NEEV: Event-driven networking library
FOSDEM 2014

Pierre TALBOT (ptalbot@hyc.io)

University of Pierre et Marie Curie

2014, February 1

Content

Add-on server

1. Manage user-made contents (UMC).
2. Dependencies resolution.
3. Synchronization with translation.

I didn't even finish (1).

About me

- Belgian (French side).
- Still a student, at the University Pierre et Marie Curie in Paris.
- Interest in software engineering, language design and concurrency.

Open-source involvement

- **Boost C++ library** Working on Boost.Check and Boost.Expected
- **Wesnoth** The add-on server (UMCD).

What is NEEV?

- Generic code extracted from UMCD and packaged as a library.
- Tons of changes since.
- NEEV stands for "Network events".

Details

- Host on Github: `https://github.com/ptal/neev`
- Licensed under the Boost software license.
- Currently you need the latest Boost version (1.55) and C++11.

Which games?

- Best suited for turn-based games.
- We might add support later for real-time games.
- Not limited to game application.

Content

Neev in a nutshell

Traits

- Event-driven
- Asynchronous
- Extensible

Before diving into NEEV, let's see how Boost.Asio works.

Boost.Asio in a really thin nutshell

Example

1. You initiate an asynchronous operation on a socket.
2. When it finishes, the result is put onto a completion queue.
3. And the proactor dispatches the result to the completion handler.

It's easier to see Boost.Asio as a producer-consumer queue.

Start with empty queue

main()

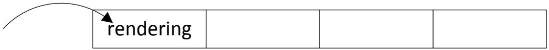
- io_service.post(rendering);
- io_service.post(sfml_events);
- io_service.run();

Empty task queue

Push onto the queue

main()

- **io_service.post(rendering);**
- io_service.post(sfml_events);
- io_service.run();



Task queue (size = 1)

Push onto the queue

main()

- io_service.post(rendering);
- **io_service.post(sfml_events);**
- io_service.run();

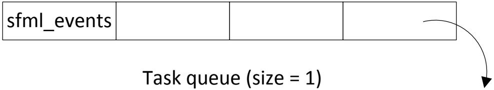
sfml_events	rendering		

Task queue (size = 2)

Pop from the queue

main()

- io_service.post(rendering);
- io_service.post(sfml_events);
- **io_service.run();**



Task queue (size = 1)

rendering();

rendering()

- **render()**
- io_service.post(rendering);

Execute a task

main()

- io_service.post(rendering);
- io_service.post(sfml_events);
- **io_service.run();**

rendering	sfml_events		

Task queue (size = 2)

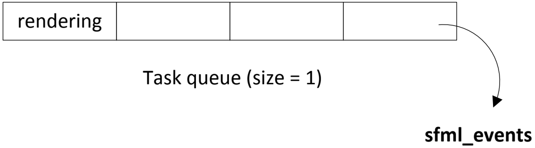
rendering()

- **i = i + 1**
- **io_service.post(rendering);**

Pop from the queue

main()

- io_service.post(rendering);
- io_service.post(sfml_events);
- **io_service.run();**



Task queue (size = 1)

Start two threads

main()

- io_service.post(rendering);
- io_service.post(sfml_events);
- std::thread t1(&io_service::run, std::ref(io_service));
- std::thread t2(&io_service::run, std::ref(io_service));
- t1.join();
- t2.join();

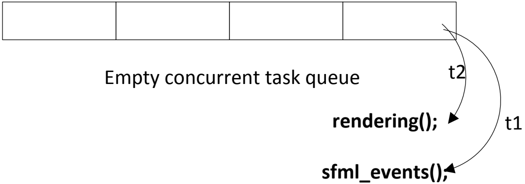
sfml_events	rendering		

Concurrent task queue (size = 2)

Concurrently execute threads

main()

- io_service.post(rendering);
- io_service.post(sfml_events);
- std::thread t1(&io_service::run, std::ref(io_service));
- std::thread t2(&io_service::run, std::ref(io_service));
- t1.join();
- t2.join();



