Streaming in CAF

Dominik Charousset

October 2018
C++ User Group Hamburg

Why care about alternative programming paradigms?

Multi-cores are Here to Stay

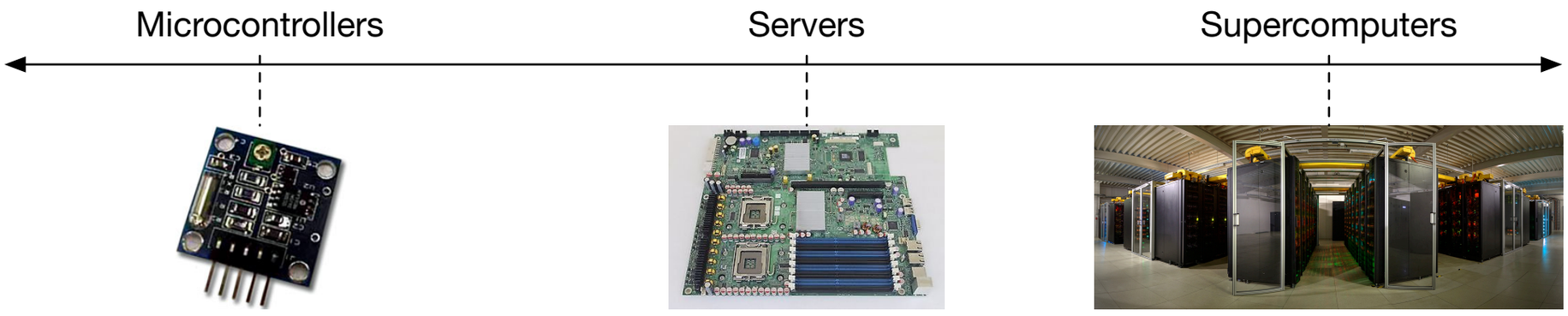
- The number of **CPU cores will only increase**
  - **Concurrency** cannot be an afterthought
  - **Software needs to adapt** to available hardware
- Programming concurrent systems **should be easy**
  - Low-level abstractions **error-prone and slow**
  - Common **idioms break** in concurrent settings

Beyond Concurrency

- **Microservices** challenge monolithic software design
  - **Modular and distributed** by design
    - **Orchestration** of loosely coupled services
- **Cloud deployments** demand flexibility
  - **Re-deployment** and automated vertical scaling
  - **Partial failures** of the system common

Why we need Actors

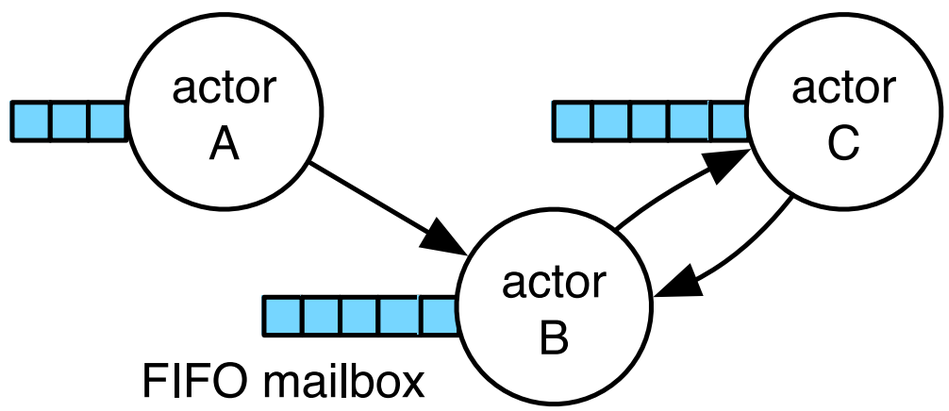
- **No race conditions** by design (without locks!)
- **Concurrency & distribution** at high abstraction level
- **Compose large systems** from small components
- **Scale without code changes** from IoT up to HPC



What is the actor model?

