C++ Actor Framework

Transparent Scaling from IoT to Datacenter Apps

Matthias Vallentin

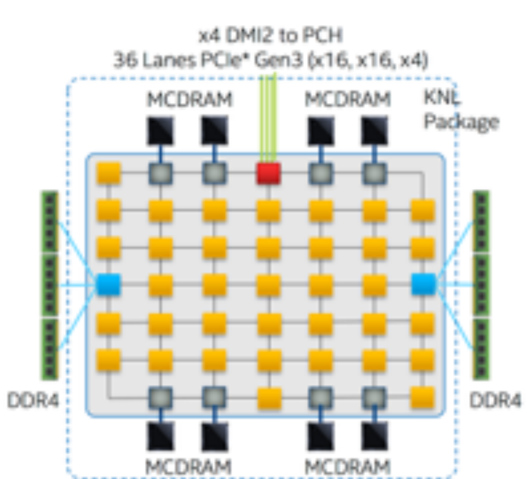
UC Berkeley

RISElab seminar

November 21, 2016

Heterogeneity

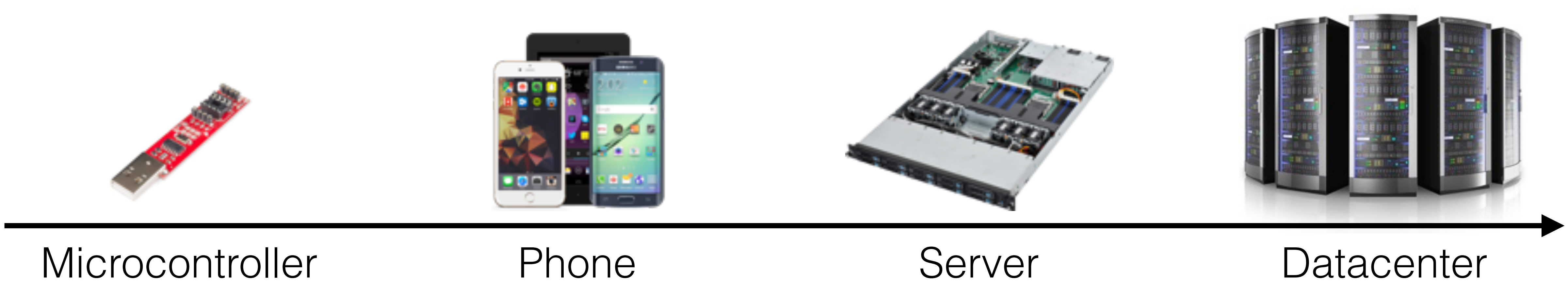
- More cores on desktops and mobile
- Complex accelerators/co-processors
- Highly distributed deployments
- Resource-constrained devices





