CAF
C++ Actor Framework

Matthias Vallentin

UC Berkeley

Berkeley C++ Summit

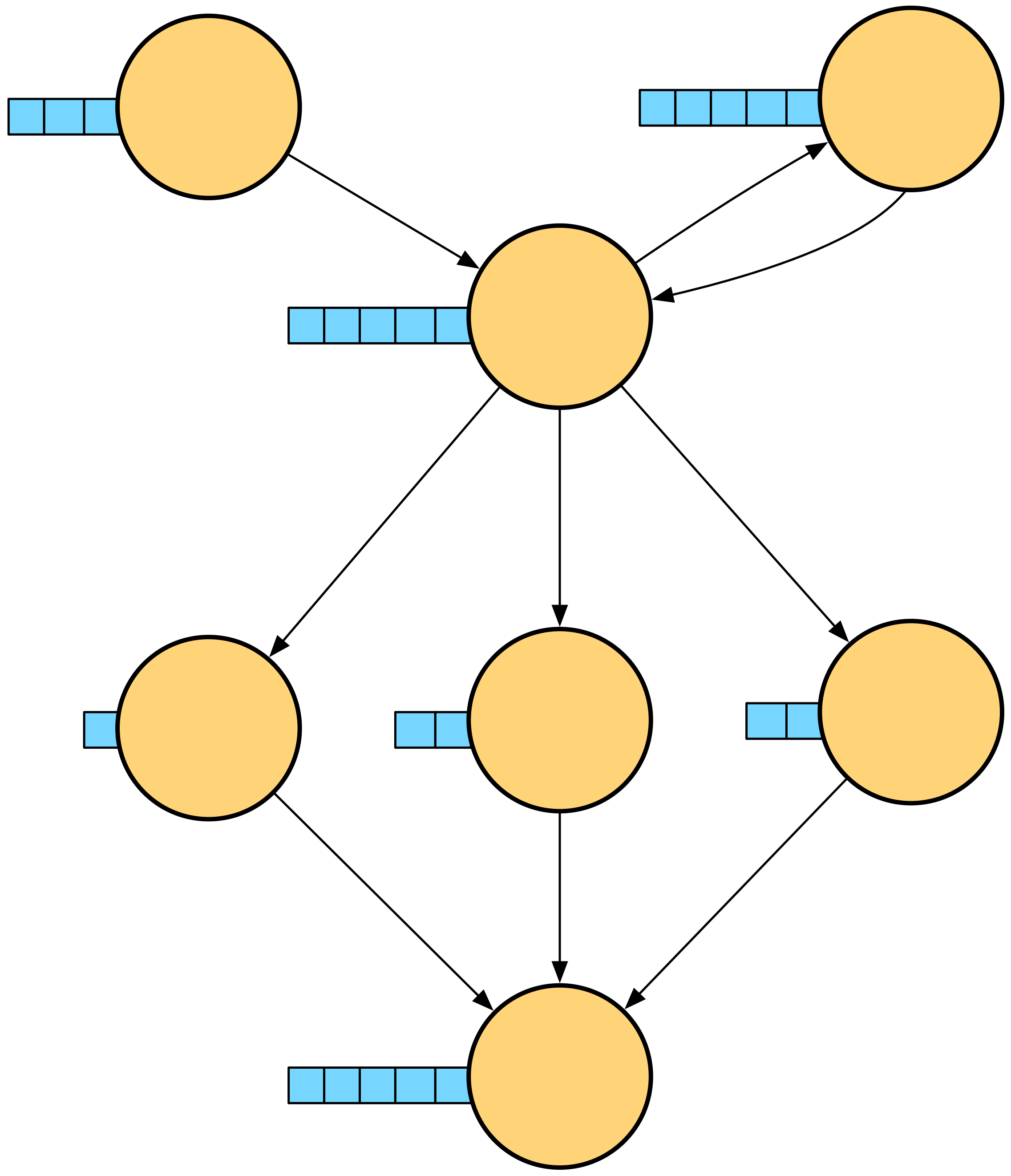
October 17, 2016

Outline

- Actor Model
- CAF
- Evaluation

Actor Model

- **Actor**: sequential unit of computation
- **Message**: tuple
- **Mailbox**: message queue
- **Behavior**: function how to process next message



Actor Semantics

- All actors execute **concurrently**
- Actors are **reactive**
- In response to a message, an actor can do *any* of:
  1. Creating (*spawning*) new actors
  2. Sending messages to other actors
  3. Designating a behavior for the next message

CAF
(C++ Actor Framework)

Example #1

An **actor** is typically implemented as a **function**

```
behavior adder() {
  return {
    [](int x, int y) {
      return x + y;
    },
    [](double x, double y) {
      return x + y;
    }
  };
}
```

A list of **lambdas** determines the **behavior** of the actor.

A non-void return value sends a **response message** back to the sender

Example #2

```
int main() {
  actor_system_config cfg;
  actor_system sys{cfg};
  // Create (spawn) our actor.
  auto a = sys.spawn(adder);
  // Send it a message.
  scoped_actor self{sys};
  self->send(a, 40, 2);
  // Block and wait for reply.
  self->receive(
    [](int result) {
      cout << result << endl; // prints “42”
    }
  );
}
```

Encapsulates all global state (worker threads, actors, types, etc.)

Spawns an actor valid only for the current scope.

Example #3

Optional first argument to running actor.

```
auto a = sys.spawn(adder);
sys.spawn(
  [=](event_based_actor* self) -> behavior {
    self->send(a, 40, 2);
    return {
      [=](int result) {
        cout << result << endl;
        self->quit();
      }
    };
  }
);
```

Capture **by value** because `spawn` returns immediately.

