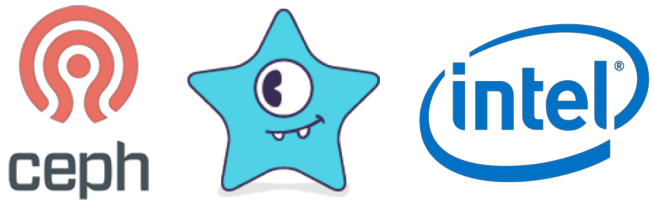
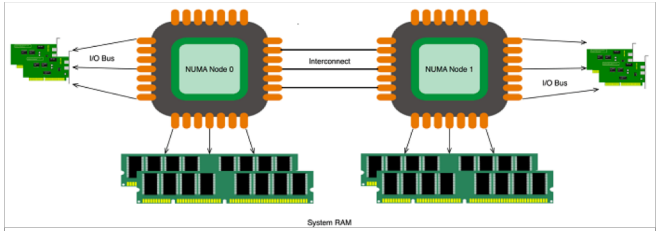
A glimpse of the new Ceph messenger built on seastar

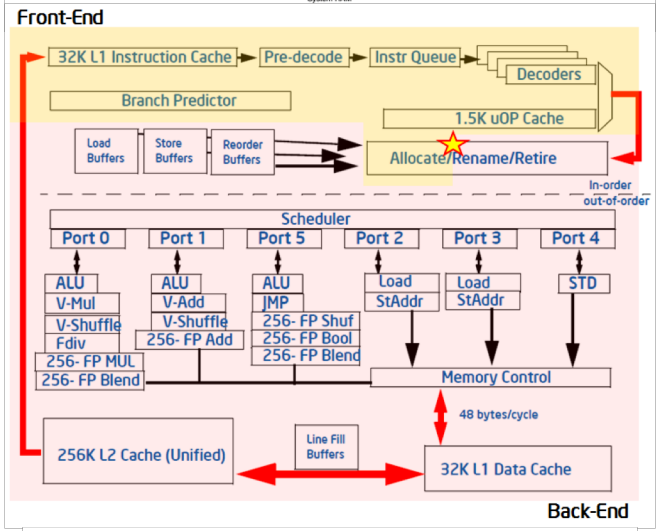
Yingxin Cheng (yingxin.cheng@intel.com)



Goal of Crimson

- Optimal performance on modern machines
  - Multi-core (NUMA)
    - Practically scalable: Nodes, IO buses
    - Resources: Cache, Memory, PCI devices
  - CPU pipeline
    - Out-of-order execution, instruction-level parallelism
    - Front-end / Back-end
  - High-speed devices
    - Storage: NVMe SSD, Persistent memory
    - Network: 100 Gbps
- Ceph architecture
  - Shared data
  - POSIX threads
  - Blockings





Why SEASTAR.

- Modern hardware friendly
  - Multi-core (NUMA)
    - Linear performance improvements by adding cores
    - Forces lock-free & share-nothing design
    - NUMA-aware memory allocation
  - CPU pipeline
    - User-space task scheduling
    - Explicit message passing between cores
  - High-speed devices
    - Pure asynchronous event-driven: epoll
    - DPDK, DMA for network & disk I/O

Why SEASTAR.

- C++ programming language:
  - Full control:
    - memory allocation, layout
    - lock-free synchronization primitives with memory-ordering
    - low-level optimizations
  - Zero run-time overhead:
    - sophisticated compile-time code generation and optimization
  - Modern language semantics
- Higher-level abstraction for asynchronous programming
  - futures and continuation
  - Sharded data structures, 0-copy utilities
- Well tested and validated
  - Cassandra/ScyllaDB, Kafka/smf, Redis/Pedis, Httpd, Memcached
  - Published results

