Asynchronous Programming with Seastar

Nadav Har'El - nyh@ScyllaDB.com   Avi Kivity - avi@ScyllaDB.com

Contents

1 Introduction

1.1 Asynchronous programming

A server for a network protocol, such as the classic HTTP (Web) or SMTP (e-mail) servers, inherently deals with parallelism: Multiple clients send requests in parallel, and we cannot start handling one request before finishing to handle the next. A request may, and often does, need to block because of various reasons — a full TCP window (i.e., a slow connection), disk I/O, or even the client holding on to an inactive connection — and the server needs to handle other connections as well.

The most straightforward way to handle such parallel connections, employed by classic network servers such as Inetd, Apache Httpd and Sendmail, is to use a separate operating-system process per connection. This technique evolved over the years to improve its performance: At first, a new process was spawned to handle each new connection; Later, a pool of existing processes was kept and each new connection was assigned to an unemployed process from the pool; Finally, the processes were replaced by threads. However, the common idea behind all these implementations is that at each moment, each process handles exclusively a single connection. Therefore, the server code is free to use blocking system calls, such as reading or writing to a connection, or reading from disk, and if this process blocks, all is well because we have many additional processes handling other connections in parallel.

Programming a server which uses a process (or a thread) per connection is known as *synchronous* programming, because the code is written linearly, and one line of code starting to run after the previous line finished. For example, the code may read a request from a socket, parse the request, and then piecemeal read a file from disk and write it back to the socket. Such code is easy to write, almost like traditional non-parallel programs. In fact, it's even possible to run an external non-parallel program to handle each request — this is for example how Apache HTTPd ran "CGI" programs, the first implementation of dynamic Web-page generation.

> NOTE: although the synchronous server application is written in a linear, non-parallel, fashion, behind the scenes the kernel helps ensure that everything happens in parallel and the machine's resources — CPUs, disk and network — are fully utilized. Beyond the obvious parallelism (we have multiple processes handling multiple connections in parallel), the kernel may even parallelize the work of one individual connection — for example process an outstanding disk request (e.g., read from a disk file) in parallel with handling the network connection (send buffered-but-yet-unsent data, and buffer newly-received data until the application is ready to read it).

But synchronous, process-per-connection, server programming didn't come without disadvantages and costs. Slowly but surely, server authors realized that starting a new process is slow, context switching is slow, and each process comes with significant overheads — most notably the size of its stack. Server and kernel authors worked hard to mitigate these overheads: They switched from processes to threads, from creating new threads to thread pools, they lowered default stack size of each thread, and increased the virtual memory size to allow more partially-utilized stacks. But still, servers with synchronous designs had unsatisfactory performance, and scaled badly as the number of concurrent connections grew. In 1999, Dan Kigel popularized "the C10K problem", the need of a single server to efficiently handle 10,000 concurrent connections — most of them slow or even inactive.

The solution, which became popular in the following decade, was to abandon the cozy but inefficient synchronous server design, and switch to a new type of server design — the *asynchronous*, or *event-driven*, server. An event-driven server has just one thread, or more accurately, one thread per CPU. This single thread runs a tight loop which, at each iteration, checks, using `poll()` (or the more efficient `epoll`) for new events on many open file descriptors, e.g., sockets. For example, an event can be a socket becoming readable (new data has arrived from the remote end) or becoming writable (we can send more data on this connection). The application handles this event by doing some non-blocking operations, modifying one or more of the file descriptors, and maintaining its knowledge of the *state* of this connection.

However, writers of asynchronous server applications faced, and still face today, two significant challenges:

- **Complexity:** Writing a simple asynchronous server is straightforward. But writing a *complex* asynchronous server is notoriously difficult. The handling of a single connection, instead of being a simple easy-to-read function call, now involves a large number of small callback functions, and a complex state machine to remember which function needs to be called when each event occurs.
- **Non-blocking:** Having just one thread per core is important for the performance of the server application, because context switches are slow. However, if we only have one thread per core, the event-handling functions must *never* block, or the core will remain idle. But some existing programming languages and frameworks leave the server author no choice but to use blocking functions, and therefore multiple threads.
  For example, `Cassandra` was written as an asynchronous server application; But because disk I/O was implemented with `mmap`ed files, which can uncontrollably block the whole thread when accessed, they are forced to run multiple threads per CPU.

Moreover, when the best possible performance is desired, the server application, and its programming framework, has no choice but to also take the following into account:

- **Modern Machines**: Modern machines are very different from those of just 10 years ago. They have many cores and deep memory hierarchies (from L1 caches to NUMA) which reward certain programming practices and penalizes others: Unscalable programming practices (such as taking locks) can devastate performance on many cores; Shared memory and lock-free synchronization primitives are available (i.e., atomic operations and memory-ordering fences) but are dramatically slower than operations that involve only data in a single core's cache, and also prevent the application from scaling to many cores.
- **Programming Language:** High-level languages such Java, Javascript, and similar "modern" languages are convenient, but each comes with its own set of assumptions which conflict with the requirements listed above. These languages, aiming to be portable, also give the programmer less control over the performance of critical code. For really optimal performance, we need a programming language which gives the programmer full control, zero run-time overheads, and on the other hand — sophisticated compile-time code generation and optimization.

Seastar is a framework for writing asynchronous server applications which aims to solve all four of the above challenges: It is a framework for writing *complex* asynchronous applications involving both network and disk I/O. The framework's fast path is entirely single-threaded (per core), scalable to many cores and minimizes the use of costly sharing of memory between cores. It is a C++14 library, giving the user sophisticated compile-time features and full control over performance, without run-time overhead.

