Chapter 19. Client-Side Network Protocol Modules

Python's standard library supplies several modules to simplify the use of internet protocols on both the client and server sides. These days, the **Python Package Index**, best known as *PyPI*, offers many more such packages. Because many of the standard library modules date back to the previous century, you will find that nowadays third-party packages support a wider array of protocols, and several offer better APIs than the standard library's equivalents. When you need to use a network protocol that's missing from the standard library, or covered by the standard library in a way you think is not satisfactory, be sure to search PyPI—you're likely to find better solutions there.

In this chapter, we cover some standard library packages that allow relatively simple uses of network protocols: these let you code without requiring third-party packages, making your application or library easier to install on other machines. You may therefore come across them when dealing with legacy code, and their simplicity also makes them interesting reading for the Python student. We also mention a few third-party packages covering important network protocols not included in the standard library, but we do not cover third-party packages using asynchronous programming.

For the very frequent use case of HTTP clients and other network resources (such as anonymous FTP sites) best accessed via URLs, the third-party **`requests` package** is even recommended in the Python documentation, so we cover that and recommend its use instead of standard library modules.

Email Protocols

Most email today is *sent* via servers implementing the Simple Mail Transport Protocol (SMTP) and *received* via servers and clients using Post Office Protocol version 3 (POP3) and/or Internet Message Access Protocol version 4 (IMAP4).[1] Clients for these protocols are supported by the Python standard library modules `smtplib`, `poplib`, and `imaplib`, respectively, the first two of which we cover in this book. When you need to handle *parsing* or *generating* email messages, use the `email` package, covered in **Chapter 21**.

If you need to write a client that can connect via either POP3 or IMAP4, a standard recommendation would be to pick IMAP4, since it is more powerful and—according to Python's own online docs—often more accurately implemented on the server side. Unfortunately, `imaplib` is very complex, and far too vast to cover in this book. If you do choose to go that route, use the **online docs**, inevitably complemented by the IMAP RFCs, and possibly other related RFCs, such as 5161 and 6855 for capabilities and 2342 for namespaces. Using the RFCs in addition to the online docs for the standard library module can't be avoided: many of the arguments passed to `imaplib` functions and methods, and results from calling them, are strings with formats that are only documented in the RFCs, not in Python's own docs. A highly recommended alternative is to use the simpler, higher-abstraction-level third-party package **IMAPClient**, available with a **`pip install`** and well documented **online**.

The poplib Module

The `poplib` module supplies a class, `POP3`, to access a POP mailbox.[2] The constructor has the following signature:

`POP3` **`class`** `POP3(`*`host`*`, port=110)`
Returns an instance *p* of class `POP3` connected to the specified *`host`* and `port`. The class `POP3_SSL` behaves just the same, but connects to the host (by default on port 995) over a secure TLS channel; it's needed to connect to

email servers that demand some minimum security, such as `pop.gmail.com`.[a]

[a] To connect to a Gmail account, in particular, you need to configure that account to enable POP, "allow less secure apps," and avoid two-step verification—things that in general we don't recommend, as they weaken your email's security.

An instance *p* of the class `POP3` supplies many methods; the most frequently used are listed in **Table 19-1**. In each case, *msgnum*, the identifying number of a message, can be a string containing an integer value or an `int`.

Table 19-1. Methods of an instance `p` of `POP3`

`dele`	*p*`.dele(`*msgnum*`)` Marks message *msgnum* for deletion and returns the server response string. The server queues such deletion requests, and executes them only when you terminate this connection by calling *p*`.quit`.[a]
`list`	*p*`.list(`*msgnum*`=`**None**`)` Returns a three-item tuple `(`*response*`,` *messages*`,` *octets*`)`, where *response* is the server response string; *messages* a list of bytestrings, each of two words `b'`*msgnum bytes*`'`, the message number and length, in bytes, of each message in the mailbox; and *octets* is the length, in bytes, of the total response. When *msgnum* is not **None**, `list` returns a string, the response for the given *msgnum*, not a tuple.
`pass_`	*p*`.pass_(`*password*`)` Sends the password to the server, and returns the server response string. Must be called after *p*`.user`. The trailing underscore in the name is needed because **pass** is a Python keyword.