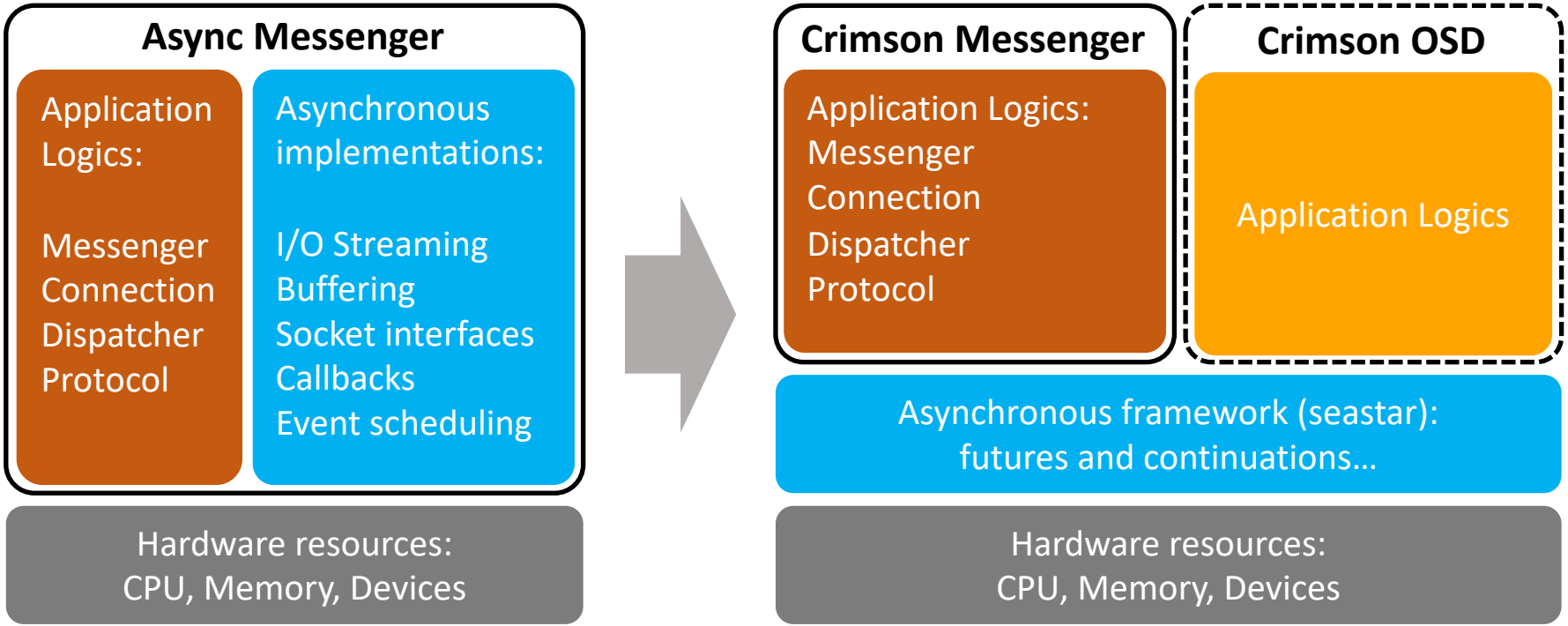
Goals of Crimson messenger

1. Uniform architecture
2. Core-to-core connection
3. Performance with seastar
4. Better code

1. Uniform architecture
Async messenger design

	Async Messenger	Seastar framework
Event-driven asynchronous I/O	EventDriver • EpollDriver • KqueueDriver • SelectDriver	reactor_backend • reactor_backend_epoll • reactor_backend_aio • reactor_backend_osv
User-space scheduling	Worker/EventCenter • Poller • EventCallback	reactor engine • poller • task
High-speed devices	NetworkStack • PosixNetworkStack • DPDKStack • RDMAStack	network_stack • posix_(ap_)network_stack • native_network_stack