1.2 Seastar

Seastar is an event-driven framework allowing you to write non-blocking, asynchronous code in a relatively straightforward manner (once understood). Its APIs are based on futures. Seastar utilizes the following concepts to achieve extreme performance:

- **Cooperative micro-task scheduler**: instead of running threads, each core runs a cooperative task scheduler. Each task is typically very lightweight – only running for as long as it takes to process the last I/O operation's result and to submit a new one.
- **Share-nothing SMP architecture**: each core runs independently of other cores in an SMP system. Memory, data structures, and CPU time are not shared; instead, inter-core communication uses explicit message passing. A Seastar core is often termed a shard. TODO: more here `https://github.com/scylladb/seastar/wiki/SMP`
- **Future based APIs**: futures allow you to submit an I/O operation and to chain tasks to be executed on completion of the I/O operation. It is easy to run multiple I/O operations in parallel - for example, in response to a request coming from a TCP connection, you can issue multiple disk I/O requests, send messages to other cores on the same system, or send requests to other nodes in the cluster, wait for some or all of the results to complete, aggregate the results, and send a response.
- **Share-nothing TCP stack**: while Seastar can use the host operating system's TCP stack, it also provides its own high-performance TCP/IP stack built on top of the task scheduler and the share-nothing architecture. The stack provides zero-copy in both directions: you can process data directly from the TCP stack's buffers, and send the contents of your own data structures as part of a message without incurring a copy. Read more…
- **DMA-based storage APIs**: as with the networking stack, Seastar provides zero-copy storage APIs, allowing you to DMA your data to and from your storage devices.

This tutorial is intended for developers already familiar with the C++ language, and will cover how to use Seastar to create a new application.

TODO: copy text from `https://github.com/scylladb/seastar/wiki/SMP`
`https://github.com/scylladb/seastar/wiki/Networking`

2 Getting started

The simplest Seastar program is this:

```
#include "core/app-template.hh"
#include "core/reactor.hh"
#include <iostream>

int main(int argc, char** argv) {
    app_template app;
    app.run(argc, argv, [] {
            std::cout << "Hello world\n";
            return make_ready_future<>();
    });
}
```

As we do in this example, each Seastar program must define and run, an `app_template` object. This object starts the main event loop (the Seastar *engine*) on one or more CPUs, and then runs the given function - in this case an unnamed function, a *lambda* - once.

The "`return make_ready_future<>();`" causes the event loop, and the whole application, to exit immediately after printing the "Hello World" message. In a more typical Seastar application, we will want event loop to remain alive and process incoming packets (for example), until explicitly exited. Such applications will return a *future* which determines when to exit the application. We will introduce futures and how to use them below. In any case, the regular C `exit()` should not be used, because it prevents Seastar or the application from cleaning up appropriately.

To compile this program, first make sure you have downloaded and built Seastar. Below we'll use the symbol `$SEASTAR` to refer to the directory where Seastar was built (Seastar doesn't yet have a "`make install`" feature).

Now, put the above program in a source file anywhere you want, let's call the file `getting-started.cc`. You can compile it with the following command:

```
c++ ‘pkg-config --cflags --libs $SEASTAR/build/release/seastar.pc‘ getting-started.cc
```

Linux's pkg-config is a useful tool for easily determining the compilation and linking parameters needed for using various libraries - such as Seastar.

The program now runs as expected:

```
$ ./a.out
Hello world
$
```

3 Threads and memory

3.1 Seastar threads

As explained in the introduction, Seastar-based programs run a single thread on each CPU. Each of these threads runs its own event loop, known as the *engine* in Seastar nomenclature. By default, the Seastar application will take over all the available cores, starting one thread per core. We can see this with the following program, printing `smp::count` which is the number of started threads:

```
#include "core/app-template.hh"
#include "core/reactor.hh"
#include <iostream>

int main(int argc, char** argv) {
    app_template app;
    app.run(argc, argv, [] {
            std::cout << smp::count << "\n";
            return make_ready_future<>();
    });
}
```

On a machine with 4 hardware threads (two cores, and hyperthreading enabled), Seastar will by default start 4 engine threads:

```
$ ./a.out
4
```

Each of these 4 engine threads will be pinned (a la **taskset(1)**) to a different hardware thread. Note how, as we mentioned above, the app's initialization function is run only on one thread, so we see the ouput "4" only once. Later in the tutorial we'll see how to make use of all threads.

The user can pass a command line parameter, `-c`, to tell Seastar to start fewer threads than the available number of hardware threads. For example, to start Seastar on only 2 threads, the user can do:

```
$ ./a.out -c2
2
```

When the machine is configured as in the example above - two cores with two hyperthreads on each - and only two threads are requested, Seastar ensures that each thread is pinned to a different core, and we don't get the two threads competing as hyperthreads of the same core (which would, of course, damage performance).

We cannot start more threads than the number of hardware threads, as allowing this will be grossly inefficient. Trying it will result in an error:

```
$ ./a.out -c5
terminate called after throwing an instance of 'std::runtime_error'
  what():  insufficient processing units
abort (core dumped)
```

The error is an exception thrown from app.run, which we did not catch, leading to this ugly uncaught-exception crash. It is better to catch this sort of startup exceptions, and exit gracefully without a core dump:

```
#include "core/app-template.hh"
#include "core/reactor.hh"
#include <iostream>
#include <stdexcept>

int main(int argc, char** argv) {
    app_template app;
    try {
        app.run(argc, argv, [] {
            std::cout << smp::count << "\n";
            return make_ready_future<>();
        });
    } catch(std::runtime_error &e) {
        std::cerr << "Couldn't start application: " << e.what() << "\n";
        return 1;
    }
    return 0;
}
```

```
$ ./a.out -c5
Couldn't start application: insufficient processing units
```

Note that catching the exceptions this way does **not** catch exceptions thrown in the application's actual asynchronous code. We will discuss these later in this tutorial.

3.2 Seastar memory

As explained in the introduction, Seastar applications shard their memory. Each thread is preallocated with a large piece of memory (on the same NUMA node it is running on), and uses only that memory for its allocations (such as `malloc()` or `new`).