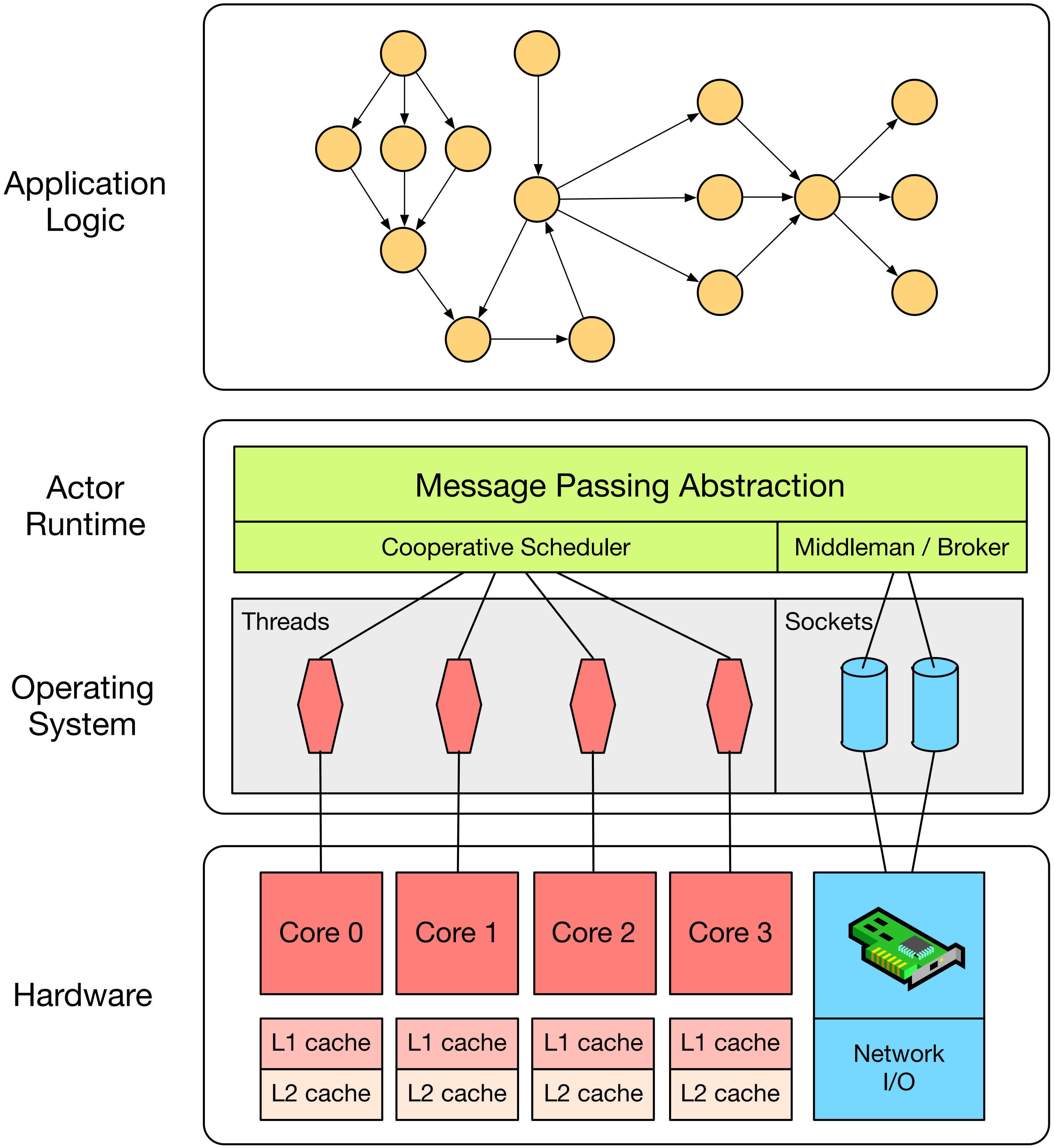
Example #4

Request-response communication requires timeout.
(`std::chrono::duration`)

```
auto a = sys.spawn(adder);
sys.spawn(
  [=](event_based_actor* self) {
    self->request(a, seconds(1), 40, 2).then(
      [=](int result) {
        cout << result << endl;
      }
    };
  }
);
```

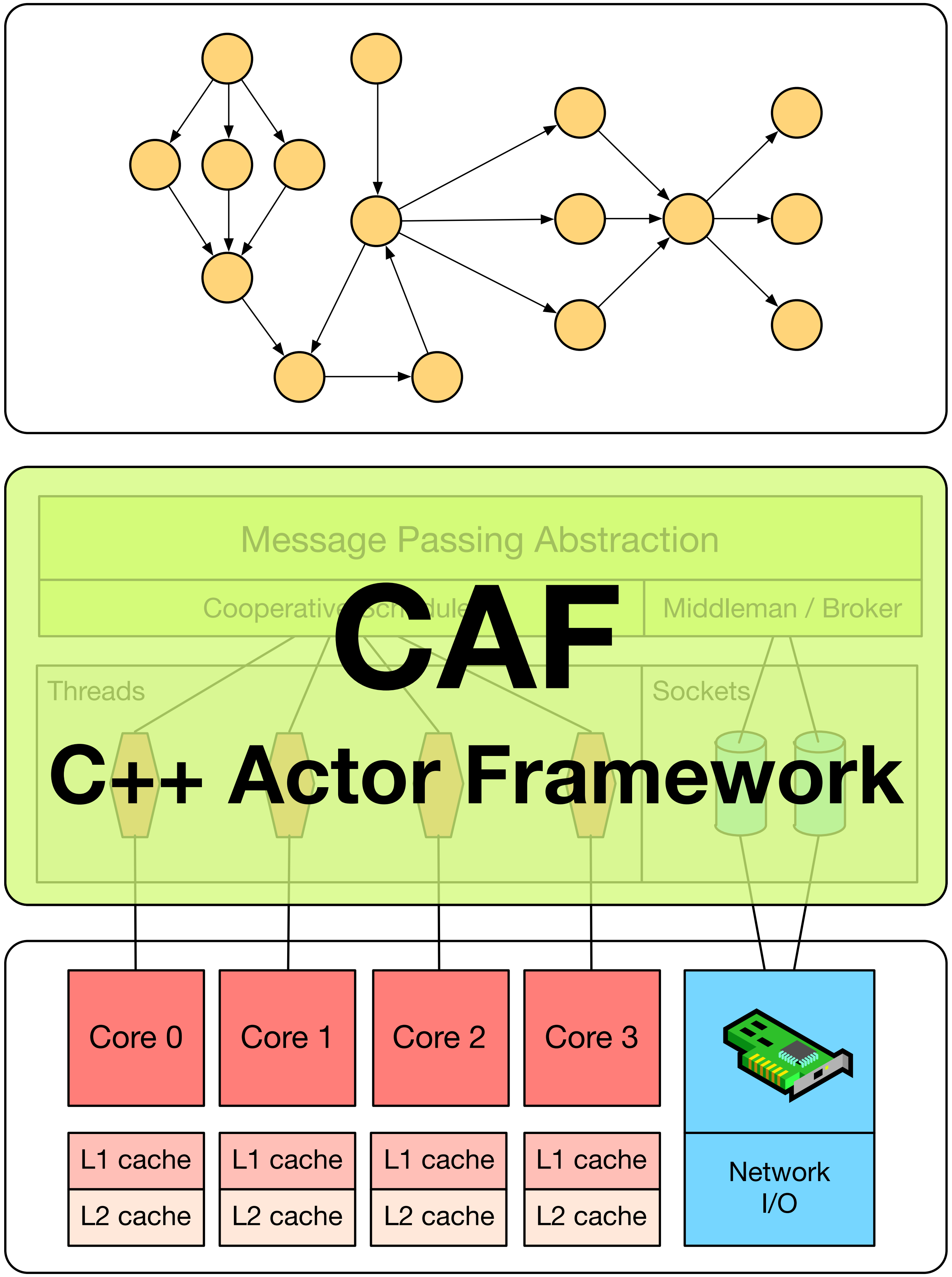
Continuation specified as `behavior`.

No behavior returned,
actor terminates after executing one-shot continuation.


Application Logic
Actor Runtime
Message Passing Abstraction
Cooperative Scheduler
Middleman / Broker
Operating System
Threads
Sockets
Hardware
Core 0
Core 1
Core 2
Core 3
L1 cache
L2 cache
L1 cache
L2 cache
L1 cache
L2 cache
L1 cache
L2 cache
Network I/O

