—*dj* and *piers*—have interacted with the script directly, that is by sending the script a 'normal' (or 'chat') `<message/>`. The other person, with the conference nickname "roscoe", is in the "cellar" room and has sent the script a message routed through the *Conference* component in the same way that "flash" sent *qmacro* a

message in Example 8-2 earlier: the JID of the sender belongs to (has the hostname set to) the conference component. Technically there's nothing to distinguish the three JIDs here, it's just that we know from the name that `conf.merlix.dyndns.org` is the name that identifies such a component.

dj wants to be notified if any Web or FTP urls are mentioned. *piers* is interested in references to two of his favorite languages and his favorite ERP solution, and *roscoe*, well...

We said we'd give the script a little bit of intelligence. This was a reference to the ability for users to interact with the script while it runs, rather than have to give the script a static list of words and phrases in a configuration file. *dj*, *piers* and the user with the "roscoe" nickname have all done this, sending the script messages with simple keyword commands, such as:

dj: "*watch http:*"
script: "*ok, watching for http:*"

dj: "*watch gopher:*"
script: "*ok, watching for gopher:*"

dj: "*watch ftp:*"
script: "*ok, watching for ftp:*"

dj: "*ignore gopher:*"
script: "*ok, now ignoring gopher:*"

...

piers: "*list*"
script: "*watching for: Perl, Java, SAP R/3*"

...

roscoe: "*stop*"
script: "*ok, I've stopped watching*"

Step by step

Taking the script step by step, the first section is probably familiar if you've seen the previous Python-based scripts in the section called *CVS notification* in Chapter 7 and the section called *Presence-sensitive CVS notification* in Chapter 7.

```
import Jabber, XMLStream
from string import split, join, find
import sys
```

We bring in all the functions and libraries that we'll be needing. We'll be using the `find` function from the `string` library to help us with our searching.

Next, we declare our hash, or dictionary, which will hold a list of the words that the script is to look out for, for each person, as shown in Example 8-3.

```
keywords = {}
```

Maintaining the keyword hash

To maintain this hash, we need a couple of subroutines that will add and remove words from a person's individual list. These subroutines will be called when a "command" such as "*watch*" or "*ignore*" is recognised, in the callback subroutine that will handle incoming `<message/>` elements.

Here are those two subroutines:

```
def addword(jid, word):
    if not keywords.has_key(jid):
        keywords[jid] = {}
    keywords[jid][word] = 1

def delword(jid, word):
    if keywords.has_key(jid):
        if keywords[jid].has_key(word):
            del keywords[jid][word]
```

To each of them, we pass a string representation of the JID (in `jid`) of the correspondent giving the command, along with the word or phrase specified (in `word`). The hash has two levels—the first level is keyed by JID, the second by word or phrase. We use a hash, rather than an array, at the second level simply to make removal of words and phrases easier.

Message callback

Next, we define our callback to handle incoming `<message/>` elements.

```
def messageCB(con, msg):

    type = msg.getType()
    if type == None:
        type = 'normal'
```

As usual, we're expecting our message callback to be passed the Jabber *Connection* object (in `con`) and the message object itself (in `msg`). How this callback is to proceed is determined by the *type* of message received. We determine the type (taken from the `<message/>` element's `type` attribute) and store it in the `type` variable. Remember that if no `type` attribute is present, a type of "`normal`" is assumed. [5]

The two sorts of incoming messages we're expecting this script to receive are those conveying the room's conversation—in "`groupchat`" type messages—and those over which the commands such as "watch" and "ignore" are carried, which we expect in the form of "`normal`" or "`chat`" type messages.

Incoming commands

The first main section of the `messageCB` handler deals with the incoming commands:

```
# deal with interaction
if type == 'chat' or type == 'normal':
    jid = str(msg.getFrom())

    message = split(msg.getBody(), None, 1);
    reply = ""

    if message[0] == 'watch':
        addword(jid, message[1])
        reply = "ok, watching for " + message[1]

    if message[0] == 'ignore':
        delword(jid, message[1])
        reply = "ok, now ignoring " + message[1]

    if message[0] == 'list':
        if keywords.has_key(jid):
            reply = "watching for: " + join(keywords[jid].keys(), ", ")
        else:
            reply = "not watching for any keywords"

    if message[0] == 'stop':
        if keywords.has_key(jid):
            del keywords[jid]
            reply = "ok, I've stopped watching"

    if reply:
        con.send(msg.build_reply(reply))
```

Here's that section one chunk at a time.

If the `<message/>` element turns out to be of the type in which we're expecting a potential command, we want to determine the JID of the correspondent who sent that message. Calling the `getFrom()` method will return us a JID object. What we need is the string representation of that, which can be determined by calling the `str()` function on that JID object:

```
jid = str(msg.getFrom())
```

Then we grab the content of the message by calling the `getBody()` on the `msg` object, and split the whole thing on the first bit of whitespace. This should be enough for us to distinguish a command ("watch", "ignore", and so on) from the keywords. After the split, the first element (index 0) in the `message` array will be the command, and the second element (index 1) will be the word or phrase, if given. At this stage we also declare an empty reply.

```
message = split(msg.getBody(), None, 1);
reply = ""
```

Now it's time to determine whether what the script was sent made sense as a command:

```
if message[0] == 'watch':
    addword(jid, message[1])
    reply = "ok, watching for " + message[1]

if message[0] == 'ignore':
    delword(jid, message[1])
    reply = "ok, now ignoring " + message[1]

if message[0] == 'list':
    if keywords.has_key(jid):
        reply = "watching for: " + join(keywords[jid].keys(), ", ")
    else:
        reply = "not watching for any keywords"

if message[0] == 'stop':
    if keywords.has_key(jid):
        del keywords[jid]
        reply = "ok, I've stopped watching"
```

We go through a series of checks, taking appropriate action for our supported commands:

- `watch` (watch for a particular word or phrase)
- `ignore` (stop watching for a particular word or phrase)
- `list` (list the words and phrases currently being watched)
- `stop` (stop watching altogether—remove my list of words and phrases)

The `addword()` and `delword()` functions defined earlier are used here, as well as other simpler functions that list:

```
keywords[jid].keys()
```

or remove:

```
del keywords[jid]
```

the words and phrases for a particular JID.

If there was something recognisable for the script to do, we get it to reply appropriately:

```
if reply:
    con.send(msg.build_reply(reply))
```

The `build_reply()` function creates a reply out of a message object by setting the `to` to the value of the original `<message/>` element's `from` attribute, and preserving the element `type` attribute and `<thread/>` tag, if present. The `<body/>` of the reply object (which is, after all, just a `<message/>` element), is set to whatever is passed in the function call; in this case, it's the text in the `reply` variable.

Word and phrase scanning

Now that we've dealt with incoming commands, we just need another section in the message callback subroutine to scan for the words and phrases. The target texts for this scanning will be the snippets of room conversation, which arrive at the callback in the form of "*groupchat*" type `<message/>` elements.

```
    # scan room talk
    if type == 'groupchat':
        message = msg.getBody()
```

The `message` variable holds the string we need to scan, and it's just a case of checking for each of the words or phrases on behalf of each of the users that have asked:

```
        for jid in keywords.keys():
            for word in keywords[jid].keys():
                if find(message, word) >= 0:
                    con.send(Jabber.Message(jid, word + ": " + message))
```

If we get a hit, we construct a new `Message` object, passing the JID of the person for whom the string has matched (in the `jid` variable), and the notification consisting of the word or phrase that was found (in `word`) and the context in which it was found (the sentence uttered, in `message`). The `<message/>` so constructed, is then sent to that user. By default, the `Message` constructor specifies no `type` attribute, so that the user is sent a "normal" message.

Presence callback

Having dealt with the incoming `<message/>` elements that we're expecting, we turn our attention to `<presence/>` elements. Most of those we receive in this conference room context will be notifications of people entering and leaving the room that we're going to be in, as shown in Example 8-2. We want to perform housekeeping on our `keywords` hash so that the entries don't become stale. We also want to deal with the potential nickname conflict problem.

Nickname conflict

We want to check for the possibility of nickname conflict problems which may occur when we enter the room, and our chosen nickname is already taken.

Remembering that a conflict notification will look something like this:

```
<presence to='qmacro@jabber.com/jarltk'
        from='cellar@conf.merlix.dyndns.org/flash'
        type='error'>
  <error code='409'>Conflict</error>
</presence>
```

we test for the receipt of a `<presence/>` element so formed:

```
def presenceCB(con, prs):

    # deal with nickname conflict in room
    if str(prs.getFrom()) == roomjid and prs.getType() == 'error':
        prsnode = prs.asNode()
        error = prsnode.getTag('error')
        if error:
          if (error.getAttr('code') == '409'):
              print "cannot join room - conflicting nickname"
              con.disconnect()
              sys.exit(0)
```

The `<presence/>` element will appear to be sent from the JID that we constructed for our initial room entry negotiation (in the `roomjid` variable further down in the script), for example, in our case:

```
jdev@conference.jabber.org/kassist
```

We compare this value to the value of the incoming `<presence/>`'s `from` attribute, and also make sure that the `type` attribute is set to "`error`".

If it is, we want to extract the details from the `<error/>` tag that will be contained as a direct child of the `<presence/>`. The JabberPy library doesn't offer a direct high-level function to get at this tag from the `Presence` object (in `prs`), but we can strip away the presence object "mantle" and get at the underlying object, which is a neutral "node"—a Jabber element, or XML fragment, without any pre-conceived ideas of what it is (and therefore without any accompanying high-level methods such as `getBody()` or `setPriority()`. [6]

The `asNode()` method gives us what we need - a `Protocol` object representation of our `<presence/>` element. From this we can get to the `<error/>` tag and its contents. If we find that we do have a nickname conflict, we abort by disconnecting from the Jabber server and ending the script.

Keyword housekeeping

The general idea is that this script will run indefinitely and notify the users on a continuous basis. No presence subscription relationships are built (mostly to keep the script small and simple; you could adapt the mechanism from the recipe in the section called *Presence-sensitive CVS notification* in Chapter 7 if you wanted to make this script sensitive to presence) and so notifications will get queued up for the user if he is offline. [7] This makes a lot of sense for the most part; I still want to have the script send me notifications even if I'm offline. However, consider that the script could be sent a command, to watch for a keyword or phrase, from a user within the room. We would receive the command from a JID like this:

```
jdev@conference.jabber.org/nickname
```

This is a "transient" JID, in that it represents a user's presence in the *jdev* room for a particular session. If a word is spotted by the script hours or days later, there's a good chance that the user has left the room, making the JID invalid as a recipient—although the JID is *technically* valid and will reach the conferencing component, there

will be no real user JID that it is paired up with. Potentially worse, the room occupant's identity JID may be assigned to someone else at a later stage, if the original user left, and a new user entered choosing the same nick as the original user had chosen. the sidebar *Transient JIDs and non-existent JIDs* discusses the difference between a "transient" JID and a non-existent JID.

So as soon as we notice a user leave the room we're in, which will be indicated through a `<presence/>` element conveying that occupant's *unavailability*, we should remove any watched-for words and phrases from our hash:

```
# remove keyword list for groupchat correspondent
if prs.getType() == 'unavailable':
    jid = str(prs.getFrom())
    if keywords.has_key(jid):
        del keywords[jid]
```

As before, we obtain the string representation of the JID using the `str()` function on the JID object that represents the presence element's *sender*, obtained via the `getFrom()` method.

Transient JIDs and non-existent JIDs

What happens when you send a message to a conference room "transient" JID? Superficially, the same as when you send one to a *non-existent* JID. But there are some subtle differences.

A transient JID is one that reflects a user's alternative identity in the context of a component. In this case, the component is the *Conferencing* component. When you construct and send a message to a conference transient JID, it goes first to the conference component, because of the hostname in the JID identifies that component, for example:

`jdev@conference.jabber.org/qmacro`

The hostname `conference.jabber.org` is what the **jabberd** backbone uses to route the element. As mentioned earlier, the *Conferencing* component will relay a message to the real JID that belongs to the user that is currently in a room hosted by that component.

While the component itself is usually persistent, the room occupants (and so their transient JIDs) are not. This means that when a message is sent to the JID `jdev@conference.jabber.org/qmacro` and there is no room occupant in the "jdev" room with the nickname "qmacro", the message will still reach its *first* destination—the component—but be rejected at that stage, as shown in Example 8-4.

Although the rejection—the "Not Found" error— is the same as if a message had been sent to a JSM user that didn't exist, the difference is that the transient user always had the *potential* to exist, whereas the JSM user never did. [8]

Example 8-4. A message to a non-existent transient JID is rejected

```
SEND: <message to='jdev@conference.jabber.org/qmacro'>
```

```
        <body>Hello there</body>
      </message>

RECV: <message to='dj@gnu.mine.nu/jarl'
             from='jdev@conference.jabber.org/qmacro' type='error'>
        <body>Hello there</body>
        <error code='404'>Not Found</error>
      </message>
```

The main script

Ok. We've got our subroutines and callbacks set up. All that remains is for us to define our Jabber server and room information:

```
Server   = 'gnu.mine.nu'
Username = 'kassist'
Password = 'pass'
Resource = 'py'

Room     = 'jdev'
ConfServ = 'conference.jabber.org'
Nick     = 'kassist'
```

The `kassist` user can be set up simply by using the **reguser** script presented in the section called *User Registration Script* in Chapter 6:

```
$ ./reguser gnu.mine.nu username=kassist password=pass
[Attempt] (kassist) Successful registration
$
```

In the same way as previous recipes' scripts, a connection attempt, followed by an authentication attempt, is made:

```
con = Jabber.Connection(host=Server,debug=0,log=0)
try:
    con.connect()
except XMLStream.error, e:
    print "Couldn't connect: %s" % e
    sys.exit(0)
else:
    print "Connected"

if con.auth(Username,Password,Resource):
    print "Logged in as %s to server %s" % ( Username, Server )
else:
    print "problems authenticating: ", con.lastErr, con.lastErrCode
    sys.exit(1)
```

Then the message and presence callbacks `messageCB()` and `presenceCB()` are defined to our `Connection`

object `con`:

```
con.setMessageHandler(messageCB)
con.setPresenceHandler(presenceCB)
```

After sending initial presence, informing the JSM (and anyone that might be subscribed to `kassist`'s presence) of our availability:

```
con.send(Jabber.Presence())
```

we also construct—from the `Room`, `ConfServ`, and `Nick` variables—and send the `<presence/>` element for negotiating entry to the "jdev" room hosted by the *Conferencing* component at `conference.jabber.org`:

```
roomjid = Room + '@' + ConfServ + '/' + Nick
print "Joining " + Room
con.send(Jabber.Presence(to=roomjid))
```

`con.send()` will send a `<presence/>` element that looks like this:

```
SEND: <presence to='jdev@conference.jabber.org/kassist'/>
```

We're sending available presence to the room, to negotiate entry, but what about the initial presence? Why do we send that too, as there are no users that are likely to be subscribed to the `kassist` JID. Well, if no initial presence is sent, the JSM will merely store up any `<message/>` elements destined for `kassist`, as it will think the JID is offline. That won't help at all.

The processing loop

Once everything has been set up: initial presence has been sent and the room has been entered, we simply need to have the script sit back, wait for incoming packets, and handle them appropriately. For this, we simply call the `process()` repeatedly, waiting up to 5 seconds at a time for elements to arrive on the XML stream:

```
while(1):
    con.process(5)
```

The whole script

Here's the Keyword assistant script in its entirety.

```
import Jabber, XMLStream
from string import split, join, find
import sys

keywords = {}

def addword(jid, word):
```

```
    if not keywords.has_key(jid):
        keywords[jid] = {}
    keywords[jid][word] = 1

def delword(jid, word):
    if keywords.has_key(jid):
        if keywords[jid].has_key(word):
            del keywords[jid][word]

def messageCB(con, msg):

    type = msg.getType()
    if type == None:
        type = 'normal'

    # deal with interaction
    if type == 'chat' or type == 'normal':
        jid = str(msg.getFrom())

        message = split(msg.getBody(), None, 1);
        reply = ""

        if message[0] == 'watch':
            addword(jid, message[1])
            reply = "ok, watching for " + message[1]

        if message[0] == 'ignore':
            delword(jid, message[1])
            reply = "ok, now ignoring " + message[1]

        if message[0] == 'list':
            if keywords.has_key(jid):
                reply = "watching for: " + join(keywords[jid].keys(), ", ")
            else:
                reply = "not watching for any keywords"

        if message[0] == 'stop':
            if keywords.has_key(jid):
                del keywords[jid]
                reply = "ok, I've stopped watching"

        if reply:
            con.send(msg.build_reply(reply))


    # scan room talk
    if type == 'groupchat':
        message = msg.getBody()

        for jid in keywords.keys():
            for word in keywords[jid].keys():
```

```
                if find(message, word) >= 0:
                    con.send(Jabber.Message(jid, word + ": " + message))


def presenceCB(con, prs):

    # deal with nickname conflict in room
    if str(prs.getFrom()) == roomjid and prs.getType() == 'error':
        prsnode = prs.asNode()
        error = prsnode.getTag('error')
        if error:
          if (error.getAttr('code') == '409'):
              print "cannot join room - conflicting nickname"
              con.disconnect()
              sys.exit(0)

    # remove keyword list for groupchat correspondent
    if prs.getType() == 'unavailable':
        jid = str(prs.getFrom())
        if keywords.has_key(jid):
            del keywords[jid]

Server   = 'gnu.mine.nu'
Username = 'kassist'
Password = 'pass'
Resource = 'py'

Room     = 'jdev'
ConfServ = 'conference.jabber.org'
Nick     = 'kassist'

con = Jabber.Connection(host=Server,debug=0,log=0)
try:
    con.connect()
except XMLStream.error, e:
    print "Couldn't connect: %s" % e
    sys.exit(0)
else:
    print "Connected"

if con.auth(Username,Password,Resource):
    print "Logged in as %s to server %s" % ( Username, Server )
else:
    print "problems authenticating: ", con.lastErr, con.lastErrCode
    sys.exit(1)

con.setMessageHandler(messageCB)
con.setPresenceHandler(presenceCB)

con.send(Jabber.Presence())
```

```
roomjid = Room + '@' + ConfServ + '/' + Nick
print "Joining " + Room
con.send(Jabber.Presence(to=roomjid))

while(1):
    con.process(5)
```

Notes

[1] There is also a third protocol, called "Experimental iq:groupchat", which came inbetween the *Groupchat* and *Conference* protocols. This reflected an experimental move to adding features to the basic *Groupchat* protocol by the use of IQ elements, the contents of which were qualified by a namespace "`jabber:iq:groupchat`". This protocol has been dropped, and support for it only exists in certain versions of a couple of public Jabber clients - **WinJab** and **JIM**.

