





SEASTAR @ NYCC++UG  
Avi Kivity (@AviKivity)  
April 2016

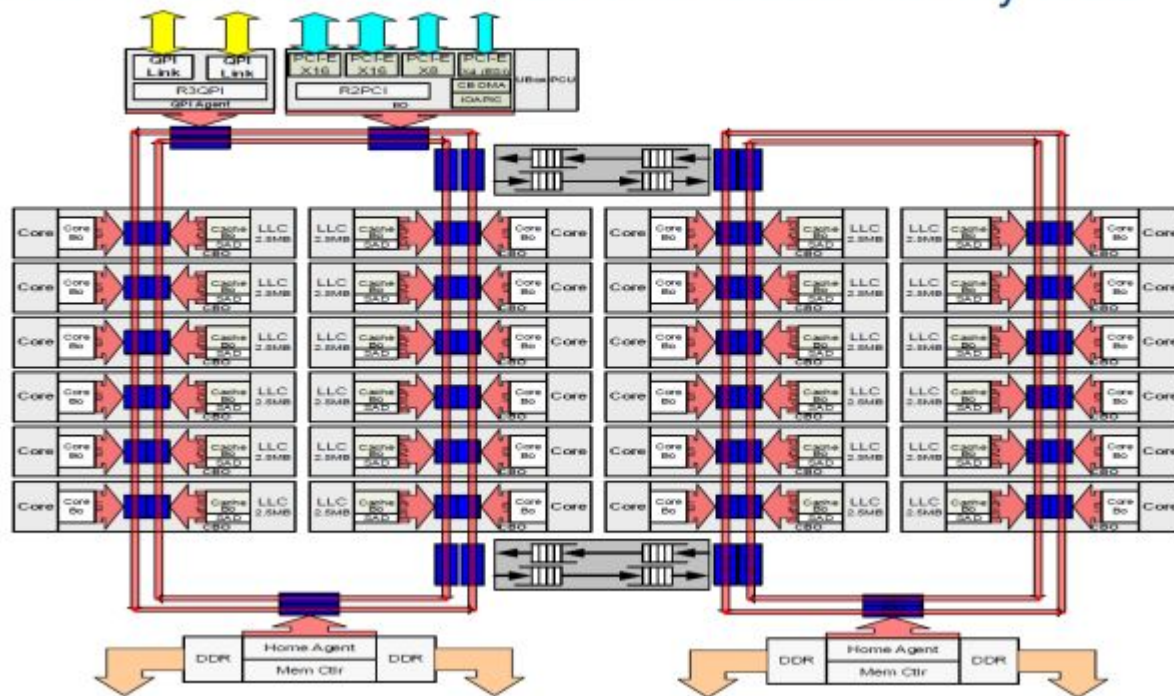


# Seastar: A C++ Asynchronous Programming Framework





## Intel® Xeon® Processor