PART I

XMPP Protocol and Architecture

1 Getting to Know XMPP

WHAT'S IN THIS CHAPTER?

- The history of XMPP
- XMPP networks and connections
- XMPP's three building block stanzas

The eXtensible Messaging and Presence Protocol (XMPP) is, at its most basic level, a protocol for moving small, structured pieces of data between two places. From this humble basis, it has been used to build large-scale instant messaging systems, Internet gaming platforms, search engines, collaboration spaces, and voice and video conferencing systems. More unique applications appear every day, further demonstrating how versatile and powerful XMPP can be.

XMPP is made of a few small building blocks, and on top of these primitives many larger constructions have been made. Within XMPP are systems for building publish-subscribe services, multi-user chat, form retrieval and processing, service discovery, real-time data transfer, privacy control, and remote procedure calls. Often, XMPP programmers create their own, unique constructions that are fitted exactly for the problem at hand.

Most social media constructs that have propelled web sites like Facebook, MySpace, and Twitter into the forefront are also baked into XMPP. Within XMPP, you'll find rosters full of contacts that create a social graph with directed or undirected edges. Presence notifications are sent automatically when contacts come online and go offline, and private and public messages are the bread and butter application of XMPP systems. Developers will sometimes choose XMPP as the underlying technology layer simply because it gives them many social features for free, leaving them to concentrate on the unique pieces of their application.

The possibilities are vast, but before you can begin, you need to know about XMPP's different pieces and how they fit together into a cohesive whole.

WHAT IS XMPP?

XMPP, like all protocols, defines a format for moving data between two or more communicating entities. In XMPP's case, the entities are normally a client and a server, although it also allows for peer-to-peer communication between two servers or two clients. Many XMPP servers exist on the Internet, accessible to all, and form a federated network of interconnected systems.

Data exchanged over XMPP is in XML, giving the communication a rich, extensible structure. Many modern protocols forgo the bandwidth savings of a binary encoding for the more practical feature of being human readable and therefore easily debugged. XMPP's choice to piggyback on XML means that it can take advantage of the large amount of knowledge and supporting software for dealing with XML.

One major feature XMPP gets by using XML is XML's extensibility. It is extremely easy to add new features to the protocol that are both backward and forward compatible. This extensibility is put to great use in the more than 200 protocol extensions registered with the XMPP Standards Foundation and has provided developers with a rich and practically unlimited set of tools.

XML is known primarily as a document format, but in XMPP, XML data is organized as a pair of streams, one stream for each direction of communication. Each XML stream consists of an opening element, followed by XMPP stanzas and other top-level elements, and then a closing element. Each XMPP stanza is a first-level child element of the stream with all its descendent elements and attributes. At the end of an XMPP connection, the two streams form a pair of valid XML documents.

XMPP stanzas make up the core part of the protocol, and XMPP applications are concerned with sending and responding to various kinds of stanzas. Stanzas may contain information about other entities' availability on the network, personal messages similar to e-mail, or structured communication intended for computer processing. An example stanza is shown here:

```
<message to='elizabeth@longbourn.lit'
         from='darcy@pemberley.lit/dance'
         type='chat'>
  <body>What think you of books?</body>
</message>
```

In a typical client-server XMPP session, a stanza such as this one from Elizabeth to Mr. Darcy will travel from Elizabeth's client to her server. Her server will notice that it is addressed to an entity on a remote server and will establish an XMPP connection with the remote server and forward the message there. This communication between servers resembles the e-mail network, but unlike e-mail servers, XMPP servers always communicate directly with each other and not through intermediate servers.

This direct communication eliminates some common vectors for spam and unauthorized messages. This is just one of the many ways in which XMPP is designed for security. It also supports encrypted communications between endpoints through use of Transport Layer Security (TLS) and strong authentication mechanisms via Simple Authentication and Security Layers (SASL).

XMPP is designed for the exchange of small bits of information, not large blobs of binary data. XMPP can, however, be used to negotiate and set up out-of-band or in-band transports, which can move large blocks from point to point. For these kinds of transfers, XMPP functions as a signaling layer.

The focus on small, structured bits of data gives the XMPP protocol extremely low latency and makes it extremely useful for real-time applications. These applications, which include collaborative spaces, games, and synchronization, are driving XMPP's growth in popularity as developers experiment with the real-time Web.

You will see how easy it is to make real-time web applications through this book's examples. By the end of the book you should have a thorough understanding of why so many people are excited about XMPP's power and promise.

A BRIEF HISTORY OF XMPP

The XMPP protocol is now more than 10 years old, and it has come a long way from its humble beginnings. Much of XMPP's design is due to the environment in which XMPP was created, and the history of XMPP provides an interesting case study in how open protocols foster adoption and innovation.

In 1996, Mirabilis released ICQ, which popularized rapid, personal communication among Internet users. Its use spread rapidly, and before long other companies were releasing similar products. In 1997, AOL launched AOL Instant Messenger. Yahoo followed suit in 1998 with Yahoo Pager (eventually renamed Yahoo Messenger), and in 1999 Microsoft finally joined the competition with MSN Messenger (now Windows Live Messenger).