`quit`	*p*`.quit()` Ends the session and tells the server to perform deletions that were requested by calls to *p*`.dele`. Returns the server response string.
`retr`	*p*`.retr(`*msgnum*`)` Returns a three-item tuple (*response*, *lines*, *bytes*), where *response* is the server response string, *lines* is the list of all lines in message *msgnum* as bytestrings, and *bytes* is the total number of bytes in the message.
`set_debuglevel`	*p*`.set_debuglevel(`*debug_level*`)` Sets the debug level to *debug_level*, an `int` with value `0` (the default) for no debugging, `1` for a modest amount of debugging output, or `2` or more for a complete output trace of all control information exchanged with the server.
`stat`	*p*`.stat()` Returns a pair (*num_msgs*, *bytes*), where *num_msgs* is the number of messages in the mailbox and *bytes* is the total number of bytes.
`top`	*p*`.top(`*msgnum*, *maxlines*`)` Like `retr`, but returns at most *maxlines* lines from the message's body (in addition to all the lines from the headers). Can be useful for peeking at the start of long messages.
`user`	*p*`.user(`*username*`)` Sends the server the username; invariably followed up by a call to *p*`.pass_`.

a The standard states that if disconnection occurs before the `quit` call, the deletions should not be actioned. Despite this, some servers will perform

the deletion after any disconnection, planned or unplanned.

The smtplib Module

The `smtplib` module supplies a class, `SMTP`, to send mail via an SMTP server.[3] The constructor has the following signature:

`SMTP` — `class SMTP([host, port=25])`
Returns an instance `s` of the class `SMTP`. When *`host`* (and optionally `port`) is given, implicitly calls `s.connect(`*`host`*`, port)`. The class `SMTP_SSL` behaves just the same, but connects to the host (by default on port 465) over a secure TLS channel; it's needed to connect to email servers that demand some minimum security, such as `smtp.gmail.com`.

An instance `s` of the class `SMTP` supplies many methods. The most frequently used of these are listed in **Table 19-2**.

Table 19-2. Methods of an instance `s` of `SMTP`

`connect` — `s.connect(host=127.0.0.1, port=25)`
Connects to an SMTP server on the given `host` (by default, the local host) and `port` (port `25` is the default port for the SMTP service; `465` is the default port for the more secure "SMTP over TLS").

`login` — `s.login(`*`user`*`, `*`password`*`)`
Logs in to the server with the given *`user`* and *`password`*. Needed only if the SMTP server requires authentication (as just about all do).

`quit` — `s.quit()`
Terminates the SMTP session.

`sendmail`	`s.sendmail(`*`from_addr`*`, `*`to_addrs`*`, `*`msg_string`*`)` Sends mail message *`msg_string`* from the sender whose address is in string *`from_addr`* to each of the recipients in the list *`to_addrs`*.[a] *`msg_string`* must be a complete RFC 822 message in a single multiline bytestring: the headers, an empty line for separation, then the body. The mail transport mechanism uses only *`from_addr`* and *`to_addrs`* to determine routing, ignoring any headers in *`msg_string`*.[b] To prepare RFC 822–compliant messages, use the package `email`, covered in **"MIME and Email Format Handling"**.
`send_message`	`s.send_message(`*`msg`*`, from_addr=`**`None`**`, to_addrs=`**`None`**`)` A convenience function taking an `email.message.Message` object as its first argument. If either or both of `from_addr` and `to_addrs` are **`None`**, they are extracted from the message instead.

a While the standard places no limits on the number of recipients in *`from_addr`*, individual mail servers may well do so, often making it advisable to batch messages with a maximum number of recipients in each one.

b This allows email systems to implement Bcc (blind copy) emails, for example, as the routing does not depend on the message envelope.