Actor Model in a Nutshell

- **Distributed by design**, perfect fit for microservices
- **Asynchronous** message passing
- **Shared nothing** architecture
- **Hierarchical** error handling
- **Divide & conquer** work flows



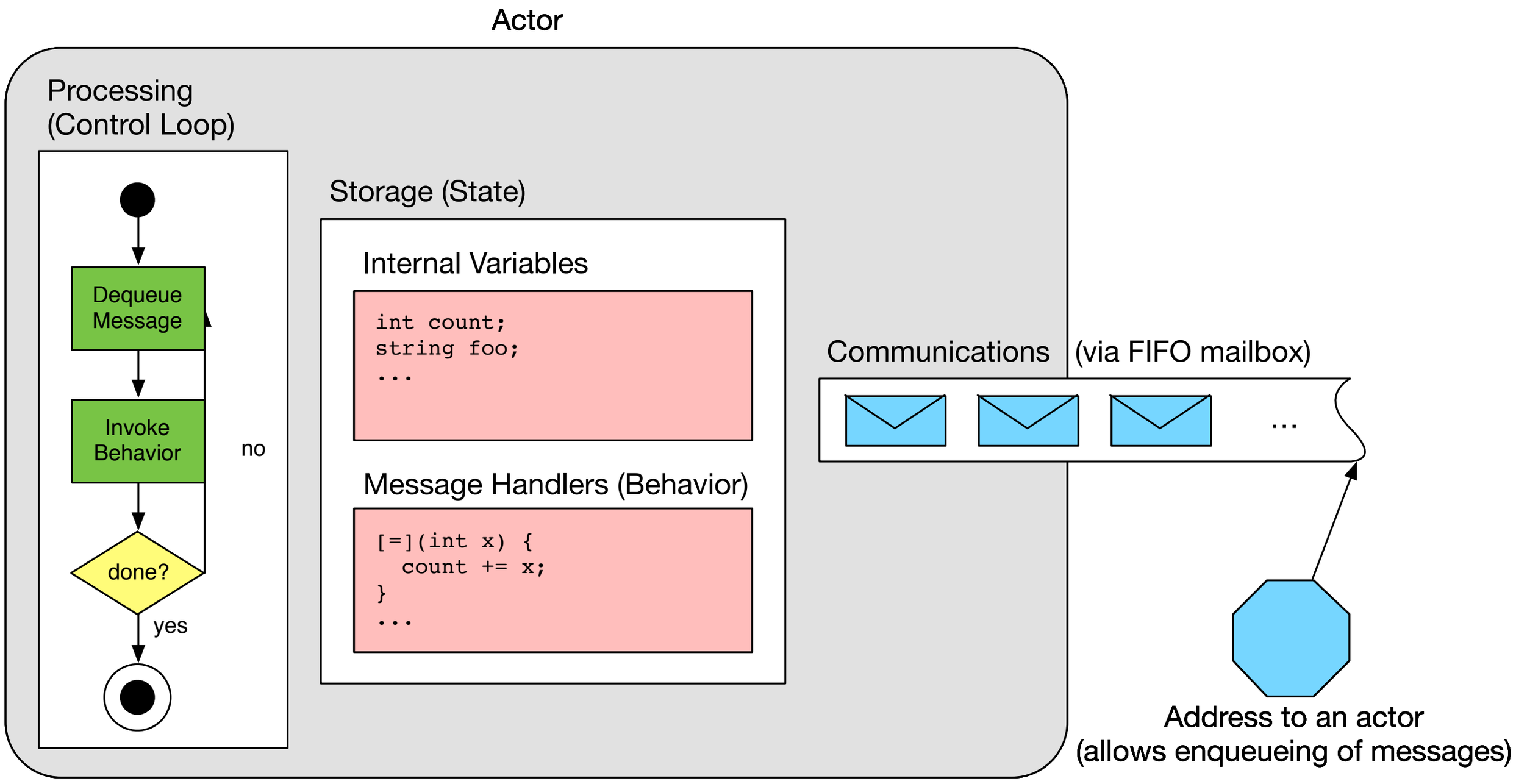
The World is a Stage

- "Actor model" refers to a **theater metaphor**
- **Each individual** acts according to a script
- Actors are agents with **intents and behaviors**
- An application is a **choreography of many**

