7
Networking

In this chapter, you will be taught how to communicate with internet servers and with sockets in general. First, we will take a look at `QNetworkAccessManager`, which makes sending network requests and receiving replies really easy. Building on this basic knowledge, we will then use Google's Distance API to get information about the distance between two locations and how long it would take to get from one to the other. This technique, and the respective knowledge, can also be used to include Facebook or Twitter in your application via their respective APIs. Then, we will take a look at Qt's Bearer API, which provides information about a device's connectivity state. In the last section, you will learn how to use sockets to create your own server and clients using TCP or UDP as the network protocol.

The main topics covered in this chapter are the following:

- Downloading files using `QNetworkAccessManager`
- Using Google's Distance Matrix API
- Implementing a TCP chat server and client
- Using UDP sockets

QNetworkAccessManager

All network-related functionality in Qt is implemented in the Qt Network module. The easiest way to access files on the internet is to use the `QNetworkAccessManager` class, which handles the complete communication between your game and the internet.

Setting up a local HTTP server

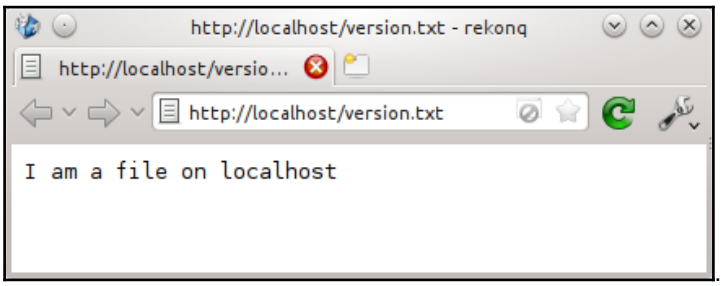
In our next example, we will be downloading a file over HTTP. If you don't have a local HTTP server, you can just use any publicly available HTTP or HTTPS resource to test your code. However, when you develop and test a network-enabled application, it is recommended that you use a private, local network if feasible. This way, it is possible to debug both ends of the connection, and errors will not expose sensitive data.

If you are not familiar with setting up a web server locally on your machine, there are, luckily, a number of all-in-one installers that are freely available. These will automatically configure Apache2, MySQL (or MariaDB), PHP, and many other servers on your system. On Windows, for example, you can use XAMPP (`https://www.apachefriends.org`), or the Uniform Server (`http://www.uniformserver.com`); on Apple computers, there is MAMP (`https://www.mamp.info`); and on Linux, you can open your preferred package manager, search for a package called `Apache2` or a similar one, and install it. Alternatively, take a look at your distribution's documentation.

Before you install Apache on your machine, think about using a virtual machine, such as VirtualBox (`http://www.virtualbox.org`) for this task. This way, you keep your machine clean, and you can easily try different settings for your test server. With multiple virtual machines, you can even test the interaction between different instances of your game. If you are on Unix, Docker (`http://www.docker.com`) might be worth taking a look at.

Preparing a URL for testing

If you've set up a local HTTP server, create a file called `version.txt` in the root directory of the installed server. This file should contain a small piece of text such as "I am a file on localhost" or something similar. As you might have guessed, a real-life scenario could be to check whether there is an updated version of your game or application on the server. To test whether the server and the file are correctly set up, start a web browser and open `http://localhost/version.txt`. You should then see the file's content:



If this fails, it may be the case that your server does not allow you to display text files. Instead of getting lost in the server's configuration, just rename the file to `version.html`. This should do the trick!

If you don't have an HTTP server, you can use the URL of your favorite website, but be prepared to receive HTML code instead of plain text, as the majority of websites use HTML. You can also use the `https://www.google.com/robots.txt` URL, as it responds with plain text.

Time for action – Downloading a file

Create a Qt Widgets project and add a widget class named `FileDownload`. Add a button that will start the download and a plain text edit that will display the result. As always, you can look at the code files provided with the book if you need any help.

Next, enable the Qt Network module by adding `QT += network` to the project file. Then, create an instance of `QNetworkAccessManager` in the constructor and put it in a private field:

```
m_network_manager = new QNetworkAccessManager(this);
```

Since `QNetworkAccessManager` inherits `QObject`, it takes a pointer to `QObject`, which is used as a parent. Thus, you do not have delete the manager later on.

Secondly, we connect the manager's `finished()` signal to a slot of our choice; for example, in our class, we have a slot called `downloadFinished()`:

```
connect(m_network_manager, &QNetworkAccessManager::finished,
        this, &FileDownload::downloadFinished);
```

We have to do this because the API of `QNetworkAccessManager` is *asynchronous*. This means that none of the network requests, or the read or write operations, will block the current thread. Instead, when the data is available or another network event occurs, Qt will send a corresponding signal so that you can handle the data.

Thirdly, we actually request the `version.txt` file from localhost when the button is clicked:

```
QUrl url("http://localhost/version.txt");
m_network_manager->get(QNetworkRequest(url));
```

With `get()`, we send a request to get the contents of the file specified by the URL. The function expects a `QNetworkRequest` object, which defines all the information needed to send a request over the network. The main information for such a request is, naturally the URL of the file. This is the reason `QNetworkRequest` takes `QUrl` as an argument in its constructor. You can also set the URL with `setUrl()` to a request. If you wish to define a request header (for example, a custom user agent), you can use `setHeader()`:

```
QNetworkRequest request;
request.setUrl(QUrl("http://localhost/version.txt"));
request.setHeader(QNetworkRequest::UserAgentHeader, "MyGame");
m_network_manager->get(request);
```

The `setHeader()` function takes two arguments: the first is a value of the `QNetworkRequest::KnownHeaders` enumeration, which holds the most common (self-explanatory) headers, such as `LastModifiedHeader` or `ContentTypeHeader`, and the second is the actual value. You can also write the header using `setRawHeader()`:

```
request.setRawHeader("User-Agent", "MyGame");
```

When you use `setRawHeader()`, you have to write the header field names yourself. Besides this, it behaves like `setHeader()`. A list of all the available headers for the HTTP protocol Version 1.1 can be found in section 14 of RFC 2616 (`https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14`).

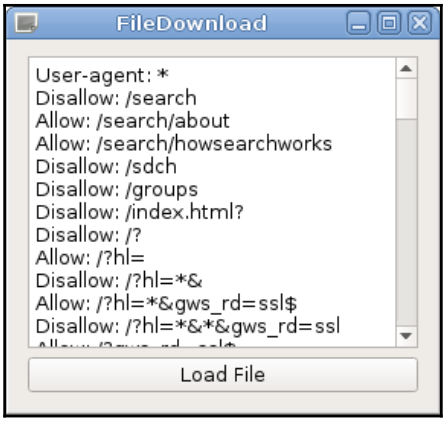
Getting back to our example, with the `get()` function, we requested the `version.txt` file from the localhost. All we have to do from now on is wait for the server to reply. As soon as the server's reply is finished, the `downloadFinished()` slot will be called that was defined by the preceding connection statement. A pointer to a `QNetworkReply` object will be passed as the argument to the slot, and we can read the reply's data and show it in `m_edit`, an instance of `QPlainTextEdit`, with the following:

```
void FileDownload::downloadFinished(QNetworkReply *reply) {
    const QByteArray content = reply->readAll();
    m_edit->setPlainText(QString::fromUtf8(content));
    reply->deleteLater();
}
```

Since `QNetworkReply` inherits `QIODevice`, there are also other possibilities to read the content of the reply. For example, you can use `QDataStream` or `QTextStream` to read and interpret binary or textual data, respectively. Here, as the fourth command, `QIODevice::readAll()` is used to get the full content of the requested file in a `QByteArray` object. This is very similar to reading from files, which was shown in the previous chapter. The responsibility for the transferred pointer to the corresponding `QNetworkReply` lies with us, so we need to delete it at the end of the slot. However, be careful and do not call `delete` on the reply directly. Always use `deleteLater()`, as the documentation suggests!

In the previous chapter, we warned you that you shouldn't use `readAll()` to read large files, as they can't fit in a single `QByteArray`. The same holds for `QNetworkReply`. If the server decides to send you a large response (for example, if you try to download a large file), the first portion of the response will be saved to a buffer inside the `QNetworkReply` object, and then the download will throttle down until you read some data from the buffer. However, you can't do that if you only use the `finished()` signal. Instead, you need to use the `QNetworkReply::readyRead()` signal and read each portion of the data in order to free the buffer and allow more data to be received. We will show how to do this later in this chapter.

The full source code can be found in the **FileDownload** example bundled with this book. If you start the small demo application and click on the **Load File** button, you should see the content of the loaded file:



Have a go hero – Extending the basic file downloader

Of course, having to alter the source code in order to download another file is far from an ideal approach, so try to extend the dialog by adding a line edit in which you can specify the URL you want to download. Also, you can offer a file dialog to choose the location where the downloaded file will be saved. The simplest way of doing that is to use the `QFileDialog::getSaveFileName()` static function.

Single network manager per application

One single instance of `QNetworkAccessManager` is enough for an entire application. For example, you can create an instance of `QNetworkAccessManager` in your main window class and pass a pointer to it to all the other places where it's needed. For ease of use, you can also create a **singleton** and access the manager through that.

A singleton pattern ensures that a class is instantiated only once. The pattern is useful for accessing application-wide configurations or—as in our case—an instance of `QNetworkAccessManager`.