1. Uniform architecture
Crimson messenger design

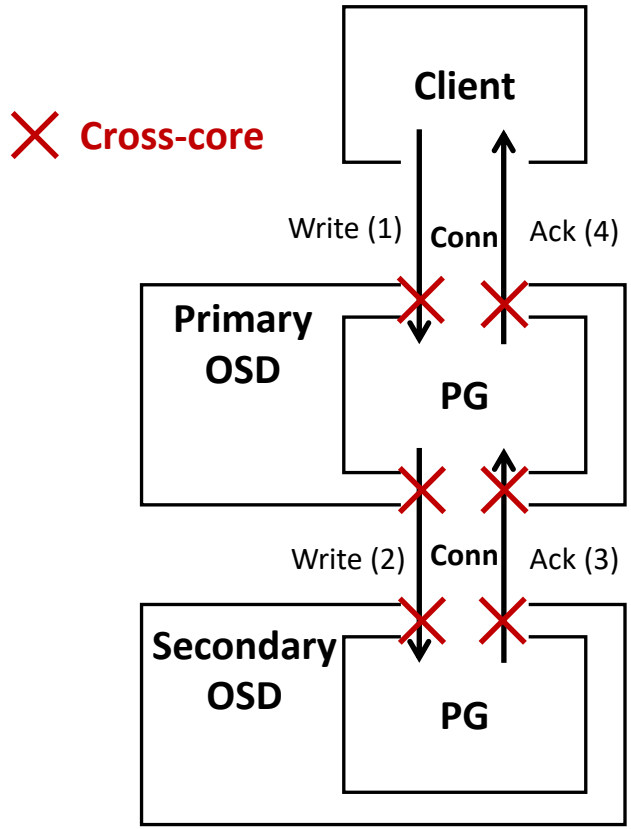


2. Core-to-core connection
Current design

Cross-core not good
- Less opportunity close to data
- Tasks submitted and queued
- Ownership managements

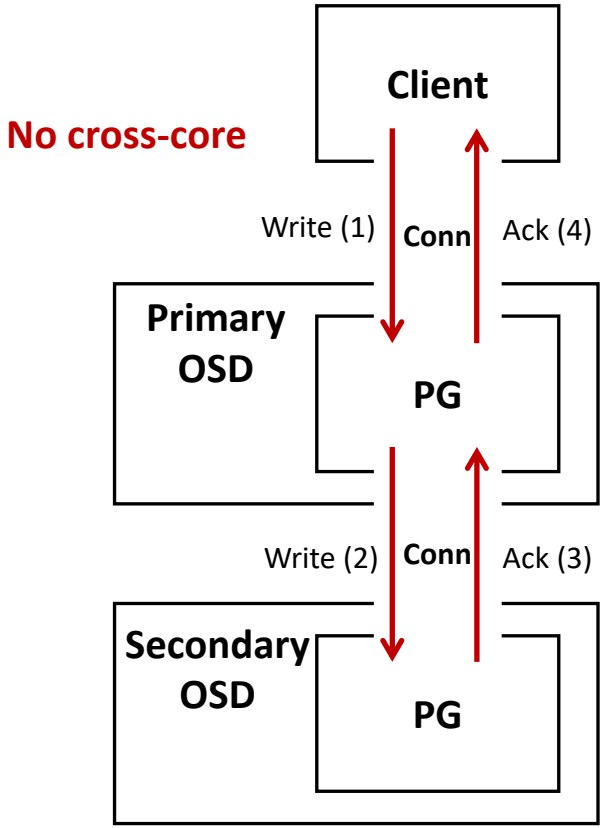
Classical: service-to-service
- Connection is per OSD <-> OSD / Client <-> OSD
- OSD is a multi-core service

With seastar share-nothing design
- Connection, PG living on fixed cores
- Client can access any PG
- Primary OSD replicates all writes to Secondary

