Sample Code

Implementing netcat with Network Framework

Build a simple `netcat` tool that establishes network connections and transfers data.

Download

macOS 10.14+ | Xcode 12.0+

Overview

The `netcat` tool (abbreviated `nc` on macOS) is a UNIX tool that lets you:

- Establish outbound TCP and UDP connections.
- Listen for inbound TCP and UDP connections.
- Transfer data between a network connection and `stdin` and `stdout`.

This sample code shows how you can build `nwcat`, which provides this functionality and also adds TLS and Bonjour support.

The `nwcat` tool supports two modes: one that makes an outbound connection and one that listens for an inbound connection. You select the mode using command-line arguments. To make an outbound connection to port 80 on `example.com`, you run:

```
$ nwcat example.com 80
```

To listen for an inbound connection, you supply the `-l` option and the port number to listen on:

```
$ nwcat -l 12345
```

By default, `nwcat` uses TCP. The `-u` argument switches to UDP, and the `-t` argument adds TLS to TCP connections and DTLS to UDP connections. The `-b` argument enables use of Bonjour service names rather than host names.

Create an Outbound Connection

In the Network framework, a bidirectional flow of data is represented as a connection object (`nw_connection_t`). If you're using TCP, there's a direct mapping between the connection object and the underlying TCP connection. If you're using UDP, a connection object represents a bidirectional flow of datagrams between a local port and a port on a specific remote peer.

To create an outbound connection object, you must supply a parameters object (`nw_parameters_t`) and an endpoint (`nw_endpoint_type_t`).

Create a Parameters Object. A parameters object, of type `nw_parameters_t`, holds all the parameters necessary to configure a network connection. These include:

- The protocols involved, like TCP or UDP, or whether to enable TLS.
- Any options for those protocols.
- Any constraints on the connection, like whether or not to use the cellular interface.

To create a parameters object, call either the `nw_parameters_create_secure_tcp(_:_:)` or the `nw_parameters_create_secure_udp(_:_:)` convenience function, depending on whether you want to use TCP or UDP.

```
nw_parameters_configure_protocol_block_t configure_tls = NW_PARAMETERS_DISABLE_PROTC
if (g_use_udp) {
    parameters = nw_parameters_create_secure_udp(
        configure_tls,
        NW_PARAMETERS_DEFAULT_CONFIGURATION
    );
} else {
    parameters = nw_parameters_create_secure_tcp(
        configure_tls,
        NW_PARAMETERS_DEFAULT_CONFIGURATION
    );
}
```

These functions take two arguments:

- The first configures the security protocol for the connection. Pass `NW_PARAMETERS_DISABLE_PROTOCOL` to use no security.

- The second configures the transport protocol for the connection. Pass `NW_PARAMETERS_DEFAULT_CONFIGURATION` to get a default configuration.

> **Note**
>
> If you're curious about enabling TLS or configuring protocol options, look at the `create_outbound_connection` function for examples of these more advanced features. For example, `nwcat` supports setting a TLS pre-shared key for both TLS listeners and clients.

Create an Endpoint. An endpoint, of type `nw_endpoint_type_t`, holds a network host or service name. For an outbound connection, the endpoint determines the remote host to which you want to connect. In most cases this consists of a host name and a port number, but there are other options. For example, you can also create endpoints that target a Bonjour service.

In the `nwcat` tool, the user supplies a host name and a port number via command-line arguments, and you will need to create an endpoint from those two strings. You do this by calling `nw_endpoint_create_host(_:_:)`.

```
nw_endpoint_t endpoint = nw_endpoint_create_host(name, port);
```

These strings support both symbolic and numeric values:

- The host name can be a DNS name, like `example.com`, or the string representation of an IP address, like `"203.0.113.7"` for IPv4 or `"2606:2800:220:1:248:1893:25c8:1946"` for IPv6.
- The port can be a numeric value, like `"80"`, or a service name, like `"https"`.

> **Important**
>
> If you pass in a DNS name, the connection object takes care of DNS resolution for you, handling the complex issue of deciding what IP address to connect to. For outbound connections there's no need to do your own DNS resolution, and in most cases doing so will result in a worse user experience.

