Asynchronous Programming with Seastar

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

Contents

1 Introduction

Seastar, which we introduce in this document, is a C++ library for writing highly efficient complex server applications on modern multi-core machines.

Traditionally, the programming languages libraries and frameworks used for writing server applications have been divided into two distinct camps: those focusing on efficiency, and those focusing on complexity. Some frameworks are extremely efficient and yet allow building only simple applications (e.g., DPDK allows applications which process packets individually), while other frameworks allow building extremely complex applications, at the cost of run-time efficiency. Seastar is our attempt to get the best of both worlds: To create a library which allows building highly complex server applications, and yet achieve optimal performance.

The inspiration and first use case of Seastar was Scylla, a rewrite of Apache Cassandra. Cassandra is a very complex application, and yet, with Seastar we were able to re-implement it with as much as 10-fold throughput increase, as well as significantly lower and more consistent latencies.

Seastar offers a complete asynchronous programming framework, which uses two concepts - **futures** and **continuations** - to uniformly represent, and handle, every type of asynchronous event, including network I/O, disk I/O, and complex combinations of other events.

Since modern multi-core and multi-socket machines have steep penalties for sharing data between cores (atomic instructions, cache line bouncing and memory fences), Seastar programs use the share-nothing programming model, i.e., the available memory is divided between the cores, each core works on data in its own part of memory, and communication between cores happens via explicit message passing (which itself happens using the SMP's shared memory hardware, of course).

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 finish handling one request before starting 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 ready to handle other connections.

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 starts 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 process 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: somewhere in the intro, talk about low tail latency and the features that the framework needs to support it (scheduler, io scheduler, and because of share-nothing, no core overcommit - no other apps on the same cpu).

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

1.3 Comparison with other asynchronous programming frameworks

TODO: Mention how other asynchronous programming techniques and their pros and cons: libevent, CSP with channels (and newer golang), and futures and continuations (mention where these are used).

Libraries like `libevent` made event-driven programming somewhat easier more approachable and portable,

1.4 A taste of Seastar (?)

TODO: Give a taste of Seastar - a very short working Seastar super-efficient http server, or something similar. Alternatively, build such code throughout the tutorial?

2 Getting started

The simplest Seastar program is this:

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

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

As shown in this example, all Seastar functions and types live in the "`seastar`" namespace. An user can either type this namespace prefix every time, or use shortcuts like "`using seastar::app_template`" or even "`using namespace seastar`" to avoid typing this prefix. We generally recommend to use the namespace prefixes `seastar` and `std` explicitly, and will will follow this style in all the examples below.

To compile this program, first make sure you have downloaded, built, and optionally installed Seastar, and put the above program in a source file anywhere you want, let's call the file `getting-started.cc`.

Linux's pkg-config is one way for easily determining the compilation and linking parameters needed for using various libraries - such as Seastar. For example, if Seastar was built in the directory `$SEASTAR` but not installed, one can compile `getting-started.cc` with it using the command:

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

If Seastar *was* installed, the `pkg-config` command line is even shorter:

```
c++ getting-started.cc `pkg-config --cflags --libs seastar`
```

Alternatively, one can easily build a Seastar program with CMake. Given the following `CMakeLists.txt`

```
cmake_minimum_required (VERSION 3.5)

project (SeastarExample)

find_package (Seastar REQUIRED)
```

```
add_executable (example
  getting-started.cc)

target_link_libraries (example
  PRIVATE Seastar::seastar)
```

you can compile the example with the following commands:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

The program now runs as expected:

```
$ ./example
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 `seastar::smp::count` which is the number of started threads:

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

int main(int argc, char** argv) {
    seastar::app_template app;
    app.run(argc, argv, [] {
            std::cout << seastar::smp::count << "\n";
            return seastar::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 <seastar/core/app-template.hh>
#include <seastar/core/reactor.hh>
#include <iostream>
#include <stdexcept>

int main(int argc, char** argv) {
    seastar::app_template app;
    try {
        app.run(argc, argv, [] {
            std::cout << seastar::smp::count << "\n";
            return seastar::make_ready_future<>();
        });
    } catch(...) {
        std::cerr << "Failed to start application: "
                  << std::current_exception() << "\n";
        return 1;
    }
    return 0;
}
```

```
$ ./a.out -c5
Couldn't start application: std::runtime_error (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 certain reservation left for the OS (defaulting to the maximum of 1.5G or 7% of total memory) 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: std::runtime_error (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**, a function which returns a future and arranges for this future to be eventually resolved. Because asynchrnous functions *promise* to eventually resolve the future which they returned, asynchronous functions are sometimes called "promises"; But we will avoid this term because it tends to confuse more than it explains.

One simple example of an asynchronous function 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 <seastar/core/app-template.hh>
#include <seastar/core/sleep.hh>
#include <iostream>

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

In this example we see us getting a future from `seastar::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 <seastar/core/app-template.hh>
#include <seastar/util/log.hh>
#include <iostream>
#include <stdexcept>

extern seastar::future<> f();

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

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

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

seastar::future<> f() {
    std::cout << "Sleeping... " << std::flush;
    using namespace std::chrono_literals;
    return seastar::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 <seastar/core/sleep.hh>
#include <iostream>

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

The `(void)` cast in the above example was necessary to avoid a compilation error when discarding the future returned by `then()`: Most code needs to do something with the future a function returns - e.g., wait for it, save it in a variable, attach a continuation to it, or return it to a caller. Accidentally discarding a future often causes hard-to-debug issues, such as freeing an object while still being used by an asynchronous function (we'll discuss this issue in the Lifetime management section). Because accidentally discarding a `future` is dangerous, Seastar marks them `[[nodiscard]]`, asking the compiler to warn when a `future` returned by a function is discarded. Casting the future to `(void)` tells the compiler that the discarding was intentional, and it should no longer warn here. In the above example, we really did want not to wait for these two futures, thereby basically putting their work "in the background", so we discarded them with `(void)`. In this simple example, this was good enough, but more realistic code usually needs to be more careful when putting work in the background: You usually need to handle errors coming out of this work, limit the amount of background work so it doesn't fill all of memory, and be able to wait until the background work is done. We'll describe mechanisms to do those things - `handle_exception()`, `semaphore` and `gate` respectively - later in the section Discarding futures, redux.

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

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

seastar::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 <seastar/core/future.hh>
#include <iostream>

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

seastar::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 <seastar/core/sleep.hh>
#include <iostream>

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

seastar::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...), 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 related 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 usually a mistake, and can lead to serious bugs. For example, if in the above example we captured a reference to i, instead of copying it,

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

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

Using capture-by-*move* in continuations is also 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 automatically deletes it. How do we use unique_ptr in a continuation? The following won't work:

```
seastar::future<int> slow_do_something(std::unique_ptr<T> obj) {
    using namespace std::chrono_literals;
    // The following line won't compile...
    return seastar::sleep(10ms).then([obj] () mutable { return do_something(std::move(obj)); });
}
```

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:

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

The extra `() mutable` syntax was needed here because by default when C++ captures a value (in this case, the value of std::move(obj)) into a lambda, it makes this value read-only, so our lambda cannot, in this example, move it again. Adding `mutable` removes this artificial restriction.

5.2 Chaining continuations

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

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 Exceptions vs. exceptional futures

An asynchronous function can fail in one of two ways: It can fail immediately, by throwing an exception, or it can return a future which will eventually fail (resolve to an exception). These two modes of failure appear similar to the uninitiated, but behave differently when attempting to handle exceptions using `finally()`, `handle_exception()`, or `then_wrapped()`. For example, consider the code:

```
#include <seastar/core/future.hh>
#include <iostream>
#include <exception>

class my_exception : public std::exception {
    virtual const char* what() const noexcept override { return "my exception"; }
};

seastar::future<> fail() {
    return seastar::make_exception_future<>(my_exception());
}

seastar::future<> f() {
    return fail().finally([] {
        std::cout << "cleaning up\n";
    });
}
```

This code will, as expected, print the "cleaning up" message - the asynchronous function `fail()` returns a future which resolves to a failure, and the `finally()` continuation is run despite this failure, as expected.

Now consider that in the above example we had a different definition for `fail()`:

```
seastar::future<> fail() {
    throw my_exception();
}
```

Here, `fail()` does not return a failing future. Rather, it fails to return a future at all! The exception it throws stops the entire function `f()`, and the `finally()` continuation does not not get attached to the future (which was never returned), and will never run. The "cleaning up" message is not printed now.

We recommend that to reduce the chance for such errors, asynchronous functions should always return a failed future rather than throw an actual exception. If the asynchronous function calls another function *before* returning a future, and that second function might throw, it should use `try/catch` to catch the exception and convert it into a failed future:

```
void inner() {
    throw my_exception();
}
seastar::future<> fail() {
    try {
        inner();
    } catch(...) {
        return seastar::make_exception_future(std::current_exception());
    }
    return seastar::make_ready_future<>();
}
```

Here, `fail()` catches the exception thrown by `inner()`, whatever it might be, and returns a failed future with that failure. Written this way, the `finally()` continuation will be reached, and the "cleaning up" message printed.

> Despite this recommendation that asynchronous functions avoid throwing, some asynchronous functions do throw exceptions in addition to returning exceptional futures. A common example are functions which allocate memory and throw `std::bad_alloc` when running out of memory, instead of returning a future. The `future<> seastar::semaphore::wait()` method is one such function: It returns a future which may be exceptional if the semaphore was `broken()` or the wait timed out, but may also *throw* an exception when failing to allocate memory it needs to hold the list of waiters. Therefore, unless a function — including asynchronous functions — is explicitly tagged "`noexcept`", the application should be prepared to handle exceptions thrown from it. In modern C++, code usually uses RAII to be exception-safe without sprinkling it with `try/catch`. `seastar::defer()` is a RAII-based idiom that ensures that some cleanup code is run even if an exception is thrown.

Seastar has a convenient generic function, `futurize_apply()`, which can be useful here. `futurize_apply(func, args...)` runs a function which may return either a future value or an immediate value, and in both cases convert the result into a future value. `futurize_apply()` also converts an immediate exception thrown by the function, if any, into a failed future, just like we did above. So using `futurize_apply()` we can make the above example work even if `fail()` did throw exceptions:

```
seastar::future<> fail() {
    throw my_exception();
}
seastar::future<> f() {
    return seastar::futurize_apply(fail).finally([] {
        std::cout << "cleaning up\n";
    });
}
```

Note that most of this discussion becomes moot if the risk of exception is inside a *continuation*. Consider the following code:

```
seastar::future<> f() {
    return seastar::sleep(1s).then([] {
        throw my_exception();
    }).finally([] {
        std::cout << "cleaning up\n";
    });
}
```

Here, the lambda function of the first continuation does throw an exception instead of returning a failed future. However, we do *not* have the same problem as before, which only happened because an asynchronous function threw an exception *before* returning a valid future. Here, `f()` does return a valid future immediately - the failure will only be known later, after `sleep()` resolves. The message in `finally()` will be printed. Under the hood, the methods which attach continuations (such as `then()` and `finally()`) run the continuation functions using `futurize_apply()`, so continuation functions may return immediate values or, in this case, throw an immediate exception, and still work properly.

7 Lifetime management

An asynchronous function starts an operation which may continue long after the function returns: The function itself returns a `future<T>` almost immediately, but it may take a while until this future is resolved.

When such an asynchronous operation needs to operate on existing objects, or to use temporary objects, we need to worry about the *lifetime* of these objects: We need to ensure that these objects do not get destroyed before the asynchronous function completes (or it will try to use the freed object and malfunction or crash), and to also ensure that the object finally get destroyed when it is no longer needed (otherwise we will have a memory leak). Seastar offers a variety of mechanisms for safely and efficiently keeping objects alive for the right duration. In this section we will explore these mechanisms, and when to use each mechanism.

7.1 Passing ownership to continuation

The most straightforward way to ensure that an object is alive when a continuation runs and is destroyed afterwards is to pass its ownership to the continuation. When continuation *owns* the object, the object will be kept until the continuation runs, and will be destroyed as soon as the continuation is not needed (i.e., it may have run, or skipped in case of exception and `then()` continuation).

We already saw above that the way for a continuation to get ownership of an object is through *capturing*:

```
seastar::future<> slow_incr(int i) {
    return seastar::sleep(10ms).then([i] { return i + 1; });
}
```

Here the continuation captures the value of `i`. In other words, the continuation includes a copy of `i`. When the continuation runs 10ms later, it will have access to this value, and as soon as the continuation finishes its object is destroyed, together with its captured copy of `i`. The continuation owns this copy of `i`.

Capturing by value as we did here - making a copy of the object we need in the continuation - is useful mainly for very small objects such as the integer in the previous example. Other objects are expensive to copy, or sometimes even cannot be copied. For example, the following is **not** a good idea:

```
seastar::future<> slow_op(std::vector<int> v) {
    // this makes another copy of v:
    return seastar::sleep(10ms).then([v] { /* do something with v */ });
}
```

This would be inefficient - as the vector `v`, potentially very long, will be copied and the copy will be saved in the continuation. In this example, there is no reason to copy `v` - it was anyway passed to the function by value and will not be used again after capturing it into the continuation, as right after the capture, the function returns and destroys its copy of `v`.

For such cases, C++14 allows *moving* the object into the continuation:

```
seastar::future<> slow_op(std::vector<int> v) {
    // v is not copied again, but instead moved:
    return seastar::sleep(10ms).then([v = std::move(v)] { /* do something with v */ });
}
```

Now, instead of copying the object `v` into the continuation, it is *moved* into the continuation. The C++11-introduced move constructor moves the vector's data into the continuation and clears the original vector. Moving is a quick operation - for a vector it only requires copying a few small fields such as the pointer to the data. As before, once the continuation is dismissed the vector is destroyed - and its data array (which was moved in the move operation) is finally freed.

TODO: talk about temporary_buffer as an example of an object designed to be moved in this way.

In some cases, moving the object is undesirable. For example, some code keeps references to an object or one of its fields and the references become invalid if the object is moved. In some complex objects, even the move constructor is slow. For these cases, C++ provides the useful wrapper `std::unique_ptr<T>`. A `unique_ptr<T>` object owns an object of type `T` allocated on the heap. When a `unique_ptr<T>` is moved, the object of type T is not touched at all - just the pointer to it is moved. An example of using `std::unique_ptr<T>` in capture is:

```
seastar::future<> slow_op(std::unique_ptr<T> p) {
    return seastar::sleep(10ms).then([p = std::move(p)] { /* do something with *p */ });
}
```

`std::unique_ptr<T>` is the standard C++ mechanism for passing unique ownership of an object to a function: The object is only owned by one piece of code at a time, and ownership is transferred by moving the `unique_ptr` object. A `unique_ptr` cannot be copied: If we try to capture p by value, not by move, we will get a compilation error.

7.2 Keeping ownership at the caller

The technique we described above - giving the continuation ownership of the object it needs to work on - is powerful and safe. But often it becomes hard and verbose to use. When an asynchronous operation involves not just one continuation but a chain of continations that each needs to work on the same object, we need to pass the ownership of the object between each successive continuation, which can become inconvenient. It is especially inconvenient when we need to pass the same object into two seperate asynchronous functions (or continuations) - after we move the object into one, the object needs to be returned so it can be moved again into the second. E.g.,

```
seastar::future<> slow_op(T o) {
    return seastar::sleep(10ms).then([o = std::move(o)] {
        // first continuation, doing something with o
        ...
        // return o so the next continuation can use it!
        return std::move(o);
    }).then([](T o)) {
        // second continuation, doing something with o
        ...
    });
}
```

This complexity arises because we wanted asynchronous functions and continuations to take the ownership of the objects they operated on. A simpler approach would be to have the *caller* of the asynchronous function continue to be the owner of the object, and just pass *references* to the object to the various other asynchronous functions and continuations which need the object. For example:

```
seastar::future<> slow_op(T& o) {           // <-- pass by reference
    return seastar::sleep(10ms).then([&o] {// <-- capture by reference
        // first continuation, doing something with o
        ...
    }).then([&o]) {                        // <-- another capture by reference
        // second continuation, doing something with o
        ...
    });
}
```

This approach raises a question: The caller of `slow_op` is now responsible for keeping the object `o` alive while the asynchronous code started by `slow_op` needs this object. But how will this caller know how long this object is actually needed by the asynchronous operation it started?

The most reasonable answer is that an asynchronous function may need access to its parameters until the future it returns is resolved - at which point the asynchronous code completes and no longer needs access to its parameters. We therefore recommend that Seastar code adopt the following convention:

> **Whenever an asynchronous function takes a parameter by reference, the caller must ensure that the referred object lives until the future returned by the function is resolved.**

Note that this is merely a convention suggested by Seastar, and unfortunately nothing in the C++ language enforces it. C++ programmers in non-Seastar programs often pass large objects to functions as a const reference just to avoid a slow copy, and assume that the called function will *not* save this reference anywhere. But in Seastar code, that is a dangerous practice because even if the asynchronous function did not intend to save the reference anywhere, it may end up doing it implicitly by passing this reference to another function and eventually capturing it in a continuation.

> It would be nice if future versions of C++ could help us catch incorrect uses of references. Perhaps we could have a tag for a special kind of reference, an "immediate reference" which a function can use use immediately (i.e, before returning a future), but cannot be captured into a continuation.

With this convention in place, it is easy to write complex asynchronous functions functions like `slow_op` which pass the object around, by reference, until the asynchronous operation is done. But how does the caller ensure that the object lives until the returned future is resolved? The following is *wrong*:

```
seastar::future<> f() {
    T obj; // wrong! will be destroyed too soon!
    return slow_op(obj);
}
```

It is wrong because the object `obj` here is local to the call of `f`, and is destroyed as soon as `f` returns a future - not when this returned future is resolved! The correct thing for a caller to do would be to create the object `obj` on the heap (so it does not get destroyed as soon as `f` returns), and then run `slow_op(obj)` and when that future resolves (i.e., with `.finally()`), destroy the object.

Seastar provides a convenient idiom, `do_with()` for doing this correctly:

```
seastar::future<> f() {
    return seastar::do_with(T(), [] (auto& obj) {
        // obj is passed by reference to slow_op, and this is fine:
        return slow_op(obj);
    }
}
```

`do_with` will *do* the given function *with* the given object alive.

`do_with` saves the given object on the heap, and calls the given lambda with a reference to the new object. Finally it ensures that the new object is destroyed after the returned future is resolved. Usually, do_with is given an *rvalue*, i.e., an unnamed temporary object or an `std::move()`ed object, and `do_with` moves that object into its final place on the heap. `do_with` returns a future which resolves after everything described above is done (the lambda's future is resolved and the object is destroyed).

For convenience, `do_with` can also be given multiple objects to hold alive. For example here we create two objects and hold alive them until the future resolves:

```
seastar::future<> f() {
    return seastar::do_with(T1(), T2(), [] (auto& obj1, auto& obj2) {
        return slow_op(obj1, obj2);
    }
}
```

While `do_with` can the lifetime of the objects it holds, if the user accidentally makes copies of these objects, these copies might have the wrong lifetime. Unfortunately, a simple typo like forgetting an "&" can cause such accidental copies. For example, the following code is broken:

```
seastar::future<> f() {
    return seastar::do_with(T(), [] (T obj) { // WRONG: should be T&, not T
        return slow_op(obj);
    }
}
```

In this wrong snippet, `obj` is mistakenly not a reference to the object which `do_with` allocated, but rather a copy of it - a copy which is destroyed as soon as the lambda function returns, rather than when the future it returns resolved. Such code will most likely crash because the object is used after being freed. Unfortunately the compiler will not warn about such mistakes. Users should get used to always using the type "auto&" with `do_with` - as in the above correct examples - to reduce the chance of such mistakes.

For the same reason, the following code snippet is also wrong:

```
seastar::future<> slow_op(T obj); // WRONG: should be T&, not T
seastar::future<> f() {
    return seastar::do_with(T(), [] (auto& obj) {
        return slow_op(obj);
    }
}
```

Here, although `obj` was correctly passed to the lambda by reference, we later acidentally passed `slow_op()` a copy of it (because here `slow_op` takes the object by value, not by reference), and this copy will be destroyed as soon as `slow_op` returns, not waiting until the returned future resolves.

When using `do_with`, always remember it requires adhering to the convention described above: The asynchronous function which we call inside `do_with` must not use the objects held by `do_with` *after* the returned future is resolved. It is a serious use-after-free bug for an asynchronous function to return a future which resolves while still having background operations using the `do_with()`ed objects.

In general, it is rarely a good idea for an asynchronous function to resolve while leaving behind background operations - even if those operations do not use the `do_with()`ed objects. Background operations that we do not wait for may cause us to run out of memory (if we don't limit their number) and make it difficult to shut down the application cleanly.

7.3 Sharing ownership (reference counting)

In the beginning of this chapter, we already noted that capturing a copy of an object into a continuation is the simplest way to ensure that the object is alive when the continuation runs and destoryed afterwards. However, complex objects are often expensive (in time and memory) to copy. Some objects cannot be copied at all, or are read-write and the continuation should modify the original object, not a new copy. The solution to all these issues are **reference counted**, a.k.a. **shared** objects:

A simple example of a reference-counted object in Seastar is a `seastar::file`, an object holding an open file object (we will introduce `seastar::file` in a later section). A `file` object can be copied, but copying does not involve copying the file descriptor (let alone the file). Instead, both copies point to the same open file, and a reference count is increased by 1. When a file object is destroyed, the file's reference count is decreased by one, and only when the reference count reaches 0 the underlying file is actually closed.

The fact that `file` objects can be copied very quickly and all copies actually point to the same file, make it very convinient to pass them to asynchronous code; For example,

```
seastar::future<uint64_t> slow_size(file f) {
    return seastar::sleep(10ms).then([f] {
        return f.size();
    });
    // NOTE: something is wrong here! This will be explained below!
}
```

Note how calling `slow_size` is as simple as calling `slow_size(f)`, passing a copy of `f`, without needing to do anything special to ensure that `f` is only destroyed when no longer needed. That simply happens naturally when nothing refers to `f` any more.

However, there is one complication. The above example is actually wrong, as the comment at the end of the function suggested. The problem is that the `f.size()` call started an asynchronous operation on `f` (the file's size may be stored on disk, so not immediately available) and yet at this point nothing is holding a copy of `f`... The method call does not increment the reference count of the object even if it an asynchronous method. (Perhaps this something we should rethink?)

So we need to ensure that something does hold on to another copy of `f` until the asynchronous method call completes. This is how we typically do it:

```
seastar::future<uint64_t> slow_size(file f) {
    return seastar::sleep(10ms).then([f] {
        return f.size().finally([f] {});
    });
}
```

What we see here is that `f` is copied not only to the continuation which runs `f.size()`, but also into a continuation (a `finally`) which will run after it. So as long as `f.size()` does not complete, that second continuation holds `f` alive. Note how the second continuation seems to have no code (just a {}). But the important thing is that the compiler automatically adds to it code to destroy its copy of `f` (and potentially the entire file if this reference count went down to 0).

The reference counting has a run-time cost, but it is usually very small; It is important to remember that Seastar objects are always used by a single CPU only, so the reference-count increment and decrement operations are not the slow atomic operations often used for reference counting, but just regular CPU-local integer operations. Moreover, judicious use of `std::move()` and the compiler's optimizer can reduce the number of unnecessary back-and-forth increment and decrement of the reference count.

C++11 offers a standard way of creating reference-counted shared objects - using the template `std::shared_ptr<T>`. A `shared_ptr` can be used to wrap any type into a reference-counted shared object like `seastar::file` above. However, the standard `std::shared_ptr` was designed with multi-threaded applications in mind so it uses slow atomic increment/decrement operations for the reference count which we already noted is unnecessary in Seastar. For this reason Seastar offers its own single-threaded implementation of this template, `seastar::shared_ptr<T>`. It is similar to `std::shared_ptr<T>` except no atomic operations are used.

Additionally, Seastar also provides an even lower overhead variant of `shared_ptr`: `seastar::lw_shared_ptr<T>`. The full-featured `shared_ptr` is complicated by the need to support polymorphic types correctly (a shared object created of one class, and accessed through a pointer to a base class). It makes `shared_ptr` need to add two words to the shared object, and two words to each `shared_ptr` copy. The simplified `lw_shared_ptr` - which does **not** support polymorphic types - adds just one word in the object (the reference count) and each copy is just one word - just like copying a regular pointer. For this reason, the light-weight `seastar::lw_shared_ptr<T>` should be preferered when possible (`T` is not a polymorphic type), otherwise `seastar::shared_ptr<T>`. The slower `std::shared_ptr<T>` should never be used in sharded Seastar applications.

7.4 Saving objects on the stack

Wouldn't it be convenient if we could save objects on a stack just like we normally do in synchronous code? I.e., something like:

```
int i = ...;
seastar::sleep(10ms).get();
return i;
```

Seastar allows writing such code, by using a `seastar::thread` object which comes with its own stack. A complete example using a `seastar::thread` might look like this:

```
seastar::future<> slow_incr(int i) {
    return seastar::async([i] {
        seastar::sleep(10ms).get();
        // We get here after the 10ms of wait, i is still available.
        return i + 1;
    });
}
```

We present `seastar::thread`, `seastar::async()` and `seastar::future::get()` in the seastar::thread section.

8 Advanced futures

8.1 Futures and interruption

TODO: A future, e.g., sleep(10s) cannot be interrupted. So if we need to, the promise needs to have a mechanism to interrupt it. Mention pipe's close feature, semaphore stop feature, etc. ## 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

9 Fibers

Seastar continuations are normally short, but often chained to one another, so that one continuation does a bit of work and then schedules another continuation for later. Such chains can be long, and often even involve loopings - see the following section, "Loops". We call such chains "fibers" of execution.

These fibers are not threads - each is just a string of continuations - but they share some common requirements with traditional threads. For example, we want to avoid one fiber getting starved while a second fiber continuously runs its continuations one after another. As another example, fibers may want to communicate - e.g., one fiber produces data that a second fiber consumes, and we wish to ensure that both fibers get a chance to run, and that if one stops prematurely, the other doesn't hang forever.

TODO: Mention fiber-related sections like loops, semaphores, gates, pipes, etc. # Loops TODO: do_until, repear 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).

TODO: See seastar commit "input_stream: Fix possible infinite recursion in consume()" for an example on why recursion is a possible, but bad, replacement for repeat(). See also my comment on https://groups.google.com/d/msg/seastar-dev/CUkLVBwva3Y/3DKGw-9aAQAJ on why Seastar's iteration primitives should be used over tail call optimization. # when_all: Waiting for multiple futures Above we've seen `parallel_for_each()`, which starts a number of asynchronous operations, and then waits for all to complete. Seastar has another idiom, `when_all()`, for waiting for several already-existing futures to complete.

The first variant of `when_all()` is variadic, i.e., the futures are given as separate parameters, the exact number of which is known at compile time. The individual futures may have different types. For example,

```
#include <seastar/core/sleep.hh>

future<> f() {
    using namespace std::chrono_literals;
    future<int> slow_two = sleep(2s).then([] { return 2; });
    return when_all(sleep(1s), std::move(slow_two),
                    make_ready_future<double>(3.5)
           ).discard_result();
}
```

This starts three futures - one which sleeps for one second (and doesn't return anything), one which sleeps for two seconds and returns the integer 2, and one which returns the double 3.5 immediately - and then waits for them. The `when_all()` function returns a future which resolves as soon as all three futures resolves, i.e., after two seconds. This future also has a value, which we shall explain below, but in this example, we simply waited for the future to resolve and discarded its value.

Note that `when_all()` accept only rvalues, which can be temporaries (like the return value of an asynchronous function or `make_ready_future`) or an `std::move()`'ed variable holding a future.

The future returned by `when_all()` resolves to a tuple of futures which are already resolved, and contain the results of the three input futures. Continuing the above example,

```
future<> f() {
    using namespace std::chrono_literals;
    future<int> slow_two = sleep(2s).then([] { return 2; });
    return when_all(sleep(1s), std::move(slow_two),
                    make_ready_future<double>(3.5)
           ).then([] (auto tup) {
            std::cout << std::get<0>(tup).available() << "\n";
            std::cout << std::get<1>(tup).get0() << "\n";
            std::cout << std::get<2>(tup).get0() << "\n";
    });
}
```

The output of this program (which comes after two seconds) is `1, 2, 3.5`: the first future in the tuple is available (but has no value), the second has the integer value 2, and the third a double value 3.5 - as expected.

One or more of the waited futures might resolve in an exception, but this does not change how `when_all()` works: It still waits for all the futures to resolve, each with either a value or an exception, and in the returned tuple some of the futures may contain an exception instead of a value. For example,

```
future<> f() {
    using namespace std::chrono_literals;
    future<> slow_success = sleep(1s);
    future<> slow_exception = sleep(2s).then([] { throw 1; });
    return when_all(std::move(slow_success), std::move(slow_exception)
           ).then([] (auto tup) {
            std::cout << std::get<0>(tup).available() << "\n";
            std::cout << std::get<1>(tup).failed() << "\n";
            std::get<1>(tup).ignore_ready_future();
    });
}
```

Both futures are `available()` (resolved), but the second has `failed()` (resulted in an exception instead of a value). Note how we called `ignore_ready_future()` on this failed future, because silently ignoring a failed future is considered a bug, and will result in an "Exceptional future ignored" error message. More typically, an application will log the failed future instead of ignoring it.

The above example demonstrate that `when_all()` is inconvenient and verbose to use properly. The results are wrapped in a tuple, leading to verbose tuple syntax, and uses ready futures which must all be inspected individually for an exception to avoid error messages.

So Seastar also provides an easier to use `when_all_succeed()` function. This function too returns a future which resolves when all the given futures have resolved. If all of them succeeded, it passes the resulting values to continuation, without wrapping them in futures or a tuple. If, however, one or more of the futures failed, `when_all_succeed()` resolves to a failed future, containing the exception from one of the failed futures. If more than one of the given future failed, one of those will be passed on (it is unspecified which one is chosen), and the rest will be silently ignored. For example,