Each of these instant messaging applications was tied to a proprietary protocol and network run by the companies that made them. Users of ICQ could not talk to Yahoo users and vice versa. It became common for users to run more than one of these applications to be able to talk to all of their contacts because no single vendor claimed 100% market share.

It didn't take long before developers desired to write their own clients for these proprietary IM networks. Some wished to make multiprotocol clients that could unite two or more of the IM networks, and others wanted to bring these applications to operating systems other than Microsoft Windows and Apple's Mac OS. These developers ran into many roadblocks; they had to reverse-engineer undocumented protocols, and the IM networks aggressively changed the protocol to thwart third-party developers.

It was in this climate that the idea for an open, decentralized IM network and protocol was born.

Jeremie Miller announced the Jabber project in January of 1999. Jabber was a decentralized instant messaging protocol based on XML and a server implementation called jabberd. A community immediately formed around the protocol and implementations spawning more clients and more ideas. By May of 2000, the core protocols were stabilized and jabberd reached a production release.

The Jabber Software Foundation (JSF) was founded in 2001 to coordinate the efforts around the Jabber protocol and its implementations. By late 2002, the JSF had submitted the core protocol specifications to the IETF process, and an IETF working group was formed. In October 2004, this standards process produced improved versions of the Jabber protocols, renamed XMPP, documented as RFCs 3920, 3921, 3922, and 3923.

During the protocol's early life, developers continued to expand its possibilities by submitting protocol extensions to the JSF. These extensions were called Jabber Extension Proposals (JEPs).

Eventually the JSF and the extensions followed the naming change from Jabber to XMPP and became the XMPP Standards Foundation (XSF) and XMPP Extension Proposals (XEPs).

By 2005, large-scale deployments of XMPP technology were well underway, highlighted by the launch of Google Talk, Google's own XMPP-based IM service.

Today, the XMPP ecosystem is quite large. Nearly 300 extensions have been accepted as XEPs, and dozens of client and server implementations have been created — both commercial and open source. Software developers of virtually any programming language can find a library to speed their XMPP application development efforts.

XMPP applications started out very IM-centric, reflecting its origins, but developers have found XMPP to be quite capable for a number of applications that weren't originally foreseen including search engines and synchronization software. This utility is a testament to the power of an open system and open standardization process.

Most recently, the IETF has formed a new XMPP working group to prepare the next versions of the XMPP specifications, incorporating all the knowledge gained since the original RFCs were published. XMPP continues to be refined and extended so that application developers and Internet users will always have an open, decentralized communications protocol.

THE XMPP NETWORK

Any XMPP network is composed of a number of actors. These actors can be categorized as servers, clients, components, and server plug-ins. An XMPP developer will write code to create or modify one of these types of actors. Each actor has its place on the XMPP network's stage.

Servers

XMPP servers, or more accurately, XMPP entities speaking the server-to-server protocol or the server end of the client-to-server protocol, are the circulatory system of any XMPP network. A server's job is to route stanzas, whether they are internal from one user to another or from a local user to a user on a remote server.

The set of XMPP servers that can mutually communicate forms an XMPP network. The set of public XMPP servers forms the global, federated XMPP network. If a server does not speak the server-to-server protocol, it becomes an island, unable to communicate with external servers.

An XMPP server will usually allow users to connect to it. It is, however, also possible to write applications or services that speak the server-to-server protocol directly in order to improve efficiency by eliminating routing overhead.

Anyone can run an XMPP server, and full-featured servers are available for nearly every platform. Ejabberd, Openfire, and Tigase are three popular open source choices that will work on Windows, Mac OS X, or Linux systems. Several commercial XMPP servers are available as well, including M-Link and Jabber XCP.

Clients

The majority of XMPP entities are clients, which connect to XMPP servers via the client-to-server protocol. Many of these entities are human-driven, traditional IM users, but there are also automated services running as *bots*.

Clients must authenticate to an XMPP server somewhere. The server routes all stanzas the client sends to the appropriate destination. The server also manages several aspects of the clients' sessions, including their roster and their bare address, which you see more of shortly.

All of the applications in this book are written as client applications. This is typically the starting point of most XMPP development. For applications without a user focus or with demanding needs, it is often preferable to create a different kind of entity, such as a server component.

Components

Clients are not the only things that may connect to XMPP servers; most servers also support external server *components*. These components augment the behavior of the server by adding some new service. These components have their own identity and address within the server, but run externally and communicate over a component protocol.

The component protocol (defined in XEP-0114) enables developers to create server extensions in a server-agnostic way. Any component using the protocol can run on any server that speaks the component protocol (assuming it doesn't use some special feature specific to a particular server). A multi-user chat service is a typical example of something that is often implemented as a component.

Components also authenticate to the server, but this authentication is simpler than the full SASL authentication for clients. Typically authentication is done with a simple password.

Each component becomes a separately addressable entity within the server and appears to the outside world as a sub-server. XMPP servers do not manage anything beyond basic stanza routing on behalf of connected components. This allows great freedom to component developers to do things exactly as they want, but places greater responsibility on them when they need functionality such as rosters and presence management.