A simple template-based approach to create a singleton will look like this (as a header file):

```
template <class T>
class Singleton
{
public:
    static T& instance()
    {
        static T static_instance;
        return static_instance;
    }
private:
    Singleton();
    ~Singleton();
    Singleton(const Singleton &);
    Singleton& operator=(const Singleton &);
};
```

In the source code, you will include that header file and acquire a singleton of a class called `MyClass` with this:

```
MyClass &singleton = Singleton<MyClass>::instance();
```

This singleton implementation is not *thread-safe*, meaning that attempting to access the instance from multiple threads simultaneously will result in undefined behavior. An example of thread-safe implementation of the singleton pattern can be found at `https://wiki.qt.io/Qt_thread-safe_singleton`.

If you are using Qt Quick—it will be explained in `Chapter 11`, *Introduction to Qt Quick*—with `QQmlApplicationEngine`, you can directly use the engine's instance of `QNetworkAccessManager`:

```
QQmlApplicationEngine engine;
QNetworkAccessManager *network_manager = engine.networkAccessManager();
```

Time for action – Displaying a proper error message

If you do not see the content of the file, something went wrong. Just as in real life, this can often happen. So, we need to ensure that there is a good error handling mechanism in such cases to inform the user about what is going on. Fortunately, `QNetworkReply` offers several possibilities to do this.

In the slot called `downloadFinished()`, we first want to check whether an error occurred:

```
if (reply->error() != QNetworkReply::NoError) {
    // error occurred
}
```

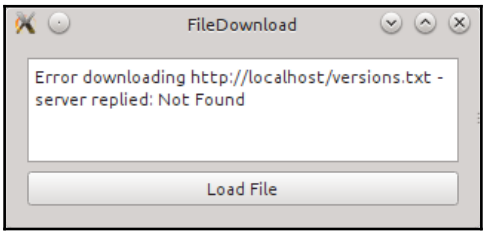
The `QNetworkReply::error()` function returns the error that occurred while handling the request. The error is encoded as a value of the `QNetworkReply::NetworkError` type. The most common errors are probably these:

Error code	**Meaning**
`QNetworkReply::ConnectionRefusedError`	The program was not able to connect to the server at all (for example, if no server was running)
`QNetworkReply::ContentNotFoundError`	The server responded with HTTP error code 404, indicating that a page with the requested URL could not be found
`QNetworkReply::ContentAccessDenied`	The server responded with HTTP error code 403, indicating that you do not have the permission to access the requested file

There are more than 30 possible error types, and you can look them up in the documentation of the `QNetworkReply::NetworkError` enumeration. However, normally, you do not need to know exactly what went wrong. You only need to know whether everything worked out—`QNetworkReply::NoError` would be the return value in this case—or whether something went wrong. To provide the user with a meaningful error description, you can use `QIODevice::errorString()`. The text is already set up with the corresponding error message, and we only have to display it:

```
if (reply->error()) {
    const QString error = reply->errorString();
    m_edit->setPlainText(error);
    return;
}
```

In our example, assuming that we made an error in the URL and wrote `versions.txt` by mistake, the application will look like this:



If the request was an HTTP request and the status code is of interest, it can be retrieved by `QNetworkReply::attribute()`:

```
int statusCode =
     reply->attribute(QNetworkRequest::HttpStatusCodeAttribute).toInt();
```

Since it returns `QVariant`, you need to use `QVariant::toInt()` to get the code as an integer. Besides the HTTP status code, you can query a lot of other information through `attribute()`. Take a look at the description of the `QNetworkRequest::Attribute` enumeration in the documentation. There, you will also find `QNetworkRequest::HttpReasonPhraseAttribute`, which holds a human-readable reason phrase for the HTTP status code, for example, "Not Found" if an HTTP error 404 has occurred. The value of this attribute is used to set the error text for `QIODevice::errorString()`. So, you can either use the default error description provided by `errorString()` or compose your own by interpreting the reply's attributes.

If a download failed and you want to resume it, or if you only want to download a specific part of a file, you can use the `Range` header. However, the server must support this.

In the following example, only the bytes from `300` to `500` will be downloaded:

```
QNetworkRequest request(url);
request.setRawHeader("Range", "bytes=300-500");
QNetworkReply *reply = m_network_manager->get(request);
```

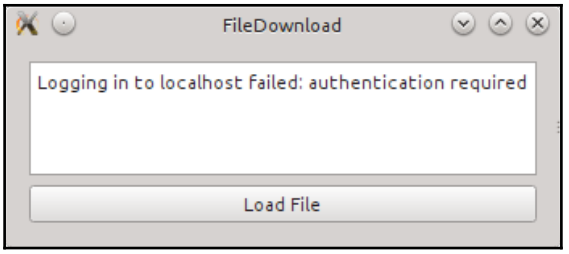
If you want to simulate sending a form on a website, you will usually need to send a POST request instead of GET. This is done by using the `QNetworkAccessManager::post()` function instead of the `get()` function we used. You will also need to specify the payload, for example, using the `QHttpMultiPart` class.

Downloading files over FTP

Downloading a file over FTP is as simple as downloading files over HTTP. If it is an anonymous FTP server for which you do not need an authentication, just use the URL as we did before. Assuming that there is again a file called `version.txt` on the FTP server on the localhost, type this:

```
m_network_manager->get(QNetworkRequest(
    QUrl("ftp://localhost/version.txt")));
```

That's all; everything else stays the same. If the FTP server requires an authentication, you'll get an error; consider this example:



Likewise, setting the username and password to access an FTP server is easy—either write it in the URL, or use the `setUserName()` and `setPassword()` functions of `QUrl`. If the server does not use a standard port, you can set the port explicitly with `QUrl::setPort()`.

To upload a file to an FTP server, use `QNetworkAccessManager::put()`, which takes `QNetworkRequest` as its first argument, calling a URL that defines the name of the new file on the server, and the actual data as its second argument, which should be uploaded. For small uploads, you can pass the content as `QByteArray`. For larger content, it's better to use a pointer to `QIODevice`. Ensure that the device is open and stays available until the upload is complete.

Downloading files in parallel

A very important note concerning `QNetworkAccessManager` is the fact that it works asynchronously. This means that you can post a network request without blocking the main event loop, and this is what keeps the GUI responsive. If you post more than one request, they are put in the manager's queue. Depending on the protocol used, they may be processed in parallel. If you are sending HTTP requests, normally up to six requests will be handled at a time. If more requests are queued, they will be automatically processed later. This will not block the application, as `QNetworkAccessManager` uses threads internally.

There is really no need to encapsulate `QNetworkAccessManager` in a thread; however, unfortunately, this unnecessary approach is frequently recommended all over the internet. Really, don't move `QNetworkAccessManager` to a thread unless you know exactly what you are doing.

If you send multiple requests, the slot connected to the manager's `finished()` signal is called in an arbitrary order, depending on how quickly a request gets a reply from the server. This is why you need to know to which request a reply belongs. This is one reason why every `QNetworkReply` carries its related `QNetworkRequest`. It can be accessed through `QNetworkReply::request()`.

Even if the determination of the replies and their purpose may work for a small application in a single slot, it will quickly get large and confusing if you send a lot of requests with different purposes. It would be better to connect requests to multiple slots that are specialized for a given task. Fortunately, this can be achieved very easily.

Any method that adds a request to `QNetworkAccessManager` (such as `get()`) returns a pointer to `QNetworkReply`. Using this pointer, you can then connect the reply's signals to your specific slots. For example, if you have several URLs, and you want to save all linked images from these sites to your hard drive, you request all web pages via `QNetworkAccessManager::get()` and connect their replies to a slot specialized for parsing the received HTML. If links to the images are found, this slot will request them again with `get()`. This time, however, the replies to these requests will be connected to a second slot, which is designed for saving the images to the disk. Thus, you can separate the two tasks: parsing HTML and saving data to a local drive.

The most important signals of `QNetworkReply` are discussed next.

The finished signal

The `finished()` signal is an equivalent of the `QNetworkAccessManager::finished()` signal that we used earlier. It is triggered as soon as a reply is returned—successfully or not. After this signal is emitted, neither the reply's data nor its metadata will be altered any more. With this signal, you are now able to connect a reply to a specific slot. This way, you can realize the scenario on saving images that was outlined in the previous section.

However, one problem remains: if you post simultaneous requests, you do not know which one has finished and thus called the connected slot. Unlike `QNetworkAccessManager::finished()`, `QNetworkReply::finished()` does not pass a pointer to `QNetworkReply`; this would actually be a pointer to itself in this case. We've already had a similar problem in `Chapter 3`, *Qt GUI Programming*, so let's remember how we can deal with it.

A quick solution to solve this problem is to use `sender()`. It returns a pointer to the `QObject` instance that has called the slot. Since we know that it was `QNetworkReply`, we can write the following:

```
QNetworkReply *reply = qobject_cast<QNetworkReply*>(sender());
if (!reply) {
    return;
}
```

In this code, we needed to cast the `QObject` pointer returned by `sender()` to a pointer of the `QNetworkReply` type.

Whenever you're casting classes that inherit `QObject`, use `qobject_cast`. Unlike `dynamic_cast`, it does not use RTTI and works across the dynamic library boundaries.

Although we can be pretty confident that the cast will work, do not forget to check whether the pointer is valid. If it is a null pointer, exit the slot.