By default, the machine's **entire memory** except a small reservation left for the OS (defaulting to 512 MB) is pre-allocated for the application in this manner. This default can be changed by *either* changing the amount reserved for the OS (not used by Seastar) with the `--reserve-memory` option, or by explicitly giving the amount of memory given to the Seastar application, with the `-m` option. This amount of memory can be in bytes, or using the units "k", "M", "G" or "T". These units use the power-of-two values: "M" is a **mebibyte**, 2^20 (=1,048,576) bytes, not a **megabyte** (10^6 or 1,000,000 bytes).

Trying to give Seastar more memory than physical memory immediately fails:

```
$ ./a.out -m10T
Couldn't start application: insufficient physical memory
```

4 Introducing futures and continuations

Futures and continuations, which we will introduce now, are the building blocks of asynchronous programming in Seastar. Their strength lies in the ease of composing them together into a large, complex, asynchronous program, while keeping the code fairly readable and understandable.

A future is a result of a computation that may not be available yet.
Examples include:

- a data buffer that we are reading from the network
- the expiration of a timer
- the completion of a disk write
- the result of a computation that requires the values from one or more other futures.

The type `future<int>` variable holds an int that will eventually be available - at this point might already be available, or might not be available yet. The method available() tests if a value is already available, and the method get() gets the value. The type `future<>` indicates something which will eventually complete, but not return any value.

A future is usually returned by an **asynchronous function**, also known as a **promise**, a function which returns a future and arranges for this future to be eventually resolved. One simple example is Seastar's function sleep():

```
future<> sleep(std::chrono::duration<Rep, Period> dur);
```

This function arranges a timer so that the returned future becomes available (without an associated value) when the given time duration elapses.

A **continuation** is a callback (typically a lambda) to run when a future becomes available. A continuation is attached to a future with the `then()` method. Here is a simple example:

```
#include "core/app-template.hh"
#include "core/sleep.hh"
#include <iostream>

int main(int argc, char** argv) {
    app_template app;
    app.run(argc, argv, [] {
        std::cout << "Sleeping... " << std::flush;
        using namespace std::chrono_literals;
        return sleep(1s).then([] {
            std::cout << "Done.\n";
        });
    });
}
```

In this example we see us getting a future from `sleep(1s)`, and attaching to it a continuation which prints a "Done." message. The future will become available after 1 second has passed, at which point the continuation is executed. Running this program, we indeed see the message "Sleeping..." immediately, and one second later the message "Done." appears and the program exits.

The return value of `then()` is itself a future which is useful for chaining multiple continuations one after another, as we will explain below. But here we just note that we `return` this future from `app.run`'s function, so that the program will exit only after both the sleep and its continuation are done.

To avoid repeating the boilerplate "app_engine" part in every code example in this tutorial, let's create a simple main() with which we will compile the following examples. This main just calls function `future<> f()`, does the appropriate exception handling, and exits when the future returned by `f` is resolved:

```
#include "core/app-template.hh"
#include <iostream>
#include <stdexcept>
```

```
extern future<> f();

int main(int argc, char** argv) {
    app_template app;
    try {
        app.run(argc, argv, f);
    } catch(std::runtime_error &e) {
        std::cerr << "Couldn't start application: " << e.what() << "\n";
        return 1;
    }
    return 0;
}
```

Compiling together with this `main.cc`, the above sleep() example code becomes:

```
#include "core/sleep.hh"
#include <iostream>

future<> f() {
    std::cout << "Sleeping... " << std::flush;
    using namespace std::chrono_literals;
    return sleep(1s).then([] {
        std::cout << "Done.\n";
    });
}
```

So far, this example was not very interesting - there is no parallelism, and the same thing could have been achieved by the normal blocking POSIX `sleep()`. Things become much more interesting when we start several sleep() futures in parallel, and attach a different continuation to each. Futures and continuation make parallelism very easy and natural:

```
#include "core/sleep.hh"
#include <iostream>

future<> f() {
    std::cout << "Sleeping... " << std::flush;
    using namespace std::chrono_literals;
    sleep(200ms).then([] { std::cout << "200ms " << std::flush; });
    sleep(100ms).then([] { std::cout << "100ms " << std::flush; });
    return sleep(1s).then([] { std::cout << "Done.\n"; });
}
```

Each `sleep()` and `then()` call returns immediately: `sleep()` just starts the requested timer, and `then()` sets up the function to call when the timer expires. So all three lines happen immediately and f returns. Only then, the event loop starts to wait for the three outstanding futures to become ready, and when each one becomes ready, the continuation attached to it is run. The output of the above program is of course:

```
$ ./a.out
Sleeping... 100ms 200ms Done.
```

`sleep()` returns `future<>`, meaning it will complete at a future time, but once complete, does not return any value. More interesting futures do specify a value of any type (or multiple values) that will become available later. In the following example, we have a function returning a `future<int>`, and a continuation to be run once this value becomes available. Note how the continuation gets the future's value as a parameter:

```
#include "core/sleep.hh"
#include <iostream>

future<int> slow() {
    using namespace std::chrono_literals;
    return sleep(100ms).then([] { return 3; });
}

future<> f() {
    return slow().then([] (int val) {
        std::cout << "Got " << val << "\n";
    });
}
```

The function `slow()` deserves more explanation. As usual, this function returns a future immediately, and doesn't wait for the sleep to complete, and the code in `f()` can chain a continuation to this future's completion. The future returned by `slow()` is itself a chain of futures: It will become ready once sleep's future becomes ready and then the value 3 is returned. We'll explain below in more details how `then()` returns a future, and how this allows *chaining* futures.

This example begins to show the convenience of the futures programming model, which allows the programmer to neatly encapsulate complex asynchronous operations. slow() might involve a complex asynchronous operation requiring multiple steps, but its user can use it just as easily as a simple sleep(), and Seastar's engine takes care of running the continuations whose futures have become ready at the right time.

4.1 Ready futures

A future value might already be ready when `then()` is called to chain a continuation to it. This important case is optimized, and *usually* the continuation is run immediately instead of being registered to run later in the next iteration of the event loop.