E5 v4 Product Family HCC





# Multi-domain async programming

Async networking

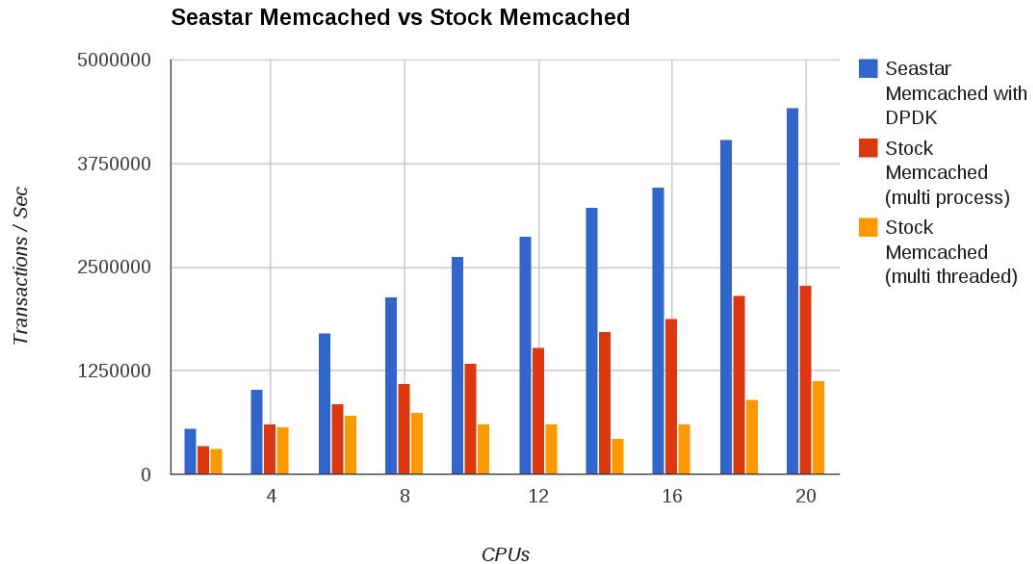
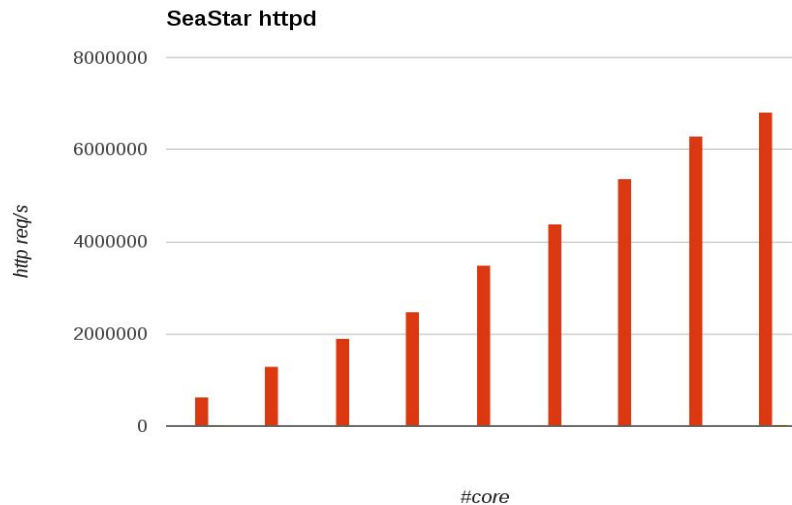
Async storage I/O