```
using namespace seastar;
future<> f() {
    using namespace std::chrono_literals;
    return when_all_succeed(sleep(1s), make_ready_future<int>(2),
                    make_ready_future<double>(3.5)
            ).then([] (int i, double d) {
        std::cout << i << " " << d << "\n";
    });
}
```

Note how the integer and double values held by the futures are conveniently passed, individually (without a tuple) to the continuation. Since `sleep()` does not contain a value, it is waited for, but no third value is passed to the continuation. That also means that if we `when_all_succeed()` on several `future<>` (without a value), the result is also a `future<>`:

```
using namespace seastar;
future<> f() {
    using namespace std::chrono_literals;
    return when_all_succeed(sleep(1s), sleep(2s), sleep(3s));
}
```

This example simply waits for 3 seconds (the maximum of 1, 2 and 3 seconds).

An example of `when_all_succeed()` with an exception:

```
using namespace seastar;
future<> f() {
    using namespace std::chrono_literals;
    return when_all_succeed(make_ready_future<int>(2),
                    make_exception_future<double>("oops")
            ).then([] (int i, double d) {
        std::cout << i << " " << d << "\n";
    }).handle_exception([] (std::exception_ptr e) {
        std::cout << "exception: " << e << "\n";
    });
}
```

In this example, one of the futures fails, so the result of `when_all_succeed` is a failed future, so the normal continuation is not run, and the `handle_exception()` continuation is done.

TODO: also explain `when_all` and `when_all_succeed` for vectors.

10 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.

10.1 Limiting parallelism with semaphores

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

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

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, either successfully or with exception, the counter is increased back by one (`signal(1)`). This way, when 100 operations have already started their work and have not yet finished, the 101st operation will wait, until one of the ongoing operations finishes and returns a unit to the semaphore. This ensures that at each time we have at most 100 concurrent `slow()` operations running in the above code.

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.

Luckily, the above code happens to be exception safe: `limit.wait(1)` can throw an exception when it runs out of memory (keeping a list of waiters), and in that case the semaphore counter is not decreased but the continuations below are not run so it is not increased either. `limit.wait(1)` can also return an exceptional future when the semaphore is *broken* (we'll discuss this later) but in that case the extra `signal()` call is ignored. Finally, `slow()` may also throw, or return an exceptional future, but the `finally()` ensures the semaphore is still increased.

However, as the application code becomes more complex, it becomes harder to ensure that we never forget to call `signal()` after the operation is done, regardless of which code path or exceptions happen. As an example of what might go wrong, consider the following *buggy* code snippet, which differs subtly from the above one, and also appears, on first sight, to be correct:

```
seastar::future<> g() {
    static thread_local seastar::semaphore limit(100);
    return limit.wait(1).then([] {
        return slow().finally([] { limit.signal(1); });
    });
}
```

But this version is **not** exception safe: Consider what happens if `slow()` throws an exception before returning a future (this is different from `slow()` returning an exceptional future - we discussed this difference in the section about exception handling). In this case, we decreased the counter, but the `finally()` will never be reached, and the counter will never be increased back. There is a way to fix this code, by replacing the call to `slow()` with `seastar::futurize_apply(slow)`. But the point we're trying to make here is not how to fix buggy code, but rather that by using the separate `semaphore::wait()` and `semaphore::signal()` functions, you can very easily get things wrong.

For exception safety, in C++ it is generally not recommended to have separate resource acquisition and release functions. Instead, C++ offers safer mechanisms for acquiring a resource (in this case seamphore units) and later releasing it: lambda functions, and RAII ("resource acquisition is initialization"):

The lambda-based solution is a function `seastar::with_semaphore()` which is a shortcut for the code in the examples above:

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

`with_semaphore()`, like the earlier code snippets, waits for the given number of units from the semaphore, then runs the given lambda, and when the future returned by the lambda is resolved, `with_semaphore()` returns back the units to the semaphore. `with_semaphore()` returns a future which only resolves after all these steps are done.

The function `seastar::get_units()` is more general. It provides an exception-safe alternative to `seastar::semaphore`'s separate `wait()` and `signal()` methods, based on C++'s RAII philosophy: The function returns an opaque units object, which while held, keeps the semaphore's counter decreased - and as soon as this object is destructed, the counter is increased back. With this interface you cannot forget to increase the counter, or increase it twice, or increase without decreasing: The counter will always be decreased once when the units object is created, and if that succeeded, increased when the object is destructed. When the units object is moved into a continuation, no matter how this continuation ends, when the continuation is destructed, the units object is destructed and the units are returned to the semaphore's counter. The above examples, written with `get_units()`, looks like this:

```
seastar::future<> g() {
    static thread_local semaphore limit(100);
    return seastar::get_units(limit, 1).then([] (auto units) {
        return slow().finally([units = std::move(units)] {});
    });
}
```

Note the somewhat convoluted way that `get_units()` needs to be used: The continuations must be nested because we need the `units` object to be moved to the last continuation. If `slow()` returns a future (and does not throw immediately), the `finally()` continuation captures the `units` object until everything is done, but does not run any code.

Seastars programmers should generally avoid using the the `seamphore::wait()` and `semaphore::signal()` functions directly, and always prefer either `with_semaphore()` (when applicable) or `get_units()`.

10.2 Limiting resource use

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:

```
seastar::future<> using_lots_of_memory(size_t bytes) {
    static thread_local seastar::semaphore limit(1000000); // limit to 1MB
    return seastar::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. Seastar doesn't do this for you.

10.3 Limiting parallelism of loops

Above, we looked at a function `g()` which gets called by some external event, and wanted to control its parallelism. In this section, we look at parallelism of loops, which also can be controlled with semaphores.

Consider the following simple loop:

```
#include <seastar/core/sleep.hh>
seastar::future<> slow() {
    std::cerr << ".";
    return seastar::sleep(std::chrono::seconds(1));
}
seastar::future<> f() {
    return seastar::repeat([] {
        return slow().then([] { return seastar::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:

```
seastar::future<> f() {
    return seastar::repeat([] {
        slow();
        return seastar::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 may 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:

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

Note how this code differs from the code we saw above for limiting the number of parallel invocations of a function `g()`: 1. Here we cannot use a single `thread_local` semaphore. Each call to `f()` has its loop with parallelism of 100, so needs its own semaphore "`limit`", kept alive during the loop with `do_with()`. 2. Here we do not wait for `slow()` to complete before continuing the loop, i.e., we do not `return` the future chain starting at `futurize_apply(slow)`. The loop continues to the next iteration when a semaphore unit becomes available, while (in our example) 99 other operations might be ongoing in the background and we do not wait for them.

In the examples in this section, we cannot use the `with_semaphore()` shortcut. `with_semaphore()` returns a future which only resolves after the lambda's returned future resolves. But in the above example, the loop needs to know when just the semaphore units are available, to start the next iteration — and not wait for the previous iteration to complete. We could not achieve that with `with_semaphore()`. But the more general exception-safe idiom, `seastar::get_units()`, can be used in this case, and is recommended:

```
seastar::future<> f() {
    return seastar::do_with(seastar::semaphore(100), [] (auto& limit) {
        return seastar::repeat([&limit] {
            return seastar::get_units(limit, 1).then([] (auto units) {
                slow().finally([units = std::move(units)] {});
                return seastar::stop_iteration::no;
            });
        });
    });
}
```

The above examples are not realistic, because they have a never-ending loop and the future returned by `f()` will never resolve. In more realistic cases, the loop has an end, and at the end of the loop we need to wait for all the background operations which the loop started. We can 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:

```
seastar::future<> f() {
    return seastar::do_with(seastar::semaphore(100), [] (auto& limit) {
        return seastar::do_for_each(boost::counting_iterator<int>(0),
                boost::counting_iterator<int>(456), [&limit] (int i) {
            return seastar::get_units(limit, 1).then([] (auto units) {
                slow().finally([units = std::move(units)] {});
            });
        }).finally([&limit] {
            return limit.wait(100);
        });
    });
}
```

The last `finally` is what ensures that we wait for the last operations to complete: After the `repeat` loop ends (whether successfully or prematurely because of an exception in one of the iterations), we do a `wait(100)` to wait for the semaphore to reach its original value 100, meaning that all operations that we started have completed. Without this `finally`, the future returned by `f()` will resolve *before* all the iterations of the loop actually completed (the last 100 may still be running).

In the idiom we saw in the above example, the same semaphore is used both for limiting the number of background operations, and later to wait for all of them to complete. Sometimes, we want several different loops to use the same semaphore to limit their *total* parallelism. In that case we must use a separate mechanism for waiting for the completion of the background operations started by the loop. The most convenient way to wait for ongoing operations is using a gate, which we will describe in detail later. A typical example of a loop whose parallelism is limited by an external semaphore:

```
thread_local seastar::semaphore limit(100);
seastar::future<> f() {
    return seastar::do_with(seastar::gate(), [] (auto& gate) {
        return seastar::do_for_each(boost::counting_iterator<int>(0),
                boost::counting_iterator<int>(456), [&gate] (int i) {
            return seastar::get_units(limit, 1).then([&gate] (auto units) {
                gate.enter();
                seastar::futurize_apply(slow).finally([&gate, units = std::move(units)] {
                    gate.leave();
                });
            });
        }).finally([&gate] {
            return gate.close();
        });
    });
}
```

In this code, we use the external semaphore `limit` to limit the number of concurrent operations, but additionally have a gate specific to this loop to help us wait for all ongoing operations to complete.

TODO: also allow `get_units()` or something similar on a gate, and use that instead of the explicit gate.enter/gate.leave.

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?)