CAF

C++ Actor Framework



Application Logic

Actor Runtime

Operating System

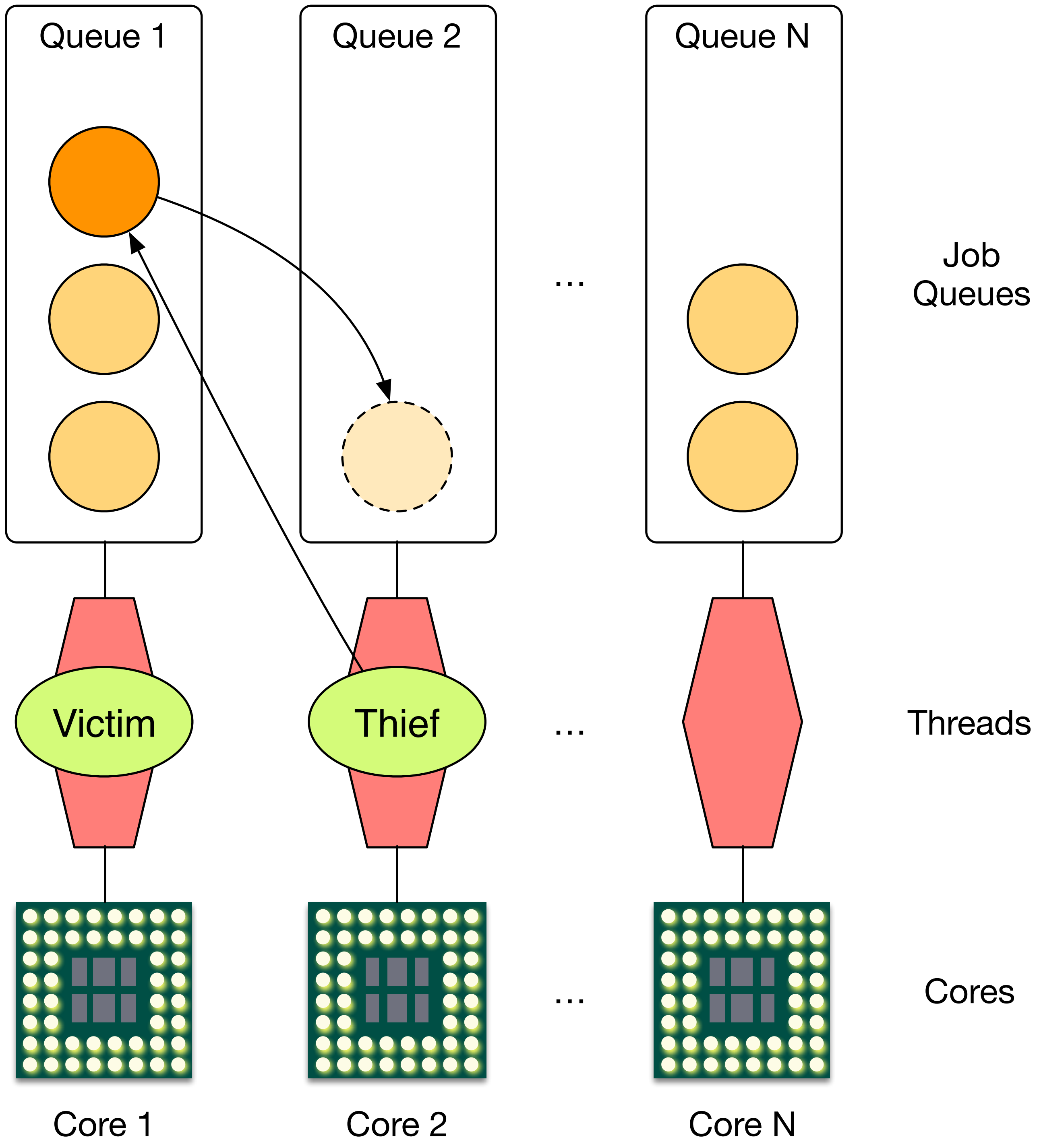
Hardware

Scheduler

- Maps N **jobs** (= actors) to M **workers** (= threads)
- Limitation: **cooperative multi-tasking** in user-space
- **Issue**: actors that block
  - Can lead to **starvation** and/or scheduling imbalances
  - Not well-suited for **I/O-heavy tasks**
  - Current solution: detach "uncooperative" actors into **separate thread**

Work Stealing*

- **Decentralized**: one job queue and worker thread per core
- On empty queue, **steal** from other thread
- Efficient if stealing is a rare event
- Implementation: deque with two spinlocks



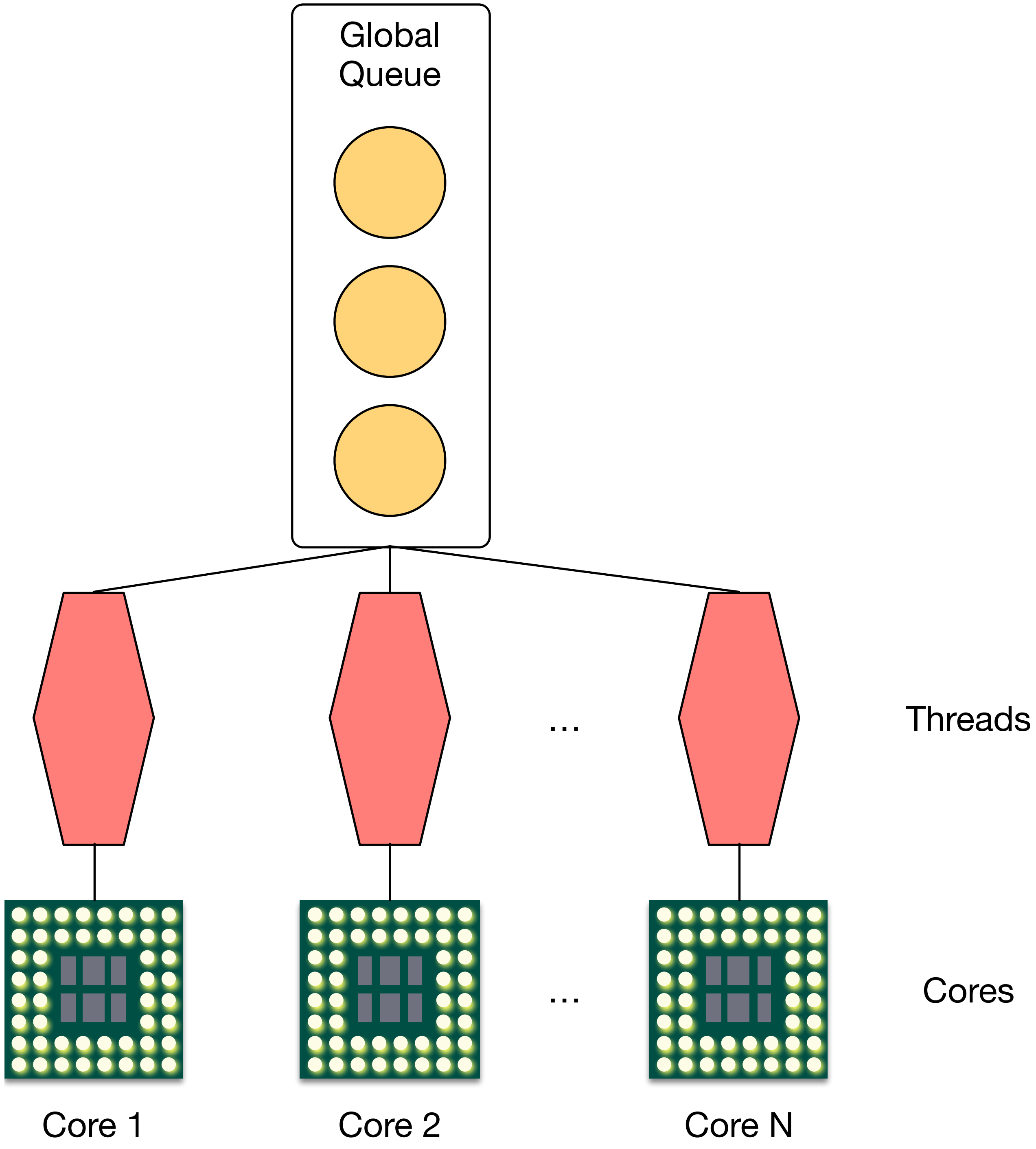
*Robert D. Blumofe and Charles E. Leiserson. *Scheduling Multithreaded Computations by Work Stealing*. J. ACM, 46(5):720–748, September 1999.

