Boost.Asio and Boost.Serialization
Design Patterns for Object Transmission

Bryce Adelstein-Lelbach, Jeoren Habraken, Thomas Heller, Hartmut Kaiser

Boost.Asio: Christopher Kohlhoff
Boost.Serialization: Robert Ramey

STE||AR
stellar.cct.lsu.edu

NSF

Introduction

- My colleagues and I develop a software framework called **HPX**, a general-purpose C++ runtime system for applications of any scale.
  - A **runtime system** manages certain aspects of a program's execution environment.
  - Asynchronous methodology: **future** and **dataflow** models.
  - For shared-memory and distributed-memory systems.
- The purpose of this talk is to share our work, experiences and analysis of Boost.Asio and Boost.Serialization.

Introduction

- Code for all examples can be found at:

  `git@github.com:STEllAR-GROUP/cppnow2013_ot.git`
- Code shown in this talk is slightly different from the code in the git repository:
  - Code in the slides have error handling removed for brevity in some places
- Assume the following:

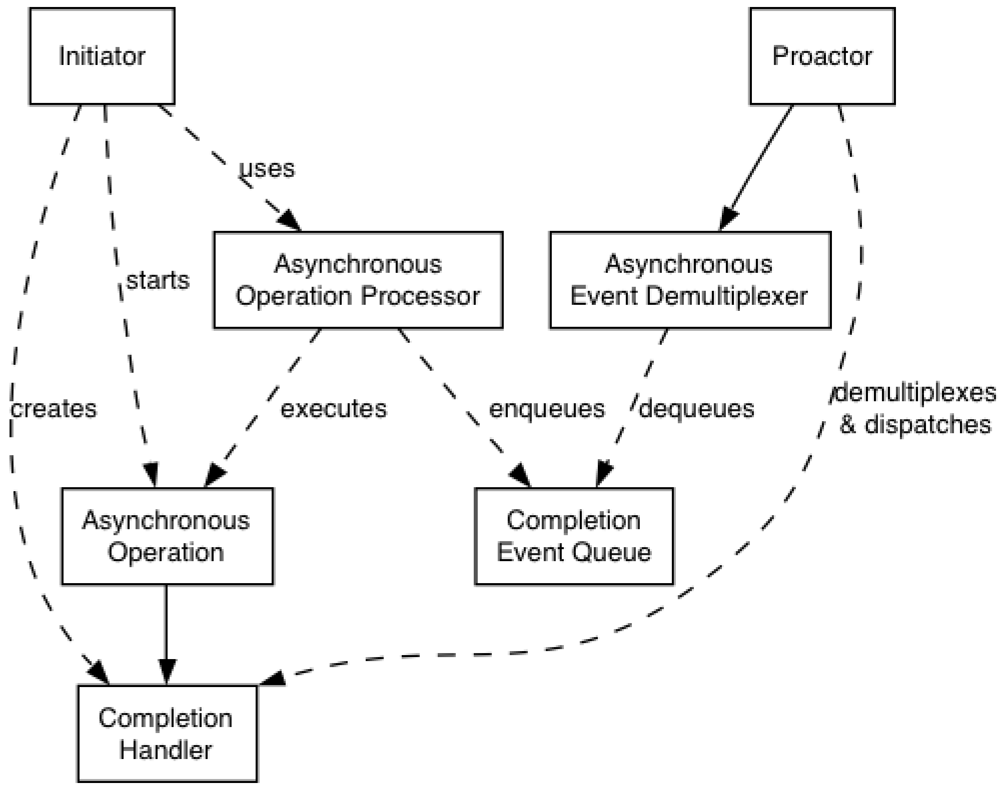
```
using boost::system::error_code;
namespace po = boost::program_options;
namespace asio = boost::asio;
typedef boost::asio::ip::tcp asio_tcp;
```

BOOST.ASIO

Asio

- Boost.Asio: a library for synchronous and asynchronous I/O.
  - Proactor-based design.
- Provides a generic framework for various types of I/O:
  - Network sockets.
    - Asio provides TCP, UDP and ICMP support.
  - Files.
  - Serial ports.
  - Direct Memory Access (DMA).
  - Interprocess communication.

Asio: Proactor Design



Asio: Echo Server

```
int main() {
    asio::io_service io_service;
    asio_tcp::endpoint endpoint(asio_tcp::v4(), 2000);
    asio_tcp::acceptor acceptor(io_service, endpoint);

    for (;;) {
        asio_tcp::socket socket(io_service); acceptor.accept(socket);

        std::size_t const max_length = 1024;
        char msg[max_length];

        std::function<void(error_code const&, std::size_t)>
            f = [&](error_code const& ec, std::size_t bytes)
                {
                    asio::async_write(socket, asio::buffer(msg, bytes),
                        [&](error_code const& ec, std::size_t)
                        {
                            auto buf = asio::buffer(msg, max_length);
                            socket.async_read_some(buf, f);
                        });
                };

        socket.async_read_some(asio::buffer(msg, max_length), f);
        io_service.run();
    }
}
```

Asio: Echo Server

```
int main() {
    asio::io_service io_service;
    asio_tcp::endpoint endpoint(asio_tcp::v4(), 2000);
    asio_tcp::acceptor acceptor(io_service, endpoint);

    for (;;) {
        asio_tcp::socket socket(io_service); acceptor.accept(socket);

        std::size_t const max_length = 1024;
        char msg[max_length];

        std::function<void(error_code const&, std::size_t)>
            f = [&](error_code const& ec, std::size_t bytes)
                {
                    asio::async_write(socket, asio::buffer(msg, bytes),
                        [&](error_code const& ec, std::size_t)
                        {
                            auto buf = asio::buffer(msg, max_length);
                            socket.async_read_some(buf, f);
                        });
                };

        socket.async_read_some(asio::buffer(msg, max_length), f);
        io_service.run();
    }
}
```

Asio: Echo Server

```
int main() {
    asio::io_service io_service;
    asio_tcp::endpoint endpoint(asio_tcp::v4(), 2000);
    asio_tcp::acceptor acceptor(io_service, endpoint);

    for (;;) {
        asio_tcp::socket socket(io_service); acceptor.accept(socket);

        std::size_t const max_length = 1024;
        char msg[max_length];

        std::function<void(error_code const&, std::size_t)>
            f = [&](error_code const& ec, std::size_t bytes)
                {
                    asio::async_write(socket, asio::buffer(msg, bytes),
                        [&](error_code const& ec, std::size_t)
                        {
                            auto buf = asio::buffer(msg, max_length);
                            socket.async_read_some(buf, f);
                        });
                };

        socket.async_read_some(asio::buffer(msg, max_length), f);
        io_service.run();
    }
}
```

Asio: Echo Server

```
int main() {
    asio::io_service io_service;
    asio_tcp::endpoint endpoint(asio_tcp::v4(), 2000);
    asio_tcp::acceptor acceptor(io_service, endpoint);

    for (;;) {
        asio_tcp::socket socket(io_service); acceptor.accept(socket);

        std::size_t const max_length = 1024;
        char msg[max_length];

        std::function<void(error_code const&, std::size_t)>
            f = [&](error_code const& ec, std::size_t bytes)
                {
                    asio::async_write(socket, asio::buffer(msg, bytes),
                        [&](error_code const& ec, std::size_t)
                        {
                            auto buf = asio::buffer(msg, max_length);
                            socket.async_read_some(buf, f);
                        });
                };

        socket.async_read_some(asio::buffer(msg, max_length), f);
        io_service.run();
    }
}
```

Asio: Echo Server

```
int main() {
    asio::io_service io_service;
    asio_tcp::endpoint endpoint(asio_tcp::v4(), 2000);
    asio_tcp::acceptor acceptor(io_service, endpoint);

    for (;;) {
        asio_tcp::socket socket(io_service); acceptor.accept(socket);

        std::size_t const max_length = 1024;
        char msg[max_length];

        std::function<void(error_code const&, std::size_t)>
            f = [&](error_code const& ec, std::size_t bytes)
                {
                    asio::async_write(socket, asio::buffer(msg, bytes),
                        [&](error_code const& ec, std::size_t)
                        {
                            auto buf = asio::buffer(msg, max_length);
                            socket.async_read_some(buf, f);
                        });
                };

        socket.async_read_some(asio::buffer(msg, max_length), f);
        io_service.run();
    }
}
```

Asio: Echo Server

```
int main() {
    asio::io_service io_service;
    asio_tcp::endpoint endpoint(asio_tcp::v4(), 2000);
    asio_tcp::acceptor acceptor(io_service, endpoint);

    for (;;) {
        asio_tcp::socket socket(io_service); acceptor.accept(socket);

        std::size_t const max_length = 1024;
        char msg[max_length];

        std::function<void(error_code const&, std::size_t)>
            f = [&](error_code const& ec, std::size_t bytes)
                {
                    asio::async_write(socket, asio::buffer(msg, bytes),
                        [&](error_code const& ec, std::size_t)
                        {
                            auto buf = asio::buffer(msg, max_length);
                            socket.async_read_some(buf, f);
                        });
                };

        socket.async_read_some(asio::buffer(msg, max_length), f);
        io_service.run();
    }
}
```

Asio: Echo Server

```
int main() {
    asio::io_service io_service;
    asio_tcp::endpoint endpoint(asio_tcp::v4(), 2000);
    asio_tcp::acceptor acceptor(io_service, endpoint);

    for (;;) {
        asio_tcp::socket socket(io_service); acceptor.accept(socket);

        std::size_t const max_length = 1024;
        char msg[max_length];

        std::function<void(error_code const&, std::size_t)>
            f = [&](error_code const& ec, std::size_t bytes)
                {
                    asio::async_write(socket, asio::buffer(msg, bytes),
                        [&](error_code const& ec, std::size_t)
                        {
                            auto buf = asio::buffer(msg, max_length);
                            socket.async_read_some(buf, f);
                        });
                };

        socket.async_read_some(asio::buffer(msg, max_length), f);
        io_service.run();
    }
}
```

Asio: Buffers

- **Buffer**: a contiguous region of memory.
  - Represented as a tuple: $\langle address, size \rangle$.
- Scatter-gather operations
  - Scatter-read receives into multiple buffers.
  - Gather-write transmits multiple buffers.
- Asio has two (basic) buffer types:
  - `const_buffer`: usable for writes.
  - `mutable_buffer`: usable for reads and writes, convertible to `const_buffer`.