The server also allows a component to internally route or manage stanzas for itself. A component can therefore create separately addressable pieces to be used as rooms, users, or whatever the developer requires. This is something that a client session cannot do and can be used to create really elegant services.

Finally, because components do not have resources managed for them, services that operate with many users or with a high amount of traffic can manage their own resources in a way that makes sense for their purpose. Developers often create services as client bots, only to discover later that the server's roster management capabilities often do not scale well to thousands upon thousands of contacts. Components can manage rosters, if they have them at all, in whichever way makes sense for the task and scale required.

Plug-ins

Many XMPP servers can also be extended via *plug-ins*. These plug-ins are usually written in the same programming language as the server itself and run inside the server's processes. Their purpose overlaps to a large degree with external components, but plug-ins may also access internal server data structures and change core server behavior.

The virtually limitless abilities afforded to server plug-ins come with a cost; plug-ins are not portable between different servers. A different server may be written in a completely different language, and its internal data structures may differ radically. This cost aside, plug-ins are sometimes the only way to get a particular job done.

Plug-ins have reduced overhead compared to components because they do not need to communicate over a network socket. They also need not parse or serialize XML and can, instead, work directly with internal server representations of stanzas. This can lead to much needed performance improvements when the application must scale.

XMPP ADDRESSING

Every entity on an XMPP network will have one or more addresses, or *JIDs*. JIDs (short for jabber identifiers) can take a variety of forms, but they normally look just like e-mail addresses. `darcy@pemberley.lit` and `elizabeth@longbourn.lit` are two examples of JIDs.

Each JID is made up of up to three pieces, the *local part*, the *domain*, and the *resource*. The domain portion is always required, but the other two pieces are optional, depending on their context.

The domain is the resolvable DNS name of the entity — a server, component, or plug-in. A JID consisting of just a domain is valid and addresses a server. Stanzas addressed to a domain are handled by the server itself and potentially routed to a component or plug-in.

The local part usually identifies a particular user at a domain. It appears at the beginning of a JID, before the domain, and it is separated from the rest of the JID by the @ character, just like the local part of an e-mail address. The local part can also be used to identify other objects; a multi-user chat service will expose each room as a JID where the local part references the room.

A JID's resource part most often identifies a particular XMPP connection of a client. For XMPP clients, each connection is assigned a resource. If Mr. Darcy, whose JID is `darcy@pemberley.lit`, is connected both from his study and his library, his connections will be addressable as `darcy@pemberley.lit/study` and `darcy@pemberley.lit/library`. Like the local part, a resource can be used to identify other things; on a multi-user chat service, the resource part of the JID is used to identify a particular user of a chat room.

JIDs are divided into two categories, *bare JIDs* and *full JIDs*. The full JID is always the most specific address for a particular entity, and the bare JID is simply the full JID with any resource part removed. For example, if a client's full JID is `darcy@pemberley.lit/library`, its bare JID would be `darcy@pemberley.lit`. In some cases, the bare JID and the full JID are the same, such as when addressing a server or a specific multi-user chat room.

Bare JIDs for clients are somewhat special, because the server itself will handle stanzas addressed to a client's bare JID. For example, a message sent to a client's bare JID will be forwarded to one or more connected resources of the user, or if the user is offline, stored for later delivery. Stanzas sent to full JIDs, however, are usually routed directly to the client's connection for that resource. You can think of bare JIDs as addressing the user's account as opposed to addressing one of the user's connected clients.

XMPP STANZAS

Work is accomplished in XMPP by the sending and receiving of XMPP stanzas over an XMPP stream. Three basic stanzas make up the core XMPP toolset. These stanzas are `<presence>`, `<message>`, and `<iq>`. Each type of stanza has its place and purpose, and by composing the right kinds of quantities of these stanzas, sophisticated behaviors can be achieved.

Remember that an XMPP stream is a set of two XML documents, one for each direction of communication. These documents have a root `<stream:stream>` element. The children of this `<stream:stream>` element consist of routable *stanzas* and stream related top-level children.

Each stanza is an XML element, including its children. The end points of XMPP communication process input and generate output on a stanza-by-stanza basis. The following example shows a simplified and short XMPP session:

```
<stream:stream>
  <iq type='get'>
    <query xmlns='jabber:iq:roster'/>
  </iq>

  <presence/>

  <message to='darcy@pemberley.lit'
           from='elizabaeth@longbourn.lit/ballroom'
           type='chat'>
    <body>I cannot talk of books in a ball-room; my head is always full of
      something else.</body>
  </message>

  <presence type='unavailable'/>
</stream:stream>
```

In this example, Elizabeth created an XMPP stream by sending the opening `<stream:stream>` tag. With the stream open, she sent her first stanza, an `<iq>` element. This `<iq>` element requested Elizabeth's roster, the list of all her stored contacts. Next, she notified the server that she was online and available with a `<presence>` stanza. After noticing that Mr. Darcy was online, she sent him a short `<message>` stanza, thwarting his attempt at small talk. Finally, Elizabeth sent another `<presence>` stanza to inform the server she was unavailable and closed the `<stream:stream>` element, ending the session.