Implementation

```
template <class Worker>
resumable* dequeue(Worker* self) {
  auto& strategies = self->data().strategies;
  resumable* job = nullptr;
  for (auto& strat : strategies) {
    for (size_t i = 0; i < strat.attempts; i += strat.step_size) {
      // try to grab a job from the front of the queue
      job = self->data().queue.take_head();
      // if we have still jobs, we're good to go
      if (job)
        return job;
      // try to steal every X poll attempts
      if ((i % strat.steal_interval) == 0) {
        if (job = try_steal(self))
         return job;
      }
      if (strat.sleep_duration.count() > 0)
        std::this_thread::sleep_for(strat.sleep_duration);
    }
  }
  // unreachable, because the last strategy loops
  // until a job has been dequeued
  return nullptr;
}
```

Work Sharing

- **Centralized**: one shared global queue
- Synchronization: **mutex & CV**
- **No polling**
  - less CPU usage
  - lower throughouput
- Good **for low-power devices**
  - Embedded / IoT



Copy-On-Write

- `caf::message` = atomic, intrusive ref-counted tuple
  - **Immutable access** permitted
  - **Mutable access** with ref count > 1 invokes copy constructor
- **Constness deduced** from message handlers
- **No data races** by design
- **Value semantics**, no complex lifetime management

```
auto heavy = vector<char>(1024 * 1024);
auto msg = make_message(move(heavy));
for (auto& r : receivers)
  send(r, msg);

behavior reader() {
  return {
    [=](const vector<char>& buf) {
      f(buf);
    }
  };
}

behavior writer() {
  return {
    [=](vector<char>& buf) {
      f(buf);
    }
  };
}
```

Type Safety

- CAF has **statically** and **dynamically typed** actors
- **Dynamic**
  - Type-erased `caf::message` hides tuple types
  - Message types checked **at runtime** only
- **Static**
  - **Type signature** verified at sender and receiver
  - Message protocol checked **at compile time**

Interface

```
// Atom: typed integer with semantics
using plus_atom = atom_constant<atom("plus")>;
using minus_atom = atom_constant<atom("minus")>;
using result_atom = atom_constant<atom("result")>;

// Actor type definition
using math_actor =
  typed_actor<
    replies_to<plus_atom, int, int>::with<result_atom, int>,
    replies_to<minus_atom, int, int>::with<result_atom, int>
  >;
```

Signature of **incoming** message

Signature of (optional) **response** message

Implementation

Dynamic

```
behavior math_fun(event_based_actor* self) {
  return {
    [](plus_atom, int a, int b) {
      return make_tuple(result_atom::value, a + b);
    },
    [](minus_atom, int a, int b) {
      return make_tuple(result_atom::value, a - b);
    }
  };
}
```

Static

```
math_actor::behavior_type typed_math_fun(math_actor::pointer self) {
  return {
    [](plus_atom, int a, int b) {
      return make_tuple(result_atom::value, a + b);
    },
    [](minus_atom, int a, int b) {
      return make_tuple(result_atom::value, a - b);
    }
  };
}
```

Error Example

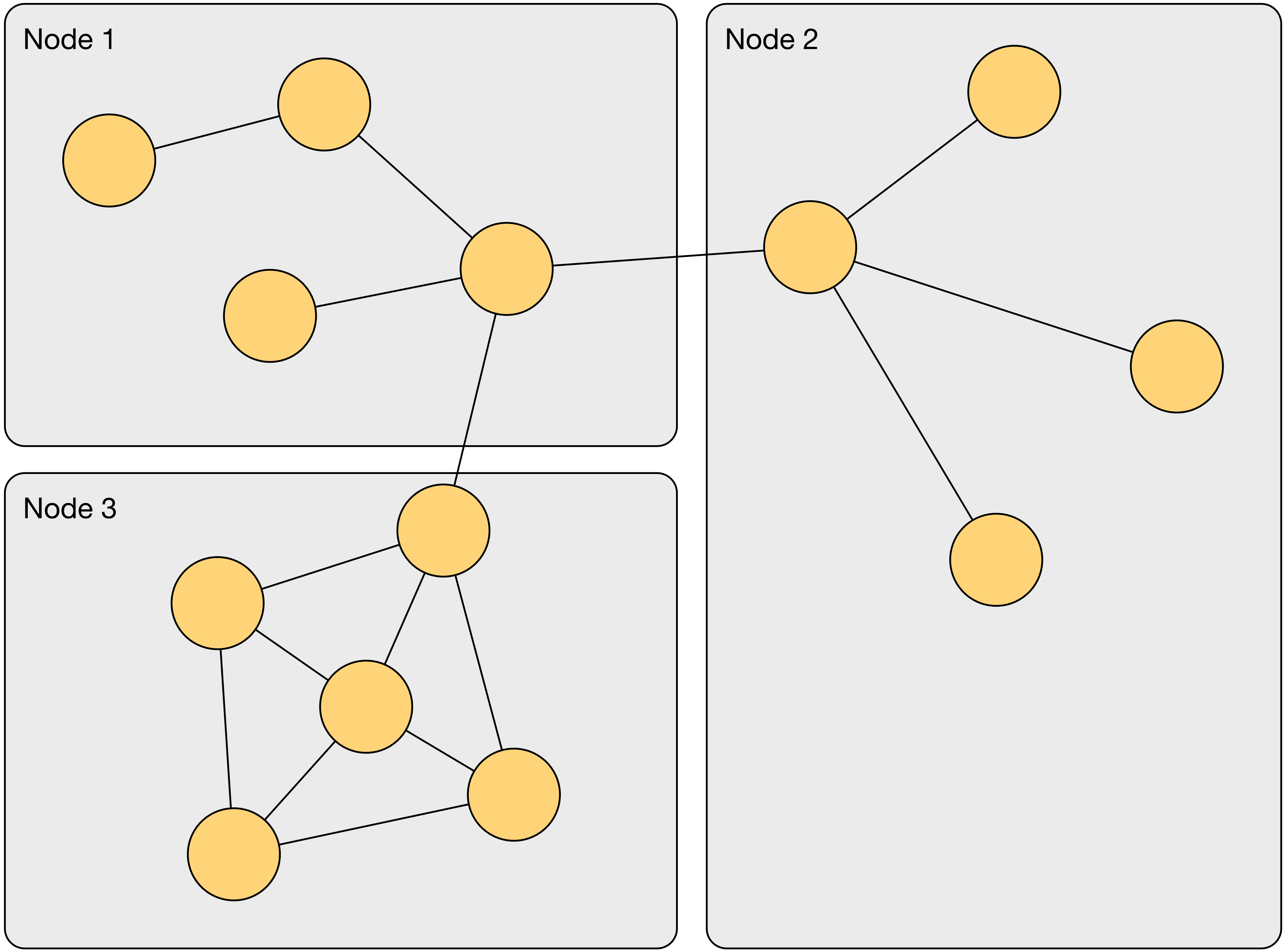
```
auto self = sys.spawn(...);
math_actor m = self->typed_spawn(typed_math);
self->request(m, seconds(1), plus_atom::value, 10, 20).then(
  [](result_atom, float result) {
    // …
  }
);
```