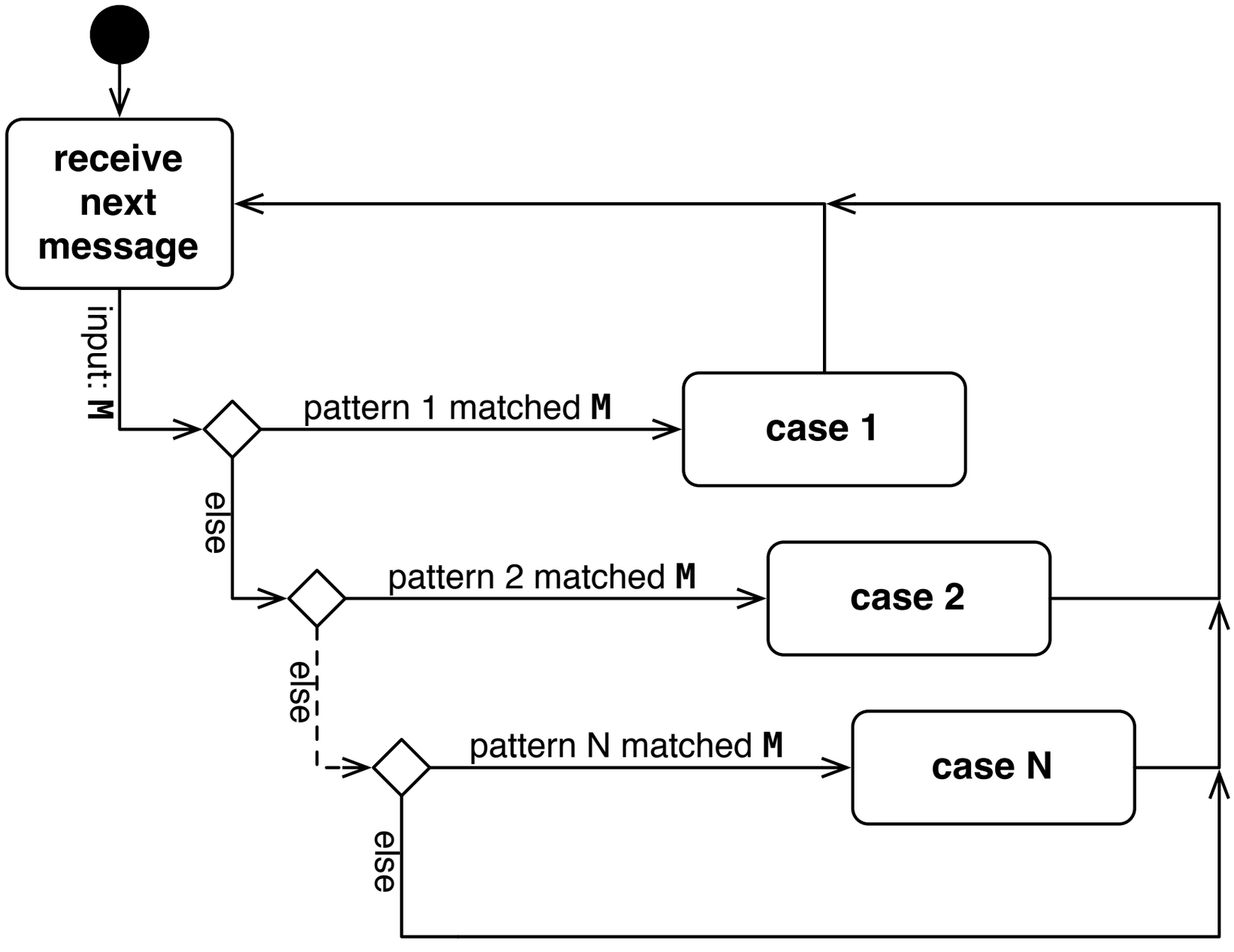
Programming Actors

- Actors operate **event-based** (message → event)
- In response to messages, **actors can**:
  - Send messages
  - Spawn more actors
  - Change their behavior
    (set of message handlers)

