Network Programming

Lecture 2

Building Highly Distributed Systems

Within 5 Minutes

Jonas Kunze

University of Mainz – NA62
Outline

- Motivation
- Boost.Asio
- Message Passing
- ØMQ
- Apache Thrift
TCP in C code

- The BSD socket API is minimalistic
  - No intrinsic multithreading support
    - Handling multiple connections typically via fork()
  - No data management (messaging)
  - Configuration a bit awkward

- There is no exception handling or OOP in C

- There is no C++ socket API in the std library
  - `std::socket` will never come
Outline

- Motivation
- **Boost.Asio**
  - Asynchronous operations
  - Concurrency without threads
  - Multithreading
- Message Passing
- ØMQ
- Apache Thrift
Boost.Asio

- Boost.Asio is a C++ library for low-level I/O programming with a consistent **asynchronous** model including a **BSD** socket interface

<table>
<thead>
<tr>
<th>BSD Socket API (Linux)</th>
<th>Equivalents in Boost.Asio</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket descriptor – int</td>
<td>For TCP: ip::tcp::socket</td>
</tr>
<tr>
<td></td>
<td>For UDP: ip::udp::socket</td>
</tr>
<tr>
<td>sockaddr_in, sockaddr_in6</td>
<td>For TCP: ip::tcp::endpoint</td>
</tr>
<tr>
<td></td>
<td>For UDP: ip::udp::endpoint</td>
</tr>
<tr>
<td>accept()</td>
<td>For TCP: ip::tcp::acceptor::accept()</td>
</tr>
<tr>
<td>bind()</td>
<td>For TCP: ip::tcp::socket::bind()</td>
</tr>
<tr>
<td></td>
<td>For UDP: ip::udp::socket::bind()</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Boost.Asio

- Boost.Asio uses an object as an interface to the operating system: io_service
- The io_service object is passed to I/O objects like tcp::socket
- The I/O objects will forward requests to the io_service object
  - io_service runs the required syscalls

```cpp
boost::asio::io_service io_service;
boost::asio::ip::tcp::socket socket(io_service);
boost::asio::ip::tcp::resolver resolver(io_service); // gethostbyname wrapper
socket.connect(*resolver.resolve({hostname, portNum}));
socket.send(boost::asio::buffer("message"));
```
Boost.Asio: Asynchronous operations

- I/O objects implement non-blocking/asynchronous operations
  - E.g. boost::asio::ip::tcp::socket::async_connect

- Completion handler function passed to async_ functions

- io_service.run() calls the completion handler as soon as results of async_ functions are available

![Diagram of asynchronous operations](image)
Boost.Asio: Asynchronous operations

- Simple TCP connection example:
  ```cpp
  void MyClass::handle_connect(const boost::system::error_code& error) {
    if (!error) { doSomething(); }
  }
  ...
  socket.async_connect(socket, *resolver.resolve({hostname, portNum}),
    boost::bind(&MyClass::handle_connect, this, boost::asio::placeholders::error));
  workWhileConnecting();
  io_service.run(); // Runs handle_connect as soon as the connection is established
  ```

- Even simpler with C++11 using a lambda function:
  ```cpp
  socket.async_connect(*resolver.resolve({hostname, portNum}),
    [this](boost::system::error_code error, tcp::resolver::iterator) {
      if (!error) { doSomething(); }
    });
  workWhileConnecting();
  io_service.run();
  ```
Concurrency without threads
Handling multiple TCP connections

- One io_service can handle several I/O objects and async operations
- io_service::run() will block until all requests have been handled

```cpp
sock1.async_read_some(readBuffer, [](boost::system::error_code error, std::size_t){
    if (!error) {std::cout << "Socket 1 received something" << std::endl;}
});

sock2.async_read_some(readBuffer, [](boost::system::error_code error, std::size_t){
    if (!error) {std::cout << "Socket 2 received something" << std::endl;}
});

io_service.run();
cout << “Both sockets received something” << endl;
```
**Multithreading**

