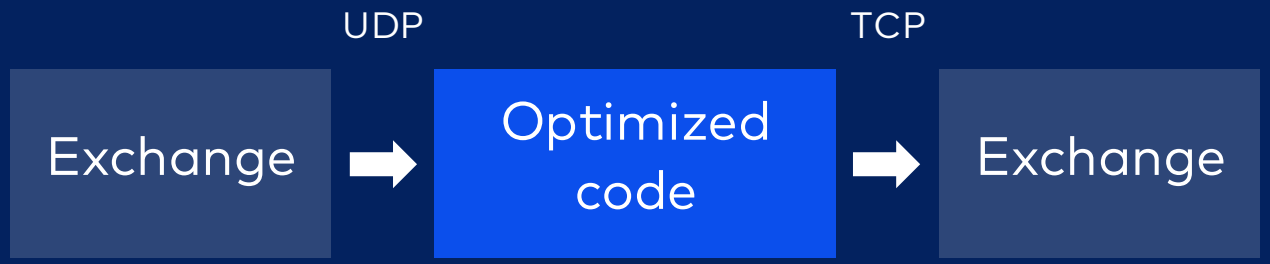
imc

LOW-LATENCY IN A MULTI-THREADED WORLD

Martijn Terpstra

Execution Team

MULTI-THREADING IS BAD



CONCURRENCY IS BAD

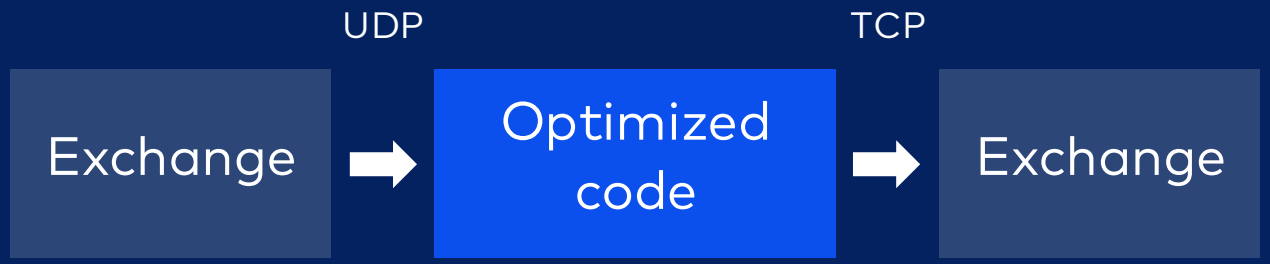
- Bad for latency
- Dangerous
- Hard

```
class OrderIdCache {
public:
    // ...

private:
    SpinLock pendingLock_;
    std::vector<int> pending_;
    std::unordered_map<int, int> mappings_;
    bool dirty_{false};
};
```