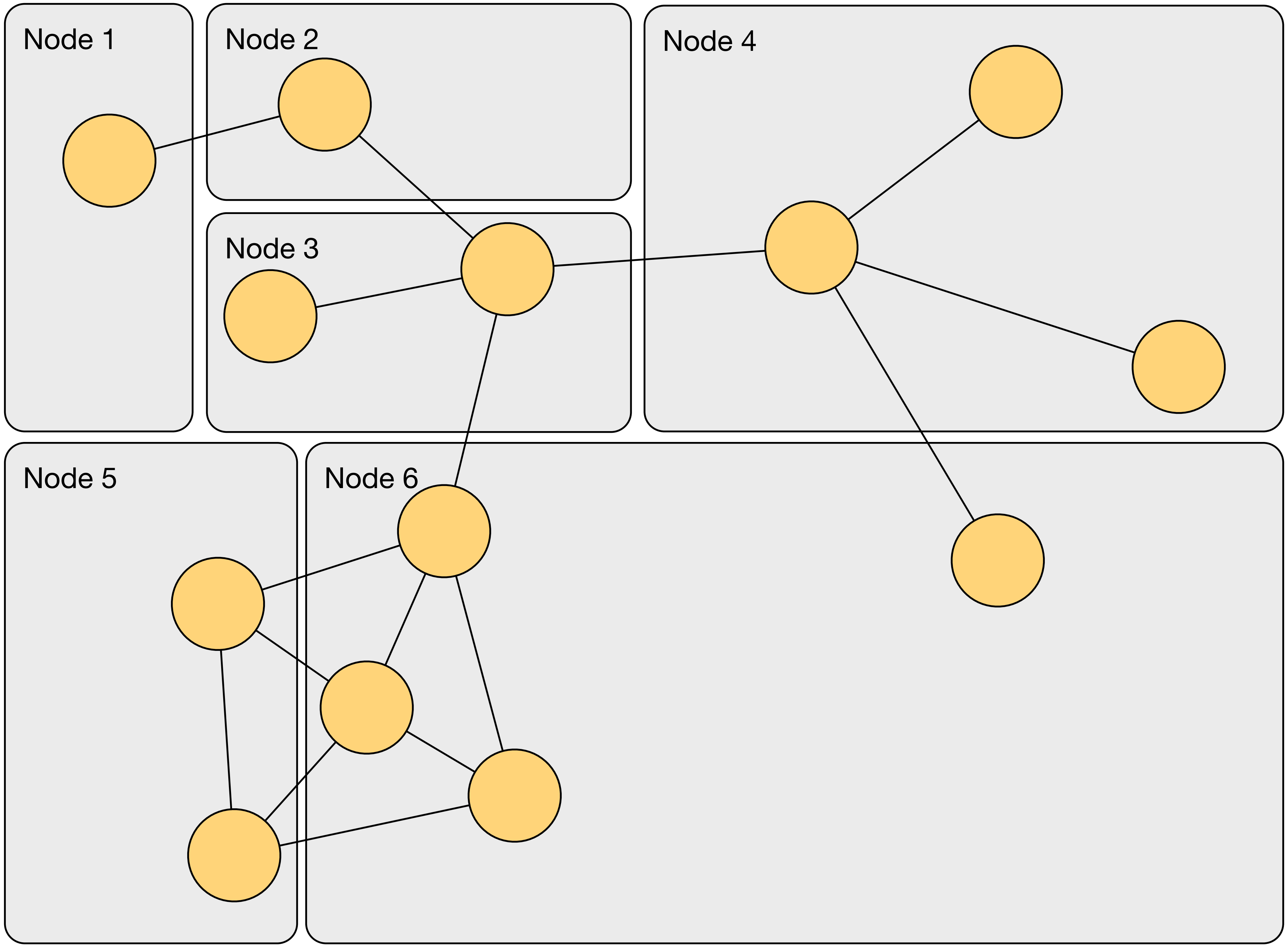
Compiler complains about invalid response type

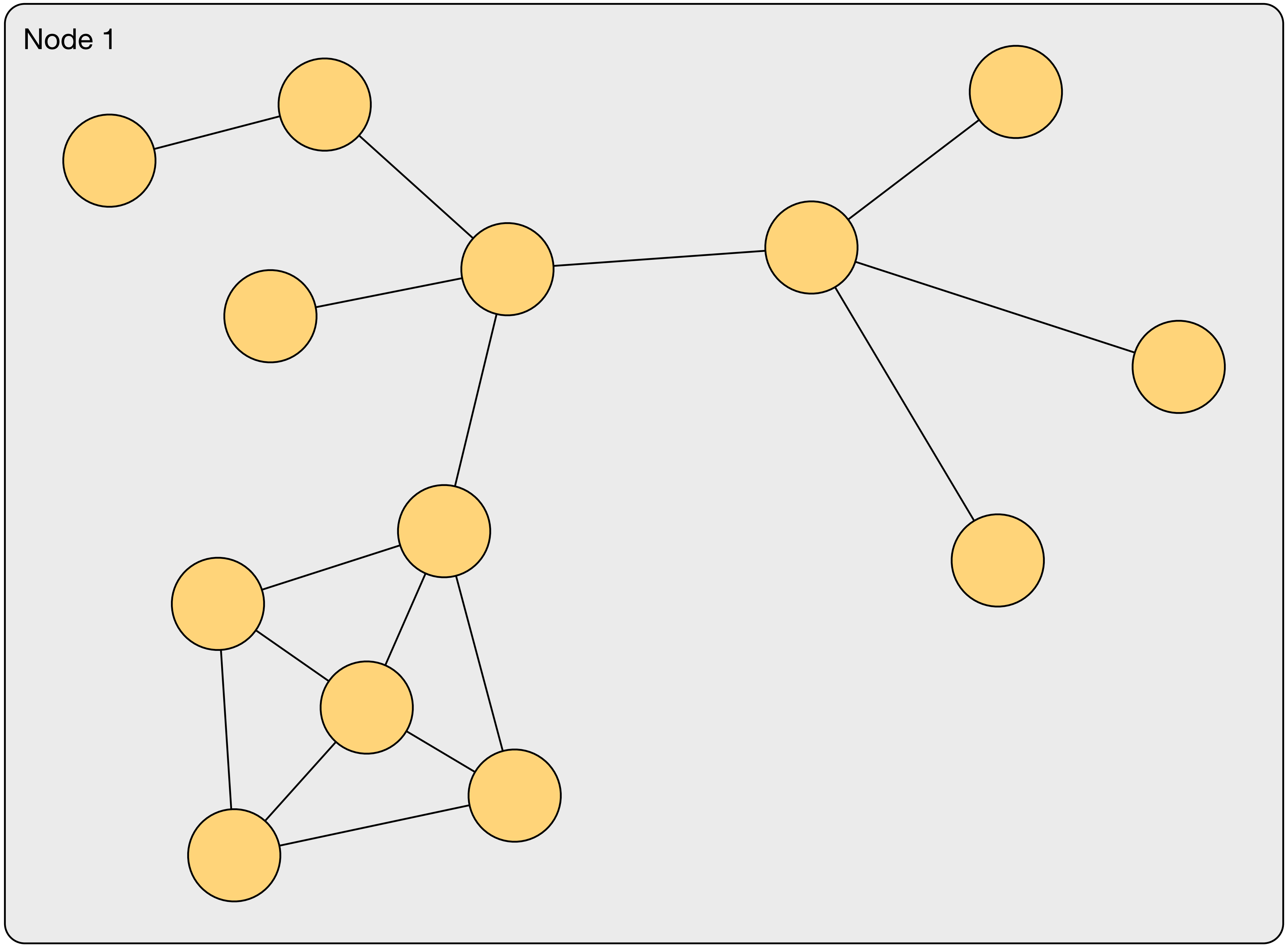
Network Transparency

Separation of **application logic** from **deployment**

- Significant **productivity gains**
  - Spend *more time* with **domain-specific code**
  - Spend *less time* with **network glue code**


Node 1
Node 2
Node 3


Node 1
Node 2
Node 3
Node 4
Node 5
Node 6


Node 1

Example

```
int main(int argc, char** argv) {
  // Defaults.
  auto host = "localhost"s;
  auto port = uint16_t{42000};
  auto server = false;
  actor_system sys{...}; // Parse command line and setup actor system.
  auto& middleman = sys.middleman();
  actor a;
  if (server) {
    a = sys.spawn(math);
    auto bound = middleman.publish(a, port);
    if (bound == 0)
      return 1;
  } else {
    auto r = middleman.remote_actor(host, port);
    if (!r)
      return 1;
    a = *r;
  }
  // Interact with actor a
}
```

Reference to CAF's network component.