Asio: Streams

- Stream-oriented I/O objects.
  - Data is a continuous byte sequence.
    - No message boundaries.
  - Reads and writes may transfer fewer bytes than requested (**short reads** and **short writes**).
- Models:
  - **`SyncReadStream`**: sync reads via `read_some()`.
  - **`AsyncReadStream`**: async reads via `async_read_some()`.
  - **`SyncWriteStream`**: sync writes via `write_some()`.
  - **`AsyncWriteStream`**: async writes via `async_write_some()`.
- `basic_stream_socket<Protocol, Service>` and `basic_serial_port` both fulfill all four models, for example.

Asio: IPC Shared Memory

- Interprocess backend for Asio
  - Built on top of shared memory IPC, uses `boost::interprocess::message_queue`.
- `hpx::parcelset::shmem::acceptor`
  - Fulfills `SocketAcceptorService`.
  - Endpoint is a string, the name of the message queue.
- `hpx::parcelset::shmem::data_window`
  - Fulfills `AsyncReadStream`, `AsyncWriteStream`, `SyncReadStream` and `SyncWriteStream`.

Asio: Infiniband

- Infiniband (IB) – high performance, low latency network interconnect.
  - November 2012 Top500 list: 37.8% Gigabit Ethernet, 44.8% Infiniband.
  - Wildly popular on clusters, costs have dropped significantly in recent years.
  - Major manufacturers: Mellanox, QLogic.
  - Switched fabric architecture.

Asio: Infiniband

- Comparison to Ethernet (data from HPC Advisory Council, Mellanox):

Interconnect	Actual Data Rate	Latency (end-to-end)	Encoding
GigE	1 Gbits/s	50 µs	8b/10b
10 GigE	10 Gbits/s	12 µs	8b/10b, 64b/66b
QDR Infiniband	32 Gbits/s	1.2 µs	8b/10b
FDR Infiniband	54.54 Gbits/s	0.7 µs	64b/66b

Asio: Infiniband

- Infiniband backend for Asio
  - Uses the Open Fabrics Verbs API for Remote Direct Memory Access (RDMA).
- `hpx::parcelset::ibverbs::acceptor`
  - Fulfills `SocketAcceptorService`.
  - Uses TCP/IP to bootstrap the IB connection.
- `hpx::parcelset::ibverbs::client_context`, `hpx::parcelset::ibverbs::server_context`
  - Fulfills `AsyncReadStream`, `AsyncWriteStream`, `SyncReadStream` and `SyncWriteStream`.

Asio: GPGPUs

- Intel Xeon Phi co-processors have an IB emulation layer that runs over PCIe. Our Asio IB layer allows HPX to interact with these co-processors through standard HPX facilities.
- Possible future backend for managing other classes of co-processors.
  - Most accelerators won't be running Asio (the Xeon Phi is unique, it's x86, and runs Linux on the co-processor).
- E.g. Asio-based library for interacting asynchronously with GPGPUs (using OpenCL, CUDA as a backend).

BOOST.SERIALIZATION

Serialization

- Boost.Serialization: A library for transforming a C++ object into a sequence of bytes (an **archive**) which can later be reconstructed into an equivalent object.
- We use the term **serialization** to refer to this transform, and **deserialization** to refer to the reconstruction.

Serialization

- Serialization features:
  - Support for serialization of **derived classes** from **base class pointers**.
  - Data structure versioning.
  - Built-in support for STL containers and some Boost data structures.
    - `boost::shared_ptr<T>`
  - Binary, text and XML archives.
  - Object tracking.
  - Intrusive and non-intrusive interfaces for defining serialization/deserialization routines.

Serialization: Archives

- **SavingArchive** (SA) model:
  - `sa << x`, `sa & x`
    - Add the sequence of bytes representing `x` to the archive `sa`.
- **LoadingArchive** (LA) model:
  - `la >> x`, `la & x`
    - Assign a value read from `la` to `x`.
- There are other type requirements for both types, but they are not important to our understanding of these concepts.

Serialization: Serializable

- **Serializable** model: an object that fulfills this model can be serialized and deserialized.
  - Primitives, STL containers, etc are Serializable.
- A type T is Serializable if:
  - It has a member function of the form

```
template <typename Archive>
void serialize(Archive& ar, const unsigned version);
```

  - Or, there is a global function of the form

```
template <typename Archive>
void serialize(Archive& ar, T& t, const unsigned version);
```

Serialization: Coordinates

```
struct coordinate
{
    boost::uint64_t x;
    boost::uint64_t y;

    template <typename Archive>
    void serialize(Archive& ar, const unsigned version)
    {
        ar & x;
        ar & y;
    }
};
```

Serialization: Coordinates

```
int main() {
    std::stringstream ss;

    {
        boost::archive::text_oarchive sa(ss);

        coordinate c{17, 42};
        sa << c;
    }

    std::cout << ss.str();

    {
        boost::archive::text_iarchive la(ss);

        coordinate c;
        la >> c;

        std::cout << "{" << c.x << ", " << c.y << "}\n";
    }
}
```

Serialization: Coordinates

```
int main() {
    std::stringstream ss;

    {
        boost::archive::text_oarchive sa(ss);

        coordinate c{17, 42};
        sa << c;
    }

    std::cout << ss.str();

    {
        boost::archive::text_iarchive la(ss);

        coordinate c;
        la >> c;

        std::cout << "{" << c.x << ", " << c.y << "}\n";
    }
}
```

Serialization: Coordinates

```
int main() {
    std::stringstream ss;

    {
        boost::archive::text_oarchive sa(ss);

        coordinate c{17, 42};
        sa << c;
    }

    std::cout << ss.str();

    {
        boost::archive::text_iarchive la(ss);

        coordinate c;
        la >> c;

        std::cout << "{" << c.x << ", " << c.y << "}\n";
    }
}
```

Serialization: Coordinates

- `./coordinates_text`:

```
22 serialization::archive 10 0 0 17 42
{17, 42}
```

- `22 serialization::archive` – signature
- `10` – archive version
- `0 0` – class id, class version
- `17 42` – representation of the object

Serialization: Derived Classes

- Serialization of derived classes through base class pointers:
  - Requires **explicit registration** of the derived class with Serialization's class registry. This is necessary because:
    - The `serialize()` code for the derived class may never be instantiated.
    - An identifier needs to be associated with the derived object in the registry. Serialization doesn't use `typeid()` because it's not portable.

Serialization: Derived Classes

```
struct A
{
    virtual ~task() {}

    template <typename Archive>
    void serialize(Archive&, const unsigned) {}
};

struct B : A
{
    template <typename Archive>
    void serialize(Archive& ar, const unsigned)
    {
        ar & boost::serialization::base_object<B>(*this);
    }
};
BOOST_CLASS_EXPORT_GUID(B, "B")
```

Serialization: Derived Classes

```
struct A
{
    virtual ~task() {}

    template <typename Archive>
    void serialize(Archive&, const unsigned) {}
};

struct B : A
{
    template <typename Archive>
    void serialize(Archive& ar, const unsigned)
    {
        ar & boost::serialization::base_object<B>(*this);
    }
};
BOOST_CLASS_EXPORT_GUID(B, "B")
```

Serialization: Derived Classes

```
struct A
{
    virtual ~task() {}

    template <typename Archive>
    void serialize(Archive&, const unsigned) {}
};

struct B : A
{
    template <typename Archive>
    void serialize(Archive& ar, const unsigned)
    {
        ar & boost::serialization::base_object<B>(*this);
    }
};
BOOST_CLASS_EXPORT_GUID(B, "B")
```

Serialization: Alternatives

- protobuf
  - Uses a custom definition file (`.proto` file), which statically specifies a protocol.
    - `.proto` files are compiled with the protocol buffer compiler, `protoc`.
  - No non-intrusive support.
  - Doesn’t interoperate with STL, Boost data structures.
- S11n
  - Supports multiple formats (XML, MySQL, etc).
  - Non-intrusive support.
  - Built-in support for STL.
- Neither supports derived class serialization through base class pointers.

OBJECT TRANSMISSION

Object Transmission

- **Object Transmission**: serializing an object and sending it over a network, to be deserialized at another endpoint.
- Used in Remote Procedure Call (RPC)/Remote Method Invocation (RMI) and **active messaging** implementations.
- Simplifies network communication (by abstracting away encoding/decoding).