Create the Connection Object. Once you have your parameters object and endpoint, you can create a connection object by calling `nw_connection_create(_:_:)`.

```
nw_connection_t connection = nw_connection_create(endpoint, parameters);
```

Start the Connection. To start the connection establishment process:

1. Call `nw_connection_set_queue(_:_:)` to set the `dispatch_queue_t` on which all callbacks will be scheduled. For a simple application like `nwcat`, you can use the main queue for your callbacks. A more complex application would typically use a custom serial queue instead.

2. Install any update handler blocks. The most important of these is the state changed handler, discussed below.

3. Start the connection by calling `nw_connection_start(_:)`.

You must set your queue before starting the connection, and you cannot change the queue after that.

```
void
start_connection(nw_connection_t connection)
{
    nw_connection_set_queue(connection, dispatch_get_main_queue());
    nw_retain(connection);
    nw_connection_set_state_changed_handler(connection, ^(nw_connection_state_t stat
        … your state changed handler …
    });
    nw_connection_start(connection);
}
```

Important

The connection object closes the underlying network connection when its last reference is released. Thus, you must retain the connection object, by calling `nw_retain`, until you're done with the connection.

A state changed handler is a block that's called by the connection object whenever the connection state changes. For a simple application, like `nwcat`, you can use a very simple state changed handler.

```
if (state == nw_connection_state_waiting) {
    … tell the user that a connection couldn't be opened but will retry when conditi
} else if (state == nw_connection_state_failed) {
    … tell the user that the connection has failed irrecoverably …
} else if (state == nw_connection_state_ready) {
    … tell the user that you are connected …
} else if (state == nw_connection_state_cancelled) {
    nw_release(connection);
}
```

Make sure you handle the `nw_connection_state_cancelled` state. Once the connection is no longer needed, you have to release the reference you took when you started the connection.

Listen for an Inbound Connection

A listener object, of type `nw_listener_t`, listens for inbound connections and creates a new connection object for each one. To create a listener object you must supply a parameters object (`nw_parameters_t`) to indicate what protocols to use and information about the local endpoint on which you want to listen.

Create a Parameters Object. Creating a parameters object for a listener object is very similar to creating a parameters object for an outbound connection. Use the `nw_parameters_create_secure_tcp(_:_:)` and `nw_parameters_create_secure_udp(_:_:)` convenience functions to define which protocols you want your listener to use. These parameters will be applied to any inbound connections your listener accepts. For example, if you enable TLS in the parameters object, all inbound connections will negotiate TLS once you call `nw_connection_start(_:)`.

Set a Local Endpoint. A listener object must know what local endpoint to listen on, that is, the endpoint to which clients must connect. The local endpoint can include a port number and an interface address, both of which are optional. If you don't specify a port number, the system chooses a port for you. If you don't specify an interface address, the system listens on all interfaces and addresses.

Most applications don't need to listen on a specific interface address and thus can create a listener using the `nw_listener_create_with_port(_:_:)` convenience function. However, `nwcat` allows the user to specify an interface address (via a command line argument) and thus you have to use a slightly more complex technique. If the user has specified an interface address or a port, you can call `nw_endpoint_create_host(_:_:)` to create an endpoint representing the address to listen on, and then call `nw_parameters_set_local_endpoint(_:_:)` to apply that to your parameters object.

```
if (address || port) {
    nw_endpoint_t local_endpoint = nw_endpoint_create_host(address ? address : "::",
    nw_parameters_set_local_endpoint(parameters, local_endpoint);
    nw_release(local_endpoint);
}
```

When calling `nw_endpoint_create_host(_:_:)` to create an endpoint for the local address, keep the following in mind:

- The `port` parameter can either be a numeric string, like `"443"`, or a service name, like `"https"`.

- If you pass `"0"` to the `port` parameter the system will choose a port on your behalf.