Time for action – Writing the OOP conform code using QSignalMapper

A more elegant way that does not rely on `sender()` would be to use `QSignalMapper` to receive the reply object in the argument of the slot. First, you need to add the `QSignalMapper *m_imageFinishedMapper` private field to your class. When you call `QNetworkAccessManager::get()` to request each image, set up the mapper as follows:

```
for(const QString& url: urls) {
    QNetworkRequest request(url);
    QNetworkReply *reply = m_network_manager->get(request);
    connect(reply, SIGNAL(finished()),
            m_imageFinishedMapper, SLOT(map()));
    m_imageFinishedMapper->setMapping(reply, reply);
}
```

In a prominent place, most likely the constructor of the class, connect the mapper's `map()` signal to a custom slot. Take this example into consideration:

```
connect(m_imageFinishedMapper, SIGNAL(mapped(QObject*)),
        this, SLOT(imageFinished(QObject*)));
```

Now your slot receives the reply object as the argument:

```
void Object::imageFinished(QObject *replyObject)
{
    QNetworkReply *reply = qobject_cast<QNetworkReply *>(replyObject);
    //...
}
```

What just happened?

First, we posted the request and fetched the pointer to the `QNetworkReply` object. Then, we connected the reply's finished signal to the mapper's slot `map()`. Next, we called the `setMapping()` method of the mapper to indicate that the sender itself should be sent as the slot's argument. The effect is very similar to the direct use of the `QNetworkAccessManager::finished(QNetworkReply *reply)` signal, but this way, we can use multiple slots dedicated to different purposes (with a separate mapper corresponding to each slot), all served by a single `QNetworkAccessManager` instance.

`QSignalMapper` also allows you to map with `int` or `QString` as an identifier instead of `QObject *`, as used in the preceding code. So, you can rewrite the example and use the URL to identify the corresponding request.

The error signal

Instead of dealing with errors in the slot connected to the `finished()` signal, you can use the reply's `error()` signal, which passes the error of the `QNetworkReply::NetworkError` type to the slot. After the `error()` signal has been emitted, the `finished()` signal will, most likely, also be emitted shortly.

The readyRead signal

Until now, we have used the slot connected to the `finished()` signal to get the reply's content. This works perfectly if you are deal with small files. However, this approach is unsuitable when dealing with large files, as they will unnecessarily bind too many resources. For larger files, it is better to read and save the transferred data as soon as it is available. We are informed by `QIODevice::readyRead()` whenever new data is available to be read. So, for large files, you should use the following code:

```
QNetworkReply *reply = m_network_manager->get(request);
connect(reply, &QIODevice::readyRead,
        this, &SomeClass::readContent);
m_file.open(QIODevice::WriteOnly);
```

This will help you connect the reply's `readyRead()` signal to a slot, set up `QFile`, and open it. In the connected slot, type in the following snippet:

```
QNetworkReply *reply = /* ... */;
const QByteArray byteArray = reply->readAll();
m_file.write(byteArray);
m_file.flush();
```

Now, you can fetch the content, which has been transferred so far, and save it to the (already open) file. This way, the resources needed are minimized. Don't forget to close the file after the `finished()` signal is emitted.

In this context, it would be helpful if you knew upfront the size of the file you want to download. With this information, we can check upfront whether there is enough space left on the disk. We can use `QNetworkAccessManager::head()` for this purpose. It behaves like the `get()` function, but it does not request the content of the file. Only the headers are transferred, and if we are lucky, the server sends the `Content-Length` header, which holds the file size in bytes. To get that information, we type this:

```
int length = reply->header(QNetworkRequest::ContentLengthHeader).toInt();
```

Time for action – Showing the download progress

Especially when a big file is downloaded, the user usually wants to know how much data has already been downloaded and approximately how long it will take for the download to finish.

In order to achieve this, we can use the reply's `downloadProgress()` signal. As the first argument, it passes the information on how many bytes have already been received and as the second argument, how many bytes there are in total. This gives us the possibility to indicate the progress of the download with `QProgressBar`. As the passed arguments are of the `qint64` type, we can't use them directly with `QProgressBar`, as it only accepts `int`. So, in the connected slot, we can do the following:

```
void SomeClass::downloadProgress(qint64 bytesReceived, qint64 bytesTotal) {
    qreal progress = (bytesTotal < 1) ? 1.0
                   : static_cast<qreal>(bytesReceived) / bytesTotal;
    progressBar->setValue(qRound(progress * progressBar->maximum()));
}
```

What just happened?

First, we calculate the percentage of the download progress. The calculated `progress` value will range from 0 (0%) to 1 (100%). Then, we set the new value for the progress bar where `progressBar` is the pointer to this bar. However, what value will `progressBar->maximum()` have and where do we set the range for the progress bar? What is nice is that you do not have to set it for every new download. It is only done once, for example, in the constructor of the class containing the bar. As range values, we would recommend this:

```
progressBar->setRange(0, 2048);
```

The reason is that if you take, for example, a range of 0 to 100 and the progress bar is 500 pixels wide, the bar would jump 5 pixels forward for every value change. This will look ugly. To get a smooth progression where the bar expands by 1 pixel at a time, a range of 0 to 99.999.999 would surely work, but it would be highly inefficient. This is because the current value of the bar would change a lot without any graphical depiction. So, the best value for the range would be 0 to the actual bar's width in pixels. Unfortunately, the width of the bar can change depending on the actual widget width, and frequently querying the actual size of the bar every time the value changes is also not a good solution. Why 2048, then? It's just a nice round number that is bigger than any screen resolution we're likely to get.
This ensures that the progress bar runs smoothly, even if it is fully expanded. If you are targeting smaller devices, choose a smaller, more appropriate number.

To be able to calculate the time remaining for the download to finish, you have to start a timer. In this case, use `QElapsedTimer`. After posting the request with `QNetworkAccessManager::get()`, start the timer by calling `QElapsedTimer::start()`. Assuming that the timer is called `m_timer`, the calculation will be as follows:

```
qreal remaining = m_timer.elapsed() *
                  (1.0 - progress) / progress;
int remainingSeconds = qRound(remaining / 1000);
```

`QElapsedTimer::elapsed()` returns the milliseconds that are counted from the moment the timer is started. Assuming that the download progress is linear, the ratio of the remaining time to the elapsed time equals `(1.0 - progress) / progress`. For example, if `progress` is 0.25 (25%), the expected remaining time will be three times bigger than the elapsed time: `(1.0 - 0.25) / 0.25) = 3`. If you divide the result by 1,000 and round it to the nearest integer, you'll get the remaining time in seconds.

`QElapsedTimer` is not to be confused with `QTimer`. `QTimer` is used to call a slot after a certain amount of time has passed. `QElapsedTimer` is merely a convenience class that is able to remember the start time and calculate the elapsed time by subtracting the start time from the current time.

Using a proxy

If you want to use a proxy, you first have to set up `QNetworkProxy`. You can define the type of proxy with `setType()`. As arguments, you will most likely want to pass `QNetworkProxy::Socks5Proxy` or `QNetworkProxy::HttpProxy`. Then, set up the hostname with `setHostName()`, the username with `setUserName()`, and the password with `setPassword()`. The last two properties are, of course, only needed if the proxy requires authentication. Once the proxy is set up, you can set it to the access manager via `QNetworkAccessManager::setProxy()`. Now, all new requests will use this proxy.

Connecting to Google, Facebook, Twitter, and co.

Since we discussed `QNetworkAccessManager`, you now have the knowledge you need to integrate Facebook, Twitter, or similar sites into your application. They all use the HTTPS protocol and simple requests in order to retrieve data from them. For Facebook, you have to use the so-called Graph API. It describes which interfaces are available and what options they offer. If you want to search for users who are called **Helena**, you have to request `https://graph.facebook.com/search?q=helena&type=user`. Of course, you can do this with `QNetworkManager`. You will find more information about the possible requests to Facebook at `https://developers.facebook.com/docs/graph-api`.

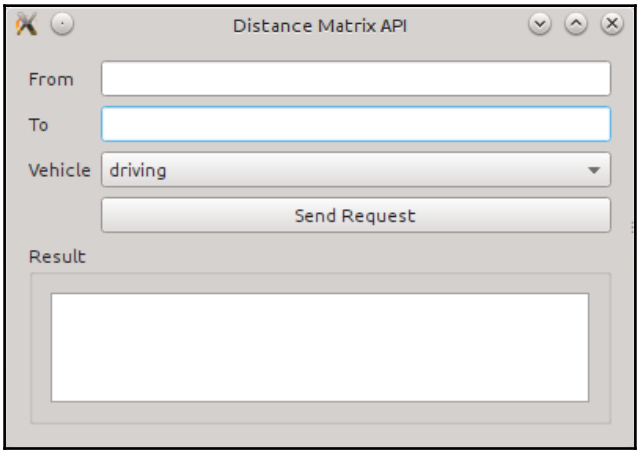
If you wish to display tweets in your game, you have to use Twitter's REST or Search API. Assuming that you know the ID of a tweet you would like to display, you can get it through `https://api.twitter.com/1.1/statuses/show.json?id=12345`, where `12345` is the actual ID for the tweet. If you would like to find tweets mentioning `#Helena`, you would write `https://api.twitter.com/1.1/search/tweets.json?q=%23Helena`. You can find more information about the parameters and the other possibilities of Twitter's API at `https://developer.twitter.com/en/docs`.