OT: Coordinates Revisited

```
struct coordinate
{
    boost::uint64_t x;
    boost::uint64_t y;

    template <typename Archive>
    void serialize(Archive& ar, const unsigned version)
    {
        ar & x;
        ar & y;
    }
};
```

OT: Coordinates Revisited

```
int main()
{
    asio_tcp::iostream s("localhost", "2000");

    {
        boost::archive::binary_oarchive sa(s);

        coordinate c{17, 42};
        sa << c;
    }
}
```

OT: Coordinates Revisited

```
int main()
{
    asio_tcp::iostream s("localhost", "2000");

    {
        boost::archive::binary_oarchive sa(s);

        coordinate c{17, 42};
        sa << c;
    }
}
```

OT: Coordinates Revisited

```
int main()
{
    asio::io_service io_service;

    asio_tcp::endpoint endpoint(asio_tcp::v4(), 2000);
    asio_tcp::acceptor acceptor(io_service, endpoint);

    asio_tcp::iostream s;

    acceptor.accept(*s.rdbuf());

    {
        boost::archive::binary_iarchive la(s);

        coordinate c;
        la >> c;

        std::cout << "{" << c.x << ", " << c.y << "}\n";
    }
}
```

OT: Coordinates Revisited

```
int main()
{
    asio::io_service io_service;

    asio_tcp::endpoint endpoint(asio_tcp::v4(), 2000);
    asio_tcp::acceptor acceptor(io_service, endpoint);

    asio_tcp::iostream s;

    acceptor.accept(*s.rdbuf());

    {
        boost::archive::binary_iarchive la(s);

        coordinate c;
        la >> c;

        std::cout << "{" << c.x << ", " << c.y << "}\n";
    }
}
```

ACTIVE MESSAGING

Active Messaging

- An active message (AM) is a message that is capable of performing computations.
- We will build a simple active messaging **runtime system** which uses two threads (per node):
  - An I/O thread running an `io_service` object's event processing loop.
    - We'll use the program's main thread for this.
  - An execution thread invoking the active messages.
- This system will use TCP/IP for communication.
- Code can be found here:

`git@github.com:STEllAR-GROUP/cppnow2013_ot.git`

AM: Thread Responsibilities

- Division of labor between our two threads:
  - I/O thread
    - Waits for I/O events.
    - Invokes completion handlers for reads, writes and accepts.
    - Responsible for joining the execution thread.
  - Execution thread
    - Performs maintenance computations.
      - Serialization, deserialization.
    - Executes active messages (e.g. user code).
    - Responsible for breaking the I/O thread out of the event processing loop when it is time to shut down.

AM: Data Structures

- Data structures we'll need:
  - `action`: base class for our active messages.
  - `runtime`: manages and implements runtime services.
    - Manages the I/O and execution threads.
    - Manages all connections.
    - Provides APIs for:
      - Initiating asynchronous writes to connected nodes.
      - Opening connections to new nodes.
      - Scheduling local tasks on the execution thread.
      - Starting and shutting down the runtime.
  - `connection`: representation of a connection to another endpoint.
    - Holds a socket, buffers.

AM: Action

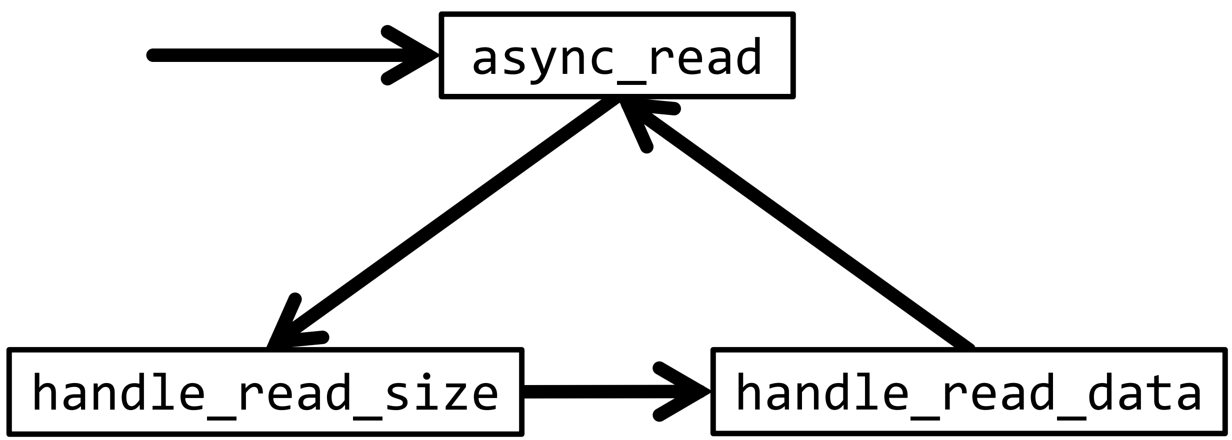
```
struct action
{
    virtual ~action() {}

    virtual void operator()(runtime& rt) = 0;

    virtual action* clone() const = 0;

    template <typename Archive>
    void serialize(Archive& ar, const unsigned) {}
};
```

AM: Read Interface

```
struct connection : std::enable_shared_from_this<connection>
{
  private:
    runtime& runtime_;

    asio_tcp::socket socket_;

    boost::uint64_t in_size_;
    std::vector<char>* in_buffer_;

  public:
    // ...

    /// Asynchronously read a parcel from the socket.
    void async_read();

    /// Handler for the parcel size.
    void handle_read_size(error_code const& error);

    /// Handler for the data.
    void handle_read_data(error_code const& error);

    // ... write operations ...
};
```

AM: System Design

- Read control flow:



AM: Read

```
void connection::async_read()
{
    BOOST_ASSERT(in_buffer_ == 0);
    in_size_ = 0;
    in_buffer_ = new std::vector<char>();

    asio::async_read(socket_,
        asio::buffer(&in_size_, sizeof(in_size_)),
            boost::bind(&connection::handle_read_size
                      , shared_from_this()
                      , asio::placeholders::error));
}
```

AM: Read

```
void connection::async_read()
{
    BOOST_ASSERT(in_buffer_ == 0);
    in_size_ = 0;
    in_buffer_ = new std::vector<char>();

    asio::async_read(socket_,
        asio::buffer(&in_size_, sizeof(in_size_)),
            boost::bind(&connection::handle_read_size
                      , shared_from_this()
                      , asio::placeholders::error));
}
```

AM: Read

```
void connection::async_read()
{
    BOOST_ASSERT(in_buffer_ == 0);
    in_size_ = 0;
    in_buffer_ = new std::vector<char>();

    asio::async_read(socket_,
        asio::buffer(&in_size_, sizeof(in_size_)),
            boost::bind(&connection::handle_read_size
                     , shared_from_this()
                     , asio::placeholders::error));
}
```

AM: Read

```
void connection::async_read()
{
    BOOST_ASSERT(in_buffer_ == 0);
    in_size_ = 0;
    in_buffer_ = new std::vector<char>();

    asio::async_read(socket_,
        asio::buffer(&in_size_, sizeof(in_size_)),
            boost::bind(&connection::handle_read_size
                      , shared_from_this()
                      , asio::placeholders::error));
}
```

AM: Read

```
void connection::async_read()
{
    BOOST_ASSERT(in_buffer_ == 0);
    in_size_ = 0;
    in_buffer_ = new std::vector<char>();

    asio::async_read(socket_,
        asio::buffer(&in_size_, sizeof(in_size_)),
            boost::bind(&connection::handle_read_size
                     , shared_from_this()
                     , asio::placeholders::error));
}
```

AM: Data Size Handler

```
void connection::handle_read_size(error_code const& error)
{
    if (error) return;

    BOOST_ASSERT(in_buffer_);

    (*in_buffer_).resize(in_size_);

    asio::async_read(socket_,
        asio::buffer(*in_buffer_),
            boost::bind(&connection::handle_read_data
                     , shared_from_this()
                     , asio::placeholders::error));
}
```

AM: Data Size Handler

```
void connection::handle_read_size(error_code const& error)
{
    if (error) return;

    BOOST_ASSERT(in_buffer_);

    (*in_buffer_).resize(in_size_);

    asio::async_read(socket_,
        asio::buffer(*in_buffer_),
            boost::bind(&connection::handle_read_data
                      , shared_from_this()
                      , asio::placeholders::error));
}
```

AM: Data Size Handler

```
void connection::handle_read_size(error_code const& error)
{
    if (error) return;

    BOOST_ASSERT(in_buffer_);

    (*in_buffer_).resize(in_size_);

    asio::async_read(socket_,
        asio::buffer(*in_buffer_),
            boost::bind(&connection::handle_read_data
                      , shared_from_this()
                      , asio::placeholders::error));
}
```

AM: Data Size Handler

```
void connection::handle_read_size(error_code const& error)
{
    if (error) return;

    BOOST_ASSERT(in_buffer_);

    (*in_buffer_).resize(in_size_);

    asio::async_read(socket_,
        asio::buffer(*in_buffer_),
            boost::bind(&connection::handle_read_data
                      , shared_from_this()
                      , asio::placeholders::error));
}
```

AM: Data Handler

```
void connection::handle_read_data(error_code const& error)
{
    if (error) return;

    std::vector<char>* raw_msg = 0;
    std::swap(in_buffer_, raw_msg);

    runtime_.get_parcel_queue().push(raw_msg);

    // Start the next read.
    async_read();
}
```