This optimization is done *usually*, though sometimes it is avoided: The implementation of `then()` holds a counter of such immediate continuations, and after many continuations have been run immediately without returning to the event loop (currently the limit is 256), the next continuation is deferred to the event loop in any case. This is important because in some cases (such as future loops, discussed later) we could find that each ready continuation spawns a new one, and without this limit we can starve the event loop. It important not to starve the event loop, as this would starve continuations of futures that weren't ready but have since become ready, and also starve the important **polling** done by the event loop (e.g., checking whether there is new activity on the network card).

`make_ready_future<>` can be used to return a future which is already ready. The following example is identical to the previous one, except the promise function `fast()` returns a future which is already ready, and not one which will be ready in a second as in the previous example. The nice thing is that the consumer of the future does not care, and uses the future in the same way in both cases.

```
#include "core/future.hh"
#include <iostream>

future<int> fast() {
    return make_ready_future<int>(3);
}

future<> f() {
    return fast().then([] (int val) {
        std::cout << "Got " << val << "\n";
    });
}
```

5 Continuations

5.1 Capturing state in continuations

We've already seen that Seastar *continuations* are lambdas, passed to the `then()` method of a future. In the examples we've seen so far, lambdas have been nothing more than anonymous functions. But C++11 lambdas have one more trick up their sleeve, which is extremely important for future-based asynchronous programming in Seastar: Lambdas can **capture** state. Consider the following example:

```
#include "core/sleep.hh"
#include <iostream>

future<int> incr(int i) {
    using namespace std::chrono_literals;
    return sleep(10ms).then([i] { return i + 1; });
}

future<> f() {
    return incr(3).then([] (int val) {
        std::cout << "Got " << val << "\n";
    });
}
```

The future operation `incr(i)` takes some time to complete (it needs to sleep a bit first :wink:), and in that duration, it needs to save the `i` value it is working on. In the early event-driven programming models, the programmer needed to explicitly define an object for holding this state, and to manage all these objects. Everything is much simpler in Seastar, with C++11's lambdas: The *capture syntax* `[i]` in the above example means that the value of i, as it existed when incr() was called() is captured into the lambda. The lambda is not just a function - it is in fact an *object*, with both code and data. In essence, the compiler created for us automatically the state object, and we neither need to define it, nor to keep track of it (it gets saved together with the continuation, when the continuation is deferred, and gets deleted automatically after the continuation runs).

One implementation detail worth understanding is that when a continuation has captured state and is run immediately, this capture incurs no runtime overhead. However, when the continuation cannot be run immediately (because the future is not yet ready) and needs to be saved till later, memory needs to be allocated on the heap for this data, and the continuation's captured data needs to be copied there. This has runtime overhead, but it is unavoidable, and is very small compared to the parallel overhead in the threaded programming model (in a threaded program, this sort of state usually resides on the stack of the blocked thread, but the stack is much larger than our tiny capture state, takes up a lot of memory and causes a lot of cache pollution on context switches between those threads).

In the above example, we captured `i` *by value* - i.e., a copy of the value of `i` was saved into the continuation. C++ has two additional capture options: capturing by *reference* and capturing by *move*:

Using capture-by-reference in a continuation is almost always a mistake, and would lead to serious bugs. For example, if in the above example we captured a reference to i, instead of copying it,

```
future<int> incr(int i) {
    using namespace std::chrono_literals;
    return sleep(10ms).then([&i] { return i + 1; });   // Oops, the "&" here is wrong.
}
```

this would have meant that the continuation would contain the address of `i`, not its value. But `i` is a stack variable, and the incr() function returns immediately, so when the continuation eventually gets to run, long after incr() returns, this address will contain unrelated content.

An exception to this rule is the `do_with()` idiom, which we will introduce later, which ensures that an object lives throughout the life of the continuation. This makes capture-by-reference possible, and very convenient.

Using capture-by-*move* in continuations, on the other hand, is valid and very useful in Seastar applications. By **moving** an object into a continuation, we transfer ownership of this object to the continuation, and make it easy for the object to be automatically deleted when the continuation ends. For example, consider a traditional function taking a std::unique_ptr.

```
int do_something(std::unique_ptr<T> obj) {
     // do some computation based on the contents of obj, let's say the result is 17
     return 17;
     // at this point, obj goes out of scope so the compiler delete()s it.
```

By using unique_ptr in this way, the caller passes an object to the function, but tells it the object is now its exclusive responsibility - and when the function is done with the object, it should delete the object. How do we use unique_ptr in a continuation? The following won't work:

```
future<int> slow_do_something(std::unique_ptr<T> obj) {
    using namespace std::chrono_literals;
    return sleep(10ms).then([obj] { return do_something(std::move(obj))}); // WON'T COMPILE
}
```

The problem is that a unique_ptr cannot be passed into a continuation by value, as this would require copying it, which is forbidden because it violates the guarantee that only one copy of this pointer exists. We can, however, *move* obj into the continuation:

```
future<int> slow_do_something(std::unique_ptr<T> obj) {
    using namespace std::chrono_literals;
    return sleep(10ms).then([obj = std::move(obj)] {
        return do_something(std::move(obj))});
}
```

Here the use of `std::move()` causes obj's move-assignment is used to move the object from the outer function into the continuation. The notion of move (*move semantics*), introduced in C++11, is similar to a shallow copy followed by invalidating the source copy (so that the two copies do not co-exist, as forbidden by unique_ptr). After moving obj into the continuation, the top-level function can no longer use it (in this case it's of course ok, because we return anyway).

The `[obj = ...]` capture syntax we used here is new to C++14. This is the main reason why Seastar requires C++14, and does not support older C++11 compilers.

5.2 Chaining continuations

We already saw chaining example in slow() above. TODO: talk about the return from then, and returning a future and chaining more thens.