Anatomy of an Actor



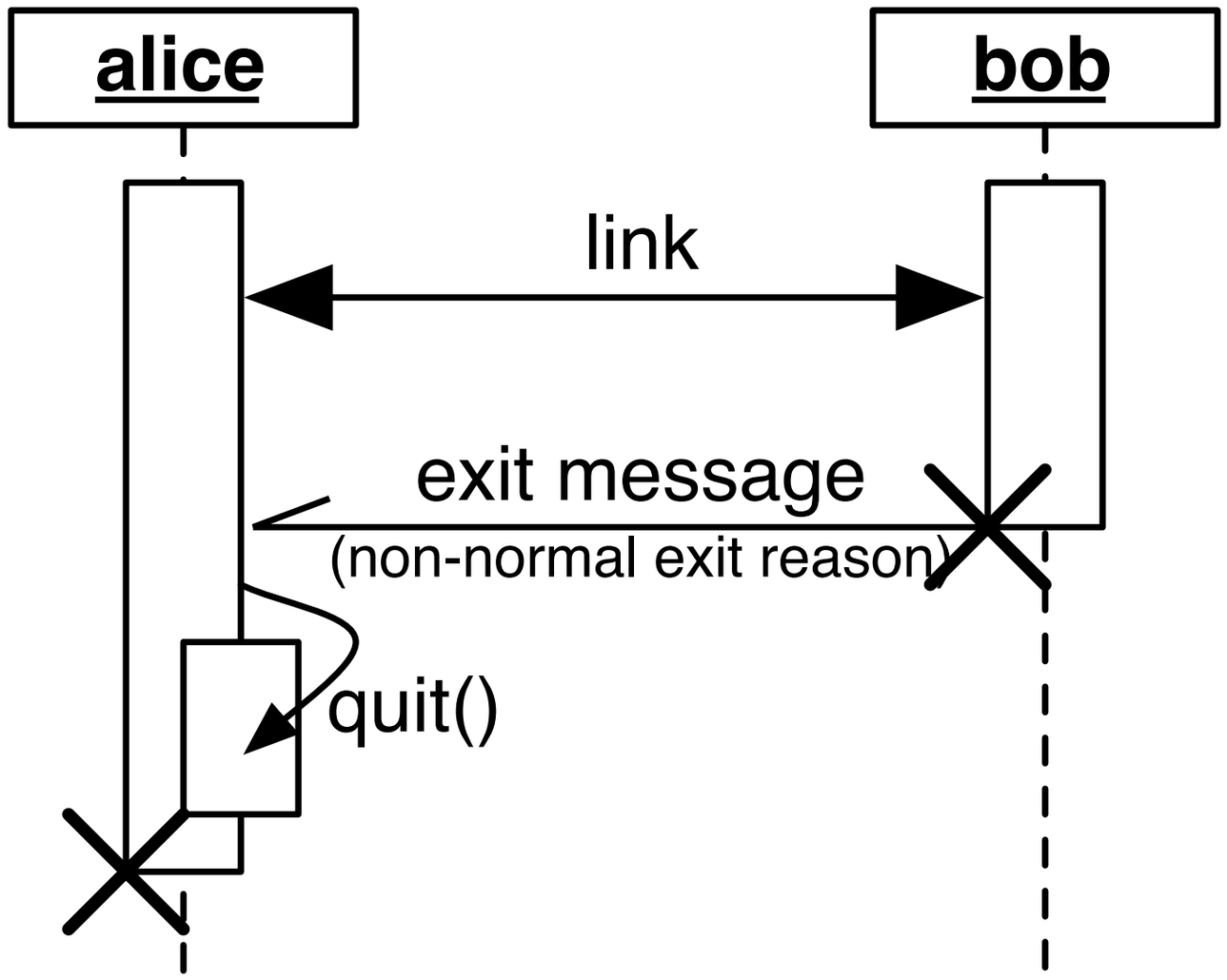
Control Flow of Actors



Error Handling

- Errors have **no side effects** between actors
- **Explicit handling of remote errors** via messages
- **Monitoring**: unidirectional observing of actors
- **Linking**: strong lifetime coupling of actors