- `io_service::run()` can be called by multiple threads simultaneously
- `async_` operations will be distributed among these threads
- A common approach is to launch a thread pool running the whole lifetime
  - N threads spawned at the beginning handling all `async_` operations
  - Recursive calls of `async_` operations (`io_service::run()` never returns)
Server with a Thread Pool

boost::asio::io_service io_service;
EchoServer s(io_service, 1234); // calls socket.async_read...

std::vector<std::thread> threadPool;
for (std::size_t i = 0; i < std::thread::hardware_concurrency(); ++i) {
    threadPool.push_back(
        std::thread(
            [&]() {
                io_service.run();
            }
        )
    );
}

for(auto& thread : threadPool){
    thread.join();
}
Outline

- Motivation
- Boost.Asio
- Message Passing
- ØMQ
- Apache Thrift
Message passing via TCP

- **TCP only offers a continuous data stream**
  - Although data is typically sent to sockets in chunks, the receiver may see different chunks (scaling window)
  - The application layer program has to split the stream into messages

- **There are three possible approaches to indicate messages in the stream:**
  - Protocol defines the message length *implicitly*
  - The message length is *explicitly* specified in a message header
  - **Line-Based**: Messages in the stream are separated by delimiters
Message passing via TCP

- **Line-Based approach easily implemented with Boost.Asio:**
  
  ```cpp
  boost::asio::read_until(socket, msgBuffer, "\n\n");
  ```

- **Other approaches are much more efficient**
  - But also hard work to implement

 **No need to reinvent the wheel!**

- ØMQ implements fast message passing using the explicit format
Building Highly Distributed Systems Within 5 Minutes

Outline

- Motivation
- Boost.Asio
- Message Passing
- ØMQ
  - Messaging Patterns
  - Broker
  - Multithreading
- Apache Thrift
What ØMQ says about their socket library (http://zguide.zeromq.org/page:all):

We took a normal TCP socket, injected it with a mix of radioactive isotopes stolen from a secret Soviet atomic research project, bombarded it with 1950-era cosmic rays (...) It's sockets on steroids.
ØMQ

- ØMQ offers a uniform API (ØMQ sockets) to transport messages over different channels:
  - TCP, multicast, IPC (process to process), inproc (thread to thread)

- Cross Platform (Linux, Windows, Mac, etc...)

- Implementations in many(!!!) different languages:
  - C/C++, Java, Python, Ruby, PHP, Perl, Node.js, C#, Clojure, CL, Delphi, Erlang, F#, Felix, Go, Haskell, Haxe, Lua, Objective-C, Q, Racket, Scala...

- OpenSource
ØMQ – Messaging Patterns

- ØMQ sockets express several messaging patterns
  - REQ and REP
  - PUB and SUB
  - PUSH and PULL
  - REQ and ROUTER
  - DEALER and REP
  - DEALER and ROUTER
  - DEALER and DEALER
  - ROUTER and ROUTER
  - PAIR and PAIR
ØMQ – REQ-REP

- Clients “connect” to a service and send a REQuest message
- The server REPLies to each request with a single message
- Sending is done asynchronously in the background
  - User writes simple non-blocking code
  - If the remote endpoint is down the message will be sent later
- This represents a remote procedure call pattern
ØMQ – Simple REQ Client

```c
int main() {
    zmq::context_t context(1); // Similar to io_service
    zmq::socket_t socket(context, ZMQ_REQ);
    socket.connect("tcp://REPServerHostName:5555");

    zmq::message_t request(6);
    memcpy((void *)request.data(), "Hello", 5);
    socket.send(request);

    zmq::message_t reply;
    socket.recv(&reply);
    return 0;
}
```
ØMQ – Simple REP Server

```cpp
int main() {
    zmq::context_t context(1); // Similar to io_service
    zmq::socket_t socket(context, ZMQ_REP);
    socket.bind("tcp://*:5555");

    while (true) {
        zmq::message_t request;
        socket.recv(&request);

        zmq::message_t reply(5);
        memcpy((void *)reply.data(), "World", 5);
        socket.send(reply);
    }
    return 0;
}
```
ØMQ – REQ-REP Notes

