# Seastar Tutorial

# Introduction

**Seastar is an advanced, open-source C++ framework for high-performance server applications on modern hardware.**

Seastar là một framework mã nguồn mở C++, nâng cao cho những ứng dụng server hiệu nâng cao trên modern hardware.

**Applications using Seastar can run on Linux or OSv. Key Seastar design features include:**

Những Application mà sử dụng Seastar có thể chạy trên Linux hoặc Osv. Những đặc điểm nổi bật trong thiết kế của Seastar bao gồm:

* **Shared-nothing design: Seastar uses a shared-nothing model that shards all requests onto individual cores.**

Shared-nothing design: Seastar sử dụng mô hình share-nothing cái mà băm các yêu cầu lên trên mỗi core riêng biệt.

* **User-space networking: Seastar offers a choice of network stack, including conventional Linux networking for ease of development, DPDK for fast user-space networking on Linux, and native networking on OSv.**
* **Futures and promises: an advanced new model for concurrent applications that offers C++ programmers both high performance and the ability to create comprehensible, testable high-quality code.**

Futures and promises: một mô hình nâng cao mới cho các ứng dụng chạy một cách đồng thời mà những ứng dụng này yêu cầu lập trình viên cả về hiệu năng cao lẫn khả năng tạo ra các dòng code chất lượng, dễ hiểu, dễ test.

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

# Getting started



**As we do in this example, each Seastar program must define and run, an app\_template object.**

Như chúng ta làm trong ví dụ này, mỗi chương trình Seatar phải định nghĩa và chạy một đối tượng gọi là App\_Template.

**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.**

Đối tượng này bắt đầu **the main event loop** (the seatar engine) trên một hoặc nhiều CPU and sau đó chạy một hàm không tên, đó làm một lambad.

**When the future returned by it resolves the app will shut down, stopping event loops on all cpus, and app.run() will return.**

Khi future được trả về bởi Lambda xử lí, app sẽ shut down,dừng vòng vòng lặp và app.run() sẽ trả về .

**There's also app.run\_deprecated() variant, still in use by some code, which differs from run() in that the application doesn't exit when the callback returns, but it has to be explicitly stopped by calling engine\_exit()**.

Cũng có một biến thể đó là app.run\_deprecated, vẫn sử dụng trong vài đoạn code, nó khác với run() trong trường hợp mà applicattion không exit khi hàm callback returns, nhưng nó phải dừng lại hoàn toàn khi gọi hàm engine\_exit().

**It should be used instead of the regular C exit(), to allow for proper cleanups when necessary (we'll discuss these cleanups later).**

Nó nên được sử dụng thay vì hàm exit(), để cho phép cleanups hoàn toàn.

# Seastar threads

**As explained in the introduction, Seastar-based programs run a single thread on each CPU.**

Như đã giải thích trong phần giới thiệu, Chương trình Seastar-based chạy một Thread trên mỗi CPU.

**Each of these threads runs its own event loop, known as the engine in Seastar nomenclature.**

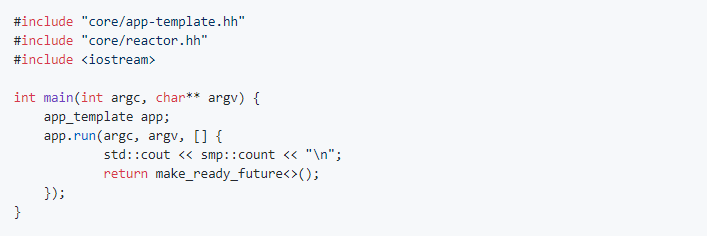
Mỗi Thread chạy  **event loop** của chính nó, được biết đến như là Engine trong thuật ngữ của Seastar

**By default, the Seastar application will take over all the available cores, starting one thread per core.**

Theo mặc định, Ứng dụng Seastar sẽ tiếp quản tất cả các core có sẵn, bắt đầu một thread trên mỗi core.

**We can see this with the following program, printing smp::count which is the number of started threads:**

Chúng ta có nhìn thấy điều này với chương trình sau, printing smp::count đó là số lượng các thread được bắt đầu:



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

Trên một máy với 4 luồng ở phần cứng( 2 core và chế độ hyperthreading được bật),

Theo mặc đinh, Seastar sẽ bắt đầu 4 thread engine:

$ ./a.out

4

**Each of these 4 engine threads will be pinned (a la taskset(1)) to a different hardware thread.**

Mỗi thread trong 4 engine thread sẽ được ghim(**a la taskset(1)**) để phân biệt với thread ở hardware.