5.3 Lifetime

TODO: Talk about how we can't capture a reference to a local variable (the continuation runs when the local variable is gone). Take about *move*ing a variable into a continuation, about why with a chain of continuations it is difficult to move it between them so we have do_with, and also explain why lw_shared_ptr is very useful for this purpose.

TODO: Make a simple example (object containing int i, us calling a "slow" method on this object doing "sleep 1; return i" needing the object to exist 1 second later). move is enough. But if we need to call two methods? Show two idiomatic solutions: do_with and lw_shared_ptr.

6 Handling exceptions

An exception thrown in a continuation is implicitly captured by the system and stored in the future. A future that stores such an exception is similar to a ready future in that it can cause its continuation to be launched, but it does not contain a value – only the exception.

Calling `.then()` on such a future skips over the continuation, and transfers the exception for the input future (the object on which `.then()` is called) to the output future (`.then()`'s return value).

This default handling parallels normal exception behavior – if an exception is thrown in straight-line code, all following lines are skipped:

```
line1();
line2(); // throws!
line3(); // skipped
```

is similar to

```
return line1().then([] {
    return line2(); // throws!
}).then([] {
    return line3(); // skipped
});
```

Usually, aborting the current chain of operations and returning an exception is what's needed, but sometimes more fine-grained control is required. There are several primitives for handling exceptions:

1. `.then_wrapped()`: instead of passing the values carried by the future into the continuation, `.then_wrapped()` passes the input future to the continuation. The future is guaranteed to be in ready state, so the continuation can examine whether it contains a value or an exception, and take appropriate action.
2. `.finally()`: similar to a Java finally block, a `.finally()` continuation is executed whether or not its input future carries an exception or not. The result of the finally continuation is its input future, so `.finally()` can be used to insert code in a flow that is executed unconditionally, but otherwise does not alter the flow.

TODO: give example code for the above. Also mention handle_exception - although perhaps delay that to a later chapter?

6.1 Futures are single use

TODO: Talk about if we have a future variable, as soon as we get() or then() it, it becomes invalid - we need to store the value somewhere else. Think if there's an alternative we can suggest

7 Fibers

7.1 Loops

TODO: do_until and friends; parallel_for_each and friends; Use boost::counting_iterator for integers. map_reduce, as a shortcut (?) for parallel_for_each which needs to produce some results (e.g., logical_or of boolean results), so we don't need to create a lw_shared_ptr explicitly (or do_with).

7.2 Limiting parallelism with semaphores

Seastar's semaphores are the standard computer-science semaphores, adapted for futures. A semaphore is a counter into which you can deposit units or take them away. Taking units from the counter may wait if not enough units are available.

The most common use for a semaphore in Seastar is for limiting parallelism, i.e., limiting the number of instances of some code which can run in parallel. This can be important when each of the parallel invocations uses a limited resource (e.g., memory) so letting an unlimited number of them run in parallel can exhaust this resource.

Consider the following simple loop:

```
#include "core/sleep.hh"
future<> slow() {
    std::cerr << ".";
    return sleep(std::chrono::seconds(1));
}
future<> f() {
    return repeat([] {
        return slow().then([] { return stop_iteration::no; });
    });
}
```

This loop runs the `slow()` function (taking one second to complete) without any parallelism — the next `slow()` call starts only when the previous one completed. But what if we do not need to serialize the calls to `slow()`, and want to allow multiple instances of it to be ongoing concurrently?

Naively, we could achieve more parallelism, by starting the next call to `slow()` right after the previous call — ignoring the future returned by the previous call to `slow()` and not waiting for it to resolve:

```
future<> f() {
    return repeat([] {
        slow();
        return stop_iteration::no;
    });
}
```

But in this loop, there is no limit to the amount of parallelism — millions of `sleep()` calls might be active in parallel, before the first one ever returned. Eventually, this loop will consume all available memory and crash.

Using a semaphore allows us to run many instances of `slow()` in parallel, but limit the number of these parallel instances to, in the following example, 100:

```
future<> f() {
    return do_with(semaphore(100), [] (auto& limit) {
        return repeat([&limit] {
            return limit.wait(1).then([&limit] {
                slow().finally([&limit] {
                    limit.signal(1);
                });
                return stop_iteration::no;
            });
        });
    });
}
```

In this example, the semaphore starts with the counter at 100. The asynchronous operation (`slow()`) is only started when we can reduce the counter by one (`wait(1)`), and when `slow()` is done, the counter is increased back by one (`signal(1)`). This way, when 100 iterations have already started their work and have not yet finished, the 101st iteration will wait, until one of the ongoing operations will finish and return a unit to the semaphore. This will ensure that at each time we have at most 100 concurrent `slow()` operations running in the above code.

When the loop has an end (unlike the infinite loop in the above example), we often want, at the end of the loop, to wait for all the background operations which the loop started. We can easily do this by `wait()`ing on the original count of the semaphore. When the full count is finally available, it means that *all* the operations have completed. For example, the following loop ends after 456 iterations:

```
future<> f() {
    return do_with(0, semaphore(100), [] (auto& i, auto& limit) {
        return repeat([&i, &limit] {
            return limit.wait(1).then([&i, &limit] {
                slow().finally([&limit] {
                    limit.signal(1);
                });
                return i++ < 456 ? stop_iteration::no : stop_iteration::yes;
            });
        }).finally([&limit] {
            return limit.wait(100);
        });
    });
}
```

Note how after the `repeat` loop ends, we do a `wait(100)` to wait for the semaphore to reach its original value 100, meaning that all operations we started have completed. Note also how we used `finally()`, rather than `then()`, in two places, to ensure that the counter is increased even if a background operation finished unsuccessfully, and to ensure that we wait for all background operations to complete, even if the looped stopped prematurely because of an exception.