Publish specific actor at a TCP port.
Returns bound port on success.

Connect to published actor at TCP endpoint.
Returns `expected<actor>`.

Failures

Components fail regularly in large-scale systems

- Actor model provides **monitors** and **links**
    - **Monitor**: subscribe to exit of actor (**unidirectional**)
    - **Link**: bind own lifetime to other actor (**bidirectional**)

Monitor Example

```
behavior adder() {
  return {
    [](int x, int y) {
      return x + y;
    }
  };
}

auto self = sys.spawn<monitored>(adder);
self->set_down_handler(
  [](const down_msg& msg) {
    cout << "actor DOWN: " << msg.reason << endl;
  }
);
```

Spawn flag denotes monitoring.
Also possible later via **`self->monitor(other);`**

Link Example

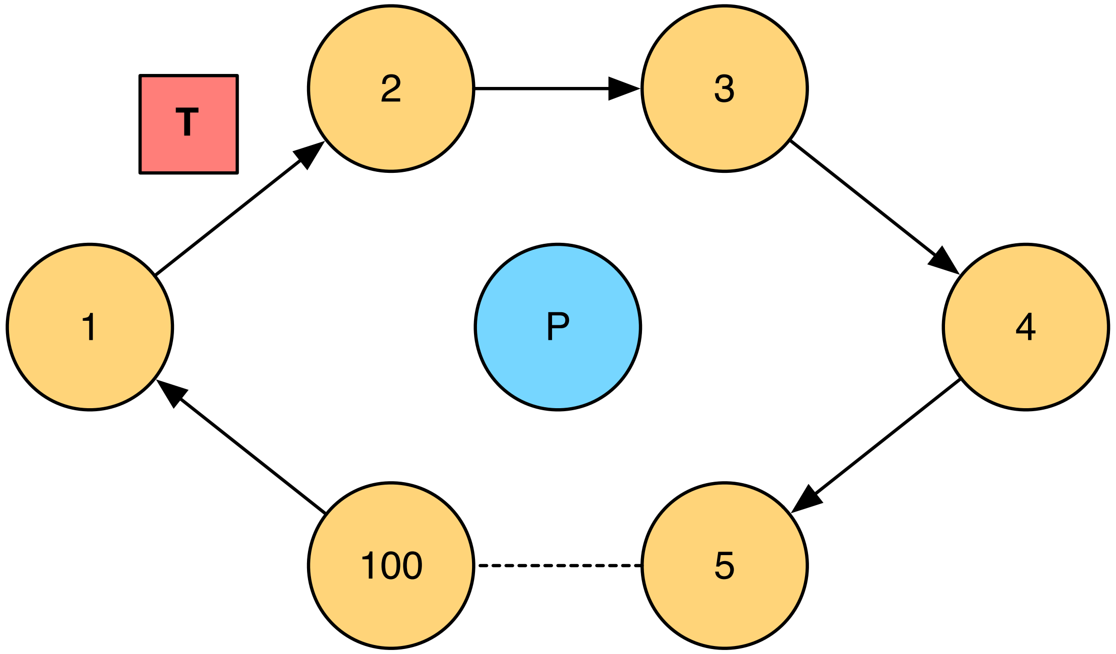
```
behavior adder() {
  return {
    [](int x, int y) {
      return x + y;
    }
  };
}

auto self = sys.spawn<linked>(adder);
self->set_exit_handler(
  [](const exit_msg& msg) {
    cout << "actor EXIT: " << msg.reason << endl;
  }
);
```

Spawn flag denotes linking.
Also possible later via `self->link_to(other);`

Evaluation

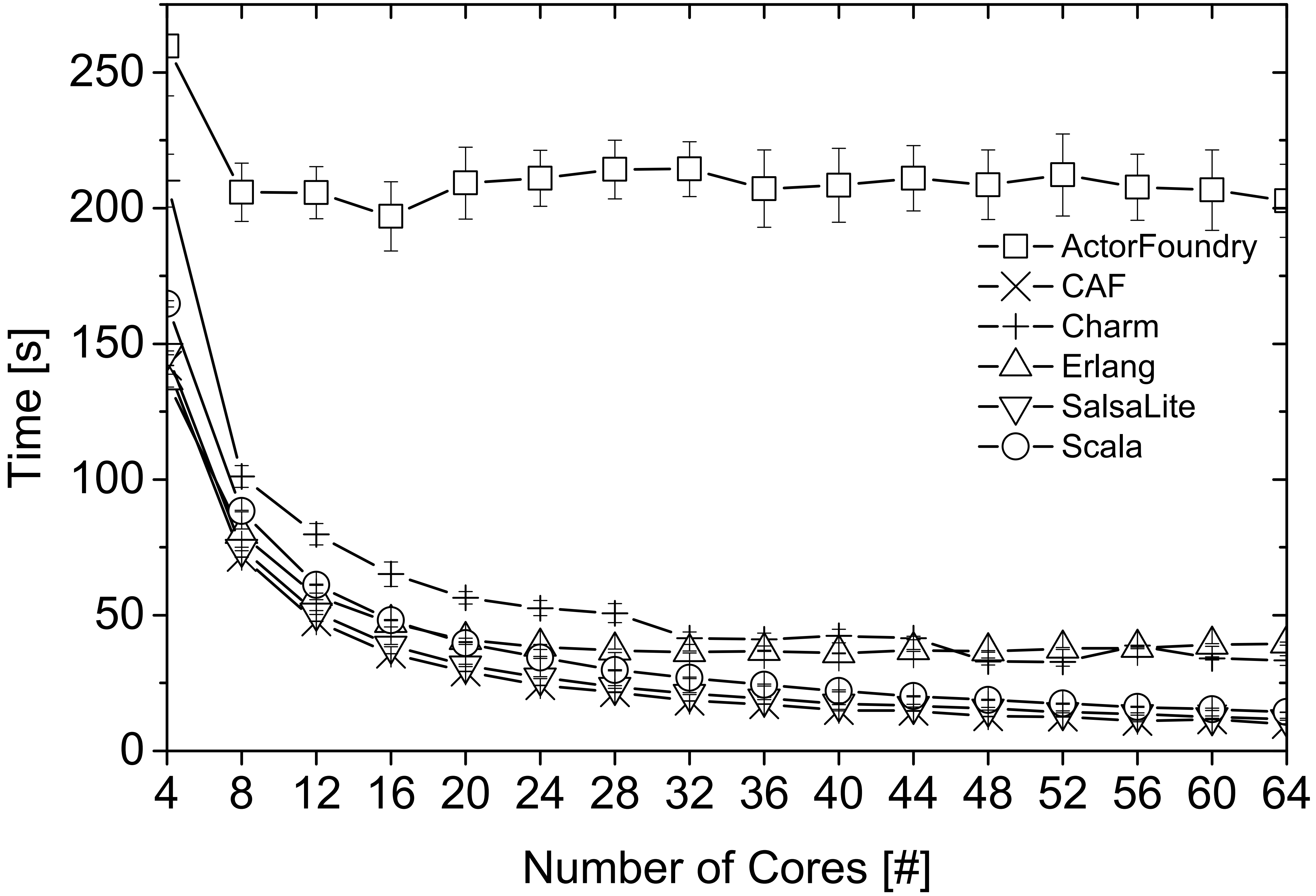
https://github.com/actor-framework/benchmarks

