**Introduction**

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

Seatar, cái mà chúng tôi giới thiệu trong document này, là một thư viện C++ cho việc viết các ứng dụng server phức tạp hiệu quả cao trong các máy multi-core hiện đại.

**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.**

Theo truyền thống, các thư viện và frameworks trong các ngôn ngữ lập trình – mà được sử dụng cho việc viết các ứng dụng server được chia thành 2 phe riêng biệt: 1 bên thì tập trung vào sự hiệu quả, 1 bên tập trung vào sự phức tạp.

# 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. Chú thích: at the cost of

## Definition of at the cost of

**:**by giving up or hurting (something else):She completed the project on time but at the cost of her health.

Một số frameworks cực kì hiệu quả và chỉ cho phép việc buid các ứng dụng đơn giản( ví dụ DPDK cho phép các ứng dụng mà thực thi các packets một cách riêng lẻ), trong khi các frameworks khác cho phép việc buid các ứng dụng cực kì phức tạp, nhưng phải trả giá là sự hiệu quả lúc run-time.

**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.**

Seastar là lỗ lực của chúng tôi để lấy thứ tốt nhất của cả hai thế giới: tạo ra một thư viện – cho phép việc buid các ứng dụng server có độ phức tạp cao, và đạt được hiệu xuất tối ưu.

**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.**

In "**and yet**", **yet** usually means "in spite of". " And" means "in addition to

# as much as

**idiom (1)**

## Definition of as much as

**(Entry 1 of 2)**

**1**—used to say that two things are equal in amount or degree

He likes hockey as much as he likes basketball.

**2**—used to say that an amount is as large as another amount

She earns as much as he does.

**3:**almost but not quite

He as much as admitted his guilt.

**as much as**

**idiom (2)**

**Definition of as much as (Entry 2 of 2)**

**:**even though **:**despite the fact that

As much as I respect him, I still have to disagree with him on this point.

Cảm hứng và use case đầu tiên của Seastar là Scylla, một bản viết lại của Apache Cassandra. Cassandra là một ứng dụng rất phức tạp, thêm vào đó, với Seastar chúng ta có thể triển khai lại nó với thông lượng tăng 10 lần, cũng như độ trễ thấp hơn đáng kể và ổn định hơn

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).

**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 mmaped 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.