- The REQ-REP socket pair is in lockstep
  - Server and client have to call send and recv alternately
  - Server automatically sends to the node it got the last message (recv) from
    - All the connection handling is done by ØMQ

- The connection can be established from both sides (true for all patterns)
**ØMQ – PUB-SUB**

- Server PUBlishes data to all connected clients
- Clients SUBscribe to the data by connecting to the server
- Subscription to messages by data prefix (filter)
- If no client is connected the data will be lost
ØMQ – Pipeline

- Ventilator: Produces task that can be processed in parallel

- These tasks are then PUSHed evenly to the connected Workers

- After processing the tasks the Workers push the results to a Sink

- Basic load balancing
```c
int main() {
    zmq::context_t context(1);
    zmq::socket_t ventilatorSocket(context, ZMQ_PULL);
    ventilatorSocket.connect("tcp://ventilator:5557");

    zmq::socket_t sinkSocket(context, ZMQ_PUSH);
    sinkSocket.connect("tcp://sink:5558");

    while (1) {
        zmq::message_t task;
        ventilatorSocket.recv(&task); // PULL
        zmq::message_t result = doSomeWork(task);
        sinkSocket.send(result); // PUSH
    }
}
```
ØMQ – N-to-M communication

- So far we had N workers pulling from one ventilator
- It is possible to connect one ØMQ socket to several endpoints
  
  ```
  ventilatorSocket.connect("tcp://ventilator1:5557");
  ventilatorSocket.connect("tcp://ventilator2:5557");
  ```
- The messages will be scheduled fairly from all ventilators
ØMQ – Broker

- With the last design the workers need to know all ventilators (hostnames)
- If a new ventilator is added all the workers have to connect (evtl. Restart)
- One easy design to fix this: Add a central broker
ØMQ – Broker

- This is easily implemented with a zmq_proxy forwarding messages:

  ```cpp
  zmq::context_t context(1);

  // Socket facing ventilators
  zmq::socket_t frontend(context, ZMQ_PULL);
  frontend.bind("tcp://*:5556");

  // Socket facing workers
  zmq::socket_t backend(context, ZMQ_PUSH);
  backend.bind("tcp://*:5557");

  // Pass messages from ventilators to workers
  zmq_proxy(frontend, backend, NULL);
  ```
ØMQ – Broker

- Now you only have to change one line in the ventilator:
  
  ```
  socket.bind("tcp://*:5559"); → socket.connect("tcp://broker:5559");
  ```

- And connect the worker to the broker instead of the ventilators
  
  ```
  ventilator.connect("tcp://ventilator1:5557");
  ventilator.connect("tcp://ventilator2:5557"); …
  ```
  Turns to:
  
  ```
  ventilator.connect("tcp://broker:5557");
  ```

- And again you can start ventilators, workers, broker and sink in whatever order you like:

  Messages are queued as close to the receiver as possible
ØMQ – IPC

- So far we used: `socket.bind("tcp://*:5555");`

- To run the same programs locally one should use:
  - `socket.bind("ipc:///tmp/helloWorld");`  // For processes
  - `socket.bind("inproc:///helloWorld");`  // For threads

- Start developing your software with many modules communicating with IPC

- Then outsource heavy loaded services to external boxes just by changing
  - `inproc/ipc://... → tcp://...`
ØMQ – Multithreading