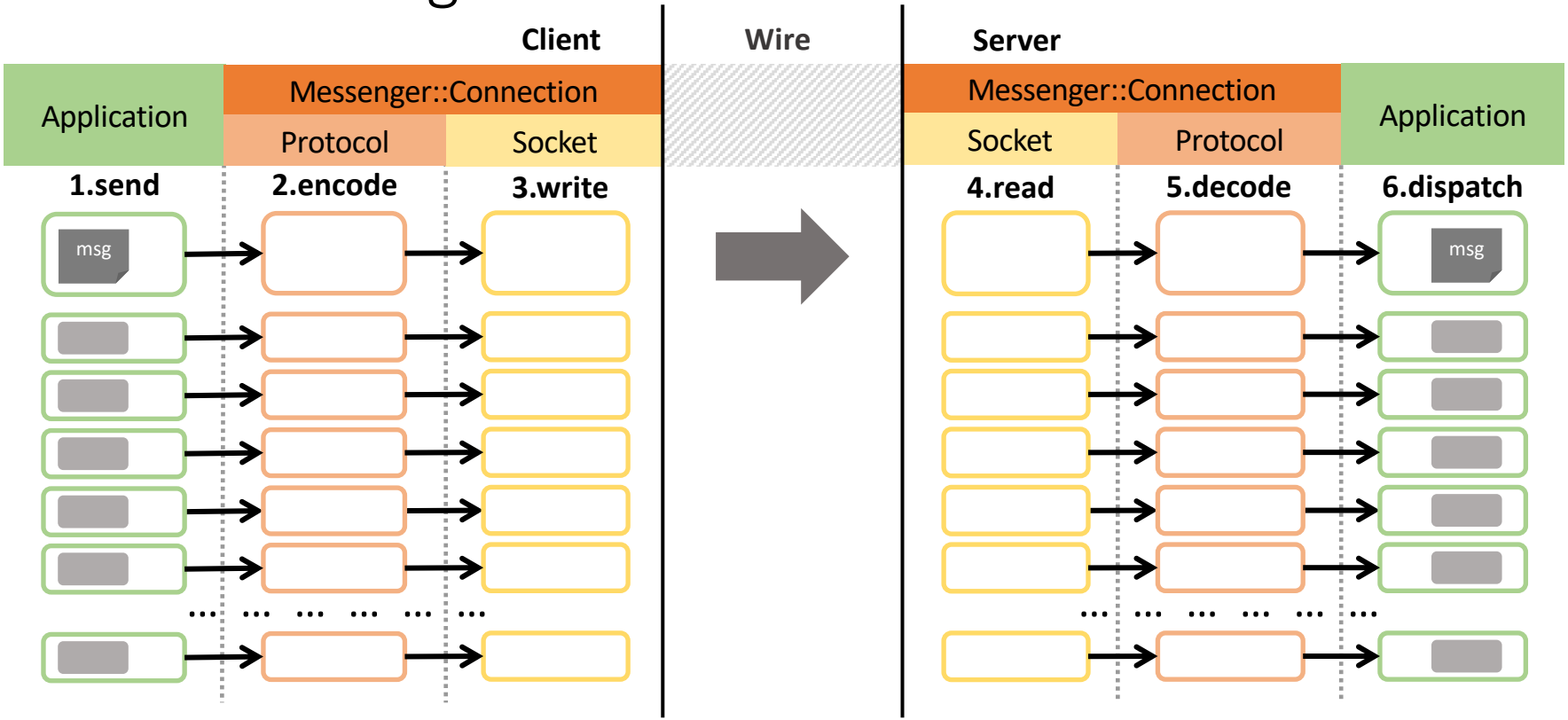


2. Core-to-core connection Design choices

- Single-core Dispatcher (current approach)
  - Messenger allocate the connection to the OSD core
- Multiple-core Dispatcher
  - Messenger expose all cores
  - Connection is per PG shard <-> PG shard
  - Require new protocol, backward compatibility
  - More CPUs to drive faster storage
  - More shared resources
- Extensive to different connection placement strategies