Linking Actors

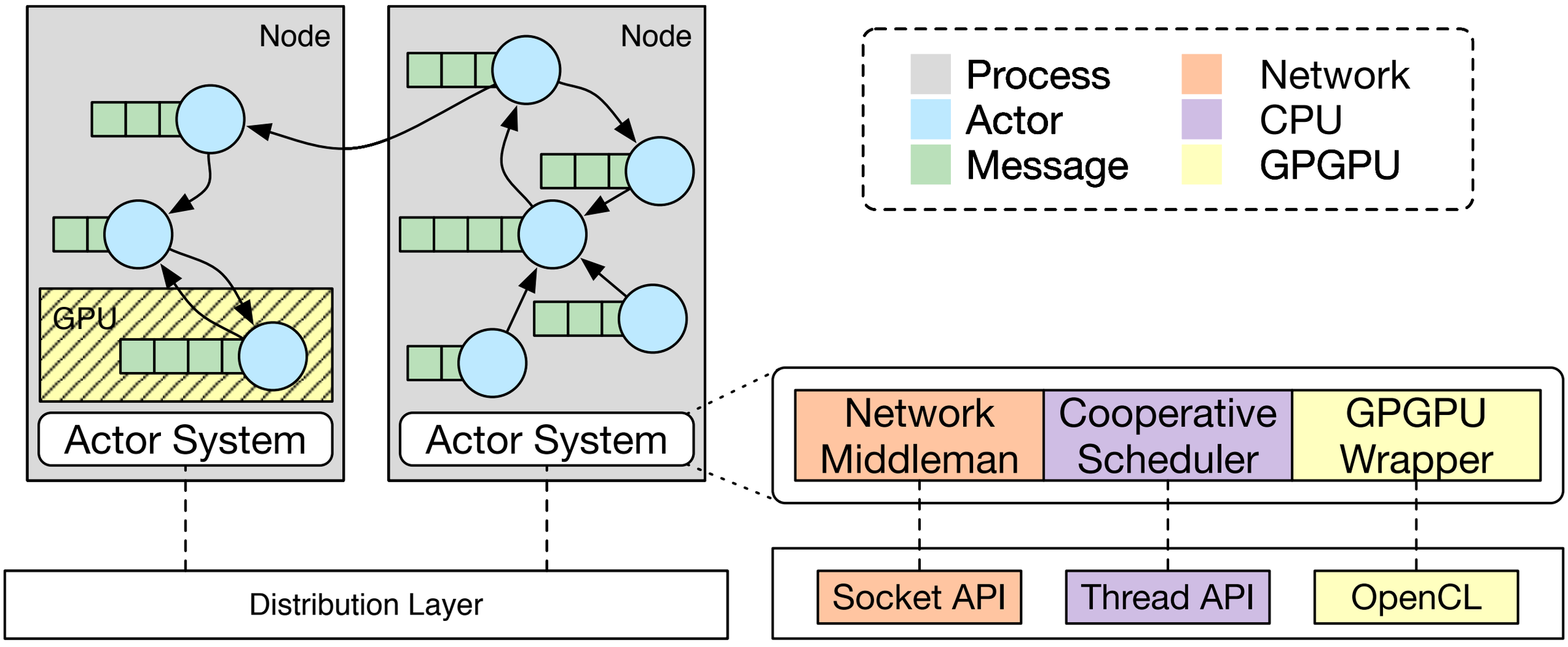


How can I program with actors in C++?

C++ Actor Framework

- **Lightweight & fast** actor model implementation
- **In active development** since 2011
- **Provides building** blocks for infrastructure software
- **~80,000 lines of code** (https://www.openhub.net/p/actor-framework)
- **International users** from MMO gaming to finance

Architecture



Our First Example

Replaces the traditional `main()`

Hosts all of our actors

Starts a new actor

```
void caf_main(actor_system& system
  // create a 'mirror'
  auto mirror_actor = system.spawn(mirror);
  // create a 'hello_world' actor
  system.spawn(hello_world, mirror_actor);
  // system will wait until both actors are destroyed
}

CAF_MAIN()
```

Mirror, mirror, on the Wall

```
behavior mirror(event_based_actor* self) {
  // return the (initial) actor behavior
  return {
    // a handler for messages containing a single string
    // that replies with a string
    [=](const string& what) -> string {
      // prints "Hello World!" via aout
      aout(self) << what << endl;
      // reply "!dlroW olleH"
      return string(what.rbegin(), what.rend());
    }
  };
}
```

Hello World!

```
void hello_world(event_based_actor* self, actor buddy) {
  // send "Hello World!" to our buddy ...
  self->request(buddy, std::chrono::seconds(10),
                "Hello World!")
  .then(
    // ... wait up to 10s for a response ...
    [=](const string& what) {
      // ... and print it
      aout(self) << what << endl;
    }
  );
}
```

Ok, but what about streams?