Setup #1

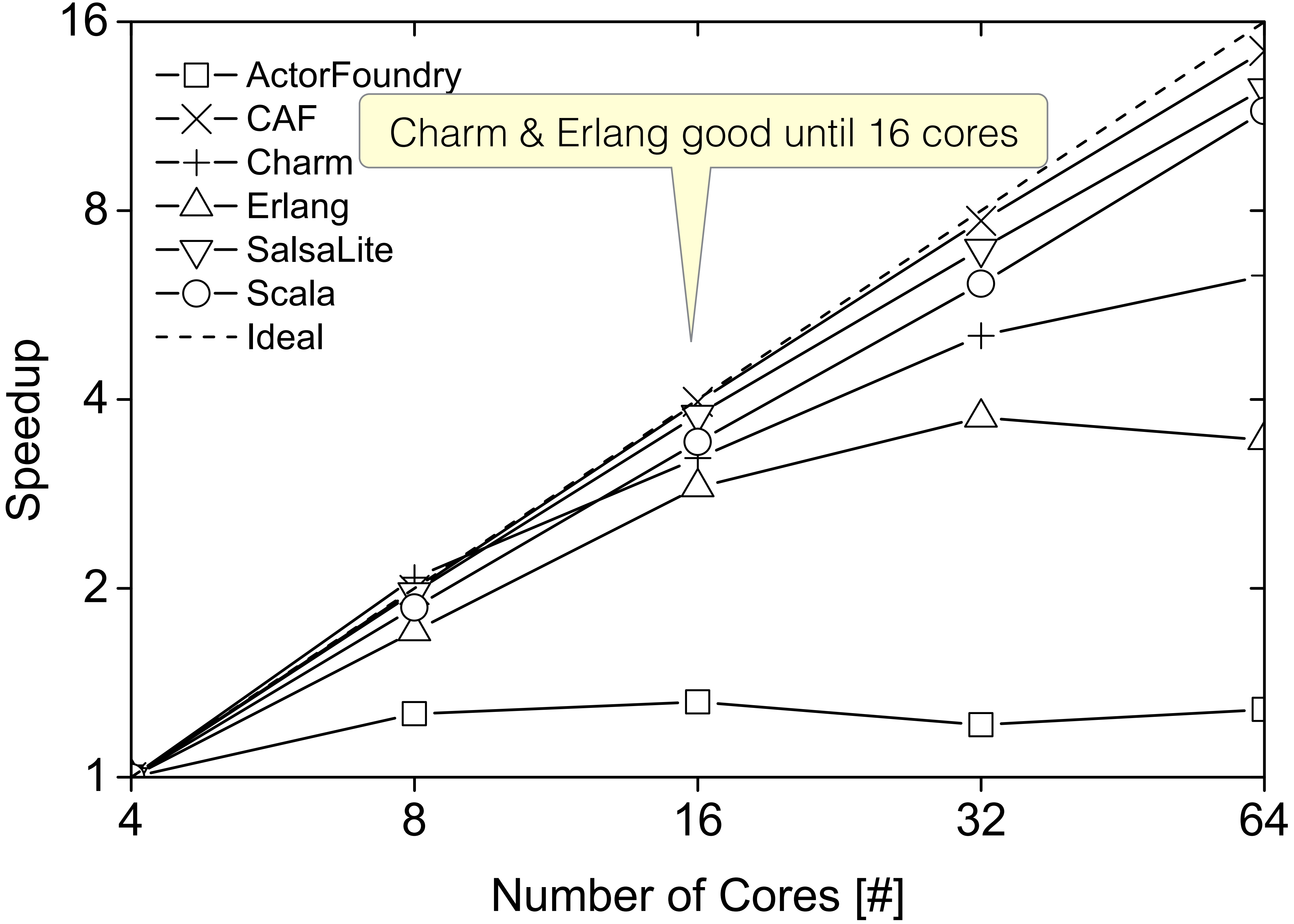


- 100 rings of 100 actors each
- Actors forward single token 1K times, then terminate
- 4 re-creations per ring
- One actor per ring performs *prime factorization*
- Resulting workload: high message & CPU pressure
- Ideal: 2 x cores $\Longrightarrow$ 0.5 x runtime