In the examples we saw in this section so far, we used a semaphore to control the parallelism of a *loop*: The loop waited for semaphore units to be available before proceeding to the next iteration. Semaphores are also useful to rare-limit work driven by external events:

Consider a case where an external source of events (e.g., incoming network requests) causes an asynchronous function `g()` to be called. Again, we want to limit the number of concurrent `g()` operations to 100: If g() is started when 100 other invocations are still ongoing, it will delay its real work until one of the other invocations has completed. As before, we can do this with a semaphore:

```
future<> g() {
    static thread_local semaphore limit(100);
    return limit.wait(1).then([] {
        return slow(); // do the real work of g()
    }).finally([&limit] {
        limit.signal(1);
    });
}
```

Note how we used a `static thread_local` semaphore, so that all calls to `g()` from the same shard count towards the same limit; As usual, a Seastar application is sharded so this limit is separate per shard (CPU thread). This is usually fine, because sharded applications consider resources to be separate per shard.

We have a shortcut `with_semaphore()` that can be used in this use case:

```
future<> g() {
    static thread_local semaphore limit(100);
    return with_semaphore(limit, 1, [] {
        return slow(); // do the real work of g()
    });
}
```

`with_semaphore()`, like the code above, waits for the given amount of units from the semaphore, then runs the given lambda, and when the future it returns resolves, it returns back the units to the semaphore. `with_semaphore()` returns a future which only resolves after all these steps are done.

This last feature of `with_semaphore()` — that it resolves only after the lambda's returned future resolves — means that this shortcut cannot be used in the loop examples in the beginning of the section. The loop needs to know when just the semaphore units are available, to start the next iteration, but `with_semaphore()` does not return such a future. Attempting to "cheat" by having the lambda passed to `with_semaphore` return an immediately-available future (and starting `slow()` without waiting for it) will not work, because now `with_semaphore` will return a unit immediately to the semaphore, rather than only doing that after `slow()` finished.

Because semaphores support waiting for any number of units, not just 1, we can use them for more than simple limiting of the number of parallel invocation. For example, consider we have an asynchronous function `using_lots_of_memory(size_t bytes)`, which uses `bytes` bytes of memory, and we want to ensure that not more than 1 MB of memory is used by all parallel invocations of this function — and that additional calls are delayed until previous calls have finished. We can do this with a semaphore:

```
future<> using_lots_of_memory(size_t bytes) {
    static thread_local semaphore limit(1000000); // limit to 1MB
    return with_semaphore(limit, bytes, [bytes] {
        // do something allocating 'bytes' bytes of memory
    });
}
```

Watch out that in the above example, a call to `using_lots_of_memory(2000000)` will return a future that never resolves, because the semaphore will never contain enough units to satisfy the semaphore wait. `using_lots_of_memory` should probably check whether `bytes` is above the limit, and throw an exception in that case.

TODO: say something about semaphore fairness - if someone is waiting for a lot of units and later someone asks for 1 unit, will both wait or will the request for 1 unit be satisfied?

TODO: say something about broken semaphores? (or in later section especially about breaking/closing/shutting down/etc?)

7.3 Pipes

7.4 Shutting down a service with a gate

Consider an application which has some long operation `slow()`, and many such operations may be started at any time. A number of `slow()` operations may even even be active in parallel. Now, you want to shut down this service, but want to make sure that before that, all outstanding operations are completed. Moreover, you don't want to allow new `slow()` operations to start while the shut-down is in progress.

This is the purpose of a `seastar::gate`. A gate `g` maintains an internal counter of operations in progress. We call `g.enter()` when entering an operation (i.e., before running `slow()`), and call `g.leave()` when leaving the operation (when a call to `slow()` completed). The method `g.close()` *closes the gate*, which means it forbids any further calls to `g.enter()` (such attempts will generate an exception); Moreover `g.close()` returns a future which resolves when all the existing operations have completed. In other words, when `g.close()` resolves, we know that no more invocations of `slow()` can be in progress - because the ones that already started have completed, and new ones could not have started.

The construct

```
seastar::with_gate(g, [] { return slow(); })
```

can be used as a shortcut to the idiom

```
g.enter();
slow().finally([&g] { g.leave(); });
```

Here is a typical example of using a gate:

```
#include "core/sleep.hh"
#include "core/gate.hh"
#include <boost/iterator/counting_iterator.hpp>

future<> slow(int i) {
    std::cerr << "starting " << i << "\n";
    return sleep(std::chrono::seconds(10)).then([i] {
        std::cerr << "done " << i << "\n";
    });
}

future<> f() {
    return do_with(seastar::gate(), [] (auto& g) {
        return do_for_each(boost::counting_iterator<int>(1),
                boost::counting_iterator<int>(6),
                [&g] (int i) {
            seastar::with_gate(g, [i] { return slow(i); });
            // wait one second before starting the next iteration
            return sleep(std::chrono::seconds(1));
        }).then([&g] {
            sleep(std::chrono::seconds(1)).then([&g] {
                // This will fail, because it will be after the close()
                g.enter();
                seastar::with_gate(g, [] { return slow(6); });
            });
            return g.close();
        });
    });
}
```

In this example, we have a function `future<> slow()` taking 10 seconds to complete. We run it in a loop 5 times, waiting 1 second between calls, and surround each call with entering and leaving the gate (using `with_gate`). After the 5th call, while all calls are still ongoing (because each takes 10 seconds to complete), we close the gate and wait for it before exiting the program. We also test that new calls cannot begin after closing the gate, by trying to enter the gate again one second after closing it.

The output of this program looks like this:

```
starting 1
starting 2
starting 3
starting 4
starting 5
WARNING: exceptional future ignored of type 'seastar::gate_closed_exception': gate closed
done 1
done 2
done 3
done 4
done 5
```

Here, the invocations of `slow()` were started at 1 second intervals. After the "`starting 5`" message, we closed the gate and another attempt to use it resulted in a `seastar::gate_closed_exception`, which we ignored and hence this message. At this point the application waits for the future returned by `g.close()`. This will happen once all the `slow()` invocations have completed: Immediately after printing "`done 5`", the test program stops.