Streams

- Conceptually: potentially **infinite lists**
- Usually **never fully present in memory** at any time
- Allow **chunked processing** of huge data volumes
- Enable **realtime event handling** (e.g. Twitter feeds)

Just Send Messages!?

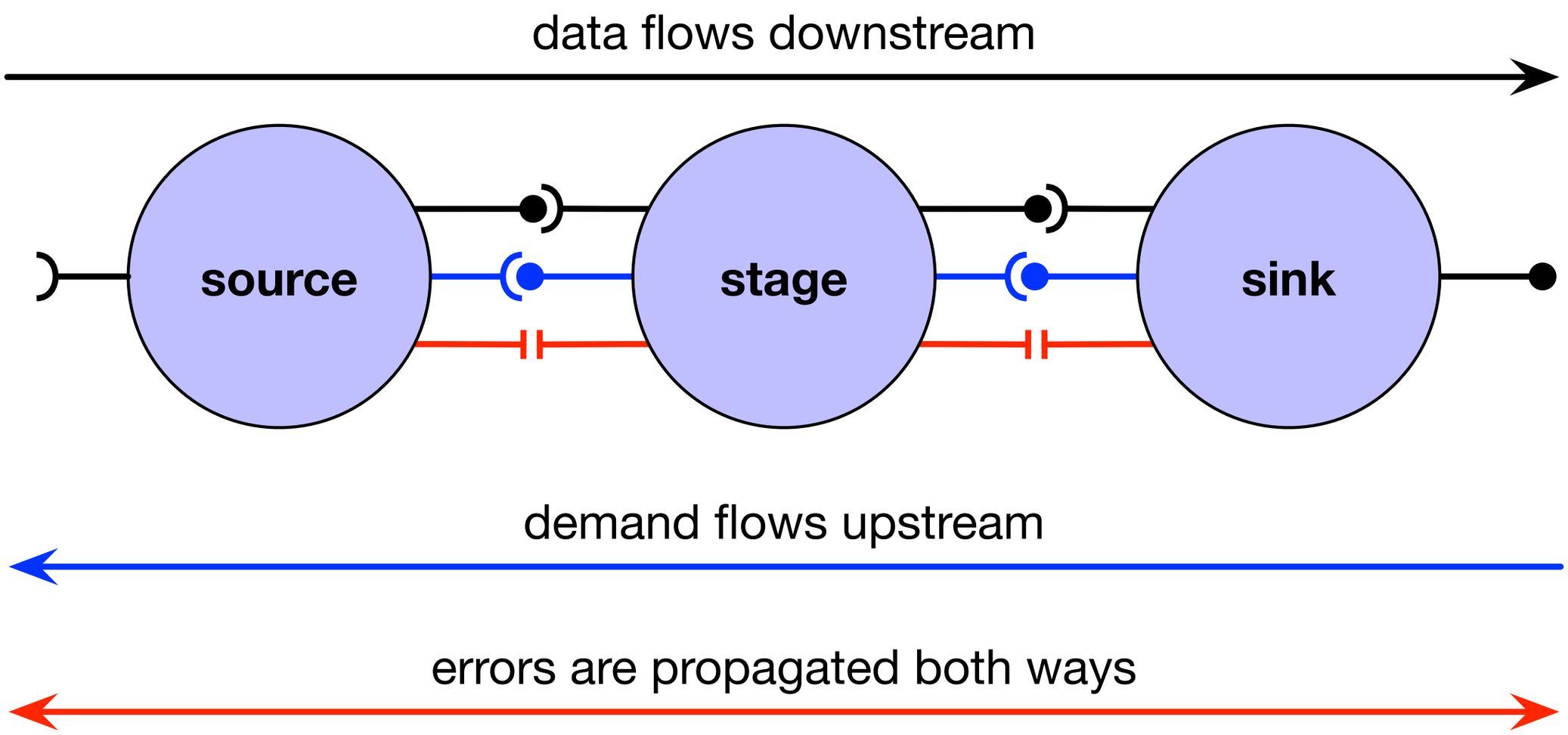
- Actors have **unbounded mailboxes**
  - **No feedback to sender** regarding mailbox load
    - Fast senders eventually **overwhelm receivers**
- **Overhead per message** too high for little data
  - Wrapping each item of a stream **wastes memory**
  - **Batch processing much faster** than one-by-one