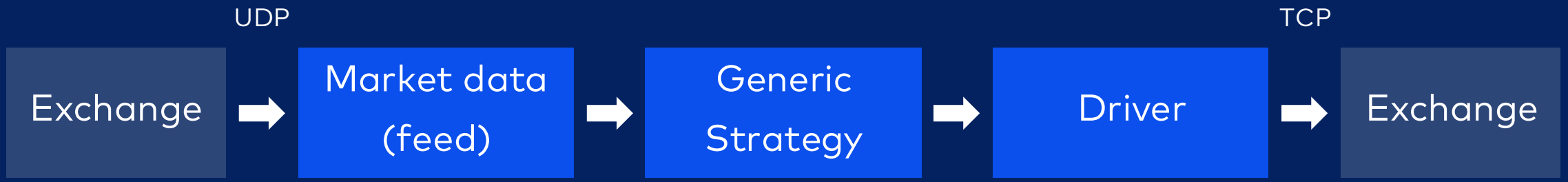
GUIDELINE NUMBER 1

Optimize before multi-threading

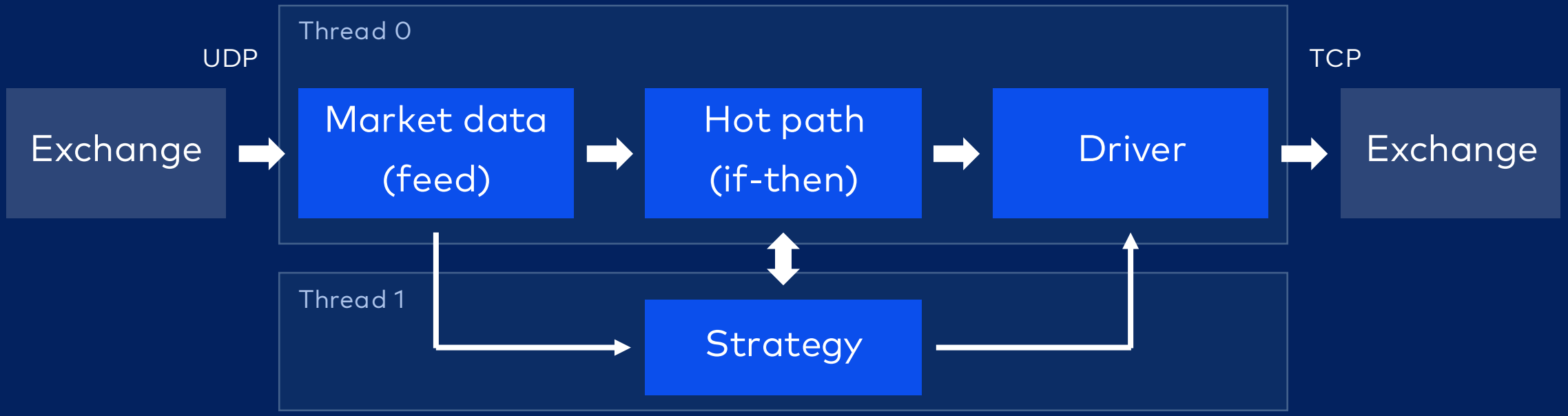
YOUR AWESOME STRATEGY



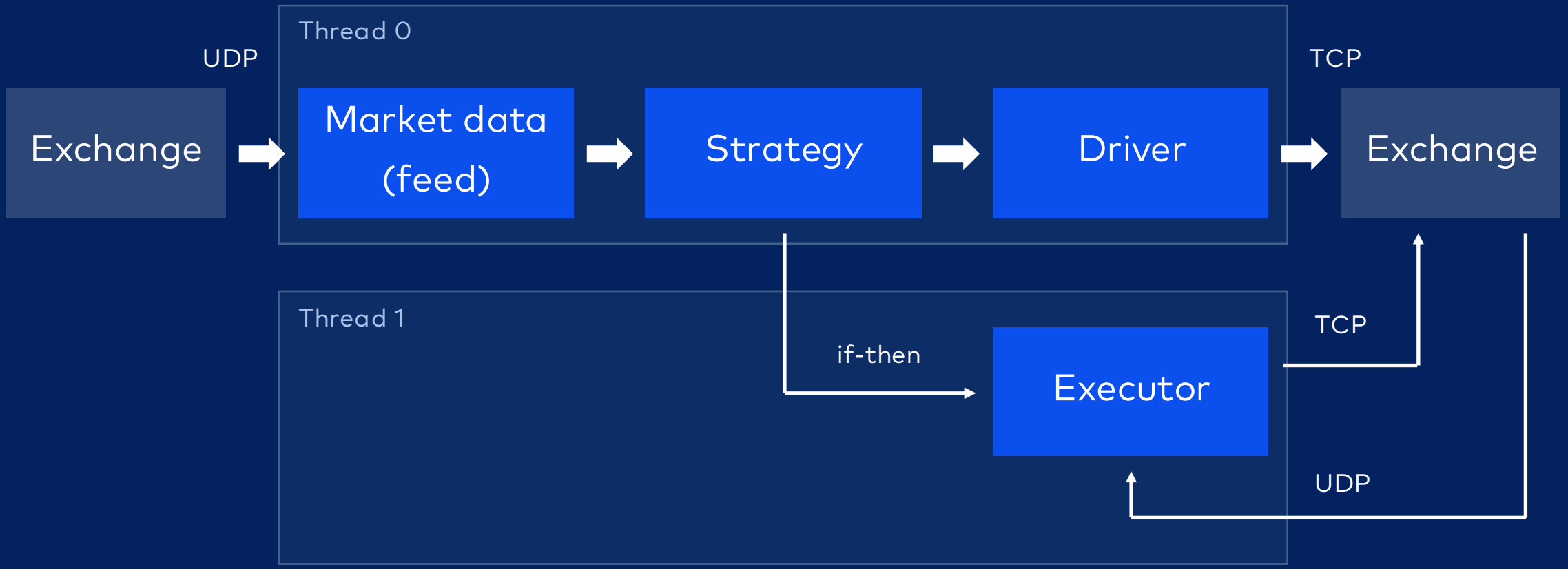
X YEARS LATER



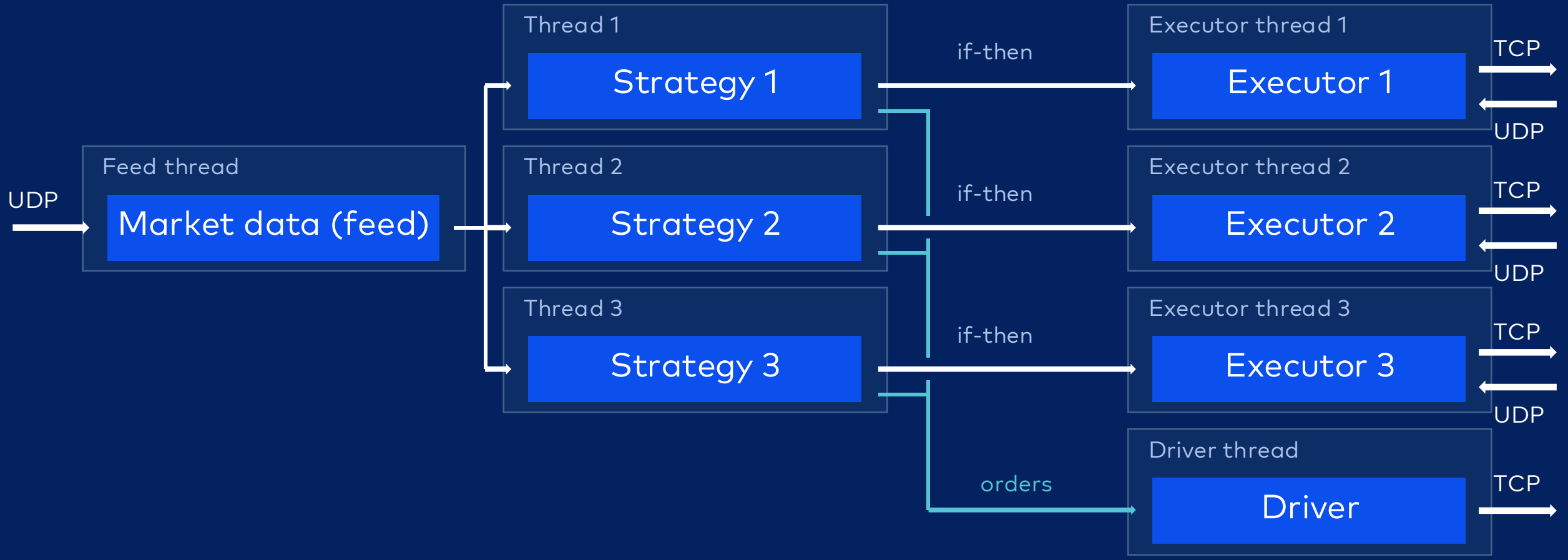
ISOLATE HOT PATH



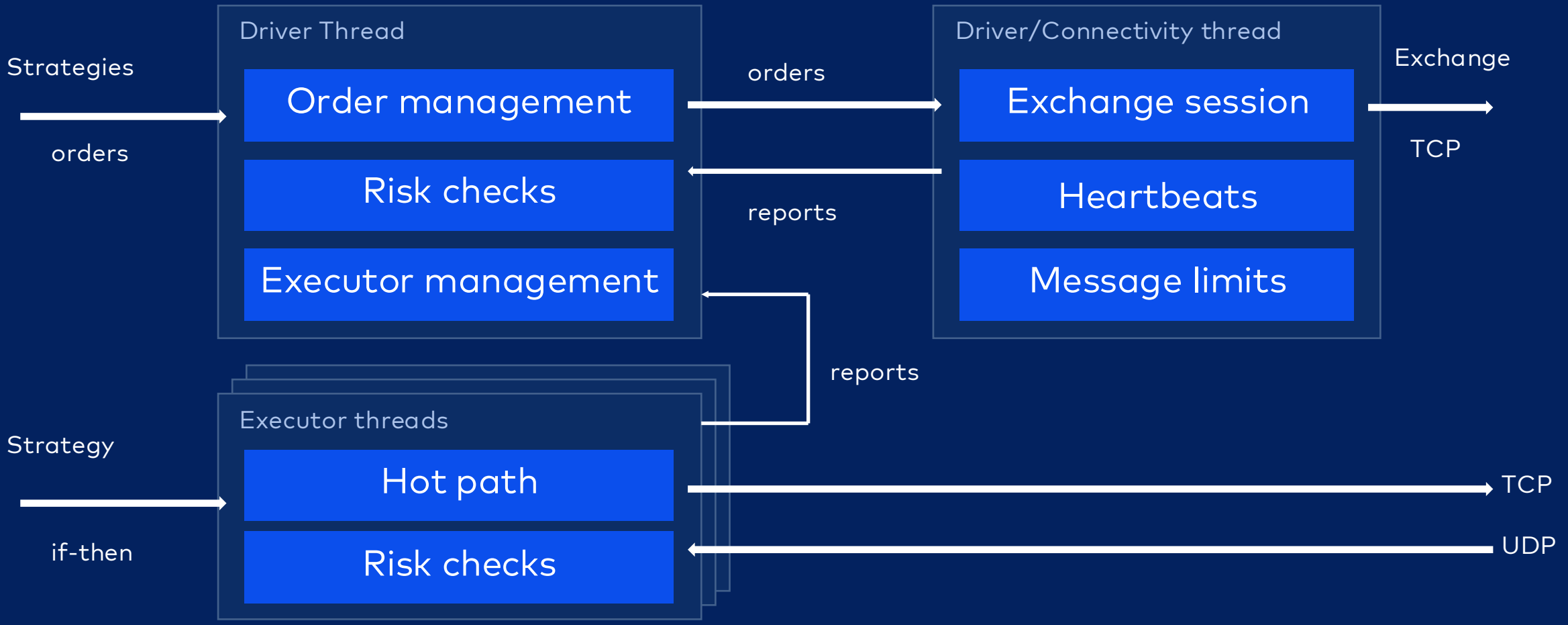
OPTIMAL: EXECUTORS



SCALING THE APPLICATION



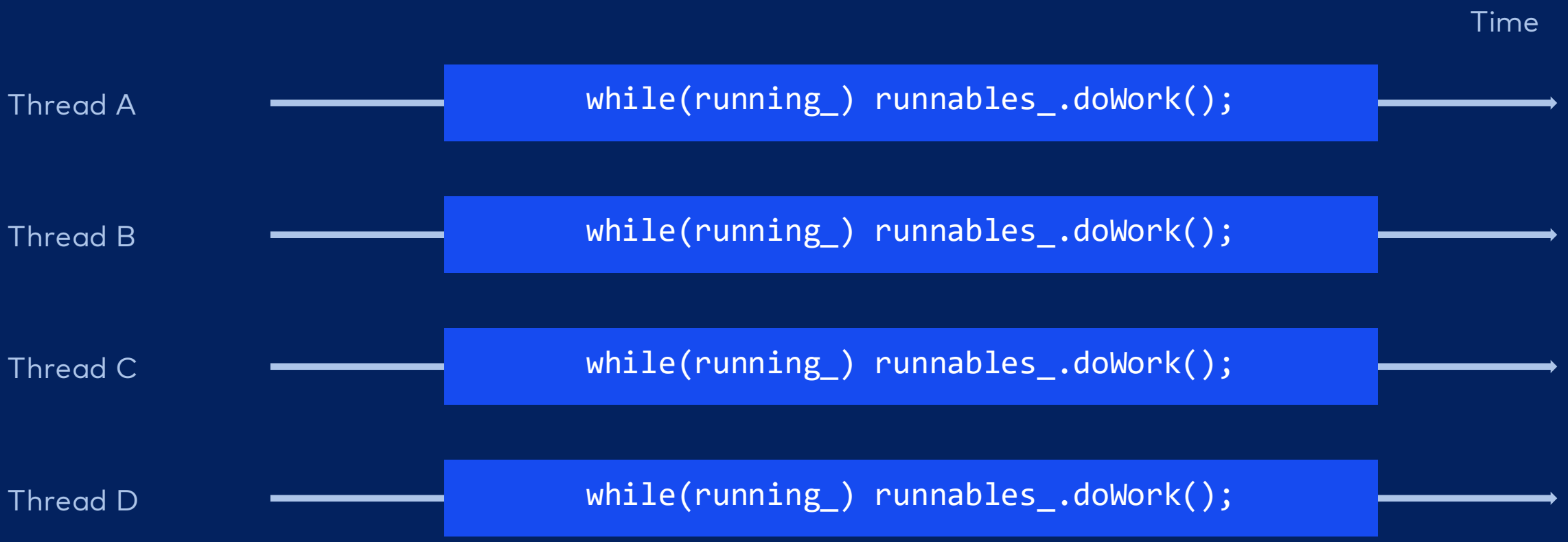
INSIDE THE DRIVER



GOALS

- Fast
- Different scenarios: 1 - 32 cores
- Simple to use, hard to misuse

THREAD RUNNERS



RUNNABLES

```
class Runnable {
public:
    virtual void doWork() = 0;
};

class ThreadRunner {
public:
    void run();

private:
    std::vector<Runnable*> runnables_;
};
```

```
class NetworkManager : public Runnable {
public:
    void doWork() override {
        // Poll network I/O
    }
};

class TimerManager : public Runnable {
public:
    void doWork() override {
        // Check timers
    }
};
```

TASK MANAGER

```
class TaskManager : public Runnable {
public:
    void doWork() override {
        if (auto task = queue_.pop()) {
            std::invoke(*task);
        }
    }

private:
    Queue queue_;
};
```

WHAT QUEUE:

- Locked queue
- SPSC with locked producer
- MPSC

LOCKED QUEUE: ONE SHARED LOCK

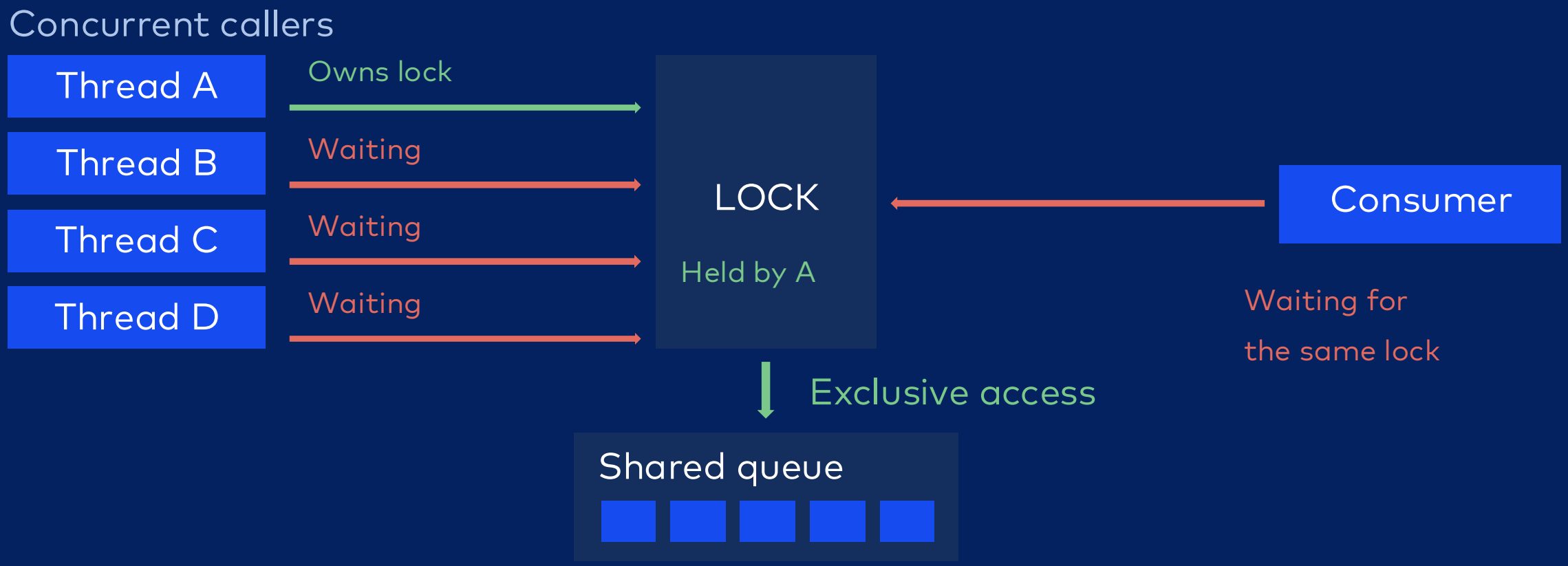
Producers

```
void push(Item item) {
    std::lock_guard guard(lock_);
    if (writePtr_ - readPtr_ >= N)
        return false;
    queue_[writePtr_++ % N] = std::move(item);
    return true;
}
```

Consumer

```
std::optional<Item> pop() {
    std::lock_guard guard(lock_);
    if (readPtr_ == writePtr_)
        return std::nullopt;
    auto item = std::move(queue_[readPtr_++ % N]);
    return item
}
```

LOCKED QUEUE: THREADS CONTEND



SPSC WITH A LOCKED PRODUCER

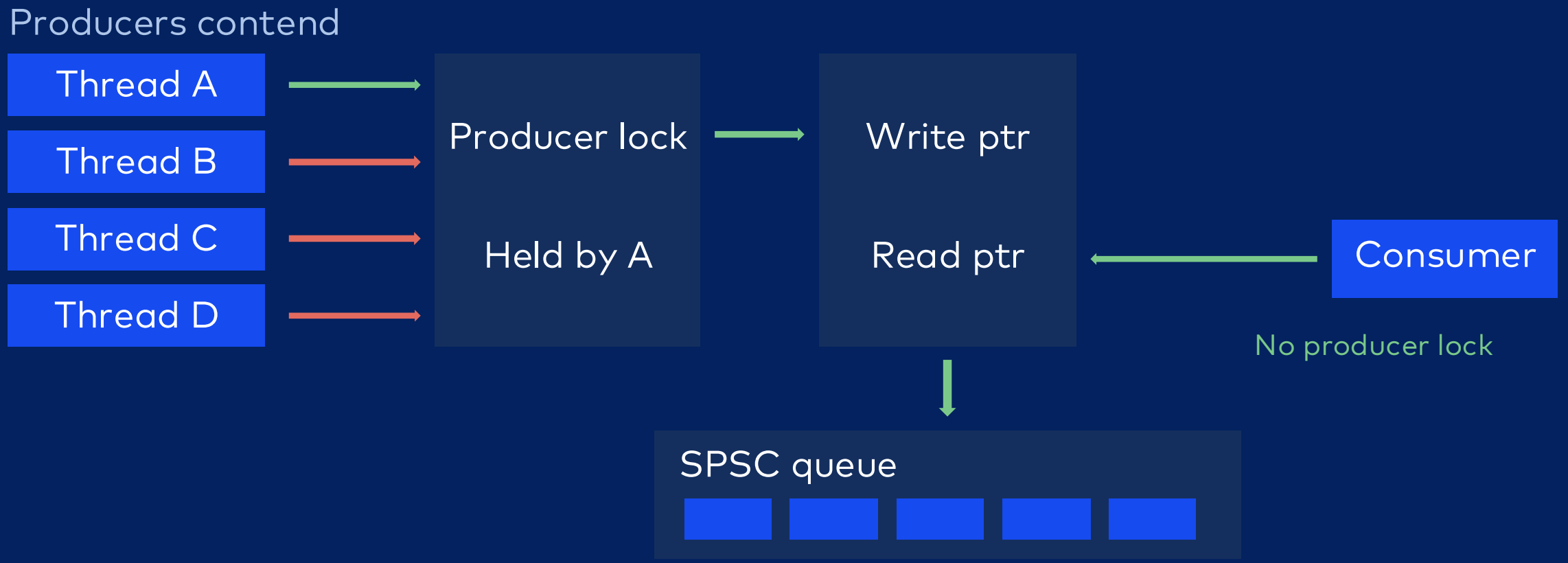
Producers: serialized

```
bool push(Item item) {
    std::lock_guard guard(lock_);
    auto slot = writePtr_.load();
    if (slot - readPtr_.load() >= N)
        return false;
    queue_[slot % N] = std::move(item);
    writePtr_.store(slot + 1);
    return true;
}
```

Single consumer: no producer lock

```
std::optional<Item> pop() {
    auto slot = readPtr_.load();
    if (slot == writePtr_.load())
        return std::nullopt;

    auto item = std::move(queue_[slot % N]);
    readPtr_.store(slot + 1);
    return item;
}
```

SPSC WITH A LOCKED PRODUCER



MPSC: RESERVE, WRITE, PUBLISH

Producers

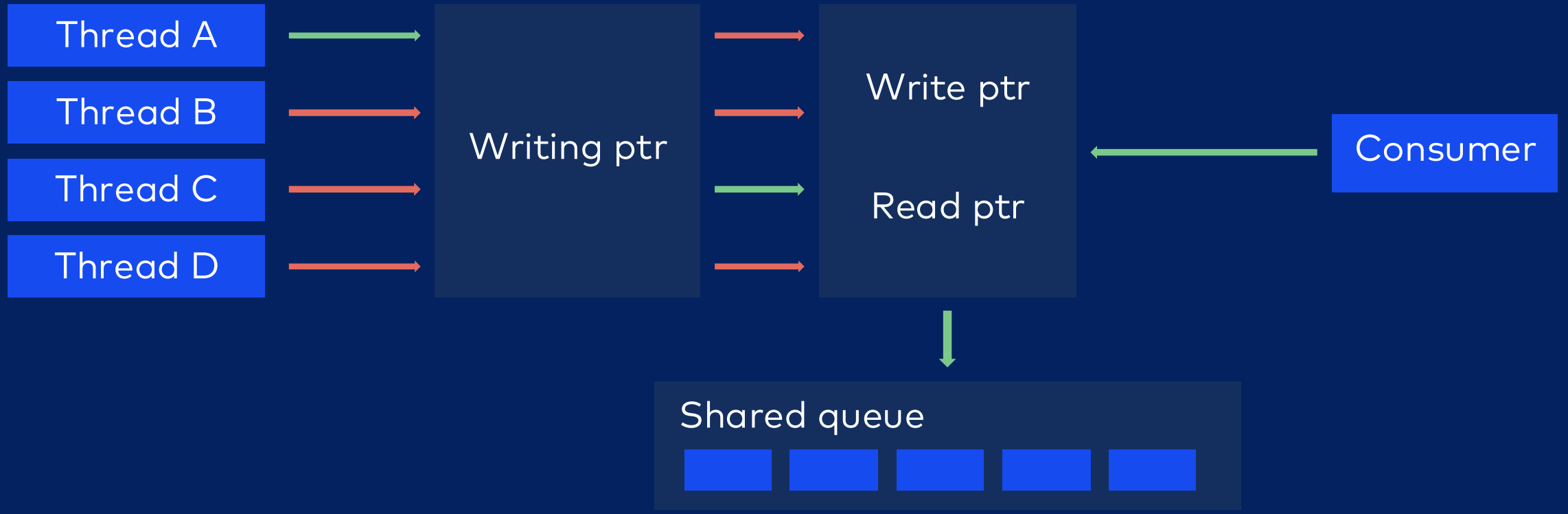
```
bool push(Item item) {
    const auto read = readPtr_.load();
    auto slot = writingPtr_.load();
    do {
        if (slot - read >= N) return false;
    } while (!writingPtr_.compare_exchange_weak(
        slot, slot + 1));
    queue_[slot % N] = std::move(item);
    while (writePtr_.load() != slot) ;
    writePtr_.store(slot + 1);
    return true;
}
```