Async communications for multi-core, NUMA



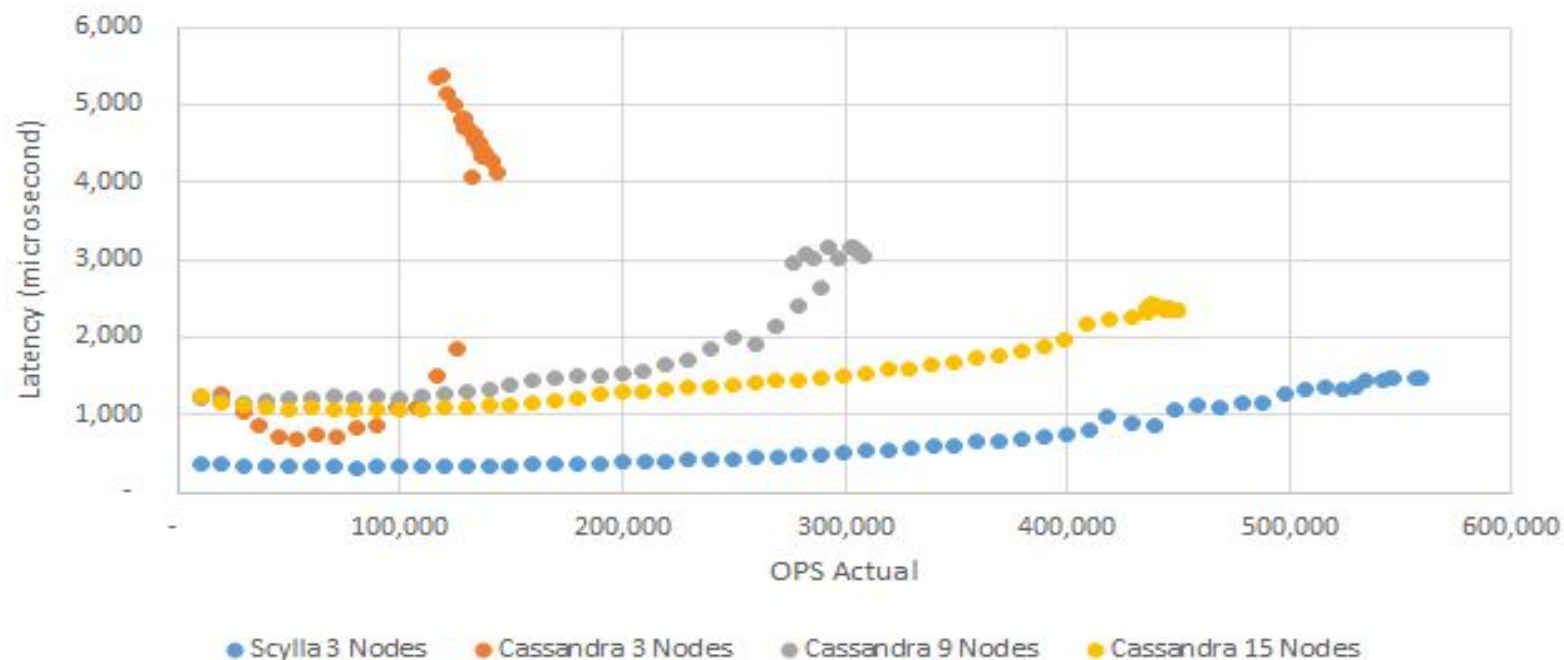


# RESULTS





## Read Latency Comparison





# THREADING MODELS

Before: Thread model

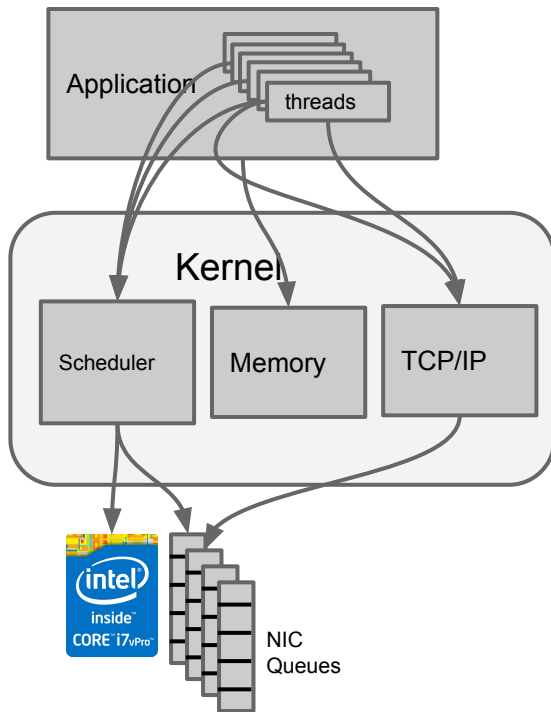


After: SeaStar shards

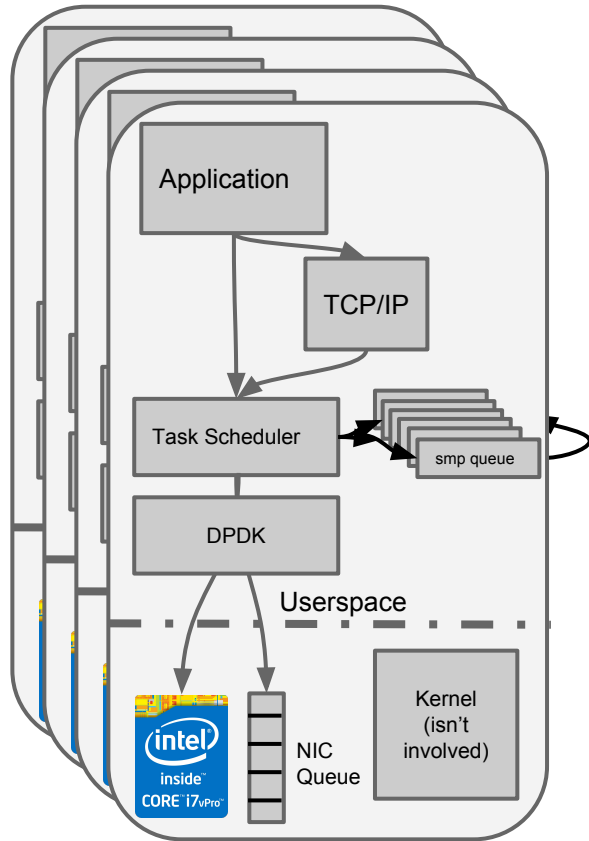




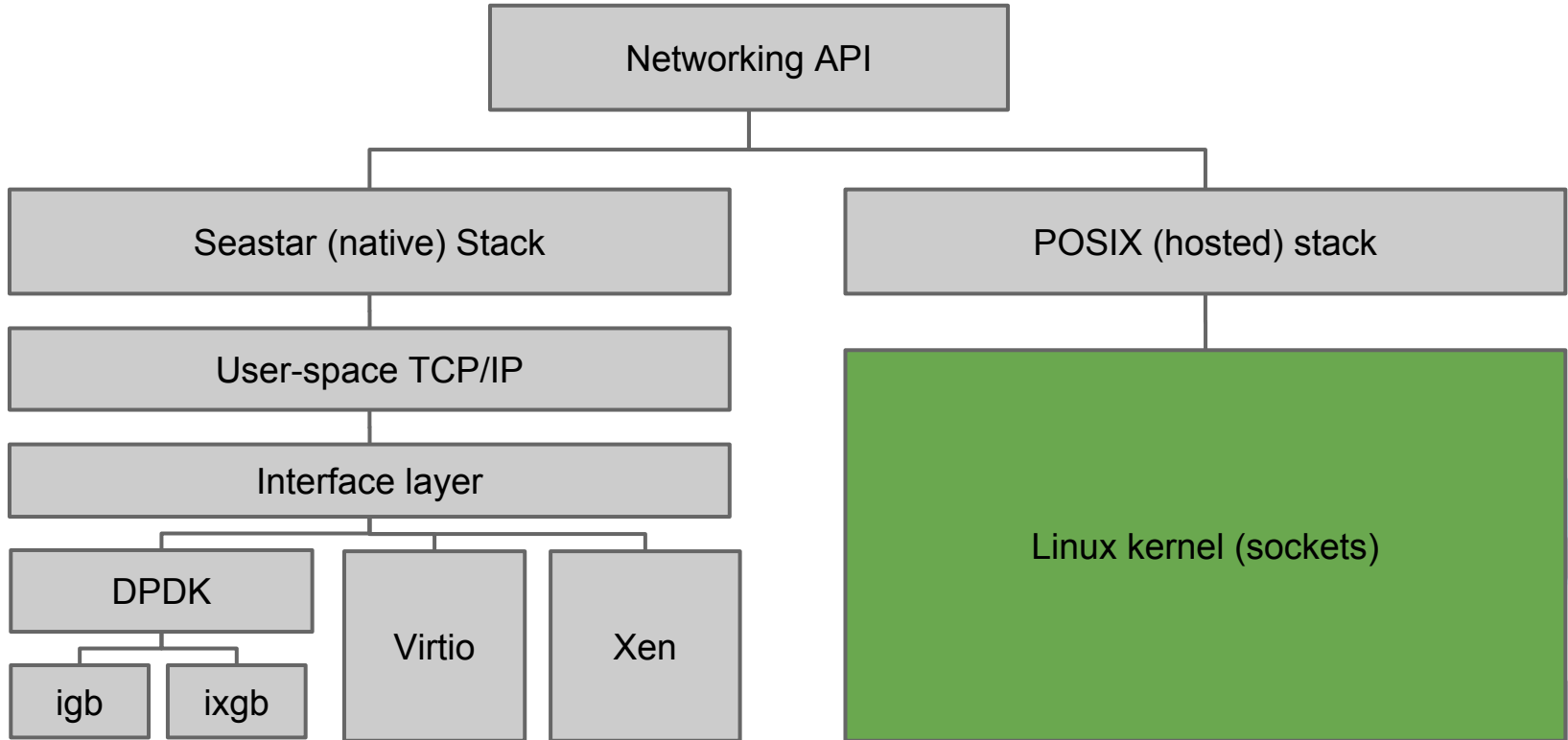
# Traditional threading model



# Seastar model



# Dual networking stacks





# Seastar model summary

- Each logical core runs a shared-nothing run-to-completion task scheduler
- Logical cores connected by point-to-point queues
- Explicit core-to-core communication
- Shard owns data
- Composable Multicore/Storage/Network APIs
- Optional userspace TCP/IP stack





CODING IT:  
Futures and promises



# BASIC MODEL

- Futures
- Promises
- Continuations





# F-P-C Defined: Future

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

- Data buffer from the network