[2] There's no rule that says the nickname can't be the same as the user part of your JID, if you're not concerned with hiding your true identity :-)

[3] Ignore the `cnu` attribute. It's put there by the component, and should never make it out to the client endpoints. The attribute name is a short name for "conference user", and refers to the internal structure that represents a conference room occupant within the component.

[4] See the section called *The Message Element* in Chapter 5 for details on the `<message/>` element.

[5] See the section called *Message Attributes* in Chapter 5 for details of `<message/>` attributes.

[6] If this seems a little cryptic, just think of it like this: each of the `Presence`, `Message`, and `IQ` classes are merely superclasses of the base class `Protocol`, which represents elements generically.

[7] By the `mod_offline` module of the Jabber Session Manager (JSM).

[8] Of course, if the JID referred to a non-existent Jabber server, then the error returned wouldn't be a "*Not Found*" error 404, but an "*Unable to resolve hostname*" error 502.

Any coffee left?

LEGO Mindstorms. What a great reason to dig out that box of LEGO that you haven't touched since your childhood. What? You're still in your childhood? Even better! When I found out that LEGO was bringing out a programmable brick - the RCX, I went straight to Hamley's on Regent Street and handed over some well-earned cash for a large box containing the RCX—a yellow lump of plastic with buttons, an LCD, and connectors for sensors and motors—an infrared (IR) port plus an IR tower to connect to the serial port of your PC, and a battery compartment. [1] The box also contained a couple of motors, touch and light sensors, and various LEGO Technic parts. The RCX is pictured in Figure 8-1.

Figure 8-1. The LEGO Mindstorms RCX, or "programmable brick"

There are plenty of ways of interacting with the RCX, also known as the "programmable brick". The Mindstorms Robotics Invention System (RIS) set comes with Windows software with which you can build programs by moving blocks of logic around graphically on the screen and chaining them together ("turn sensor 1 on" ... "while this condition is true" ... "set motor 2 backwards" ... "turn motor 3 off" and so on). In addition, various efforts on the parts of talented individuals have come up with many different ways to program the RCX. O'Reilly's "The Unofficial Guide to LEGO Mindstorms Robots" tells you all you need to know, and gives you the big picture. What's important to know for this recipe is the following:

Cross compiling or realtime control

> There are two approaches to programming an RCX. One approach is to write a program on your PC, and download it to the RCX once complete, thence start and stop the program using the buttons on the RCX itself. The download is done over the IR connection.
>
> The other approach is to control the RCX directly from a program that you write *and execute* on your PC, sending control signals and receiving sensor values over the IR connection.
>
> Both approaches have their merits. How appropriate each one is boils down to one thing: connections. On the one hand, building autonomous machines that find their way around the kitchen to scare the cat and bring you a sandwich calls for the first approach, where, once you've downloaded the program to the RCX, you can dispense with any further connection with your PC because the entire logic is situated in your creation. On the other hand, if you want to build a physical extension to a larger system that, for example, has a connection to the Internet, the second approach is likely to be more fruitful, because you can essentially use the program that runs on your PC and talks to the RCX over the IR link as a conduit, a proxy, to other programs and systems that can be reached over the network.

We're going to use the second approach.

The nature of the LEGO RIS software

The RIS software that comes as standard centers around an ActiveX control. While there are plenty of ways to talk to the RCX without using this control (the book mentioned earlier describes many of these ways), the features offered by the control—`Spirit.ocx`— are fine for many a project. And with Perl's `Win32::OLE` module, we can interact with this ActiveX control without having to resort to Visual Basic.

What we're going to do

Everyone knows that one of the virtues of a programmer is *laziness*. We're going to extend this virtue (perhaps a little too far) and enhance it with a hacker's innate ability to combine two favorite pastimes—programming and playing with LEGO—to build contrived but fun devices.

We've a filter coffee machine that sits in my wife Sabine's study. My office is in the basement. Rather than traipse all the way upstairs to find out there's no coffee left in the pot, I decided to put my RCX to good use and get it to tell me, via Jabber, whether the pot had enough for another cup or not. That way, I didn't have to leave my keyboard to find out.

The idea is that we connect a light sensor to the RCX, and use that to "see" the coffee. The coffee pot is made of glass, and so allows light to pass through it. If there's coffee in the pot, no light passes through. The coffee effectively "gets in the way of" the light. There we have a simple binary switch. No (or a small amount of) light measured: there's coffee in the pot. Some (or a larger amount of) light: there's no coffee in the pot. We want to be able to push the coffee state to all interested parties in a way that their off the shelf Jabber clients can easily understand and display.

Figure 8-2 shows our LEGO Mindstorms device in action. The brick mounted on the gantry which extends to the glass coffee pot is our light sensor; a wire runs from it back to the connector on the RCX. Behind the RCX is the IR tower which is connected to Sabine's PC.

Figure 8-2. Our device "looking at" the coffee pot

Remembering that `<presence/>` elements are a simple way of broadcasting information about availability, *and* contain a `<status/>` tag to describe the detail or context of that availability in free-form text, [2] we have a perfect mechanism that's ready to be used. What's more, most, if not all, of the off the shelf Jabber client implementations will display the content of the `<status/>` tag next to the JID to which it applies, in the client user's roster. Figure 8-2 shows how the `<status/>` tag content is displayed—as a hovering "tooltip"—in WinJab.

Here's what we need to do:

Set up the RCX

We need to set the RCX up, with the light sensor, near enough to the coffee machine to take reliable and consistent light readings. Luckily the serial cable that comes with the Mindstorms set and connects to the IR tower is long enough to stretch from Sabine's computer over to within the infrared line-of-sight to the RCX.

Make the correct calibrations

There are bound to be differences in ambient light, sensitivity of the light sensor, and how strong you make your coffee. So we need a way of calibrating the setup, so that we can find the appropriate "pivot point" light reading value that lies between the two states of *coffee* and *no-coffee*.

Set up a connection to Jabber

We need a connection to a Jabber server, and a client account there. We can set one up using the **reguser** script from the section called *User Registration Script* in Chapter 6. We also need the script to honor presence subscriptions and unsubscriptions from users who want to be informed of the coffee state.

Set up a sensor poll/presence push loop

Once the RCX has been set up, the sensor calibrations taken, and the connection has been made, we need to constantly monitor the light sensor on the RCX at regular intervals. At each interval we determine the coffee state by comparing the value received from the sensor with the pivot point determined in the calibration step, and send any change in that state as a new availability `<presence/>` element containing an appropriate description in the `<status/>` tag.

The script

We're going to use Perl, and the `Net::Jabber` libraries, to build the script. Perl allows us a comfortable way to interact with an ActiveX control, through the `Win32::OLE` module. It's Let's dive straight in.

Setup

We first declare the packages we're going to use. As well as `Net::Jabber` and `Win32::OLE`, we're going to use `Getopt::Std` which affords us a comfortable way of accepting and parsing command line switches. We also want to use the `strict` pragma, which should keep us from making silly coding mistakes by not allowing undeclared variables and the like.

We specify the "Client" string on our usage declaration for the `Net::Jabber` package to specify what should be loaded. The package is a large and comprehensive set of modules, and only some of those are relevant for what we wish to do in our script—build and work with a Jabber *client* connection. Other module sets are pulled in by specifying "Component" or "Server".

```
use Net::Jabber qw(Client);
use Win32::OLE;
```

```
use Getopt::Std;
use strict;
```

We're going to allow the command line switches `-l` and `-s`. This is what the switches do:

No switches specified (or just the `-s` switch): calibration mode

> When we run the script for the first time, we need to perform the calibration, and read values from the sensor to determine a mid point value, above which signifies the presence of light, and therefore the absence of coffee, and below which signifies the absence of light, and therefore the presence of coffee.
>
> If we specify no switch, the script will start up automatically in calibration mode.
>
> Figure 8-3 shows the script in calibration mode. The values displayed, one each second, represent the values read from the light sensor. When the sensor was picking up lots of light, the values were 60. When I moved the sensor in front of some dark coffee, the values went down to around 45. Based upon this small test, I would set the pivot point value to 50.

`-l`: specify pivot value

> Once we've determined a pivot point value, we run the script "for real" and tell it this pivot value with the `-l` ("light pivot"):
>
> ```
> C:\temp> <userinput>perl coffee.pl -l 50</userinput>
> ```

`-s`: specify sensor

> The RCX, shown in Figure 8-1, has three connectors to which you can attach sensors. They're the three grey 2x2 pieces, with the legends "1", "2" and "3", near the top end of the brick. Any sensor can be attached to any of the three connectors. The script assumes you've attached the light sensor to the one marked "1", which internally is "0" (don't you just love computer science?). If you attach it to either of the other two, you can specify the connector using the `-s` ("sensor") with a value of 1 (for the middle connector) or 2 (for the rightmost connector), like this:
>
> ```
> C:\temp> <userinput>perl coffee.pl -l 50 -s 2</userinput>
> ```
>
> You can specify the `-s` switch when running in calibration and in normal modes.

Figure 8-3. Running the coffee script in calibration mode

Here's where the switches are defined with the Getopt::Std function:

```
my %opts;
```

```
getopt('ls', \%opts);
```

Next come a raft of constants, describing the script's Jabber relationship (the server it will connect to, and the user, password and resource it will connect with), the representation of the two states of "coffee" and "no coffee" (which will be used to determine the content of the `<status/>` tag sent along inside any `<presence/>` element emitted), the identification of the connector to which the light sensor is attached, and the polling granularity of the sensor poll / presence push loop described earlier (measured in seconds).

```
use constant SERVER   => "merlix.dyndns.org";
use constant PORT     => 5222;
use constant USERNAME => "coffee";
use constant PASSWORD => "pass";
use constant RESOURCE => "perlscript";

use constant NOCOFFEE => 0;
use constant COFFEE   => 1;

use constant SENSOR   => defined($opts{'s'}) ? $opts{'s'} : 0;
use constant GRAIN    => 1;
```

The last part of the script's setup deals with the coffee state:

```
my $current_status = -1;
my @status;
$status[NOCOFFEE] = 'xa/coffeepot is empty';
$status[COFFEE]   = '/coffee is available!';
```

We use a two-element array `@status` to represent the two possible coffee states. The value of each array element is a two-part string, each part separated by a slash ("/"). Each of these parts will be transmitted in a `<presence/>` element, where the first part (which is empty in the element representing the `COFFEE` state) will represent the presence `<show/>` value, and the second part will represent the presence `<status/>` value. Example 8-5 shows what a `<presence/>` element looks like when built up with values to represent the `NOCOFFEE` state.

Example 8-5. A `<presence/>` element representing the `NOCOFFEE` state

```
<presence>
  <show>xa</show>
  <status>coffeepot is empty</status>
</presence>
```

Most Jabber clients use different icons in the roster to represent different `<show/>` values. So we use different values here ("xa" for no coffee, and ""—blank, which represents "online" or "available"— for coffee) to trigger the icon change.

Initialization and calibration

Whenever we need to talk to the RCX, some initialization via the ActiveX control is required. That's the same whether we're going to calibrate or poll for values. The `setup_RCX()` function takes a single argument—the identification of which connector the light sensor is connected to—and performs the initialization, which is described later in the section called *`setup_RCX()`*. The function returns a handle on the `Win32::OLE` object that represents the ActiveX control, which in turn represents the RCX via the IR tower.

```
my $rcx = &setup_RCX(SENSOR);
```

If the `-l` flag is not specified, it means we're going to be running calibration. So we call the `calibrate()` function to do this for us. We pass the RCX handle (in `$rcx`) so the calibration can run properly.

```
# Either calibrate if no parameters given, or
# run with the parameter given as -l, which will
# be taken as the pivot between coffee and no-coffee.
&calibrate($rcx) unless defined($opts{'l'});
```

As with the `setup_RCX()` function, `calibrate()` is described later.

The calibration mode will be terminated by ending the script with Ctrl-C, so the next thing we come across is the call to the function `set_status()` which represents the first stage in the normal script mode; `set_status()` is used to determine the *initial* coffee status.

A value is retrieved by calling the ActiveX control's `Poll()` function. (Table 8-1 lists and described the ActiveX control's functions and properties used in this script.) We specify that we're after a "Sensor Value" (the 9 as the first argument) from the sensor attached to the connector indicated by the `SENSOR` constant.

```
# Determine initial status (will be either 0 or 1)
my $s = &set_status($rcx->Poll(9, SENSOR));
```

The value retrieved—it's going to be something along the lines of one of the values displayed when the script was run in calibration mode—is passed to `set_status()` which determines whether the value is above or below the pivot value, and whether the "new" status is different to the current one. If it is (and it this case, it will be, because in this first call, the value of `$current_status` is set to -1, which represents neither the COFFEE nor the NOCOFFEE state) that status will be returned, otherwise `undef` will be returned.

Table 8-1. RCX "*Spirit*" ActiveX control properties and functions used

Property/Function	Description

`Poll(SOURCE, NUMBER)`	Retrieves information from the RCX. In this script, the value for the *SOURCE* argument is always *9*, which represents a "Sensor Value", i.e. a value measured at a sensor, as opposed to an internal RCX variable or a timer, for example. The *NUMBER* argument represents the connector to which the sensor we want to read is attached.
`SetSensorMode(NUMBER, MODE [, SLOPE])`	The sort of value returned from a sensor is determined with this function. As with `Poll()` and `SetSensorType()`, *NUMBER* represents the sensor connector. With the *MODE* argument you can determine the sensor mode: *Raw (analogue) data* (0), *Boolean* (1), *Transitional* (2), *Periodic* (3), *Percentage* (4), *Celcius* (5), *Farenheit* (6), or *Angle* (7). The *SLOPE* argument qualifies the *Boolean* mode by specifying how True and False are to be determined.
`SetSensorType(NUMBER, TYPE)`	Use this function to specify the *type* of sensor that you're going to read values from. The *NUMBER* argument is the same as for the `Poll()` and represents the sensor connector. The *TYPE* argument represents the type of sensor that you want to set: *None* (0), *Switch* (1), *Temperature* (2), *Light* (3), or *Angle* (4).
`property:ComPortNo`	This is the serial port to which the ir tower is connected (1 = COM1, 2 = COM2, and so on).
`property:InitComm`	This is more like a function than a property. When invoked, the serial communication port is initialized in preparation for the IR connection to the RCX.

Set up connection to Jabber

At this stage, we're ready to connect to Jabber. The call to `setup_Jabber()` does this for us, returning a handle to the Jabber connection object that we store in `$jabber`. We will use this handle to send out `<presence/>` elements later in the script. The `$jabber` variable contains a reference to a `Net::Jabber::Client` object, and is the equivalent of the `con` variable used in the earlier Python scripts (the section called *CVS notification* in Chapter 7 and the section called *Presence-sensitive CVS notification* in Chapter 7) to hold the `Jabber.Connection` object, and `cb`, holding the `ConnectionBean` object in the earlier Java script (the section called *Dialup system watch* in Chapter 7).

```
my $jabber = &setup_Jabber(SERVER, PORT, USERNAME, PASSWORD, RESOURCE, $s);
```

As well as passing the constants needed for our client connection to the Jabber server, we pass the initial coffee

status, held in $s. We'll have a look at what the setup_Jabber() function does with this initial status a bit later when we get to the function's definition.

Sensor poll / presence push loop

The main loop starts here:

```
# Main loop: check Jabber and RCX
# ===============================
while (1) {
  defined($jabber->Process(GRAIN)) or
    die "The connection to the Jabber server was broken\n";
  my $s = &set_status($rcx->Poll(9, SENSOR));
  &set_presence($jabber, $s) if defined $s;
}
```

The while (1) is a bit of a giveaway. This script won't stop until you force it to stop, with a Ctrl-C. But that's essentially what we want. In the loop, we call the Process() method on our Jabber connection object in $jabber.