Consumer

```
std::optional<Item> pop() {
    if (localRead_ == write_ptr_.load())
        return std::nullopt;
    auto item = std::move(queue_[localRead_ % N]);
    readPtr_.store(++localRead_);
    return item;
}
```

MPSC QUEUE

Multiple producers



SPSC

Producer: choose the destination

```
bool push(Item item) {
    if (localWrite_ - readPtr_.load() >= N)
        return false;
    queue_[localWrite_ % N] = std::move(item);
    writePtr_.store(++localWrite_);
    return true;
}
```

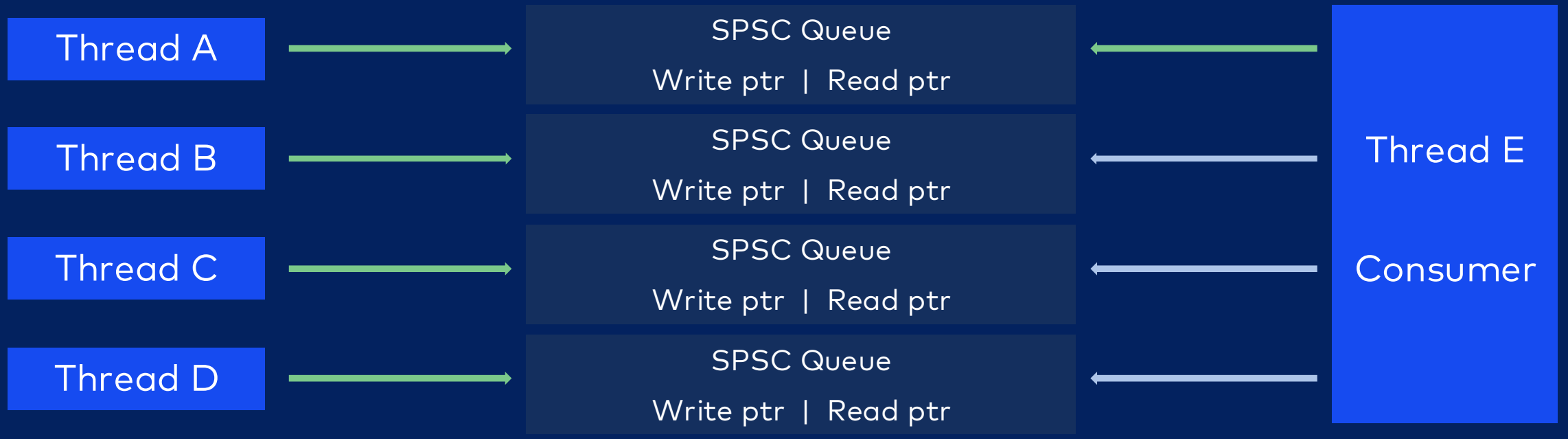
Consumer: poll incoming queues

```
std::optional<Item> pop() {
    if (localRead_ == writePtr_.load())
        return std::nullopt;
    auto item = std::move(queue_[localRead_ % N]);
    read_ptr_.store(++localRead_);
    return item;
}
```

SPSC

Independent producers

Each pair owns its pointers

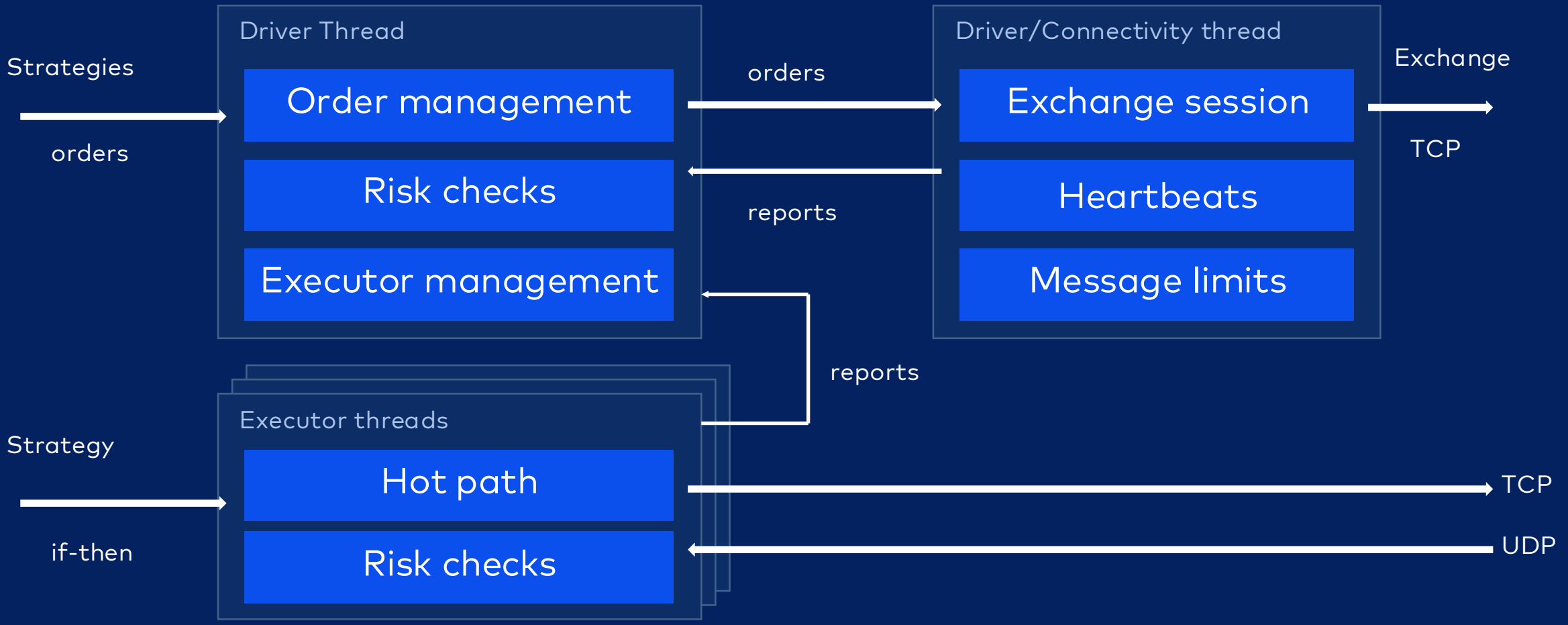


Round-robin polling

GUIDELINE NUMBER 2

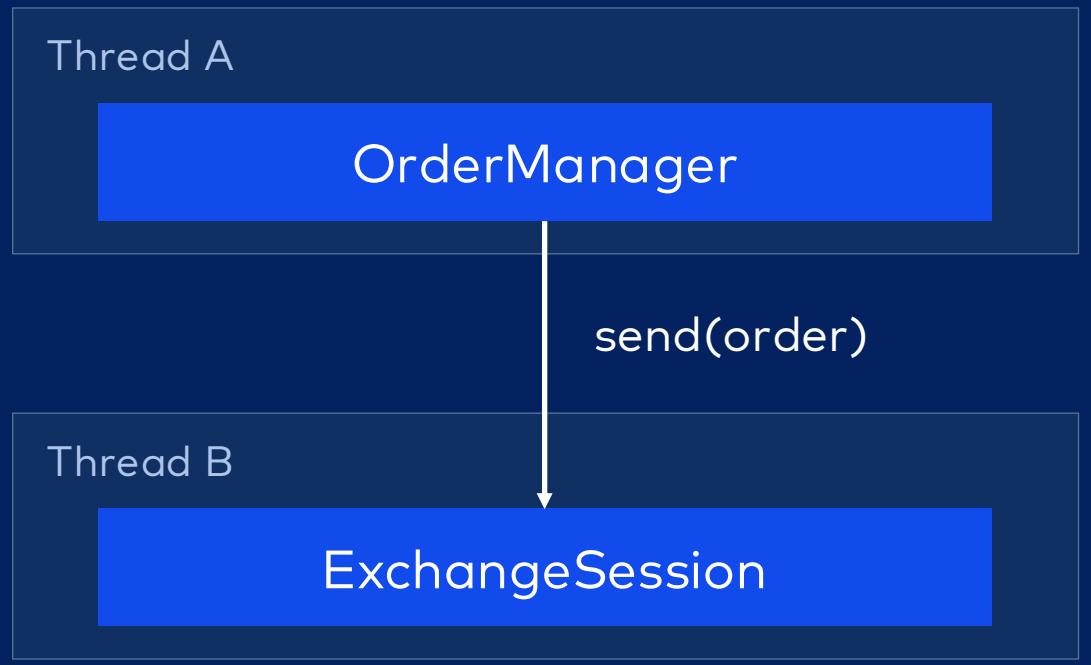
Only a single thread should write to shared data