HTTP and URL Clients

Most of the time, your code uses the HTTP and FTP protocols through the higher-abstraction URL layer, supported by the modules and packages covered in the following sections. Python's standard library also offers lower-level, protocol-specific modules that are less often used: for FTP clients, **ftplib**; for HTTP clients, `http.client` (we cover HTTP servers in **Chapter 20**). If you need to write an FTP server, look at the third-party

module **pyftpdlib**. Implementations of the newer **HTTP/2** may not be fully mature, but your best bet as of this writing is the third-party module **HTTPX**. We do not cover any of these lower-level modules in this book: we focus on higher-abstraction, URL-level access throughout the following sections.

URL Access

A URL is a type of uniform resource identifier (URI). A URI is a string that *identifies* a resource (but does not necessarily *locate* it), while a URL *locates* a resource on the internet. A URL is a string composed of several parts (some optional), called *components*: the *scheme, location, path, query*, and *fragment*. (The second component is sometimes also known as a *net location*, or *netloc* for short.) A URL with all parts looks like:

```
scheme://lo.ca.ti.on/pa/th?qu=ery#fragment
```

In *https://www.python.org/community/awards/psf-awards/#october-2016*, for example, the scheme is *http*, the location is *www.python.org*, the path is */community/awards/psf-awards/*, there is no query, and the fragment is *#october-2016*. (Most schemes default to a *well-known port* when the port is not explicitly specified; for example, 80 is the well-known port for the HTTP scheme.) Some punctuation is part of one of the components it separates; other punctuation characters are just separators, not part of any component. Omitting punctuation implies missing components. For example, in *mailto:me@you.com*, the scheme is *mailto*, the path is *me@you.com* (*mailto:me@you.com*), and there is no location, query, or fragment. No // means the URI has no location, no ? means it has no query, and no # means it has no fragment.

If the location ends with a colon followed by a number, this denotes a TCP port for the endpoint. Otherwise, the connection uses the well-known port associated with the scheme (e.g., port 80 for HTTP).

The urllib Package

The `urllib` package supplies several modules for parsing and utilizing URL strings and associated resources. In addition to the `urllib.parse` and `urllib.request` modules described here, these include the module `urllib.robotparser` (for the specific purpose of parsing a site's *robots.txt* file as per **RFC 9309**) and the module `urllib.error`, containing all exception types raised by other `urllib` modules.

The urllib.parse module

The `urllib.parse` module supplies functions for analyzing and synthesizing URL strings, and is typically imported with **`from`** `urllib` **`import`** `parse` **`as`** `urlparse`. Its most frequently used functions are listed in **Table 19-3**.

Table 19-3. Useful functions of the `urllib.parse` module

`urljoin`	`urljoin(`*`base_url_string`*`,` *`relative_url_string`*`)` Returns a URL string *u*, obtained by joining *`relative_url`* which may be relative, with *`base_url_string`*. The joinin procedure that `urljoin` performs to obtain its result may summarized as follows: • When either of the argument strings is empty, *u* is t argument. • When *`relative_url_string`* explicitly specifies a sc that is different from that of *`base_url_string`*, *u* is *`relative_url_string`*. Otherwise, *u*'s scheme is tha *`base_url_string`*. • When the scheme does not allow relative URLs (e.g. or when *`relative_url_string`* explicitly specifies a (even when it is the same as the location of *`base_url_string`*), all other components of *u* are th *`relative_url_string`*. Otherwise, *u*'s location is tha *`base_url_string`*. • *u*'s path is obtained by joining the paths of *`base_url`* and *`relative_url_string`* according to standard sy

absolute and relative URL paths.[a] For example:

```
urlparse.urljoin(
  'http://host.com/some/path/here','../other/pa
# Result is: 'http://host.com/some/other/path'
```

urlsplit

urlsplit(*url_string*, default_scheme='', allow_fragments=**True**)