Streams in CAF

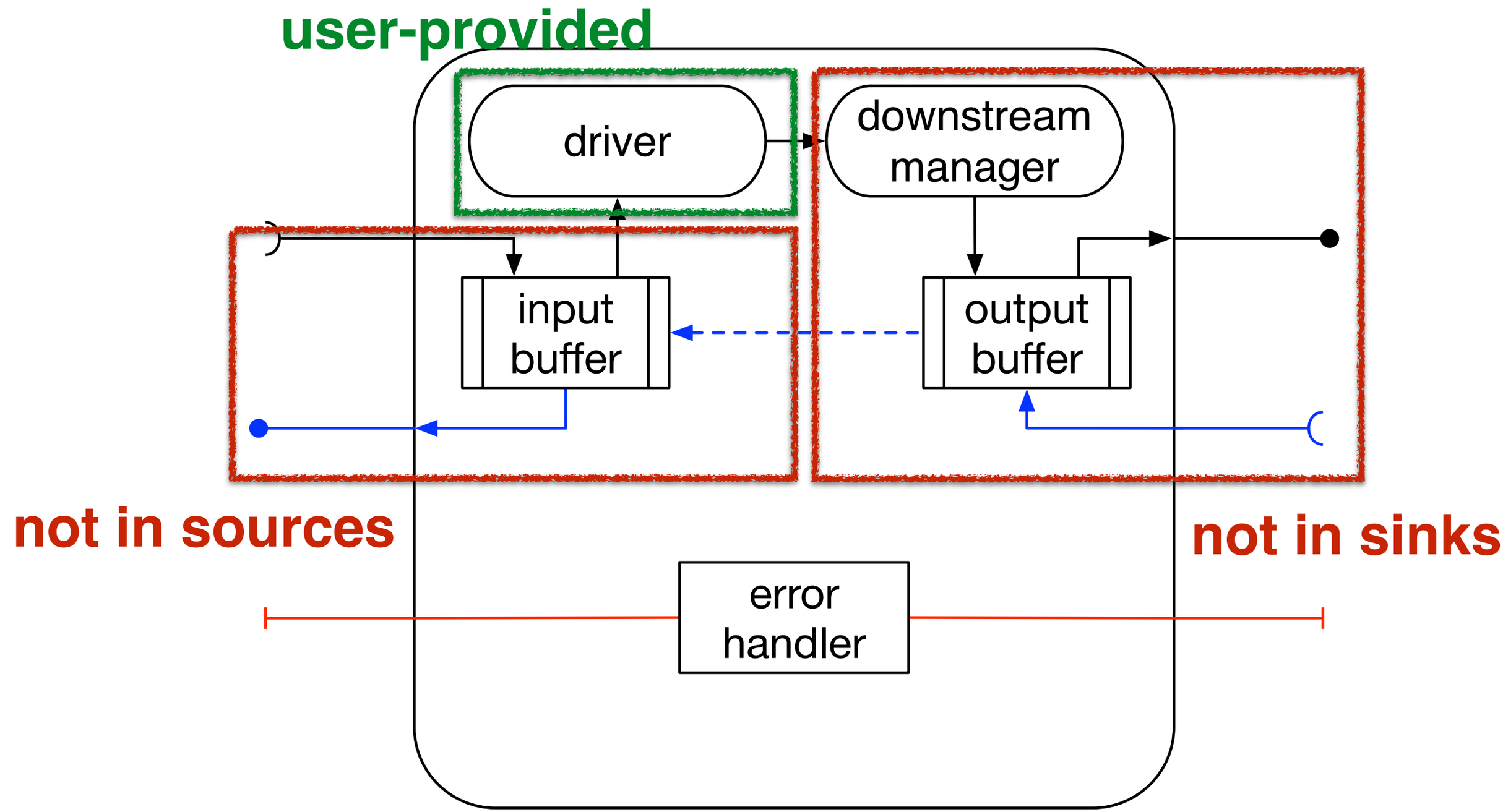
- **Stream topologies** can span any number of actors
- **Demand signaling** slows down senders if needed
- **Load-balancing and broadcasting*** via stages
- **Stream priorities*** allow fine-grained flow control