INSIDE THE DRIVER

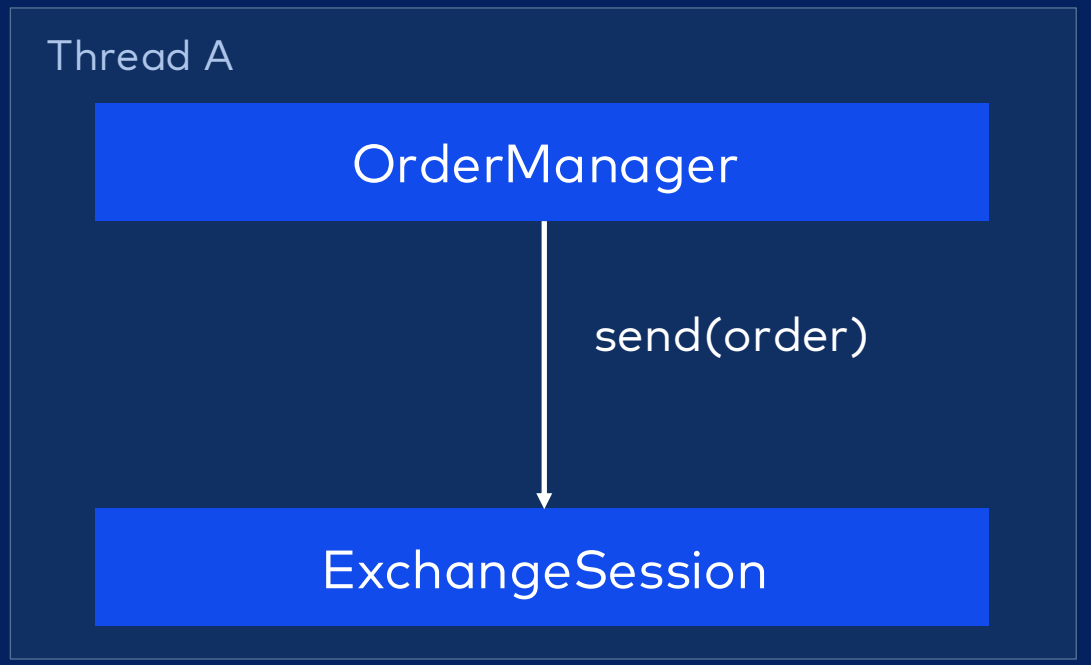


SAME COMPONENTS, DIFFERENT THREADS

Two different threads



One shared thread



RUN HERE, OR QUEUE IT THERE

```
class OrderManager {
    void onOrder(Order order) {
        inflightOrders_.insert(order);
        getDispatcher().runOn(
            exchangeSession_.thread(),
            [this, order] {
                exchangeSession_.send(order);
            });
    }

    ExchangeSession& exchangeSession_;
};
```

```
class Dispatcher {
    void runOn(ThreadId tid, auto func) {
        if (tid == localTid_) {
            std::invoke(func);
        }
        else {
            _queue[tid].enqueue(func);
        }
    }
};
```

USE AFTER OOPS

```
class OrderManager {
    void onOrder(Order order) {
        inflightOrders_.insert(order);
        getDispatcher().runOn(
            exchangeSession_.thread(),
            [this, &] {
                exchangeSession_.send(order);
            });
    }

    ExchangeSession& exchangeSession_;
};
```

```
class Dispatcher {
    void runOn(ThreadId tid, auto func) {
        if (tid == localTid_) {
            std::invoke(func);
        }
        else {
            _queue[tid].enqueue(func);
        }
    }
};
```

RUN HERE, OR QUEUE IT THERE

```
class OrderManager {
    void onOrder(Order order) {
        inflightOrders_.insert(order);
        exchangeSession_.send(order);
    }

    ExchangeSession& exchangeSession_;
};
```

```
class ExchangeSession {
    void send(Order order) {
        getDispatcher().run(
            [this, order] { sendImpl(order); });
    }
};
```

RUN HERE, OR QUEUE IT THERE

```
class ExchangeSession {
public:
    bool isConnected() const {
        return connected_.;
    }

    auto getSocket() const {
        return socket_;
    }
private:
    std::shared_ptr<TcpSocket> socket_;
    bool connected_;
};
```

```
class ExchangeSession {
public:
    void send(Order order) {
        getDispatcher().run(
            [this, order] { sendImpl(order); });
    }

    // If same thread
    void sendImpl(Order order);
};
```

RUN HERE, OR QUEUE IT THERE

```
class ExchangeSession {
public:
    bool isConnected() const {
        return connected_.load();
    }

    auto getSocket() const {
        return socket_;
    }
private:
    std::shared_ptr<TcpSocket> socket_;
    std::atomic_bool connected_;
};
```

```
class ExchangeSession {
public:
    void send(Order order) {
        getDispatcher().run(
            [this, order] { sendImpl(order); });
    }

    // If same thread
    void sendImpl(Order order);
};
```

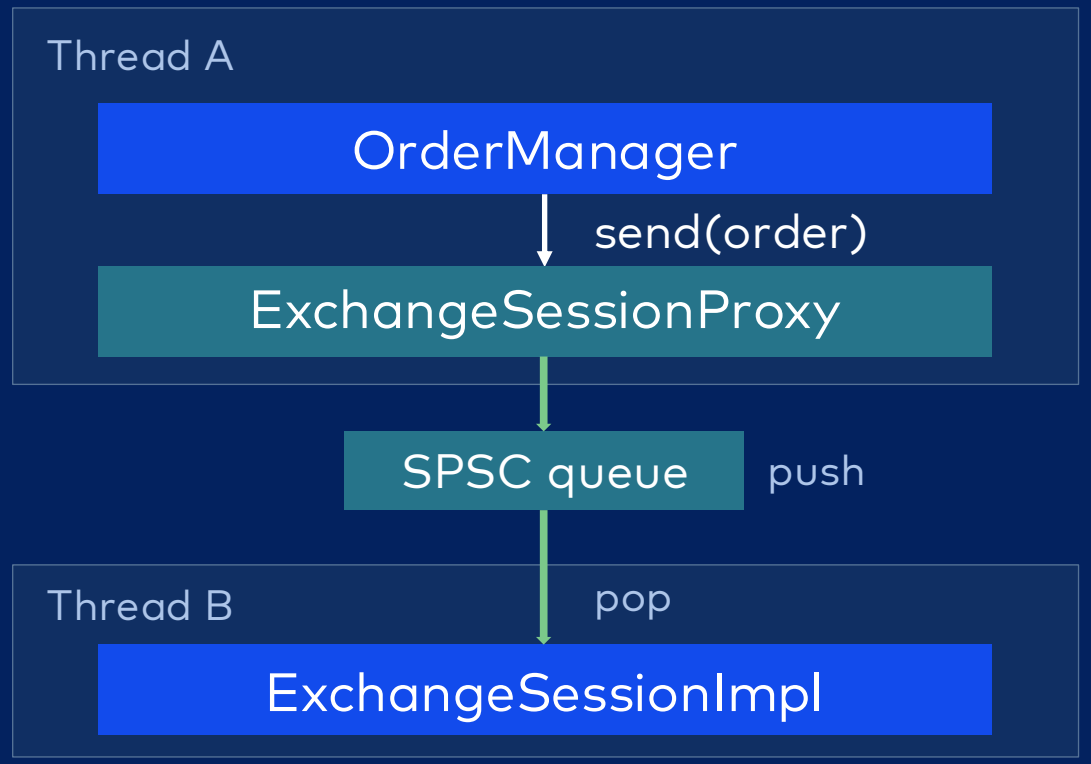
RUN HERE, OR QUEUE IT THERE

```
class OrderManager {
    void onOrder(Order order) {
        inflightOrders_.insert(order);
        exchangeSession_.send(order);
    }

    ExchangeSession& exchangeSession_;
};
```

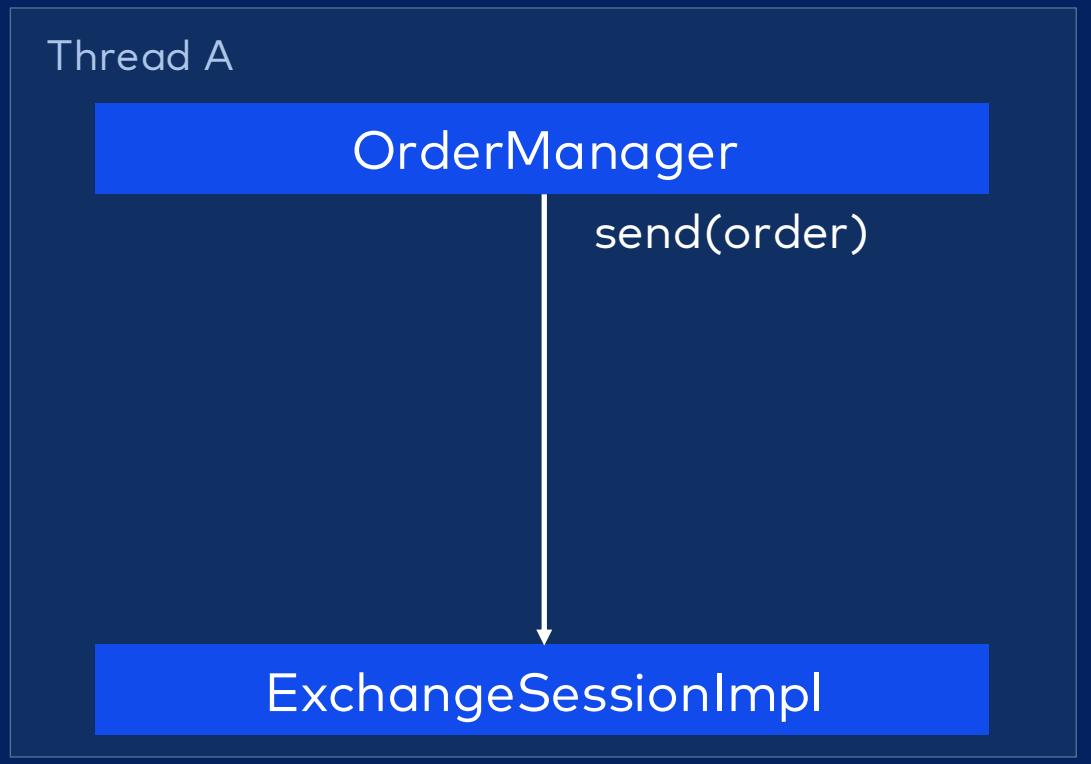
```
class ExchangeSessionProxy
    : public ExchangeSession {
    void send(Order order) {
        dispatcher_.run<&ExchangeSession::send>(
            order);
    }
};

class ExchangeSessionImpl
    : public ExchangeSession {
    void send(Order order) {
        // …
    }
};
```