Analyzes *url_string* and returns a tuple (actually an inst SplitResult, which you can treat as a tuple or use with r attributes) with five string items: *scheme*, *netloc*, *path*, *qu fragment*. *default_scheme* is the first item when the *url_* lacks an explicit scheme. When allow_fragments is **False** tuple's last item is always '', whether or not *url_string* fragment. Items corresponding to missing parts are also example:

```
urlparse.urlsplit(
  'http://www.python.org:80/faq.cgi?src=file')
# Result is:
# 'http','www.python.org:80','/faq.cgi','src=fi
```

urlunsplit

urlunsplit(*url_tuple*)

url_tuple is any iterable with exactly five items, all strin return value from a urlsplit call is an acceptable argum urlunsplit. urlunsplit returns a URL string with the giv components and the needed separators, but with no redu separators (e.g., there is no # in the result when the fragm *url_tuple*'s last item, is ''). For example:

```
urlparse.urlunsplit((
```

```
    'http','www.python.org','/faq.cgi','src=fie',
# Result is: 'http://www.python.org/faq.cgi?src
```

`urlunsplit(urlsplit(x))` returns a normalized form of string *x*, which is not necessarily equal to *x* because *x* nee normalized. For example:

```
urlparse.urlsplit('http://a.com/path/a?'))
# Result is: 'http://a.com/path/a'
```

In this case, the normalization ensures that redundant se such as the trailing `?` in the argument to `urlsplit`, are no in the result.

[a] Per RFC 1808.

The urllib.request module

The `urllib.request` module supplies functions for accessing data resources over standard internet protocols, the most commonly used of which are listed in Table 19-4. (The examples in the table assume you've imported the module.)

Table 19-4. Useful functions of the `urllib.request` module

`urlopen`	`urlopen(`*`url`*`, data=None, timeout, context=None)` Returns a response object whose type depends on the scher in *`url`*: • HTTP and HTTPS URLs return an `http.client.HTTPResponse` object (with the `msg` attribute modified to contain the same data as the `reason` attribute; for details, see the online docs). Your code

can use this object like an iterable, and as a context manager in a **with** statement.

- FTP, file, and data URLs return a `urllib.response.addinfourl` object.
  url is the string or `urllib.request.Request` object fo the URL to open. *data* is an optional `bytes` object, file-like object, or iterable of bytes, encoding additional data to send to the URL following *application/x-www form-urlencoded* format. *timeout* is an optional argu· ment for specifying, in seconds, a timeout for blockin operations of the URL opening process, applicable onl for HTTP, HTTPS, and FTP URLs. When *context* is give it must contain an `ssl.SSLContext` object specifying SSL options; *context* replaces the deprecated *cafile*, *capath*, and *cadefault* arguments. The following exa ple downloads a file from an HTTPS URL and extracts into a local `bytes` object, `unicode_db`:

```
unicode_url = ("https://www.unicode.org/Public"
               "/14.0.0/ucd/UnicodeData.txt")
with urllib.request.urlopen(unicode_url
     )as url_response:
    unicode_db = url_response.read()
```

url retrieve	urlretrieve(*url_string*, filename=**None**, report_hook=**None**, data=**None**) A compatibility function to support migration from Python legacy code. *url_string* gives the URL of the resource to download. `filename` is an optional string naming the local file in which to store the data retrieved from the URL. `report_hook` is a callable to support progress reporting during downloading, called once as each block of data is retrieved. `data` is similar to the `data` argument for `urlopen`. In its simplest form, `urlretrieve` is equivalent to:

```
def urlretrieve(url, filename=None):
    if filename is None:
        filename = ...parse filename from url...
    with urllib.request.urlopen(url
        )as url_response:
        with open(filename, "wb") as save_file:
            save_file.write(url_response.read())
        return filename, url_response.info()
```

> Since this function was developed for Python 2 compatibilit
> you may still see it in existing codebases. New code should
> use `urlopen`.