AM: Data Handler

```
void connection::handle_read_data(error_code const& error)
{
    if (error) return;

    std::vector<char>* raw_msg = 0;
    std::swap(in_buffer_, raw_msg);

    runtime_.get_parcel_queue().push(raw_msg);

    // Start the next read.
    async_read();
}
```

AM: Data Handler

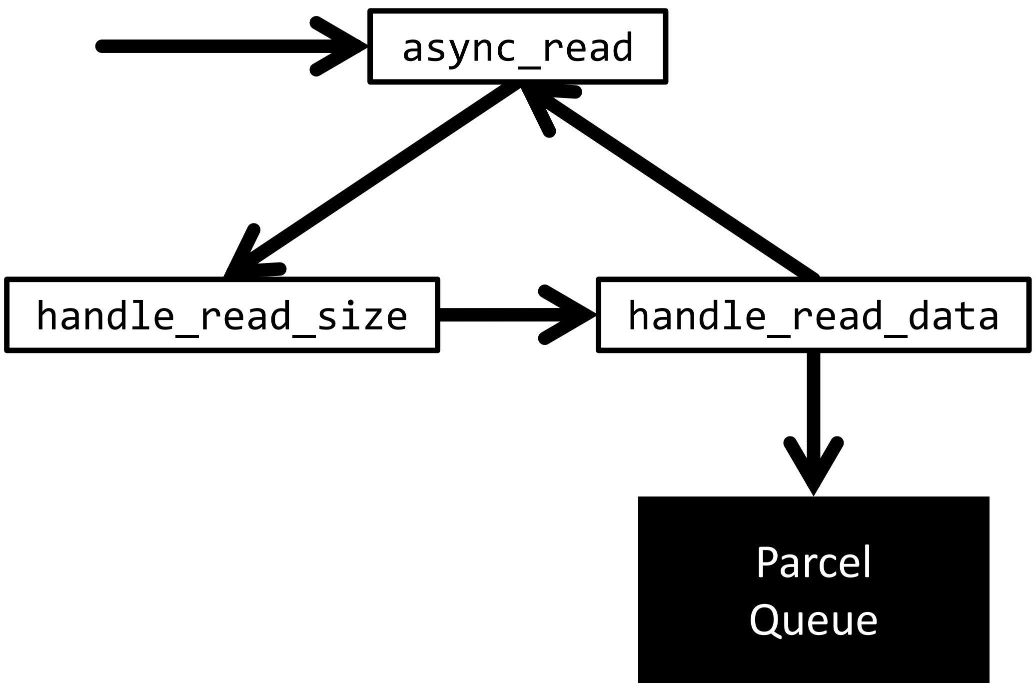
```
void connection::handle_read_data(error_code const& error)
{
    if (error) return;

    std::vector<char>* raw_msg = 0;
    std::swap(in_buffer_, raw_msg);

    runtime_.get_parcel_queue().push(raw_msg);

    // Start the next read.
    async_read();
}
```

AM: Data Handler

```
void connection::handle_read_data(error_code const& error)
{
    if (error) return;

    std::vector<char>* raw_msg = 0;
    std::swap(in_buffer_, raw_msg);

    runtime_.get_parcel_queue().push(raw_msg);

    // Start the next read.
    async_read();
}
```

AM: System Design

- Read control flow:



AM: Runtime

```
struct runtime
{
    typedef std::map<asio_tcp::endpoint, std::shared_ptr<connection> > connection_map;

  private:
    asio::io_service io_service_;
    asio_tcp::acceptor acceptor_;

    connection_map connections_;

    std::thread exec_thread_;

    boost::lockfree::queue<std::vector<char>*> parcel_queue_;
    boost::lockfree::queue<std::function<void(runtime&)>*> local_queue_;

    std::atomic<bool> stop_flag_;

    // ... other members that we won't talk about ...
  public:
    // ... member functions ...
};
```

AM: Runtime

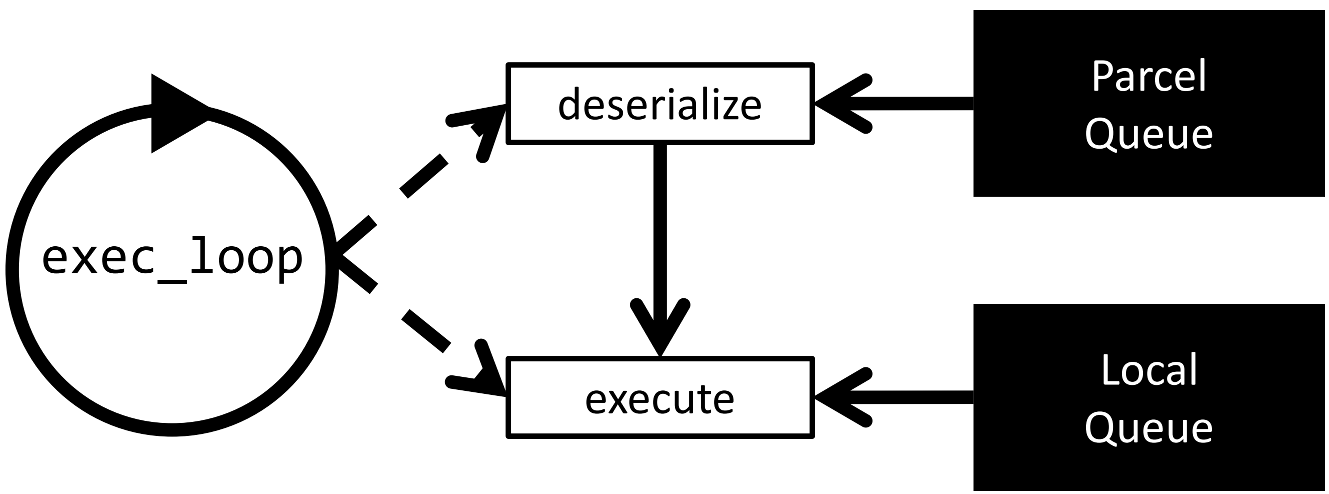
```
struct runtime
{
    /// ... other member functions ...

    void start(); /// Launch the execution thread. Then, start accepting connections.

    void stop(); /// Stop the I/O service and execution thread.

    void run(); /// Accepts connections and parcels until stop() is called.

    /// ... accept, connect related functions ...
  private:
    /// Execute actions until stop() is called.
    void exec_loop();

    /// Serializes a action object into a parcel.
    std::vector<char>* serialize_parcel(action const& act);

    /// Deserializes a parcel into a action object.
    action* deserialize_parcel(std::vector<char>& raw_msg);
};
```

AM: Runtime

```
struct runtime
{
    /// ... other member functions ...

    void start(); /// Launch the execution thread. Then, start accepting connections.

    void stop(); /// Stop the I/O service and execution thread.

    void run(); /// Accepts connections and parcels until stop() is called.

    /// ... accept, connect related functions ...
  private:
    /// Execute actions until stop() is called.
    void exec_loop();

    /// Serializes a action object into a parcel.
    std::vector<char>* serialize_parcel(action const& act);

    /// Deserializes a parcel into a action object.
    action* deserialize_parcel(std::vector<char>& raw_msg);
};
```

AM: System Design

- Execution loop control flow:



AM: Execution Loop

```
void runtime::exec_loop() {
    while (!stop_flag_.load()) {
        // First, we look for pending actions to execute.
        std::function<void(runtime&)>* act_ptr = 0;
        if (local_queue_.pop(act_ptr)) {
            BOOST_ASSERT(act_ptr);
            boost::scoped_ptr<std::function<void(runtime&)> > act(act_ptr);

            (*act)(*this);
        }

        // If we can't find any work, we try to find a parcel to deserialize and execute.
        std::vector<char>* raw_msg_ptr = 0;
        if (parcel_queue_.pop(raw_msg_ptr)) {
            BOOST_ASSERT(raw_msg_ptr);
            boost::scoped_ptr<std::vector<char> > raw_msg(raw_msg_ptr);
            boost::scoped_ptr<action> act(deserialize_parcel(*raw_msg));

            (*act)(*this);
        }
    }
}
```

AM: Execution Loop

```
void runtime::exec_loop() {
    while (!stop_flag_.load()) {
        // First, we look for pending actions to execute.
        std::function<void(runtime&)>* act_ptr = 0;
        if (local_queue_.pop(act_ptr)) {
            BOOST_ASSERT(act_ptr);
            boost::scoped_ptr<std::function<void(runtime&)> > act(act_ptr);

            (*act)(*this);
        }

        // If we can't find any work, we try to find a parcel to deserialize and execute.
        std::vector<char>* raw_msg_ptr = 0;
        if (parcel_queue_.pop(raw_msg_ptr)) {
            BOOST_ASSERT(raw_msg_ptr);
            boost::scoped_ptr<std::vector<char> > raw_msg(raw_msg_ptr);
            boost::scoped_ptr<action> act(deserialize_parcel(*raw_msg));

            (*act)(*this);
        }
    }
}
```

AM: Execution Loop

```
void runtime::exec_loop() {
    while (!stop_flag_.load()) {
        // First, we look for pending actions to execute.
        std::function<void(runtime&)>* act_ptr = 0;
        if (local_queue_.pop(act_ptr)) {
            BOOST_ASSERT(act_ptr);
            boost::scoped_ptr<std::function<void(runtime&)> > act(act_ptr);

            (*act)(*this);
        }

        // If we can't find any work, we try to find a parcel to deserialize and execute.
        std::vector<char>* raw_msg_ptr = 0;
        if (parcel_queue_.pop(raw_msg_ptr)) {
            BOOST_ASSERT(raw_msg_ptr);
            boost::scoped_ptr<std::vector<char> > raw_msg(raw_msg_ptr);
            boost::scoped_ptr<action> act(deserialize_parcel(*raw_msg));

            (*act)(*this);
        }
    }
}
```