Process() is the equivalent of the JabberPy's process() method in the Python scripts. Process() waits around for up to the number of seconds specified as the single argument (or not at all if no argument is specified) for XML to appear on the stream connection from the Jabber server. If complete fragments do appear, callbacks, defined to the connection object, are called with the elements (<iq/>s, <message/>s and <presence/>s) that the fragments represent. This is in the same way as, for example, callbacks are called in Python scripts using the JabberPy libraries. The setup_Jabber(), coming next, is where the callback definition is made.

The Process() method returns undef if the connection to the Jabber server is terminated while waiting for XML. The undef value is dealt with appropriately by ending the script.

The GRAIN constant, set to one second in the script's setup section, is used to specify how long to wait. For the most part, we're not expecting to receive much incoming Jabber traffic. The occasional presence subscription (or unsubscription) request, perhaps (see later), but other than that, the only packets travelling over the connection to the Jabber server will be availability <presence/> packets representing coffee state changes, sent from the script. So normally, this delay of 1 second will take place. That's a comfortable polling interval for our light sensor too. So we do that within the same loop, happy in the knowledge that's it's most likely been a second since the last poll.

Calling the ActiveX control's Poll() again with the same arguments as before ("get a sensor value from the sensor attached to the SENSORth connector"), we pass the value to the set_status() to determine the coffee state. If the state was different from last time (if $s receives a value, and not undef), then we want to emit a <presence/> element to reflect that state. We achieve this by calling the set_presence() function, passing it the connection object and the state.

setup_Jabber()

Here we define the `setup_Jabber()` function, which is called once to set up the connection to the Jabber server and authenticate with a pre-defined user.

```
# Set up Jabber client connection, sending intial presence
# -------------------------------------------------------
sub setup_Jabber {
  my ($server, $port, $user, $pass, $resource, $initial_status) = @_;
  my $connection = new Net::Jabber::Client;

  # Connect
  my $status = $connection->Connect( hostname => $server,
                                     port     => $port );
  die "Cannot connect to Jabber server $server on port $port\n"
    unless $status;

  # Callbacks
  $connection->SetCallBacks( presence => \&InPresence );

  # Ident/Auth
  my @result = $connection->AuthSend( username => $user,
                                      password => $pass,
                                      resource => $resource );
  die "Ident/Auth failed: $result[0] - $result[1]\n"
    if $result[0] ne "ok";

  # Roster
  $connection->RosterGet();

  # Initial presence dependent upon initial status
  &set_presence($connection, $initial_status);

  return $connection;
}
```

First, we instantiate a new `Net::Jabber::Client` object. `Net::Jabber` distinguishes between client-based and component-based connections to Jabber; the component-based equivalent of this class is `Net::Jabber::Component`. The `connection()` method is passed arguments that specify the hostname and port of the Jabber server to connect to. It returns a zero status if the connection could not be made.

We can register handlers for Jabber elements that are received over the XML stream which is carried via the connection we've just made. Here we are only interested in incoming `<presence/>` elements—indeed, only those carrying presence subscription or unsubscription requests, as we'll see in the definition of the `InPresence()` function.

The single method `SetCallBacks()` does what the collective `Jabber.Connection` methods `setPresenceHandler()`, `setMessageHandler()`, and `setIqHandler()` do in a single call, taking a list of element types and subroutine references, in the form of a hash.

After registering our callback for `<presence/>` elements, it's time to authenticate, passing the username, password and resource that we defined in our list of constants at the start of the script. If the authentication is successful, the result of the call to the `AuthSend()` method is a single string with the value "ok". If not, that value is replaced with an error code and the descriptive text is available in a further string. (This is why we catch the results of a call in an array, called `@result`). The full list of Jabber error codes and texts are shown in Table 5-3.

Why `RosterGet()`? We're not subscribing to anyone, and we're not really interested in anything but the values we're polling from our brick. Yes, you guessed it—we want the script to receive and process presence subscription and unsubscription requests, but we aren't sent those by the JSM unless we've requested our roster beforehand. See the section called *Request for roster* in Chapter 7 for an explanation as to why.

Once we've kicked the JSM into sending any presence requests on to us, the job is almost done. The last thing to do in setting up our Jabber connection is to send our initial availability information. `setup_Jabber()` receives the initial coffee status as the last argument in the call (in `$initial_status`), which it now duly passes on to the function that sends a `<presence/>` element, `set_presence()`. Along with the initial coffee status, we also send the `$connection` object that represents the connection to the Jabber server that we've just established (and that is referred to outside of this function with the `$jabber` variable. This is so the `set_presence()` function can use the connection handle to send the element down the stream.

set_presence()

This function is used by `setup_Jabber()` to send the script's (and the coffee's) initial presence. It's also used within the main sensor poll / presence push loop to send further presence packets if the coffee's state changes.

```
sub set_presence {
  my ($connection, $s) = @_;
  my $presence = Net::Jabber::Presence->new();
  my ($show, $status) = split("/", $status[$s], 2);
  $presence->SetPresence( show   => $show,
                          status => $status );
  print $status, "\n";
  $connection->Send($presence);
}
```

On receipt of the Jabber connection object and the status, which will be 0 (`NOCOFFEE`) or 1 (`COFFEE`), `set_presence()` constructs a new `Net::Jabber::Presence` object. This object represents a `<presence/>` element, upon which we can make method calls to hone the element as we wish. `SetPresence()` is one of these methods, with which we can set values for each of the `<show/>` and `<status/>` tags. We retrieve the values for each of these tags by pulling the strings from the appropriate

member of the `@status` array, as described in the section called *Setup*.

We print the coffee's status (remember, this function is only called when the status *changes*, not every time the sensor is polled), and send our newly built `<presence/>` element off down the XML stream to the Jabber server by passing the presence object as an argument to the `Send()` method of the connection object in `$connection`. This works in the same way as the `send()` function in JabberPy, and the `send()` function in JabberBeans. And by the diffusive magic of the presence subscription model (see the section called *Presence Subscription* in Chapter 5), everyone who has subscribed to the script user's presence, and who is available, will receive the coffee status information.

Figure 8-4 shows the status information received in the WinJab client. The string sent in the `<status/>` tag is shown in the tooltip that appears when the mouse hovers over the "coffee" roster item.

Figure 8-4. Receiving information on the coffee's status in WinJab

`InPresence()`

Our presence handler, the callback subroutine `InPresence()`, exists for a single purpose: to honor requests for subscription and unsubscription to the script user's (and therefore the coffee's) presence. This callback is designed to work in the same way as the `presenceCB()` callback in the Python recipe described in the section called *Presence-sensitive CVS notification* in Chapter 7.

However, while the Python JabberPy library hands to the callbacks a `Jabber.Connection` object and the element to be handled, the Perl Net::Jabber library hands over a session ID and the element to be handled. The session ID is related to functionality for building Jabber servers, functionality that is not yet complete. We can and should ignore it for our script's purposes. What is important is the element to be handled, which appears as the second argument passed to the subroutine, which we collect into the `$presence` variable from `$_[1]`.

What is common between the two libraries is that the element that is passed to be handled, as the subject, so to speak, of the callback, is an instance of the class that the callback represents. In other words, here we have a callback to handle `<presence/>` elements, and the element received is an instance of the `Net::Jabber::Presence` class (just as the element received by a JabberPy presence callback is an instance of the `Jabber.Presence` class).

```
# Handle presence messages
# ------------------------
sub InPresence
{
  my $presence = $_[1];
  my $from = $presence->GetFrom();
  my $type = $presence->GetType();
```

```
  if ($type eq "subscribe") {
    print "Subscribe request ($from) ...\n";
    $jabber->Send($presence->Reply(type => 'subscribed'));
  }

  if ($type eq "unsubscribe") {
    print "Unsubscribe request ($from) ...\n";
    $jabber->Send($presence->Reply(type => 'unsubscribed'));
  }
}
```

With an object in `$presence`, we can get information from the element using data retrieval methods such as those used here: `GetFrom()` and `GetType()`, which extract the values from the `from` and `type` attributes of the `<presence/>` element respectively.

If the `<presence/>` element type represents a subscription request (`type='subscribe'`), we unquestioningly honor the request, by sending back an affirmative reply. The `Reply()` method of the presence object is one of a number of high-level functions that make it very comfortable to turn elements around and send them back. In this case, the method replaces the value of the `<presence/>`'s `to` attribute with the value of the `from` attribute, and preserves any `id` attribute. It also allows us to pass arguments as if we were calling the `SetPresence()` method described earlier. Rather than set the `<show/>` and `<status/>` tags as we did earlier in the `set_presence()` function, we merely set the element's `type` attribute to "subscribed" or "unsubscribed", depending on the request.

So with an incoming `<presence/>` element in `$presence` that looks like this:

```
<presence from='qmacro@jabber.org/office' type='subscribe'
            to='coffee@merlix.dyndns.org' id='21'>
```

calling the `Reply()` method would cause the element in $presence to change to this: [3]

```
<presence to='qmacro@jabber.org/office' type='subscribed' id='21'>
```

Note that the script doesn't ask for a subscription to the user's presence in return. The script isn't interested whether or not the people that have subscribed to its presence are available or not. It's sole job in life is to emit coffee availability.

setup_RCX()

This function is called once every time the script is started, and is required to initialize the RCX:

```
sub setup_RCX {
  my $sensor = shift;
  my $rcx = Win32::OLE->new('SPIRIT.SpiritCtrl.1');
```

```
  $Win32::OLE::Warn = 0;
  $rcx->{ComPortNo} = 1;
  $rcx->{InitComm};
  $rcx->SetSensorType($sensor, 3);
  $rcx->SetSensorMode($sensor, 2);
  return $rcx;
}
```

A `Win32::OLE` object representing the RCX's ActiveX control "Spirit" is instantiated. A `Win32::OLE` function is used to suppress warnings, and the RCX is initialized by setting the COM port to COM1 and initializing the serial communiations on that port. The sensor type and mode are set for the light sensor attached to the connector identified by the value passed into the `$sensor` variable. Table 5-1 shows us that sensor type 3 represents "Light", and sensor mode 2 specifies a "Transitional" measurement mode, the upshot of which is that the values returned on a poll are all within a certain restricted range, which makes it easier to decide on the coffee or no-coffee status.

We return the `Win32::OLE` RCX object to be used elsewhere in the script for calibration and polling.

calibrate()

The `calibrate()` function, called if the script is started without the `-l` switch, simply prints a message, waits for the user to press Enter, and then goes into a a gentle loop, emitting whatever value was polled from the light sensor, so the user can determine the pivot point:

```
sub calibrate {
  my $rcx = shift;

  print <<EOT;
Calibration mode.
Note the sensor values and decide on a 'pivot' value
above which 'no coffee' is signified and below which
'coffee' is signified.

End the calibration mode with Ctrl-C.

Press Enter to start calibration...
EOT

  <STDIN>;

  while (1) {
    print $rcx->Poll(9, SENSOR), " ";
    sleep 1;
  }

}
```

The output produced from this function can be seen in Figure 8-3.

set_status()

The `set_status()` function simply receives the latest light value as polled from the sensor and compares it with the pivot value. If the status so determined (in `$new_status`) is different from the *current* status (in `$current_status`) the current status is updated and returned. Otherwise `undef` is returned:

```
sub set_status {
  my $val = shift;

  my $new_status = $val < $opts{'l'} ? COFFEE : NOCOFFEE;

  if ($new_status != $current_status) {
    $current_status = $new_status;
    return $current_status;
  }
  else {
    return undef;
  }
}
```

If this function returns a status value, a new `<presence/>` element is generated and emitted by the script. Otherwise, there's no change ("the coffee's still there", or "there's still no coffee!") and nothing happens.

The whole script

Here's the script in its entirety.

```
use Net::Jabber qw(Client);
use Win32::OLE;
use Getopt::Std;
use strict;

my %opts;
getopt('ls', \%opts);

use constant SERVER   => "merlix.dyndns.org";
use constant PORT     => 5222;
use constant USERNAME => "coffee";
use constant PASSWORD => "pass";
use constant RESOURCE => "perlscript";

use constant NOCOFFEE => 0;
```

```
use constant COFFEE   => 1;

use constant SENSOR   => defined($opts{'s'}) ? $opts{'s'} : 0;
use constant GRAIN    => 1;

my $current_status = -1;
my @status;
$status[NOCOFFEE] = 'xa/coffeepot is empty';
$status[COFFEE]   = '/coffee is available!';

my $rcx = &setup_RCX(SENSOR);

# Either calibrate if no parameters given, or
# run with the parameter given as -l, which will
# be taken as the pivot between coffee and no-coffee.
&calibrate($rcx) unless defined($opts{'l'});

# Determine initial status (will be either 0 or 1)
my $s = &set_status($rcx->Poll(9, SENSOR));

my $jabber = &setup_Jabber(SERVER, PORT, USERNAME, PASSWORD, RESOURCE, $s);

# Main loop: check Jabber and RCX
# ===============================
while (1) {
  defined($jabber->Process(GRAIN)) or
    die "The connection to the Jabber server was broken\n";
  my $s = &set_status($rcx->Poll(9, SENSOR));
  &set_presence($jabber, $s) if defined $s;
}


# Set up Jabber client connection, sending intial presence
# --------------------------------------------------------
sub setup_Jabber {
  my ($server, $port, $user, $pass, $resource, $initial_status) = @_;
  my $connection = new Net::Jabber::Client;

  # Connect
  my $status = $connection->Connect( hostname => $server,
                                     port     => $port );
  die "Cannot connect to Jabber server $server on port $port\n"
    unless $status;

  # Callbacks
  $connection->SetCallBacks( presence => \&InPresence );

  # Ident/Auth
```

```
  my @result = $connection->AuthSend( username => $user,
                                      password => $pass,
                                      resource => $resource );
  die "Ident/Auth failed: $result[0] - $result[1]\n"
    if $result[0] ne "ok";

  # Roster
  $connection->RosterGet();

  # Initial presence dependent upon initial status
  &set_presence($connection, $initial_status);

  return $connection;
}


sub set_presence {
  my ($connection, $s) = @_;
  my $presence = Net::Jabber::Presence->new();
  my ($show, $status) = split("/", $status[$s], 2);
  $presence->SetPresence( show   => $show,
                          status => $status );
  print $status, "\n";
  $connection->Send($presence);
}


# Handle presence messages
# ------------------------
sub InPresence
{
  my $presence = $_[1];
  my $from = $presence->GetFrom();
  my $type = $presence->GetType();

  if ($type eq "subscribe") {
    print "Subscribe request ($from) ...\n";
    $jabber->Send($presence->Reply(type => 'subscribed'));
  }

  if ($type eq "unsubscribe") {
    print "Unsubscribe request ($from) ...\n";
    $jabber->Send($presence->Reply(type => 'unsubscribed'));
  }
}


sub setup_RCX {
```

```
  my $sensor = shift;
  my $rcx = Win32::OLE->new('SPIRIT.SpiritCtrl.1');
  $Win32::OLE::Warn = 0;
  $rcx->{ComPortNo} = 1;
  $rcx->{InitComm};
  $rcx->SetSensorType($sensor, 3);
  $rcx->SetSensorMode($sensor, 2);
  return $rcx;
}


sub calibrate {
  my $rcx = shift;

  print <<EOT;
Calibration mode.
Note the sensor values and decide on a 'pivot' value
above which 'no coffee' is signified and below which
'coffee' is signified.

End the calibration mode with Ctrl-C.

Press Enter to start calibration...
EOT

  <STDIN>;

  while (1) {
    print $rcx->Poll(9, SENSOR), " ";
    sleep 1;
  }

}


sub set_status {
  my $val = shift;

  my $new_status = $val < $opts{'l'} ? COFFEE : NOCOFFEE;

  if ($new_status != $current_status) {
    $current_status = $new_status;
    return $current_status;
  }
  else {
    return undef;
  }
}
```

Notes

[1] There's also a DC power socket for those of us without rechargeable batteries.

[2] See the section called *The Presence Element* in Chapter 5 for details on the `<presence/>` element

[3] Remember, the `from` attribute on elements originating from clients is set by the *server*, not by the *client*.

Prev | Home | Next
Keyword assistant | Up | RSS punter

RSS punter

RSS—*RDF Site Summary* [1] or alternatively *Really Simple Syndication*—is an XML format used for describing the content of a web site, where that site typically contains news items, diary entries, event information or generally anything that grows, item by item, over time. A classic application of RSS is to describe a news site such as JabberCentral. [2] JabberCentral's main page (see Figure 8-5) consists of a number of news items—in the "Recent News" section—about Jabber and the community (what else?). These items appear in reverse chronological order, and each one is fairly succinct, sharing a common set of properties:

Title

: Each item has a title ("JabberCon Update 11:45am - Aug 20").

Short description

: For each item, there's a short piece of text describing the content and context of the news story ("Jabbercon Update - Monday Morning").

Link to main story

: The short description should be enough to help the reader decide if he wants to read the whole item. If he does, there's a link ("Read More") to the news item itself.

Figure 8-5. JabberCentral's main page

It is this collection of item-level properties that are summarized in an RSS file. The formality of the XML structure makes it a straightforward matter for automating the retrieval of story summaries for inclusion in other sites (syndication), for the combination of these items with items from other similar sources (aggregation), and for simply checking to see if there is any new content (new items) since the last visit.

Example 8-6 shows what the RSS XML for the JabberCentral news items shown in Figure 8-5 looks like.