Since both Facebook and Twitter need an authentication to use their APIs, we will take a look at Google instead. Let's use Google's Distance Matrix API in order to get information about how long it would take for us to get from one city to another. The technical documentation for the API we will use can be found at `https://developers.google.com/maps/documentation/distancematrix`.

Time for action – Using Google's Distance Matrix API

The GUI for this example is kept simple—the source code is attached with the book. It consists of two line edits (`ui->from` and `ui->to`) that allow you to enter the origin and destination of the journey. It also provides you with a combobox (`ui->vehicle`) that allows you to choose a mode of transportation—whether you want to drive a car, ride a bicycle, or walk—a push button (`ui->search`) to start the request, and a text edit, or (`ui->result`) to show the results.

It looks like this:



`MainWindow`—a subclass of `QMainWindow`—is the application's main class that holds two private members: `m_network_manager`, which is a pointer to `QNetworkAccessManager`, and `m_reply`, which is a pointer to `QNetworkReply`.

Time for action – Constructing the query

Whenever the button is pressed, the `sendRequest()` slot is called:

```
void MainWindow::sendRequest()
{
    if (m_reply != nullptr && m_reply->isRunning()) {
        m_reply->abort();
    }
    ui->result->clear();
    //...
}
```

In this slot, we first check whether there is an old request, which was stored in `m_reply`, and whether it is still running. If that is `true`, we abort the old request, as we are about to schedule a new one. Then, we also wipe out the result of the last request by calling `QPlainTextEdit::clear()` on the text edit.

Next, we will construct the URL for the request. We can do this by composing the string by hand where we add the query parameters to the base URL similar to the following:

```
// don't do this!
QString url = baseUrl + "?origin=" + ui->from->text() + "&...";
```

Besides the problem that this quickly becomes hard to read when we include multiple parameters, it is also rather error-prone. The values of the line edits have to be encoded to fit the criteria for a valid URL. For every user value, we, therefore, have to call `QUrl::toPercentEncoding()` explicitly. A much better approach, which is easier to read and less error-prone, is to use `QUrlQuery`. It circumvents the problem that may result when you forget to encode the data. So, we do this:

```
QUrlQuery query;
query.addQueryItem(QStringLiteral("sensor"),   QStringLiteral("false"));
query.addQueryItem(QStringLiteral("language"), QStringLiteral("en"));
query.addQueryItem(QStringLiteral("units"),    QStringLiteral("metric"));
query.addQueryItem(QStringLiteral("mode"),     ui->vehicle->currentText());
query.addQueryItem(QStringLiteral("origins"),  ui->from->text());
query.addQueryItem(QStringLiteral("destinations"), ui->to->text());
```

The usage is pretty clear: we create an instance and then add the query parameters with `addQueryItem()`. The first argument is taken as the key and the second as the value resulting in a string such as "key=value". The value will be automatically encoded when we use `QUrlQuery` in conjunction with `QUrl`. Other benefits of using `QUrlQuery` are that we can check whether we have already set a key with `hasQueryItem()`, taking the key as an argument, or removed a previously set key by calling `removeQueryItem()`.

Let's review which parameters we have set. The `sensor` key is set to `false` as we are not using a GPS device to locate our position. The `language` key is set to `English`, and for units, we favor metric over imperial. Then, the search-related parameters are set. The `origins` key holds the places we want to start from. As its value, the text of the `ui->from` line edit is chosen. If you want to query multiple starting positions, you just have to combine them using `|`. Equivalent to the origins, we set up the value for destinations. Last, we pass the value of the combo box to mode, which defines whether we want to go by a car, bicycle, or whether we want to walk. Next, we execute the request:

```
QUrl url(QStringLiteral(
     "https://maps.googleapis.com/maps/api/distancematrix/json"));
url.setQuery(query);
m_reply = m_network_manager->get(QNetworkRequest(url));
```

We create `QUrl` that contains the address to which the query should be posted. By including `json` at the end, we define that the server should transfer its reply using the JSON format. Google also provides the option for us to get the result as XML. To achieve this, simply replace `json` with `xml`. However, since the APIs of Facebook and Twitter return JSON, we will use this format.

Then, we set the previously constructed `query` to the URL by calling `QUrl::setQuery()`. This automatically encodes the values, so we do not have to worry about that. Last, we post the request by calling the `get()` function and store the returned `QNetworkReply` in `m_reply`.

Time for action – Parsing the server's reply

In the constructor, we have connected the manager's `finished()` signal to the `finished()` slot of the `MainWindow` class. It will thus be called after the request has been posted:

```
void MainWindow::finished(QNetworkReply *reply)
{
    if (m_reply != reply) {
        reply->deleteLater();
        return;
    }
    //...
}
```

First, we check whether the reply that was passed is the one that we have requested through `m_network_manager`. If this is not the case, we delete the `reply` and exit the function. This can happen if a reply was aborted by the `sendRequest()` slot. Since we are now sure that it is our request, we set `m_reply` to `nullptr`, because we have handled it and do not need this information any more:

```
m_reply = nullptr;
if (reply->error() != QNetworkReply::NoError) {
    ui->result->setPlainText(reply->errorString());
    reply->deleteLater();
    return;
}
```

Next, we check whether an error occurred, and if it did, we put the reply's error string in the text edit, delete the reply, and exit the function. After this, we can finally start decoding the server's response:

```
const QByteArray content = reply->readAll();
const QJsonDocument doc = QJsonDocument::fromJson(content);
if (!doc.isObject()) {
    ui->result->setPlainText(tr("Error while reading the JSON file."));
    reply->deleteLater();
    return;
}
```

With `readAll()`, we get the content of the server's reply. Since the transferred data is not large, we do not need to use partial reading with `readyRead()`. The content is then converted to `QJsonDocument` using the `QJsonDocument::fromJson()` static function, which takes `QByteArray` as an argument and parses its data. If the document is not an object, the server's reply wasn't valid, as the API call should respond with a single object. In this case, we show an error message on the text edit, delete the reply, and exit the function. Let's look at the next part of the code:

```
const QJsonObject obj = doc.object();
const QJsonArray origins = obj.value("origin_addresses").toArray();
const QJsonArray destinations =
obj.value("destination_addresses").toArray();
```

Since we have now ensured that there is an object, we store it in `obj`. Furthermore, due to the API, we also know that the object holds the `origin_addresses` and `destination_addresses` keys. Both values are arrays that hold the requested origins and destinations. From this point on, we will skip any tests if the values exist and are valid since we trust the API. The object also holds a key called `status`, whose value can be used to check whether the query may have failed and if yes, why. The last two lines of the source code store the origins and destinations in two variables. With `obj.value("origin_addresses")`, we get `QJsonValue` that holds the value of the pair specified by the `origin_addresses` key, and `QJsonValue::toArray()` converts this value to `QJsonArray`. The returned JSON file for a search requesting the distance from Warsaw or Erlangen to Birmingham will look like this:

```
{
    "destination_addresses" : [ "Birmingham, West Midlands, UK" ],
    "origin_addresses" : [ "Warsaw, Poland", "Erlangen, Germany" ],
    "rows" : [ ... ],
    "status" : "OK"
}
```

The `rows` key holds the actual results as an array. The first object in this array belongs to the first origin, the second object to the second origin, and so on. Each object holds a key named `elements`, whose value is also an array of objects that belong to the corresponding destinations:

```
"rows" : [
    {
        "elements" : [{...}, {...}]
    },
    {
        "elements" : [{...}, {...}]
    }
  ],
```

Each JSON object (`{...}` in the preceding example) for an origin-destination pair consists of two pairs with the distance and duration keys. Both values of these keys are arrays that hold the `text` and `value` keys, where `text` is a human-readable phrase for `value`. The object for the Warsaw-Birmingham search looks as shown in the following snippet:

```
{
    "distance" : {
        "text" : "1,835 km",
        "value" : 1834751
    },
    "duration" : {
        "text" : "16 hours 37 mins",
        "value" : 59848
    },
    "status" : "OK"
}
```

As you can see, the value of `value` for distance is the distance expressed in meters—since we have used `units=metric` in the request—and the value of `text` is value transformed into kilometers with the "km" postfix. The same applies to duration. Here, value is expressed in seconds, and text is value converted into hours and minutes.

Now that we know how the returned JSON is structured, we display the value of each origin-destination pair in the text edit. Therefore, we loop through each possible pairing using the two `QJsonArray`. We need the indices as well as values, so we use the classic `for` loop instead of the range-based one:

```
QString output;
for (int i = 0; i < origins.count(); ++i) {
    const QString origin = origins.at(i).toString();
    const QJsonArray row = obj.value("rows").toArray().at(i).toObject()
```

```
            .value("elements").toArray();
    for (int j = 0; j < destinations.count(); ++j) {
```

First, we create an `output` string variable to cache the constructed text. Before starting the second loop, we calculate two variables that will be the same for all destinations. The `origin` variable holds the text representation of the current origin, and the `row` variable contains the corresponding row of the table. Whenever we try to get an item out of a `QJsonArray` or a `QJsonObject`, the returned value will have the `QJsonValue` type, so each time we do that, we need to convert it to an array, an object, or a string, depending on what we expect to get according to the API. When we calculate the `row` variable, starting at the reply's root object, we fetch the value of the `rows` key and convert it to an array (`obj.value("rows").toArray()`). Then, we fetch the value of the current row (`.at(i)`), convert it to an object, and fetch its `elements` key (`.toObject().value("elements")`). Since this value is also an array—the columns of the row—we convert it to an array.