Empty concurrent task queue

Content

I'm the client, how do I connect?

```
void handler(const socket_ptr& socket);

int main()
{
  boost::asio::io_service io_service;
  neev::client client(io_service);
  client.on_event<neev::connection_success>(handler);
  client.async_connect("localhost", "12222");
  io_service.run();
  return 0;
}
```

Client connection events

- **try_connecting_with_ip**: `void handler(const std::string&);`
- **connection_success**: `void handler(const socket_ptr&);`
- **connection_failure**: `void handler(const boost::system::error_code&);`

How to launch the server?

```
void on_new_client(const socket_type& socket);

int main()
{
  basic_server server;
  server.on_event<new_client>(on_new_client);
  server.launch("12222");
  return 0;
}
```

Server connection events

- **start_success**: void handler(const endpoint_type&);
- **new_client**: void handler(const socket_ptr&);
- **run_exception**: void handler(const std :: exception&);
- **run_unknown_exception**: void handler();
- **start_failure**: void handler();
- **endpoint_failure**: void handler(const std :: string&);

Simple position structure

```
struct Position
{
  std::int32_t x, y, z;

  template <class Archive>
  void serialize(Archive& ar, const unsigned int)
  {
    ar & x & y & z;
  }
};
```

Sending

```
void send_random_pos(const socket_type& socket)
{
  auto sender = neev::make_archive16_sender<no_timer>(
    socket,
    make_random_position());

  sender->on_event<neev::transfer_complete>(handler);

  sender->async_transfer();
}
```

Receiving

```
void receive_random_pos(const socket_type& socket)
{
  auto receiver = make_archive16_receiver<Position,no_timer>(
      socket);

  receiver->on_event<transfer_complete>([=](){
    std::cout << receiver->data() << std::endl;});

  receiver->async_transfer();
}
```

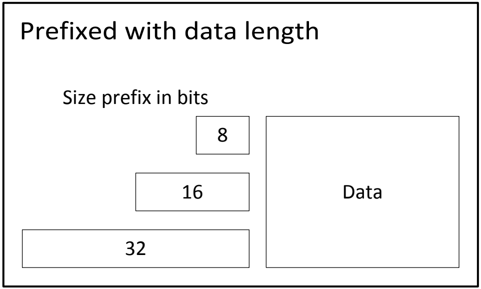
Transfer events

- **transfer_complete**: `void handler();`
- **transfer_error**: `void handler(const boost::system::error_code&);`
- **transfer_on_going**: `void handler(std::size_t, std::size_t);`
- **chunk_complete**: `void handler(events_subscriber_view<transfer_events>);`

Buffers

Different kinds

- Fixed-size buffers (e.g. text data, files, ...).
- Non-delimited buffers (e.g. JSON, XML, binary data).



Non-delimited buffers

First packet

{"age":22; "na

Parsing

Parsing stopped at « key value». Store the current context and return « undeterminate state ».

Second packet

me"="Church"}

Parsing

Restore context, continue parsing and return parsed tree.

Why event-driven?

- Clean design.
- Less code and better factorization.
- Compositionality.

Spaghetti code?

- It might be if we relaunched events inside event-handlers.
- Forbidden by design.

Why asynchronous?

- Inherits the pros and cons of Boost.Asio

Advantages

- Portability and efficiency (use native asynchronous I/O API if available).
- Decoupling threads from concurrency.
- Scalability.

Disadvantages

- Program complexity (separation in time and space between operation initiation and completion).
- Pending operation must be allocated on the heap.

Content

More buffer

Plenty of work

- XML;
- JSON;
- WML;
- True binary data.

- Support for existing XML/JSON libraries.

More low-level protocols support

- UDP support
- SSL transmission
- Simple heartbeat protocol

And finally...

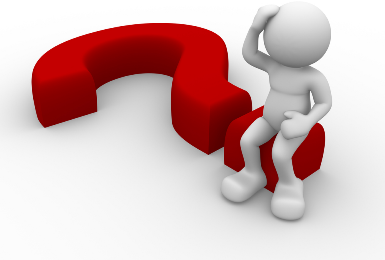
- Allocator support.
- More documentation, examples and tests.
- Concurrent development with the Wesnoth add-on server.

Contributors

Do not hesitate to contribute ;-)

Questions

Feel free to ask any questions.



Thanks for your attention!