Example 8-6. RSS source for JabberCentral

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN"
     "http://my.netscape.com/publish/formats/rss-0.91.dtd">

<rss version="0.91">

  <channel>
```

```
    <title>JabberCentral</title>

    <description>
      JabberCentral is the premiere Jabber end-user news and support
      site. Many Jabber developers are actively involved at JabberCentral
      to provide fresh and authoritative information for users.
    </description>

    <language>en-us</language>
    <link>http://www.jabbercentral.com/</link>
    <copyright>Copyright 2000, Aspect Networks</copyright>

    <image>
      <url>http://jabbercentral.com/images/jc_button.gif</url>
      <title>JabberCentral</title>
      <link>http://www.jabbercentral.com/</link>
    </image>

    <item>
      <title>JabberCon Update 11:45am - Aug 20</title>
      <link>http://www.jabbercentral.com/news/view.php?news_id=998329970</link>
      <description>JabberCon Update - Monday Morning</description>
    </item>

    <item>
      <title>Jabcast Promises Secure Jabber Solutions</title>
      <link>http://www.jabbercentral.com/news/view.php?news_id=998061331</link>
      <description>
        Jabcast announces their intention to release security
        plugins with their line of products and services.
      </description>
    </item>

    ... (more items) ...

  </channel>

</rss>
```

The structure is very straightforward. Each RSS file describes a "channel" (`<channel/>`). Here's how a channel is defined:

Channel Information

The channel in this case is JabberCentral. The channel header information includes the channel's title (`<title/>`), short description (`<description/>`), main URL (`<link/>`) and so on.

Channel Image

Often RSS information is rendered into HTML to provide a concise "current index" summary of the channel it

describes. An image can be used in that summary rendering, and its definition is held in the `<image/>` section of the file.

Channel Items

The bulk of the RSS file content is made up of the individual `<item/>` sections, each of which reflect an item on the site that the channel represents. We can see in Example 8-6 that the first `<item/>` tag:

```
<item>
  <title>JabberCon Update 11:45am - Aug 20</title>
  <link>http://www.jabbercentral.com/news/view.php?news_id=998329970</link>
  <description>JabberCon Update - Monday Morning</description>
</item>
```

describes the most recent news item shown on JabberCentral's main page—"JabberCon Update 11:45am - Aug 20". Each of the news item properties are contained within that `<item/>` tag: the *title* (`<title/>`), *short description* (`<description/>`), and *link to main story* (`<link/>`).

Channel Interactive Feature

There is a possibility for each channel to describe an interactive feature on the site it represents; often this is a search engine which is fronted by a text input field and submit button. The interactive feature section of an RSS file is used to describe how that mechanism is to work (the name of the input field and the submit button, and the URL to invoke when the button is pressed, for example). This is so that HTML renderings of the site can include the feature otherwise only available on the original site.

This interactive feature definition is not shown in our RSS example.

RSS information lends itself very well to various methods of viewing. There are custom "headline viewer" clients available—focused applications that allow you to select from a vast array of RSS sources and have links to items displayed on your desktop (so yes, the personal newspaper—of sorts—is here!). There are also possibilities for having RSS items scroll by on your desktop control bar.

And then there's Jabber. As described in the section called *The Message Element* in Chapter 5, the Jabber `<message/>` element can represent something that looks suspiciously like an RSS item. The message type "headline" defines a message that carries news headline information. In this case, the `<message/>` element itself is usually embellished with an *extension*, qualified by the `jabber:x:oob` namespace, which is described in the section called *`jabber:x:oob`* in Chapter 5a. If the first news item from the JabberCentral site were to be carried in a headline message, Example 8-7 shows what the element would look like.

Example 8-7. A headline message carrying a JabberCentral news item

```
<message type='headline' to='dj@qmacro.dyndns.org'>
  <subject>JabberCon Update 11:45am - Aug 20</subject>
  <body>JabberCon Update - Monday Morning</body>
  <x xmlns='jabber:x:oob'>
    <url>http://www.jabbercentral.com/news/view.php?news_id=998329970</url>
    <desc>JabberCon Update - Monday Morning</desc>
  </x>
```

```
</message>
```

It's the extension, qualified by the `jabber:x:oob` namespace, that carries the crucial parts of the RSS item. And there are off the shelf clients such as WinJab and Jarl that can understand this extension and do something useful if a headline type message is received: the item content is displayed in a clickable list of lines, each one representing a single RSS item, akin to the headline viewer clients mentioned earlier.

Of course, we could simply punt RSS items to clients in non-headline type messages:

```
<message type='headline' to='dj@qmacro.dyndns.org'>
  <subject>JabberCon Update 11:45am - Aug 20</subject>
  <body>
    JabberCon Update - Monday Morning
    http://www.jabbercentral.com/news/view.php?news_id=998329970
  </body>
  </x>
</message>
```

where the complete item information is transmitted in a combination of the `<subject/>` and `<body/>` tags. This works too, but has the disadvantage of presenting no context within which the URL can be interpreted by the receiving clients, meaning all they can do with the message is display it as they would display any other message. The key is that we can send formalized *metadata* which increases the value of our message content enormously. Figure 8-7 shows Jarl displaying RSS-sourced news headlines.

Distributing RSS-sourced headlines over Jabber to standard Jabber clients is a great combination of off the shelf technologies. In fact, we'll see in the next section that it's not just standard Jabber clients that fit the bill; we'll write a Jabber-based headline viewer to show that not all Jabber clients are, nor should they be, made equal. But anyway, let's get to it!

The plan

We're going to write an RSS punter. A mechanism that checks pre-defined sources for new RSS items, and punts (or pushes) them to people who are interested in receiving them. For the sake of simplicity, we'll define the list of RSS sources in the script itself. See the section called *Further ideas* for ideas on how to develop this recipe further.

The script as *component*

Until now, the recipes we've written—such as the CVS Notification, the Dialup System Watch, and the Keyword Assistant, all in Chapter 7—have all existed as Jabber *clients*. That is, they've performed a service while connected to the Jabber server via the JSM (Jabber Session Manager). There's nothing wrong with this, indeed it's more than just fine to build Jabber-based mechanisms using a Jabber client stub connection; that way, your script, through its *identity*—the user JID— can avail itself of all the IM-related functions that the JSM offers—presence, storage and forwarding of messages, and so on. Perhaps even more interesting is that the mechanism needs only an account, a username and password, on a Jabber server to be part of the big connected picture. One can look at this sort of client-connected mechanism as an effective and low-cost entry to building the 'A's in our Jabber-connected A2A, A2P and P2A world of acronyms (or "acronym-qualified worlds"—you decide).

However, we know from Chapter 4 that there are other entities that connect to Jabber to provide services. These entities are called *components*. You can look upon components as being philosophically less "transient" than their client-connected brethren; and also closer to the Jabber server in terms of function and connection.

We know from the section called ***jabberd*** *and Components* in Chapter 4 that there are various ways to connect a component: *Library load*, *STDIO*, and *TCP sockets*. The first two dictate that the component be located on the same host as the **jabberd** backbone to which it connects. [3] The *TCP sockets* connection type, a method using a socket connection between the component and the **jabberd** backbone, over which streamed XML documents are exchanged (in the same way as they are exchanged in a client connection), allows us to run components on any host, and connect them to a Jabber server running on another host if we wish. Because of the connection flexibility, this approach is in many ways the most desirable. But it's not just the flexibility; because it abstracts the component away from the Jabber server core libraries, it leaves it up to us to decide how the component should be written. All the component has to do to get the Jabber server to cooperate is to establish the socket connection as described in the component instance configuration, perform an authenticating handshake, and correctly exchange XML stream headers.

Let's review how a TCP socket-based component connects. We'll base the review on what we're actually going to have to do to get our RSS punter up and running.

First, we have to tell the Jabber server that it is to expect an incoming socket connection attempt, which it is to *accept*. We do this by defining a component instance definition (or "description"—see the section called *Component instances* in Chapter 4) for our component. We include this definition in the main Jabber server configuration file, usually called `jabber.xml`. Example 8-8 shows a component instance definition for our RSS punter mechanism, known as `rss.qmacro.dyndns.org`.

Example 8-8. A component instance definition for our RSS punter mechanism

```
<service id='rss.qmacro.dyndns.org'>
  <accept>
    <ip>localhost</ip>
    <port>5999</port>
    <secret>secret</secret>
  </accept>
</service>
```

The name of the host on which the main Jabber server is running is `qmacro.dyndns.org`; it just so happens that our plan is to run the RSS punter component on the same host. We give it a unique name (**`rss`**`.qmacro.dyndns.org`) to enable the **jabberd** backbone, or hub, to distinguish it from other components and to be able to route elements to it. An alternative way of writing the component instance definition is shown in Example 8-9. The difference is simply in the way we specify the name. In Example 8-8 we specified an `id` in the `<service/>` tag with the value "`rss.qmacro.dyndns.org`". In the absence of any `<host/>` tag specification in the definition, this `id` value is used by the **jabberd** routing logic as the identification for the component when determining where elements addressed with that destination should be sent. In Example 8-9, we have an explicit `<host/>` specification which will be used instead, and we simply identify the service with an `id` attribute value of "`rss`". In this latter case, it doesn't really matter from an addressability point of view what we specify as the value for the `id` attribute.

Example 8-9. An alternative instance definition for our RSS punter mechanism

```
<service id='rss'>
  <host>rss.qmacro.dyndns.org</host>
  <accept>
    <ip>localhost</ip>
    <port>5999</port>
    <secret>secret</secret>
  </accept>
</service>
```

The instance definition contains all the information the Jabber server needs. We can tell from the `<accept/>` tag that this definition definition describes a *TCP sockets* connection. The socket connection detail is held in the `<ip/>` and `<port/>` tags. In this case, as we're going to run the RSS punter component on the same host as the Jabber server itself, we might as well kill two related birds with one stone by specifying *`localhost`* in the `<ip/>` tag: [4]

Performance

Connecting over the loopback device, as opposed to a real network interface, will give us a slight performance boost.

Security

Accepting only on the loopback device is a simple security measure that leaves one less port open to the world.

The `<secret/>` tag holds the secret that the connecting component must present in the authentication handshake.

Now let's look at the component's view of things. It will need to establish a socket connection to `127.0.0.1:5999`. Once that connection has been established, **jabberd** will be expecting it to announce itself by sending its XML document stream header. Example 8-10 shows a typical stream header that our component will need to send.

Example 8-10. The RSS component's stream header

```
SEND: <?xml version='1.0'?>
      <stream:stream xmlns='jabber:component:accept'
                     xmlns:stream='http://etherx.jabber.org/streams'
                     to='localhost'>
```

This matches the description of a Jabber XML stream header (also known as a stream "root" as it's the root tag of the XML document) from the section called *XML Streams* in Chapter 5. The namespace that is specified as the one qualifying the *content* of the stream is `jabber:component:accept`. This namespace "matches" the component connection method (*TCP sockets*) and the significant tag name in the component instance definition (*`<accept/>`*). [5] The value specified in the `to` attribute matches the hostname specified in the configuration's `<ip/>` tag.

After receiving a valid stream header, **jabberd** responds with a similar root to head up it's own XML document stream going in the opposite direction (from server to component). A typical response to the header in Example 8-10, received from the server by the component, is shown in Example 8-11.

Example 8-11. The server's stream header reply

```
RECV: <?xml version='1.0'?>
      <stream:stream xmlns:stream='http://etherx.jabber.org/streams'
                     id='3B8E3540'
                     xmlns='jabber:component:accept'
                     from='rss'>
```

The stream header sent in response shows that the server is confirming the component instance's identification as "`rss`". This reflects whatever was specified in the `<service/>` tag's `id` attribute of the component instance definition. Here, the value of the `id` attribute was "`rss`" as in Example 8-9.

It also contains an ID for the component instance itself: `id='3B8E3540'` in our example. The significance of this ID is as a random string shared between both connecting parties; the value is used in the next stage of the connection attempt—the authenticating handshake.

the section called *Digest Authentication Method* in Chapter 6 describes the digest authentication method of authentication for clients connecting to the JSM. This method uses a similar shared random string. On receipt of the server's stream header, the component takes the ID and and prepends it onto the secret that it must authenticate itself with. It then creates a NIST-SHA1 message digest (in a hexadecimal format) of that value:

SHA1_HEX(ID+SECRET)

Having created the digest, it sends it as the first XML fragment following the root, in a `<handshake/>` element:

```
SEND: <handshake id='1'>14d437033d7735f893d509c002194be1c69dc500</handshake>
```

On receipt of this authentication request, **jabberd** does the same thing: combines the ID value (after all, it knows what it is as it was **jabberd** that generated the value) with the value from the `<secret/>` tag in the component instance definition, and performs the same digest algorithm. If the digests match, the component is deemed to have authenticated itself correctly, and is send back an empty `<handshake/>` tag in conformation:

```
<handshake/>
```

The component may commence sending (and being sent) elements.

If the component sends an invalid handshake value—the secret may be wrong, or the digest may not have been calculated correctly—the connection is closed: **jabberd** sends a stream error and therewith ends the conversation:

```
RECV: <stream:error>Invalid handshake</stream:error>
```

Who gets punted what?

So, the definitions of the RSS sources are held within the script. But there's no reference to who might want to receive new items from which sources. We need a way for our component to accept requests, from users, that say things like:

"I'd like to have pointers to new items from the Slashdot site punted to me please"

or

"I'd *also* like pointers to new items from Jon Udell's site please"

or even

"Whoa, information overflow! Stop all my feeds!".

Let's take a leaf out of other components' books. There's a common theme that binds together components such as the JUD (Jabber User Directory), and the Transports to other IM systems such as Yahoo! and ICQ. This theme is *registration*. We've seen this before in the form of *user registration*, described in the section called *User Registration* in Chapter 6. This is the process of creating a new account with the JSM. Registration with a *service* such as the JUD, or an IM transport, however, follows a similar process. And both types of registration have one thing in common:

`jabber:iq:register`

The `jabber:iq:register` namespace is what's used in all cases to qualify the exchange of information during a registration process. [6] the section called *`jabber:iq:register`* in Chapter 5a describes the `jabber:iq:register` namespace. It shows us how a typical conversation between requester and responder takes place:

1. The client sends an IQ-get: "*How do I register?*"
2. The component sends an IQ-result: "*Here's how. Follow these instructions to fill in these fields.*"
3. The client then sends an IQ-set with values in the fields: "*Ok, here's my registration request.*"
4. To which the component responds, with another IQ-result: "*Looks fine. Your registration details have been stored.*"

It's clear that this sort of model will lend itself well to the process of allowing users to make requests to receive pointers to new items from RSS sources chosen from a list. Example 8-12 shows this conversational model in Jabber XML. There are many fields that can be used in a registration request; the description in the section called *`jabber:iq:register`* in Chapter 5a includes a few of these— `<name/>`, `<first/>`, `<last/>`, and `<email/>`—but there are more. We'll take the `<text/>` field to accept the name of an RSS source when a user wishes to register his interest to receive pointers to new items from that source. The conversational model is shown from the component's perspective.

Example 8-12. A registration conversation for RSS sources

"*How do I register?*"

```
RECV: <iq type='get' id='JCOM_3' to='rss.qmacro.dyndns.org'
        from='dj@qmacro.dyndns.org/basement'>
        <query xmlns='jabber:iq:register'/>
      </iq>
```

"*Here's how:*"

```
SEND: <iq id='JCOM_3' type='result' to='dj@qmacro.dyndns.org/basement'
        from='rss.qmacro.dyndns.org'>
        <query xmlns='jabber:iq:register'>
          <instructions>
            Choose an RSS source from: Slashdot, JonUdell[, ...]
          </instructions>
          <text/>
        </query>
     </iq>
```

"*Ok, here's my registration request:*"

```
RECV: <iq type='set' id='JCOM_5' to='rss.qmacro.dyndns.org'
        from='dj@qmacro.dyndns.org/basement'>
        <query xmlns='jabber:iq:register'>
          <text>Slashdot</text>
        </query>
      </iq>
```

"*Looks fine. Thanks.*"

```
SEND: <iq id='JCOM_5' type='result' to='dj@qmacro.dyndns.org/basement'
        from='rss.qmacro.dyndns.org'>
        <query xmlns='jabber:iq:register'>
          <text>Slashdot</text>
        </query>
      </iq>
```

(Time passes...)

"*Whoa. I want out!*"

```
RECV: <iq id='JCOM_11' to='rss.qmacro.dyndns.org' type='set'
        from='dj@qmacro.dyndns.org/basement'>
        <query xmlns='jabber:iq:register'>
          <remove/>
        </query>
      </iq>
```

"*Ok, you're out.*"

```
SEND: <iq id='JCOM_11' to='dj@qmacro.dyndns.org/basement' type='result'
        from='rss.qmacro.dyndns.org'>
        <query xmlns='jabber:iq:register'>
          <remove/>
        </query>
```

```
      </iq>
```

We'll use a lightweight persistent storage system for our user/source registrations—DBM—to keep the script fairly simple.

One more thing, before we leave this registration section. How will the users know that they can register? What's even more critical—how will they know that the RSS punter actually exists? the section called *Browsable Service Information* in Chapter 4 explains the way that services can be described, *announced* even, by the JSM. Most clients, having connected to the server and established a session with the JSM, make a request for a list of *agents* (old terminology) or *services* (new terminology) that are available on that Jabber server, like this:

```
SEND: <iq id="wjAgents" to="qmacro.dyndns.org" type="get">
        <query xmlns="jabber:iq:agents"/>
      </iq>
```

The response to the request, which looks like this:

```
RECV: <iq id='wjAgents' to='dj@qmacro.dyndns.org/basement'
        type='result' from='qmacro.dyndns.org'>
        <query xmlns='jabber:iq:agents'>
          <agent jid='conf.qmacro.dyndns.org'>
            <name>Public Chatrooms</name>
            <service>public</service>
            <groupchat/>
          </agent>
          <agent jid='users.jabber.org'>
            <name>Jabber User Directory</name>
            <service>jud</service>
            <search/>
            <register/>
          </agent>
        </query>
      </iq>
```

reflects the contents of the `<browse/>` section in the JSM configuration as shown in Example 8-13.

Example 8-13. The JSM configuration's `<browse/>` section

```
<browse>
  <conference type="public" jid="conf.qmacro.dyndns.org"
      name="Public Chatrooms"/>
  <service type="jud" jid="users.jabber.org" name="Jabber User Directory">
    <ns>jabber:iq:search</ns>
    <ns>jabber:iq:register</ns>
  </service>
</browse>
```

If we *add* a stanza for our RSS punter to the `<browse/>` section of the JSM configuration, that described our component, like this:

```
<service type="rss" jid="rss.qmacro.dyndns.org" name="RSS punter">
  <ns>jabber:iq:register</ns>
</service>
```

then we'll end up with an extra section in the `jabber:iq:agents` response from the server:

```
<agent jid='rss.qmacro.dyndns.org'>
  <name>RSS punter</name>
  <service>rss</service>
  <register/>
</agent>
```

The client-side effect of the agents response is exactly what we're looking for. Figure 8-6 shows WinJab's *Agents* menu displaying a summary of what it received in response to its `jabber:iq:agents` query. We can see that the stanza for our RSS punter was present in the `<browse/>` section and the component is faithfully displayed in the agent list, along with "Public Chatrooms" and "Jabber User Directory". In the main window of the screenshot we can see the "Supported Namespaces" list; it contains the namespace that we specified in our stanza. By specifying

```
<ns>jabber:iq:register</ns>
```

we're effectively telling the client that the component will support a registration conversation.

Figure 8-6. WinJab's "Agents" menu

But that's not all! We've advertised our RSS punter in the `<browse/>` section of the configuration for the JSM on the Jabber server running on `qmacro.dyndns.org`. That's why we got the information about the RSS punter agent when we connected as user *dj* to `qmacro.dyndns.org`—see the window's title bar in Figure 8-6. You may have noticed something odd about the definition of the other two agents, or services, in the `<browse/>` section earlier, or in the corresponding `jabber:iq:agents` IQ response. Let's take a look at this response again, this time with the extra detail about our component:

```
RECV: <iq id='wjAgents' to='dj@qmacro.dyndns.org/basement'
        type='result' from='qmacro.dyndns.org'>
        <query xmlns='jabber:iq:agents'>
          <agent jid='rss.qmacro.dyndns.org'>
            <name>RSS punter</name>
            <service>rss</service>
            <register/>
          </agent>
          <agent jid='conf.qmacro.dyndns.org'>
            <name>Public Chatrooms</name>
            <service>public</service>
```

```
        <groupchat/>
      </agent>
      <agent jid='users.jabber.org'>
        <name>Jabber User Directory</name>
        <service>jud</service>
        <search/>
        <register/>
      </agent>
    </query>
  </iq>
```

Imposter alert! While the `jid` attribute values for the RSS punter and Public Chatrooms agents show that they are components that are connected to the Jabber server we've just authenticated with (i.e. they both have JIDs in the `qmacro.dyndns.org` "space", and so are connected to the Jabber server running at `qmacro.dyndns.org`), the `jid` attribute for the Jabber User Directory points to a name in the `jabber.org` "space"! This is actually perfectly ok, and indeed is a side-effect of the power and foresight of Jabber's architectural design. If we connect a component, whether it's one we've built ourselves or one we've downloaded from the http://download.jabber.org site, we can give it an *internal* or an *external* identity when we describe it in the `jabber.xml` configuration.

Example 8-8 and Example 8-9 show two examples of an instance definition for our RSS punter component. Both specify potentially *external* identities. What this means is that if the hostname `rss.qmacro.dyndns.org` is a valid and resolvable hostname, the component can be reached from anywhere, not just from within the Jabber server to which it is connected. If the hostname wasn't resolvable by the outside world, by having a simple name such as `rss`, it could only be reached from the Jabber system to which it was connected.

So let's say `rss.qmacro.dyndns.org` *is* a valid and resolvable hostname. [7] If your client is connected to a Jabber server running on, say, `yourserver.org`, this is what would happen if you were to send, say, a registration request—an `<iq/>` element with a query qualified by the `jabber:iq:register` namespace—addressed to `rss.qmacro.dyndns.org`:

Packet reaches JSM on `yourserver.org`

> You send the IQ from your client, which is connected to your Jabber server's JSM. So this is where the packet first arrives.

Internal routing tables consulted

> `yourserver.org`'s **jabberd** looks in its list of internally registered destinations, and doesn't find `rss.qmacro.dyndns.org` in there.

Name resolved and routing established

> `yourserver.org`'s `dnsrv` (*Hostname Resolution*) service is used to resolve the `rss.qmacro.dyndns.org`'s address. Then, according to `dnsrv`'s instance configuration (specifically the
>
> ```
> <resend>s2s</resend>
> ```
>
> part—see the section called *Component Instance: dnsrv* in Chapter 4), the IQ is then routed on to the `s2s` (*Server to Server)* component.

Server to server connection established

> `yourserver.org` establishes a connection to `qmacro.dyndns.org` via `s2s` and sends the IQ across the connection.

Packet arrives at RSS punter component on `qmacro.dyndns.org`

> **jabberd** on `qmacro.dyndns.org` routes the packet correctly to `rss.qmacro.dyndns.org`.

So, what do we learn from this?

As exemplified by the reference to the JUD running at `users.jabber.org` that comes pre-defined in the standard `jabber.xml` with the 1.4.1. version of the Jabber server, you can specify references to services, components, *on other Jabber servers*. If you take this RSS punter script (when we finally get to it!), and run it against your own Jabber server, there's no reason why you can't share its services with your friends who run their own Jabber server.

The key is not the reference in the `<browse/>` section. The key is the resolvability of component names as hostnames, and the ability of Jabber servers to route packets to each other. The stanza in `<browse/>` just makes it easier *for the clients* to know about and automatically be able to interact with services in general. Even if a service offered by a public component that *wasn't* described in the result of a `jabber:iq:agents` query, it wouldn't stop you from reaching it. But you'd have to be good at writing XML by hand in your browser's raw/debug mode ;-)

The version query in Example 8-14 is a good example of this. Regardless of whether the conference component at `gnu.mine.nu` was listed in the `<browse/>` section of the `qmacro.dyndns.org`'s JSM, the user *dj* was able to make a version query by specifying the component's address, which was a valid and resolvable hostname, in the IQ-get's `to` attribute.

Polling the RSS sources

Now a quick word about the polling of the RSS sources. Remembering that the programming model with Jabber is usually event-based, and that we want to poll the RSS sources on a regular basis (although not every second!), we need some way of "interrupting" the process of checking for incoming elements and dispatching them to the callbacks, while we retrieve the RSS data and check for new items. There are many ways of achieving this; we're writing this component in Perl, so we could use the `alarm()` feature to set an alarm and have a subroutine invoked, to poll the RSS sources, when that alarm went off. This recipe uses the `Jabber::Connection` library, which negates the needs for an external alarm, as we will see when we come to the script.

Every time it's appropriate for us to poll the RSS sources, this is what we need to do, for each one:

1. Try and retrieve the source from the URL we have
2. Attempt to parse the source's XML
3. Go through the items, until we come across one we've seen before. The ones we go through until then are deemed to be new. (We need a special case the first time around, so that we don't flood everyone with every item of a source the first time we retrieve it.)
4. For new items, look in our registrations database for the users that have registered for that source, construct a headline message like the one shown in Example 8-7, and send it to those users.
5. Remember the first of the new items, so that we don't go beyond it next time.

Other things to bear in mind

There are differences between programming a component and programming a client. We're already aware of many of the major ones, described in the section called *The script as component*. There are, however, also more subtle differences that we need to bear in mind.

As we know, components, unlike clients, do not connect to the JSM. They connect as a *peer* of the JSM. Not only does this mean, as already stated, that they cannot partake of the IM feast of features made available by JSM's modules (see the section called *Component Connection Method* in Chapter 4 for a list of these modules), but also that they must do more for themselves. [8] When constructing an element as a client, we should not specify a `from` attribute before we send it; this is added "by the server"—more precisely, *by the JSM*—as it arrives. This is to prevent JID spoofing. Because a component does not connect through the JSM, no "from-stamping" takes place: the component itself must stamp the element with a `from` attribute.

The addressing of a component is also slightly different. Whereas client addresses reflect the fact that they're connected to the JSM, always having the form:

```
[user]@[hostname]/[resource]
```

(the resource being optional), the basic address form of a *component* is simply:

```
[hostname]
```

This doesn't mean to say that the address of a component cannot have a `[user]` or a `[resource]` part. It's just that *all* elements addressed to:

```
anything@[hostname]/anything
```

will be routed by **jabberd** to the component. This means our component can play multiple roles, and have many personalities. We'll see an example of this in the script, where we construct an "artificial" `[user]@[hostname]` address for the `from` attribute of a `<message/>` element, to convey information.

The component will respond to IQ queries in the `jabber:iq:register` namespace. It is, in fact, "customary" for components to respond to queries in a set of common IQ namespaces, although by no means mandatory. Taking the JUD and Conferencing components, for example, we see that they both respond to IQ queries in the `jabber:iq:time` and `jabber:iq:version` namespaces. Example 8-14 shows a typical version query on a Conferencing component. This responsiveness is simply to provide a basic level of administrative information. We want our component to conform to the customs, so we'll make sure it also responds to queries in these namespaces.

Example 8-14. A Conferencing component responds to a version query

```
SEND: <iq type='get' to='conf.gnu.mine.nu'>
        <query xmlns='jabber:iq:version'/>
      </iq>

RECV: <iq type='result' to='dj@qmacro.dyndns.org/study'
```

```
      from='conf.gnu.mine.nu'>
      <query xmlns='jabber:iq:version'>
        <name>conference</name>
        <version>0.4</version>
        <os>Linux 2.2.13</os>
      </query>
    </iq>
```

The script

It's time to let the dog see the rabbit. The component is written in Perl. You might want to refer to the script as a whole unit while reading through this section—you'll find it in the section called *The script in its entirety*. Ok. Let's go.

Setup

```
use strict;
use Jabber::Connection;
use Jabber::NodeFactory;
use Jabber::NS qw(:all);
use MLDBM 'DB_File';
use LWP::Simple;
use XML::RSS;
```

We're going to be using the `Jabber::Connection` library, so we declare that here; the library actually consists of three modules, and we're going to use them all. `Jabber::Connection` manages our connection to the server, and parses and dispatches incoming elements. `Jabber::NodeFactory` allows us to manipulate elements (generically called "nodes" by the module), and `Jabber::NS` provides us with a raft of constants that reflect namespaces and other common strings used in Jabber server, client and component programming.

We need a way of storing the registration information between invocations of the component script, and we'll use `MLDBM` for that. `MLDBM` is a really useful wrapper around the `DB_File` module. `DB_File` provides access to Berlekey DB database facilities using the `tie()` function. While you can't store references (i.e. complex data structures) via `DB_File`, you can when you use the `MLDBM` wrapper.

We will use the `LWP::Simple` module to grab the RSS sources by URL, and the `XML::RSS` module to parse those sources once retrieved.

```
my $NAME     = 'RSS Punter';
my $ID       = 'rss.qmacro.dyndns.org';
my $VERSION  = '0.1';
my $reg_file = 'registrations';
my %reg;

my %cache;

my %sources = (
```

```
  'jonudell' => 'http://udell.roninhouse.com/udell.rdf',
  'slashdot' => 'http://slashdot.org/slashdot.rdf',

  # etc ...

);
```

We start by declaring a few variables. We will see later in the script that `$NAME`, `$ID` and `$VERSION` will be used to reflect information in response to IQ queries. The variable `$reg_file` defines the name of the DB file to which we'll be `tie()`ing our registration hash `%reg`. `%cache` is our RSS item cache, to hold items that we've already seen, so we know when we've come to the end of the new items in a particular source.

We define our RSS sources in `%sources`. You may wish to define these differently, perhaps outside of the script. There are a couple of examples here; add your own favourite channels to taste.

```
tie (%reg, 'MLDBM', $reg_file) or die "Cannot tie to $reg_file: $!\n";
```

This magic line makes any data we store in the `%reg` hash persistent. It works by binding the operations on the hash (add, delete, and so on) to Berlekey DB operations, using the MLDBM module to stringify (and reconstruct) complex data structures so that they can be stored (and retrieved).

Connection

Right. We're ready to connect to the Jabber server as a component. Despite what's involved (described in the section called *The script as component*) it's very easy using a library such as `Jabber::Connection`:

```
my $c = new Jabber::Connection(
  server    => 'localhost:5999',
  localname => $ID,
  ns        => 'jabber:component:accept',
);
```

We construct a `Jabber::Connection` object, specifying the details of the connecion we wish to make. The `server` argument is used to specify the hostname, and optionally the port, of the Jabber server to which we wish to connect. In the case of a component, we must always specify the port (which is 5999 in our case, according to the component instance definition shown in Example 8-8). The same constructor can be used to create a client connection to Jabber, in which case a default port of 5222—the standard port for client connections—is assumed if none is explicitly specified. The `localname` argument is used to specify our name—the component's name—which in this case is `rss.qmacro.dyndns.org`. We specify the stream namespace with the `ns` argument. In the same way that a default port of 5222 is assumed if none is specified, a default stream namespace of `jabber:client` is assumed if no `ns` argument is specified. We wish to connect as a component using the *TCP sockets* connection method, so we must specify the appropriate namespace: `jabber:component:accept`.

This constructor call results in a stream header being prepared, one that looks like the one shown in Example 8-10.

The actual connection attempt, including the sending of the component's stream header, is done by calling the `connect()` method on the connection object in `$c`:

```
unless ($c->connect()) { die "oops: ".$c->lastError; }
```

This will return a true value if the connect succeeded (success is measured in whether the socket connection was established and whether the Jabber server sent a stream header in response). If it didn't succeed, we can retrieve details of what happened using the `lastError()` method.

We're connected. Before performing the authenticating handshake, we're going to do a bit of preparation:

```
$SIG{HUP} = $SIG{KILL} = $SIG{TERM} = $SIG{INT} = \&cleanup;
```

The idea is that the component will be run and only stopped in exceptional circumstances. If it is stopped, we want to clean things up before the script ends. Most importantly, we need to make sure our registration data is safe, but also we want to play nicely with the server and gracefully disconnect. This is done in the `cleanup()` function.

Preparation of the RSS event function and element handlers

```
debug("registering RSS beat");
$c->register_beat(1800, \&rss);
```

`Jabber::Connection` offers a simple way of having a function execute at regular intervals. It avoids the need for setting and re-setting `alarm()`s. Calling the `register_beat()` method takes two arguments. The first represents the interval, in seconds. The second is a reference to the function that should be invoked at each interval. Here, we're saying we want the `rss()` function called every half an hour.

```
debug("registering IQ handlers");
$c->register_handler('iq',\&iq_register);
$c->register_handler('iq',\&iq_version);
$c->register_handler('iq',\&iq_browse);
$c->register_handler('iq',\&iq_notimpl);
```

Most of the traffic relating to our component will be the headline messages emanating from it. However, we are expecting incoming IQ elements, particularly for registration in the `jabber:iq:register` namespace. We've also already mentioned that it's customary for components to honor basic "administrative" queries such as version checks. So the list of calls to the `register_handler()` method here reflects what we want to offer in terms of handling these IQ elements.

Whereas with `Net::Jabber`'s `SetCallBacks()` function, and with `JabberPy`'s `setIqHandler()` method we specify a single function to act as a handler for incoming `<iq/>` elements, we can specify as many handlers as we want for each element type with the `register_handler()` method in `Jabber::Connection`. The first argument refers to the element name (the name of the element's outermost tag), and the second refers to a function that will be called on receipt of an element of that name. Each of the handlers for a particular element will be called in the order they were registered. So when an `<iq/>` element is received over the XML stream, `Jabber::Connection` will dispatch it to `iq_register()`, then to `iq_version()`, then to `iq_browse()`, and then to `iq_notimpl()`. That is, unless one of those handler functions decides that the element has been handled once and for all, and that the dispatch processing for that element should stop there. In this case, that handler simply returns a special value (defined in `Jabber::NS`) and the dispatching stops for that element. The handlers can also cooperate,

in that the dispatcher will pass whatever one handler returns, into the next handler in the list, and so on, so that you can effectively share data across handler events for a particular element, building up a complex response as you go.

This "contextual response chain" model works in a similar way to how the `mod_auth_*` authentication modules in JSM work. Each one that wishes to express its interest in authenticating a user adds its "stamp" to the response to an IQ-get in the `jabber:iq:auth` namespace, before that response is returned to the client. [9]

Authenticating handshake and launch of main loop

Once we've set up our handlers, we're ready to make the authenticating handshake. This is simply a call to the `auth()` method:

```
$c->auth('secret');
```

It takes one or three arguments, depending on whether the authentication is for a client or a component. `Jabber::Connection` decides which authentication context is required by looking at the namespace specified (or defaulted) in the connection constructor call. As we specified the namespace `jabber:component:accept`, the `auth()` method is expecting a single argument which is the secret specified in the `<secret/>` tag of the component instance definition. `auth()` performs the message digest function and sends the `<handshake/>` element.

It's now appropriate for us to "launch" the component, with the `start()` method:

```
$c->start;
```

This is the equivalent of the `MainLoop()` method in Perl's Tk library, and is a method from which there's no exit. Calling `start()` causes the connection object to perform an endless loop, which internally calls a `process()` method on a regular basis, receiving, examining and dispatching elements received on the XML stream. It also starts and maintains the *heartbeat*, to which the `register_beat()` method is related. [10]

Handling registration requests

The first of the handlers defined for `<iq/>` elements is the `iq_register()` function. We put it first in the list as we consider receipt of `<iq/>` elements in the `jabber:iq:register` namespace to be the most common. We want this function to deal with the complete registration conversation. This means it must respond to IQ-get and IQ-set type requests.

