Chapter No.7
"Using Common Internet Services"
In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.7 "Using Common Internet Services

Information on where to buy this book

About the Authors

**Wojciech Kocjan** is a programmer having ten years of experience with Tcl. His past experience includes both open source projects, involvement in commercial/ government projects, as well as working for international corporations.

Over the years, Wojciech has contributed and created multiple open source projects related to Tcl. He is also the author of *Learning Nagios 3.0*, Packt Publishing and multiple publications on IT related subjects.

I'd like to thank my wife Joanna and my son Kacper for all the help and support during the writing of this book.

**Piotr Beltowski** is an IT and software engineer, and co-author of several US patents and publications. He has a master of Science in Telecommunications, and his technical experience includes various aspects of telecommunications, such as designing IP networks or implementing support of network protocols. Piotr's experience as Level 3 customer support lead makes him focused on finding simple solutions to complex technical issues.

For More Information:

His professional career includes working in international corporations such as IBM and Ericsson (as a contractor).

I would like to thank all my family and friends, especially Kasia and Monika for all the help and support they gave me during the writing of this book.
Tcl 8.5 Network Programming

Tcl or Tool Command Language is a very powerful, but easy-to-learn dynamic programming language, suitable for a very wide range of uses. Tcl is a mature yet evolving language used by developers, from one man shops to large corporations.

It is commonly used for rapid prototyping, scripted applications, GUIs, and testing. Tcl is used on embedded-system platforms, both in its full form and in several other small-footprinted versions. Tcl is also used for CGI scripting and as the scripting language for the Eggdrop bot.

Tcl is popularly used today in many automated test harnesses, both for software and hardware, and has a loyal following in the Network Testing and SQA communities. According to the Tcl website, many have called it the "best-kept secret in the software industry". It is used for rapid development, truly cross-platform GUIs, flexible integration, elegant networking model, great I18N and thread support—all that and it's easy to learn and easy to deploy!

This book will delve into the network programming capabilities of Tcl: one of the best-kept secrets of the software industry.

What This Book Covers

Chapter 1, Introducing Tcl serves as a quick introduction to the Tcl world. It explains what the Tcl is, how to install it and start using it. It presents some tools for making development easy, such as Eclipse and Komodo. It describes Tcl language basics, such as language syntax, data types, and the important commands. This and the following chapters build a solid base for Tcl programming.

Chapter 2, Advanced Tcl Features focuses on advanced topics. More common commands are introduced. The reader learns how to handle time, namespaces, and use Tcl as an object-oriented programming language. A large part of the chapter is dedicated to file I/O operations. The reader also learns how to create Tcl packages or modules. Event loop and various types of events are also discussed. Threads are introduced in the end.

Chapter 3, Tcl Standalone Binaries teaches you how to break free from the constraint of having the Tcl interpreter installed, and how to build portable, standalone executable applications written in Tcl. You will also learn about the Metakit database concepts and its usage.

Chapter 4, Troubleshooting Tcl Applications focuses on topics related to "fixing" the applications. First, the reader learns how to introduce, configure, and apply logging information to the code. Later, the available tools and ways of debugging both local and remote applications are presented.

For More Information:
Chapter 5, Data Storage discusses topics such as text data storage and processing. Some of these include working with SQL databases (both remote systems such as MySQL and Postgresql, and local ones like SQLite), processing XML data (parsing and generation), handling plaintext data while focusing on common encoding issues.

Chapter 6, Networking in Tcl presents basic information about the TCP/UDP networking abilities of Tcl, from both the client and server sides. Core concepts and commands are discussed extensively along with examples.

Chapter 7, Using Common Internet Services teaches you how to send e-mails from Tcl, build e-mails containing HTML and various kinds of messages. It discusses how to retrieve, parse, and process e-mail messages, and also covers how to get data over HTTP, read RSS feeds, and upload and download files over FTP.

Chapter 8, Using Additional Internet Services introduces the reader to additional Internet protocols: querying information using DNS and querying the current date and time over an NTP service. It also discusses LDAP and how to use it to query information about user registries and how to authorize users. It describes the comm protocol, which can be used to communicate between local and remote Tcl applications. It presents how the protocol can be used and how the comm commands can be made secure.

Chapter 9, Learning SNMP describes SNMP (Simple Network Management Protocol) and its use for management of multiple types of devices – such as routers, switches, modems, and servers. It introduces the reader to how data is structured in SNMP by describing MIB (Management Information Base). It describes Scotty package, which provides SNMP functionality to Tcl. Step-by-step instructions are given for retrieving information and retrieving notifications from devices. It also describes how Tcl can be used to create custom SNMP agents using the Scotty package. This chapter also describes ICMP-related features; performing ping and trace routes.

Chapter 10, Web Programming in Tcl introduces Tclhttpd: a Tcl-based web server that can be used standalone and embedded in other applications. Using it for providing static files and web applications is described in detail. Built-in templating system is shown as well. This chapter also gives information on other subjects, using CGI scripts, handling specific types of files from Tcl, authorizing requests, and creating custom template systems.

Chapter 11, TclHttpd in Client-Server Applications provides details on creating HTTP-based client-server applications. It gives the reader a good understanding of how such applications are created, by providing a step-by-step example of building such an application. It also discusses building the client and the server as standalone binaries and providing automatic updates of the client. This example is then extended to show how modules can be added to allow additional code to be deployed to clients on demand.

Chapter 12, SOAP and XML-RPC covers creating and using web services from Tcl. It describes how to invoke methods over XML-RPC, along with an example of how this
can be done to automatically add blog posts. SOAP is also described. This chapter shows how to create an SOAP-based web service and how such services can be used from within Tcl.

Chapter 13, SSL and Security focuses on the security aspects of network programming. It shows the reader how to secure communications and ensure that applications or users can authenticate remote systems. Step-by-step instructions are given on how to create a public key infrastructure and securely transfer keys to systems in the network. The chapter also introduces the reader to Tcl's safe interpreter feature and how it can be used to create a role-based security model for running unsecure Tcl code.

For More Information:  
By now we know that Tcl has the functionality to create networked applications. We also know how to build simple client-server communication. However, as with any programming language, being able to plug into existing protocols and communicate using standards that other systems are using as well is very important. This chapter talks about various standards, how Tcl fits into these technologies, and how to use Tcl for various types of protocols and data encoding.

First, we'll show how to manage files remotely over **File Transfer Protocol (FTP)**. This can be used as the base for many operations, such as remote copies and filesystem synchronization. We'll also learn how to download a file from a website using **Hypertext Transfer Protocol (HTTP)**, which can then be used to get content from the Internet. This knowledge will also be valuable when we move on to embedding and using web servers in our application in the chapters that follow.

We will also talk about communicating with users using e-mail, starting with sending simple e-mails, followed by learning how to send attachments and multipart types. Using this we'll show how to send and receive e-mails in order to do tasks such as sending information to users and/or customers. You will also learn how to read e-mails in your Tcl code, parse them, and how simple mechanisms can be built to create an automated e-mail management system.

It's worth noting that whenever we're doing checks of any data retrieved from remote systems, we should check if all conditions are satisfied instead of checking for known problems; for example, we should check if the status is one of the acceptable status values instead of checking if it is one of the known error statuses. Troubleshooting Tcl applications and more recommendations for making applications less error prone are described in more detail in Chapter 4, *Troubleshooting Tcl Applications.*

For More Information:  
Handling e-mail

Before working with e-mail, we need to understand a bit about how e-mail works, as well as what it provides and what our application needs to perform on its own. In general, e-mails are easy to understand — someone sends a message, the e-mailing system takes care of delivering it to the correct target machine(s), and the recipients are then able to retrieve that message. From the e-mail system's perspective, it does not care about what the e-mail contains, as long as it knows who it is from and who it should be delivered to.

From the user's perspective, he/she does not need to know how it is delivered—their mail application delivers the message to the server handling their e-mail, and all messages can be retrieved from that same server. When we interact with e-mails, it works the same way for us. In the majority of cases, our application only needs to interact with our e-mail server.

All e-mail messages are built using a common structure—each message consists of headers that describe the message and the body. Headers describe who the message is from, its recipients, and the subject of the message. They also provide the content type, which tells e-mail applications what type of data the message contains. Message headers can also contain a history of the servers it passed, additional information such as the e-mail application used to generate this message, and any other information that the e-mail application has added. The message body is the actual text and/or data that was sent. Information about what is in the message body is described in the headers, for example we can send plain text, HTML message, or simply an image.

Learning MIME

Multipurpose Internet Mail Extensions (MIME) is a standard that extends the e-mail format. It defines how messages can include character sets other than 7-bit ASCII in the message headers and body, and introduces the concept of multiple parts of an e-mail along with attachments. Over time, MIME became such an integral part of e-mail handling that all e-mails are now sent in accordance with MIME standards.

Content type

MIME introduced the concept of content type, which was originally meant for defining types of files in an e-mail. This was introduced so that e-mail applications could present the content of a message differently, depending on the actual file type. This grew to other protocols and can now be referred to as the Internet media type standard. The standard consists of two parts—MIME type and MIME subtype separated by a slash. The content type describes the type of a piece of media, for example, image. The subtype defines file format—for example, jpeg. In this example, the MIME type is image/jpeg.

For More Information:
A full list of standardized format types can be found on the following page:
http://www.iana.org/assignments/media-types/

Whenever an application needs to use its own content type, it is recommended that an x- prefix is appended to the subtype—for example, application/x-tcl-dict could be used to transmit a dictionary’s contents.

The MIME standard defines several possibilities for embedding data that is outside a 7-bit ASCII character set, that is, data such as binary files, messages using different character sets, and so on. The Base64 standard is commonly used for encoding binary files within an e-mail—this standard uses 64 characters only, and requires 4 bytes to encode 3 bytes of actual data. This means that a 1M file will use up over 1.3M when sent via e-mail. Base64 is described in more detail at:
http://en.wikipedia.org/wiki/Base64

The standard also defines the quoted-printable standard that is used for sending 8-bit data. Characters outside of 7-bit character set are encoded as multiple characters; this idea is described in more detail at:
http://en.wikipedia.org/wiki/Quoted-printable

For the purpose of this book, we do not need to go into details of how both Base64 and quoted-printable encodings work.