- Timer expiration
- Completion of a disk write
- Computation on another core
- Result computation that requires the values from one or more other futures.







# F-P-C Defined: Promise

A promise is an object or function that provides you with a future, with the expectation that it will fulfil the future.





# F-P-C Defined: Continuation

A continuation is a computation that is executed when a future becomes ready (yielding a new future).





# Basic Future/Promise

```
future<int> get(); // promises an int will be produced eventually
```

```
future<> put(int) // promises to store an int
```

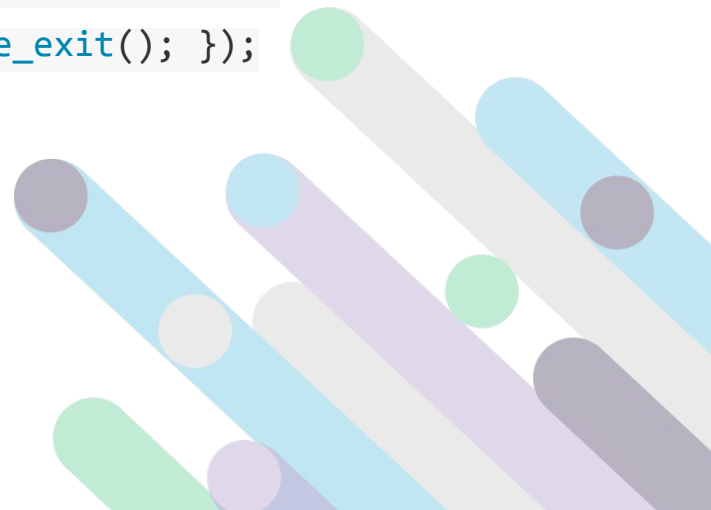
```
future<> f() {  
    return get().then([] (int value) {  
        return put(value + 1).then([] {  
            std::cout << "value stored successfully\n";  
        });  
    });  
}
```





# Parallelism

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





# Zero-copy

```
future<temporary_buffer<char>> connected_socket::read(size_t n);
```

temporary\_buffer points at driver-provided pages if possible  
discarded after use





# Rich APIs

- HTTP Server
- HTTP Client
- RPC client/server
- map\_reduce
- parallel\_for\_each
- iostreams
- iosched
- threads!
- sharded<>
- when\_all()
- timers
- sleep
- semaphore
- gate
- pipe/queue
- Memory reclaimer



USE CASES



# Applicability

- High I/O to compute ratio
- High concurrency
- Mix of disk and network I/O
- Complex loads
- Cluster (sharded) applications







# Applicability

- Distributed databases
- Object stores, file systems
- Complex proxies/caches





## MORE INFORMATION

<http://github.com/scylladb/seastar>

<http://seastar-project.com>

<http://docs.seastar-project.com>

<https://github.com/scylladb/seastar/wiki/Seastar-Tutorial>

[@ScyllaDB](#)





Thank you.