The scope inside the two loops will be reached for each combination. Think of the transferred result as a table where the origins are rows and the destinations are columns:

```
output += tr("From: %1\n").arg(origin);
output += tr("To: %1\n").arg(destinations.at(j).toString());
```

First, we add the `"From:"` string and the current origin to output. The same is done for the destination, which results in the following as the value for output:

```
From: Warsaw, Poland
To: Birmingham, West Midlands, UK
```

Next, we will read the duration and distance from the corresponding `QJsonObject` from where we call `data`:

```
const QJsonObject data = row.at(j).toObject();
const QString status = data.value("status").toString();
```

In this code, we fetch the current column from the row (`at(j)`) and convert it to an object. This is the object that contains the distance and duration for an origin-destination pair in the `(i; j)` cell. Besides `distance` and `duration`, the object also holds a key called `status`. Its value indicates whether the search was successful (`OK`), whether the origin or destination could not be found (`NOT_FOUND`), or whether the search could not find a route between the origin and destination (`ZERO_RESULTS`). We store the value of `status` in a local variable that is also named `status`.

Next, we check the status and append the distance and the duration to the output:

```
if (status == "OK") {
    output += tr("Distance: %1\n").arg(
        data.value("distance").toObject().value("text").toString());
    output += tr("Duration: %1\n").arg(
        data.value("duration").toObject().value("text").toString());
} else { /*...*/ }
```

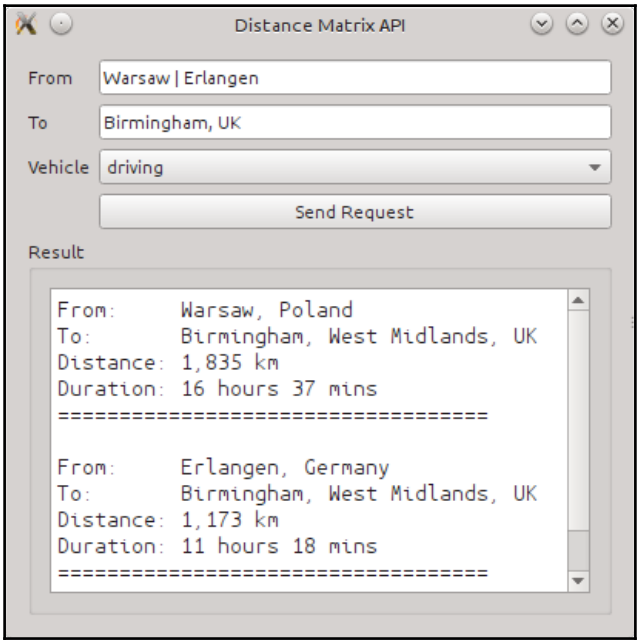
For distance, we want to show the phrased result. Therefore, we first get the JSON value of the distance key (`data.value("distance")`), convert it to an object, and request the value for the text key (`toObject().value("text")`). Lastly, we convert `QJsonValue` to `QString` using `toString()`. The same applies for duration. Finally, we need to handle the errors the API might return:

```
} else if (status == "NOT_FOUND") {
    output += tr("Origin and/or destination of this "
                 "pairing could not be geocoded.\n");
} else if (status == "ZERO_RESULTS") {
    output += tr("No route could be found.\n");
} else {
    output += tr("Unknown error.\n");
}
output += QStringLiteral("=").repeated(35) + QStringLiteral("\n");
```

At the end of the output for each cell, we add a line consisting of `35` equals signs (`QStringLiteral("=").repeated(35)`) to separate the result from the other cells. Finally, after all loops finish, we put the text into the text edit and delete the reply object:

```
ui->result->setPlainText(output);
reply->deleteLater();
```

The actual result then looks as follows:



Have a go hero – Choosing XML as the reply's format

To hone your XML skills, you can use `https://maps.googleapis.com/maps/api/distancematrix/xml` as the URL to which you send the requests. Then, you can parse the XML file, as we did with JSON, and display the retrieved data likewise.

Controlling the connectivity state

Before trying to access a network resource, it's useful to check whether you have an active connection to the internet. Qt allows you to check whether the computer, mobile device, or tablet is online. You can even start a new connection if the operating system supports it.

The relevant API mainly consists of four classes. `QNetworkConfigurationManager` is the base and starting point. It holds all network configurations available on the system. Furthermore, it provides information about the network capabilities, for example, whether you can start and stop interfaces. The network configurations found by it are stored as `QNetworkConfiguration` classes.

`QNetworkConfiguration` holds all information about an access point but not about a network interface, as an interface can provide multiple access points. This class also provides only the information about network configurations. You can't configure an access point or a network interface through `QNetworkConfiguration`. The network configuration is up to the operating system and therefore `QNetworkConfiguration` is a read-only class. With `QNetworkConfiguration`, however, you can determine whether the type of connection is an Ethernet, WLAN, or 4G connection. This may influence what kind of data and, more importantly, what size of data you will download.

With `QNetworkSession`, you can then start or stop system network interfaces, which are defined by the configurations. This way, you gain control over an access point. `QNetworkSession` also provides session management that is useful when a system's access point is used by more than one application. The session ensures that the underlying interface only gets terminated after the last session has been closed. Lastly, `QNetworkInterface` provides classic information, such as the hardware address or interface name.

QNetworkConfigurationManager

`QNetworkConfigurationManager` manages all network configurations that are available on a system. You can access these configurations by calling `allConfigurations()`. Of course, you have to create an instance of the manager first:

```
QNetworkConfigurationManager manager;
QList<QNetworkConfiguration> cfgs = manager.allConfigurations();
```

The configurations are returned as a list. The default behavior of `allConfigurations()` is to return all possible configurations. However, you can also retrieve a filtered list. If you pass `QNetworkConfiguration::Active` as an argument, the list only contains configurations that have at least one active session. If you create a new session based on such a configuration, it will be active and connected. By passing `QNetworkConfiguration::Discovered` as an argument, you will get a list with configurations that can be used to immediately start a session. Note, however, that at this point, you cannot be sure whether the underlying interface can be started. The last important argument is `QNetworkConfiguration::Defined`. With this argument, `allConfigurations()` returns a list of configurations that are known to the system but are not usable right now. This may be a previously used WLAN hotspot, which is currently out of range.

You will be notified whenever the configurations change. If a new configuration becomes available, the manager emits the `configurationAdded()` signal. This may happen, for example, if mobile data transmission becomes available or if the user turns his/her device's WLAN adapter on. If a configuration is removed, for example, if the WLAN adapter is turned off, `configurationRemoved()` is emitted. Lastly, when a configuration is changed, you will be notified by the `configurationChanged()` signal. All three signals pass a constant reference to the configuration about what was added, removed, or changed. The configuration passed by the `configurationRemoved()` signal is, of course, invalid. It still contains the name and identifier of the removed configuration.

To find out whether any network interface of the system is active, call `isOnline()`. If you want to be notified about a mode change, track the `onlineStateChanged()` signal.

Since a WLAN scan takes a certain amount of time, `allConfigurations()` may not return all the available configurations. To ensure that configurations are completely populated, call `updateConfigurations()` first. Due to the long time it may take to gather all the information about the system's network configurations, this call is asynchronous. Wait for the `updateCompleted()` signal and only then, call `allConfigurations()`.

`QNetworkConfigurationManager` also informs you about the Bearer API's capabilities. The `capabilities()` function returns a flag of the `QNetworkConfigurationManager::Capabilities` type and describes the available possibilities that are platform-specific. The values you may be most interested in are as follows:

Value	**Meaning**
`CanStartAndStopInterfaces`	This means that you can start and stop access points.
`ApplicationLevelRoaming`	This indicates that the system will inform you if a more suitable access point is available, and that you can actively change the access point if you think there is a better one available.
`DataStatistics`	With this capability, `QNetworkSession` contains information about the transmitted and received data.

QNetworkConfiguration

`QNetworkConfiguration` holds, as mentioned earlier, information about an access point. With `name()`, you get the user-visible name for a configuration, and with `identifier()`, you get a unique, system-specific identifier. If you develop games for mobile devices, it may be of advantage to you to know which type of connection is being used. This might influence the data that you request; for example, the quality and thus, the size of a video. With `bearerType()`, the type of bearer used by a configuration is returned. The returned enumeration values are rather self-explanatory: `BearerEthernet`, `BearerWLAN`, `Bearer2G`, `BearerCDMA2000`, `BearerWCDMA`, `BearerHSPA`, `BearerBluetooth`, `BearerWiMAX`, and so on. You can look up the full-value list in the documentation for `QNetworkConfiguration::BearerType`.

With `purpose()`, you get the purpose of the configuration, for example, whether it is suitable to access a private network (`QNetworkConfiguration::PrivatePurpose`) or to access a public network (`QNetworkConfiguration::PublicPurpose`). The state of the configuration, if it is defined, discovered or active, as previously described, can be accessed through `state()`.

QNetworkSession

To start a network interface or to tell the system to keep an interface connected for as long as you need it, you have to start a session:

```
QNetworkConfigurationManager manager;
QNetworkConfiguration cfg = manager.defaultConfiguration();
QNetworkSession *session = new QNetworkSession(cfg, this);
session->open();
```

A session is based on a configuration. When there is more than one session and you are not sure which one to use, use `QNetworkConfigurationManager::defaultConfiguration()`. It returns the system's default configuration. Based on this, you can create an instance of `QNetworkSession`. The first argument, the configuration, is required. The second is optional but is recommended since it sets a parent, and we do not have to take care of the deletion. You may want to check whether the configuration is valid (`QNetworkConfiguration::isValid()`) first.