**Multipart messages**

MIME also introduces the concept of multipart content. An e-mail message can only consist of a single item. However, the MIME standard provides ways to send multipart content by enclosing multiple items in a single message. It can also be used recursively, one of the elements can also contain additional parts. We’ll see this in the following example:

There are multiple types of multipart contents:

- `multipart/related` is used to send messages that should be treated as a whole. The first part is the content that the e-mail application should use and other parts are related to it, for example, images that are used in a HTML message. However, adding a part that should be inline requires that this element also has specific headers, which is discussed in more detail later in this chapter.

- `multipart/mixed` is used for sending mixed content types. It is up to the e-mail application to decide how to handle this, but parts that it can show inline will be shown within e-mail application, and parts that it cannot show directly will be shown only as attachments. A typical example is attaching images and documents—e-mail applications will show images inline, but require documents to be opened in an external application.
Using Common Internet Services

- multipart/alternative is used to define multiple parts, where each part is an alternate version of the same content. A typical example is sending plain text and HTML messages. E-mail applications choose the best format that they can handle. Representations should be sent in a way they are ordered by—preferable representation should be the last part.

Multipart content types allow each part to have its own individual headers—this is required in order to define which content type each part is, along with how it should be treated. Also, as each part can have its own type, each part can also be a multipart element on its own.

The following diagram illustrates how both multipart/mixed, multipart/alternative, and multipart/related can be used to send e-mail that contains plain text message and HTML message, inlined images as well as attachments. This is actually how the majority of e-mail applications will embed such a message. The structure of the entire message would look as follows:

```
<table>
<thead>
<tr>
<th></th>
<th>E-mail message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E-mail headers</td>
</tr>
<tr>
<td>1.1</td>
<td>E-mail body: content type: multipart/mixed</td>
</tr>
<tr>
<td>1.2</td>
<td>Contents of the message; content type: multipart/alternative</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Plain text version of the message; content type: text/plain</td>
</tr>
<tr>
<td>1.2.1.1</td>
<td>Container for HTML version of the message; content type: multipart/related</td>
</tr>
<tr>
<td>1.2.1.2</td>
<td>HTML message – actual HTML; content type: text/html</td>
</tr>
<tr>
<td>1.2.1.2.1</td>
<td>HTML message – inlined image; content type: image/gif</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Attached image; content type: image/jpeg</td>
</tr>
</tbody>
</table>
```

Now that we know how our e-mail might appear, let's proceed to building such a structure from within Tcl.

**MIME in Tcl**

In order to send such an e-mail from within Tcl, we will need to use Tcl's mime package. It is a part of the tcllib package and is available in ActiveTcl distributions of Tcl.

This package allows the building and parsing of messages and handles all aspects of a message—headers, content, and support for multipart messages. It also handles conversion between various content encodings such as base64 and quoted-printable. Thanks to this we'll only need to build the message parts and combine them into a final message.
Creating messages

The command `mime::initialize` is used to set up a part or the content of the entire message. This command accepts one or more options and returns a new token that identifies the new MIME part. Based on the options specified, there are two modes in which it can be used—the first is to parse content (such as parse a received e-mail message), and the second is to create content. We will focus on the second case and leave parsing for sections that talk about receiving e-mail.

Whenever we want to create a MIME part we need to specify the `-canonical` option and provide the content type for this part. Type is the MIME type described earlier. There are three possibilities for creating MIME objects—from a file or from a string, and when creating multipart content.

- To create it from a file or a string, we need to specify the `-file` or `-string` option and provide either the path to the file or the content of this part as string or binary data. We should also specify the `-encoding` option that states how content should be handled so that it can be passed over a 7-bit protocol such as SMTP. For binary files, we should usually use base64 encoding and for text files, it is best to use quoted-printable.

- When creating a MIME part, we can also specify one or more headers that it should have by adding the `-header` option. This option can be specified multiple times and each parameter to this option should be a list containing a header name and corresponding value. These headers are then added to the actual MIME body. Their names and corresponding values are part of MIME's specifications. We'll cover a small subset that we need to know in order to send an e-mail with both inlined elements and attachments.

For example, in order to create a simple plaintext element, we can run the following command:

```tcl
set token [mime::initialize -canonical "text/html" "Hello world!"
```

If we want to send it, all we would need to do is use the `smtp` package:

```tcl
smtp::sendmessage $token -recipients "someone@domain.com"
```

Sending e-mails is described in more detail later in this section—the preceding code simply shows that both packages can be combined very easily.
Multipart elements
In order to multipart content, we should provide the -parts option to the mime::initialize command. The value for this option should contain a list of all parts that should be included in this multipart content. Parts are included in the same order as provided in the list.

Let’s walk through an exercise of building up an e-mail that we described earlier.

This code uses several files, mainly message.html and message.txt for the text of the e-mail, companylogo.gif for logo that is used in message.html, and attachment.jpg as an attachment.

The files as well as the complete source code for the example in this section are located in the 01mime directory in the source code examples for this chapter.

First we have to load the mime package and create an HTML part:

```tcl
package require mime

# create actual HTML part
# (1.2.1.2.1 from diagram)
set part_html [mime::initialize -canonical "text/html" \
   -encoding quoted-printable -file message.html]

# create logo as inlined image
# (1.2.1.2.2 from diagram)
set part_logo [mime::initialize -canonical "image/gif" \
   -encoding base64 -file companylogo.gif \
   -header [list Content-Disposition "inline"] \
   -header [list Content-ID "companylogo.gif"] \
   ]
```

This code builds up two elements—a part containing HTML version of the message and an image that we add, inline, in the message. Following that, we use these to build up the multipart/related part (element 1.2.1.2 from preceding diagram) that contains two elements created using the preceding code:

```tcl
set part_htmlrelated [mime::initialize \
   -canonical multipart/related \
   -parts [list $part_html $part_logo]]
```

Next it’s time to create a plain text version of the e-mail (element 1.2.1.1 from diagram) and build the multipart/alternative element that binds the HTML message and the plain text message into one piece, which is element 1.2.1.

```tcl
set part_txt [mime::initialize \
   -canonical "text/plain" \
   -encoding quoted-printable -file message.txt]
```

For More Information:
Finally, we create a part for the attachment (element 1.2.2 from diagram) and create an element that combines the previously created container for the plain text and HTML message along with the attachment—element 1.2 from diagram.

```
set part_attachment [mime::initialize \
 -canonical "image/jpeg" \
 -header [list Content-Disposition \n "attachment; filename=attachment.jpg"] \n -header [list Content-ID "attachment.jpg"] \n -encoding base64 -file attachment.jpg
```

```
set all [mime::::initialize -canonical multipart/mixed \n -parts [list $part_alternative $part_attachment]]
```

This makes our code complete and a full version of the message is now ready.

There are three types of elements that we are building:

- HTML and plain text messages: Their context is defined by multipart elements they are included in, therefore, we only need to define content type.
- JPEG image: It is an attachment, therefore, we need to provide more information in the part headers—filename, Content-ID, and disposition.
- Multipart elements: These are used to combine other types of elements into a structure that we've described earlier.

Sending text messages also relates to character sets, encodings, and issues with internationalization. When sending messages that contain characters outside of 7-bit ASCII, we need to be aware of two things.

First of all, Tcl sends both strings and file contents in binary form. If we want to send text from a file, then that file needs to be encoded properly, using encodings such as UTF-8. If we want to send text from Tcl, we need to convert that text to proper encoding. Secondly, we need to specify the encoding of a part when specifying the canonical type—usually this means appending a semi-colon and charset=<<charsetName>>. For example:

```
set part_html [mime::initialize -canonical \n "text/html; charset=UTF-8" \n -encoding quoted-printable -string [encoding \n convertto utf-8 "\u25ba Omega symbol: \u2126 \u25c4"]]
```

For More Information:
Using Common Internet Services

The previous example will cause our e-mail message to contain the text ► Omega symbol: Ω ◄— in our example, \uxxxx means the Unicode characters for ►, Ω, and ◄ accordingly.

The command encoding convertto converts text to specified encoding and is described in more detail in Chapter 4. For more information about standardized character encodings, please refer to: http://www.iana.org/assignments/character-sets.

Next we have an inlined image—in this case, we need to define additional headers. The first header is Content-Disposition, which specifies how this part should be handled. Specifying inline means that this is an element that will be referenced from the main document and should not be shown as an attachment. The second header is Content-ID, which identifies and names an element. This is how an element can then be referenced from other parts. Any references should be made in the format of cid:<Content-ID>, so in our case, it would be cid:companylogo.gif. For example, our message.html file can contain the following HTML tag:

```
<img src="cid:companylogo.gif" width="400" height="40" />
```

Elements that are regular attachments should have Content-Disposition set to attachment. Also, it is recommended to add filename=<name> to this parameter, separated from the disposition type by a semi-colon. Content-ID in this case specifies an attachment name and should be the same as the filename specified in the Content-Disposition header. This is how the attachment.jpg file is sent.

There is also a difference between naming parts within an e-mail and actual filenames. However, this example names files from MIME's perspective in the same way as files are named on disk. It is common to add prefixes and/or suffixes to avoid naming collisions, especially when a message contains parts from different sources.

For example, we add create inlined image in the following way:

```
set part_logo [mime::initialize -canonical "image/gif" \
-encoding base64 -file "/path/to/template/logo.gif" \
-header [list Content-Disposition "inline"] \
-header [list Content-ID "template.logo.gif@$messageId"] ]
```

We can then build the HTML to include such an image from Tcl by doing something like:

```
set html "<img src="cid:template.logo.gif@$messageId" />
```

For More Information:
It is a good idea to generate unique identifiers for each message and append them to inlined parts' identifiers. This prevents poorly written e-mail applications from having issues with forwarding or replying to e-mails with such images. It can be done using the `uuid` package and the `uuid::uuid` generate command, but any mechanism for generating a unique ID, such as from a related database entry, will work.