Performance

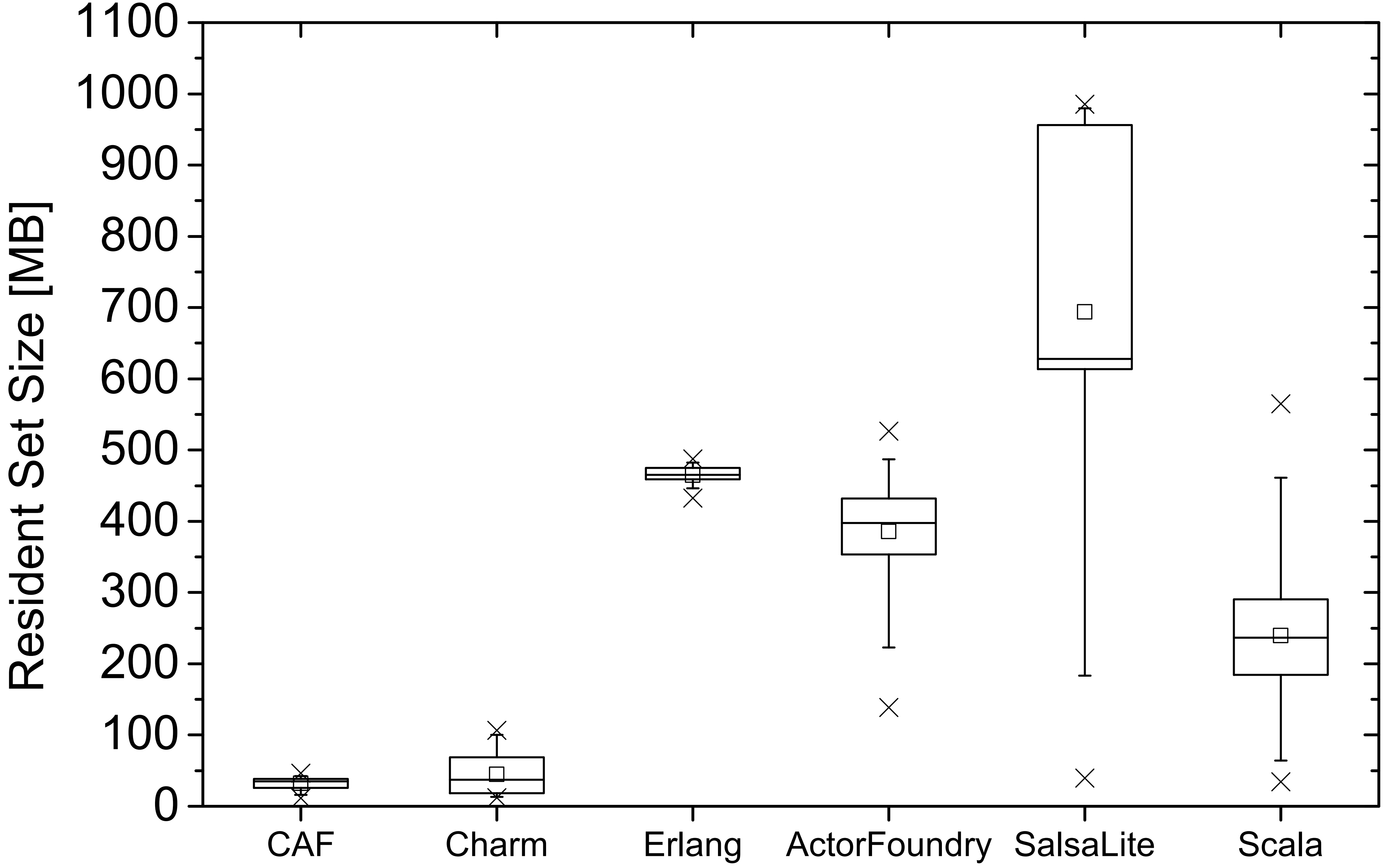


(normalized)

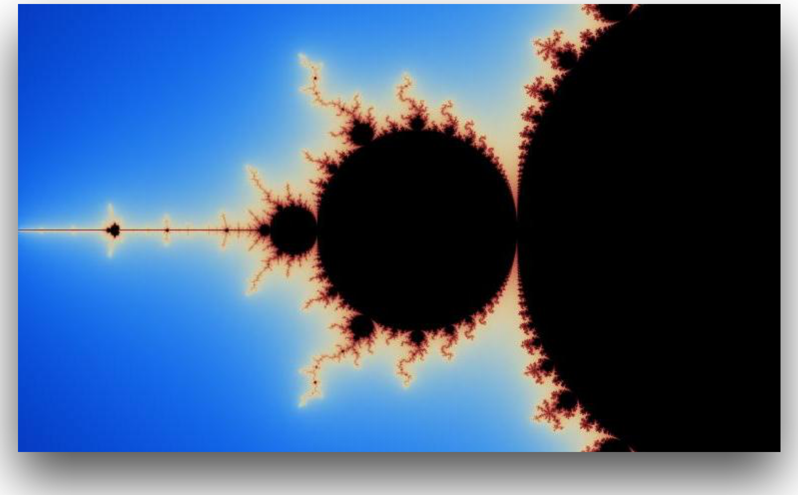


Charm & Erlang good until 16 cores

Memory Overhead

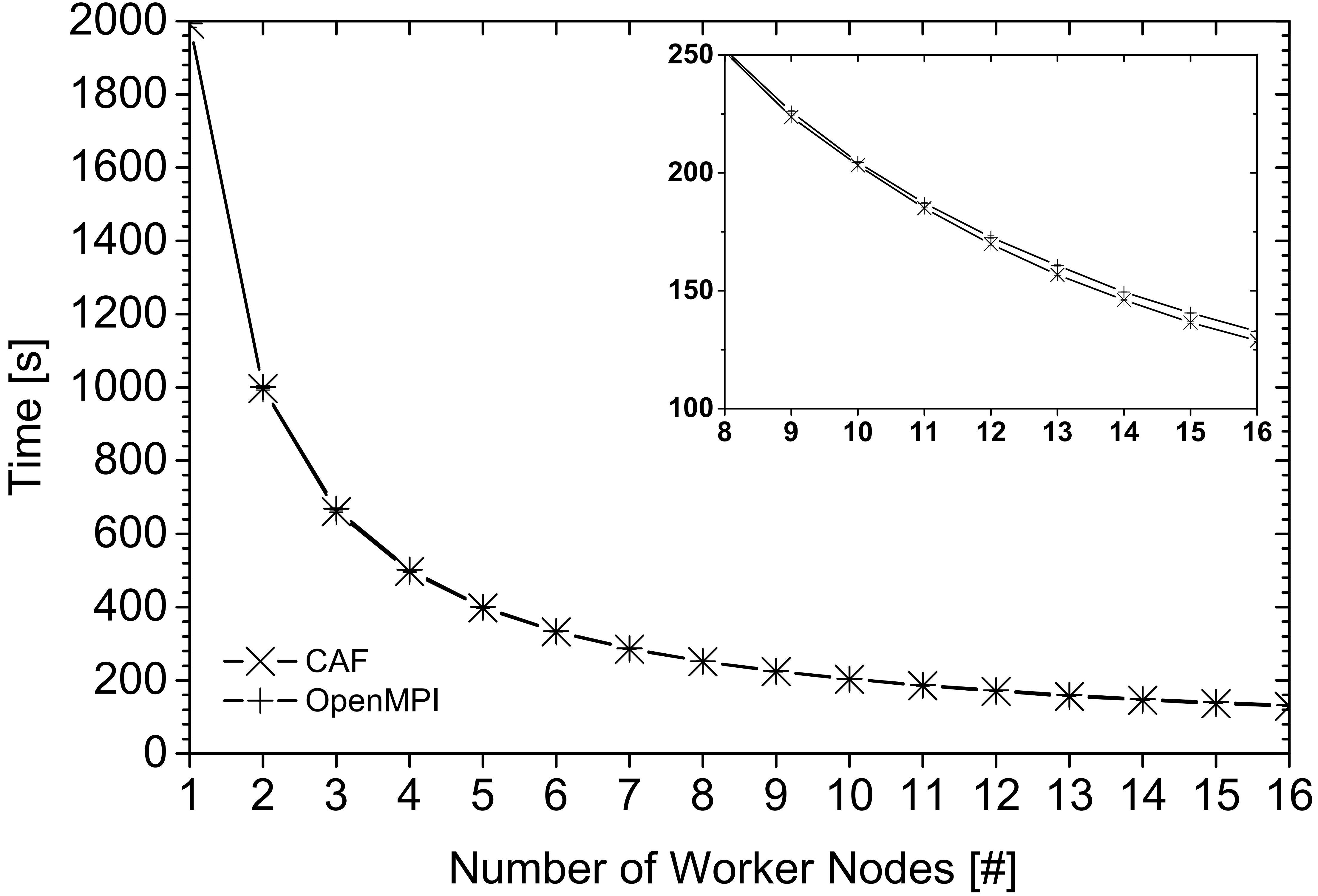


Setup #2



- Compute images of Mandelbrot set
- Divide & conquer algorithm
- Compare against OpenMPI (via Boost.MPI)
  - Only message passing layers differ
- 16-node cluster: quad-core Intel i7 3.4 GHz

CAF vs. OpenMPI



Project

- Lead: **Dominik Charousset** (HAW Hamburg)
  - Started CAF as Master's thesis
  - Active development as part of his Ph.D.
- Dual-licensed: 3-clause **BSD** & **Boost**
- Fast growing community (~1K stars on github, active ML)
- Presented CAF twice at C++Now
  - Feedback resulted in **type-safe actors**
- Production-grade code: extensive unit tests, comprehensive CI

Summary

- Actor model is a natural fit for today's systems
- CAF offers an efficient C++ runtime
  - High-level message passing abstraction
  - Type-safe messaging APIs *at compile time*
  - Network-transparent communication
  - Well-defined failure semantics

Questions?

http://actor-framework.org

https://github.com/actor-framework