TODO: Have a few paragraphs, or even a section, on additional uses of semaphores. One is for mutual exclusion using semaphore(1) - we need to explain why although why in Seastar we don't have multiple threads touching the same data, if code is composed of different continuations (i.e., a fiber) it can switch to a different fiber in the middle, so if data needs to be protected between two continuations, it needs a mutex. Another example is something akin to wait_all: we start with a semaphore(0), run a known number N of asynchronous functions with finally sem.signal(), and from all this return the future sem.wait(N). PERHAPS even have a separate section on mutual exclusion, where we begin with semaphore(1) but also mention shared_mutex

11 Pipes

Seastar's `pipe<T>` is a mechanism to transfer data between two fibers, one producing data, and the other consuming it. It has a fixed-size buffer to ensures a balanced execution of the two fibers, because the producer fiber blocks when it writes to a full pipe, until the consumer fiber gets to run and read from the pipe.

A `pipe<T>` resembles a Unix pipe, in that it has a read side, a write side, and a fixed-sized buffer between them, and supports either end to be closed independently (and EOF or broken pipe when using the other side). A `pipe<T>` object holds the reader and write sides of the pipe as two separate objects. These objects can be moved into two different fibers. Importantly, if one of the pipe ends is destroyed (i.e., the continuations capturing it end), the other end of the pipe will stop blocking, so the other fiber will not hang.

The pipe's read and write interfaces are future-based blocking. I.e., the write() and read() methods return a future which is fulfilled when the operation is complete. The pipe is single-reader single-writer, meaning that until the future returned by read() is fulfilled, read() must not be called again (and same for write). Note: The pipe reader and writer are movable, but *not* copyable. It is often convenient to wrap each end in a shared pointer, so it can be copied (e.g., used in an std::function which needs to be copyable) or easily captured into multiple continuations.

12 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 <seastar/core/sleep.hh>
#include <seastar/core/gate.hh>
#include <boost/iterator/counting_iterator.hpp>

seastar::future<> slow(int i) {
    std::cerr << "starting " << i << "\n";
    return seastar::sleep(std::chrono::seconds(10)).then([i] {
        std::cerr << "done " << i << "\n";
    });
}
seastar::future<> f() {
    return seastar::do_with(seastar::gate(), [] (auto& g) {
        return seastar::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 seastar::sleep(std::chrono::seconds(1));
        }).then([&g] {
            seastar::sleep(std::chrono::seconds(1)).then([&g] {
                // This will fail, because it will be after the close()
                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:

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

13 Discarding futures, redux

In the section Introducing futures and continuations, we mentioned that *discarding* a future, by casting it to `void` to avoid the compiler's `[[nodiscard]]` warning, leaves unwaited work in the background. We said that this needs to be done carefully. After the previous sections introduced mechanisms such as `handle_exception()`, `sempahore` and `gate`, we can offer more concrete advice on how to **safely** discard futures:

1. **Handle values**: Naturally, if the discarded future has a value (`future<T>`), this value will be lost right after it is generated. If this value should be collected somewhere, a continuation can be attached to the future to do something with the value.
2. **Handle exceptions**: Similarly, if the discarded future resolves with an exception, this exception will be lost. Because exceptions are rare, programmers often forget to handle them. If a rare

exception were to be silently dropped, the programmer might never notice this bug. Therefore, if a future is destroyed before the exception it holds get used, Seastar logs a warning about an "Exceptional future ignored". Nevertheless, it is better for the application code to explicitly handle exceptions and not rely on this logging. A future which is about to be discarded should be attached a `.handle_exception()` continuation. This continuation can do various things with this exception - perhaps log it, aggregate many exceptions into one, or even ignore it - but the choice should be deliberate and explicit in the code.

3. **Limit the amount of background work**: When the code that starts background work runs in a loop, or background work is generated as a response to other events, the amount of work in the background may grow and grow until depleting all memory and crashing the application. To avoid this problem, the code should use a semaphore to limit the total background work to some finite amount. The application must wait for semaphore units to become available before starting more background work. We saw examples of this technique above.

4. **Provide a way to wait for background work**: Even if the application has no interest to directly wait for some background work to finish, at some point we inevitably need a way to wait for it to complete. One important case is shutting down a service: To safely terminate a service, we usually want to know that the ongoing background work has completed. The `gate` we mentioned above offers such a mechanism - waiting for some background work to have finished without having saved the individual futures anywhere. Background work that does not use a gate in this manner cannot be waited for, and we can never be sure if it completed before the shutdown. In some cases this is fine, but in many cases it isn't, and a gate should be used.

   Note that *shutdown* here doesn't necessarily need refer to shutting down the entire application. It can also be about the completion of some long operation - if this operation started a bunch of background work, we would like for all of it to finish before claiming that the operation has completed. Here too we would not like to announce that the operation completed while in fact it still has work ongoing in the background.

14 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.

15 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.

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

16 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)).

For optimal performance, Seastar's network stack is sharded just like Seastar applications are: each shard (thread) takes responsibility for a different subset of the connections. 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.

In the examples we saw earlier, `main()` ran our function `f()` only once, on the first thread. Unless the server is run with the `"-c1"` option (one thread only), this will mean that any connection arriving to a different thread will not be handled. So in all the examples below, we will need to run the same service loop on all cores. We can easily do this with the `smp::submit_to` function:

```
seastar::future<> service_loop();

seastar::future<> f() {
    return seastar::parallel_for_each(boost::irange<unsigned>(0, seastar::smp::count),
            [] (unsigned c) {
        return seastar::smp::submit_to(c, service_loop);
    });
}
```

Here we ask each of Seastar cores (from 0 to `smp::count`-1) to run the same function `service_loop()`. Each of these invocations returns a future, and `f()` will return when all of them have returned (in the examples below, they will never return - we will discuss shutting down services in later sections).

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

seastar::future<> service_loop() {
    return seastar::do_with(seastar::listen(seastar::make_ipv4_address({1234})),
            [] (auto& listener) {
        return seastar::keep_doing([&listener] () {
            return listener.accept().then(
                [] (seastar::connected_socket s, seastar::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. We use `do_with()` to ensure that the listener socket lives throughout the loop. 3. 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`). 4. 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).

Output from this server looks like the following example:

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

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

```
$ ./a.out
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:

```
    seastar::listen_options lo;
    lo.reuse_address = true;
    return seastar::do_with(seastar::listen(seastar::make_ipv4_address({1234}), lo),
```

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

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

seastar::future<> service_loop() {
    seastar::listen_options lo;
    lo.reuse_address = true;
    return seastar::do_with(seastar::listen(seastar::make_ipv4_address({1234}), lo),
            [] (auto& listener) {
        return seastar::keep_doing([&listener] () {
            return listener.accept().then(
                [] (seastar::connected_socket s, seastar::socket_address a) {
                    auto out = s.output();
                    return seastar::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 (or in the TCP case, there is enough room in the TCP congestion window) 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 <seastar/core/seastar.hh>
#include <seastar/core/reactor.hh>
#include <seastar/core/future-util.hh>

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

seastar::future<> service_loop() {
    seastar::listen_options lo;
    lo.reuse_address = true;
    return seastar::do_with(seastar::listen(seastar::make_ipv4_address({1234}), lo),
            [] (auto& listener) {
        return seastar::keep_doing([&listener] () {
            return listener.accept().then(
                [] (seastar::connected_socket s, seastar::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 `service_loop()` 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`.

17 Sharded services

In the previous section we saw that a Seastar application usually needs to run its code on all available CPU cores. We saw that the `seastar::smp::submit_to()` function allows the main function, which initially runs only on the first core, to start the server's code on all `seastar::smp::count` cores.

However, usually one needs not just to run code on each core, but also to have an object that contains the state of this code. Additionally, one may like to interact with those different objects, and also have a mechanism to stop the service running on the different cores.

The `seastar::sharded<T>` template provides a structured way create such a *sharded service*. It creates a separate object of type `T` in each core, and provides mechanisms to interact with those copies, to start some code on each, and finally to cleanly stop the service.

To use `seastar::sharded`, first create a class for the object holding the state of the service on a single core. For example:

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

class my_service {
public:
    std::string _str;
    my_service(const std::string& str) : _str(str) { }
    seastar::future<> run() {
        std::cerr << "running on " << seastar::engine().cpu_id() <<
            ", _str = " << _str << \n";
        return seastar::make_ready_future<>();
    }
    seastar::future<> stop() {
        return seastar::make_ready_future<>();
    }
};
```

The only mandatory method in this object is `stop()`, which will be called in each core when we want to stop the sharded service and want to wait until it stops on all cores.

Now let's see how to use it:

```
#include <seastar/core/sharded.hh>

seastar::sharded<my_service> s;

seastar::future<> f() {
    return s.start(std::string("hello")).then([] {
        return s.invoke_on_all([] (my_service& local_service) {
            return local_service.run();
        });
    }).then([] {
        return s.stop();
    });
}
```

The `s.start()` starts the service by creating a `my_service` object on each of the cores. The arguments to `s.start()`, if any (in this example, `std::string("hello")`), are passed to `my_service`'s constructor.

But `s.start()` did not start running any code yet (besides the object's constructor). For that, we have the `s.invoke_on_all()` which runs the given lambda on all the cores - giving each lambda the local `my_service` object on that core. In this example, we have a `run()` method on each object, so we run that.

Finally, at the end of the run we want to give the service on all cores a chance to shut down cleanly, so we call `s.stop()`. This will call the `stop()` method on each core's object, and wait for all of them to finish. Calling `s.stop()` before destroying `s` is mandatory - Seastar will warn you if you forget to do it.

In addition to `invoke_on_all()` which runs the same code on all shards, another feature a sharded service often needs is for one shard to invoke code another specific shard. This is done by calling the sharded service's `invoke_on()` method. For example:

```
seastar::sharded<my_service> s;
...
return s.invoke_on(0, [] (my_service& local_service) {
    std::cerr << "invoked on " << seastar::engine().cpu_id() <<
        ", _str = " << local_service._str << "\n";
});
```

This runs the lambda function on shard 0, with a reference to the local `my_service` object on that shard.

18 File I/O

19 Shutting down cleanly

TODO: Handling interrupt, shutting down services, etc.