For full coverage of `urllib.request` see the **online docs** and Michael Foord's **HOWTO**, which includes examples on downloading files given a URL. There's a short example using `urllib.request` in **"An HTML Parsing Example with BeautifulSoup"**.

The Third-Party requests Package

The third-party **`requests` package** (very well documented **online**) is how we recommend you access HTTP URLs. As usual for third-party packages, it's best installed with a simple **`pip install requests`**. In this section, we summarize how best to use it for reasonably simple cases.

Natively, `requests` only supports the HTTP and HTTPS transport protocols; to access URLs using other protocols, you need to install other third-party packages (known as *protocol adapters*), such as **requests-ftp** for FTP URLs, or others supplied as part of the rich **requests-toolbelt** package of `requests` utilities.

The `requests` package's functionality hinges mostly on three classes it supplies: `Request`, modeling an HTTP request to be sent to a server; `Response`, modeling a server's HTTP response to a request; and `Session`, offering continuity across a sequence of requests, also known as a *ses-*

sion. For the common use case of a single request/response interaction, you don't need continuity, so you may often just ignore `Session`.

Sending requests

Typically, you don't need to explicitly consider the `Request` class: rather, you call the utility function `request`, which internally prepares and sends the `Request` and returns the `Response` instance. `request` has two mandatory positional arguments, both `str`s: `method`, the HTTP method to use, and `url`, the URL to address. Then, many optional named parameters may follow (in the next section, we cover the most commonly used named parameters to the `request` function).

For further convenience, the `requests` module also supplies functions whose names are those of the HTTP methods `delete`, `get`, `head`, `options`, `patch`, `post`, and `put`; each takes a single mandatory positional argument, `url`, then the same optional named arguments as the function `request`.

When you want some continuity across multiple requests, call `Session` to make an instance `s`, then use `s`'s methods `request`, `get`, `post`, and so on, which are just like the functions with the same names directly supplied by the `requests` module (however, `s`'s methods merge `s`'s settings with the optional named parameters to prepare each request to send to the given `url`).

request's optional named parameters

The function `request` (just like the functions `get`, `post`, and so on, and methods with the same names on an instance `s` of class `Session`) accepts many optional named parameters. Refer to the `requests` package's excellent **online docs** for the full set if you need advanced functionality such as control over proxies, authentication, special treatment of redirection, streaming, cookies, and so on. **Table 19-5** lists the most frequently used named parameters.

Table 19-5. Named parameters accepted by the `request` function

`data`	A `dict`, a sequence of key/value pairs, a bytestring, or a file-like object to use as the body of the request
`files`	A `dict` with names as keys and file-like objects or *file tuples* as values, used with the POST method to specify a multipart-encoding file upload (we cover the format of values for `files` in the next section)
`headers`	A `dict` of HTTP headers to send in the request
`json`	Python data (usually a `dict`) to encode as JSON as the body of the request
`params`	A `dict` of (*`name, value`*) items, or a bytestring to send as the query string with the request
`timeout`	A `float` number of seconds, the maximum time to wait for the response before raising an exception

`data`, `json`, and `files` are mutually incompatible ways to specify a body for the request; you should normally use at most one of them, and only for HTTP methods that do use a body (namely PATCH, POST, and PUT). The one exception is that you can have both a `data` argument passing a `dict` and a `files` argument. That is very common usage: in this case, both the key/value pairs in the `dict` and the files form the body of the request as a single *multipart/form-data* whole.[4]

The files argument (and other ways to specify the request's body)

When you specify the request's body with `json` or `data` (passing a bytestring or a file-like object, which must be open for reading, usually in binary mode), the resulting bytes are directly used as the request's body. When you specify it with `data` (passing a `dict` or a sequence of key/value pairs), the body is built as a *form*, from the key/value pairs formatted in

application/x-www-form-urlencoded format, according to the relevant **web standard**.