AM: Execution Loop

```
void runtime::exec_loop() {
    while (!stop_flag_.load()) {
        // First, we look for pending actions to execute.
        std::function<void(runtime&)>* act_ptr = 0;
        if (local_queue_.pop(act_ptr)) {
            BOOST_ASSERT(act_ptr);
            boost::scoped_ptr<std::function<void(runtime&)> > act(act_ptr);

            (*act)(*this);
        }

        // If we can't find any work, we try to find a parcel to deserialize and execute.
        std::vector<char>* raw_msg_ptr = 0;
        if (parcel_queue_.pop(raw_msg_ptr)) {
            BOOST_ASSERT(raw_msg_ptr);
            boost::scoped_ptr<std::vector<char> > raw_msg(raw_msg_ptr);
            boost::scoped_ptr<action> act(deserialize_parcel(*raw_msg));

            (*act)(*this);
        }
    }
}
```

AM: Deserialization

```
action* runtime::deserialize_parcel(std::vector<char>& raw_msg)
{
    typedef container_device<std::vector<char> > io_device_type;
    boost::iostreams::stream<io_device_type> io(raw_msg);

    action* act_ptr = 0;

    {
        boost::archive::binary_iarchive archive(io);
        archive & act_ptr;
    }

    BOOST_ASSERT(act_ptr);

    return act_ptr;
}
```

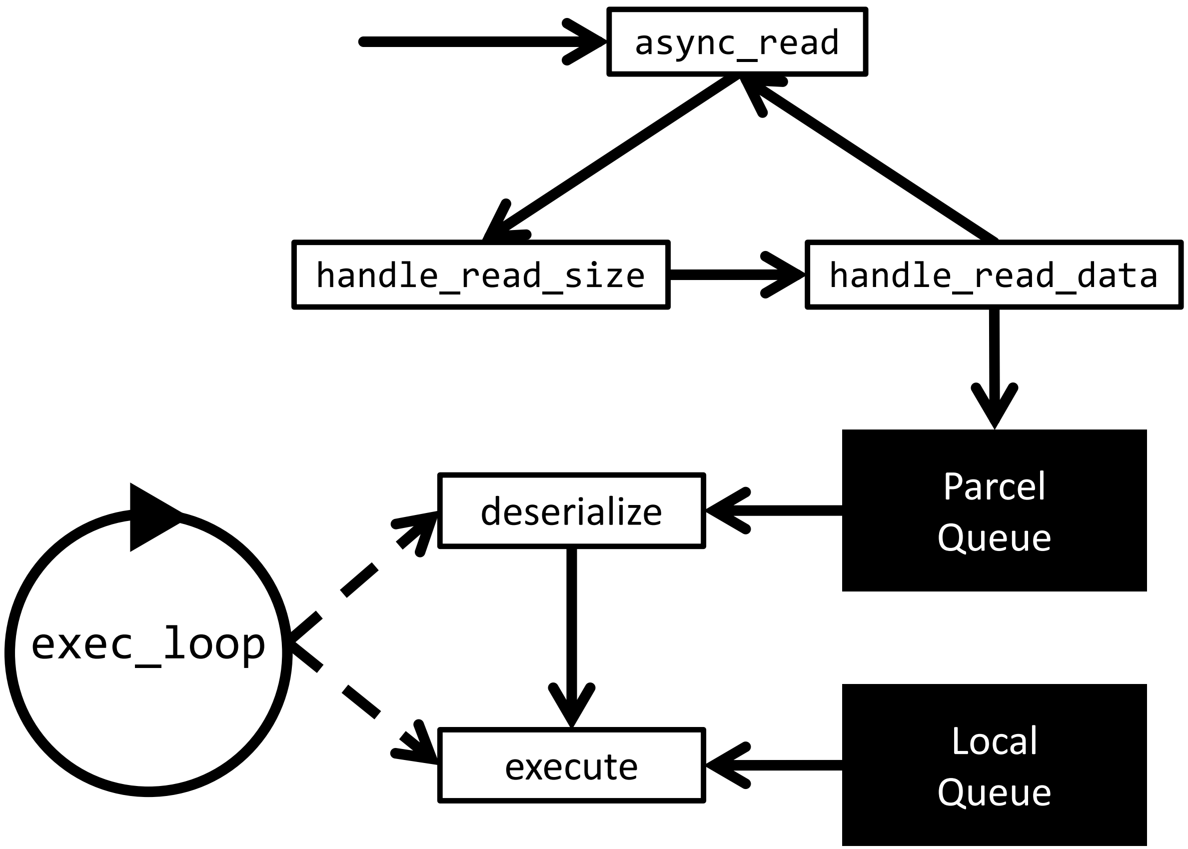
AM: Deserialization

```
action* runtime::deserialize_parcel(std::vector<char>& raw_msg)
{
    typedef container_device<std::vector<char> > io_device_type;
    boost::iostreams::stream<io_device_type> io(raw_msg);

    action* act_ptr = 0;

    {
        boost::archive::binary_iarchive archive(io);
        archive & act_ptr;
    }

    BOOST_ASSERT(act_ptr);

    return act_ptr;
}
```

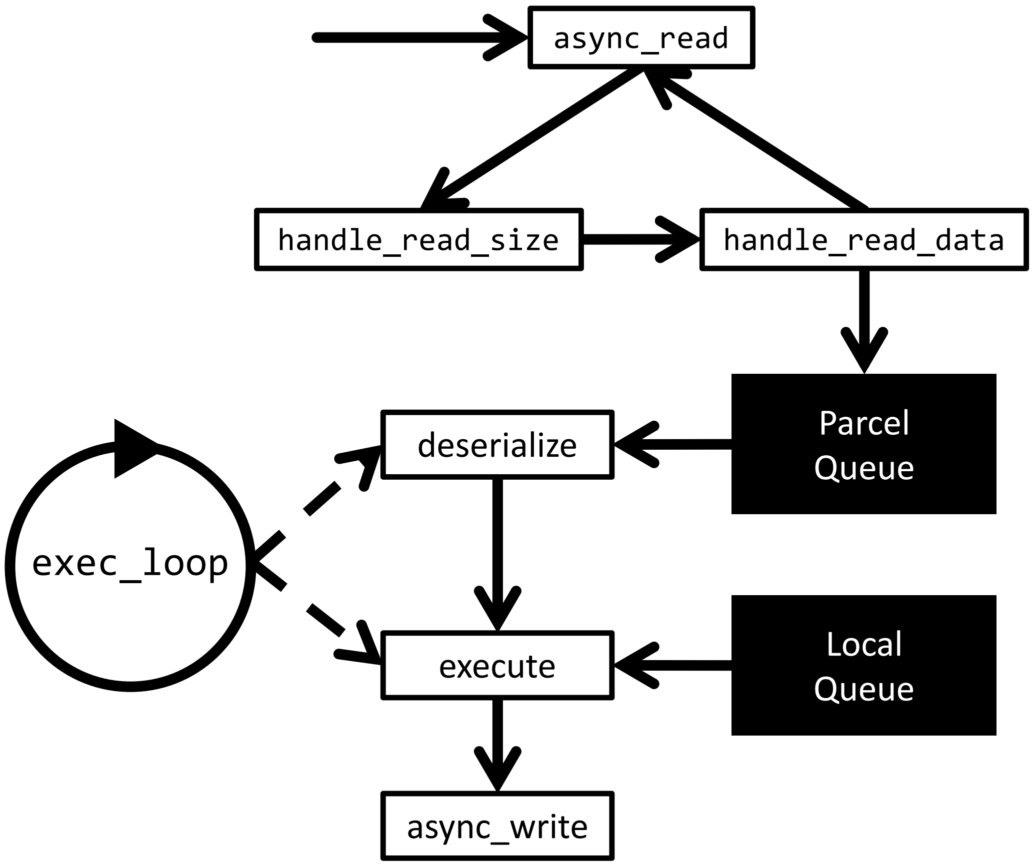
AM: Deserialization

```
action* runtime::deserialize_parcel(std::vector<char>& raw_msg)
{
    typedef container_device<std::vector<char> > io_device_type;
    boost::iostreams::stream<io_device_type> io(raw_msg);

    action* act_ptr = 0;

    {
        boost::archive::binary_iarchive archive(io);
        archive & act_ptr;
    }

    BOOST_ASSERT(act_ptr);

    return act_ptr;
}
```

AM: System Design



AM: System Design



AM: Write Interface

```
struct connection : std::enable_shared_from_this<connection>
{
    /// ...

    /// Asynchronously write a action to the socket.
    void async_write(action const& act, std::function<void(error_code const&)> handler);

    /// This function is scheduled in the local_queue by async_write. It does
    /// the actual work of serializing the action.
    void async_write_worker(
        std::shared_ptr<action> act
      , std::function<void(error_code const&)> handler
        );

    /// Write handler.
    void handle_write(
        error_code const& error
      , std::shared_ptr<boost::uint64_t> out_size
      , std::shared_ptr<std::vector<char> > out_buffer
      , std::function<void(error_code const&)> handler
        );
};
```