```
sub iq_register {

  my $node = shift;

  debug("[iq_register]");
```

The primary piece of data that the dispatcher passes to a callback is the element to be handled. We receive this into the `$node` variable; it's a `Jabber::NodeFactory::Node` object. [11] The first thing we should do is make sure it's appropriate to continue inside this function, which is only designed to handle `jabber:iq:register` qualified queries. The namespace `jabber:iq:register` is represented with the constant `NS_REGISTER`, imported from the `Jabber::NS` module.

```
  return unless my $query = $node->getTag('', NS_REGISTER);
  debug("--> registration request");
```

The `getTag()` method takes up to two arguments. The first can be used to specify the name of the tag you want to get. A namespace can be specified in the second argument to narrow down the request; if an element were to contain two child tags with the same name, for example, the two `<x/>` elements in this `<message/>` element here:

```
<message to='dj@qmacro.dyndns.org' from='piers@jabber.org' id='2941'>
  <body>Let me know when you're ready to go</body>
  <x xmlns='jabber:x:event'><displayed/></x>
  <x xmlns='jabber:x:delay'
     from='dj@qmacro.dyndns.org'
     stamp='20010831T08:58:30'>Offline Storage</x>
</message>
```

We could distinguish one from the other by specifying either the `jabber:x:event` or the `jabber:x:delay` namespace.

Although normally the query tag within an `<iq/>` element has the name "query", we see from the section called *IQ Subelements* in Chapter 5 that it *could* be anything. So:

```
$node->getTag('', NS_REGISTER)
```

says "get a single child tag of our `<iq/>` node; doesn't matter what the name of the tag is, what's important is that it's qualified by the `jabber:iq:register` namespace."

If we call the `getTag()` function in scalar context, and there is more than one tag that matches, only the first one found will be returned. If we call it in list context, all the matching tags are returned. Assuming the call is successful, the variable `$query` then contains the `<query/>` tag and all its subtags. So if we received this in `$node`:

```
RECV: <iq type='set' id='JCOM_5' to='rss.qmacro.dyndns.org'
        from='dj@qmacro.dyndns.org/basement'>
        <query xmlns='jabber:iq:register'>
          <text>Slashdot</text>
        </query>
      </iq>
```

then `$query` would contain a `Jabber::NodeFactory::Node` object that represented this bit:

```
<query xmlns='jabber:iq:register'>
  <text>Slashdot</text>
</query>
```

If no child tag qualified by the `jabber:iq:register` namespace can be found, `iq_register()` returns, and the dispatcher calls the next handler in line—`iq_version()`. However, let's assume that we do have a registration IQ on our hands.

The function must handle both IQ-gets and IQ-sets. We first deal with a potential IQ-get:

```
  # Reg query
  if ($node->attr('type') eq IQ_GET) {
    $node = toFrom($node);
    $node->attr('type', IQ_RESULT);
    my $instructions = "Choose an RSS source from: ".join(", ", keys %sources);
    $query->insertTag('instructions')->data($instructions);
    $query->insertTag('text');
    $c->send($node);
  }
```

The `attr()` method called on a node will return the value of the node's attribute of the name specified as the first argument. We test to see if the `<iq/>`'s `type` attribute is "get" (IQ_GET). If it is, we need to return an IQ-result as shown in Example 8-12.

Rather than create a new element from scratch, to return in response, we simply "convert" the incoming element by making necessary changes to it, turn it around and sent it back out as our response. So the first thing we do is swap around the values for the `from` and `to` attributes in the `<iq/>` tag (in `$node`) by calling the `toFrom()` function (see the section called *Helper functions*), and setting the value for the `type` attribute to "`result`" by calling a 2-argument version of the `attr()` function, turning this:

```
<iq type='set' id='JCOM_5' to='rss.qmacro.dyndns.org'
  from='dj@qmacro.dyndns.org/basement'>
```

into this:

```
<iq type='result' id='JCOM_5' from='rss.qmacro.dyndns.org'
  to='dj@qmacro.dyndns.org/basement'>
```

Notice that we retain the `from` attribute; this is required as we're a component, and our response won't get stamped with one.

We must pass the instructions and an empty `<text/>` tag back in our response. We combine the names of the sources into a list, and insert an `<instructions/>` tag into our query node (in `$query`) containing the text. This is done with two method calls; the first to `insertTag()`, which returns a `Jabber::NodeFactory::Node` object that represents the newly inserted tag, and the second to `data()` which inserts (or retrieves) data into (or out of) a node. The line:

```
    $query->insertTag('instructions')->data($instructions);
```

could have been written as:

```
    my $instructions = $query->insertTag('instructions');
    $instructions->data($instructions);
```

The response, once constructed, and which now looks like this:

```
<iq type='result' id='JCOM_5' from='rss.qmacro.dyndns.org'
  to='dj@qmacro.dyndns.org/basement'>
  <query xmlns='jabber:iq:register'>
    <instructions>
      Choose an RSS source from: jonudell, slashdot [...]
    </instructions>
    <text/>
  </query>
</iq>
```

is sent using the `send()` method of the connection object.

If the query wasn't an IQ-get, then it might be an IQ-set:

```
  # Reg request
  if ($node->attr('type') eq IQ_SET) {

    # Strip JID to user@host
    my $jid = stripJID($node->attr('from'));

    $node = toFrom($node);
    my $source;
```

In this case, the user is requesting to receive new items for an RSS source he's specified in the `<text/>` field carried in the query part of the IQ-set. The user's JID can be found in the `from` attribute of the element, which we extract with the `attr()` method. But there's one thing we should do before using that JID as a key in storing that user's RSS source preferences. Look at what the JID was in the examples earlier:

```
dj@qmacro.dyndns.org/basement
```

It's a full-blown `[user]@[hostname]/[resource]` style JID. That's fine for using in a returning a response to an IQ request, but we need something less specific, something less "of the moment". The *resource* part of the JID reflects the client connection of the user at the time of registration request. In the future, when we have an RSS item to punt to him, he might be connected with a different resource. We want the RSS item to go the right place, so we use the more generic form of the JID—`[user]@[hostname]`—to store preferences and subsequently address our headline messages. We obtain the more generic form of the JID by calling the `stripJID()` function, described later.

After swapping the `from` and `to` values as before, we deal with the two different types of IQ-set requests—a request to receive a specific source, or a request to cancel registration (i.e. "unregistration"):

```
    # Could be an unregister
    if ($query->getTag('remove')) {
      delete $reg{$jid};
      $node->attr('type', IQ_RESULT);
    }
    # Otherwise it's a registration for a source
```

```
    elsif ($source = $query->getTag('text')->data
           and exists($sources{$source})) {
      my $element = $reg{$jid};
      $element->{$source} = 1;
      $reg{$jid} = $element;
      $node->attr('type', IQ_RESULT);
    }
```

Sending a `<remove/>` tag in an IQ-set registration context represents a request to *unregister*. So we honour that by removing all trace of the user's JID from our registration hash, and simply changing the `type` of the `<iq/>` element to "`result`".

Otherwise, we interpret the IQ-set as a request to 'subscribe' to the RSS source that they've specified in the `<text/>` tag. We extract that source's name into `$source`, check that it's valid, and add a reference to the user's list of sources in the registration hash `%reg`. Example 8-15 shows what the registration hash looks like.

Example 8-15. Typical contents of the registration hash

```
(
  'dj@qmacro.dyndns.org' => {
                              'slashdot' => 1
                            }
  'piers@jabber.org'     => {
                              'jonudell' => 1
                              'slashdot' => 1
                            }
  ...
)
```

In case you're wondering about the little value "dance" that's going on with the `$element` variable, it's because of a current restriction with `MLDBM`. Although it allows us to store complex structures via `DB_File`, we can't manipulate those structures directly, and have to do it via a "proxy" variable—`$element`.

Once this is done, we also mark the fact that the request was completed by setting the IQ element's `type` attribute to "`result`".

Finally, it's worth telling the requester that anything else sent just isn't cricket:

```
    else {
      $node->attr('type', IQ_ERROR);
      my $error = $node->insertTag('error');
      $error->attr('code', '405');
      $error->data('Not Allowed');
    }
```

That is, if we haven't understood what the IQ-set was—it wasn't a `<remove/>` request, nor was it a subscription to a source we recognise—we simply return it with an `<error/>` tag like this:

```
RECV: <iq type="set" id="jimAgentID657" to="rss.qmacro.dyndns.org">
        <query xmlns="jabber:iq:register">
          <text>banana</text>
        </query>
      </iq>

SEND: <iq id='jimAgentID657' type='error' from='rss.qmacro.dyndns.org'
          to='dj@qmacro.dyndns.org/basement'>
        <query xmlns='jabber:iq:register'>
          <text>banana</text>
        </query>
        <error code='405'>Not Allowed</error>
      </iq>
```

The `<iq/>` type is set to "`error`" to draw the client's attention to the `<error/>` tag. Sending an element back in error is a great example of where reusing an incoming element to build the outgoing response works very well; we don't have any work to reproduce what was in error, as it's already contained in what we're turning around.

In all of the IQ-set cases, we want to send something back, so we do this now:

```
    $c->send($node);

  }

  return r_HANDLED;

}
```

Note that we also return a special value `r_HANDLED`. The fact that we've got this far means that we received an IQ element, it was a registration-related element, and we've handled it, so there's no point in the other callbacks registered to handle IQ elements to get a look in. So we tell the dispatcher to stop the invocation chain for the element we've just processed.

Handling version requests

Now we've seen the `iq_register()` function, the function to handle `jabber:iq:version` queries looks pretty straightforward:

```
sub iq_version {

  my $node = shift;
  debug("[iq_version]");

  return unless my $query = $node->getTag('', NS_VERSION)
            and $node->attr('type', IQ_GET);

  debug("--> version request");
```

```
  $node = toFrom($node);
  $node->attr('code', IQ_RESULT);
  $query->insertTag('name')->data($NAME);
  $query->insertTag('version')->data($VERSION);
  $query->insertTag('os')->data(`uname -sr`);
  $c->send($node);

  return r_HANDLED;

}
```

As we check for whether the element is appropriate to handle in `iq_register()`, so we do here, this time looking for an IQ-get with a query child tag qualified by the `NS_VERSION` (`jabber:iq:version`) namespace, which we snag into `$query`.

Setting the `<iq/>`'s type to "`result`" and flipping the addresses, we then just have to add `<name/>`, `<version/>`, and `<os/>`, tags to the query child with appropriate values, to end up with a response like the one shown in Example 8-14.

If we've done this, we deem the IQ to have been handled, and return the special `r_HANDLED` value to stop the dispatching going any further for this element.

Handling browse requests

Next in line to handle the incoming `<iq/>` element is the `iq_browse()` function. Of course, if we've already handled the element, `iq_browse()` won't even get a shot at responding. But if it did, it would proceed along similar lines to the `iq_version()` function:

```
sub iq_browse {

  my $node = shift;
  debug("[iq_browse]");

  return unless my $query = $node->getTag('', NS_BROWSE)
            and $node->attr('type', IQ_GET);

  debug("--> browse request");

  $node = toFrom($node);
  $node->attr('type', IQ_RESULT);
  my $rss = $query->insertTag('service');
  $rss->attr('type', 'rss');
  $rss->attr('jid', $ID);
  $rss->attr('name', $NAME);
  $rss->insertTag('ns')->data(NS_REGISTER);
  $c->send($node);

  return r_HANDLED;
```

```
}
```

The only real difference is the fact that we want this function to handle IQ-gets in the `jabber:iq:browse` namespace, and return a browse result. We'll be looking at browsing in more detail in the section called *Browsing LDAP* in Chapter 9. For now, we'll content ourselves in returning a top-level browse result that reflects what might be returned if a similar browse request were made of the JSM, as described in the section called *`jabber:iq:browse`* in Chapter 5a. What `iq_browse()` will return is shown in Example 8-16.

Example 8-16. RSS punter responds to `jabber:iq:browse` requests via `iq_browse()`

```
RECV: <iq type="get" id="browser_JCOM_2" to="rss.qmacro.dyndns.org">
        <query xmlns="jabber:iq:browse"/>
      </iq>

SEND: <iq id='browser_JCOM_2' type='result'
        to='dj@qmacro.dyndns.org/winjab'
        from='rss.qmacro.dyndns.org'>
        <query xmlns='jabber:iq:browse'>
          <service jid='rss.qmacro.dyndns.org'
               type='rss'
               name='RSS Punter'>
            <ns>jabber:iq:register</ns>
          </service>
        </query>
      </iq>
```

Other requests

Any other requests? If you hum it, I'll play it. But seriously, there are untold IQ elements that could be sent to the component. While it would be *possible* just to ignore them, we ought to do the done thing and at least respond with a "not supported" type of response. So we have `iq_notimpl()` as a catch-all. If the dispatcher manages to make its way through to here, we know that the `iq/>` element is not anything we recognise as wanting to respond to.

So let's just tell the requester that what they're asking for is not implemented:

```
sub iq_notimpl {

  my $node = shift;
  $node = toFrom($node);
  $node->attr('type', IQ_ERROR);
  my $error = $node->insertTag('error');
  $error->attr('code', '501');
  $error->data('Not Implemented');
  $c->send($node);

  return r_HANDLED;

}
```

As you can see, all this does is set the `<iq/>` type to "`error`", switches the `from` and `to`, adds an `<error/>` tag that looks like this:

```
<error code='501'>Not Implemented</error>
```

and throws the modified element back to the requester.

The RSS mechanism

Now we've set up the functions to handle the incoming queries, all that's left is for us to define what happens every time the heartbeat in the `Jabber::Connection` loop ticks past the 30 minute mark. We registered this `rss()` function with the `register_beat()` method earlier in the script:

```
sub rss {

  debug("[rss]");

  # Create NodeFactory
  my $nf = new Jabber::NodeFactory;
```

While in the IQ handlers we turned the incoming request elements around into responses, we'll actually be building elements, headline messages to be precise, from scratch here. This is why we need an instance of the `Jabber::NodeFactory`.

```
  # Go through each of the RSS sources
  foreach my $source (keys %sources) {

    # Retrieve attempt
    my $data = get($sources{$source});

    # Didn't get it? Next one
    unless (defined($data)) {
      debug("cannot retrieve $source");
      next;
    }

    # Parse the RSS
    my $rss = XML::RSS->new();
    eval { $rss->parse($data) };

    if ($@) {
      debug("problems parsing $source");
      next;
    }
```

The procedure in this function reflects what we described in the section called *Polling the RSS sources* earlier. Each time `rss()` is called, it goes through each of the sources defined in the list (`%sources`), and tries to retrieve it, with

`get()`, a function from the `LWP::Simple` library, and parse it, with an instance of `XML::RSS`. [12] Because `XML::RSS` uses `XML::Parser`, which `die()`s if it encounters invalid XML, we wrap the call to the `parse()` method in `eval`.