When you specify the request's body with `files`, the body is also built as a form, in this case with the format set to *multipart/form-data* (the only way to upload files in a PATCH, POST, or PUT HTTP request). Each file you're uploading is formatted into its own part of the form; if, in addition, you want the form to give to the server further nonfile parameters, then in addition to `files`, you need to pass a `data` argument with a `dict` value (or a sequence of key/value pairs) for the further parameters. Those parameters get encoded into a supplementary part of the multipart form.

For flexibility, the value of the `files` argument can be a `dict` (its items are taken as a sequence of (*`name, value`*) pairs), or a sequence of (*`name, value`*) pairs (order is maintained in the resulting request body).

Either way, each value in a (*`name, value`*) pair can be a `str` (or, better,[5] a `bytes` or `bytearray`) to be used directly as the uploaded file's contents, or a file-like object open for reading (then, `requests` calls `.read()` on it and uses the result as the uploaded file's contents; we strongly urge that in such cases you open the file in binary mode to avoid any ambiguity regarding content length). When any of these conditions apply, `requests` uses the *`name`* part of the pair (e.g., the key into the `dict`) as the file's name (unless it can improve on that because the open file object is able to reveal its underlying filename), takes its best guess at a content type, and uses minimal headers for the file's form part.

Alternatively, the value in each (*`name, value`*) pair can be a tuple with two to four items, (*`fn, fp`*`[, `*`ft`*`[, `*`fh`*`]]`) (using square brackets as meta-syntax to indicate optional parts). In this case, *`fn`* is the file's name, *`fp`* provides the contents (in just the same way as in the previous paragraph), optional *`ft`* provides the content type (if missing, `requests` guesses it, as in the previous paragraph), and the optional `dict` *`fh`* provides extra headers for the file's form part.

How to interpret requests examples

In practical applications, you don't usually need to consider the internal instance *r* of the class `requests.Request`, which functions like `requests.post` is building, preparing, and then sending on your behalf. However, to understand exactly what `requests` is doing, working at a lower level of abstraction (building, preparing, and examining *r*—no need to send it!) is instructive. For example, after importing `requests`, passing `data` as in the following example:

```
r = requests.Request('GET', 'http://www.example.com',
    data={'foo': 'bar'}, params={'fie': 'foo'})
p = r.prepare()
print(p.url)
print(p.headers)
print(p.body)
```

prints out (splitting the *p*`.headers` `dict`'s printout for readability):

```
http://www.example.com/?fie=foo
{'Content-Length': '7',
 'Content-Type': 'application/x-www-form-urlencoded'}
foo=bar
```

Similarly, when passing `files`:

```
r = requests.Request('POST', 'http://www.example.com',
    data={'foo': 'bar'}, files={'fie': 'foo'})
p = r.prepare()
print(p.headers)
print(p.body)
```

this prints out (with several lines split for readability):

```
{'Content-Length': '228',
 'Content-Type': 'multipart/form-data; boundary=dfd600d8aa58496270'}
b'--dfd600d8aa58496270\r\nContent-Disposition: form-data;
="foo"\r\n\r\nbar\r\n--dfd600d8aa584962709b936134b1cfce\r\n
Content-Disposition: form-data; name="fie" filename="fie"\r\n\r\nfoo\r\n
--dfd600d8aa584962709b936134b1cfce--\r\n'
```

Happy interactive exploring!

The Response class

The one class from the `requests` module that you always have to consider is `Response`: every request, once sent to the server (typically, that's done implicitly by methods such as `get`), returns an instance *`r`* of `requests.Response`.

The first thing you usually want to do is to check *`r`*`.status_code`, an `int` that tells you how the request went, in typical "HTTPese": `200` means "everything's fine," `404` means "not found," and so on. If you'd rather just get an exception for status codes indicating some kind of error, call *`r`*`.raise_for_status`; that does nothing if the request went fine, but raises `requests.exceptions.HTTPError` otherwise. (Other exceptions, not corresponding to any specific HTTP status code, can and do get raised without requiring any such explicit call: e.g., `ConnectionError` for any kind of network problem, or `TimeoutError` for a timeout.)