* only partially implemented as of CAF 0.16

Streaming Concept



Per-Stream State



Implementing a Source

```
behavior int_source(event_based_actor* self) { // Makes ints [0, n)
  return {
    [=](open_atom, int n) {
      return self->make_source(
        [](int& x) { // Initializer
          x = 0;
        },
        [n](int& x, downstream<int>& out, size_t hint) { // Generator
          auto max_x = std::min(x + static_cast<int>(hint), n);
          for (; x < max_x; ++x)
            out.push(x);
        },
        [n](const int& x) { // End predicate
          return x == n;
        });
    }
  };
}
```

Driver implementation

Implementing a Stage

```
behavior int_selector(event_based_actor* self) { // Drops odd numbers
  return {
    [=](stream<int> in) {
      return self->make_stage(
        in, // Our input source
        [](unit_t&) { // Initializer (uses unit_t for "no state")
          // nop
        },
        [](unit_t&, downstream<int>& out, int val) { // Processor
          if (val % 2 == 0)
            out.push(val);
        },
        [=](unit_t&, const error& err) { // Finalizer
          // Check for error ...
        });
    }
  };
}
```

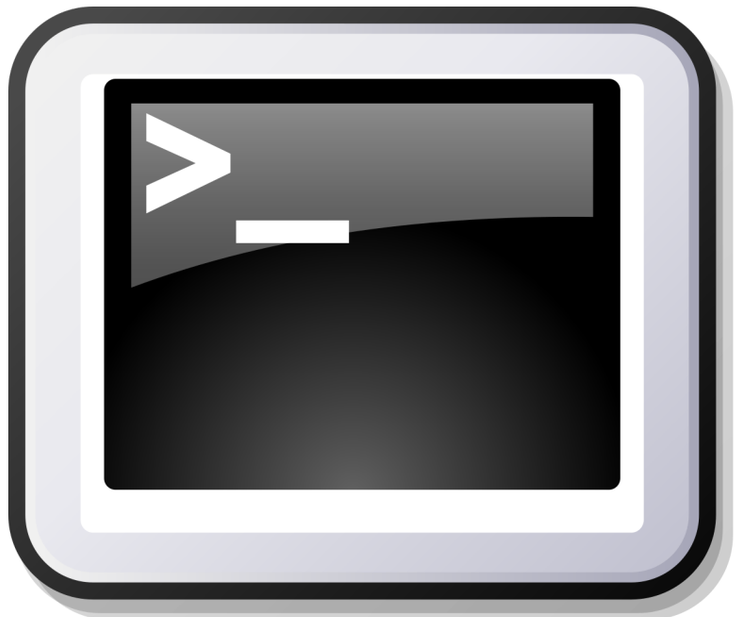
Implementing a Sink

```
behavior int_sink(event_based_actor* self) {
  return {
    [=](stream<int> in) {
     return self->make_sink(
         in, // Our input source
         [](std::vector<int>&) { // Initializer
           // nop
         },
         [](std::vector<int>& xs, int val) { // Consumer
           xs.emplace_back(val);
         },
         [=](std::vector<int>& xs, const error& err) { // Finalizer
           // Check for error, do something with xs ...
         });
    }
  };
}
```

Putting it Together

```
void caf_main(actor_system& sys, const config& cfg) {
  auto src = sys.spawn(int_source);
  auto stg = sys.spawn(int_selector);
  auto snk = sys.spawn(int_sink);
  auto pipeline = snk * stg * src;
  anon_send(pipeline, open_atom::value, cfg.n);
}
```

How fast is it, tho?



Demo Time

Thanks for Listening!

actor-framework

actor_framework