SAME COMPONENTS, DIFFERENT THREADS

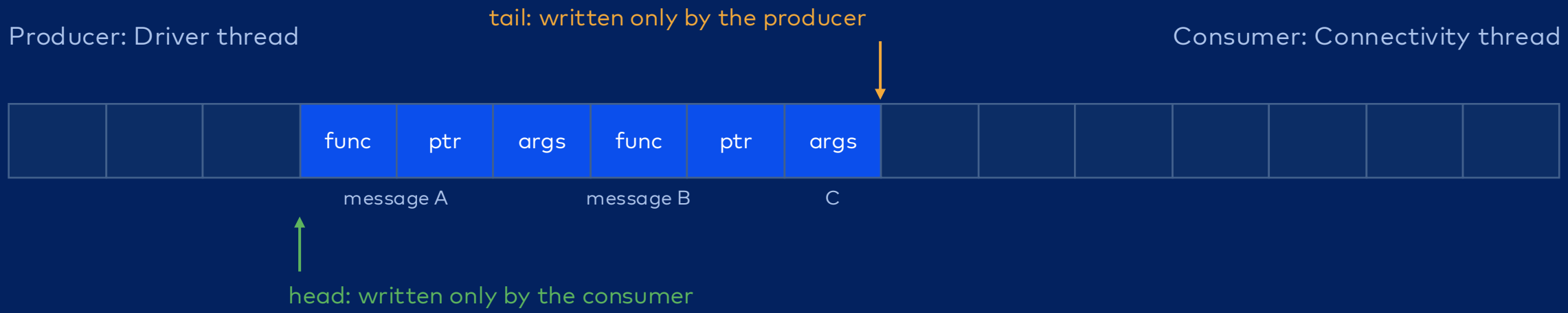
Two different threads



One virtual call



THE QUEUE: AN SPSC RING OF BYTES




```
auto remoteFunc = +[](SpscSliceReader slice) {
    auto target = decodeArg<TargetT*>(slice);
    auto&& [...params] = decodeArgs<decltype(args)...>(slice);
    std::invoke(Method, target, std::forward<decltype(params)>(params)...);  // Runs on target thread
});
```

DISPATCHER::RUN

```
template<auto Method>
bool Dispatcher<TargetT>::run(auto&&... args) {
    auto remoteFunc = +[](SpscSliceReader slice) {
        auto target = decodeArg<TargetT*>(slice);
        auto&& [...params] = decodeArgs<decltype(args)...>(slice);
        std::apply(Method, target, std::forward<decltype(params)>(params)...);  // Runs on target thread
    });
    // Get ring buffer slice, maybe load shared readPtr_
    const SpscSliceWriter slice = queueProducer_.getWritableSlice();

    encodeArgs(slice, remoteFunc, target_, args...);
    queueProducer_.commit(slice.writePtr()); // Commit shared writePtr_
}
```

QUEUE AND QUEUEPRODUCER

```
class QueueProducer {
public:
    SpscSliceWriter getWritableSlice() {
        if (localWrite_ - localRead_ < MaxMsgSize)
            return {queue_.data(), localWrite_};

        localRead_ = queue_.readPtr_.load(
            std::memory_order_acquire);
        if (localWrite_ - localRead_ < MaxMsgSize)
            return {queue_.data(), localWrite_};
        return {}; // Empty, back pressure
    }

    std::uint64_t localWrite_, localRead_;
    SpscRing& queue_;
};
```

```
struct SpscRing {
    static constexpr auto CacheLine =
        std::hardware_destructive_interference_size;

    alignas(CacheLine)
    std::atomic_uint64_t writePtr_;
    alignas(CacheLine)
    std::atomic_uint64_t readPtr_;

    std::array<std::byte, N> buffer_;
};
```

QUEUE AND QUEUEPRODUCER

```
class QueueProducer {
public:
    SpscSliceWriter getWritableSlice() {
        if (localWrite_ - localRead_ < MaxMsgSize)
            return {queue_.data(), localWrite_};

        localRead_ = queue_.readPtr_.load(
            std::memory_order_acquire);
        if (localWrite_ - localRead_ < MaxMsgSize)
            return {queue_.data(), localWrite_};
        return {}; // Empty, back pressure
    }

    std::uint64_t localWrite_, localRead_;
    SpscRing& queue_;
};
```

```
class QueueProducer {
public:
    void commit(std::uint64_t writePtr) {
        localWrite_ = writePtr;

        queue_.writePtr_.store(localWrite_,
            std::memory_order_release);









    }
};
```

QUEUE AND QUEUEPRODUCER

```
class QueueProducer {
public:
    SpscSliceWriter getWritableSlice() {
        if (localWrite_ - localRead_ < MaxMsgSize)
            return {queue_.data(), localWrite_};

        localRead_ = queue_.readPtr_.load(
            std::memory_order_acquire);
        if (localWrite_ - localRead_ < MaxMsgSize)
            return {queue_.data(), localWrite_};
        return {}; // Empty, back pressure
    }

    std::uint64_t localWrite_, localRead_;
    SpscRing& queue_;
};
```

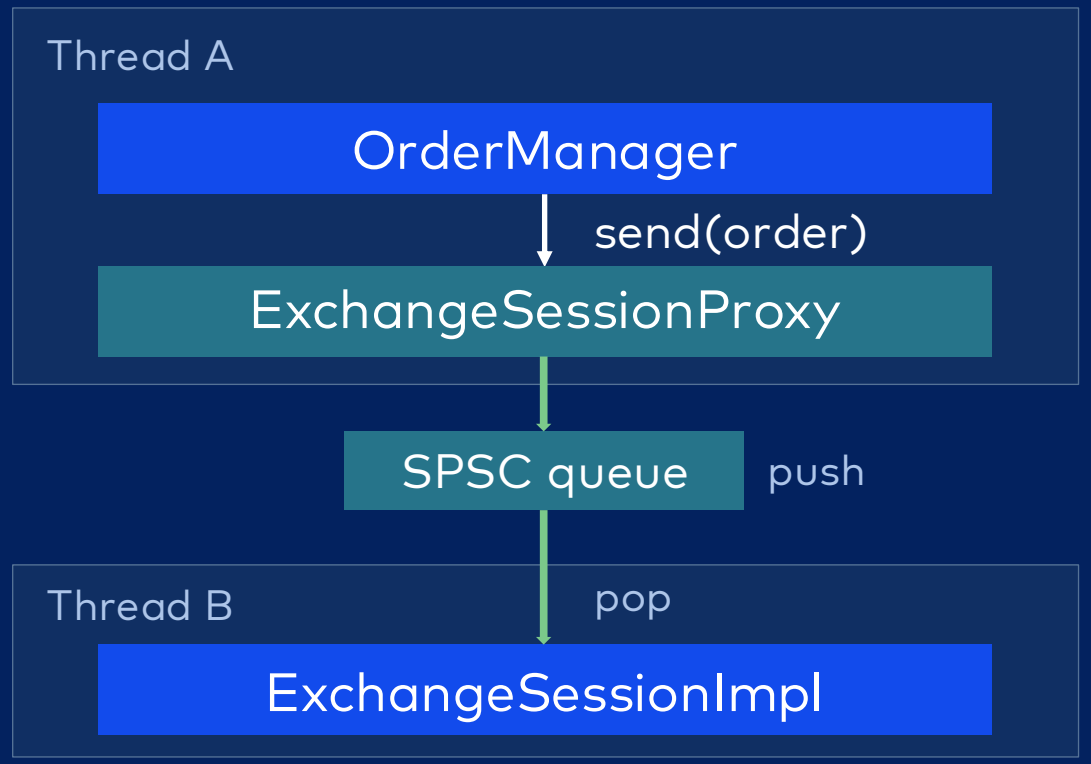
```
class QueueProducer {
public:
    void commit(std::uint64_t writePtr) {
        localWrite_ = writePtr;

        queue_.writePtr_.store(localWrite_,
            std::memory_order_release);

        // Prefetch next slice
        __builtin_prefetch(
            &queue_.buffer_[localWrite_ % N],
            /*rw=Write*/ 1,
            /*locality=Keep*/ 3);
    }
};
```