As explained so far, a gate can prevent new invocations of an operation, and wait for any in-progress operations to complete. However, these in-progress operations may take a very long time to complete. Often, a long operation would like to know that a shut-down has been requested, so it could stop its work prematurely. An operation can check whether its gate was closed by calling the gate's `check()` method: If the gate is already closed, the `check()` method throws an exception (the same `seastar::gate_closed_exception` that `enter()` would throw at that point). The intent is that the exception will cause the operation calling it to stop at this point.

In the previous example code, we had an un-interruptible operation `slow()` which slept for 10 seconds. Let's replace it by a loop of 10 one-second sleeps, calling `g.check()` each second:

```
future<> slow(int i, seastar::gate &g) {
    std::cerr << "starting " << i << "\n";
    return do_for_each(boost::counting_iterator<int>(0),
                       boost::counting_iterator<int>(10),
            [&g] (int) {
        g.check();
        return sleep(std::chrono::seconds(1));
    }).finally([i] {
        std::cerr << "done " << i << "\n";
    });
}
```

Now, just one second after gate is closed (after the "starting 5" message is printed), all the `slow()` operations notice the gate was closed, and stop. As expected, the exception stops the `do_for_each()` loop, and the `finally()` continuation is performed so we see the "done" messages for all five operations.

8 Introducing shared-nothing programming

TODO: Explain in more detail Seastar's shared-nothing approach where the entire memory is divided up-front to cores, malloc/free and pointers only work on one core.
TODO: Introduce our shared_ptr (and lw_shared_ptr) and sstring and say the standard ones use locked instructions which are unnecessary when we assume these objects (like all others) are for a single thread. Our futures and continuations do the same.

9 More about Seastar's event loop

TODO: Mention the event loop (scheduler). remind that continuations on the same thread do not run in parallel, so do not need locks, atomic variables, etc (different threads shouldn't access the same data - more on that below). continuations obviously must not use blocking operations, or they block the whole thread.

Talk about polling that we currently do, and how today even sleep() or waiting for incoming connections or whatever, takes 100% of all CPUs.

10 Introducing Seastar's network stack

TODO: Mention the two modes of operation: Posix and native (i.e., take a L2 (Ethernet) interface (vhost or dpdk) and on top of it we built (in Seastar itself) an L3 interface (TCP/IP)).

We begin with a simple example of a TCP network server written in Seastar. This server repeatedly accepts connections on TCP port 1234, and returns an empty response:

```
#include "core/seastar.hh"
#include "core/reactor.hh"
#include "core/future-util.hh"
#include <iostream>

future<> f() {
    return do_with(listen(make_ipv4_address({1234})), [] (auto& listener) {
        return keep_doing([&listener] () {
            return listener.accept().then(
                [] (connected_socket s, socket_address a) {
                    std::cout << "Accepted connection from " << a << "\n";
                });
        });
    });
}
```

This code works as follows:

1. The `listen()` call creates a `server_socket` object, `listener`, which listens on TCP port 1234 (on any network interface).
2. To handle one connection, we call `listener`'s `accept()` method. This method returns a `future<connected_socket, socket_address>`, i.e., is eventually resolved with an incoming TCP connection from a client (`connected_socket`) and the client's IP address and port (`socket_address`).
3. To repeatedly accept new connections, we use the `keep_doing()` loop idiom. `keep_doing()` runs its lambda parameter over and over, starting the next iteration as soon as the future returned by the previous iteration completes. The iterations only stop if an exception is encountered. The future returned by `keep_doing()` itself completes only when the iteration stops (i.e., only on exception).
4. We use `do_with()` to ensure that the listener socket lives throughout the loop.

Output from this server looks like the following example:

```
$ ./a.out -c1
Accepted connection from 127.0.0.1:47578
Accepted connection from 127.0.0.1:47582
...
```

Note how we ran this Seastar application on a single thread, using the `-c1` option. Unintuitively, this options is actually necessary for running this program, as it will *not* work correctly if started on multiple threads. To understand why, we need to understand how Seastar's network stack works on multiple threads:

For optimum performance, Seastar's network stack is sharded just like Seastar applications are: each shard (thread) takes responsibility for a different subset of the connections. In other words, each incoming connection is directed to one of the threads, and after a connection is established, it continues to be handled on the same thread. But in our example, our server code only runs on the first thread, and the result is that only some of the connections (those which are randomly directed to thread 0) will get serviced properly, and other connections attempts will be ignored.

If you run the above example server immediately after killing the previous server, it often fails to start again, complaining that:

```
$ ./a.out -c1
program failed with uncaught exception: bind: Address already in use
```

This happens because by default, Seastar refuses to reuse the local port if there are any vestiges of old connections using that port. In our silly server, because the server is the side which first closes the connection, each connection lingers for a while in the "TIME_WAIT" state after being closed, and these prevent `listen()` on the same port from succeeding. Luckily, we can give listen an option to work despite these remaining `TIME_WAIT`. This option is analogous to `socket(7)`'s `SO_REUSEADDR` option:

```
    listen_options lo;
    lo.reuse_address = true;
    return do_with(listen(make_ipv4_address({1234}), lo), [] (auto& listener) {
```

Most servers will always turn on this `reuse_address` listen option. Stevens' book "Unix Network Programming" even says that "All TCP servers should specify this socket option to allow the server to be restarted". Therefore in the future Seastar should probably default to this option being on — even if for historic reasons this is not the default in Linux's socket API.