Create the Listener Object. Once you've set up your parameters object, you can create a listener object by calling `nw_listener_create(_:)`.

```
nw_listener_t listener = nw_listener_create(parameters);
```

Start the Listener. Starting a listener object is much like starting a connection object, with one significant difference: in addition to setting a state changed handler, you must also set a new connection handler, which is called whenever the listener object receives a new inbound connection.

```
nw_listener_set_queue(listener, dispatch_get_main_queue());
nw_retain(listener);
nw_listener_set_state_changed_handler(listener, ^(nw_listener_state_t state, nw_erro
    … your state changed handler …
});
nw_listener_set_new_connection_handler(listener, ^(nw_connection_t connection) {
    … your new connection handler …
});
nw_listener_start(listener);
```

Accept or Reject Inbound Connections. Your new connection handler is responsible for either starting the network connection or rejecting it. The `nwcat` command can only handle one connection at a time, so if there's already a network connection in place, call `nw_connection_cancel(_:)` to reject the new connection. If not, retain the network connection and then run the connection using the same `start_connection` function you used in the outbound case.

```
if (g_inbound_connection != NULL) {
    nw_connection_cancel(connection);
} else {
    g_inbound_connection = connection;
    nw_retain(g_inbound_connection);

    start_connection(g_inbound_connection);
}
```

Transfer Data

Once you have created and started a connection, either outbound or inbound, you'll need code to transfer data on that connection. Each connection has two directions:

- Inbound data is received from the network connection and written to `stdout`.
- Outbound data is read from `stdin` and sent to the network connection.

Both directions are asynchronous. When receiving data from the network, you supply a completion handler that's called when data is available. Similarly, when writing data to the network, you supply a completion handler that's called when the data has been accepted for transmission.

When working with asynchronous networking, you need to consider flow control. For example, if you receive data from the network faster than you can write it to `stdout`, you'll waste a lot of memory buffering that data. You'll have similar problems if you read data from `stdin` faster than you can send it over the network. You can solve this problem by using asynchronous routines for reading from `stdin` and writing to `stdout`. The basic strategy is this:

1. Start an asynchronous read.

2. When the read completes, start an asynchronous write.

3. When the write completes, set up the next asynchronous read, which starts again at step 1.

You use a similar strategy for both inbound and outbound data, but there are some subtle differences, discussed in the sections that follow.

Receive Data. You can receive data with code like this.

```
void
receive_loop(nw_connection_t connection)
{
    nw_connection_receive(connection, 1, UINT32_MAX, ^(dispatch_data_t content, nw_c

        nw_retain(context);
        dispatch_block_t schedule_next_receive = ^{
            … discussed below …
            nw_release(context);
        };

        if (content != NULL) {
            schedule_next_receive = Block_copy(schedule_next_receive);
            dispatch_write(STDOUT_FILENO, content, dispatch_get_main_queue(), ^(__ur
                if (stdout_error != 0) {
                    … error logging …
                } else {
                    schedule_next_receive();
                }
```

```
                Block_release(schedule_next_receive);
            });
        } else {
            // No content, so directly schedule the next receive
            schedule_next_receive();
        }
    });
}
```

> **Note**
>
> While this function is called `receive_loop`, it's not actually a loop. Rather, it's the asynchronous equivalent of a loop, bouncing between an asynchronous receive from the network and an asynchronous write to `stdout`.

The function starts by calling `nw_connection_receive(_:_:_:_:)`, which is an asynchronous function that receives data from the connection object. The function has two parameters that control the minimum and maximum amount of data to be received. The exact amount of data received isn't relevant here, so pass 1 and `UINT32_MAX` respectively.

> **Note**
>
> The minimum and maximum receive parameters are useful when you're implementing a record-oriented protocol. Many network protocols transfer records over a TCP stream by sending a record length followed by the record. If you're working with such a protocol, you can perform an initial receive for the length field and then, once you know the length of the record, do a second receive for the full record body.

When the receive is complete, `nw_connection_receive(_:_:_:_:)` calls the completion handler that you pass it. The completion handler has four parameters:

- A `dispatch_data_t` which, if not `NULL`, contains the data received.
- A content context, discussed below.
- An `is_complete` parameter that is true if the data received is the last part of a logical unit of data.
- A `receive_error` parameter, which is not `NULL` if an error occurred during the receive process.

