Socketry Mockery

or

POSIX Sockets and Boost:
What Do They Look Like?
Can We Mock Them??
Let's Find Out!

Boost sockets

- Part of a huge modern C++ framework
- Strongly typed: static protocol details

  PRO: hard to do something unexpected

  CON: hard to do something unexpected
- Generic operations are templates

  ```
  read(socket, buffer);
  ```
- Special operations are members

  ```
  tcp_socket.connect(address);
  udp_socket.send_to(buffer, address);
  ```

Boost example

- A class to manage a service connection

```
class connection {
public:
    connection(ip::tcp::socket&&);
     // public API...
private:
    ip::tcp::socket socket;
    // private gubbins...
};
```

Boost example (usage)

- Make a “client” connection:

```
socket.connect(address);
connection c(std::move(socket));
```

- Make a “server” connection:

```
acceptor.accept(socket);
connection c(std::move(socket));
```

- Lovely! Now how do we test it?
  - Unit tests shouldn't use a network.
  - How about a local domain socket pair?

Boost problem

- Mock a network socket using local sockets?

```
local::stream_protocol::socket s1;
local::stream_protocol::socket s2;
local::connect_pair(s1, s2);
connection c(std::move(s1)); // NOPE!
```

- Protocol details are specified by socket type.
- Need to allow different static types.

Boost solution

- Templates everywhere:

```
template <class Socket>
class connection {
public:
    connection(Socket&&);
     // public API (all templates)
private:
    Socket socket;
    // private gubbins (all templates)
};
```

POSIX sockets

- Old-school system-level C API
- Weak typing - “everything is a file”
- Integer handles
  - analogous to raw pointers, need taming
- No type-checking:

```
read(socket, buffer, size);
read(file, buffer, size);
sendto(udp_socket, …);
sendto(file, …); // Whoops!
```

POSIX taming

- Movable resource management class

```
class descriptor {
public:
  descriptor(int fd);
  ~descriptor(); // close if open
  descriptor(descriptor &&);
  descriptor &operator=(descriptor&&);
private:
  int fd;
};
```

POSIX example

- A class to manage a service connection

```
class connection {
public:
    connection(descriptor &&);
     // public API...
private:
    descriptor socket;
    // private gubbins...
};
```

POSIX example (usage)

- Make a “client” connection:

```
connection c(connect(address));
```

- Make a “server” connection:

```
connection c(accept(listener));
```

- Test with a local domain socket pair

```
local_pair sockets;
connection c(std::move(sockets[0]));
descriptor & tester = sockets[1];
```

- Lovely!

Conclusions

- Tension between type safety and testability
- There's more than one way to resolve it
  - Template proliferation á la Boost
  - Loss of type checking á la POSIX
  - Runtime polymorphism, type erasure, ...
- Finding a good solution needs judgement
- Use Your Brain!