Calling `open()` will start the session and connect the interface if needed and supported. Since `open()` can take some time, the call is asynchronous. So, either listen to the `opened()` signal, which is emitted as soon as the session is open, or to the `error()` signal if an error happened. The error information is represented using the `QNetworkSession::SessionError` type. Alternatively, instead of checking the `opened()` signal, you can also watch the `stateChanged()` signal. The possible states for a session can be `Invalid`, `NotAvailable`, `Connecting`, `Connected`, `Closing`, `Disconnected`, and `Roaming`.

If you want to open the session in a synchronous way, call `waitForOpened()` right after calling `open()`. It will block the `event` loop until the session is open. This function will return `true` if successful and `false` otherwise. To limit the waiting time, you can define a time-out. Just pass the milliseconds that you are willing to wait as an argument to `waitForOpened()`. To check whether a session is open, use `isOpen()`.

To close the session, call `close()`. If no session is left on the interface, it will be shot down. To force an interface to disconnect, call `stop()`. This call will invalidate all the sessions that are based on that interface.

You may receive the `preferredConfigurationChanged()` signal, which indicates that the preferred configuration, that is, for example, the preferred access point, has changed. This may be the case if a WLAN network is now in range and you do not have to use 2G anymore. The new configuration is passed as the first argument, and the second one indicates whether changing the new access point will also alter the IP address. Besides checking for the signal, you can also inquire whether roaming is available for a configuration by calling `QNetworkConfiguration::isRoamingAvailable()`. If roaming is available, you have to decide to either reject the offer by calling `ignore()` or to accept it by calling `migrate()`. If you accept roaming, it will emit `newConfigurationActivated()` when the session is roamed. After you have checked the new connection, you can either accept the new access point or reject it. The latter means that you will return to the previous access point. If you accept the new access point, the previous one will be terminated.

QNetworkInterface

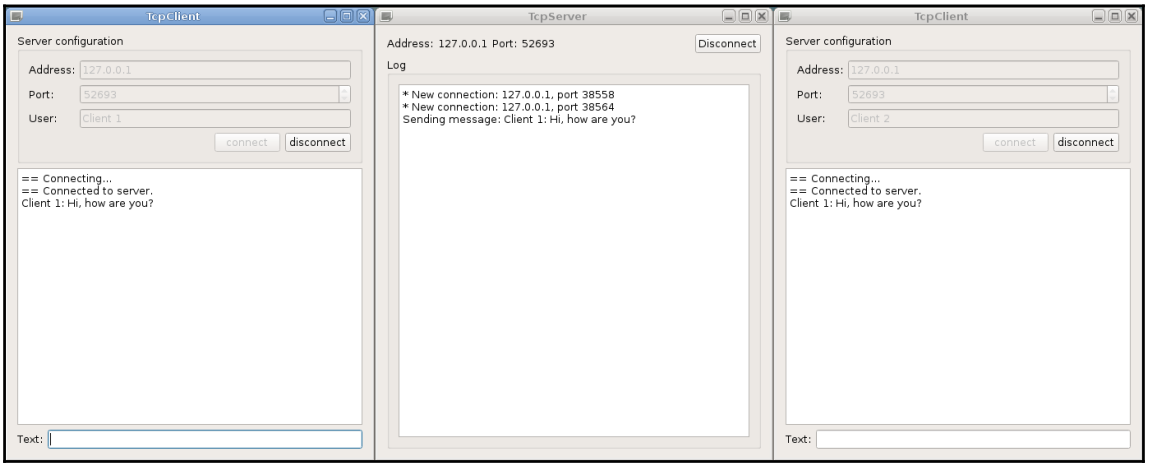
To get the interface that is used by a session, call `QNetworkSession::interface()`. It will return the `QNetworkInterface` object, which describes the interface. With `hardwareAddress()`, you get the low-level hardware address of the interface that is normally the MAC address. The name of the interface can be obtained by `name()`, which is a string such as "eth0" or "wlan0". A list of IP addresses as well as their netmasks and broadcast addresses registered with the interface is returned by `addressEntries()`. Furthermore, information about whether the interface is a loopback or whether it supports multicasting can be queried with `flags()`. The returned bitmask is a combination of these values: `IsUp`, `IsRunning`, `CanBroadcast`, `IsLoopBack`, `IsPointToPoint`, and `CanMulticast`.

Communicating between games

After having discussed Qt's high-level network classes such as `QNetworkAccessManager` and `QNetworkConfigurationManager`, we will now take a look at lower-level network classes and see how Qt supports you when it comes to implementing TCP or UDP servers and clients. This becomes relevant when you plan to extend your game by including a multiplayer mode. For such a task, Qt offers `QTcpSocket`, `QUdpSocket`, and `QTcpServer`.

Time for action – Realizing a simple chat program

To get familiar with `QTcpServer` and `QTcpSocket`, let's develop a simple chat program. This example will teach you the basic knowledge of network handling in Qt so that you can use this skill later to connect two or more copies of a game. At the end of this exercise, we want to see something like this:



On both the left-hand side and the right-hand side of the preceding screenshot, you can see a client, whereas the server is in the middle. We'll start by taking a closer look at the server.

The server – QTcpServer

As a protocol for communication, we will use **Transmission Control Protocol** (**TCP**). You may know this network protocol from the two most popular internet protocols: HTTP and FTP. Both use TCP for their communication and so do the globally used protocols for email traffic: SMTP, POP3, and IMAP. The main advantage of TCP is its reliability and connection-based architecture. Data transferred by TCP is guaranteed to be complete, ordered, and without any duplicates. The protocol is furthermore stream oriented, which allows us to use `QDataStream` or `QTextStream`. A downside to TCP is its speed. This is because the missing data has to be retransmitted until the receiver fully receives it. By default, this causes a retransmission of all the data that was transmitted after the missing part. So, you should only choose TCP as a protocol if speed is not your top priority, but rather the completeness and correctness of the transmitted data. This applies if you send unique and nonrepetitive data.

Time for action – Setting up the server

A look at the server's GUI shows us that it principally consists of `QPlainTextEdit` (`ui->log`) that is used to display system messages and a button (`ui->disconnectClients`), which allows us to disconnect all the currently connected clients. On the top, next to the button, the server's address and port are displayed (`ui->address` and `ui->port`). After setting up the user interface in the constructor of the server's class `TcpServer`, we initiate the internally used `QTcpServer`, which is stored in the `m_server` private member variable:

```
if (!m_server->listen(QHostAddress::LocalHost, 52693)) {
    ui->log->setPlainText(tr("Failure while starting server: %1")
                          .arg(m_server->errorString()));
    return;
}
connect(m_server, &QTcpServer::newConnection,
        this, &TcpServer::newConnection);
```

What just happened?

With `QTcpServer::listen()`, we defined that the server should listen to the localhost and the port `52693` for new incoming connections. The value used here, `QHostAddress::LocalHost` of the `QHostAddress::SpecialAddress` enumeration, will resolve to `127.0.0.1`. Instead, if you pass `QHostAddress::Any`, the server will listen to all IPv4 interfaces as well as to IPv6 interfaces. If you only want to listen to a specific address, just pass this address as `QHostAddress`:

```
m_server->listen(QHostAddress("127.0.0.1"), 0);
```

This will behave like the one in the preceding code, only in that the server will now listen to a port that will be chosen automatically. On success, `listen()` will return `true`. So, if something goes wrong in the example, it will show an error message on the text edit and exit the function. To compose the error message, we are using `QTcpServer::errorString()`, which holds a human-readable error phrase.

To handle the error in your game's code, the error string is not suitable. In any case where you need to know the exact error, use `QTcpServer::serverError()`, which returns the enumeration value of `QAbstractSocket::SocketError`. Based on this, you know exactly what went wrong, for example, `QAbstractSocket::HostNotFoundError`. If `listen()` was successful, we connect the server's `newConnection()` signal to the class's `newConnection()` slot. The signal will be emitted every time a new connection is available. Lastly, we show the server's address and port number that can be accessed through `serverAddress()` and `serverPort()`:

```
ui->address->setText(m_server->serverAddress().toString());
ui->port->setText(QString::number(m_server->serverPort()));
```

This information is required by the clients so that they are able to connect to the server.

Time for action – Reacting on a new pending connection

As soon as a client tries to connect to the server, the `newConnection()` slot will be called:

```
void TcpServer::newConnection()
{
    while (m_server->hasPendingConnections()) {
        QTcpSocket *socket = m_server->nextPendingConnection();
        m_clients << socket;
        ui->disconnectClients->setEnabled(true);
        connect(socket, &QTcpSocket::disconnected,
                this, &TcpServer::removeConnection);
        connect(socket, &QTcpSocket::readyRead,
                this, &TcpServer::readyRead);
        ui->log->appendPlainText(tr("* New connection: %1, port %2\n")
                                 .arg(socket->peerAddress().toString())
                                 .arg(socket->peerPort()));
    }
}
```

What just happened?