Let's advance our example server by outputting some canned response to each connection, instead of closing each connection immediately with an empty reply.

```
#include "core/seastar.hh"
#include "core/reactor.hh"
#include "core/future-util.hh"
#include <iostream>

const char* canned_response = "Seastar is the future!\n";

future<> f() {
    listen_options lo;
    lo.reuse_address = true;
    return do_with(listen(make_ipv4_address({1234}), lo), [] (auto& listener) {
        return keep_doing([&listener] () {
            return listener.accept().then(
                [] (connected_socket s, socket_address a) {
                    auto out = s.output();
                    return do_with(std::move(s), std::move(out),
                        [] (auto& s, auto& out) {
                            return out.write(canned_response).then([&out] {
                                return out.close();
                    });
            });
            });
        });
    });
}
```

The new part of this code begins by taking the `connected_socket`'s `output()`, which returns an `output_stream<char>` object. On this output stream `out` we can write our response using the `write()` method. The simple-looking `write()` operation is in fact a complex asynchronous operation behind the scenes, possibly causing multiple packets to be sent, retransmitted, etc., as needed. `write()` returns a future saying when it is ok to `write()` again to this output stream; This does not necessarily guarantee that the remote peer received all the data we sent it, but it guarantees that the output stream has enough buffer space to allow another write to begin.

After `write()`ing the response to `out`, the example code calls `out.close()` and waits for the future it returns. This is necessary, because `write()` attempts to batch writes so might not have yet written anything to the TCP stack at this point, and only when close() concludes can we be sure that all the data we wrote to the output stream has actually reached the TCP stack — and only at this point we may finally dispose of the `out` and `s` objects.

Indeed, this server returns the expected response:

```
$ telnet localhost 1234
...
Seastar is the future!
Connection closed by foreign host.
```

In the above example we only saw writing to the socket. Real servers will also want to read from the socket. The `connected_socket`'s `input()` method returns an `input_stream<char>` object which can be used to read from the socket. The simplest way to read from this stream is using the `read()` method which returns a future `temporary_buffer<char>`, containing some more bytes read from the socket — or an empty buffer when the remote end shut down the connection.

`temporary_buffer<char>` is a convenient and safe way to pass around byte buffers that are only needed temporarily (e.g., while processing a request). As soon as this object goes out of scope (by normal return, or exception), the memory it holds gets automatically freed. Ownership of buffer can also be transferred by `std::move()`ing it. We'll discuss `temporary_buffer` in more details in a later section.

Let's look at a simple example server involving both reads an writes. This is a simple echo server, as described in RFC 862: The server listens for connections from the client, and once a connection is established, any data received is simply sent back - until the client closes the connection.

```
#include "core/seastar.hh"
#include "core/reactor.hh"
#include "core/future-util.hh"

future<> handle_connection(connected_socket s, socket_address a) {
    auto out = s.output();
    auto in = s.input();
    return do_with(std::move(s), std::move(out), std::move(in),
        [] (auto& s, auto& out, auto& in) {
            return repeat([&out, &in] {
                return in.read().then([&out] (auto buf) {
                    if (buf) {
                        return out.write(std::move(buf)).then([] {
                            return stop_iteration::no;
                        });
                    } else {
                        return make_ready_future<stop_iteration>(stop_iteration::yes);
                    }
                });
            }).then([&out] {
                return out.close();
            });
        });
}

future<> f() {
    listen_options lo;
    lo.reuse_address = true;
    return do_with(listen(make_ipv4_address({1234}), lo), [] (auto& listener) {
        return keep_doing([&listener] () {
            return listener.accept().then(
                [] (connected_socket s, socket_address a) {
                    // Note we ignore, not return, the future returned by
                    // handle_connection(), so we do not wait for one
                    // connection to be handled before accepting the next one.
                    handle_connection(std::move(s), std::move(a));
                });
        });
    });
}
```

The main function `f()` loops accepting new connections, and for each connection calls `handle_connection()` to handle this connection. Our `handle_connection()` returns a future saying when handling this connection completed, but importantly, we do ***not*** wait for this future: Remember that `keep_doing` will only start the next iteration when the future returned by the previous iteration is resolved. Because we want to allow parallel ongoing connections, we don't want the next `accept()` to wait until the previously accepted connection was closed. So we call `handle_connection()` to start the handling of the connection, but return nothing from the continuation, which resolves that future immediately, so `keep_doing` will continue to the next `accept()`.

This demonstrates how easy it is to run parallel *fibers* (chains of continuations) in Seastar - When a continuation runs an asynchronous function but ignores the future it returns, the asynchronous operation continues in parallel, but never waited for.

It is often a mistake to silently ignore an exception, so if the future we're ignoring might resolve with an except, it is recommended to handle this case, e.g. using a `handle_exception()` continuation. In our case, a failed connection is fine (e.g., the client might close its connection will we're sending it output), so we did not bother to handle the exception.

The `handle_connection()` function itself is straightforward — it repeatedly calls `read()` read on the input stream, to receive a `temporary_buffer` with some data, and then moves this temporary buffer into a `write()` call on the output stream. The buffer will eventually be freed, automatically, when the `write()` is done with it. When `read()` eventually returns an empty buffer signifying the end of input, we stop `repeat`'s iteration by returning a `stop_iteration::yes`.

11 Sharded servers

TODO: show how to fix the network server to work on multiple threads

12 Shutting down cleanly

Handling interrupt, shutting down services, etc.

13 User-defined command-line options

14 Debugging a Seastar program

handle SIGUSR1 pass noprint
handle SIGALRM pass noprint

15 Promise objects

As we already defined above, An **asynchronous function**, also called a **promise**, is a function which returns a future and arranges for this future to be eventually resolved. As we already saw, an asynchronous function is usually written in terms of other asynchronous functions, for example we saw the function `slow()` which waits for the existing asynchronous function `sleep()` to complete, and then returns 3:

```
future<int> slow() {
    using namespace std::chrono_literals;
    return sleep(100ms).then([] { return 3; });
}
```

The most basic building block for writing promises is the **promise object**, an object of type `promise<T>`. A `promise<T>` has a method `future<T> get_future()` to returns a future, and a method `set_value(T)`, to resolve this future. An asynchronous function can create a promise object, return its future, and the `set_value` method to be eventually called - which will finally resolve the future it returned.

CONTINUE HERE. write an example, e.g., something which writes a message every second, and after 10 messages, completes the future.