You have now seen an example of each kind of XMPP stanza in action. Each of these is explained in more detail, but first, you should learn about what properties they all share.

Common Attributes

All three stanzas support a set of common attributes. Whether they are attributes of `<presence>`, `<message>`, or `<iq>` elements, the following attributes all mean the same thing.

from

Stanzas almost always have a `from` attribute. This attribute identifies the JID of the stanza's origin. Setting the `from` attribute on outgoing stanzas is not recommended; the server adds the correct `from` attribute to all stanzas as they pass through, and if you set the `from` attribute incorrectly, the server may reject your stanza altogether.

If the `from` attribute is missing on a received stanza in a client-to-server stream, this is interpreted to mean that the stanza originated from the server itself. In the server-to-server protocol, a missing `from` attribute is an error.

Note that the example stanzas in this book often include the `from` attribute. This is done for clarity and disambiguation.

to

XMPP servers route your stanzas to the JID supplied in the `to` attribute. Similarly to the `from` attribute, if the `to` attribute is missing in a client-to-server stream, the server assumes it is a message intended for the server itself. It is recommended that you omit the `to` attribute when you address the server itself.

If the JID specified in the `to` attribute is a user, the server potentially handles the stanza on the user's behalf. If the destination is a bare JID, the server handles the stanza. This behavior is different for the three stanza types, and is explained alongside each type. If a full JID is specified as the destination, the server routes the stanza directly to the user.

type

The `type` attribute specifies the specific kind of `<presence>`, `<message>`, or `<iq>` stanza. Each of the three basic stanzas has several possible values for the `type` attribute, and these are explained when each stanza is covered in detail.

All three stanzas may have their `type` attribute set to a value of `error`. This indicates that the stanza is an error response to a received stanza of the same kind. You must not respond to a stanza with an `error` type, to avoid feedback loops on the network.

id

Stanzas may be given an `id` attribute to aid in identifying responses. For `<iq>` stanzas, this attribute is required, but for the other two it is optional. If a stanza is generated in reply to a stanza with an `id` attribute, the reply stanza must contain an `id` attribute with the same value.

The `id` attribute needs to be unique enough that the stanza's sender can use it to disambiguate responses. Often, it is easiest just to make these unique in a given stream to avoid any ambiguity.

Reply stanzas for `<message>` and `<presence>` stanzas are generally limited to reporting errors. Reply stanzas for `<iq>` can signal successful operations, acknowledge a command, or return requested data. In all these cases, the client uses the `id` attribute of the reply stanza to identify which request stanza it is associated with. In cases where many stanzas of the same type are sent in a short time frame, this capability is essential because the replies may be delivered out of order.

Presence Stanzas

The `<presence>` stanza controls and reports the availability of an entity. This availability can range from simple online and offline to the more complex away and do not disturb. In addition, `<presence>` stanzas are used to establish and terminate presence subscriptions to other entities.

In traditional instant messaging systems, presence notifications are the main source of traffic. To enable instant communication, it is necessary to know when the other party is available to communicate. When you send an e-mail, you have no idea if the recipient is currently checking and responding to e-mail, but with instant messages and presence notifications, you know before the message is sent if the recipient is around.

For applications in other domains, presence notifications can be used to signal similar kinds of information. For example, some developers have written bots that set their presence to do not disturb when they are too busy to accept more work. The basic online and offline states can let applications know whether a service is currently functioning or down for maintenance.

Normal Presence Stanzas

A normal `<presence>` stanza contains no `type` attribute or a `type` attribute that has the value `unavailable` or `error`. These stanzas set or indicate an entity's presence or availability for communication.

There is no `available` value for the `type` attribute because this is indicated instead by the lack of a `type` attribute.

Users manipulate their own presence status by sending `<presence>` stanzas without a `to` attribute, addressing the server directly. You've seen two short examples of this already, and these are included along with some longer examples here:

```
<presence/>

<presence type='unavailable'/>

<presence>
  <show>away</show>
  <status>at the ball</status>
</presence>

<presence>
  <status>touring the countryside</status>
  <priority>10</priority>
</presence>

<presence>
  <priority>10</priority>
</presence>
```

The first two stanzas set a user's presence status to online or offline, respectively. These are also typically the first and last presence stanzas sent during an XMPP session.

The next two examples both show extra presence information in the form of `<show>`, `<status>`, and `<priority>` children.

The `<show>` element is used to communicate the nature of the user's availability. The element is named "show" because it requests that the recipient's client use this information to update a visual indicator of the sender's presence. Only one `<show>` child is allowed in a `<presence>` stanza, and this element may only contain the following possible values: `away`, `chat`, `dnd`, and `xa`. These values communicate that a user is away, is interested in chatting, does not wish to be disturbed, or is away for an extended period.

A `<status>` element is a human-readable string that the user can set to any value in order to communicate presence information. This string is generally displayed next to the contact's name in the recipient's chat client.