Next, you may want to check the response's HTTP headers: for that, use *`r`*`.headers`, a `dict` (with the special feature of having case-insensitive string-only keys indicating the header names as listed, e.g., in **Wikipedia**, per the HTTP specs). Most headers can be safely ignored, but sometimes you'd rather check. For example, you can verify whether the response specifies which natural language its body is written in, via *`r`*`.headers.get('content-language')`, to offer different presentation choices, such as the option to use some kind of language translation service to make the response more usable for the user.

You don't usually need to make specific status or header checks for redirects: by default, `requests` automatically follows redirects for all methods except HEAD (you can explicitly pass the `allow_redirection` named parameter in the request to alter that behavior). If you allow redirects, you may want to check `r.history`, a list of all `Response` instances accumulated along the way, oldest to newest, up to but excluding `r` itself (`r.history` is empty if there have been no redirects).

Most often, maybe after checking status and headers, you want to use the response's body. In simple cases, just access the response's body as a bytestring, `r.content`, or decode it as JSON (once you've checked that's how it's encoded, e.g., via `r.headers.get('content-type')`) by calling `r.json`.

Often, you'd rather access the response's body as (Unicode) text, with the property `r.text`. The latter gets decoded (from the octets that actually make up the response's body) with the codec `requests` thinks is best, based on the Content-Type header and a cursory examination of the body itself. You can check what codec has been used (or is about to be used) via the attribute `r.encoding`; its value will be the name of a codec registered with the `codecs` module, covered in **"The codecs Module"**. You can even *override* the choice of codec to use by *assigning* to `r.encoding` the name of the codec you choose.

We do not cover other advanced issues, such as streaming, in this book; see the `requests` package's **online docs** for further information.

Other Network Protocols

Many, *many* other network protocols are in use—a few are best supported by Python's standard library, but for most of them you'll find better and more recent third-party modules on **PyPI**.

To connect as if you were logging in to another machine (or a separate login session on your own node), you can use the **Secure Shell (SSH)** protocol, supported by the third-party module **`paramiko`** or the higher abstrac-

tion layer wrapper around it, the third-party module **spur**. (You can also, with some likely security risks, still use classic **Telnet**, supported by the standard library module **telnetlib**.)

Other network protocols include, among many others:

- **NNTP**, to access Usenet News servers, supported by the standard library module `nntplib`
- **XML-RPC**, for a rudimentary remote procedure call functionality, supported by **xmlrpc.client**
- **gRPC**, for a more modern remote procedure functionality, supported by third-party module **grpcio**
- **NTP**, to get precise time off the network, supported by third-party module **ntplib**
- **SNMP**, for network management, supported by third-party module **pysnmp**

No single book (not even this one!) could possibly cover all these protocols and their supporting modules. Rather, our best suggestion in the matter is a strategic one: whenever you decide that your application needs to interact with some other system via a certain networking protocol, don't rush to implement your own modules to support that protocol. Instead, search and ask around, and you're likely to find excellent existing Python modules (third-party or standard-library ones) supporting that protocol.[6]

Should you find some bug or missing feature in such modules, open a bug or feature request (and, ideally, supply a patch or pull request that would fix the problem and satisfy your application's needs). In other words, become an active member of the open source community, rather than just a passive user: you will be welcome there, scratch your own itch, and help many others in the process. "Give forward," since you cannot "give back" to all the awesome people who contributed to give you most of the tools you're using!

1 IMAP4, per **RFC 1730**; or IMAP4rev1, per **RFC 2060**.

2 The specification of the POP protocol can be found in **RFC 1939**.

3 The specification of the SMTP protocol can be found in **RFC 2821**.

4 According to **RFC 2388**.

5 As it gives you complete, explicit control of exactly what octets are uploaded.

6 Even more importantly, if you think you need to invent a brand-new protocol and implement it on top of sockets, think again, and search carefully: it's far more likely that one or more of the huge number of existing internet protocols meets your needs just fine!