TASK MANAGER

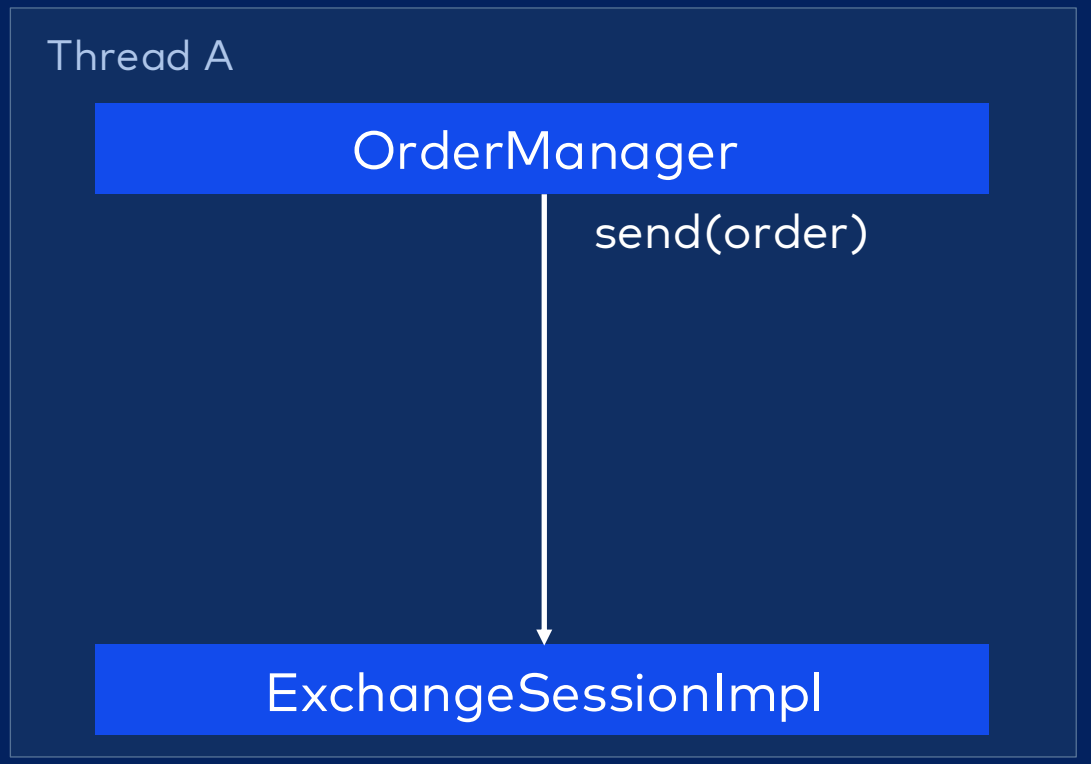
```
class TaskManager : public Runnable {
public:
    void doWork() override {
        using RemoteFunc = void (*)(SpscSliceReader&);
        for (auto& consumer : consumers_) {
            auto slice = consumer.getReadableSlice();
            while (slice.hasBytes()) {
                auto remoteFunc = decodeArg<RemoteFunc>(slice);
                std::invoke(remoteFunc, slice);
            }
            consumer.commit(slice.readPtr());
        }
    }
private:
    std::vector<QueueConsumer> consumers_;
};
```

SAME COMPONENTS, DIFFERENT THREADS

Two different threads



One virtual call



CHALLENGES

- Return values
- Getting state

RETURN TO SENDER

```
class OrderManager {
    void onOrder(Order order) {
        inflightOrders_.insert(order);

        auto report =
            exchangeSession_.send(order);
    }

    ExchangeSession& exchangeSession_;
};
```

```
class ExchangeSession {
public:
    virtual Report send(Order order) = 0;
};
```

RETURN TO SENDER

```
class OrderManager {
    void onOrder(Order order) {
        inflightOrders_.insert(order);

        auto report =
            exchangeSession_.send(order);
    }

    ExchangeSession& exchangeSession_;
};
```

```
class ExchangeSession {
public:
    virtual Continuation<Report> send(
        Order order) = 0;
};
```

RETURN TO SENDER

```
class OrderManager {
    void onOrder(Order order) {
        inflightOrders_.insert(order);

        exchangeSession_.send(order)
            .then([this](const Report& report) {
                onReport(report);
            });
    }

    ExchangeSession& exchangeSession_;
};
```

RETURN TO SENDER

```
template<class T>
class Continuation {
public:
    Continuation(T value);
    Continuation(Promise<T>* promise);
    Continuation(Continuation&& other);

    template<VoidCallback F>
    void then(F&& callback) const;

    template<ContinuationCallback F>
    ResultOf<F> then(F&& callback) const;

private:
    std::variant<T, Promise<T>*> state_;
};
```

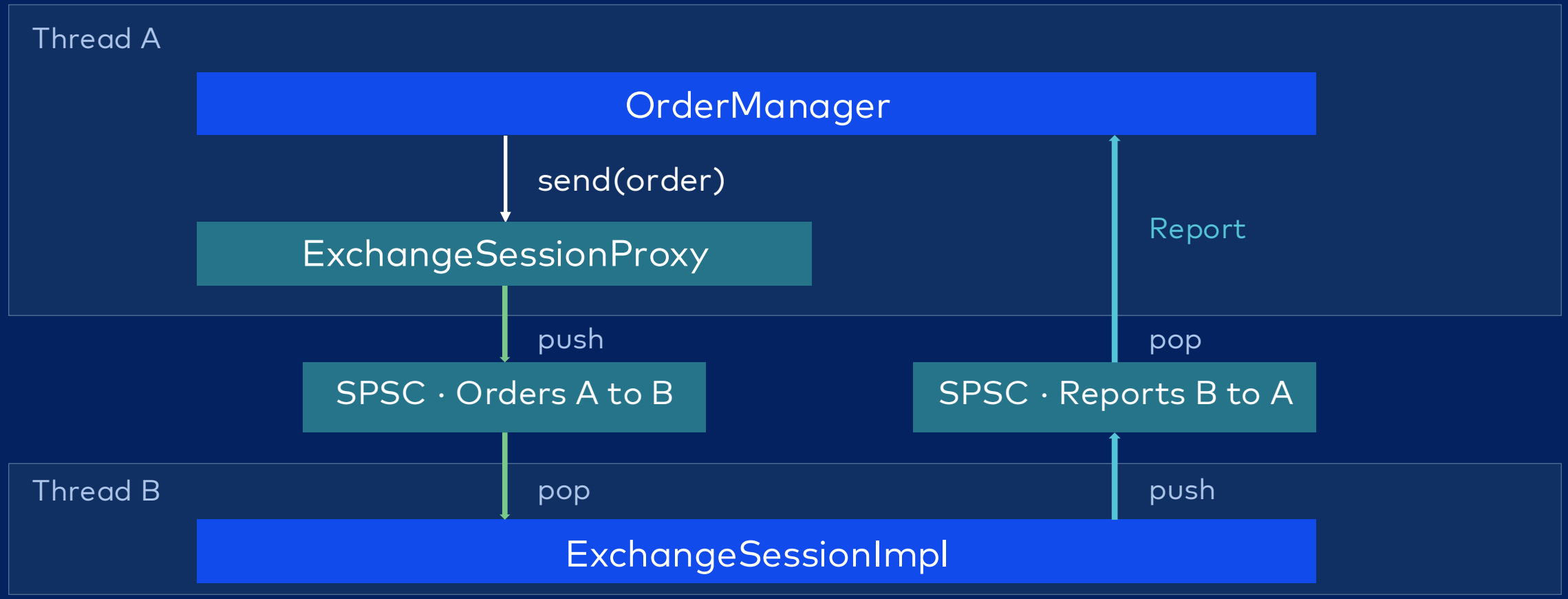
```
template<class T>
template<VoidCallback F>
void Continuation<T>::then(F&& callback) const {
    std::visit(overload{
        [&](const T& value) {
            std::invoke(
                std::forward<F>(callback),
                value);
        },
        [&](Promise<T>* promise) {
            promise->addCallback(
                std::forward<F>(callback));
        }
    }, state_);
};
```

RETURN TO SENDER

```
class ExchangeSessionProxy
    : public ExchangeSession {
    Continuation<Report> send(Order order) {
        return dispatcher_
            .run<&ExchangeSession::send>(order);
    }
};
```

```
class ExchangeSessionImpl
    : public ExchangeSession {
    Continuation<Report> send(Order order) {
        return Report {
            .success = true,
            .requestId = ...
        };
    }
};
```

REPORTS RETURN ON A SEPARATE SPSC



DISPATCHER::RUN — SIMPLIFIED

```
template<VoidReturning Method>
void Dispatcher<TargetT>::run(auto&&... args) {
    auto remoteFunc = +[](QueueProducer&, ContinuationRegistry&, SpscSliceReader slice) {
        auto target = decodeArg<TargetT*>(slice);
        auto&& [...params] = decodeArgs<decltype(args)...>(slice);
        std::invoke(Method, target, std::forward<decltype(params)>(params)...);  // Runs on target thread
    };
    // Get ring buffer slice, maybe load shared readPtr_
    const SpscSliceWriter slice = queueProducer_.getWritableSlice();

    encodeArgs(slice, remoteFunc, target_, args...);
    queueProducer_.commit(slice.writePtr()); // Commit shared writePtr_
}
```

DISPATCHER::RUN — SIMPLIFIED

```
template<ContinuationReturning Method>
auto Dispatcher<TargetT>::run(auto&&... args) {
    auto remoteFunc = +[](QueueProducer& returnProducer, ContinuationRegistry& ,SpscSliceReader slice) {
        auto target = decodeArg<TargetT*>(slice);
        auto&& [...params] = decodeArgs<decltype(args)...>(slice);
        std::invoke(Method, target, std::forward<decltype(params)>(params)...);  // Runs on target thread
    };
    // Get ring buffer slice, maybe load shared readPtr_
    const SpscSliceWriter slice = queueProducer_.getWritableSlice();

    encodeArgs(slice, remoteFunc, target_, args...);
    queueProducer_.commit(slice.writePtr()); // Commit shared writePtr_
}
```

DISPATCHER::RUN — SIMPLIFIED

```
template<ContinuationReturning Method>
auto Dispatcher<TargetT>::run(auto&&... args) {
    auto [id, promise] = continuationRegistry_.registerContinuation();
    auto remoteFunc = +[](QueueProducer& returnProducer, ContinuationRegistry&, SpscSliceReader slice) {
        auto target = decodeArg<TargetT*>(slice);
        auto continuationId = decodeArg<ContinuationId>(slice);
        auto&& [...params] = decodeArgs<decltype(args)...>(slice);
        std::invoke(Method, target, std::forward<decltype(params)>(params)...);
    };
    // Get ring buffer slice, maybe load shared readPtr_
    const SpscSliceWriter slice = queueProducer_.getWritableSlice();

    encodeArgs(slice, remoteFunc, target_, id, args...);
    queueProducer_.commit(slice.writePtr()); // Commit shared writePtr_
    return std::invoke_result_t<decltype(Method)>{promise};
}
```