Each connected resource of a user has a priority between –128 and 127. This priority is set to zero by default, but can be manipulated by including a `<priority>` element in `<presence>` stanzas. Users with multiple simultaneous connections may use this to indicate which resource should receive chat messages addressed to their bare JID. The server will deliver such messages to the resource with the highest priority. A negative priority has a special meaning; resources with a negative priority will never have messages delivered to them that were addressed to the bare JID. Negative priorities are extremely useful for automated applications that run on the same JID as a human is using for regular chat.

Extending Presence Stanzas

It is tempting for developers to want to extend `<presence>` stanzas to include more detailed information such as the song the user is currently listening to or the person's mood. Because `<presence>` stanzas are broadcast to all contacts (even those that may not have an interest in the information) and constitute a large share of the network traffic in the XMPP network, this practice is discouraged. These kinds of extensions are handled by protocols that more tightly focus delivery of this extra information.

Presence Subscriptions

The user's server automatically broadcasts presence information to contacts that have a presence subscription to the user. Similarly, users receive presence updates from all contacts for which they have a presence subscription. Presence subscriptions are established and controlled by use of `<presence>` stanzas.

Unlike some social network and IM systems, presence subscriptions in XMPP are directional. If Elizabeth has a subscription to Mr. Darcy's presence information, this does not imply that Mr. Darcy has a subscription to Elizabeth. If a bidirectional subscription is desired, a subscription must be separately established in both directions. Bidirectional subscriptions are often the norm for human communicators, but many services (and even some users) are interested in only one of the directions.

Presence subscription stanzas can be identified by a `type` attribute that has a value of `subscribe`, `unsubscribe`, `subscribed`, or `unsubscribed`. The first two values request that a new presence subscription be established or an existing subscription be removed, and the other two are the answers to such requests.

The following example shows Elizabeth and Mr. Darcy establishing a mutual presence subscription:

```
<presence from='elizabeth@longbourn.lit/outside'
          to='darcy@pemberley.lit'
          type='subscribe'/>

<presence from='darcy@pemberley.lit/library'
          to='elizabeth@longbourn.lit/outside'
          type='subscribed'/>

<presence from='darcy@pemberley.lit/library'
          to='elizabeth@longbourn.lit'
          type='subscribe'/>

<presence from='elizabeth@longbourn.lit/outside'
          to='darcy@pemberley.lit/library'
          type='subscribed'/>
```

After this exchange of stanzas, both Elizabeth and Mr. Darcy will find each other in their rosters and be notified of each other's presence updates.

Chapter 6 explores a fairly traditional IM application with the ability to establish and remove subscriptions as well as showing contacts' presence statuses.

Directed Presence

The final kind of `<presence>` stanza is *directed presence*. A directed presence stanza is a normal `<presence>` stanza addressed directly to another user or some other entity. These can be used to communicate presence to entities that do not have a presence subscription, usually because the presence information is needed only temporarily.

One important feature of directed presence is that the recipient of the presence information is automatically notified when the sender becomes unavailable even if the sender forgets to notify the recipient explicitly. Services can use directed presence to establish temporary knowledge of a user's availability that won't accidentally get out of date.

You see directed presence in action in Chapter 8 because it is quite important for multi-user chat.

Message Stanzas

As their name implies, `<message>` stanzas are used to send messages from one entity to another. These messages may be simple chat messages that you are familiar with from other IM systems, but they can also be used to transport any kind of structured information. For example, the SketchCast application in Chapter 9 uses `<message>` stanzas to transport drawing instructions, and in Chapter 11 `<message>` stanzas are used to communicate game state and new game moves.

A <message> stanza is fire and forget; there is no built in reliability, similar to e-mail messages. Once the message has been sent, the sender has no information on whether it was delivered or when it was received. In some cases, such as when sending to a non-existent server, the sender may receive an error stanza alerting them to the problem. Reliable delivery can be achieved by layering acknowledgments into your application's protocol (see Message Receipts in XEP-0184 for an example of this).

Here are some example <message> stanzas:

```
<message from='bingley@netherfield.lit/drawing_room'
         to='darcy@pemberley.lit'
         type='chat'>
  <body>Come, Darcy, I must have you dance.</body>
  <thread>4fd61b376fbc4950b9433f031a5595ab</thread>
</message>

<message from='bennets@chat.meryton.lit/mrs.bennet'
         to='mr.bennet@longbourn.lit/study'
         type='groupchat'>
  <body>We have had a most delightful evening, a most excellent ball.</body>
</message>
```

The first example shows a typical <message> stanza for a private chat, including a thread identifier. The second example is a multi-user chat message that Mrs. Bennet has sent to the bennets@chat.meryton.lit room, received by Mr. Bennet.

Message Types

Several different types of <message> stanzas exist. These types are indicated with the `type` attribute, and this attribute can have the value `chat`, `error`, `normal`, `groupchat`, or `headline`. Sometimes the message's type is used to inform a user's client how best to present the message, but some XMPP extensions, multi-user chat being a prime example, use the `type` attribute to disambiguate context.