A content context, of type `nw_content_context_t`, holds extra information about the data received. A typical application that uses only TCP can often ignore this value entirely. However, a

`netcat` implementation must work equally well with TCP and UDP, and you need the content context to do that.

The completion handler you pass to `nw_connection_receive(_:_:_:_:)` does the following:

1. It processes any data that was received by starting an asynchronous write to `stdout`.
2. When that asynchronous write completes—or immediately if there was no content—it calls `schedule_next_receive` to continue the receive.

In `schedule_next_receive` you must handle three cases:

- If you just received the end of the data stream, call `exit` to terminate the program. This is how the program stops when the remote peer closes the network connection.
- If the receive failed with an error, handle that error.
- Otherwise, start the next asynchronous receive by calling `receive_loop`.

```
nw_retain(context);
dispatch_block_t schedule_next_receive = ^{
    if (is_complete &&
        (context == NULL || nw_content_context_get_is_final(context))) {
        exit(0);
    }
    if (receive_error == NULL) {
        receive_loop(connection);
    } else {
        … error logging …
    }
    nw_release(context);
};
```

Important

You must process any received data before checking for the other states (end of data stream or error) because it's possible for your completion handler to be called with both data and one of the other states.

To check for the end of the data stream:

- Generally, you can use the technique shown by `schedule_next_receive`, that is, check both the `is_complete` flag and that the context is marked as final or is not present. This technique works correctly for both TCP and UDP, and thus is necessary for a `netcat` implementation.

- If you only handle TCP, you can simply test the `is_complete` flag.

Send Data. Your send code should have the same basic structure as your receive code.

```
void
send_loop(nw_connection_t connection)
{
    dispatch_read(STDIN_FILENO, 8192, dispatch_get_main_queue(), ^(dispatch_data_t _
        if (stdin_error != 0) {
            … error logging …
        } else if (read_data == NULL) {
            … handle end of file …
        } else {
            nw_connection_send(connection, read_data, NW_CONNECTION_DEFAULT_MESSAGE_
                if (error != NULL) {
                    … error logging …
                } else {
                    send_loop(connection);
                }
            });
        }
    });
}
```

There are, however, some subtle differences:

- If, as in this case, you support UDP, you must limit the amount of data you read from `stdin`. If you read more than 64 Kibibytes (KiB), the resulting read won't fit in a single UDP datagram. This code use a 8 KiB limit.
- You must tell the network connection to send the data as a single message by calling `nw_connection_send(_:_:_:_:_:)` with `NW_CONNECTION_DEFAULT_MESSAGE_CONTEXT` and passing true to the `is_complete` parameter. This approach is appropriate for both TCP and UDP connections. For TCP connections, message boundaries don't affect how the protocol sends data, but the boundaries are required for sending UDP datagrams.
- Finally, when you receive an end of file from `stdin`, you must close the send side of your connection. The code for this is shown below.

```
nw_connection_send(connection, NULL, NW_CONNECTION_FINAL_MESSAGE_CONTEXT, true, ^
    if (error != NULL) {
        … handle error …
    }
```

```
    // Stop reading from stdin, so don't schedule another send_loop
});
```

This passes a special context, `NW_CONNECTION_FINAL_MESSAGE_CONTEXT`, and passes true to the `is_complete`. Together, these actions indicate that no more data will be sent, allowing the network connection to close the sending side of the connection.

See Also

Connections and Listeners

`typealias` `nw_connection_t`

A bidirectional data connection between a local endpoint and a remote endpoint.

`typealias` `nw_listener_t`

An object you use to listen for incoming network connections.

`typealias` `nw_browser_t`

An object you use to browse for available network services.

`typealias` `nw_connection_group_t`

An object you use to communicate with a group of endpoints, such as an IP multicast group on a local network.

`typealias` `nw_ethernet_channel_t`

An object you use to send and receive custom Ethernet frames.