**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.**

Như chúng tôi đã đề cập ở trên, Lưu ý làm thế nào mà, **the app's initialization function** chỉ chạy một thread, vì vậy chúng ta có thể thấy “4” chỉ một lần.

**Later in the tutorial we'll see how to make use of all threads.**

Phần sau trong tutorial, chúng ta sẽ biết làm thế nào để sử dụng tất cả thread.

**The user can pass a command line parameter, -c, to tell Seastar to start fewer threads than the available number of hardware threads.**

Người sử dụng có thể truyền vào một tham số dòng lệnh,-c, để nói với Seastar để start ít thread hơn so với số lượng thread hardware.

**For example, to start Seastar on only 2 threads, the user can do:**

Ví dụ, để start Seastar chỉ trên 2 threads, người sử dụng có thể thực hiện:

$ ./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).**

Khi machine được configuired như trong ví dụ ở trên – 2 cores với 2 hyperthreads và chỉ 2 thread được yêu cầu, Seastar đảm bảo rằng mỗi thread sẽ được ghim tới một core khác nhau, và chúng ta không có 2 luồng cạnh tranh nhau như hyperthreads của cùng một core( dĩ nhiên, nó sẽ gây hại đến hiệu năng).

**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:**

Chúng ta không start nhiều threads hơn số lượng của thread hardware, như cho chép điều đó sẽ rất kém hiệu quả. Cố gắng làm điều đó sẽ dẫn đến lỗi:

$ ./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.**

Lỗi này là một exception nhảy ra từ app.run, cái mà chúng ta không bắt lỗi, dẫn đến

**uncaught-exception crash** xấu xí này.

**It is better to catch this sort of startup exceptions, and exit gracefully without a core dump:**

Việc catch những exceptions khởi động này là tốt hơn, và thoát ra khỏi chúng một cách tinh tế,nhẹ nhafg mà không một dump core.

#include "core/app-template.hh"

#include "core/reactor.hh"

#include <iostream>

#include <stdexcept>

int main(int argc, char\*\* argv) {

app\_template app;

try {

app.run(argc, argv, [] {

std::cout << smp::count << "\n";

return make\_ready\_future<>();

});

} catch(std::runtime\_error &e) {

std::cerr << "Couldn't start application: " << e.what() << "\n";

return 1;

}

return 0;

}

$ ./a.out -c5

Couldn't start application: insufficient processing units

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

Lưu ý rằng việc Catch exception theo cách này ko bắt được exception được ném trong code bất đồng bộ thực tế của ứng dụng. Chúng ta sẽ thảo luận chúng trong phần sau của tutorial.

# Seastar memory

**As explained in the introduction, Seastar applications shard their memory.**

Như đã giải thích trong phần giới thiệu, Ứng dụng Seastar chia sẻ memory của chúng.

**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).**

Mỗi thread được phân bổ với một phần lớn của bộ nhớ ( trên cùng một node NUMA mà nó đang chạy), và chỉ sử dụng memory đó cho **allocations** của nó (như là malloc hoặc new).

**By default, the machine's entire memory except a small reservation left for the OS (defaulting to 512 MB) is pre-allocated for the application in this manner.**

Theo mặc định, entire memory trừ ra một vùng nhớ nhỏ còn lại cho OS(mặc định là 512 MB) được pre-allocated cho ứng dụng trong manner.

**This default can be changed by eitherchanging 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.**

Mặc định này có thể thay đổi bằng cách thay đổi vùng nhớ dành cho hệ điều hành với lựa chọn **--reserve-memory,** hoặc bằng cách cung cấp rõ ràng vùng nhớ của memory được cấp cho application Seastar, với option –m.

**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).**

Dung lượng bộ nhớ có thể theo Byte hoặc sử dụng KB,MB,GB,TB.

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

Cố gắng cấp cho Seastar nhiều vùng nhớ hơn physical memory dẫn đến lỗi ngay lập tức.

$ ./a.out -m10T

Couldn't start application: insufficient physical memory

# Introducing futures and continuations

**Futures and continuations, which we will introduce now, are the building blocks of asynchronous programming in Seastar.**

Futures và continuation , những thứ mà chúng tôi sẽ đề cập ngay bây giờ, là những **building blocks** của lập trình bất đồng bộ trong Seastar.