The `type` attribute of a <message> stanza is optional, but it is recommended that applications provide one. Also, any reply <message> stanza should mirror the `type` attribute received. If no `type` attribute is specified, the <message> stanza is interpreted as if it had a `type` attribute set to `normal`.

Messages of type `chat` are sent in the context of a one-to-one chat conversation. This type is the most common in IM applications, which are primarily concerned with private, one-to-one communication.

The `error` type is used in reply to a message that generated an error. These are commonly seen in response to malformed addressing; sending a <message> stanza to a non-existent domain or user results in a reply stanza with the `type` attribute set to `error`.

A <message> stanza with a type of `normal` has been sent outside the context of a one-to-one chat. This type is rarely used in practice.

The `groupchat` type is used for messages sent from multi-user chats. It is used to disambiguate direct, private messages from a multi-user chat participant from the broadcast messages that participant sends to everyone in the room. A private message has the `type` attribute set to `chat`, whereas a message sent to everyone in the room contains a `type` attribute set to `groupchat`.

The last `<message>` stanza type is `headline`. These types of messages are used mostly by automated services that do not expect or support replies. If automatically generated e-mail had a `type` attribute, it would use a value of `headline`.

Message Contents

Though `<message>` stanzas are allowed to contain arbitrary extension elements, the `<body>` and `<thread>` elements are the normal mechanisms provided for adding content to messages. Both of these child elements are optional.

The `<body>` element contains the human-readable contents of the message. More than one `<body>` element can be included as long as each of them contains a distinct `xml:lang` attribute, and this allows for `<message>` stanzas to be sent with content in multiple languages.

Conversations, like e-mail, can form *threads*, where each message in a thread is related to the same conversation. Threads are created by adding a `<thread>` element to a `<message>` stanza. The content of the `<thread>` element is some unique identifier that distinguishes the thread. A reply stanza should contain the same `<thread>` element as the one it is a reply to.

In IM contexts, among others, there are a few commonly used extensions to the message contents. XHTML-IM, defined in XEP-0071, is used to provide formatting, hyperlinking, and rich media in messages. Chapter 5's microblogging client, Arthur, uses XHTML-IM to provide enhanced message bodies. Another extension, Chat State Notifications (XEP-0085), allows users to notify each other of when they are composing a message or have gone idle. The Gab application in Chapter 6 uses these notifications to provide a nice user experience when one party is typing for a long time; the recipient will have some indication that the other party is still actively engaged in the conversation.

IQ Stanzas

The `<iq>` stanza stands for Info/Query and provides a request and response mechanism for XMPP communication. It is very similar to the basic workings of the HTTP protocol, allowing both get and set queries, similar to the GET and POST actions of HTTP.

Each `<iq>` stanza is required to have a response, and, as mentioned previously, the stanza's required `id` attribute is used to associate a response with the request that caused it. The `<iq>` stanza comes in four flavors differentiated by the stanza's `type` attribute. There are two types of `<iq>` stanza requests, `get` and `set`, and two types of responses, `result` and `error`. Throughout the book these are often abbreviated as *IQ-get*, *IQ-set*, *IQ-result*, and *IQ-error*.

Every IQ-get or IQ-set must receive a response IQ-result or IQ-error. The following examples show some common `<iq>` stanzas and their possible responses. Note that unlike `<message>` and `<presence>` stanzas, which have defined children elements, `<iq>` stanzas typically contain only extension elements relating to their function. Also, each pair of `<iq>` stanzas has a matching `id` attribute.

```
<iq from='jane@longbourn.lit/garden'
    type='get'
    id='roster1'>
  <query xmlns='jabber:iq:roster'/>
</iq>

<iq to='jane@longbourn.lit/garden'
```

```
    type='error'
    id='roster1'>
  <query xmlns='jabber:iq:roster'/>
  <error type='cancel'>
    <feature-not-implemented xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
  </error>
</iq>
```

Jane sent a malformed roster retrieval request to her server. The server replied with an error. Error stanzas are covered in detail later.

```
<iq from='jane@longbourn.lit/garden'
    type='get'
    id='roster2'>
  <query xmlns='jabber:iq:roster'/>
</iq>

<iq to='jane@longbourn.lit/garden'
    type='result'
    id='roster2'>
  <query xmlns='jabber:iq:roster'>
    <item jid='elizabeth@longbourn.lit' name='Elizabeth'/>
    <item jid='bingley@netherfield.lit' name='Bingley'/>
  </query>
</iq>
```

After resending a corrected request, the server replied to Jane with her small roster. You can see that Elizabeth and Bingley are both in Jane's contact list.

```
<iq from='jane@longbourn.lit/garden'
    type='set'
    id='roster3'>
  <query xmlns='jabber:iq:roster'>
    <item jid='darcy@pemberley.lit' name='Mr. Darcy'/>
  </query>
</iq>

<iq to='jane@longbourn.lit/garden'
    type='result'
    id='roster3'/>
```

Jane attempts to add Mr. Darcy to her roster, and the server indicates success with a blank IQ-result. In the cases where the response is simply an acknowledgment of success, the IQ-result stanza will often be empty.