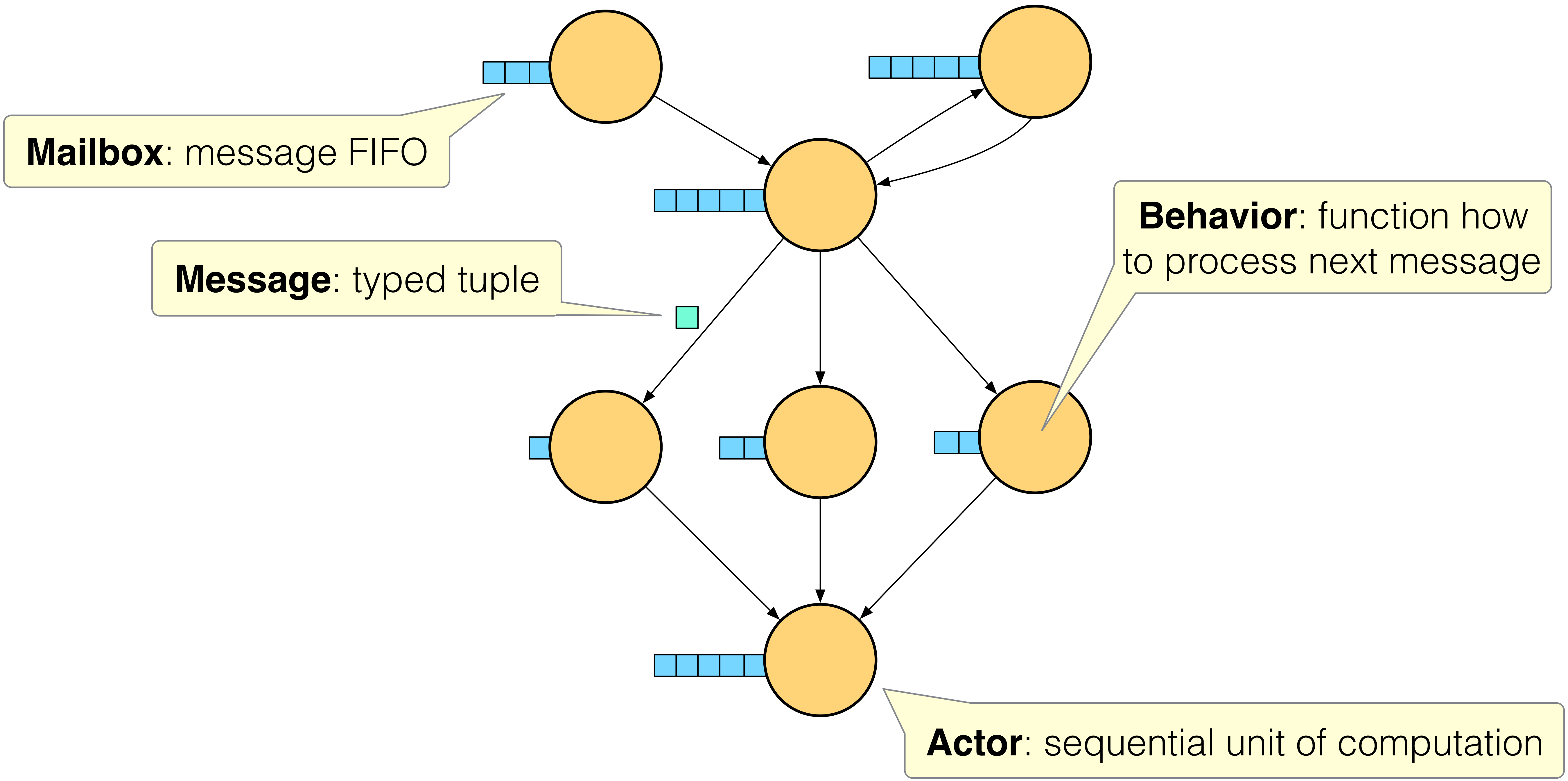
Scalable Abstractions

- **Uniform API** for concurrency and distribution
- **Compose** small components into large systems
- **Scale** runtime from IoT to HPC



Actor Model

The Actor Model



Actor Semantics

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

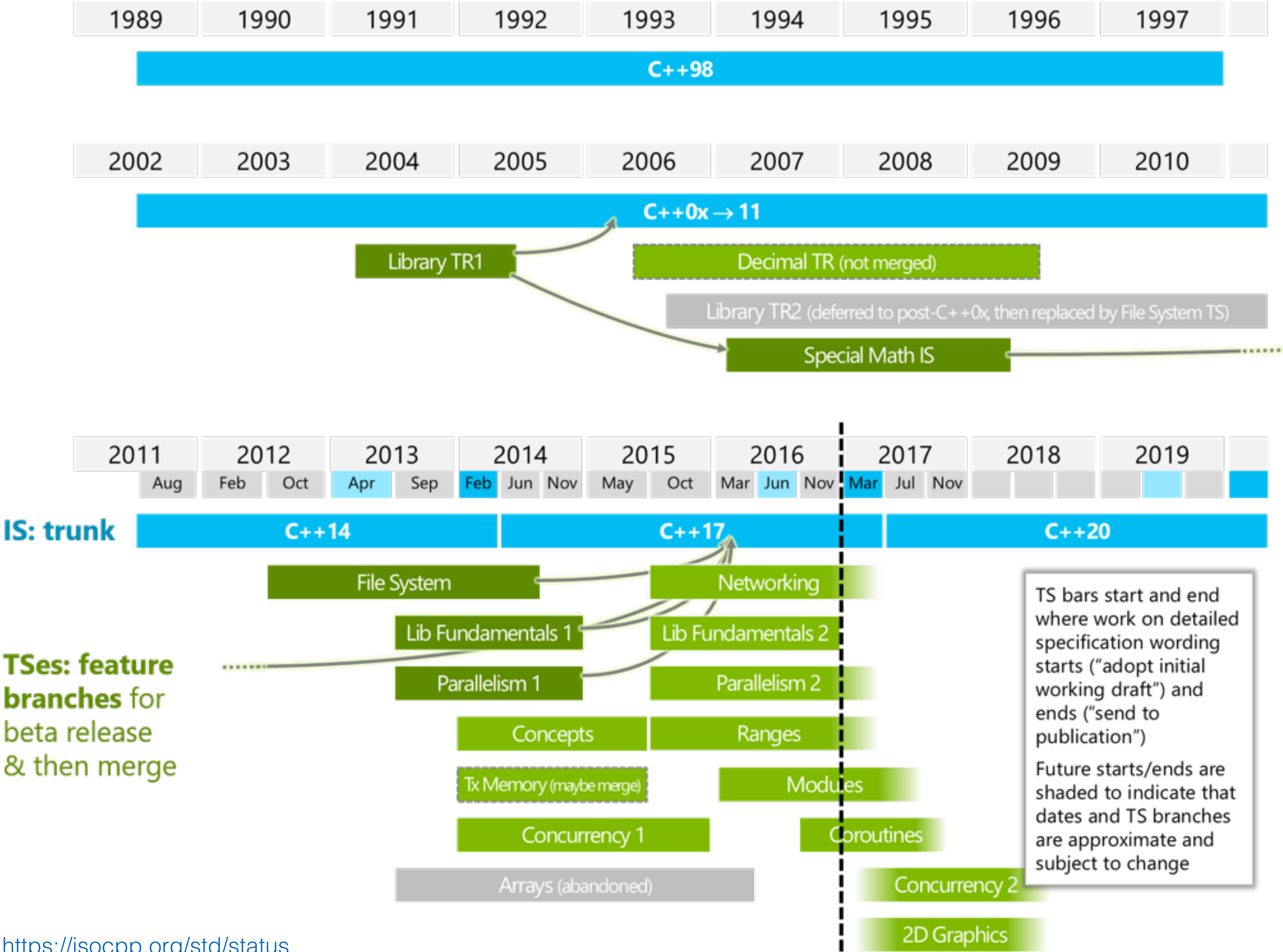
C++ Actor Framework
(CAF)