**Their strength lie in the ease of composing them together into a large, complex, asynchronous program, while keeping the code fairly readable and understandable.**

Sức mạnh của chúng nằm ở sự dễ dàng kết hợp chúng lại thành một chương trình bất đồng bộ lớn và phức tạp, trong khi vẫn giữ được code khá dễ đọc và dễ hiểu.

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

Một Future là một kết quả của sự tính toán có thể chưa có sẵn. Những ví dụ bao gồm:

* **a data buffer that we are reading from the network**

Một buffer dữ liệu mà chúng ta đang đọc nó từ network.

* **the expiration of a timer**

Hết hiệu lực của một timer.

* **the completion of a disk write**

Một sự hoàn thành của quá trình ghi ra đĩa.

* **the result of a computation that requires the values from one or more other futures.**

Kết quả của một sự tính toán mà yêu cầu giá trị từ một hoặc nhiều Future khác.

**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.**

Biến kiểu future<int> giữ một int mà nó đến cuối cùng sẽ có sẵn, tại thới điểm này có thể giá trị này đã có sẵn hoặc chưa.

**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.**

Phương thức avaible() kiểm tra nếu một giá trị là có sẵn, và phương thức get() sẽ lấy giá trị đó. Kiểu future<> chỉ ra rằng cái gì đó cuối cùng sẽ hoàn thành, nhưng không trả về bất kỳ giá trị nào.

**A future is usually returned by a promise, also known as an asynchronous function, a function or object which returns a future and arranges for this future to be eventually resolved.**

Một future thường được trả về bởi một promise, cũng được biết đến như là một hàm bất đồng bộ, một hàm hoặc một đối tượng mà trả về một future và sắp xếp cho Future này kiểu gì cũng được giải quyết.

**One simple example is Seastar's function sleep():**

Một ví dụ đơn giản là một hàm sleep() của Seatar:

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.**

Function này sắp xếp một timer để Future được trả về trở nên khả dụ (với không giá trị nào được trả về) khi một khoảng thời gian nhất định trôi qua.

**A continuation is a callback (typically a lambda) to run when a future becomes available.**

**Một continuation là một hàm callback (điển hình là một lambda) để thực thi khi mà một future trở nên khả dụ.**

**A continuation is attached to a future with the then() method. Here is a simple example:**

Một ontinuation được đính kém với một Future với phương thức then(). Dưới đây là một ví dụ đơn giản:”

#include "core/app-template.hh"

#include "core/sleep.hh"

#include <iostream>

int main(int argc, char\*\* argv) {

app\_template app;

app.run(argc, argv, [] {

std::cout << "Sleeping... " << std::flush;

using namespace std::chrono\_literals;

return sleep(1s).then([] {

std::cout << "Done.\n";

});

});

}

In this example we see us getting a sleep(1s) future, and attaching to it a continuation which prints a message and exits. 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.

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 the function future<> f(), and does the appropriate exception handling:

#include "core/app-template.hh"

#include <iostream>

#include <stdexcept>

extern future<> f();

int main(int argc, char\*\* argv) {

app\_template app;

try {

app.run(argc, argv, f);

} catch(std::runtime\_error &e) {

std::cerr << "Couldn't start application: " << e.what() << "\n";

return 1;

}

return 0;

}

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

#include "core/sleep.hh"

#include <iostream>

future<> f() {

std::cout << "Sleeping... " << std::flush;

using namespace std::chrono\_literals;

return sleep(1s).then([] {

std::cout << "Done.\n";

});

}

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

#include "core/sleep.hh"

#include <iostream>

future<> f() {

std::cout << "Sleeping... " << std::flush;

using namespace std::chrono\_literals;

sleep(200ms).then([] { std::cout << "200ms " << std::flush; });

sleep(100ms).then([] { std::cout << "100ms " << std::flush; });

return sleep(1s).then([] { std::cout << "Done.\n"; });

}

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

$ ./a.out

Sleeping... 100ms 200ms Done.

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

#include "core/sleep.hh"

#include <iostream>

future<int> slow() {

using namespace std::chrono\_literals;

return sleep(100ms).then([] { return 3; });

}

future<> f() {

return slow().then([] (int val) {

std::cout << "Got " << val << "\n";

});

}

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

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

## Ready futures

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

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

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

#include "core/future.hh"

#include <iostream>

future<int> fast() {

return make\_ready\_future<int>(3);

}

future<> f() {

return fast().then([] (int val) {

std::cout << "Got " << val << "\n";

});

}