The `<iq>` stanza is quite useful in any case where result data or simple acknowledgment is required. Most XMPP extension protocols use a mix of `<iq>` and `<message>` stanzas to accomplish their goals. The `<iq>` stanzas are used for things like configuration and state changes, whereas `<message>` stanzas are used for regular communication. In some cases `<iq>` stanzas are used for communication because stanza acknowledgment can be used for rate limiting.

Error Stanzas

All three of the XMPP stanzas have an error type, and the contents of each type of error stanza are arranged in the same pattern. Error stanzas have a well-defined structure, often including the contents of the original, offending stanza, the generic error information, and, optionally, an application-specific error condition and information.

All error stanzas must have a `type` attribute set to `error` and one `<error>` child element. Many error stanzas also include the original stanza's contents, but this is not required and, in some cases, not desirable.

The `<error>` child has a required `type` attribute of its own, which can be one of `cancel`, `continue`, `modify`, `auth`, or `wait`. The `cancel` value signifies that the action should not be retried, because it will always fail. A value of `continue` generally indicates a warning and is not frequently encountered. An error type of `modify` communicates that the data sent needs some change in order to be accepted. An `auth` error informs the entity that the action should be retried after authenticating in some way. Finally, the `wait` value reports that the server is temporarily having some problem, and the original stanza should be resent unmodified after a short time.

An `<error>` child is also required to contain an error condition from a list of defined conditions as a child element. It may also contain a `<text>` element giving further details about the error. An application-specific error condition can also be specified in a child element of the `<error>` element under its own namespace.

Table 1-1 lists the most common defined error conditions. For more information on these, please refer to Section 3.9.2 of RFC 3920. Note that each of these condition elements must be under the `urn:ietf:params:xml:ns:xmpp-stanzas` namespace.

TABLE 1-1: Common Defined Error Conditions

CONDITION ELEMENT	DESCRIPTION
`<bad-request/>`	The request was malformed or includes unexpected data.
`<conflict/>`	Another resource or session exists with the same name.
`<feature-not-implemented/>`	The feature requested is not implemented by the service.
`<forbidden/>`	The client does not have authorization to make the request.
`<internal-server-error/>`	The server had an undefined internal error that prevented it from processing the request.
`<item-not-found/>`	The item involved in the request does not exist. This error is equivalent to the HTTP 404 error.
`<recipient-unavailable/>`	The intended recipient is temporarily unavailable.
`<remote-server-not-found/>`	The remote server does not exist or could not be reached.
`<remote-server-timeout/>`	Communication with the remote server has been interrupted.
`<service-unavailable/>`	The requested service is not provided.

The following example IQ-error stanza shows a fully constructed error response to a publish-subscribe related `<iq>` stanza:

```
<iq from='pubsub.pemberley.lit'
    to='elizabeth@longbourn.lit/sitting_room'
    type='error'
    id='subscribe1'>
  <pubsub xmlns='http://jabber.org/protocol/pubsub'>
    <subscribe node='latest_books'
               jid='elizabeth@longbourn.lit'/>
  </pubsub>
  <error type='cancel'>
    <not-allowed xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'/>
    <closed-node xmlns='http://jabber.org/protocol/pubsub#errors'/>
    <text xmlns='urn:ietf:params:xml:ns:xmpp-stanzas'>
      You must be on the whitelist to subscribe to this node.
    </text>
  </error>
</iq>
```

The error's type is `cancel`, indicating that this action should not be retried and the condition `<not-allowed/>` indicates the general failure. The `<text/>` child contains a description of the problem. Finally, the application condition element, `<closed-node/>`, gives the exact application error.

THE CONNECTION LIFE CYCLE

The three stanzas you've learned about can accomplish virtually any task in XMPP when combined properly. However, sending stanzas usually requires an authenticated XMPP session be established. This section describes the other portions of an XMPP connection's life cycle — connection, stream set up, authentication, and disconnection.

Connection

Before any stanzas are sent, an XMPP stream is necessary. Before an XMPP stream can exist, a connection must be made to an XMPP server. XMPP includes some sophisticated support for establishing connections to the right servers.

Typically clients and servers utilize the domain name system (DNS) to resolve a server's domain name into an address they can connect to. E-mail services in particular use mail exchange (MX) records to provide a list of servers that handle mail for a given domain so that one well-known server address does not have to handle every service. E-mail, being an early Internet application, got special treatment in DNS. These days, service records (SRV) are used to provide a similar function for arbitrary services.

The first thing an XMPP client or server does when connecting to another XMPP server is to query the appropriate SRV record at the server's domain. The response may include multiple SRV records, which can be used to load balance connections across multiple servers.

If an appropriate SRV record cannot be found, the application tries to connect to the given domain directly as a fallback. Most libraries also allow you to specify a server to connect to explicitly.

Stream Set Up

Once a connection is established to a given XMPP server, an XMPP stream is started. An XMPP stream is opened by sending the opening `<stream:stream>` element to the server. The server responds by sending the response stream's opening `<stream:stream>` tag.

Once XMPP streams are open in both directions, elements can be sent back and forth. At this stage of the connection life cycle, these elements will be related to the stream and the stream's features.