AM: Write

```
void connection::async_write(
    action const& act
  , std::function<void(error_code const&)> handler
    )
{
    std::shared_ptr<action> act_ptr(act.clone());

    runtime_.get_local_queue().push(
        new std::function<void(runtime&)>(
             boost::bind(&connection::async_write_worker
                       , shared_from_this()
                       , act_ptr
                       , handler)));
}
```

AM: Write Worker

```cpp
void connection::async_write_worker(
    std::shared_ptr<action> act
  , std::function<void(error_code const&)> handler
    )
{
    std::shared_ptr<std::vector<char> >
        out_buffer(runtime_.serialize_parcel(*act));

    std::shared_ptr<boost::uint64_t>
        out_size(new boost::uint64_t(out_buffer->size()));

    std::vector<boost::asio::const_buffer> buffers;
    buffers.push_back(boost::asio::buffer(&*out_size, sizeof(*out_size)));
    buffers.push_back(boost::asio::buffer(*out_buffer));

    boost::asio::async_write(socket_, buffers,
        boost::bind(&connection::handle_write
                  , shared_from_this()
                  , boost::asio::placeholders::error
                  , out_size
                  , out_buffer
                  , handler));
}
```

AM: Write Worker

```
void connection::async_write_worker(
    std::shared_ptr<action> act
  , std::function<void(error_code const&)> handler
    )
{
    std::shared_ptr<std::vector<char> >
        out_buffer(runtime_.serialize_parcel(*act));

    std::shared_ptr<boost::uint64_t>
        out_size(new boost::uint64_t(out_buffer->size()));

    std::vector<boost::asio::const_buffer> buffers;
    buffers.push_back(boost::asio::buffer(&*out_size, sizeof(*out_size)));
    buffers.push_back(boost::asio::buffer(*out_buffer));

    boost::asio::async_write(socket_, buffers,
        boost::bind(&connection::handle_write
                  , shared_from_this()
                  , boost::asio::placeholders::error
                  , out_size
                  , out_buffer
                  , handler));
}
```

AM: Write Worker

```
void connection::async_write_worker(
    std::shared_ptr<action> act
  , std::function<void(error_code const&)> handler
    )
{
    std::shared_ptr<std::vector<char> >
        out_buffer(runtime_.serialize_parcel(*act));

    std::shared_ptr<boost::uint64_t>
        out_size(new boost::uint64_t(out_buffer->size()));

    std::vector<boost::asio::const_buffer> buffers;
    buffers.push_back(boost::asio::buffer(&*out_size, sizeof(*out_size)));
    buffers.push_back(boost::asio::buffer(*out_buffer));

    boost::asio::async_write(socket_, buffers,
        boost::bind(&connection::handle_write
                  , shared_from_this()
                  , boost::asio::placeholders::error
                  , out_size
                  , out_buffer
                  , handler));
}
```

AM: Write Worker

```
void connection::async_write_worker(
    std::shared_ptr<action> act
  , std::function<void(error_code const&)> handler
    )
{
    std::shared_ptr<std::vector<char> >
        out_buffer(runtime_.serialize_parcel(*act));

    std::shared_ptr<boost::uint64_t>
        out_size(new boost::uint64_t(out_buffer->size()));

    std::vector<boost::asio::const_buffer> buffers;
    buffers.push_back(boost::asio::buffer(&*out_size, sizeof(*out_size)));
    buffers.push_back(boost::asio::buffer(*out_buffer));

    boost::asio::async_write(socket_, buffers,
        boost::bind(&connection::handle_write
                  , shared_from_this()
                  , boost::asio::placeholders::error
                  , out_size
                  , out_buffer
                  , handler));
}
```

AM: Serialization

```
std::vector<char>* runtime::serialize_parcel(action const& act)
{
    std::vector<char>* raw_msg_ptr = new std::vector<char>();

    typedef container_device<std::vector<char> > io_device_type;
    boost::iostreams::stream<io_device_type> io(*raw_msg_ptr);

    action const* act_ptr = &act;

    {
        boost::archive::binary_oarchive archive(io);
        archive & act_ptr;
    }

    return raw_msg_ptr;
}
```

AM: Connect

```cpp
std::shared_ptr<connection> runtime::connect(std::string host, std::string service)
{
    asio_tcp::resolver resolver(io_service_);
    asio_tcp::resolver::query query(asio_tcp::v4(), host, service);
    asio_tcp::resolver::iterator it = resolver.resolve(query), end;

    // Are we already connected to this node?
    for (asio_tcp::resolver::iterator i = it; i != end; ++i)
        if (connections_.count(*i) != 0) return connections_[*i];

    std::shared_ptr<connection> conn(new connection(*this));

    // Waits for up to 6.4 seconds (0.001 * 100 * 64) for the runtime to become available.
    for (boost::uint64_t i = 0; i < 64; ++i) {
        error_code ec;
        asio::connect(conn->get_socket(), it, ec);
        if (!ec) break;

        // Otherwise, we sleep and try again.
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    connections_[conn->get_remote_endpoint()] = conn;

    conn->async_read(); // Start reading.

    return conn;
}
```

AM: Connect

```
std::shared_ptr<connection> runtime::connect(std::string host, std::string service)
{
    asio_tcp::resolver resolver(io_service_);
    asio_tcp::resolver::query query(asio_tcp::v4(), host, service);
    asio_tcp::resolver::iterator it = resolver.resolve(query), end;

    // Are we already connected to this node?
    for (asio_tcp::resolver::iterator i = it; i != end; ++i)
        if (connections_.count(*i) != 0) return connections_[*i];

    std::shared_ptr<connection> conn(new connection(*this));

    // Waits for up to 6.4 seconds (0.001 * 100 * 64) for the runtime to become available.
    for (boost::uint64_t i = 0; i < 64; ++i) {
        error_code ec;
        asio::connect(conn->get_socket(), it, ec);
        if (!ec) break;

        // Otherwise, we sleep and try again.
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    connections_[conn->get_remote_endpoint()] = conn;

    conn->async_read(); // Start reading.

    return conn;
}
```

AM: Connect

```
std::shared_ptr<connection> runtime::connect(std::string host, std::string service)
{
    asio_tcp::resolver resolver(io_service_);
    asio_tcp::resolver::query query(asio_tcp::v4(), host, service);
    asio_tcp::resolver::iterator it = resolver.resolve(query), end;

    // Are we already connected to this node?
    for (asio_tcp::resolver::iterator i = it; i != end; ++i)
        if (connections_.count(*i) != 0) return connections_[*i];

    std::shared_ptr<connection> conn(new connection(*this));

    // Waits for up to 6.4 seconds (0.001 * 100 * 64) for the runtime to become available.
    for (boost::uint64_t i = 0; i < 64; ++i) {
        error_code ec;
        asio::connect(conn->get_socket(), it, ec);
        if (!ec) break;

        // Otherwise, we sleep and try again.
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    connections_[conn->get_remote_endpoint()] = conn;

    conn->async_read(); // Start reading.

    return conn;
}
```

AM: Connect

```
std::shared_ptr<connection> runtime::connect(std::string host, std::string service)
{
    asio_tcp::resolver resolver(io_service_);
    asio_tcp::resolver::query query(asio_tcp::v4(), host, service);
    asio_tcp::resolver::iterator it = resolver.resolve(query), end;

    // Are we already connected to this node?
    for (asio_tcp::resolver::iterator i = it; i != end; ++i)
        if (connections_.count(*i) != 0) return connections_[*i];

    std::shared_ptr<connection> conn(new connection(*this));

    // Waits for up to 6.4 seconds (0.001 * 100 * 64) for the runtime to become available.
    for (boost::uint64_t i = 0; i < 64; ++i) {
        error_code ec;
        asio::connect(conn->get_socket(), it, ec);
        if (!ec) break;

        // Otherwise, we sleep and try again.
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    connections_[conn->get_remote_endpoint()] = conn;

    conn->async_read(); // Start reading.

    return conn;
}
```

AM: Connect

```
std::shared_ptr<connection> runtime::connect(std::string host, std::string service)
{
    asio_tcp::resolver resolver(io_service_);
    asio_tcp::resolver::query query(asio_tcp::v4(), host, service);
    asio_tcp::resolver::iterator it = resolver.resolve(query), end;

    // Are we already connected to this node?
    for (asio_tcp::resolver::iterator i = it; i != end; ++i)
        if (connections_.count(*i) != 0) return connections_[*i];

    std::shared_ptr<connection> conn(new connection(*this));

    // Waits for up to 6.4 seconds (0.001 * 100 * 64) for the runtime to become available.
    for (boost::uint64_t i = 0; i < 64; ++i) {
        error_code ec;
        asio::connect(conn->get_socket(), it, ec);
        if (!ec) break;

        // Otherwise, we sleep and try again.
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    connections_[conn->get_remote_endpoint()] = conn;

    conn->async_read(); // Start reading.

    return conn;
}
```

AM: Hello World

- Now let's use this runtime.
- Hello world example:
  - Two or more nodes.
  - One node acts as the bootstrap node – everyone else initially connects to it.
  - Server sends an action to all other nodes that prints out "hello world", and then shuts down that node.
  - After all the actions have been sent, the server shuts down.
- Running the example:
  - In shell #1 (server):

```
./hello_world --port=9000
```

  - In shell #2 (client):

```
./hello_world --port=9001             \
              --remote-host=localhost \
              --remote-port=9000
```

AM: Hello World

- We need some mechanism to invoke code once all the nodes have connected.
- The runtime on the bootstrap node will handle this for us; we pass it a “main” function and a node count in its constructor. It will invoke the “main” function when the number of connected nodes is the same as the specified node count.