Cleaning up a MIME item requires running the `mime::finalize` command and passing the token of a MIME part to it. In order to delete all elements that are used in that element recursively, we can add the `-subordinates` option with the value `all`. For example:

```tcl
mime::finalize $all -subordinates all
```

The preceding code will delete the token created for the entire message along with all other elements we've created.

Information about all commands from `mime` package can be found in its documentation available at:

http://tcllib.sourceforge.net/doc/mime.html

## Sending e-mails

Now that we're able to build MIME parts and the contents of messages, we can send our message. Messages are sent over the **Simple Mail Transfer Protocol** (SMTP), which is the most popular standard for the delivery of e-mail messages. This is the protocol that almost all e-mail applications use to send e-mail and all service providers give you an SMTP server to send messages through.

From the Tcl perspective, we need to use the `smtp` package. This package uses the same tokens as MIME to send messages, which means that if you have created a message using the `mime` package, sending it requires only a few lines of code.

The package `smtp` offers a single command—`smtp::sendmessage`. This command requires passing a token to MIME part and accepts additional options. The first one is `-servers`, which is a list of SMTP servers to try. If this is not specified, then message is sent using the SMTP server on the local machine. If the machine our code is run on has an SMTP server, it is okay to not specify this option. If our SMTP server is using a non-standard port, the option `-ports` can be used to specify which ports we should try to connect to. For the majority of users, it is not necessary to use this flag.
Using Common Internet Services

We can also specify additional headers that should be added when building this e-mail by adding the header option in one or multiple times. Similar to the mime package, it accepts a list where the first element is the header name and the second is the value. Usually when sending an e-mail, we want to set the From, To, and/or the Cc and Subject header fields. For fields such as To and From, we need to specify addresses in a specific way. If we want to specify both the full name and e-mail address, it should be in the form of "[Full Name]" <email@address>. If only the e-mail address is to be passed, it should be in the form of <email@address>. Multiple items should be separated by a comma succeeded by a space. For example:

```tcl
smtp::sendmessage $token -header \
    [list To "<someone@domain.com>, <another-address@domain.com>"]
```

We should pass the -originator option in order to specify address from which an e-mail is sent. Specifying who the message should be delivered to can be done using the -recipients option. If -originator is not passed, the value is taken from the From or Resent-From header field. If -recipients was not specified, the value is taken from the To, cc, Bcc, or Resent-To header fields and is concatenated to one list. Please note that only headers specified to smtp::sendmessage are parsed, not headers set in the MIME token provided to the command.

Based on the previous example, we can use the message we have just created and add to it the code that will send it:

```tcl
package require smtp

smtp::sendmessage $all -recipients $recipient -originator $sender -header [list Subject $mailsubject] -header [list From "\"$senderName\" <$sender>"] -header [list To "\"$recipientName\" <$recipient>"] -servers $smtpservers
```

And a sample initialization of configuration variables needed for this code is as follows:

```tcl
set sender sender@localhost
set senderName "My Tcl Application"
set recipient recipient@localhost
set recipientName "My Tcl Application User"
set mailsubject "Screenshot of Eclipse and DLTK"
set smtpservers 127.0.0.1
```

For More Information:
The complete example along with code from the previous example that creates all MIME elements is located in the `02smtp` directory in the source code examples for this chapter.

If our SMTP server requires authentication, we will need to pass this information to the `smtp::sendmessage` command. The options `-username` and `-password` can be used to specify credentials and if server requires authentication, the `smtp` package will use these. Usually authentication is needed if your e-mail server is being accessed remotely. In these cases, the username and password are provided by your ISP or network administrators.

**Automated e-mail sending**

Now that we know how to send an e-mail from start to end, we can move on to more advanced issues. Often our application will just send e-mails sporadically. However, there are many applications that mainly send e-mails. Many companies run their business by sending customized information to their customers and need a robust system that can send large volumes of e-mails, which handles failures of e-mail servers and needs to be customizable from both business logic and template perspectives.

If we plan on sending e-mail messages, we need to automate the process of building them and creating a system that will allow other people to customize both content and the appearance of our e-mails. We can either create a set of standalone tools or libraries that we can then use in our applications, depending on our needs.

One of the most important things that such an application will offer is an easy way to change contents of messages easily. In many cases, it is enough to have a separate directory for content of a message along with standardized file naming.

**Creating template based e-mails**

For example, let's assume our messages are stored in a separate directory. Files called `message.txt` and `message.html` will be used for body of the message. The subject of the message is in the `subject.txt` file. All files starting with the `inline_` prefix will be included and contained within `multipart/related` along with a HTML version of the message. We'll also assume our messages are in UTF-8 encoding, which allows support for various encodings around the world.

For More Information:

We'll reuse some of the ideas from the previous code sample and make it more generic. Let's start with creating a namespace, and create a procedure called readText for reading text files. This procedure will also use the subst command to allow substitution of commands and variables. We'll also load packages that we'll use throughout this example and initialize the proper namespace, which is needed before creating any procedures within that namespace:

```tcl
namespace eval emailtemplate {}

package require mime
package require uuid
package require fileutil::magic::mimetype

proc emailtemplate::readText {filename messageId data} {
    set fh [open $filename r]
    fconfigure $fh -encoding utf-8 -translation auto
    set text [read $fh]
    close $fh
    set text [subst $text]
    return $text
}
```

This procedure also requires the messageId variable to be passed, which we'll later use to add inline images in the correct way. Our code reads a file as UTF-8, runs subst, and returns result of that command, which causes any occurrence of [command] or $variable to be evaluated as if it was part of Tcl code.

For example, we can add this to our HTML part of the message:

```
Welcome <b>[dict get $data firstname]</b>,
```

This causes value of firstname key from $data dictionary to be shown in bold.

Variable $data is passed to emailtemplate::readText command in the first place. We can use it for passing data to our templates as dictionaries. For example, if we were to send a notification that a payment is due, data would contain first name, last name, and other information about the recipient, and the amount and payment date. Our template would then include these values in the text by using dict get command as seen in this section.

We've loaded the uuid and fileutil::magic::mimetype packages. The first one is used to generate the message's unique identifier, whereas the second one is used to get the MIME content type from files, so that these don't have to be specified explicitly by e-mail template providers.
Next we'll create a `buildMessage` procedure that builds a MIME part containing each element based on directory where the template is placed and provided data. It also accepts an optional list of attachments, which allows us to add additional files, such as invoices as PDF files, if this is needed.

```tcl
proc emailtemplate::buildMessage {directory data {files ""}} {
    set messageId [uuid::uuid generate]
    
    The following steps are performed as part of `emailtemplate::buildMessage`:
    
    The first step is to read the subject and both versions of messages. We'll also make sure that the subject does not include any newline characters as this will create errors when either building or parsing the message. We do this by splitting the subject into a list of lines and choose the first element from that list.
    
    ```tcl
    set subject [readText \
        [file join $directory subject.txt] $data]
    set msgtext [readText \
        [file join $directory message.txt] $data]
    set msghtml [readText \
        [file join $directory message.html] $data]
    
    set subject [lindex [split $subject \n] 0]
    ```
    
    After we've read the subject and both types of messages, we now build the MIME parts for these. Please note that, at this point, our messages are already formatted using the data that was provided. This step is very similar to how it was previously built for our first message.
    
    ```tcl
    set part_text [mime::initialize \
        -canonical "text/plain; encoding=UTF-8" \
        -string $msgtext -encoding quoted-printable]
    
    set part_html [mime::initialize \
        -canonical "text/html; encoding=UTF-8" \
        -string $msghtml -encoding quoted-printable]
    ```
    
    The third step is to build a list of parts for the `multipart/related` element that contains an HTML message along with sending images inline. We create a list of parts and add only one element to it for now—the actual HTML part.
    
    ```tcl
    set parts_related [list $part_html]
    ```
We then list all files in the template directory beginning—matching inline_* pattern. The variable file contains full path to the file and filename contains just the filename.

```tcl
foreach file [glob -directory $directory
    -nocomplain -type f inline_*] {
    set filename [file tail $file]
}
```

For each of these files, we'll find the proper MIME content type to use and add it to the list of items. We specify that this item should be added inline and that its identifier is <filename>@<messageId>.

```tcl
set ftype [fileutil::magic::mimetype $file]
lappend parts_related [mime::initialize \  
    -canonical $ftype -encoding base64 -file $file \ 
    -header [list Content-Disposition "inline"] \ 
    -header [list Content-ID "$filename@$messageId"] \ 
    ]
}
```

Finally, we'll build a multipart/related element by combining HTML along with the inline images and add this to a multipart/alternative element so that the e-mail application reading this message can choose between plain text or HTML.

```tcl
set part_related [mime::initialize \ 
    -canonical "multipart/related" \ 
    -parts $parts_related]
set result [mime::initialize \ 
    -canonical "multipart/alternative" \ 
    -parts [list $part_text $part_related]]
```

The steps that follow take place only if this procedure has been passed any attachments. If this is the case, we create a new list of items for multipart/mixed. We initialize it to just one element—the result from previous step, the complete message containing plaintext, HTML, and images added inline.

Now we iterate through the list of attachments, taking two elements from the list at a time. The first one is the path to attachment file and the second one is the name that will be shown to the recipients.

```tcl
if {([llength $files] > 0)} {
    set elements [list $result]
    foreach {file name} $files {
```
For each of the attachments, we guess its MIME content type, and build its MIME element and add that to the list of elements.

```tcl
set ftype [fileutil::magic::mimetype $file]
lappend elements [mime::initialize 
    -canonical $ftype 
    -encoding base64 -file $file 
    -header [list Content-Disposition 
        "attachment; filename=$name"] 
    -header [list Content-ID "$name"] 
]
```

Finally, we create a final multipart/mixed element that will contain the message and all attachments.

```tcl
set result [mime::initialize 
    -canonical "multipart/mixed" 
    -parts $elements]
```

The last thing is to set the Subject header to indicate the subject that was provided by the template and return the element back to the caller of this procedure.