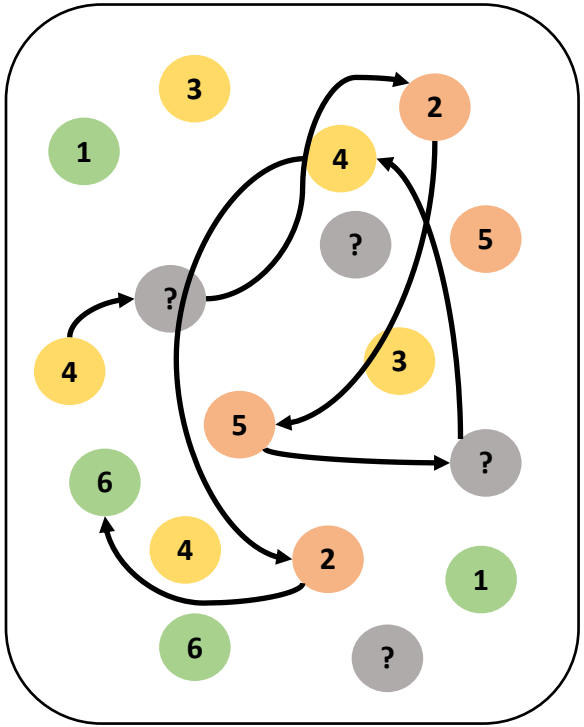


3. Performance with seastar Messenger workloads



3. Performance with seastar
Initial design

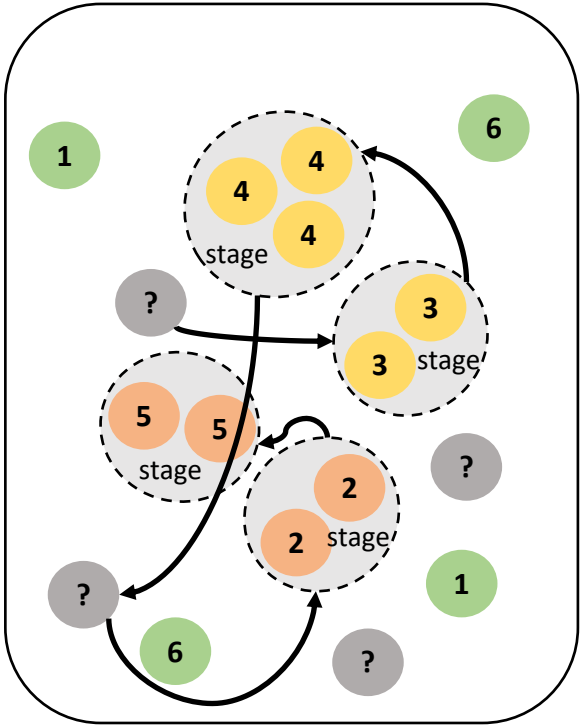
Reactor view



Keep the order

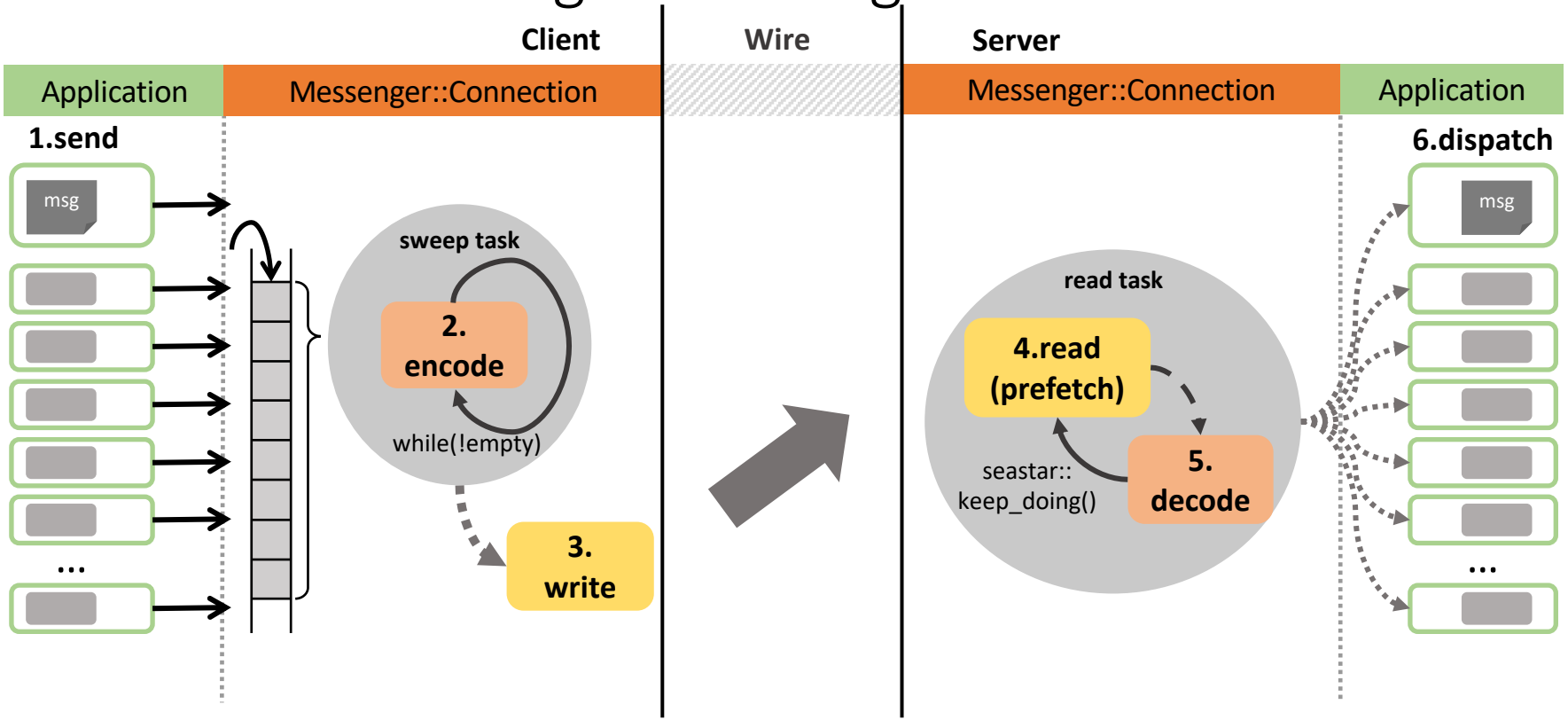
```
291 seastar::future<>¬
292 SocketConnection::do_send(MessageRef msg)¬
293 {¬
294   // chain the message¬
295   // after the last message is sent¬
296   seastar::shared_future<> f =¬
297   send_ready.then(¬
298   [this, msg=std::move(msg)] {¬
299     if (state == state_t::closing) {¬
300       return seastar::now();¬
301     }¬
302     return write_message(std::move(msg));¬
303   });¬
304 ¬
305   // chain any later messages¬
306   // after this one completes¬
307   send_ready = f.get_future();¬
308 ¬
309   // allow the caller to¬
310   // wait on the same future¬
311   return f.get_future();¬
312 }¬
```