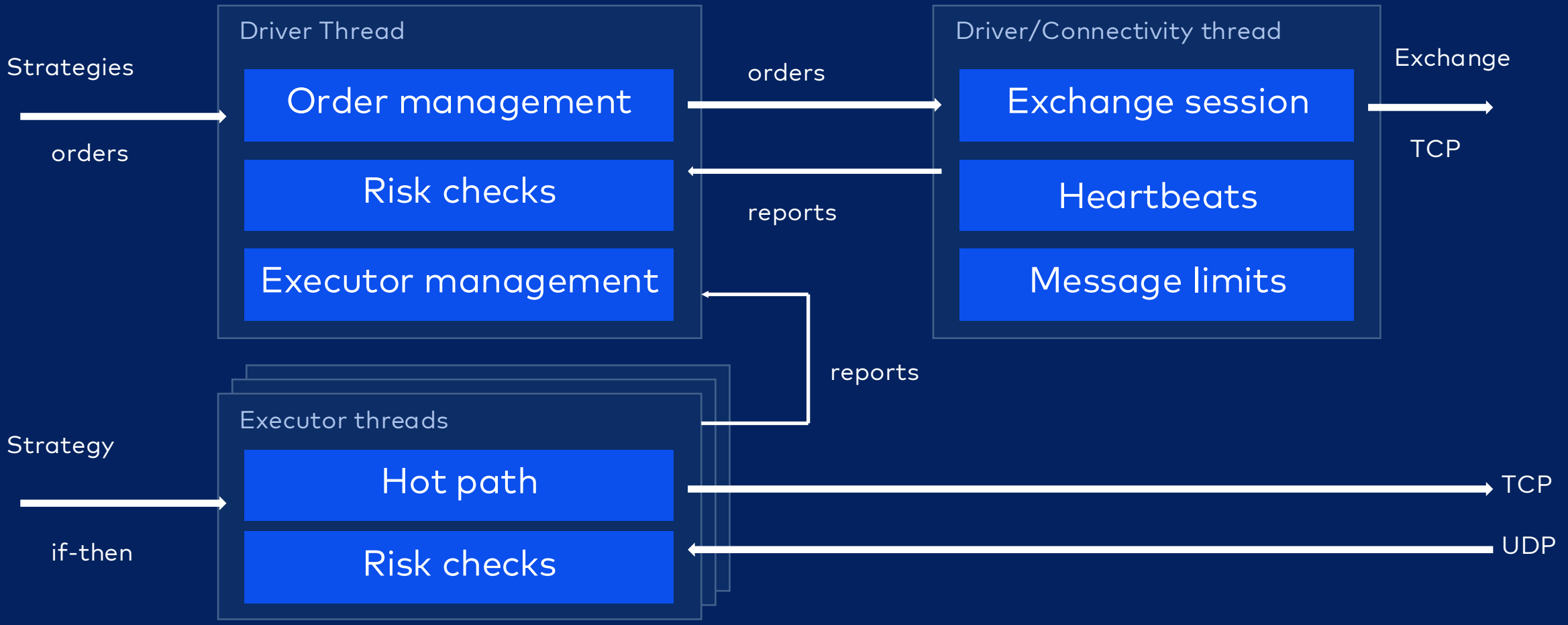
DISPATCHER::RUN — SIMPLIFIED

```
auto remoteFunc = +[](QueueProducer& returnProducer, ContinuationRegistry&, SpscSliceReader slice) {
    // In ExchangeSession thread!
    // Decode target, continuationId, params
    auto result = std::invoke(Method, target, std::forward<decltype(params)>(params)...);

    auto returnFunc = +[](QueueProducer&, ContinuationRegistry& registry, SpscSliceReader slice) {
        // Runs back in the caller's thread!
        auto continuationId = decodeArg<ContinuationId>(slice);
        auto continuationResult = decodeArg<ContinuationResultOf<decltype(Method)>>(slice);
        registry.completeContinuation(continuationId, std::move(continuationResult));
    };
    const SpscSliceWriter slice = queueProducer_.getWritableSlice();
    encodeArgs(slice, returnFunc_, id, std::move(result));
    returnProducer.commit(slice.writePtr()); // Commit shared writePtr_
};
```

HOME STRETCH

- TaskManager polling all SPSCs
- Dispatcher sends the func and args
- Continuations mean you can get results back
- Final hurdle: How do we set this all up?

INSIDE THE DRIVER



THREADING FRAMEWORK

```
// Application side: roles, and where they collapse to.
enum class Role : uint8_t { Background = 1, Driver = 2,
                            Connectivity = 4, Executor = 8 };

ThreadMap<Role> map{cfg.affinity};                    // "4-7,9,11"
map.pin(Role::Background, cfg.bgThread.value_or(CpuIndex{0}));          // core 0: never pinned
map.pin(Role::Driver, cfg.driverThread);                                // core 1
map.pinOptional(Role::Connectivity, cfg.connectivityThread,             // core 2
             Role::Driver);                          // fallback, then Driver
map.pinRange(Role::Executor,     cfg.executorThreads);                   // core 3 - 7


// Framework side: one entry per thread, roles are a bitmask.
struct ThreadEntry { CpuIndex cpu_; Role roles_; ComponentRegistry localComponents_; };
```

THREADING FRAMEWORK

```
template <typename Interface, std::derived_from<Interface> Proxy>
struct PublishAs {};

template <typename Role>
class Builder {
public:
    template <ComponentFactory F, typename I = void, typename P = void>
    Builder& with(Role role, F&& factory, PublishAs<I, P> = {}) {
        threadFor(role).addService(+factory, PublishAs<I, P>{});
        return *this;
    }
};

// Usage
builder.with(Role::Connectivity,
    [](ThreadContext& context) {
        return std::make_unique<ExchangeSessionImpl>(context.getNetworkManager());
    },
    PublishAs<ExchangeSession, ExchangeSessionProxy>{});
```

THREADING FRAMEWORK

```
builder.with(Role::Driver,
    [](ThreadContext& context) {
        return std::make_unique<OrderManagerImpl>(
           context.getTimerManager(),
           context.get<ExchangeSession>(Communication::SyncIfPossible));
    },
    PublishAs<OrderManager, OrderManagerProxy>{});
```

```
// Usage
builder.with(Role::Connectivity,
    [](ThreadContext& context) {
        return std::make_unique<ExchangeSessionImpl>(context.getNetworkManager());
    },
    PublishAs<ExchangeSession, ExchangeSessionProxy>{});
```

THREAD CONTEXT

```
enum class Communication { Async, Sync, SyncIfPossible };

class ThreadContext {
    template<typename ComponentT>
    ComponentT& get(Communication how = Communication::Sync);

    TimerFactory&  getTimerManager();
    NetworkManager&  getNetworkManager();
};
```

Mode	Meaning	If the target lives on another thread
Sync	I need the real object, now, on this thread.	Throw at startup
Async	Always give me a proxy, even if the target is local.	Proxy
SyncIfPossible	Direct call if we share a thread, proxy otherwise.	Proxy

THREADING FRAMEWORK

```
ComponentT& ThreadContext::get<ComponentT>(Communication how) {
    Local& local = localComponents_.find<ComponentT>();        // this thread's view
    bool wantProxy = how == Communication::Async
        || (how == Communication::SyncIfPossible && !local.hasImpl());

    if (wantProxy) {
        if (local.hasProxy()) return local.proxy();    // reuse
        auto proxy = globalComponents_.proxyFor<ComponentT>();
        local.setProxy(proxy);
        return ref;
    }
    if (local.hasImpl()) return local.impl();      // same thread: direct
    throw NotAvailable {util::typename_of<ComponentT>(), threadName_};
}
```

GUIDELINE NUMBER 3

Crash during startup, not during runtime

THREADING FRAMEWORK

```
void MyApplicationConfigurator::configure(ApplicationBuilder& builder) {
    builder.with(Role::Connectivity, [](ThreadContext& context) {
        return std::make_unique<ExchangeSessionImpl>(
            context.getTimerManager(),
            context.getNetworkManager(),
            context.getConfig<SessionConfig>());
    }, PublishAs<ExchangeSession, ExchangeSessionProxy>{})
    .with(Role::Driver, [](ThreadContext& context) {
        return std::make_unique<OrderManagerImpl>(
            context.getTimerManager(),
            context.get<RiskCalculator>(Communication::SyncIfPossible),
            context.get<ExchangeSession>(Communication::SyncIfPossible),
            context.getConfig<DriverConfig>());
    }, PublishAs<OrderManager, OrderManagerProxy>{})
    .with(Role::Executor, [](ThreadContext& context) {
        return std::make_unique<ExchangeSpecificExecutor>(
            context.getNetworkManager(),
            context.get<OrderManager>(Communication::Async),
            context.getConfig<ExecutorConfig>());
    });
}
```

GOALS

- Fast
- Different scenarios: 1 - 32 cores
- Simple to use, hard to misuse

GUIDELINE NUMBER 3

1. Optimize before multi-threading

2. Only a single thread should write to shared data

3. Crash during startup, not during runtime

imc

THANKS FOR YOUR TIME