```tcl
mime::setheader $result Subject $subject
```

```tcl
return $result
```

This provides a complete mechanism for sending messages based on templates.

Setting the Subject header before sending the message is different from the previous example. Although the From, To, Cc and Bcc fields should be set during the smtp::sendmessage step, Subject and all other headers that are not related to message delivery can be set before sending the message.

Now that we have completed our code for preparing the messages, we can proceed with adding a small piece of code to send it:

```tcl
set data [dict create firstname "John" lastname "Doe"]
set token [emailtemplate::buildMessage "template1" $data]
smtp::sendmessage $token \
    -recipients $recipient -originator $sender \ 
    -header [list From "\"$senderName\" <$sender>"] \ 
    -header [list To "\"$recipientName\" <$recipient>"] \ 
    -servers $smtpservers
```
This code is based on the same variables as the previous example. We're also passing two values to the message—firstname and lastname. If we want to send an attachment, all we need to do is add such a list to the emailtemplate::buildMessage invocation similar to the following example:

```tcl
set token [emailtemplate::buildMessage "template1" $data \ 
[list "attachment/attachment.jpg" "screenshot.jpg"]]
```

The directory template1 is the location where the template of the e-mail is kept. It has to contain subject.txt and message.txt, and message.html which are the subject and content of the message in plain text and HTML respectively. The directory can also contain files that should be embedded as part of the message.

The list provided as the last argument is the list of attachments to include. While the actual file is called attachment.jpg, the user will actually see it named as screenshot.jpg. It is common to use this approach, for example, when file names are unique identifiers of the content, such as customerinvoice_0013_0143.pdf while we want the user to see a more readable filename such as Invoice.pdf.

The example in this section is located in the 03emailtemplate directory in the source code examples for this chapter. It also contains a sample template that uses data variables for passing information.

Documentation on mime and smtp packages can be found on the tcllib project website at SourceForge:

**Receiving e-mails**

In many cases, apart from sending e-mails, our application should also be able to receive messages sent via e-mail. One example is that if we send out mass e-mails to users, we might also want to be able to receive messages for unsubscribing from them and process them. Also, our application can also receive "delivery failed" notifications and process them—for example, starting with temporarily not sending messages and ending with removing failing e-mail addresses from our records permanently.

One of the most flexible options is to set up a dedicated SMTP server that will accept e-mails from other systems. This option, however, requires more setup and is discussed in the next chapter.

A simpler option that will allow us to receive e-mails is to use existing servers and read e-mails from them. This is similar to how e-mail applications work—the e-mail is stored on a remote server and you receive and/or delete it whenever your computer is online.
For this purpose, we'll use the **Post Office Protocol version 3 (POP3)**, which all e-mail servers offer. In order to read e-mail over POP3, we need to know the POP3 server name, and the username and password required to access our account. Usually, this is provided by our system administrators. For the majority of Unix servers, the hostname, username, and password are the same as for accessing it over SSH, Telnet, FTP, or any other protocol. POP3 works so that applications connect, list and read messages, delete the ones they have already read, and then disconnect. This is then repeated periodically.

POP3 uses positive integers to identify all messages. These messages range from 1 to the number of messages — therefore, if our inbox has 23 messages, they are 1, 2, 3, ..., 23 accordingly. Throughout the entire connection to a POP3 server, each number is guaranteed to point to the same message. Therefore, until we disconnect, message 19 will always point to the same message.

If we delete a message, we can no longer reference that particular message, but all other messages will still have the same index — for example, if we delete message 4, messages from 5 and up will still have the same identifiers. Also, when new messages arrive during our connection to the server, our connection will not be able to access them.

POP3 is designed so that the application logs in, reads messages, and disconnects. Our program might be doing this frequently, but it should always set up a new connection to the server.

## Using POP3 protocol in Tcl

Tcl offers a `pop3` package that allows you to retrieve e-mails from POP3 servers. Unlike SMTP, it is not integrated with the MIME package, but it can easily be used along with MIME to process messages.

In order to initialize a connection, we need to run the `pop3::open` command by providing it with the hostname, username, and password. Optionally, we can specify the TCP port to connect to as the next argument, otherwise the POP3 default port 110 is the used. This command returns a channel that is used by all other commands from this package.

For example, the following will connect to your Gmail account:

```tcl
set h [pop3::open "pop.gmail.com" "user@gmail.com" "password"]
```

After that variable `h` will keep a handle, which we can use to issue commands later on. This token is used by all other commands from the `pop3` package and is always passed to these commands as the first argument.

---

For More Information:

We can also provide this command with several options, which should be put before the host name. The first one is `-socketcmd`, which allows us to specify the command that should be used for creating the socket. Its main use is to support **Secure Socket Layer (SSL)** and for providing a command that connects using a proxy server if this is needed. While SSL is described in detail in *Chapter 12*, it is worth mentioning that if we are using SSL, we also need to specify the port to use, which is 995 for POP3 over SSL. For example:

```tcl
set h [pop3::open -socketcmd tls::socket "pop.gmail.com" 
   "user@gmail.com" "password" 995]
```

Another flag is `-msex`, which takes a Boolean value and specifies whether the server we are communicating with is a Microsoft Exchange server. If this is the case, the POP3 package needs to work around some non-standard behavior that occurs when communicating with a Microsoft Exchange server.

Having the channel to a mail server open, we can now retrieve information. The command `pop3::status` can be used to find out about the total number of messages and bytes used, and requires passing only the POP3 channel. It returns two numbers — the first one being number of messages in your inbox and the second one specifies the total number of bytes used by all messages. For example:

```tcl
lassign [pop3::status $h] numMessages bytesUsed
```

This will retrieve the status and assign it to `numMessages` and `bytesUsed` variables. The command `lassign` is described in more detail in *Chapter 1*. The number of messages specifies both how many messages are present in our inbox as well as the maximum value that we can use for specifying message identifier.

### Listing all messages

We can also list all messages — by receiving either their sizes or their unique identifiers. The command `pop3::list` returns a list of messages that we can access as a list. The list consists of two elements for each message — its identifier (integer number) and size in bytes. For example, in order to list size of all messages, we can run the following:

```tcl
foreach {id size} [pop3::list $h] {
   puts "Message $id has $size byte(s)"
}
```

The command `pop3::uidl` works similarly, but returns the unique identifier of a message instead of the size. For example:

```tcl
foreach {id uidl} [pop3::uidl $h] {
   puts "Message $id is $uidl"
}
```
Both `pop3::list` and `pop3::uidl` can be run by specifying message number—in this case, data is returned for a single message only.

Please note that unique identifiers might not be supported by all e-mail servers and this command might return an error in this case. Otherwise, these identifiers allow for identifications of e-mails across POP3 sessions. The POP3 standard does not define the format of these identifiers, so our application should not assume anything about it and about the ways of comparing identifiers.

Unique identifiers combined with a persistent storage can be used to track which messages are new and should be processed, and which ones our application has already processed. This is especially useful if our application does not delete messages after receiving them.

**Retrieving e-mails using POP3**

Once we know which messages we want to retrieve, we can either retrieve just part of the message or the entire message. In order to retrieve headers and part of the message, we can use the `pop3::top` command. It accepts the POP3 channel, the number of the message, and the number of lines from the actual message to retrieve. For example, in order to retrieve the headers and 5 topmost lines from the first message, we can run the following:

```tcl
puts [pop3::top $h 1 5]
```

Retrieving an entire message or a group of messages can be achieved by using the `pop3::retrieve` command, by specifying the channel name as the first argument and the index of message as the second argument. This command always returns a list of the messages, even if only one index was specified. For example, we can retrieve and print the entire first message by running:

```tcl
puts [lindex [pop3::retrieve $h 1] 0]
```

We can also specify two indexes—the first being the start index and the second being the end index. In this case, the command will retrieve multiple messages and return a list containing them. For example, we can retrieve messages 1, 2, and 3 by running:

```tcl
pop3::retrieve $h 1 3
```
We can also specify special values for the start and/or end indexes. The special index start means the first message and is the same as 1. The special index end specifies the last message, based on how many messages are present on the server. For example:

```tcl
set id 0
foreach msg [pop3::retrieve $h start end] {
  incr id
  puts "Message $id has [string length $msg] bytes"
}
```

This will print out the size of each message. The same values can be retrieved using the `pop3::list` command, because in both cases, the message is handled in a binary way—without encoding and/or newline translations.

Deleting one or more messages can be done using the `pop3::delete` command. It accepts the channel name, followed by one or two indexes. Similar to `pop3::retrieve`, if a single index is specified here, then only this message is deleted. If two indexes are specified, then multiple messages are deleted. The special keywords start and end can also be used. For example, deleting all messages can be done by invoking:

```
pop3::delete $h start end
```

As the POP3 protocol does not support retrieving or deleting multiple messages at a time, internally, both `pop3::retrieve` and `pop3::delete` iterate through the specified range.

When we have finished working with our remote e-mails, we should close the connection. The command `pop3::close` is used for this. The only argument it needs is the POP3 channel name. For example:

```
pop3::close $h
```

### Parsing incoming e-mails

After retrieving our messages, we can parse them to properly handle their content. This can be done by using the `mime` package for this purpose, similar to how we built messages.

In order to have the `mime` package parse an already built message, we can use the `mime::initialize` command. It will parse the message, parse headers, and split it into appropriate parts if we don't provide the -canonical flag. For example:

```tcl
set message [lindex [pop3::retrieve $h $id] 0]
set token [mime::initialize -string $message]
```
It is a good idea to catch exceptions from \texttt{mime::initialize}, because if the message is not correctly formatted, the command will throw an error. As our messages can come from an unknown source, it is a good idea not to assume all messages can be properly handled.

We can use the token to access information in it and subsequent parts. In order to do this, we'll use three commands: \texttt{mime::getheader}, \texttt{mime::getproperty}, and \texttt{mime::getbody}.

The first command can be used to retrieve MIME headers such as \texttt{From}, \texttt{To}, \texttt{Subject} or \texttt{Content-ID}. The command \texttt{mime::getproperty} returns information about a particular token. The property \texttt{content} specifies content type, \texttt{encoding} specified content transfer encoding (base64 or quoted-printable), and \texttt{size} provides the size of the content before decoding it. The \texttt{parts} property provides a list of subordinates (children) of this node and it is only set if this node has children.

Both commands accept a MIME token as the first argument and, optionally, a property or header name as the second argument. If a name is not specified, the commands return a name-value pair list. Both commands also throw an error if a header or property is not set for this token.

The command \texttt{mime::getbody} decodes and returns the body of a message or a subordinate. It is used to extract the non-multipart elements of a message.

Let's use these methods in a more practical example. Assume we want to create an application that will check e-mails, look for messages that have an attachment with \texttt{jobdata.raw} filename and insert the contents of each of these attachments to a database. We'll also keep track of messages we've already parsed and will not delete them.

For the database, we'll use SQLite3, because it will not require the reader to set up any database server. SQLite3 is included in ActiveTcl, so we should have it installed already. More details about databases can be found in Chapter 5.

We'll start by initializing our code and database. We'll load the appropriate packages and initialize the database:

```tcl
package require pop3
package require tls
package require sqlite3
package require mime

if {![file exists "database.db"]} {
    sqlite3 db "database.db"
}
```

**For More Information:**

Using Common Internet Services

```tcl
db eval {
    CREATE TABLE emails
        (uidl TEXT, status INTEGER)
}

db eval {
    CREATE TABLE jobs
        (uidl TEXT, id INTEGER, jobdata TEXT)
} else {
    sqlite3 db "database.db"
}
```

If the database file does not exist, we'll create the appropriate tables. If the database exists, we'll assume it already has the proper tables set up. We're creating a table called `emails` that keeps unique identifiers of messages we've already parsed and a table called `jobs` which keeps track of jobs along with source messages.

As one of the first steps, we open the POP3 connection:

```tcl
set h [pop3::open -socketcmd tls::socket
    $hostname $username $password 995]
```

Now let's fetch the unique identifier information and iterate through it. We'll store a list of IDs and unique identifiers that we do not know in the `idlist` variable, which we'll initialize using an empty string:

```tcl
set idlist [list]
```

# read all messages and find out which ones we should process
foreach {id uidl} [pop3::uidl $h] {

Following that, we fetch information for this UIDL from our local database and see if it is not empty:

```tcl
set status [lindex [db eval
    {SELECT status FROM emails WHERE uidl=$uidl} \ 0]
if {$status == ""} {
```

If the status is empty, it means that the entry in the database does not exist — the database returned 0 items and first element from this list is empty. In this case, we'll add our message to the list of messages to process. Otherwise, we'll skip it. In both cases, we'll also print out the status to standard output.

```tcl
lappend idlist $id $uidl
puts "We'll process message $id ($uidl)"
} else {
```

For More Information:
# this entry was already processed
puts "Message $id ($uidl) was already processed\n    and has status $status"
}

And that finishes our loop as well:
}

Now the idlist variable contains a list of messages that we need to process kept as id-uidl pairs. We'll iterate through it and fetch the messages:

    foreach {id uidl} idlist {
        puts "Retrieving message $id"
        set message [lindex [pop3::retrieve $h $id] 0]
        if {[catch {
            set token [mime::initialize -string $message]
            error]}
            puts "Error while parsing message: $error"
            db eval {
                INSERT INTO emails (uidl, status)
                VALUES($uidl, "ERROR")
            }
        } else {
            Our application retrieves a message and tries to parse it. If the parsing fails, we mark this fact in our database. If not, we proceed to handling the message:

                set attachments [findAttachments $token jobdata.raw]
                set aid 0
                foreach attachment $attachments {
                    incr aid
                    set jobdata [mime::getbody $attachment]
                    db eval {
                        INSERT INTO jobs (uidl, id, jobdata)
                        VALUES($uidl, $aid, $jobdata)
                    }
                }

                set status "FETCHED; COUNT=[llength $attachments]"
                db eval {
                    INSERT INTO emails (uidl, status)
                    VALUES($uidl, $status)
                }
                mime::finalize $token -subordinates all

For More Information:
Once we have the token, we invoke `findAttachments` procedure. It’s described in the following code, but returns a list of all MIME tokens that are an attachment and their filename is `jobdara.raw`. We’ll go through this procedure in detail after finalizing our main part of the application.

Next we need to close our condition and loop, and close the POP3 connection:

```tcl
}
}

pop3::close $h
```

Our helper procedure to find attachments looks like:

```tcl
proc findAttachments {token filename} {
    set result [list]

    # if it is a multipart node, iterate over children
    # otherwise process this part
    if {[catch {
        set children [mime::getproperty $token parts]
    }]} {
        # getting parts failed - it’s a leaf
        # in this case, get Content-ID header
        # and compare it to requested filename
        if {[catch {
            set contentID [mime::getheader $token "Content-ID"]
        }]} {
            # if filename matches Content-ID
            if {$contentID == $filename} {
                lappend result $token
            }
        } else {
            # for multipart/*, we’ll iterate over children
            # to see if any of them contains the attachment
            foreach token $children {
                set result [concat $result \n                    [findAttachments $token $filename] \n                ]
            }
        }
    } else {
        # for multipart/*, we’ll iterate over children
        # to see if any of them contains the attachment
        foreach token $children {
            set result [concat $result \n                [findAttachments $token $filename] \n            ]
        }
    }
    return $result
}
```

For More Information:

What we do here is try to get parts property for the token. If this fails, we proceed with finding Content-ID header. If this does not fail, and is what we expected to find, we add this token to results. If this part had children, we iterate through them and recursively call our procedure again. We also combine results from all of these calls so that it is possible to send multiple attachments, and/or they can be put in various parts of the e-mail.

The example in this section is located in the 05pop3andmime directory in the source code examples for this chapter.

Documentation on pop3 package can be found on the tcllib project website on SourceForge: http://tcllib.sourceforge.net/doc/pop3.html

Transferring files and data

Transferring files over the Internet is one of the most common things that networked applications do. In this section, we'll focus on two very popular protocols—HyperText Transfer Protocol (HTTP) and File Transfer Protocol (FTP). The first one is used to download websites, images, or other files from the Web and second one is used to download or upload files. There are subtle differences between the two and we'll need to go into a bit more detail to understand them.

HTTP is a for retrieving a single item over network. It allows us to retrieve a single element and is a lightweight protocol. It also allows us to handle large numbers of requests and is the protocol used for serving websites to web browsers. HTTP is used for both static files (such as HTML pages, images, downloads, and so on) and dynamic files (such as PHP or Tcl scripts building pages when users access them).

FTP protocol on the other hand is designed to transfer files over a network. It features authentication, and offers a lot of features that are specific to file management—creating directories, renaming and deleting items, listing contents of a directory, and the idea of a working directory. FTP is also a more heavyweight protocol and is less commonly used for offering downloads to wide audience.

Resources and uri package

Specifying locations of a resource on the Internet is done by specifying a URL—Unique Resource Location. It consists of a protocol, an optional username, a password, a hostname, and a port followed by the path to the resource. For example, a URL could look like http://wiki.tcl.tk/tcllib

For More Information:
Tcl offers the `uri` package, which is part of Tcllib, and can be used to split and join URLs from parts. These parts include `scheme`, `user`, `pwd`, `host`, `port`, `path`, `query`, and `fragment`. Not all of these are always present in all types of URLs. The first one is the only key only present and defines protocol that is used—for example, `http` and `ftp`. Credentials are optional and are specified as `user` and `pwd`. The parts `host` and `port` specify the hostname and port to connect to; `port` can be empty, which means it is the default port for specified protocol. The location of the resource is specified as `path`, and `query` is an optional part that defines a query sent via the URL (mainly for `http` requests); `fragment` points to a fragment of a page and is also used only for the HTTP protocol.

Currently, `ftp`, `http`, `https`, `file`, `mailto`, and `news` protocols are supported.

We can split a URL into elements using the `uri::split` command. It returns one or more name-value pairs which represent each part. For example, we can do the following:

```tcl
set uridata [uri::split "http://wiki.tcl.tk/tcllib"]

foreach name {scheme user pwd host port path query fragment} {
    if {[dict exists $uridata $name]} {
        puts "$name = [dict get $uridata $name]"
    }
}
```

This will print the following result:

```
scheme = http
user =
pwd =
host = wiki.tcl.tk
port =
path = tcllib
query =
```

We can also create a URL by specifying various parts using the `uri::join` command. It takes all arguments and parses them as name-value pairs, specifying parts of the address it should generate. For example:

```tcl
puts [uri::join \scheme http \host www.packtpub.com \port 80 \path books ]
```

The preceding code will print out the address `http://www.packtpub.com/books`. Please note that `port` part was skipped, because 80 is the default port for the HTTP protocol.

For More Information:
We can also use the result from splitting and join it back by running:

```tcl
grep http://www.google.com
```

This will split the address into parts and pass that to the `uri::join` command—`{*}` will cause all elements of the list to be appended as separate arguments, because the command expects it.

The sample code shown in this section is located in the `06uri` directory in the source code examples for this chapter.

More information about the `uri` package can be found in its documentation available at:
http://tcllib.sourceforge.net/doc/uri.html

### Using HTTP

HTTP is a stateless protocol that uses a simple request-response message exchange pattern. This means that whenever a client, such as our application, wants to access a particular resource, it sends a HTTP request. The server then processes it and sends back a response, usually being the requested response or information that it could not be found or an error occurred.

HTTP works by sending a request to the server. A request describes whether we are getting information or sending data to server, the path to the resource, and the version of the protocol we're using. A request also consists of several headers, which are name-value pairs and can either be standard or custom ones. Finally, a request can also provide data that we are uploading to server.

### Retrieving data over HTTP

After receiving and parsing the request, the web server returns the response. The response consists of a status line, one or more headers, and the body of the response. After the response is sent, the current connection is either closed or reused for the next request. However, from the HTTP perspective, each of these requests is treated independently.

Tcl comes with a `http` package built-in. This package offers a basic, but complete HTTP client which can be used to perform both basic and more advanced operations. The command `http::geturl` is used to initiate or perform a request and is the starting point for performing HTTP operations. This command accepts a URL followed by one or more options.

For More Information:
A commonly used option is `-binary`, which specifies whether the transfer should be done in binary mode and defaults to `false`. By default, Tcl does newlines and encoding conversions for text documents—therefore, if a server sends out some HTML in UTF-8, Tcl converts that to proper string. If the `-binary` option is enabled, this is not performed and all types of documents are retrieved as bytes, not depending on whether it is a text document or not.

The `http::geturl` command always returns a token that can be used to get information and data related to this query. For example, in order to get the contents of Google's main page, we can simply run:

```tcl
package require http
set token [http::geturl "http://www.google.com/"
puts [http::data $token]
```

The second line receives the request and returns token that we can use later on. The command `http::data` returns body of server's response, which we then print to standard output.

The command `http::cleanup` should be used after we are done working with a request. It will clean up all resources used for this request. For example:

```tcl
http::cleanup $token
```

We can also save contents of the response directly to any open channel. For example:

```tcl
set fh [open "google-index.html" w]
set token [http::geturl "http://www.google.com/" -channel $fh]
http::cleanup $token
close $fh
```

This will cause a response to be written to the specified channel and not stored in memory. This can be used for downloading large files and/or if you plan to save the contents of the response to a file.

There are important things that our previous example is missing, for example, checking for errors. Although the `http` package will throw an error if the web server is unreachable, there are cases when a web server will send a response stating that a resource is unavailable or an error has occurred. In these cases, it is not translated into an error as this might be the desired response from our application's perspective.
We can check the status of handling the request by using the \texttt{http::status} command. It will return one of the following values:

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ok</td>
<td>Indicates that the HTTP request was completed successfully</td>
</tr>
<tr>
<td>eof</td>
<td>Indicates that the server closed connection without replying</td>
</tr>
<tr>
<td>error</td>
<td>Indicates an error</td>
</tr>
</tbody>
</table>

If the status is \texttt{error}, we can also retrieve actual error message using the \texttt{http::error} command.

HTTP server sends status codes which specify the outcome of processing this request. We can retrieve the status code using the \texttt{http::ncode} command. Usually, it is sufficient to check if the code equals \texttt{200}, which means that the request has been processed correctly.

The most frequently used status codes are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>The request has been successfully executed</td>
</tr>
<tr>
<td>206</td>
<td>The request has been successfully executed and result is partial content, used to download parts of a file over HTTP</td>
</tr>
<tr>
<td>301</td>
<td>Moved permanently—indicates that resource has been permanently moved to a new location; response header \texttt{Location} gives new location to resource</td>
</tr>
<tr>
<td>302</td>
<td>Moved temporarily—indicates that resource has been temporarily moved to a new location; response header \texttt{Location} gives new location to resource</td>
</tr>
<tr>
<td>401</td>
<td>Unauthorized—indicates that a server has requested the client to authenticate</td>
</tr>
<tr>
<td>403</td>
<td>Forbidden—this indicates that access to this resource is forbidden</td>
</tr>
<tr>
<td>404</td>
<td>Not found—indicates that a specified resource cannot be found</td>
</tr>
<tr>
<td>500</td>
<td>Internal error—indicates that there was a problem serving this request; for example, because HTTP server configuration is broken or module/script failed</td>
</tr>
</tbody>
</table>

For example, we can print out the status of our request by running:

```tcl
switch -- [http::status $token] {
  error {
    puts "ERROR: [http::error $token]"
  }
  eof {
    puts "EOF reading response"
  }
  ok {
```

For More Information:

Using Common Internet Services

```tcl
puts "OK; code: [http::ncode $token]"
puts "Data:"
puts [http::data $token]
}
}
```

We can also get all headers from an HTTP response by using the `http::meta` command. It returns a list of name-value pairs that can be used as dictionaries or arrays. For example, to get contents of the `Location` header, we can do the following:

```tcl
set code [http::ncode $token]
if {($code == 301) || ($code == 302)} {
    set newURL [dict get [http::meta $token] Location]
    # go to new location
}
```

The complete example is located in the `07httprequest` directory in the source code examples for this chapter.

### Submitting information using GET and POST

We can also use the `http` package to submit information to a web server—for automating things such as filling in forms. Data from a form can be formatted using the `http::formatQuery` command. It can now be sent in two ways—either as part of the path in the URL or sending data as separate data. The first case is done using a GET request and an example is searching using Google, like `http://www.google.com/search?q=tcl` — the query is passed after `?` character. The other approach is sending a POST request and the data is sent after the actual request.

POST is used for sending a larger amount of data and usually takes place when the request is modifying/sending data. GET is usually used for reading information as it can send smaller amount of data. POST requests are used for sending data, they can send much larger amount of data and POST requests are not cached by proxy servers.

For both GET and POST, data is sent as name-value pairs—`q=tcl` means value for field `q` is `tcl`. Multiple values are separated using `&` character. Tcl offers a command for generating such data, that is `http::formatQuery`. It accepts zero or more name-value pairs as arguments and formats proper query as an output.

Sending data using GET requires that we append the query to actual URL, for example:

```tcl
set query [http::formatQuery q "Tcl Programming"]
set url "http://www.google.com/search?$query"
```
Sending POST data requires passing the data as a -query option to `http::geturl`. For example, we can do the following:

```tcl
set query [http::formatQuery search "tcl programming"]
set url "http://www.packtpub.com/search"
set token [http::geturl $url -query $query]
```

This will cause a query to be sent as POST and data from -query will be sent.

By default, data is sent as encoded form data, it is also possible to send different query data. Usually, this is accompanied by sending the appropriate query type to the server. We can do this by adding the -type flag when sending the query. If a type is not specified, it defaults to `application/x-www-form-urlencoded`, which is the default MIME type for encoded form data. Many applications expecting XML or [JavaScript Object Notation (JSON)](https://www.packtpub.com/tcl-8-5-network-programming/book) data require that data sent in XML/JSON is sent with the appropriate MIME type in the headers of the request.

For example, we can send XML with accompanying data by doing:

```tcl
http::geturl $url -command onCompleteXMLPost -type "text/xml" -query [$dom asXML]
```

This will cause the appropriate value for the `Content-Type` header to be sent in the query. Details on XML and its handling can be found in Chapter 5 and show how to read and write documents.

Examples related to basic HTTP functionality are placed in [basic.tcl](https://www.packtpub.com/tcl-8-5-network-programming/book) file in the 07httprequest directory in the source code examples for this chapter.

By default, `http` queries are done in a synchronous way, meaning that the `http::geturl` command returns after the command has been executed. In many cases, it is better to use an asynchronous approach, where the command exits instantly, uses events to process requests, and issues our callback, which is a Tcl command that will be run when the operation is completed.

The `http` package also offers advanced features such as passing additional headers to requests. This can be done by providing both the -timeout and -command options to the `http::geturl` command. In this case, the command returns immediately and returns a token that will be used. Accessing the data should be done from the command passed to the -command option. In asynchronous requests, `http::geturl` might still throw an error, for cases such as "no existing hostname". It is still recommended to catch such exceptions and handle them appropriately.
For example, in order to download the Google page asynchronously, we can do the following:

```tcl
if {{catch {
    set token [http::geturl "http://www.google.com/" \ 
        -timeout 300000 -command doneGet]
} error} {
    puts stderr "Error while getting URL: $error"
}}
```

Next we can create the command that will be invoked as the callback. It will be run with an additional parameter—the token of the request. For example, our command, based on previous examples, can be run as follows:

```tcl
proc doneGet {token} {
    switch -- [http::status $token] {
        error {
            puts "ERROR: [http::error $token]"
        }
        eof {
            puts "EOF reading response"
        }
        ok {
            puts "OK; code: [http::ncode $token]"
            puts " Size: [http::size $token]"
            puts " Data:"
            puts [http::data $token]
        }
    }
    http::cleanup $token
}
```

The token can be used in the same way as we used it with synchronous requests. We are also responsible for cleaning up the token, which is done in the last line of the example.

Examples related to basic HTTP functionality are located in the async.tcl file in the 07httprequest directory in the source code examples for this chapter.

Advanced topics
The package http can also be used for more advanced features such as partial content downloading, sending cookies, and HTTP-level authorization.

For More Information:
The majority of these functions can be carried out using the -headers option passed to the \texttt{http::geturl} command. This option accepts a list of one or more name-value pairs. These can be any headers and values, but these should be headers that the server can understand. For example, we can use it to send cookie values to a site or authorize over HTTP for sites that use it.

There are two common ways that users are authorized within the Web—at the HTTP level and using HTML forms and cookies. The first one provides the username and password information as a HTTP header. The latter uses sending form data and cookies to track users, and is mainly related to handling cookies properly at the HTTP level.

For now, we'll focus on HTTP level authorization. A lot of web-based applications and data are protected using this mechanism. Let's assume we want to retrieve data from a specified URL. We need to connect to it without providing any credentials, and at this point, the server should include HTTP status 401. The following code would be a good start for checking if authorization is needed:

```tcl
set token [http::geturl $url] 
if {
    [http::status $token] != "ok"} {
    puts stderr "Error while retrieving URL"
    http::cleanup $token 
    exit 1 
}
if {
    [http::ncode $token] == 401} {

} 
```

If this condition is true, we should resend our request. The server will provide the \texttt{WWW-Authenticate} header in the response that will indicate the type of authentication and realm, which specifies the descriptive name of the resource we are currently trying to authenticate to. We can print it out by running:

```tcl
    set realm [dict get [http::meta $token] \n    "WWW-Authenticate"]
    puts "Authenticate information: $realm"
```

Next we need to clean up the previous request and send a new one with proper authentication information. Except for a few cases, the authentication type of Basic is used by HTTP servers. It requires sending a \texttt{<username>:<password>} string encoded as \texttt{base64}, preceded by the word Basic. We'll use the package \texttt{base64} for this along with the \texttt{base64::encode} command:

```tcl
package require base64
set authinfo [base64::encode $\{username\}:$\{password\}]
set headers [list Authorization "Basic $authinfo"]
```
Using Common Internet Services

The second line contains the Authorization header to be sent to the server, along with the credentials as base64. Next we're sending a new request by doing:

```
set token [http::geturl $url -headers $headers]
if {[http::status $token] != "ok"} {
    http::cleanup $token
    puts stderr "Error while retrieving URL"
    exit 1
}
```

We can then check if our current username and password were correct. If not, then the status for a new request will also be 401:

```
if {[http::ncode $token] == 401} {
    puts stderr "Invalid username and/or password"
    http::cleanup $token
    exit 1
}
```

An example related to basic authorization is located in the auth.tcl file in the 07httprequest directory in the source code examples for this chapter.

Cookies in Tcl

An additional feature that headers are useful for is supporting cookies. While the http package itself does not provide this functionality, it is easy to support this in the majority of cases. Standards for setting and getting cookies define expiration dates, and paths and domains that cookies should be valid for. However, in the majority of code that we write, it is enough to assume that the cookie you’re getting is needed for all subsequent requests.

Cookies work in such a way that HTTP responses from servers may include one or more Set-Cookie headers. These headers need to be parsed and all cookies should be passed in the Cookie header. The server might send a response similar to this one:

```
Set-Cookie: mycookie=TEST0123; path=/
Set-Cookie: i=1; expires=Thu, 27-Oct-2011 11:07:24 GMT; path=/
```

This causes the cookie mycookie to be set to TEST0123 and i to be set to 1. Each subsequent request to this server should include the following header:

```
Cookie: mycookie=TEST0123; i=1
```

All changes to existing cookies overwrite them and new cookies cause a new value to be set, which is similar to behavior of arrays and dictionaries in Tcl. Writing code that handles cookies without taking parameters into account is relatively easy.

For More Information:

Let's start by writing a command that processes the HTTP response for cookies. We define the namespace for our code, the reference variable specified by user, and iterate over HTTP headers from the provided token:

```tcl
namespace eval cookies {}

proc cookies::processCookies {varname token} {
    upvar 1 $varname d
    foreach {name value} [http::meta $token] {
        if {[string equal -nocase $name "Set-Cookie"]} {
            if {
                [regexp "^(.*?)=(.*)$" $value
                - cname cvalue] {
                dict set d $cname $cvalue
            }
        }
    }
}
```

This will cause the dictionary that is stored in the `varname` variable to be updated. Next, in all requests, we need to pass all cookies. A small function to generate the appropriate value for the `Cookies` header would look like:

```tcl
proc cookies::prepareCookies {var} {
    set rc [list]
    dict for {name value} $var {
        lappend rc "$name=$value"
    }
    return [join $rc "; "]
}
```

Here we only take each cookie, append it, and join all cookies using a semi-colon followed by a space. In order to use this to query Tcler's wiki we can do the following:

```tcl
set c [dict create]
set h [http::geturl http://wiki.tcl.tk/]
cookies::processCookies c $h
http::cleanup $h

set query [http::formatQuery _charset_ utf-8 S cookie]
set h [http::geturl http://wiki.tcl.tk/_/search?$query
    -headers [list Cookie [cookies::prepareCookies $c]]]
```

For More Information:
The first request gets the main page of the Wiki, which causes a cookie to be set. We need to pass this cookie to the second request in order to be able to perform a search. In this case, we’re searching for the cookie string. Without passing the cookie from previous request, this site will not allow us to perform the search.

An example related to basic authorization is located in the `cookies.tcl` file in the `07httprequest` directory in the source code examples for this chapter.

### HTTP and encryption

HTTP can handle both encrypted and unencrypted communication. The default is not to encrypt the connection, which is in fact `http` protocol when specifying URLs. It is also possible to use HTTP over SSL encrypted connection, which is usually called `https`.

The Tcl package `http` allows registering additional protocols to run HTTP on with the command `http::register`. It requires that we specify the name of the protocol, default port, and command that should be invoked to create a socket. This is mainly used for SSL connections. In order to enable the use of the `https` protocol, we need to add the following code to our application:

```tcl
package require tls
http::register https 443 tls::socket
```

The `tls` package provides SSL-enabled sockets to the Tcl language and it provides the command `tls::socket`, which is an equivalent of the `socket` command, except for enabling SSL for connection. SSL and security is described in more detail in Chapter 12.

More information about the `http` package as well as remaining configuration options can be found in its documentation at:

```
http://www.tcl.tk/man/tcl8.5/TclCmd/http.htm
```

### Retrieving RSS information

Really Simple Sindication (RSS) is a format for publishing frequently updated information, such as blog entries, news headlines, audio, and video in a standard format. An RSS document (often also called a feed or channel) provides a list of items recently published along with metadata about these items. RSS is provided by a majority of content providers, such as portals, blog engines, and so on. Even Packt Publishing has its own RSS feed that we'll use later on in an example.
RSS itself is an XML document published over HTTP. This means that using the http and tdom packages, we can easily retrieve and parse an RSS feed and find out about recent documents. The RSS standard describes the structure of the XML document, which we'll learn later. All we need to know is the URL to RSS feed to start with. Information about the address to the RSS feed is usually stored in the website's metadata. This as well is standardized and usually looks like this:

```xml
<link rel="alternate" type="application/rss+xml" href="/rss.xml" title="Packt Publishing News"/>
```

The previous example is from Packt Publishing's website. Your browser also probably supports this and a small icon on the bottom or near address of the page indicates that an RSS feed is present—clicking on it will go to the RSS feed and allow you to subscribe to it from your browser and get the address of the actual RSS feed.


We'll start with Tcler's Wiki and its feeds. The feed looks as follows:

```xml
<?xml version='1.0'?>
<rss version='0.91'>
  <channel>
    <title>The Tcler's Wiki - Recent Changes</title>
    <link>http://wiki.tcl.tk/</link>
    <description>Recent changes to The Tcler's Wiki</description>
    <item>
      <title>tDOM</title>
      <link>http://wiki.tcl.tk/1948</link>
      <pubDate>Wed, 14 Apr 2010 01:05:27 GMT</pubDate>
      <description>Modified by CMcC (898 characters)
        (actual description of the Wiki change goes here)
      </description>
    </item>
    <item>
      <title>WISH User Help</title>
      <link>http://wiki.tcl.tk/20914</link>
      <pubDate>Wed, 14 Apr 2010 00:59:10 GMT</pubDate>
      <description>Modified by pa_mcclamrock (194 characters)
        (actual description of the Wiki change goes here)
      </description>
    </item>
  </channel>
</rss>
```

For More Information:
Using Common Internet Services

In order to read RSS, we need to find the <rss> tag and iterate over all <channel> tags. The first one includes information about the RSS feed and each <channel> instance can describe a different channel. It is possible that one RSS feed describes multiple channels, although usually an RSS feed covers only one channel. Each channel has a title, link, and list of items.

In order to get all items in a channel, we need to iterate over the <item> tags inside the channel. Each item describes a single element in a feed, such as one entry on a blog, in this case, one change in the wiki. Each item has a title, link, publication date, and description. Many RSS feeds provide additional information, which can be checked and handled properly if needed.

We can retrieve the RSS by simply doing:

```tcl
set token [http::geturl "http://wiki.tcl.tk/rss.xml"]
if {[http::status $token] != "ok"} {
    puts "Error retrieving RSS file"
    exit 1
}
set data [http::data $token]
http::cleanup $token
```

We now have the RSS document in the `data` variable, and we can parse it using `tdom`:

```tcl
set dom [dom parse $data]
```

The `tdom` package is described in more detail in Chapter 5.

Now we can iterate over each channel by doing:

```tcl
foreach channel [$dom selectNodes "rss/channel"] {
```

This will use the `selectNodes` method to find all channel tags. We can then find the <title> tag in our channel and use `asText` method for that node to get title of current channel and print it:

```tcl
set nodes [$channel selectNodes "title"]
set title [[lindex $nodes 0] asText]
puts "Channel "$title":"
```

We can now iterate over all items for a channel in similar way:

```tcl
foreach item [$channel selectNodes "item"] {
    set nodes [$item selectNodes "link"]
    set link [[lindex $nodes 0] asText]
    set nodes [$item selectNodes "title"]
```
We first use the `selectNodes` method to find `<item>` tags, iterate over them, get the link and title by finding proper nodes and using the `asText` method. We then print information on each element.

Finally we need to close the loop iterating over channels:

```
}
```

The source code in this section is located in the `rss-basic.tcl` file in the `08rss` directory in the source code examples for this chapter.

In many cases our applications will need to check and retrieve RSS periodically. In such cases, it is a good idea to cache the RSS on disk or in memory. If our application offers a web interface to consolidate multiple RSS channels or filter them to only include specified items, this would be the best approach.

In order to do this, all we need to do is change how our DOM tree is created. We'll start by setting URL of the feed and name of the file to store it:

```
set url "http://www.packtpub.com/rss.xml"
set filename "packtpub-rss.xml"
```

Next we can check if the local copy exists and if it was created in the last 30 minutes by doing:

```
if {(![file exists $filename]) ||
([file mtime $filename] < [clock scan "-30 minutes"])} {
```

This checks whether the file does not exist or if it has been created earlier than 30 minutes ago. If any of these conditions are met, then we download the RSS by doing:

```
set token [http::geturl $url -binary true]
```

```
if {[http::status $token] != "ok"} {
    puts "Error retrieving RSS file"
    exit 1
}
```