Batching

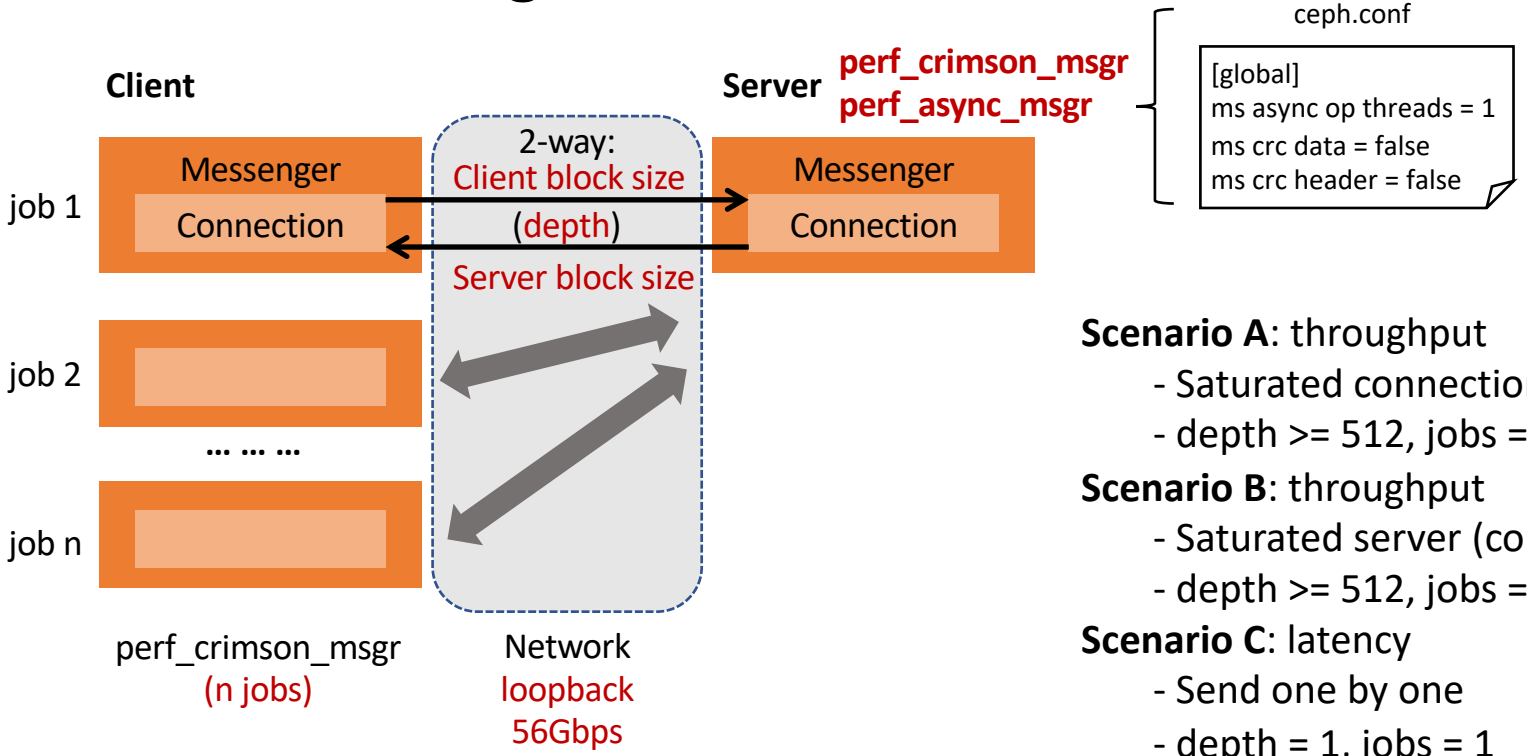


3. Performance with seastar

Current design: batching & minimize tasks



3. Performance with seastar
Test settings



Scenario A: throughput
- Saturated connection
- depth >= 512, jobs = 1

Scenario B: throughput
- Saturated server (core)
- depth >= 512, jobs = 4

Scenario C: latency
- Send one by one
- depth = 1, jobs = 1

3. Performance with seastar
Comparison results

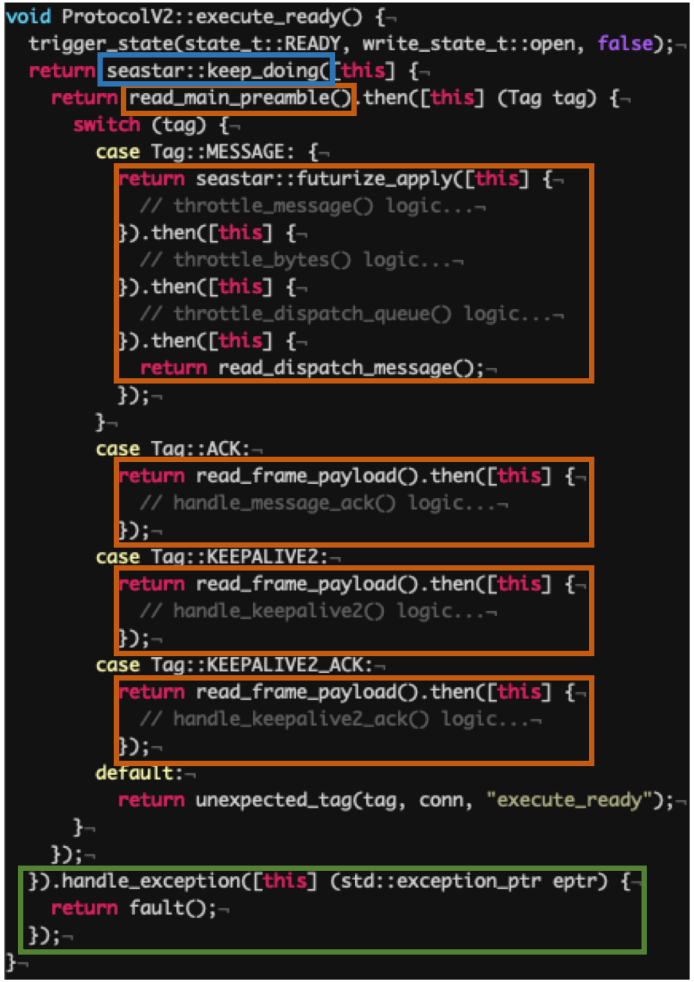
Scenarios (2-way, r+w)	Block size	56Gbps network			loopback		
		Crimson server	Async server	(>1: better)	Crimson server	Async server	(>1: better)
		Throughput-KIOPS		Cr/As	Throughput-KIOPS		Cr/As
A. Saturated connection (depth>=512, jobs=1)	256K	284.2	177.6	*1.600225225*	297.8	207.5	*1.435180723*
	4K	186.1	109.7	*1.69644485*	229.7	143.3	*1.602930914*
	64K	23.69	23.45	*1.010234542*	39.81	39.62	*1.004795558*
	1M	1.883	1.36	*1.384558824*	3.394	1.304	*2.602760736*
B. Saturated core (depth>=512, jobs=4)	256K	294.25	177.8	*1.654949381*	268.5	212.5	*1.263529412*
	4K				231.9	145.2	*1.597107438*
	64K				38.57	39.16	*0.984933606*
	1M	1.87	1.436	*1.302228412*	2.661	2.544	*1.045990566*
		latency-us		As/Cr	latency-us		As/Cr
C. Latency (depth=1, jobs=1)	256K	40.8			24.85	33.07	*1.330784708*
	4K	46.6	9.3	*1.057939914*	26.89	38.9	*1.446634437*
	64K	141.4	133.5	*0.944130127*	64.34	70.93	*1.102424619*
	1M	1063	1242	*1.168391345*	365.2	852.7	*2.334884995*