Since more than one connection may be pending, we use `hasPendingConnections()` to determine whether there is at least one more pending connection. Each one is then handled in the iteration of the `while` loop. To get a pending connection of the `QTcpSocket` type, we call `nextPendingConnection()` and add this connection to a private vector called `m_clients`, which holds all active connections. In the next line, as there is now at least one connection, we enable the button that allows all connections to be closed. The slot connected to the button's `click()` signal will call `QTcpSocket::close()` on each single connection. When a connection is closed, its socket emits a `disconnected()` signal. We connect this signal to our `removeConnection()` slot. With the last connection, we react to the socket's `readyRead()` signal, which indicates that new data is available. In such a situation, our `readyRead()` slot is called. Lastly, we print a system message that a new connection has been established. The address and port of the connecting client and peer can be retrieved by the socket's `peerAddress()` and `peerPort()` functions.

If a new connection can't be accepted, the `acceptError()` signal is emitted instead of `newConnection()`. It passes the reason for the failure of the `QAbstractSocket::SocketError` type as an argument. If you want to temporarily decline new connections, call `pauseAccepting()` on `QTcpServer`. To resume accepting new connections, call `resumeAccepting()`.

Time for action – Forwarding a new message

When a connected client sends a new chat message, the underlying socket—since it inherits `QIODevice`—emits `readyRead()`, and thus, our `readyRead()` slot will be called.

Before we take a look at this slot, there is something important that you need to keep in mind. Even though TCP is ordered and without any duplicates, this does not mean that all the data is delivered in one big chunk. So, before processing the received data, we need to ensure that we get the entire message. Unfortunately, there is neither an easy way to detect whether all data was transmitted nor a globally usable method for such a task. Therefore, it is up to you to solve this problem, as it depends on the use case. Two common solutions, however, are to either send magic tokens to indicate the start and end of a message, for example, single characters or XML tags, or you can send the size of the message upfront.

The second solution is shown in the Qt documentation where the length is put in a `quint16` in front of the message. We, on the other hand, will look at an approach that uses a simple magic token to handle the messages correctly. As a delimiter, we use the "End of Transmission Block" character—ASCII code 23—to indicate the end of a message. We also choose UTF-8 as the encoding of transmitted messages to ensure that clients with different locales can communicate with each other.

Since the processing of received data is quite complex, we will go through the code step by step this time:

```
void TcpServer::readyRead()
{
    QTcpSocket *socket = qobject_cast<QTcpSocket*>(sender());
    if (!socket) {
        return;
    }
    //...
}
```

To determine which socket called the slot, we use `sender()`. If the cast to `QTcpSocket` is unsuccessful, we exit the slot.

Note that `sender()` is used for simplicity. If you write real-life code, it is better to use `QSignalMapper`.

Next, we read the transferred—potentially fragmentary—message with `readAll()`:

```
QByteArray &buffer = m_receivedData[socket];
buffer.append(socket->readAll());
```

Here, `QHash<QTcpSocket*, QByteArray> m_receivedData` is a private class member where we store the previously received data for each connection. When the first chunk of data is received from a client, `m_receivedData[socket]` will automatically insert an empty `QByteArray` into the hash and return a reference to it. On subsequent calls, it will return a reference to the same array. We use `append()` to append the newly received data to the end of the array. Finally, we need to identify the messages that were completely received by now, if there are any such messages:

```
while(true) {
    int endIndex = buffer.indexOf(23);
    if (endIndex < 0) {
        break;
    }
```

```
        QString message = QString::fromUtf8(buffer.left(endIndex));
        buffer.remove(0, endIndex + 1);
        newMessage(socket, message);
    }
```

On each iteration of the loop, we try to find the first separator character. If we didn't find one (`endIndex < 0`), we exit the loop, and leave the remaining partial message in `m_receivedData`. If we found a separator, we take the first message's data using the `left(endIndex)` function that returns the leftmost `endIndex` bytes from the array. To remove the first message from `buffer`, we use the `remove()` function that will remove the specified number of bytes, shifting the remaining bytes to the left. We want to remove `endIndex + 1` bytes (the message itself and the separator after it). Following our transmission protocol, we interpret the data as UTF-8 and call our `newMessage()` function that will handle the received message.

In the `newMessage()` function, we append the new message to the server log and send it to all clients:

```
void TcpServer::newMessage(QTcpSocket *sender, const QString &message)
{
    ui->log->appendPlainText(tr("Sending message: %1\n").arg(message));
    QByteArray messageArray = message.toUtf8();
    messageArray.append(23);
    for(QTcpSocket *socket: m_clients) {
        if (socket->state() == QAbstractSocket::ConnectedState) {
            socket->write(messageArray);
        }
    }
    Q_UNUSED(sender)
}
```

In this function, we encode the message according to our transmission protocol. First, we use `toUtf8()` to convert `QString` to `QByteArray` in UTF-8 encoding. Next, we append the separator character. Finally, we iterate over the list of clients, check whether they are still connected, and send the encoded message to them. Since the socket inherits `QIODevice`, you can use most of the functions that you know from `QFile`. The current behavior of our server is very simple, so we have no use for the `sender` argument, so we add the `Q_UNUSED` macro to suppress the unused argument warning.

Have a go hero – Using QSignalMapper

As discussed earlier, using `sender()` is a convenient, but not an object-oriented, approach. Thus, try to use `QSignalMapper` instead to determine which socket called the slot. To achieve this, you have to connect the socket's `readyRead()` signal to a mapper and the slot directly. All the signal-mapper-related code will go into the `newConnection()` slot.

The same applies to the connection to the `removeConnection()` slot. Let's take a look at it next.

Time for action – Detecting a disconnect

When a client terminates the connection, we have to delete the socket from the local `m_clients` list. The socket's `disconnected()` signal is already connected to the `removeConnection()` slot, so we just need to implement it as follows:

```
void TcpServer::removeConnection()
{
    QTcpSocket *socket = qobject_cast<QTcpSocket*>(sender());
    if (!socket) {
        return;
    }
    ui->log->appendPlainText(tr("* Connection removed: %1, port %2\n")
                             .arg(socket->peerAddress().toString())
                             .arg(socket->peerPort()));
    m_clients.removeOne(socket);
    m_receivedData.remove(socket);
    socket->deleteLater();
    ui->disconnectClients->setEnabled(!m_clients.isEmpty());
}
```

What just happened?

After getting the socket that emitted the call through `sender()`, we post the information that a socket is being removed. Then, we remove the socket from `m_clients`, remove the associated buffer from `m_receivedData` and call `deleteLater()` on it. Do not use `delete`. Lastly, if no client is left, the disconnect button is disabled.

The server is ready. Now let's take a look at the client.

The client

The GUI of the client (`TcpClient`) is pretty simple. It has three input fields to define the server's address (`ui->address`), the server's port (`ui->port`), and a username (`ui->user`). Of course, there is also a button to connect to (`ui->connect`) and disconnect from (`ui->disconnect`) the server. Finally, the GUI has a text edit that holds the received messages (`ui->chat`) and a line edit (`ui->text`) to send messages.

Time for action – Setting up the client

After providing the server's address and port and choosing a username, the user can connect to the server:

```
void TcpClient::on_connect_clicked()
{
    //...
    if (m_socket->state() != QAbstractSocket::ConnectedState) {
        ui->chat->appendPlainText(tr("== Connecting..."));
        m_socket->connectToHost(ui->address->text(), ui->port->value());
        //...
    }
}
```

What just happened?

The `m_socket` private member variable holds an instance of `QTcpSocket`. If this socket is already connected, nothing happens. Otherwise, the socket is connected to the given address and port by calling `connectToHost()`. Besides the obligatory server address and port number, you can pass a third argument to define the mode in which the socket will be opened. For possible values, you can use `OpenMode` just like we did for `QIODevice`.

Since this call is asynchronous, we print a notification to the chat so that the user is informed that the application is currently trying to connect to the server. When the connection is established, the socket sends the `connected()` signal that prints "Connected to server" on the chat to indicate that we have connected to a slot. Besides the messages in the chat, we also updated the GUI by, for example, disabling the connect button, but this is all basic stuff. You won't have any trouble understanding this if you have had a look at the sources. So, these details are left out here.

Of course, something could go wrong when trying to connect to a server, but luckily, we are informed about a failure as well through the `error()` signal, passing a description of error in the form of `QAbstractSocket::SocketError`. The most frequent errors will probably be `QAbstractSocket::ConnectionRefusedError` if the peer refused the connection or `QAbstractSocket::HostNotFoundError` if the host address could not be found. If the connection, however, was successfully established, it should be closed later on. You can either call `abort()` to immediately close the socket, whereas `disconnectFromHost()` will wait until all pending data has been written.

Time for action – Receiving text messages

In the constructor, we have connected the socket's `readyRead()` signal to a local slot. So, whenever the server sends a message through `QTcpSocket::write()`, we read the data and decode it:

```
m_receivedData.append(m_socket->readAll());
while(true) {
    int endIndex = m_receivedData.indexOf(23);
    if (endIndex < 0) {
        break;
    }
    QString message = QString::fromUtf8(m_receivedData.left(endIndex));
    m_receivedData.remove(0, endIndex + 1);
    newMessage(message);
}
```

This code is very similar to the `readyRead()` slot of the server. It's even simpler because we only have one socket and one data buffer, so `m_receivedData` is a single `QByteArray`. The `newMessage()` implementation in the client is also much simpler than in the server:

```
void TcpClient::newMessage(const QString &message)
{
    ui->chat->appendPlainText(message);
}
```

Here, we just need to display the received message to the user.