- ØMQ sockets are not thread safe!
- But they are extremely lightweight
  - Create one (or more) sockets per thread
  - Use these ØMQ sockets to exchange messages between the threads
  - Use a proxy to distribute work among the threads
ØMQ – Multithreaded Worker

```cpp
void workerThread(zmq::context_t& context) {
    zmq::socket_t ventilatorProxy(context, ZMQ_PULL);
    ventilatorProxy.connect("inproc://workers");

    zmq::socket_t sink(context, ZMQ_PUSH);
    sink.connect("tcp://sink:5558");

    while (1) {
        zmq::message_t task;
        ventilatorProxy.recv(&task);

        zmq::message_t result = doSomeWork(task);
        sink.send(result);
    }
}
```
ØMQ – Multithreaded Worker

```cpp
int main() {
    zmq::context_t context(1);
    zmq::socket_t ventilatorProxy(context, ZMQ_PULL);
    ventilatorProxy.connect("tcp://broker:5557");
    zmq::socket_t workers(context, ZMQ_PUSH);
    workers.bind("inproc://workers");

    std::vector< std::thread > threadPool;
    for (std::size_t i = 0; i < std::thread::hardware_concurrency(); ++i) {
        threadPool.push_back(std::thread([&]() {
            workerThread(context); // will connect with inproc://workers
        }));
    }

    zmq::proxy(ventilatorProxy, workers, NULL);
}
```
ØMQ – Notes

- With ØMQ messages still need to be translated to procedure executions
- Object serialization has to be implemented on top of ØMQ

- There's much more functionality in ØMQ!
- Read the great guide: [http://zguide.zeromq.org](http://zguide.zeromq.org)
- The examples in this lecture are based on the examples from the zguide
Outline

- Motivation
- Boost.Asio
- Message Passing
- ØMQ
- Apache Thrift
Apache Thrift

- Remote Procedure Calls (RPCs):
  Executing subroutines (functions, methods) on a program running remotely

- Thrift is a scalable cross-language RPC framework developed by Facebook
  - It implements the missing object serialization

- It's an open source project in the Apache Software Foundation
Apache Thrift

- The developer defines services in an Interface Definition Language (IDL) file

- Thrift generates code (Interfaces) to be used to call these services remotely
  - E.g. calling a Java Method from a PHP script running on a remote host

```
thriftd --gen cpp
```

![Diagram of Thrift usage](image-url)
Thrift – Interface Definition

- **Interface Definition Language (.thrift) files**
  - Define namespace, data structures, types, methods, services
  - Similar to C syntax
  - Basic types are bool, byte, i16/32/64, double, string, map,<t1,t2>, list,<t1>, set,<t1>

```thrift
namespace cpp ch.cern.icsc14

enum Operation {
    ADD = 1,
    SUBTRACT = 2,
    MULTIPLY = 3,
    DIVIDE = 4
}

struct Work {
    1: i32 num1,
    2: i32 num2,
    3: Operation op
}

service Calculator {
    i32 calculate(1:Work w)
}
```
Thrift – Compiling Thrift Files

- Thrift compiles the IDL files to server (and client) source code
- It generates thousands of lines of code with placeholders

**Calculator_server.skeleton.cpp:**

```cpp
using namespace ::ch::cern::icsc14;
class CalculatorHandler : virtual public CalculatorIf {
  public:
    CalculatorHandler() {
      // Your initialization goes here
    }

    int32_t calculate(const Work& w) {
      // Your implementation goes here
    }
};
```
Thrift – Documentation

- There is only very little documentation online

- Useful links:
  - http://wiki.apache.org/thrift/ThriftUsage
  - http://thrift-tutorial.readthedocs.org/
  - http://www.slideshare.net/dvirsky/introduction-to-thrift

Good Luck!
Summary

- There is no native C++ library for network programming
- There are many different libraries for different purposes
  - Boost.Asio for easy asynchronous and multithreaded socket programming
  - ØMQ additionally provides message passing and helpful patterns
  - Apache Thrift provides an efficient RPC framework
- All these libraries are cross-platform capable
- ØMQ and Thrift provide interfaces for many languages

Visit https://github.com/JonasKunze for code snippets and these slides