Native-stack...

Alignment...

4. Better code

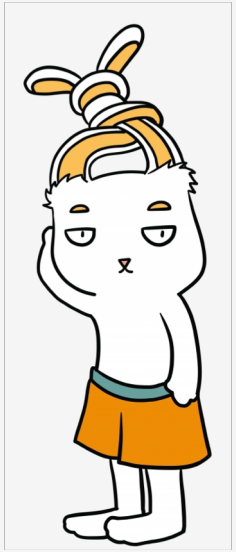
- Chained continuations
  - lambda functions, then()
- Continuation-level semantics
  - Organize: keep_doing(), repeat(), map_reduce()
  - Preconditions: when_all(), repeat_until_value()
- Exception handling
  - handle_exception(), handle_exception_type()
  - finally()
- Resource limit & balance
  - CPU shares: scheduling_group
  - Memory & I/O bandwith: condition_variable, semaphore
- Tool-chain
  - gate, smart ptr, sharded<>, clocks…
  - chunked_fifo, queues, circular buffers…
- Rethink the design during rewrite …



```
void ProtocolV2::execute_ready() {
  trigger_state(state_t::READY, write_state_t::open, false);
  return seastar::keep_doing([this] {
    return read_main_preamble().then([this] (Tag tag) {
      switch (tag) {
        case Tag::MESSAGE: {
          return seastar::futurize_apply([this] {
            // throttle_message() logic...
          }).then([this] {
            // throttle_bytes() logic...
          }).then([this] {
            // throttle_dispatch_queue() logic...
          }).then([this] {
            return read_dispatch_message();
          });
        }
        case Tag::ACK:
          return read_frame_payload().then([this] {
            // handle_message_ack() logic...
          });
        case Tag::KEEPALIVE2:
          return read_frame_payload().then([this] {
            // handle_keepalive2() logic...
          });
        case Tag::KEEPALIVE2_ACK:
          return read_frame_payload().then([this] {
            // handle_keepalive2_ack() logic...
          });
        default:
          return unexpected_tag(tag, conn, "execute_ready");
      }
    });
  }).handle_exception([this] (std::exception_ptr eptr) {
    return fault();
  });
}
```

Caveats

- Extensive
  - InputStream interface
  - Socket placement
- Complexity of the new programming model
  - Futures and Continuations
  - Lockless
  - Sharded data structures
  - Cross-core communication
  - Keep up with the changes
- Imbalance between the cores
- Performance
- Debugging

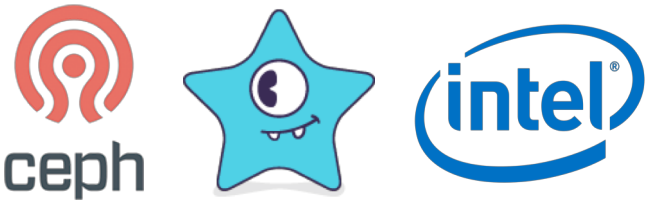


Status and plans

- Current
  - Lossy policy for msgr v1 & v2 protocol
- Complete features
  - Lossless policy
  - Loopback
  - …
- Improve reliability with validation tools
- Seastar
  - Required modifications
  - Native-stack enabling
- Async messenger improvements
  - Performance
  - Architecture

Thank you!

yingxin.cheng@intel.com



Goal of Crimson

Why seastar?

Goals of Crimson messenger

Caveats

Status and plans