AM: Runtime Constructor

```
struct runtime
{
    // ...
  public:
    runtime(
        std::string service
      , std::function<void(runtime&)> f
      , boost::uint64_t wait_for = 1
        );

    // ...
};
```

AM: Hello World Action

```
struct hello_world_action : action
{
    void operator()(runtime& rt)
    {
        std::cout << "hello world\n";

        rt.stop(); // Stop this node.
    }

    action* clone() const
    {
        return new hello_world_action;
    }

    template <typename Archive>
    void serialize(Archive& ar, const unsigned int)
    {
        ar & boost::serialization::base_object<action>(*this);
    }
};

BOOST_CLASS_EXPORT_GUID(hello_world_action, "hello_world_action");
```

AM: Hello World Action

```
struct hello_world_action : action
{
    void operator()(runtime& rt)
    {
        std::cout << "hello world\n";

        rt.stop(); // Stop this node.
    }

    action* clone() const
    {
        return new hello_world_action;
    }

    template <typename Archive>
    void serialize(Archive& ar, const unsigned int)
    {
        ar & boost::serialization::base_object<action>(*this);
    }
};

BOOST_CLASS_EXPORT_GUID(hello_world_action, "hello_world_action");
```

AM: Hello World Main

```
void hello_world_main(runtime& rt)
{
    auto conns = rt.get_connections();

    std::shared_ptr<boost::uint64_t>
        count(new boost::uint64_t(conns.size()));

    for (auto node : conns)
        node.second->async_write(hello_world_action(),
            [count, &rt](error_code const& ec)
            {
                if (--(*count) == 0)
                    rt.stop();
            });
}
```

AM: Hello World Main

```
void hello_world_main(runtime& rt)
{
    auto conns = rt.get_connections();

    std::shared_ptr<boost::uint64_t>
        count(new boost::uint64_t(conns.size()));

    for (auto node : conns)
        node.second->async_write(hello_world_action(),
            [count, &rt](error_code const& ec)
            {
                if (--(*count) == 0)
                    rt.stop();
            });
}
```

AM: Hello World Main

```
void hello_world_main(runtime& rt)
{
    auto conns = rt.get_connections();

    std::shared_ptr<boost::uint64_t>
        count(new boost::uint64_t(conns.size()));

    for (auto node : conns)
        node.second->async_write(hello_world_action(),
            [count, &rt](error_code const& ec)
            {
                if (--(*count) == 0)
                    rt.stop();
            });
}
```

AM: Hello World Main

```
void hello_world_main(runtime& rt)
{
    auto conns = rt.get_connections();

    std::shared_ptr<boost::uint64_t>
        count(new boost::uint64_t(conns.size()));

    for (auto node : conns)
        node.second->async_write(hello_world_action(),
            [count, &rt](error_code const& ec)
            {
                if (--(*count) == 0)
                    rt.stop();
            });
}
```

AM: Hello World Main

```
void hello_world_main(runtime& rt)
{
    auto conns = rt.get_connections();

    std::shared_ptr<boost::uint64_t>
        count(new boost::uint64_t(conns.size()));

    for (auto node : conns)
        node.second->async_write(hello_world_action(),
            [count, &rt](error_code const& ec)
            {
                if (--(*count) == 0)
                    rt.stop();
            });
}
```

AM: Hello World Main

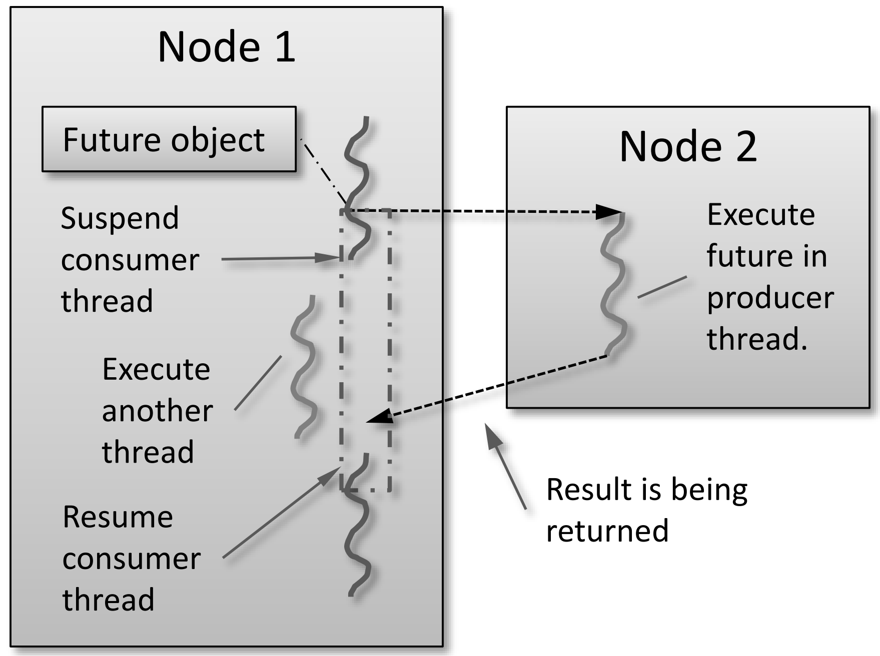
```
void hello_world_main(runtime& rt)
{
    auto conns = rt.get_connections();

    std::shared_ptr<boost::uint64_t>
        count(new boost::uint64_t(conns.size()));

    for (auto node : conns)
        node.second->async_write(hello_world_action(),
            [count, &rt](error_code const& ec)
            {
                if (--(*count) == 0)
                    rt.stop();
            });
}
```

AM: Hello World

- What would happen if `hello_world_main` waited for the `async_writes` to finish?
  - E.g. if we moved the call to `rt.stop()` from the lambda into `hello_world_main`, and we used a condition variable inside the lambda to wait for all the `async_writes` to finish, what behavior would we observe?
- Both processes would deadlock. Why?

AM: Future Steps

- What do we need to create remote futures (e.g. futures that represent remote computations)?
  - Such a future would work via two parcels.
    - One to start executing the remote action.
    - One to send the result back to the caller.
  - What else would we need?



AM: Future Steps

- What else do we need to create remote futures?
  - We need **user-level threading**.
    - We do not want to block the execution thread (or threads) at the OS-level. This means that when executing our actions, we need a way to suspend execution of the action and return control to the execution loop. Later, we must be able to resume the action's execution.
    - E.g. Boost.Context, Boost.Coroutines.
  - We need a **global address space** that can manage the lifetime of the future's state.
    - The back-parcel that returns the result needs a way to locate the future's state. We could just pass along the local memory address.
    - If we pass along the local memory address, we also have to pass along information identifying the calling node.
    - Giving the future's state a **globally unique identifier** provides a nice abstraction and allows us to **move the object to another node**; we simply update the identifying information that the global identifier references.
    - How do we ensure that the future state stays alive?
    - We need some sort of distributed smart pointer.

AM: Future Steps

- HPX has other needs for user-level threading.
  - Our execution model calls for extremely fine-grained threads.
    - Thousands of active threads per core.
  - Target thread lifetime: ~100µs.
  - Amortized thread overhead: <1ns on Sandy Bridge
    - E.g. time for allocation, creation, scheduling and cleanup of an empty thread.
- HPX has other needs for a global address space.
  - We don’t want to do everything with free function style actions, we want to have method actions.
  - To invoke method actions on objects, the objects need a global name that can be accessed from any connected node.
  - Again, by using a global address space, we can support the migration of objects without validating references to them.
- We call objects with a global name **components** in HPX.
- Actions are to functions what components are to classes.
- **Remotable**: A function or method is remotable if it can be called remotely (e.g. via an active message). A class is remotable if it has at least one remotable method.

HPX

HPX: Serializable Structures

- `hpx::tuple`, `hpx::fusion`
- `hpx::any`
- `hpx::function`
- `hpx::exception`
- `hpx::bind`
- `hpx::phoenix`

HPX: Recursion is Parallelism

- Recursion is Parallelism: tongue-in-cheek description of our mindset.
  - Traditionally, functional programming languages and paradigms have been avoided in HPC codes due to perceived performance concerns.
  - The HPX programming model uses functional programming to decompose algorithms into smaller sub-algorithms which have clearly defined dependencies (e.g. function arguments) and outputs (e.g. return values).
    - These sub-algorithms are easy to dynamically load balance, because they are fine-grained and have clearly defined dependencies.
  - Writing functional-style code is an easy way to parallelize our code.

HPX: 1D Wave Equation

- Evolving the following equation in time:

$$\frac{\partial^2 U}{\partial t^2} = c^2 \frac{\partial^2 U}{\partial x^2} \qquad \alpha = \frac{c\Delta t}{\Delta x} \leq 1$$

- c – Propagation speed of the wave.
- Central-difference scheme, periodic boundary conditions. Discretization:

$$\begin{aligned} U(t+dt,x) = {} & \alpha^2\left(U(t,x+dx) + U(t,x-dx)\right) \\ & + 2(1-\alpha^2)U(t,x) - U(t-dt,x) \end{aligned}$$

HPX: 1D Wave Equation

$$U(t+dt,x) = \alpha^2(U(t,x+dx) + U(t,x-dx)) + 2(1-\alpha^2)U(t,x) - U(t-dt,x)$$

- From this equation, we can see the dependencies. Calculating the value of the equation at a particular timestep requires the values of the equation:
  - One timestep ago at $x + dx$.
  - One timestep ago at $x - dx$.
  - One timestep ago at x.
  - Two timesteps ago at x.