Move the seastar::gate section here.

20 Command line options

20.1 Standard Seastar command-line options

All Seastar applications accept a standard set of command-line arguments, such as those we've already seen above: The `-c` option for controlling the number of threads used, or `-m` for determining the amount of memory given to the application.

TODO: list and explain more of these options.

Every Seastar application also accepts the `-h` (or `--help`) option, which lists and explains all the available options — the standard Seastar ones, and the user-defined ones as explained below. ## User-defined command-line options Seastar parses the command line options (`argv[]`) when it is passed to `app_template::run()`, looking for its own standard options. Therefore, it is not recommended that the application tries to parse `argv[]` on its own because the application might not understand some of the standard Seastar options and not be able to correctly skip them.

Rather, applications which want to have command-line options of their own should tell Seastar's command line parser of these additional application-specific options, and ask Seastar's command line parser to recognize them too. Seastar's command line parser is actually the Boost library's `boost::program_options`. An application adds its own option by using the `add_options()` and `add_positional_options()` methods on the `app_template` to define options, and later calling `configuration()` to retrieve the setting of these options. For example,

```
#include <iostream>
#include <seastar/core/app-template.hh>
#include <seastar/core/reactor.hh>
int main(int argc, char** argv) {
    seastar::app_template app;
    namespace bpo = boost::program_options;
    app.add_options()
        ("flag", "some optional flag")
        ("size,s", bpo::value<int>()->default_value(100), "size")
        ;
    app.add_positional_options({
       { "filename", bpo::value<std::vector<seastar::sstring>>()->default_value({}),
         "sstable files to verify", -1}
    });
    app.run(argc, argv, [&app] {
        auto& args = app.configuration();
        if (args.count("flag")) {
            std::cout << "Flag is on\n";
        }
        std::cout << "Size is " << args["size"].as<int>() << "\n";
        auto& filenames = args["filename"].as<std::vector<seastar::sstring>>();
        for (auto&& fn : filenames) {
            std::cout << fn << "\n";
        }
        return seastar::make_ready_future<>();
    });
    return 0;
}
```

In this example, we add via `add_options()` two application-specific options: `--flag` is an optional parameter which doesn't take any additional agruments, and `--size` (or `-s`) takes an integer value, which defaults (if this option is missing) to 100. Additionally, we ask via `add_positional_options()` that an unlimited number of arguments that do not begin with a "`-`" — the so-called *positional* arguments — be collected to a vector of strings under the "filename" option. Some example outputs from this program:

```
$ ./a.out
Size is 100
$ ./a.out --flag
Flag is on
Size is 100
$ ./a.out --flag -s 3
Flag is on
Size is 3
$ ./a.out --size 3 hello hi
Size is 3
hello
hi
$ ./a.out --filename hello --size 3 hi
Size is 3
hello
hi
```

`boost::program_options` has more powerful features, such as required options, option checking and combining, various option types, and more. Please refer to Boost's documentation for more information.

21 Debugging a Seastar program

21.1 Debugging ignored exceptions

If a future resolves with an exception, and the application neglects to handle that exception or to explicitly ignore it, the application may have missed an important problem. This is likely to be an application bug.

Therefore, Seastar prints a warning message to the log if a future is destroyed when it stores an exception that hasn't been handled.

For example, consider this code:

```
#include <seastar/core/future.hh>
#include <seastar/core/sleep.hh>

class myexception {};

seastar::future<> g() {
    return seastar::make_exception_future<>(myexception());
}

seastar::future<> f() {
    g();
    return seastar::sleep(std::chrono::seconds(1));
}
```

Here, the main function `f()` calls `g()`, but doesn't do anything with the future it returns. But this future resolves with an exception, and this exception is silently ignored. So Seastar prints this warning message about the ignored exception:

```
WARN  2018-01-11 13:23:17,976 [shard 0] seastar - Exceptional future ignored:
myexception, backtrace: 0x41ce24
  0x63729e
  0x636cd5
  0x4fa6ec
  0x4fb0e8
  0x5010e0
  0x41bf96
  0x41c26c
  0x46a734
  0x4fd661
  0x4fe1e2
  0x4fe332
  0x417a0b
  /lib64/libc.so.6+0x21009
  0x417b99
```

This message says that an exceptional future was ignored, and that the type of the exception was "`myexception`". The type of the exception is usually not enough to pinpoint where the problem happened, so the warning message also includes the backtrace - the call chain - leading to where the exceptional future was destroyed. The backtrace is given as a list of addresses, where code in other shared libraries is written as a shared library plus offset (when ASLR is enabled, the shared libraries are mapped in a different address each time).

Seastar includes a utility, `seastar-addr2line`, for translating these addresses into readable backtraces including exact method names, source files and line numbers. This utility needs the *unstripped* executable. Typically, a stripped executable is used for production, but an unstripped copy is kept separately to be used in debugging - including `seastar-addr2line`.

To decode the backtrace, we run

```
seastar-addr2line -e a.out
```

And then paste the list of addresses in the warning message, and conclude with a `control-D` (it's also possible, if you want, to put the list of addresses in the `seastar-addr2line` command line). The result looks like this:

```
seastar::report_failed_future(std::__exception_ptr::exception_ptr) at /home/nyh/seastar/core/reactor
seastar::future<>::~future() at /home/nyh/seastar/core/future.hh:828
 (inlined by) f() at /home/nyh/seastar/doc/code/26.cc:11
std::_Function_handler<seastar::future<> (), seastar::future<> (*)()>::_M_invoke(std::_Any_data cons
std::function<seastar::future<> ()>::operator()() const at /usr/include/c++/7/bits/std_function.h:70
 (inlined by) operator() at /home/nyh/seastar/core/app-template.cc:119
 (inlined by) _M_invoke at /usr/include/c++/7/bits/std_function.h:302
std::function<seastar::future<int> ()>::operator()() const at /usr/include/c++/7/bits/std_function.h
 (inlined by) seastar::future<int> seastar::futurize<seastar::future<int> >::apply<std::function<sea
 (inlined by) auto seastar::futurize_apply<std::function<seastar::future<int> ()>&>(std::function<se
 (inlined by) operator() at /home/nyh/seastar/core/app-template.cc:108
 (inlined by) _M_invoke at /usr/include/c++/7/bits/std_function.h:316
std::function<void ()>::operator()() const at /usr/include/c++/7/bits/std_function.h:706
 (inlined by) seastar::apply_helper<std::function<void ()>, std::tuple<>&&, std::integer_sequence<un
 (inlined by) auto seastar::apply<std::function<void ()>>(std::function<void ()>&&, std::tuple<>&&)
 (inlined by) std::enable_if<!seastar::is_future<std::result_of<std::function<void ()> ()>::type>::v
 (inlined by) seastar::future<> seastar::futurize<void>::apply<std::function<void ()>>(std::function
 (inlined by) seastar::future<> seastar::future<>::then<std::function<void ()>, seastar::future<> >(
 (inlined by) seastar::continuation<seastar::future<> seastar::future<>::then<std::function<void ()>
seastar::reactor::run_tasks(seastar::reactor::task_queue&) at /home/nyh/seastar/core/reactor.cc:2487
seastar::reactor::run_some_tasks(std::chrono::time_point<std::chrono::_V2::steady_clock, std::chrono
seastar::reactor::run_some_tasks(std::chrono::time_point<std::chrono::_V2::steady_clock, std::chrono
 (inlined by) seastar::reactor::run() at /home/nyh/seastar/core/reactor.cc:3028
seastar::app_template::run_deprecated(int, char**, std::function<void ()>&&) at /home/nyh/seastar/co
seastar::app_template::run(int, char**, std::function<seastar::future<int> ()>&&) at /home/nyh/seast
seastar::app_template::run(int, char**, std::function<seastar::future<> ()>&&) at /home/nyh/seastar/
main at /home/nyh/seastar/doc/code/main.cc:11
__libc_start_main at ??:?
```

Most of the lines at the bottom of this backtrace are not interesting, and just showing the internal details of how Seastar ended up running the main function `f()`. The only interesting part is the *first* few lines:

```
seastar::report_failed_future(std::__exception_ptr::exception_ptr) at
 /home/nyh/seastar/core/reactor.cc:4201
seastar::future<>::~future() at /home/nyh/seastar/core/future.hh:828
 (inlined by) f() at /home/nyh/seastar/doc/code/26.cc:11
```

Here we see that the warning message was printed by the `seastar::report_failed_future()` function which was called when destroying a future (`future<>::~future`) that had not been handled. The future's destructor was called in line 11 of our test code (`26.cc`), which is indeed the line where we called `g()` and ignored its result.
This backtrace gives us an accurate understanding of where our code destroyed an exceptional future without handling it first, which is usually helpful in solving these kinds of bugs. Note that this technique does not tell us where the exception was first created, nor what code passed around the exceptional future before it was destroyed - we just learn where the future was destroyed. To learn where the exception was originally thrown, see the next section:

21.2 Finding where an exception was thrown

Sometimes an application logs an exception, and we want to know where in the code the exception was originally thrown. Unlike languages like Java, C++ does not have a builtin method of attaching a backtrace to every exception. So Seastar provides functions which allow adding to an exception the backtrace recorded when throwing it.

For example, in the following code we throw and catch an `std::runtime_error` normally:

```
#include <seastar/core/future.hh>
#include <seastar/util/log.hh>
#include <exception>
#include <iostream>

seastar::future<> g() {
    return seastar::make_exception_future<>(std::runtime_error("hello"));
}

seastar::future<> f() {
    return g().handle_exception([](std::exception_ptr e) {
        std::cerr << "Exception: " << e << "\n";
    });
}
```

The output is

```
Exception: std::runtime_error (hello)
```

From this output, we have no way of knowing that the exception was thrown in `g()`. We can solve this if we use `make_exception_future_with_backtrace` instead of `make_exception_future`:

```
#include <util/backtrace.hh>
seastar::future<> g() {
    return seastar::make_exception_future_with_backtrace<>(std::runtime_error("hello"));
}
```

Now the output looks like

```
Exception: seastar::internal::backtraced<std::runtime_error> (hello Backtrace:   0x678bd3
  0x677204
  0x67736b
  0x678cd5
  0x4f923c
  0x4f9c38
  0x4ff4d0
...
)
```

Which, as above, can be converted to a human-readable backtrace by using the `seastar-addr2line` script.