Time for action – Sending text messages

What is left now is to describe how to send a chat message. On hitting return button inside the line edit, a local slot will be called that checks whether there is actual text to send and whether `m_socket` is still connected. If everything is ready, we construct a message that contains the self-given username, a colon, and the text of the line edit:

```
QString message = QStringLiteral("%1: %2")
                  .arg(m_user).arg(ui->text->text());
```

Then, we encode and send the message, just like we did on the server side:

```
QByteArray messageArray = message.toUtf8();
messageArray.append(23);
m_socket->write(messageArray);
```

That's all. It's like writing and reading from a file. For the complete example, take a look at the sources bundled with this book and run the server and several clients.

You can see that the server and client share a significant amount of code. In a real project, you definitely want to avoid such duplication. You can move all repeating code to a common library used by both server or client. Alternatively, you can implement server and client in a single project and enable the needed functionality using command-line arguments or conditional compilation.

Have a go hero – Extending the chat server and client

This example has shown us how to send a simple text. If you now go on and define a schema for how the communication should work, you can use it as a base for more complex communication. For instance, if you want to enable the client to receive a list of all other clients (and their usernames), you need to define that the server will return such a list if it gets a special message from a client. You can use special text commands such as `/allClients`, or you can implement a more complex message structure using `QDataStream` or JSON serialization. Therefore, you have to parse all messages received by the server before forwarding them to all the connected clients. Go ahead and try to implement such a requirement yourself.

By now, it is possible that multiple users have chosen the same username. With the new functionality of getting a user list, you can prevent this from happening. Therefore, you have to send the username to the server that keeps track of them. In the current implementation, nothing stops the client from sending messages under a different username each time. You can make the server handle usernames instead of trusting the client's each message.

Synchronous network operations

The example we explained uses a nonblocking, asynchronous approach. For example, after asynchronous calls such as `connectToHost()`, we do not block the thread until we get a result, but instead, we connect to the socket's signals to proceed. On the Internet as well as Qt's documentation, on the other hand, you will find dozens of examples explaining the blocking and the synchronous approaches. You will easily spot them by their use of `waitFor...()` functions. These functions block the current thread until a function such as `connectToHost()` has a result—the time `connected()` or `error()` will be emitted. The corresponding blocking function to `connectToHost()` is `waitForConnected()`. The other blocking functions that can be used are `waitForReadyRead()`, which waits until new data is available on a socket for reading; `waitForBytesWritten()`, which waits until the data has been written to the socket; and `waitForDisconnected()`, which waits until the connection has been closed.

Look out! Even if Qt offers these `waitFor...()` functions, do not use them! The synchronous approach is not the smartest one, since it will freeze your game's GUI. A frozen GUI is the worst thing that can happen in your game, and it will annoy every user. So, when working inside the GUI thread, you are better to react to the `QIODevice::readyRead()`, `QIODevice::bytesWritten()`, `QAbstractSocket::connected()`, and `QAbstractSocket::disconnected()` signals.

`QAbstractSocket` is the base class of `QTcpSocket` as well as of `QUdpSocket`.

Following the asynchronous approach shown, the application will only become unresponsive while your own slots are being executed. If your slots contain more heavy computations, you will need to move them to an extra thread. Then, the GUI thread will only get signals, passing the new messages, and to send, it will simply pass the required data to the worker thread. This way, you will get a super fluent velvet GUI.

Using UDP

In contrast to TCP, UDP is unreliable and connectionless. Neither the order of packets nor their delivery is guaranteed. These limitations, however, allow UDP to be very fast. So, if you have frequent data, which does not necessarily need to be received by the peer, use UDP. This data could, for example, be real-time positions of a player that get updated frequently or live video/audio streaming. Since `QUdpSocket` is mostly the same as `QTcpSocket`—both inherit `QAbstractSocket`—there is not much to explain. The main difference between them is that TCP is stream-oriented, whereas UDP is datagram-oriented. This means that the data is sent in small packages, containing among the actual content, the sender's as well as the receiver's IP address and port number.

Unlike `QTcpSocket` and `QTcpServer`, UDP does not need a separate server class because it is connectionless. A single `QUdpSocket` can be used as a server. In this case, you have to use `QAbstractSocket::bind()` instead of `QTcpServer::listen()`. Like `listen()`, `bind()` takes the addresses and ports that are allowed to send datagrams as arguments. Note that TCP ports and UDP ports are completely unrelated to each other.

Whenever a new package arrives, the `QIODevice::readyRead()` signal is emitted. To read the data, use the `receiveDatagram()` or `readDatagram()` function. The `receiveDatagram()` function accepts an optional `maxSize` argument that allows you to limit the size of the received data. This function returns a `QNetworkDatagram` object that contains the datagram and has a number of methods to get the data. The most useful of them are `data()`, which returns the payload as a `QByteArray` as well as `senderAddress()` and `senderPort()` that allow you to identify the sender.

The `readDatagram()` function is a more low-level function that takes four parameters. The first one of the `char*` type is used to write the data in, the second specifies the amount of bytes to be written, and the last two parameters of the `QHostAddress*` and `quint16*` types are used to store the sender's IP address and port number. This function is less convenient, but you can use it more efficiently than `receiveDatagram()`, because it's possible to use the same data buffer for all datagrams instead of allocating a new one for each datagram.

`QUdpSocket` also provides the overloaded `writeDatagram()` function for sending the data. One of the overloads simply accepts a `QNetworkDatagram` object. You can also supply the data in the form of `QByteArray` or a `char*` buffer with a size, but in these cases, you also need to specify the recipient's address and port number as separate arguments.

Time for action – Sending a text via UDP

As an example, let's assume that we have two sockets of the `QUdpSocket` type. We will call the first one `socketA` and the other `socketB`. Both are bound to the localhost, `socketA` to the `52000` port and `socketB` to the `52001` port. So, if we want to send the string `Hello!` from `socketA` to `socketB`, we have to write in the application that is holding `socketA`:

```
socketA->writeDatagram(QByteArray("Hello!"),
                       QHostAddress("127.0.0.1"), 52001);
```

The class that holds `socketB` must have the socket's `readyRead()` signal connected to a slot. This slot will then be called because of our `writeDatagram()` call, assuming that the datagram is not lost! In the slot, we read the datagram and the sender's address and port number with:

```
while (socketB->hasPendingDatagrams()) {
    QNetworkDatagram datagram = socketB->receiveDatagram();
    qDebug() << "received data:" << datagram.data();
    qDebug() << "from:" << datagram.senderAddress()
             << datagram.senderPort();
}
```

As long as there are pending datagrams—this is checked by `hasPendingDatagrams()`—we read them using the high-level `QNetworkDatagram` API. After the datagram was received, we use the getter functions to read the data and identify the sender.

Have a go hero – Connecting players of the Benjamin game

With this introductory knowledge, you can go ahead and try some stuff by yourself. For example, you can take the game Benjamin the elephant and send Benjamin's current position from one client to another. This way, you can either clone the screen from one client to the other, or both clients can play the game and, additionally, can see where the elephant of the other player currently is. For such a task, you would use UDP, as it is important that the position is updated very fast while it isn't a disaster when one position gets lost.

Keep in mind that we only scratched the surface of networking due to its complexity. Covering it fully would have exceeded this beginner's guide. For a real game, which uses a network, you should learn more about Qt's possibilities for establishing a secure connection via SSL or some other mechanism.

Pop quiz

Q1. Which class can you use to read the data received over the network?

1. `QNetworkReply`
2. `QNetworkRequest`
3. `QNetworkAccessManager`

Q2. What should you usually do with the `QNetworkReply *reply` object in the `finished()` signal handler?

1. Delete it using `delete reply`
2. Delete it using `reply->deleteLater()`
3. Don't delete it

Q3. How to ensure that your application won't freeze because of processing an HTTP request?

1. Use `waitForConnected()` or `waitForReadyRead()` functions
2. Use `readyRead()` or `finished()` signals
3. Move `QNetworkAccessManager` to a separate thread

Q4. Which class can you use to create a UDP server?

1. `QTcpServer`
2. `QUdpServer`
3. `QUdpSocket`

Summary

In the first part of this chapter, you familiarized yourself with `QNetworkAccessManager`. This class is at the heart of your code whenever you want to download or upload files to the internet. After having gone through the different signals that you can use to fetch errors, to get notified about new data or to show the progress, you should now know everything you need on that topic.

The example about the Distance Matrix API depended on your knowledge of `QNetworkAccessManager`, and it shows you a real-life application case for it. Dealing with JSON as the server's reply format was a summary of `Chapter 4`, *Qt Core Essentials*, but it was highly needed since Facebook and Twitter only use JSON to format their network replies.

In the last section, you learned how to set up your own TCP server and clients. This enables you to connect different instances of a game to provide the multiplayer functionality. Alternatively, you were taught how to use UDP.

You are now familiar with Qt widgets, the Graphics View framework, the core Qt classes, and the networking API. This knowledge will already allow you to implement games with rich and advanced functionality. The only large and significant part of Qt we will explore is Qt Quick. However, before we get to that, let's consolidate our knowledge of what we already know and investigate some advanced topics.

Now we are returning to the world of widgets. In `Chapter 3`, *Qt GUI Programming*, we only used the widget classes provided by Qt. In the next chapter, you will learn to create your own widgets and integrate them into your forms.