```
my @items = @{$rss->{items}};

# Check new items
debug("$source: looking for new items");
foreach my $item (@items) {

  # Stop checking if we get to items already seen
  last if exists $cache{$source} and $cache{$source} eq $item->{link};

  debug("$source: new item $item->{title}");
```

Pulling the items from the RSS source into `@items`, we look through them, but stop looking if we come across one that we've seen previously (and stored in the `%cache`).

If we do have a new item to send out, we create a headline message containing the item's details:

```
  # Create a headline message
  my $msg = $nf->newNode('message');

  $msg->attr('type', 'headline');
  $msg->attr('from', join('@', $source, $ID));
  $msg->insertTag('subject')->data($item->{title});
  $msg->insertTag('body')->data($item->{description});

  my $xoob = $msg->insertTag('x', NS_XOOB);
  $xoob->insertTag('url')->data($item->{link});
  $xoob->insertTag('desc')->data($item->{description});
```

We use our nodefactory in `$nf` to create a new empty `<message/>` element, with the `newNode()` method. We build up this element into a full blown headline message with a `jabber:x:oob` qualified `<x/>` extension containing the RSS item information. We can see here that the call `insertTag()` used here has two arguments. The second is used to specify an optional namespace with which the new node (or tag) will be qualified. What this call creates, in `$xoob`, is a `Jabber::NodeFactory::Node` object that looks like this:

```
<x xmlns='jabber:x:oob'/>
```

This is then embellished with the usual `<url/>` and `<desc/>` tags. What's still missing is the address information. We've specified the `from`; indeed taking a departure from the values we've specified for the `from` in our IQ responses, here we specify something slightly different, with `join('@', $source, $ID)`— a `[user]@[hostname]` style address. For the *slashdot* source, this would be:

```
slashdot@rss.qmacro.dyndns.org
```

This is mostly because it conveys more information than just the component name `rss.qmacro.dyndns.org`

would do. While the component's address would not normally be seen by a client user in the context of the IQ responses, many Jabber clients that support the headline message type show the message sender in the headline list display. You can see this in Figure 8-7, where the *From* column in the headline list shows clearly the RSS source where the item originated. This has a little bit of future for the component built in, too. If we wanted to extend the component for more interaction with the clients, we could have the client send a message to the `[RSS source]@[componentname]` JID and on receipt, the component would immediately have context information on which source the message was about, without the client user having to do anything other than specify a JID.

Figure 8-7. Jarl's headline display window

Now we've built our headline message, which looks like the one in Example 8-7, we can fire it off to each of the users who have registered for that RSS source:

```
      # Deliver to all that want it
      foreach my $jid (keys %reg) {

        my $registration = $reg{$jid};

        if (exists($registration->{$source})) {
          $msg->attr('to', $jid);
          debug("punting to $jid");
          $c->send($msg);
        }

      }
```

The first time we encounter an RSS source, we won't have any record of a "last seen" item in the `%cache`. So we avoid flooding people with all the items of a new RSS source by jumping out of the item loop if there's no cache info:

```
      # Prevent all items counted as new the
      # first time around
      last unless exists($cache{$source});

    }
```

Finally, we make a mark in the cache for the "latest" item we've just encountered, ready for next time:

```
    # Remember the latest new item
    $cache{$source} = $items[0]->{link};

  }

}
```

The `cleanup()` function

The `cleanup()` is called if an attempt is made to shut the script down; it `untie()`s the registration hash, ensuring no data is lost, and disconnects from the Jabber server:

```
sub cleanup {

  debug("cleaning up");
  untie %reg;
  $c->disconnect;
  exit;

}
```

Helper functions

Any script over a certain small size is bound to have helper functions; our RSS punter is no exception. Here we have the function to switch the `from` and `to` attribute values of a node (`toFrom()`), the function to remove the resource from a JID (`stripJID()`), and something not much better than a debugging-style `print` statement :-)

```
sub toFrom {
  my $node = shift;
  my $to = $node->attr('to');
  $node->attr('to', $node->attr('from'));
  $node->attr('from', $to);
  return $node;
}


sub stripJID {

  my $JID = shift;
  $JID =~ s|/.*$||;
  return $JID;

}


sub debug {

  print STDERR "debug: ", @_, "\n";

}
```

Further ideas

Ok, we're done! Of course, there's only so much that can be included in a demonstration script. There's plenty of scope for improvement, even if you don't count re-writing it all from scratch. You'll probably want to store the registrations in a SQL database, or alternatively using the Jabber server's own XDB component. More importantly, a static list of RSS sources is rather restrictive. How about allowing the user to register their own URLs? Or building an administrative mode which accepts a special IQ from certain JIDs, with which the RSS source list can be maintained?

The browsing response function would be an ideal candidate for extension— how about allowing a next level of browsing that would return browse items that reflect the specific user's RSS source registrations? And how could we use the power of addressing the component to include the RSS source, to extend the interactive facilities?

The script in its entirety

Here's the script in its entirety.

```
my $NAME     = 'RSS Punter';
my $ID       = 'rss.qmacro.dyndns.org';
my $VERSION  = '0.1';
my $reg_file = 'registrations';
my %reg;

my %cache;

my %sources = (

  'jonudell' => 'http://udell.roninhouse.com/udell.rdf',
  'slashdot' => 'http://slashdot.org/slashdot.rdf',

  # etc ...

);

tie (%reg, 'MLDBM', $reg_file) or die "Cannot tie to $reg_file: $!\n";

my $c = new Jabber::Connection(
  server    => 'localhost:5999',
  localname => $ID,
  ns        => 'jabber:component:accept',
);

unless ($c->connect()) { die "oops: ".$c->lastError; }

$SIG{HUP} = $SIG{KILL} = $SIG{TERM} = $SIG{INT} = \&cleanup;

debug("registering RSS beat");
$c->register_beat(1800, \&rss);

debug("registering IQ handlers");
$c->register_handler('iq',\&iq_register);
$c->register_handler('iq',\&iq_version);
$c->register_handler('iq',\&iq_browse);
$c->register_handler('iq',\&iq_notimpl);

$c->auth('secret');

$c->start;
```

```
sub iq_register {

  my $node = shift;

  debug("[iq_register]");
  return unless my $query = $node->getTag('', NS_REGISTER);
  debug("--> registration request");

  # Reg query
  if ($node->attr('type') eq IQ_GET) {
    $node = toFrom($node);
    $node->attr('type', IQ_RESULT);
    my $instructions = "Choose an RSS source from: ".join(", ", keys %sources);
    $query->insertTag('instructions')->data($instructions);
    $query->insertTag('text');
    $c->send($node);
  }

  # Reg request
  if ($node->attr('type') eq IQ_SET) {

    # Strip JID to user@host
    my $jid = stripJID($node->attr('from'));

    $node = toFrom($node);
    my $source;

    # Could be an unregister
    if ($query->getTag('remove')) {
      delete $reg{$jid};
      $node->attr('type', IQ_RESULT);
    }

    # Otherwise it's a registration for a source
    elsif ($source = $query->getTag('text')->data
           and exists($sources{$source})) {
      my $element = $reg{$jid};
      $element->{$source} = 1;
      $reg{$jid} = $element;
      $node->attr('type', IQ_RESULT);
    }

    else {
      $node->attr('type', IQ_ERROR);
      my $error = $node->insertTag('error');
      $error->attr('code', '405');
      $error->data('Not Allowed');
    }
```

```
    $c->send($node);

  }

  return r_HANDLED;

}


sub iq_version {

  my $node = shift;
  debug("[iq_version]");

  return unless my $query = $node->getTag('', NS_VERSION)
            and $node->attr('type', IQ_GET);

  debug("--> version request");

  $node = toFrom($node);
  $node->attr('code', IQ_RESULT);
  $query->insertTag('name')->data($NAME);
  $query->insertTag('version')->data($VERSION);
  $query->insertTag('os')->data(`uname -sr`);
  $c->send($node);

  return r_HANDLED;

}


sub iq_browse {

  my $node = shift;
  debug("[iq_browse]");

  return unless my $query = $node->getTag('', NS_BROWSE)
            and $node->attr('type', IQ_GET);

  debug("--> browse request");

  $node = toFrom($node);
  $node->attr('type', IQ_RESULT);
  my $rss = $query->insertTag('service');
  $rss->attr('type', 'rss');
  $rss->attr('jid', $ID);
  $rss->attr('name', $NAME);
  $rss->insertTag('ns')->data(NS_REGISTER);
  $c->send($node);

  return r_HANDLED;
```

```
}


sub iq_notimpl {

  my $node = shift;
  $node = toFrom($node);
  $node->attr('type', IQ_ERROR);
  my $error = $node->insertTag('error');
  $error->attr('code', '501');
  $error->data('Not Implemented');
  $c->send($node);

  return r_HANDLED;

}


sub rss {

  debug("[rss]");

  # Create NodeFactory
  my $nf = new Jabber::NodeFactory;

  # Go through each of the RSS sources
  foreach my $source (keys %sources) {

    # Retrieve attempt
    my $data = get($sources{$source});

    # Didn't get it? Next one
    unless (defined($data)) {
      debug("cannot retrieve $source");
      next;
    }

    # Parse the RSS
    my $rss = XML::RSS->new();
    eval { $rss->parse($data) };

    if ($@) {
      debug("problems parsing $source");
      next;
    }

    my @items = @{$rss->{items}};

    # Check new items
    debug("$source: looking for new items");
    foreach my $item (@items) {
```

```
      # Stop checking if we get to items already seen
      last if exists $cache{$source} and $cache{$source} eq $item->{link};

      debug("$source: new item $item->{title}");

      # Create a headline message
      my $msg = $nf->newNode('message');

      $msg->attr('type', 'headline');
      $msg->attr('from', join('@', $source, $ID));
      $msg->insertTag('subject')->data($item->{title});
      $msg->insertTag('body')->data($item->{description});

      my $xoob = $msg->insertTag('x', NS_XOOB);
      $xoob->insertTag('url')->data($item->{link});
      $xoob->insertTag('desc')->data($item->{description});

      # Deliver to all that want it
      foreach my $jid (keys %reg) {

        my $registration = $reg{$jid};

        if (exists($registration->{$source})) {
          $msg->attr('to', $jid);
          debug("punting to $jid");
          $c->send($msg);
        }

      }

      # Prevent all items counted as new the
      # first time around
      last unless exists($cache{$source});

    }

    # Remember the latest new item
    $cache{$source} = $items[0]->{link};

  }

}


sub cleanup {

  debug("cleaning up");
  untie %reg;
  $c->disconnect;
  exit;

}
```

```
sub toFrom {
  my $node = shift;
  my $to = $node->attr('to');
  $node->attr('to', $node->attr('from'));
  $node->attr('from', $to);
  return $node;
}


sub stripJID {

  my $JID = shift;
  $JID =~ s|/.*$||;
  return $JID;

}


sub debug {

  print STDERR "debug: ", @_, "\n";

}
```

Notes

[1] RDF stands for Resource Description Framework.

[2] http://www.jabbercentral.org

[3] although a Jabber server could consist of a *collection* of **jabberd**s running on separate hosts

[4] Note that despite the tag name, you can specify an IP address or a *hostname* in `<ip/>`.

[5] Likewise, the namespace `jabber:component:exec` "matches" the STDIO component connection method and the significant tag name in *it's* component instance definition format: (*<exec/>*)—see the section called *STDIO* in Chapter 4.

[6] The registration process with the JSM to create a new user account uses `jabber:iq:register` to qualify the registration data exchanged. The registration process with the JSM to modify the account details (name, email address, and so on) also uses `jabber:iq:register` to qualify the account amendment data exchanged. Both types of registration request are addressed to the JSM. The key difference, which allows the JSM to distinguish between *what* is being requested, is that in the new user registration process, no session is active on the stream between client and server, whereas in the account amendment process, a session *is* active. This is also mentioned in the section called *Passwords* in Chapter 6.

[7] It won't be by the time you read this, so don't try it! :-)

[8] This isn't as bad as it seems. Take store and forward, for example, a feature provided by JSM's `mod_offline` module. While a message sent to a component won't be stored and forwarded if that component is not connected, a message sent from a component to a client *will* get stored and forwarded (if the client is offline), because the message will be routed to the JSM (because of the `[hostname]` in the address), which can decided what action to take—pass directly to the client if he's online, or store and forward later.

[9] Indeed, the author of the `Jabber::Connection` library (ahem) has taken the (heart)beat idea, the handler chain idea, and even the low-level `NodeFactory` mechanisms, directly from the JSM and the server libraries, in homage to the Jabber server's classic design.

[10] If you wish to have more granular control over your script, you can of course use the `process()` function directly, just as you would with the `Net::Jabber` and `JabberPy` libraries. Be aware, however, that a heartbeat is only maintained in the context of the `start()` method.

[11] `Jabber::NodeFactory` is the "wrapper" around the class that actually represents the elements (the nodes), which is `Jabber::NodeFactory::Node`. Nodes are created using the `Jabber::NodeFactory` class.

[12] Ideally we'd just use one instance of `XML::RSS` for the whole `rss()` function, but the way `XML::RSS` currently works requires us to create a new instance for every source we wish to work with.

Headline viewer

Wow. That was a biggie. So we're going to close this Chapter on a lighter note, and what could be more fitting than to build a complementary script with which we can keep an eye on those RSS headlines that are punted to us throughout the day.

In most of my workplaces, the monitor I have is 17 inches. Which means that every application must make a justification for existence on a plot of the pixel real-estate, a justification relative to it's window size. So something that allows me to watch RSS headlines as they scroll by ought to be small. It has no reason to be large.

Until now, I've droned on about the great leverage that there is to building solutions that make use of off the shelf clients. [1] But it's time to buck the trend, indeed to make another point. While you can orientate the features of your Jabber-based applications in the direction of standard clients, to take advantage of the installed base of Jabber clients, if you do want to create a client that works differently, a client that fits your needs exactly, then go ahead: it will be surprisingly straightforward. The mantra of "server side complex, client side simple" (with apologies to George Orwell) is there to help us. What's more, we can put into action an idea expressed earlier in the book (in the section called *Custom Clients* in Chapter 2):

> A Jabber client is a piece of software that implements as much of the Jabber protocol as required to get the job done.

If we're going to build a headline viewer client, and know that the information is going to get punted to us in headline type `<message/>` elements, why have our viewer client understand or deal with anything else? To implement a Jabber solution, we pick and choose the parts of the protocol that make sense in the context of that solution. If you want to transport RPC calls in data payloads between two endpoints, why bother with roster management or rich-text chat facilities? If you just want to join a conference room to talk with the other room occupants, why bother with threaded one-on-one conversations? If you need routing and presence capabilities to have your oven know when you've arrived home, why engineer anything else?

What we're going to write here is a simple headline viewer. Nothing more. Nothing less. It will know the tiniest thing about presence— as the headlines come in as `<message/>` elements, it will need to announce its availability to the JSM—and the viewer we're going to build will be a Jabber client that will have a session context with the JSM, we need to tell the JSM that we're available when we start the client up. Otherwise the headlines will simply be queued up by the store and forward mechanism ready for the "next time" we're available.

We'll leave the registration with the RSS punter to another client that knows about agents (services) and can interact in a `jabber:iq:register` kind of way. I'm not a big fan of the "one size fits all, one client for everything" philosophy; I prefer to use different features of different programs to get me through my day. So while our headline viewer will receive, understand and display the `<message type='headline'/>`s, we'll

use WinJab, or even JIM (Jabber Instant Messenger) to manage our RSS source subscriptions. Figure 8-8 illustrates the process of registration with our RSS punter component, using JIM.

Figure 8-8. Registering with the RSS punter with JIM

The suggestion of JIM as a client to complement (cheers!) or make up for the lack of features in (boos!) our headline viewer is deliberately ironic. JIM's remit is to provide support for core Jabber client features, of which (as yet) headline messages are not considered to be a part. So while JIM can interact with services and register (and unregister—which will send the `<remove/>` tag in the query, as described in the section called *Handling registration requests*), it doesn't handle headline type messages. Which is perfectly fine. Our headline viewer won't handle chat or normal messages. The point is, it's not *supposed* to.

It's worth pointing out that another reason why our headline viewer client can remain simple is because the RSS punter component will be doing all the hard work for us. Unlike other (non-Jabber) headline viewer programs that are available, our script depends upon the RSS punter. It's that component that will maintain the list of RSS sources. It's also that component that will retrieve those sources at regular intervals and check for new items. All we have to do is sit back and have those new items pushed to us, at which point our client has to make a slight effort to insert the details of those new items into the viewer display. That's more or less it.

The plan

The viewer is visual, so let's see at what it's going to look like. Figure 8-9 shows the headline viewer client in action. There's a scrollable area where the headline titles are displayed. We can *clear* that area, or select a headline and call up a web browser to *fetch* the story by passing the URL to it.

Figure 8-9. The headline viewer client

It's also nice and small, visually and in the amount of code we're going to have to write. We connect to our Jabber server, set up a handler for the incoming headline messages, build our display, send our availability, and sit back.

Actually, we need to say a few things about the "sitting back" bit. We know that Jabber programming implies an event model. We're going to use Tk—the widget library for building GUI applications, with bindings for many languages. Tk itself has an event model, which in many ways reflects Jabber's. Table 8-2 shows how Jabber and Tk reflect in this programming model puddle.

Table 8-2. Jabber and Tk event model reflections

Jabber	Tk
Establish connection to server	Construction of widgets

Definition of callbacks to handle incoming elements	Definition of callbacks to handle UI events
Setting of a heartbeat function [a]	Setting of a command to execute regularly with `repeat()`
Launching the event loop	Starting `MainLoop()`
Notes: a. See the section called *Preparation of the RSS event function and element handlers*.	

Having one program governed by two independent event loops is not what we want to try and achieve. We want Jabber and Tk's event models to cooperate. This is achievable by making one of the two models the master and the other one the slave. Using Tk's `repeat()` method to invoke a function that calls our Jabber library's `process()` method should do the trick. We can hand over control to Tk with `MainLoop()`, and know that our Jabber event model will get a look in because of the Tk event callback we've defined with `repeat()`.

The script

We're going to use Perl, and the `Net::Jabber` Jabber library. Starting with the declarations of the libraries we're going to use, along with some constants:

```
use Tk;
use Net::Jabber qw(Client);
use strict;

use constant SERVER   => 'gnu.pipetree.com';
use constant PORT     => 5222;
use constant USER     => 'dj';
use constant PASSWORD => 'secret';
use constant RESOURCE => 'hlv';

use constant BROWSER  => '/usr/bin/konqueror';
```

The script will connect to Jabber as a client, so we specify that in our `use` statement to have the appropriate `Net::Jabber` modules loaded. We're going to be connecting to the Jabber server at `gnu.pipetree.com`, although, as we said, the RSS punter might live somewhere else. It just so happens that in our scenario, there's a reference to the component in `gnu.pipetree.com`'s JSM `<browse/>` section, so that we can carry out our registration conversations with it (using JIM, for example).

If the *Fetch* button is pressed when an item in the list is selected (see Figure 8-9), we want to jump to the story by launching a web browser. The constant `BROWSER` used here refers to the browser on our local machine. [2]

```
my @headlines;
my @list;
```

We declare two arrays: `@headlines`, which we'll use to hold the items as they arrive contained in the headline

<message/> elements on the XML stream, and @list, to hold the URLs that relate to those items in @headlines.

After connecting to and authenticating ourselves with the Jabber server (this is very similar to the way the coffee monitor script connects and authenticates in the section called *setup_Jabber()*):

```
my $connection = Net::Jabber::Client->new();

$connection->Connect(
  hostname => SERVER,
  port     => PORT,
) or die "Cannot connect ($!)\n";

my @result = $connection->AuthSend(
  username => USER,
  password => PASSWORD,
  resource => RESOURCE,
);

if ($result[0] ne "ok") {
  die "Ident/Auth with server failed: $result[0] - $result[1]\n";
}
```

we set up our callback to take care of incoming <message/> elements. This is the handle_message() function:

```
$connection->SetCallBacks( message => \&handle_message );
```

Now it's time to build our GUI. We start by creating a main window, giving it a title and geometry, and establishing the cooperation between the two event models with the repeat() method:

```
my $main = MainWindow->new( -title => "Headline Viewer" );
$main->geometry('50x5+10+10');
$main->repeat(5000, \&check_headlines);
```

repeat() will arrange Tk's main event loop to hiccup every five seconds (the first argument is measured in milliseconds) and call our check_headlines() function.

Next, we build a frame to hold the three buttons *Clear*, *Fetch*, and *Exit*, and a scrollable list to contain the item titles as we receive them:

```
# Button frame
my $buttons = $main->Frame();
$buttons->pack(qw/-side bottom -fill x/);

# Headline list
```

```
my $list = $main->Scrolled(qw/Listbox -scrollbars e -height 40 -setgrid 1/);
```

Defining the buttons, one at a time, brings our attention to the Tk UI event model, in that we define the handlers using the `-command` argument of the `Button()` method. The handlers' jobs are quite small, so we can get away with writing them "in-line":

```
# Clear button
my $button_clear = $buttons->Button(
             -text      => 'Clear',
             -underline => '0',
             -command   => sub
             {
               @list = (); $list->delete(0, 'end')
             },
           );
```

If called, the *Clear* button will clear the scrollable display by calling the `delete()` method on our `$list` object, and emptying the corresponding array of URLs.

The *Fetch* button extracts the URL from the item that is highlighted in the scrollable list (using the `curselection()` method to retrieve the index value) which is then used to look up the `@list` array, and calls the external browser program in the background, passing it that URL. Many browsers accept a URL as the first argument, if your choice of browser doesn't, you'll need to modify this call slightly.

```
# Fetch Button
my $button_fetch = $buttons->Button(
             -text      => 'Fetch',
             -underline => '0',
             -command   => sub
             {
               system(
                 join(" ", (BROWSER, $list[$list->curselection], "&"))
               )
             },
           );
```

The *Exit* button, if pressed, uses `destroy()` to, well, destroy the main window. This will cause Tk's main event loop to come to an end, passing control back to the statement in the script following where that main event loop was launched (with `MainLoop()`).

```
# Exit button
my $button_exit = $buttons->Button(
             -text      => 'exit',
             -underline => '0',
             -command   => [$main => 'destroy'],
           );
```

Having created all the buttons, and packed everything into our window with the `pack()` method:

```
$button_clear->pack(qw/-side left -expand 1/);
$button_fetch->pack(qw/-side left -expand 1/);
$button_exit->pack(qw/-side left -expand 1/);

$list->pack(qw/-side left -expand 1 -fill both/);
```

we announce to the JSM that we're available:

```
$connection->PresenceSend();
```

All that remains for us to do is start Tk's main event loop. We include a call to the `Net::Jabber Disconnect()` method for when the *Exit* button is pressed and control returns to the script, so we can gracefully end our Jabber connection:

```
MainLoop();

$connection->Disconnect;
exit(0);
```

We defined the `check_headlines()` function as the function to invoke every five seconds.

```
sub check_headlines {
  $connection->Process(1);
  while (@headlines) {
    my $headline = pop @headlines;
    $list->insert(0, $headline->{title});
    unshift @list, $headline->{link};
  }
}
```

To check for any messages that have arrived on the XML stream, we can call the `Process()` method on our connection object. If there *are* any waiting messages, the callback that we defined to handle them—`handle_message()`—will be called:

```
sub handle_message {
  my $msg = new Net::Jabber::Message($_[1]);
  return unless $msg->GetType eq 'headline';

  my ($oob) = $msg->GetX('jabber:x:oob');
  push @headlines, {
                     link => $oob->GetURL(),
                     title => $msg->GetSubject(),
                   };
```

```
}
```

This message handling callback will ignore everything but `<message/>` elements that have the value "`headline`" in the `type` attribute. Remembering that a headline message, complete with an `<x/>` extension, qualified with the `jabber:x:oob` namespace, looks like this:

```
<message type='headline' to='dj@qmacro.dyndns.org'>
  <subject>JabberCon Update 11:45am - Aug 20</subject>
  <body>JabberCon Update - Monday Morning</body>
  <x xmlns='jabber:x:oob'>
    <url>http://www.jabbercentral.com/news/view.php?news_id=998329970</url>
    <desc>JabberCon Update - Monday Morning</desc>
  </x>
</message>
```

we can see fairly easily what the `GetX()` method does. It returns, in list context, all the `<x/>` elements contained in the element represented by `$msg` that are qualified by the `jabber:x:oob` namespace. We're only expecting there to be one, which is why we plan to throw all but the first array item away with the `($oob)` construction. After the call to `GetX()`, the object in `$oob` represents this part of the message:

```
<x xmlns='jabber:x:oob'>
  <url>http://www.jabbercentral.com/news/view.php?news_id=998329970</url>
  <desc>JabberCon Update - Monday Morning</desc>
</x>
```

The item's details—the URL and title—are pushed onto the `@headlines` list, and our headline type message handling function has done its job. Control passes back to the `check_headlines()` script, to immediately after the call to the `Process()` method.

The `handle_message()` function may have bee called multiple times, depending on how many elements had arrived; so the `@headlines` array might contain more than one item. We run through the array, `pop()`ping off each headline in turn, inserting the title into our scrollable list object, and the URL into the corresponding position in our `@list` array:

```
$list->insert(0, $headline->{title});
unshift @list, $headline->{link};
```

The script

Here's the script in its entirety.

```
use Tk;
use Net::Jabber qw(Client);
use strict;
```

```
use constant SERVER   => 'gnu.pipetree.com';
use constant PORT     => 5222;
use constant USER     => 'dj';
use constant PASSWORD => 'secret';
use constant RESOURCE => 'hlv';

use constant BROWSER  => '/usr/bin/konqueror';

my @headlines;
my @list;

my $connection = Net::Jabber::Client->new();

$connection->Connect(
  hostname => SERVER,
  port     => PORT,
) or die "Cannot connect ($!)\n";

my @result = $connection->AuthSend(
  username => USER,
  password => PASSWORD,
  resource => RESOURCE,
);

if ($result[0] ne "ok") {
  die "Ident/Auth with server failed: $result[0] - $result[1]\n";
}

$connection->SetCallBacks( message => \&handle_message );

my $main = MainWindow->new( -title => "Headline Viewer" );
$main->geometry('50x5+10+10');
$main->repeat(5000, \&check_headlines);

# Button frame
my $buttons = $main->Frame();
$buttons->pack(qw/-side bottom -fill x/);

# Headline list
my $list = $main->Scrolled(qw/Listbox -scrollbars e -height 40 -setgrid 1/);

# Clear button
my $button_clear = $buttons->Button(
             -text      => 'Clear',
             -underline => '0',
             -command   => sub
             {
               @list = (); $list->delete(0, 'end')
```

```
            },
          );

# Fetch Button
my $button_fetch = $buttons->Button(
            -text      => 'Fetch',
            -underline => '0',
            -command   => sub
            {
              system(
                join(" ", (BROWSER, $list[$list->curselection], "&"))
              )
            },
          );

# Exit button
my $button_exit = $buttons->Button(
            -text      => 'exit',
            -underline => '0',
            -command   => [$main => 'destroy'],
          );


$button_clear->pack(qw/-side left -expand 1/);
$button_fetch->pack(qw/-side left -expand 1/);
$button_exit->pack(qw/-side left -expand 1/);

$list->pack(qw/-side left -expand 1 -fill both/);

$connection->PresenceSend();

MainLoop();

$connection->Disconnect;
exit(0);

sub check_headlines {
  $connection->Process(1);
  while (@headlines) {
    my $headline = pop @headlines;
    $list->insert(0, $headline->{title});
    unshift @list, $headline->{link};
  }
}

sub handle_message {
  my $msg = new Net::Jabber::Message($_[1]);
  return unless $msg->GetType eq 'headline';
```

```
  my ($oob) = $msg->GetX('jabber:x:oob');
  push @headlines, {
                     link => $oob->GetURL(),
                     title => $msg->GetSubject(),
                   };
}
```

Notes

[1] Alright, I said it. "Leverage." It's the only occasion in the book, ok?

[2] (*Konqueror*, my browser of choice in the KDE environment).

Prev RSS punter | Home | Up | Next Pointers for further development

Chapter 9. Pointers for further development

Table of Contents

From client to component

XML-RPC over Jabber

The ERP connection

Browsing LDAP

Conferencing

Appendices. Appendices

Table of Contents

Appendix A. The `jabber.xml` Contents

This Appendix contains the contents of the default `jabber.xml` configuration file installed with version 1.4.1 of the Jabber server. The contents have been extended with the addition of Conferencing and local Jabber User Directory (JUD) components. Each 'section' of the configuration has been separated from the next with comment style dividing lines, and the configuration has been set up for the host on which we installed our Jabber server in Chapter 3 - `yak`.

The structure and contents of this `jabber.xml` configuration are discussed in the section called *A Tour of `jabber.xml`* in Chapter 4.

Figure A-1. Version 1.4.1 `jabber.xml` with JUD and Conferencing

```
<jabber>

  <!---------------------------------->

  <service id="sessions">

    <host><jabberd:cmdline flag="h">yak</jabberd:cmdline></host>

    <jsm xmlns="jabber:config:jsm">
      <filter>
          <default/>
          <max_size>100</max_size>
          <allow>
              <conditions>
                  <ns/>
                  <unavailable/>
                  <from/>
                  <resource/>
                  <subject/>
                  <body/>
                  <show/>
                  <type/>
                  <roster/>
                  <group/>
              </conditions>
              <actions>
                  <error/>
                  <offline/>
                  <forward/>
                  <reply/>
                  <continue/>
```

```
                <settype/>
            </actions>
        </allow>
    </filter>
    <vCard>
      <FN>Jabber Server on yak</FN>
      <DESC>A Jabber Server!</DESC>
      <URL>http://yak/</URL>
    </vCard>
    <register notify="yes">
      <instructions>Choose a userid and password to register.</instructions>
      <name/>
      <email/>
    </register>
    <welcome>
      <subject>Welcome!</subject>
      <body>Welcome to the Jabber server on yak</body>
    </welcome>
    <!--
    <admin>
      <read>support@yak</read>
      <write>admin@yak</write>
      <reply>
        <subject>Auto Reply</subject>
        <body>This is a special administrative address.</body>
      </reply>
    </admin>
    -->
    <update><jabberd:cmdline flag="h">yak</jabberd:cmdline></update>
    <vcard2jud/>
    <browse>
      <service type="jud" jid="jud.yak" name="yak User Directory">
        <ns>jabber:iq:search</ns>
        <ns>jabber:iq:register</ns>
      </service>
      <conference type="public" jid="conference.yak" name="yak Conferencing"/>
    </browse>

  </jsm>

  <load main="jsm">
    <jsm>./jsm/jsm.so</jsm>
    <mod_echo>./jsm/jsm.so</mod_echo>
    <mod_roster>./jsm/jsm.so</mod_roster>
    <mod_time>./jsm/jsm.so</mod_time>
    <mod_vcard>./jsm/jsm.so</mod_vcard>
    <mod_last>./jsm/jsm.so</mod_last>
    <mod_version>./jsm/jsm.so</mod_version>
    <mod_announce>./jsm/jsm.so</mod_announce>
```

```
    <mod_agents>./jsm/jsm.so</mod_agents>
    <mod_browse>./jsm/jsm.so</mod_browse>
    <mod_admin>./jsm/jsm.so</mod_admin>
    <mod_filter>./jsm/jsm.so</mod_filter>
    <mod_offline>./jsm/jsm.so</mod_offline>
    <mod_presence>./jsm/jsm.so</mod_presence>
    <mod_auth_plain>./jsm/jsm.so</mod_auth_plain>
    <mod_auth_digest>./jsm/jsm.so</mod_auth_digest>
    <mod_auth_0k>./jsm/jsm.so</mod_auth_0k>
    <mod_log>./jsm/jsm.so</mod_log>
    <mod_register>./jsm/jsm.so</mod_register>
    <mod_xml>./jsm/jsm.so</mod_xml>
  </load>

</service>

<!---------------------------------->

<xdb id="xdb">

  <host/>

  <load>
    <xdb_file>./xdb_file/xdb_file.so</xdb_file>
  </load>

  <xdb_file xmlns="jabber:config:xdb_file">
    <spool><jabberd:cmdline flag='s'>./spool</jabberd:cmdline></spool>
  </xdb_file>

</xdb>

<!---------------------------------->

<service id="c2s">

  <load>
    <pthsock_client>./pthsock/pthsock_client.so</pthsock_client>
  </load>

  <pthcsock xmlns='jabber:config:pth-csock'>
    <authtime/>
    <karma>
      <init>10</init>
      <max>10</max>
      <inc>1</inc>
      <dec>1</dec>
      <penalty>-6</penalty>
      <restore>10</restore>
    </karma>
    <ip port="5222"/>
```

```
  </pthcsock>

</service>

<!----------------------------------->

<log id='elogger'>
  <host/>
  <logtype/>
  <format>%d: [%t] (%h): %s</format>
  <file>error.log</file>
  <stderr/>
</log>

<!----------------------------------->

<log id='rlogger'>
  <host/>
  <logtype>record</logtype>
  <format>%d %h %s</format>
  <file>record.log</file>
</log>

<!----------------------------------->

<service id="dnsrv">

  <host/>

  <load>
    <dnsrv>./dnsrv/dnsrv.so</dnsrv>
  </load>

  <dnsrv xmlns="jabber:config:dnsrv">
      <resend service="_jabber._tcp">s2s</resend>
      <resend>s2s</resend>
  </dnsrv>

</service>

<!----------------------------------->

<service id="s2s">

  <load>
    <dialback>./dialback/dialback.so</dialback>
  </load>

  <dialback xmlns='jabber:config:dialback'>
    <legacy/>
    <ip port="5269"/>
    <karma>
```

```
      <init>50</init>
      <max>50</max>
      <inc>4</inc>
      <dec>1</dec>
      <penalty>-5</penalty>
      <restore>50</restore>
    </karma>
  </dialback>

</service>

<!---------------------------------->

<service id='conf'>

  <host>conference.yak</host>

  <load>
    <conference>./conference-0.4.1/conference.so</conference>
  </load>

  <conference xmlns="jabber:config:conference">
    <public/>
    <vCard>
      <FN>yak Chatrooms</FN>
      <DESC>This service is for public chatrooms.</DESC>
      <URL>http://yak/chat</URL>
    </vCard>
    <history>20</history>
    <notice>
      <join> has become available</join>
      <leave> has left</leave>
      <rename> is now known as </rename>
    </notice>
    <room jid="kitchen@conference.yak">
      <name>The Kitchen</name>
      <notice>
        <join> has entered the cooking melee</join>
        <leave> can't stand the heat</leave>
        <rename> now answers to </rename>
      </notice>
    </room>
  </conference>

</service>

<!---------------------------------->

<service id="jud">

  <host>jud.yak</host>
```

```
    <load>
      <jud>./jud-0.4/jud.so</jud>
    </load>

    <jud xmlns="jabber:config:jud">
      <vCard>
        <FN>JUD on yak</FN>
        <DESC>yak User Directory Services</DESC>
        <URL>http://yak/</URL>
      </vCard>
    </jud>

  </service>

  <!----------------------------------->

  <io>

    <karma>
      <heartbeat>2</heartbeat>
      <init>64</init>
      <max>64</max>
      <inc>6</inc>
      <dec>1</dec>
      <penalty>-3</penalty>
      <restore>64</restore>
    </karma>

    <rate points="5" time="25"/>

  </io>

  <!----------------------------------->

  <pidfile>./jabber.pid</pidfile>

  <!----------------------------------->

</jabber>
```