In addition to `seastar::make_exception_future_with_backtrace()`, Seastar also provides a function `throw_with_backtrace()`, to throw an exception instead of returning an exceptional future. For example:

```
    seastar::throw_with_backtrace<std::runtime_error>("hello");
```

In the current implementation, both `make_exception_future_with_backtrace` and `throw_with_backtrace` require that the original exception type (in the above example, `std::runtime_error`) is a subclass of the `std::exception` class. The original exception provides a `what()` string, and the wrapped exception adds the backtrace to this string, as demonstrated above. Moreover, the wrapped exception type is a *subclass* of the original exception type, which allows `catch(...)` code to continue filtering by the exception original type - despite the addition of the backtrace.

21.3 Debugging with gdb

handle SIGUSR1 pass noprint handle SIGALRM pass noprint

22 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:

```
seastar::future<int> slow() {
    using namespace std::chrono_literals;
    return seastar::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.

23 Memory allocation in Seastar

23.1 Per-thread memory allocation

Seastar requires that applications be sharded, i.e., that code running on different threads operate on different objects in memory. We already saw in Seastar memory how Seastar takes over a given amount of memory (often, most of the machine's memory) and divides it equally between the different threads. Modern multi-socket machines have non-uniform memory access (NUMA), meaning that some parts of memory are closer to some of the cores, and Seastar takes this knowledge into account when dividing the memory between threads. Currently, the division of memory between threads is static, and equal - the threads are expected to experience roughly equal amount of load and require roughly equal amounts of memory.

To achieve this per-thread allocation, Seastar redefines the C library functions `malloc()`, `free()`, and their numerous relatives — `calloc()`, `realloc()`, `posix_memalign()`, `memalign()`, `malloc_usable_size()`, and `malloc_trim()`. It also redefines the C++ memory allocation functions, `operator new`, `operator delete`, and all their variants (including array versions, the C++14 delete taking a size, and the C++17 variants taking required alignment).

It is important to remember that Seastar's different threads *can* see memory allocated by other threads, but they are nontheless strongly discouraged from actually doing this. Sharing data objects between threads on modern multi-core machines results in stiff performance penalties from locks, memory barriers, and cache-line bouncing. Rather, Seastar encourages applications to avoid sharing objects between threads when possible (by *sharding* — each thread owns a subset of the objects), and when threads do need to interact they do so with explicit message passing, with `submit_to()`, as we shall see later.

23.2 Foreign pointers

An object allocated on one thread will be owned by this thread, and eventually should be freed by the same thread. Freeing memory on the *wrong* thread is strongly discouraged, but is currently supported (albeit slowly) to support library code beyond Seastar's control. For example, `std::exception_ptr` allocates memory; So if we invoke an asynchronous operation on a remote thread and this operation returns an exception, when we free the returned `std::exception_ptr` this will happen on the "wrong" core. So Seastar allows it, but inefficiently.

In most cases objects should spend their entire life on a single thread and be used only by this thread. But in some cases we want to reassign ownership of an object which started its life on one thread, to a different thread. This can be done using a `seastar::foreign_ptr<>`. A pointer, or smart pointer, to an object is wrapped in a `seastar::foreign_ptr<P>`. This wrapper can then be moved into code running in a different thread (e.g., using `submit_to()`).

The most common use-case is a `seastar::foreign_ptr<std::unique_ptr<T>>`. The thread receiving this `foreign_ptr` will get exclusive use of the object, and when it destroys this wrapper, it will go back to the original thread to destroy the object. Note that the object is not only freed on the original shard - it is also *destroyed* (i.e., its destructor is run) there. This is often important when the object's destructor needs to access other state which belongs to the original shard - e.g., unlink itself from a container.

Although `foreign_ptr` ensures that the object's *destructor* automatically runs on the object's home thread, it does not absolve the user from worrying where to run the object's other methods. Some simple methods, e.g., methods which just read from the object's fields, can be run on the receiving thread. However, other methods may need to access other data owned by the object's home shard, or need to prevent concurrent operations. Even if we're sure that object is now used exclusively by the receiving thread, such methods must still be run, explicitly, on the home thread:

```
// fp is some foreign_ptr<>
return smp::submit_to(fp.get_owner_shard(), [p=fp.get()]
    { return p->some_method(); });
```

So `seastar::foreign_ptr<>` not only has functional benefits (namely, to run the destructor on the home shard), it also has *documentational* benefits - it warns the programmer to watch out every time the object is used, that this is a *foreign* pointer, and if we want to do anything non-trivial with the pointed object, we may need to do it on the home shard.

Above, we discussed the case of transferring ownership of an object to a another shard, via `seastar::foreign_ptr<std::un` However, sometimes the sender does not want to relinquish ownership of the object. Sometimes, it wants the remote thread to operate on its object and return with the object intact. Sometimes, it wants to send the same object to multiple shards. In such cases, `seastar::foreign_ptr<seastar::lw_shared_ptr<T>> is useful. The user needs to watch out, of course, not to operate on the same object from multiple threads concurrently. If this cannot be ensured by program logic alone, some methods of serialization must be used - such as running the operations on the home shard with`submit_to()‘ as described above.

Normally, a `seastar::foreign_ptr` cannot not be copied - only moved. However, when it holds a smart pointer that can be copied (namely, a `shared_ptr`), one may want to make an additional copy of that pointer and create a second `foreign_ptr`. Doing this is inefficient and asynchronous (it requires communicating with the original owner of the object to create the copies), so a method `future<foreign_ptr> copy()` needs to be explicitly used instead of the normal copy constructor.

TODO: continue here: destroy is asynchronous but doesn't return a future. So is reset. So watch out for flow control.

23.3 Smart pointers

TODO: Talk about proper C++ style is to only use sharedptr/uniqptr to avoid any risk of memory leaks, etc.

23.4 Out of memory

TODO: Talk about std::bad_alloc and why allocation must be able to fail rather than just wait until memory becomes available: If we had future<void> *allocation and milin paralell requests we could deadlock (all waiting for more memory) instead of some of the requests failing. Also mention admitance throttling and semaphores as a way noto fail requests. Mention that future<void>* allocation could still be useful to avoid latency spikes by splitting works to several task quotas.

talk about reclaim. ## Heap profiling? Etc.? ## Default allocator talk about the option not to redefine malloc and why it's needed when we have alien code. ## Movable memory allocation TODO: Talk about one of the downsides of explicit alloc/free is fragmentation. Say that GC solves this problem, because it moves allocation, but also comes with a feature that complicates everything (and we don't need) which is to find where the garbage is. So LSA is a mechanism to allow allocated memory to move, reducing fragmentation, while still needing explicit allocations and frees. ## Allocator implementation talk about the implementation of the allocator? Or maybe not.

24 RPC

TODO: explain how Seastar is mostly about code running on a single machine, and mention again the mechanisms we have to communicate between its cores and to run code on other cores. But if we want to write a distributed program on a cluster on multiple machines (a cluster), we need a way to communicate with other nodes - and this is RPC.

25 More about the sharded network stack

TODO: how it really works, e.g., in Posix stack connections are actually accepted on one core but then forward to the destination core.

TODO: How does batching/buffering in our network stack works? If we write() a short string, or several short strings, when is the packet sent?

26 Seastar::thread

Seastar's programming model, using futures and continuations, is very powerful and efficient. However, as we've already seen in examples above, it is also relatively verbose: Every time that we need to wait before proceeding with a computation, we need to write another continuation. We also need to worry about passing the data between the different continuations (using techniques like those described in the Lifetime management section). Simple flow-control constructs such as loops also become more involved using continuations. For example, consider this simple classical synchronous code:

```
std::cout << "Hi.\n";
for (int i = 1; i < 4; i++) {
    sleep(1);
    std::cout << i << "\n";
}
```

In Seastar, using futures and continuations, we need to write something like this:

```
std::cout << "Hi.\n";
return seastar::do_for_each(boost::counting_iterator<int>(1),
    boost::counting_iterator<int>(4), [] (int i) {
    return seastar::sleep(std::chrono::seconds(1)).then([i] {
        std::cout << i << "\n";
    });
});
```

But Seastar also allows, via `seastar::thread`, to write code which looks more like synchronous code. A `seastar::thread` provides an execution environment where blocking is tolerated; You can issue an asyncrhonous function, and wait for it in the same function, rather then establishing a callback to be called with `future<>::then()`:

```
seastar::thread th([] {
    std::cout << "Hi.\n";
    for (int i = 1; i < 4; i++) {
        seastar::sleep(std::chrono::seconds(1)).get();
        std::cout << i << "\n";
    }
});
```

A `seastar::thread` is **not** a separate operating system thread. It still uses continuations, which are scheduled on Seastar's single thread (per core). It works as follows:

The `seastar::thread` allocates a 128KB stack, and runs the given function until the it *blocks* on the call to a future's `get()` method. Outside a `seastar::thread` context, `get()` may only be called on a future which is already available. But inside a thread, calling `get()` on a future which is not yet available stops running the thread function, and schedules a continuation for this future, which continues to run the thread's function (on the same saved stack) when the future becomes available.

Just like normal Seastar continuations, `seastar::threads` always run on the same core they were launched on. They are also cooperative: they are never preempted except when `seastar::future::get()` blocks or on explict calls to `seastar::thread::yield()`.

It is worth reiterating that a `seastar::thread` is not a POSIX thread, and it can only block on Seastar futures, not on blocking system calls. The above example used `seastar::sleep()`, not the `sleep()` system call. The `seastar::thread`'s function can throw and catch exceptions normally. Remember that `get()` will throw an exception if the future resolves with an exception.

In addition to `seastar::future::get()`, we also have `seastar::future::wait()` to wait *without* fetching the future's result. This can sometimes be useful when you want to avoid throwing an exception when the future failed (as `get()` does). For example:

```
    future<char> getchar();
    int try_getchar() noexcept { // run this in seastar::thread context
        future fut = get_char();
        fut.wait();
        if (fut.failed()) {
            return -1;
        } else {
            // Here we already know that get() will return immediately,
            // and will not throw.
            return fut.get();
        }
    }
```

TODO: talk about task quota and maybe_yield(), and should_yield().

26.1 Starting and ending a seastar::thread

After we created a `seastar::thread` object, we need wait until it ends, using its `join()` method. We also need to keep that object alive until `join()` completes. A complete example using `seastar::thread` will therefore look like this:

```
#include <seastar/core/sleep.hh>
#include <seastar/core/thread.hh>
seastar::future<> f() {
    seastar::thread th([] {
        std::cout << "Hi.\n";
        for (int i = 1; i < 4; i++) {
            seastar::sleep(std::chrono::seconds(1)).get();
            std::cout << i << "\n";
        }
    });
    return do_with(std::move(th), [] (auto& th) {
        return th.join();
    });
}
```

The `seastar::async()` function provides a convenient shortcut for creating a `seastar::thread` and returning a future which resolves when the thread completes:

```
#include <seastar/core/sleep.hh>
#include <seastar/core/thread.hh>
seastar::future<> f() {
    return seastar::async([] {
        std::cout << "Hi.\n";
        for (int i = 1; i < 4; i++) {
            seastar::sleep(std::chrono::seconds(1)).get();
            std::cout << i << "\n";
        }
    });
}
```

`seastar::async()`'s lambda may return a value, and `seastar::async()` returns it when it completes. For example:

```
seastar::future<seastar::sstring> read_file(sstring file_name) {
    return seastar::async([file_name] () {  // lambda executed in a thread
        file f = seastar::open_file_dma(file_name).get0();  // get0() call "blocks"
        auto buf = f.dma_read(0, 512).get0();  // "block" again
        return seastar::sstring(buf.get(), buf.size());
    });
};
```

While `seastar::threads` and `seastar::async()` make programming more convenient, they also add overhead beyond that of programming directly with continuations. Most notably, each `seastar::thread` requires additional memory for its stack. It is therefore not a good idea to use a `seastar::thread` to handle a highly concurrent operation. For example, if you need to handle 10,000 concurrent requests, do not use a `seastar::thread` to handle each — use futures and continuations. But if you are writing code where you know that only a few instances will ever run concurrently, e.g., a background cleanup operation in your application, `seastar::thread` is a good match. `seastar::thread` is also great for code which doesn't care about performance — such as test code.

27 Streams

TODO: Put this section somewhere. Discuss input_stream and output_stream, how they assume one reader / writer - no parallel reads (otherwise we don't know anything about order - if you must do this use a semaphore to serialize). Talk about underlying layers being parallel anyway - input from network continues to process TCP packets in parallel, and input from disk pre-fetches more chunks in parallel.

28 SIGINT handling

TODO:

29 Isolation of application components

Seastar makes multi-tasking very easy - as easy as running an asynchronous function. It is therefore easy for a server to do many unrelated things in parallel. For example, a server might be in the process of answering 100 users' requests, and at the same time also be making progress on some long background operation.

But in the above example, what percentage of the CPU and disk throughput will the background operation get? How long can one of the user's requests be delayed by the background operation? Without the mechanisms we describe in this section, these questions cannot be reliably answered:

- The background operation may be a very "considerate" single fiber, i.e., run a very short continuation and then schedule the next continuation to run later. At each point the scheduler sees 100 request-handling continuations and just one of the background continuations ready to run. The background task gets around 1% of the CPU time, and users' requests are hardly delayed.
- On the other hand, the background operation may spawn 1,000 fibers in parallel and have 1,000 ready-to-run continuations at each time. The background operation will get about 90% of the runtime, and the continuation handling a user's request may get stuck behind 1,000 of these background continuations, and experience huge latency.

Complex Seastar applications often have different components which run in parallel and have different performance objectives. In the above example we saw two components - user requests and the background operation. The first goal of the mechanisms we describe in this section is to *isolate* the performance of each component from the others; In other words, the throughput and latency of one component should not depend on decisions that another component makes - e.g., how many continuations it runs in parallel.

The second goal is to allow the application to *control* this isolation, e.g., in the above example allow the application to explicitly control the amount of CPU the background operation recieves, so that it completes at a desired pace.

In the above examples we used CPU time as the limited resource that the different components need to share effectively. As we show later, another important shared resource is disk I/O.

29.1 Scheduling groups (CPU scheduler)

Consider the following asynchronous function `loop()`, which loops until some shared variable `stop` becomes true. It keeps a `counter` of the number of iterations until stopping, and returns this counter when finally stopping.

```
seastar::future<long> loop(int parallelism, bool& stop) {
    return seastar::do_with(0L, [parallelism, &stop] (long& counter) {
        return seastar::parallel_for_each(boost::irange<unsigned>(0, parallelism),
            [&stop, &counter]  (unsigned c) {
                return seastar::do_until([&stop] { return stop; }, [&counter] {
                    ++counter;
                    return seastar::make_ready_future<>();
                });
            }).then([&counter] { return counter; });
    });
}
```

The `parallelism` parameter determines the parallelism of the silly counting operation: `parallelism=1` means we have just one loop incrementing the counter; `parallelism=10` means we start 10 loops in parallel all incrementing the same counter.

What happens if we start two `loop()` calls in parallel and let them run for 10 seconds?

```
seastar::future<> f() {
    return seastar::do_with(false, [] (bool& stop) {
        seastar::sleep(std::chrono::seconds(10)).then([&stop] {
            stop = true;
        });
        return seastar::when_all_succeed(loop(1, stop), loop(1, stop)).then(
            [] (long n1, long n2) {
                std::cout << "Counters: " << n1 << ", " << n2 << "\n";
            });
    });
}
```

It turns out that if the two `loop()` calls had the same parallelism `1`, we get roughly the same amount of work from both of them:

```
Counters: 3'559'635'758, 3'254'521'376
```

But if for example we ran a `loop(1)` in parallel with a `loop(10)`, the result is that the `loop(10)` gets 10 times more work done:

```
Counters: 629'482'397, 6'320'167'297
```

Why does the amount of work that loop(1) can do in ten seconds depends on the parallelism chosen by its competitor, and how can we solve this?

The reason this happens is as follows: When a future resolves and a continuation was linked to it, this continuation becomes ready to run. By default, Seastar's scheduler keeps a single list of ready-to-run continuations (in each shard, of course), and runs the continuations at the same order they became ready to run. In the above example, `loop(1)` always has one ready-to-run continuation, but `loop(10)`, which runs 10 loops in parallel, always has ten ready-to-run continuations. So for every continuation of `loop(1)`, Seastar's default scheduler will run 10 continuations of `loop(10)`, which is why loop(10) gets 10 times more work done.

To solve this, Seastar allows an application to define separate components known as **scheduling groups**, which each has a separate list of ready-to-run continuations. Each scheduling group gets to run its own continuations on a desired percentage of the CPU time, but the number of runnable continuations in one scheduling group does not affect the amount of CPU that another scheduling group gets. Let's look at how this is done:

A scheduling group is defined by a value of type `scheduling_group`. This value is opaque, but internally it is a small integer (similar to a process ID in Linux). We use the `seastar::with_scheduling_group()` function to run code in the desired scheduling group:

```
seastar::future<long>
loop_in_sg(int parallelism, bool& stop, seastar::scheduling_group sg) {
    return seastar::with_scheduling_group(sg, [parallelism, &stop] {
        return loop(parallelism, stop);
    });
}
```

TODO: explain what `with_scheduling_group` group really does, how the group is "inherited" to the continuations started inside it.

Now let's create two scheduling groups, and run `loop(1)` in the first scheduling group and `loop(10)` in the second scheduling group:

```
seastar::future<> f() {
    return seastar::when_all_succeed(
            seastar::create_scheduling_group("loop1", 100),
            seastar::create_scheduling_group("loop2", 100)).then(
        [] (seastar::scheduling_group sg1, seastar::scheduling_group sg2) {
        return seastar::do_with(false, [sg1, sg2] (bool& stop) {
            seastar::sleep(std::chrono::seconds(10)).then([&stop] {
                stop = true;
            });
            return seastar::when_all_succeed(loop_in_sg(1, stop, sg1), loop_in_sg(10, stop, sg2)).th
                [] (long n1, long n2) {
                    std::cout << "Counters: " << n1 << ", " << n2 << "\n";
                });
        });
    });
}
```

Here we created two scheduling groups, `sg1` and `sg2`. Each scheduling group has an arbitrary name (which is used for diagnostic purposes only), and a number of *shares*, a number traditionally between 1 and 1000: If one scheduling group has twice the number of shares than a second scheduling group, it will get twice the amount of CPU time. In this example, we used the same number of shares (100) for both groups, so they should get equal CPU time.

Unlike most objects in Seastar which are separate per shard, Seastar wants the identities and numbering of the scheduling groups to be the same on all shards, because it is important when invoking tasks on remote shards. For this reason, the function to create a scheduling group, `seastar::create_scheduling_group()`, is an asynchronous function returning a `future<scheduling_group>`.

Running the above example, with both scheduling group set up with the same number of shares (100), indeed results in both scheduling groups getting the same amount of CPU time:

```
Counters: 3'353'900'256, 3'350'871'461
```

Note how now both loops got the same amount of work done - despite one loop having 10 times the parallelism of the second loop.

If we change the definition of the second scheduling group to have 200 shares, twice the number of shares of the first scheduling group, we'll see the second scheduling group getting twice the amount of CPU time:

```
Counters: 2'273'783'385, 4'549'995'716
```

29.2 Latency

TODO: Task quota, preempt, loops with built-in preemption check, etc. ## Disk I/O scheduler TODO ## Network scheduler TODO: Say that not yet available. Give example of potential problem - e.g., sharing a slow WAN link. ## Controllers TODO: Talk about how to dynamically change the number of shares, and why. ## Multi-tenancy TODO