The server first sends a `<stream:features>` element, which details all the supported features on the XMPP stream. These mostly relate to encryption and authentication options that are available. For example, the server will specify if encryption (TLS) is available and whether or not anonymous logins are allowed.

You don't normally need to know much detail about this stage of an XMPP connection as the many libraries for XMPP development handle this work for you, but the following example shows a typical exchange of `<stream:stream>` elements as well as the server's feature list.

First, the client sends the opening element to the server:

```
<?xml version='1.0'?>
<stream:stream xmlns='jabber:client'
               xmlns:stream='http://etherx.jabber.org/streams'
               version='1.0'
               to='pemberley.lit'>
```

The server replies:

```
<?xml version='1.0'?>
<stream:stream xmlns='jabber:client'
               xmlns:stream='http://etherx.jabber.org/streams'
               version='1.0'
               from='pemberley.lit'
               id='893ca401f5ff2ec29499984e9b7e8afc'
               xml:lang='en'>
  <stream:features>
    <stream:features>
      <starttls xmlns='urn:ietf:params:xml:ns:xmpp-tls'/>
      <compression xmlns='http://jabber.org/features/compress'>
        <method>zlib</method>
      </compression>
      <mechanisms xmlns='urn:ietf:params:xml:ns:xmpp-sasl'>
        <mechanism>DIGEST-MD5</mechanism>
        <mechanism>PLAIN</mechanism>
      </mechanisms>
  </stream:features>
```

From this exchange, you know that the pemberley.org server supports TLS, stream compression via zlib, and several authentication mechanisms.

The XMPP streams set up between two servers look identical except that the top-level namespace is `jabber:server` instead of `jabber:client`.

Authentication

XMPP allows for Transport Layer Security (TLS) encryption, and most clients use this by default. Once TLS support is advertised by the server, the client starts the TLS connection and upgrades the current socket to an encrypted one without disconnecting. Once TLS encryption is established, a new pair of XMPP streams is created.

Authentication in XMPP uses the Simple Authentication and Security Layers (SASL) protocol, and depending on the server involved, can support a number of authentication mechanisms. Normally servers provide plain text authentication and MD5 digest-based authentication, but some servers support authenticating via Kerberos or special tokens.

These same encryption and authentication technologies are also used in many other protocols — e-mail and LDAP are two examples — and common libraries exist for supporting TLS and SASL that can be used equally well for XMPP.

Once authentication is complete, a client must bind a resource for the connection and start a session. If you are watching XMPP traffic on the wire, you will see `<bind>` and `<session>` elements — inside `<iq>` stanzas — being sent to do these jobs. If the client does not provide a resource to bind, the server chooses one for it, usually randomly. Also, the server may alter the user's chosen resource even if the client provides one.

When two servers connect to each other, the authentication steps are slightly different. The servers exchange and verify TLS certificates, or the recipient server uses a *dialback* protocol to verify the sender's identity via DNS.

Disconnection

When users are done with their XMPP sessions, they terminate the sessions and disconnect. The most polite way to terminate a session is to first send unavailable presence and then close the `<stream:stream>` element.

By sending a final unavailable presence, the user's contacts can be informed about the reasons for the user's departure. Closing the stream explicitly allows any in-flight stanzas to arrive safely.

A polite disconnection would look like this:

```
  <presence type='unavailable'/>
</stream:stream>
```

The server then terminates its stream to the client.

SUMMARY

In this chapter, you met the XMPP protocol and learned about its history, use cases, addressing, vocabulary, and the connection life cycle. You've also seen several example XMPP stanzas and learned about the different entities composing an XMPP network.

You should now understand:

- XMPP is an open, standardized protocol, originally developed to replace proprietary IM networks.
- The XMPP protocol is more than a decade old and quite mature.
- XMPP is great for writing IM applications, but it also excels at any task that benefits from exchanging structured messages.
- Servers, clients, components, and plug-ins are all parts of XMPP networks and have their special uses.
- XMPP addresses, called JIDs, resemble e-mail addresses and decompose into three parts: the local part, the domain, and the resource.
- Full JIDs are the most specific addresses for an entity; for example, darcy@pemberley.lit/library.
- Bare JIDs are the same as full JIDs without the resource; for example, darcy@pemberley.lit.
- Servers will handle stanzas to a client's bare JID, potentially routing them to one or more connected resources.
- Stanzas addressed to a full JID are delivered directly to the given resource.
- There are three stanzas in the main XMPP vocabulary: `<message>`, `<presence>`, and `<iq>`.
- The `<message>` stanza is used for fire-and-forget messages between two entities.
- `<presence>` stanzas communicate presence status changes and are used to manipulate presence subscriptions.
- The `<iq>` stanza provides request-response semantics very similar to the GET and POST operations of the HTTP protocol.
- Every XMPP session has a life cycle consisting of several phases: connection, stream set up, authentication, the session body, and disconnection.

The basic concepts and protocol syntax of XMPP are only once piece of the puzzle. You learn about how to use these ideas to design XMPP applications in the next chapter.