HPX: 1D Wave Equation

$$U(t+dt,x) = \alpha^2\big(U(t,x+dx) + U(t,x-dx)\big) + 2(1-\alpha^2)U(t,x) - U(t-dt,x)$$

- The following lines of code creates the required dependency structure:

```
wave_action act;
hpx::future<double> u_t_minus_1_x_plus_dx  = hpx::async(act, hpx::find_here(), t-1, x+1);
hpx::future<double> u_t_minus_1_x_minus_dx = hpx::async(act, hpx::find_here(), t-1, x-1);
hpx::future<double> u_t_minus_1_x          = hpx::async(act, hpx::find_here(), t-1, x);
hpx::future<double> u_t_minus_2_x          = hpx::async(act, hpx::find_here(), t-2, x);

u[t][x] = alpha_2(u_t_minus_1_x_plus_dx.get() + u_t_minus_1_x_minus_dx.get())
        + 2(1-alpha_2)u_t_minus_1_x.get() - u_t_minus_2_x.get()
```

HPX

- The ability to use functional programming techniques such as function composition is important to us.
  - Thus, serializable `std::bind`, serializable `std::function`, etc.
- These constructs will be especially important for interoperation with generic parallel algorithms.
  - For example, `hpx::for_each`, etc.

TIPS & TRICKS

Tips & Tricks

- Object transmission
  - Zero copy
- Asio
  - TCP Congestation
  - I/O Service Pools
- Serialization
  - Bitwise serialization
  - Array optimizations
  - Exporting templates
  - XML & NVP
  - Portable Archives

OT: Zero Copy

- There's a problem with how we're doing serialization to the network – with Serialization, we're copying data that could be passed directly to Asio.
- Zero copy – e.g. passing contiguous blocks of memory for writes/read directly to the kernel, to avoid unnecessary memory copies.
  - Remember scatter/gather from before?
- How can we achieve this with Serialization?

OT: Zero Copy

- Special zero copy archive:
  - Uses a `std::vector<>` of Asio buffers instead of streams.
  - Every contiguous block of memory is placed into a buffer.
- Two-pass write/read.
- Write:
  - Pass 1: Iterate over the buffers and create an array containing the sizes of each buffer (a meta-data array). Write the size of this array. Then, write this meta-data array.
  - Pass 2: Iterate over all the buffers and write them.
- Read:
  - Pass 1: Read the size of the meta-data array. Read the meta-data array. Iterate over all the sizes, and set up data structures as needed (resize, etc).
  - Pass 2: Iterate over all the buffers and read into them.

Asio: TCP Congestation

- TCP_NODELAY option: disables the Nagle algorithm.

```
asio_tcp::socket socket(io_service);
// ...
asio_tcp::no_delay option(true);
socket.set_option(option);
```

- The Nagle algorithm reduces TCP/IP congestation by buffering small outgoing messages if there are unacknowledged writes pending.

```
send(new_data):
    buffered_data += new_data
    if (buffered_data >= limit) or (∄ unACKed writes)
        send buffered_data
```

Asio: TCP Congestation

- TCP_QUICKACK option: disable buffering of TCP acknowledgement (ACK).

```
asio_tcp::socket socket(io_service);
// ...
asio::detail::socket_option::boolean<
    IPPROTO_TCP, TCP_QUICKACK> quickack(true);
socket.set_option(quickack);
```

- Important note: you must set TCP_QUICKACK after every read operation (the OS will reset the flag after each read).
- RFC 1122 allows a host to delay sending an ACK for up to 500ms.

Asio: TCP Congestation

- When to use `TCP_NODELAY` and `TCP_QUICKACK`:
  - When latency is more important than bandwidth.
    - Real time systems.
  - When you have a high throughput network
    - High performance computing (HPC).
  - When you don't care about congestion.
- Bad things can happen if ACKs are buffered and Nagle is on.
  - A sends data to B.
  - B receives data from A, and buffers the ACK.
  - A tries to send a small message to B, and the message gets buffered because A hasn't received an ACK for the first write.
- Remember, B can wait up to 500ms to send.

Asio: I/O Service Pools

- `io_service_pool` – a pool of `io_service` objects and `std::thread` objects.
  - One `std::thread` for each `io_service` object.
  - Each thread is calling `io_service::run()`.
  - `io_service::run()` will normally return if no work is available, but we use a trick to keep the threads in the event processing loop.

    ```
    asio::io_service::work w(io_service);
    io_service.run();
    ```
  - Round robin scheme for distributing work across the `io_service` objects.

Asio: I/O Service Pools

```
struct io_service_pool : boost::noncopyable
{
    explicit io_service_pool(boost::uint64_t pool_size);

    ~io_service_pool();

    /// Run all io_service objects in the pool. If join_threads is true
    /// this will also wait for all threads to complete.
    bool run(bool join_threads = true);

    /// Stop all io_service objects in the pool.
    void stop();

    /// Join all io_service threads in the pool.
    void join();

    // Get an io_service object. If n != -1, then it returns
    // the nth io_service object. Otherwise, a round robin scheme
    // is used to determine which io_service object to return.
    asio::io_service& get_io_service(int index = -1);

    std::size_t size() const;
};
```

Serialization: Bitwise Serialization

- For certain types, Serialization will simply copy the bits of an object to serialize it. A type will be serialized in this fashion if the type trait `is_bitwise_serializable<T>` evalutes to `boost::mpl::true_`.
- This should generally be used only for POD types.
- `BOOST_IS_BITWISE_SERIALIZABLE(T)` will define the trait for the simple cases.

Serialization: Bitwise Serialization

```
namespace mpl = boost::mpl;
namespace serialization = boost::serialization;

struct are_elements_bitwise_serializable
{
    template <typename State, typename T>
    struct apply : mpl::and_<
        serialization::is_bitwise_serializable<T>, State> {};
};

template <typename T>
struct is_sequence_bitwise_serializable
  : mpl::fold<T, mpl::true_, are_elements_bitwise_serializable> {};

namespace boost { namespace serialization {

template <typename... T>
struct is_bitwise_serializable<std::tuple<T...> >
  : is_sequence_bitwise_serializable<std::tuple<T...> > {};

}}
```

Serialization: Bitwise Serialization

```
namespace mpl = boost::mpl;
namespace serialization = boost::serialization;

struct are_elements_bitwise_serializable
{
    template <typename State, typename T>
    struct apply : mpl::and_<
        serialization::is_bitwise_serializable<T>, State> {};
};

template <typename T>
struct is_sequence_bitwise_serializable
  : mpl::fold<T, mpl::true_, are_elements_bitwise_serializable> {};

namespace boost { namespace serialization {

template <typename... T>
struct is_bitwise_serializable<std::tuple<T...> >
  : is_sequence_bitwise_serializable<std::tuple<T...> > {};

}}
```

Serialization: Bitwise Serialization

```
namespace mpl = boost::mpl;
namespace serialization = boost::serialization;

struct are_elements_bitwise_serializable
{
    template <typename State, typename T>
    struct apply : mpl::and_<
        serialization::is_bitwise_serializable<T>, State> {};
};

template <typename T>
struct is_sequence_bitwise_serializable
  : mpl::fold<T, mpl::true_, are_elements_bitwise_serializable> {};

namespace boost { namespace serialization {

template <typename... T>
struct is_bitwise_serializable<std::tuple<T...> >
  : is_sequence_bitwise_serializable<std::tuple<T...> > {};

}}
```

Serialization: Array Optimizations

- When serializing arrays (and `std::vector<>`, `std::array<>`, etc), Serialization can optionally use array optimizations (serializing an entire contiguous block of memory).
  - **O**(1) time complexity.
- If these optimizations are not used, these data structures will be serialized elementwise.
  - **O**(N) time complexity.
  - Depending on the archive, may have a larger representation.
- If you are writing your own archive, you have to explicitly enable these:

```
BOOST_SERIALIZATION_USE_ARRAY_OPTIMIZATION(my_archive)
```

Serialization: Exporting Templates

- What if we have a template derived class that we want to serialize through a base class pointer?
- We can register each instantiation explicitly.
- Or we can use Serialization's GUID customization point (recently added feature).

```
HPX_SERIALIZATION_REGISTER_TEMPLATE(
    (template <typename T>), (A<T>)
);
```

- What does this do?
  - Hooks into Serialization, uses `typeid()` to generate the key for each instantiation.
  - Hashes the keys with SHA1 to decrease key size.
    - The keys go into our archives, and we have some templates in our code with LONG identifiers.

Serialization: Portable Archives

- HPX uses portable binary archives which are compatible across systems with different endianness:
  `hpx::util::portable_binary_iarchive`
  `hpx::util::portable_binary_oarchive`
- Tested between x86 and ARM systems in big endian mode.
- These archives also expose an API for compression.
- They can use a container with a `value_type` of `char` instead of a stream for input/output.
- You can tell these archives to not write out the Serialization archive header. This saves a couple of bytes for each archive.

STE||AR

stellar.cct.lsu.edu

```
git@github.com:STEllAR-GROUP/cppnow2013_ot.git
```