```
set fh [open $filename w]
fconfigure $fh -translation binary
puts $fh [http::data $token]
close $fh
http::cleanup $token
```

For More Information:

This is similar to previous example and to the HTTP examples shown earlier. The main difference is that we're downloading the file in binary mode. This prevents the http package from converting the file's encoding.

We will use the tDOM::xmlReadFile command to read the RSS. This command is part of the tdom package and handles encoding issues when reading files such as detecting encoding. It also handles the Byte Order Mark (BOM) markers that many RSS feeds have. This is a set of bytes at beginning of XML file that specifies encoding of the file and is described in more detail at:
http://en.wikipedia.org/wiki/Byte_order_mark

In order to read and parse the file, all we need to do is:

    set dom [dom parse [tDOM::xmlReadFile $filename]]

After that, we can use the same set of iterations as previously to list all entries in the RSS feed:

    foreach channel [$dom selectNodes "rss/channel"] {
        set nodes [$channel selectNodes "title"]
        set title [[lindex $nodes 0] asText]
        puts "Channel "$title":"
        foreach item [$channel selectNodes "item"] {
            set nodes [$item selectNodes "link"]
            set link [[lindex $nodes 0] asText]
            set nodes [$item selectNodes "title"]
            set title [[lindex $nodes 0] asText]
            puts "- "$link\" $title"
        }
    }

The source code in this section is located in the rss-file.tcl file in the 08rss directory in the source code examples for this chapter.

Using FTP

The File Transfer Protocol (FTP) is a stateful protocol for transferring files. It requires logging in, keeps the connection alive across transfers, and is not a lightweight protocol. It is mainly used for retrieving or transferring multiple files.

Tcl has a package called ftp, which is a part of Tcllib, and can be used to download and upload files over FTP. It offers functionality for connecting, getting file information, and uploading and downloading files.

For More Information:
Establishing connections

The command `ftp::Open` can be used to set up a connection to an FTP server. It accepts the server name, username, and password followed by any additional options we might want to provide. It returns a token that we can later use for all other operations.

Anonymous FTP connections require specifying `anonymous` as the username and the e-mail as the password. For example, in order to open an anonymous connection to `ftp.tcl.tk`, we can do:

```tcl
set token [ftp::Open ftp.tcl.tk anonymous my@email.com]
```

An FTP session has a dedicated connection to the server. For each additional FTP transfer such as listing files, downloading and uploading, additional connections are made with the server for the purpose of each transfer. The FTP protocol uses two modes for communication—active and passive. Active connections work in such a way that FTP server connects to its client for sending data, passive connections work the opposite way, the client connects to FTP server.

While the default for `ftp` package is to use active mode, it might be necessary to use passive mode if our computer does not have a public IP address. Passive mode is also the default for majority of clients as it works regardless of having a public IP address, so it is a good idea to use passive mode whenever possible. Specifying the mode can be done using the `-mode` flag appended to the `ftp::Open` command. Acceptable values are `active` and `passive`. For example:

```tcl
set token [ftp::Open ftp.tcl.tk anonymous my@email.com -mode passive]
```

Retrieving files

Another important aspect of FTP we should be aware of is transfer type. Due to how different operating systems store information, FTP differentiates between text (ASCII) and binary files. We can do this using the `ftp::Type` command. It accepts a token as the first argument and the transfer type as additional argument. It can be either `ascii` or `binary`. To set our transfer type to binary, we can do:

```tcl
ftp::Type $token binary
```
We can now retrieve files over FTP. We can use the command `ftp::Get` for this purpose. It works in different modes depending on the arguments supplied. It first accepts the token of the connection, followed by the path to the remote file. We run this command without any arguments—in this case, the file will be downloaded with the same name as the remote path. If we specify a local filename as the next argument, it will be downloaded as that name. Instead, we can also specify `-variable` or `-channel` options, followed by a value. This will cause file data to be downloaded to a variable or saved in a specified channel. In case of a channel, it will not be closed after file is retrieved.

For example, we can retrieve remote file `tcl8.5.7-src.tar.gz` from `pub/tcl/tcl8_5` remote directory as same file in local filesystem by doing:

```
ftp::Get "pub/tcl/tcl8_5/tcl8.5.7-src.tar.gz" \
"tcl8.5.7-src.tar.gz"
```

Similarly, we can download a file to variable by doing:

```
ftp::Get "pub/tcl/tcl8_5/tcl8.5.7-src.tar.gz" \
-variable fileContents
```

Please note that due to how this is implemented, the variable name is global and not local to the code invoking `ftp::Get` command. It is best to use namespace-based variables or object variables for this.

We can also resume an interrupted transfer by using the `ftp::Reget` command. It requires that we specify a token, a remote filename and, optionally, a local filename. If a local name is not specified, it is assumed to be the same as remote name. We can also specify offsets at which to begin and end download at, but by default, Tcl will download remaining part of the file.

For example, in order to complete transfer of `tcl8.5.7-src.tar.gz` file, we can simply invoke:

```
ftp::Reget $token tcl8.5.7-src.tar.gz
```

### Uploading files

Similarly, there are commands for putting and appending to remote files. The command `ftp::Put` can be used to upload a file, while `ftp::Append` will append data to an already existing file, which can be used to continue an interrupted transfer. In both cases, the syntax is the same — the first argument is the token of the FTP session to use, followed by either the local filename, `-data` or `-channel` options. In the first case only a filename is needed, in the second option actual data or channel to use needs to be specified. The last argument is the remote filename to use. If the remote filename is missing, it is assumed to be the same as the local one.
For example, to upload a file, we can do:

```tcl
ftp::Put my-logs.tar.gz
```

In order to append data to a file, we can do:

```tcl
ftp::Append -data "Some text\n" remote-logs.txt
```

When downloading or uploading data, it will be treated as binary data—that is, if we are downloading text, we can use the `encoding` command to convert it from/to proper encoding.

### Listing files and directories

FTP also introduces the concept of the current directory for a specified FTP session. We can change the directory by invoking the `ftp::Cd` command and retrieve the current directory by invoking `ftp::Pwd`. The first command expects the FTP session token and the path to the directory, which can be relative or absolute. The second command always returns an absolute path, which can be used when comparing and/or analyzing current location.

For example:

```tcl
puts "Changing directory"
ftp::Cd $token "pub/tcl/tcl8_5"
puts "Changed to [ftp::Pwd $token]"
```

We can also retrieve information about remote files. The command `ftp::FileSize` returns size of a file in bytes. The command `ftp::ModTime` returns the time when a file was last modified, as Unix time. Both commands require a token to the FTP session and a filename. For example:

```tcl
set size [ftp::FileSize $token tcl8.5.7-src.tar.gz]
puts "tcl8.5.7-src.tar.gz is $size bytes"
set mtime [ftp::ModTime $token tcl8.5.7-src.tar.gz]
set mtext [clock format $mtime]
puts "tcl8.5.7-src.tar.gz last modified on $mtext"
```

We can also list the contents of a directory. The command `ftp::NList` can be used to list all files and directories in the current or specified directory. It accepts a token to the session and we can also provide directory to list. If this is not specified, listing of current directory is performed. This command returns a list of all items found in a directory, each element of a list being the name of the file or directory.

For example:

```tcl
foreach file [ftp::NList $token] {
    puts $file
}
```
Using Common Internet Services

The command `ftp::List` returns a long listing of a directory. This returns a list of items, where each item is represented by a line, similar to output of `ls -l` command in Unix. For example:

```tcl
foreach line [ftp::List $token] {
    puts $line
}
```

The preceding code would print out the following line, among others:

```
-rw-r--r-- 1 ftp ftp 4421720 Apr 15 2009 tcl8.5.7-src.tar.gz
```

While this provides much more information, we need additional code to parse such lines. Let's start with creating a command for this:

```tcl
proc parseListLine {line} {
    # First we try to search for filenames with spaces and remove symbolic link definitions
    # which are in the form of filename -> actual_file it points to.
    if {[regexp {([^\s]|[^0-9])+$} $line name]} {
        # Check for links
        if {[set idx [string first " -> " $name]] != -1} {
            incr idx -1
            set name [string range $name 0 $idx]
        }
    }
    # Following that we remove any multiple spaces and create a list of items by splitting
    # the resulting string by spaces:
    regsub -all "\[ \t\]+" $line " " line
    set items [split $line " "]
    if {![info exists name]} {set name [lindex $items end]}
    # We then try to get the permissions and file size information, if possible:
    set perm [lindex $items 0]
    if {[string is integer [lindex $items 4]]} {
        set size [lindex $items 4]
    } else {
        lappend result ""
    }
}
```

For More Information:  
Based on the permissions we've extracted, we take the first character and gather the actual file type based on it:

```tcl
switch -- [string index $perm 0] {
    d {
        set type "directory"
    }
    c - b {
        set type "device"
    }
    l {
        set type "symlink"
    }
    default {
        set type "file"
    }
}
```

We then return a list that consists of the filename, type, size, and permissions:

```tcl
return [list $name $type $size $perm]
```

This code is based on `ftp.tcl` from the `tclvfs` package, which is licensed under the BSD license. The package is available at: http://sourceforge.net/projects/tclvfs/

We can then test it in the following way:

```tcl
foreach line [ftp::List $token] {
    puts "\nOriginal line: $line"
    lassign [parseListLine $line] 
    name type size perm
    puts "Filename '$name' ($type), size $size, $perm"
}
```

In addition to this, we can also modify remote filesystem contents. The command `ftp::MkDir` can be used to create a directory. It expects a token to the session as the first argument and the name of the directory to create as the second argument.

The command `ftp::Rename` can be used to rename a file or directory. It requires a token of the FTP session, and the old and new names.

The commands `ftp::RmDir` and `ftp::Delete` can be used to delete a directory or file, respectively. Both accept token of the FTP session and name of the directory or file to delete.
Using Common Internet Services

Closing a connection to an FTP server can be done using the `ftp::Close` command, specifying token of the FTP session. For example:

```
ftp::Close $token
```

The source code in this section is located in the `09ftp` directory in the source code examples for this chapter.

More information about the `ftp` package as well as the remaining configuration options can be found in its documentation in SourceForge project at: http://tcllib.sourceforge.net/doc/ftp.html

Summary

Tcl can communicate with applications and devices using a variety of protocols and talk to different types of hardware—embedded devices, routers, desktops, and servers.

We can use e-mail to communicate with users and/or applications. Tcl allows us to send and retrieve e-mails. It also offers mechanisms for parsing and building MIME messages that include support for attachments, *inline* images, multiple document types, and different encoding systems.

We can use Tcl to transfer files over FTP and HTTP. We can use this to retrieve information, communicate with server part of our application, download updates, upload results as well as many other operations. We can also automate web-based operations using HTTP client.

In this chapter, we have learned how to:

- Send e-mails from Tcl
- Build e-mails containing HTML and plain text message, attachments, and *inline* images
- Retrieve, parse, and process e-mail messages
- Get data over HTTP
- Read RSS feeds
- Upload and download files over FTP

The next chapter talks about some of additional network protocols and how they can be used from Tcl. It describes how to query for DNS information, and retrieve the current date and time from remote servers.

For More Information:
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.