Why C++

High degree of abstraction

without

sacrificing performance



https://isocpp.org/std/status

CAF

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 #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”
    }
  );
}
```

Blocking

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.

Designate how to handle next message.
(= set the actor behavior)

Example #3

```
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();
      }
    };
  }
);
```

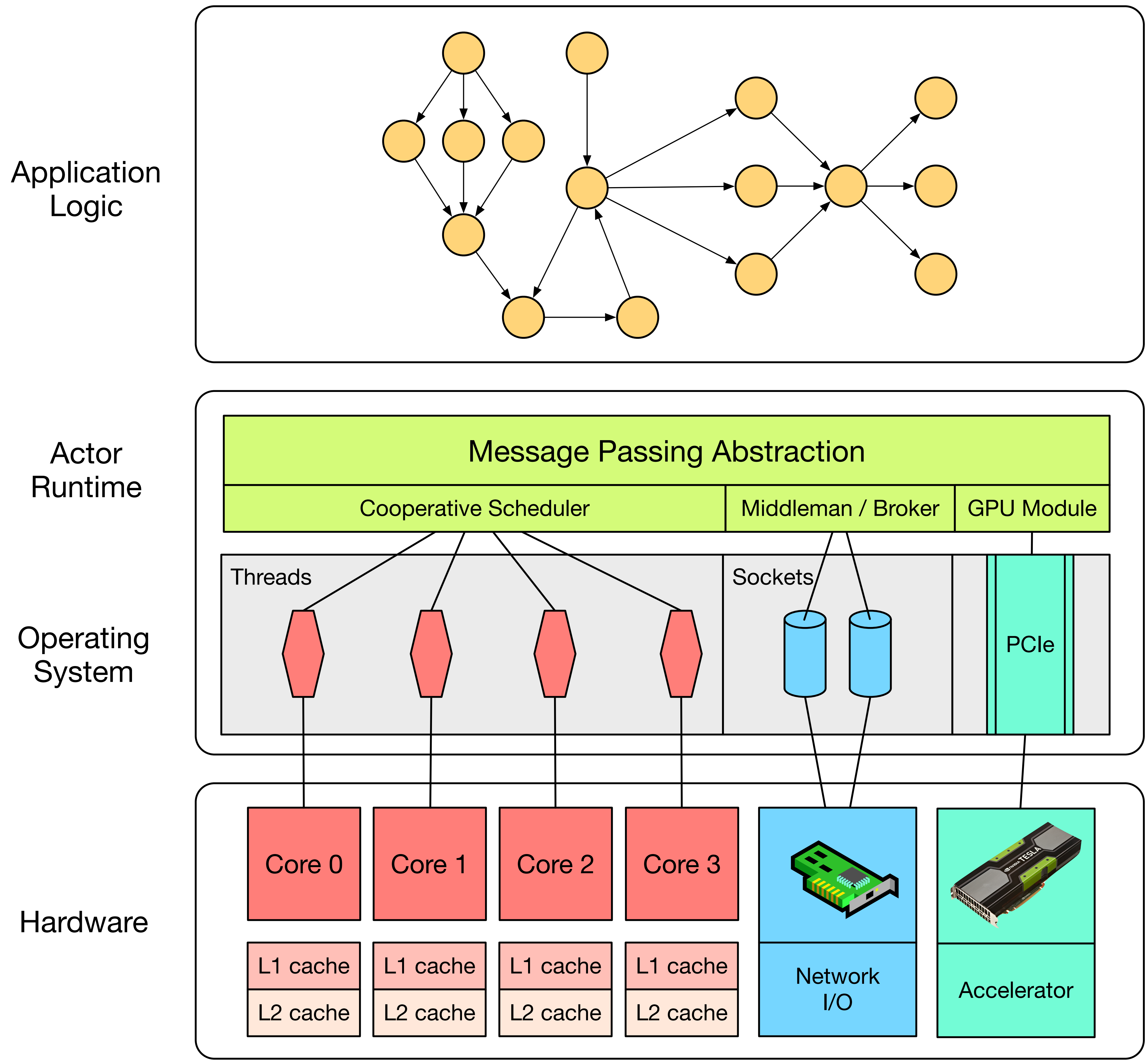
Non-Blocking

Example #4

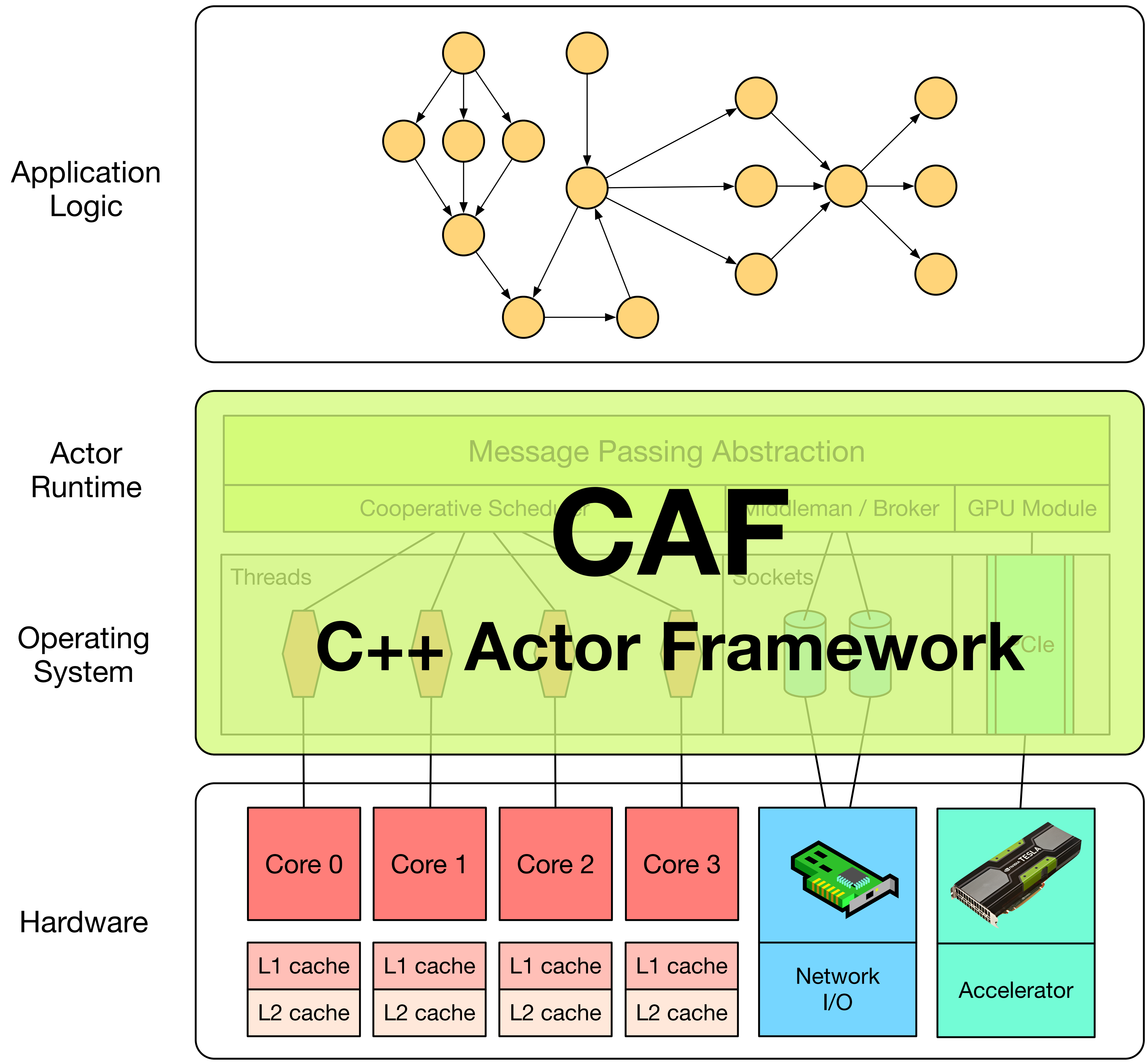
Request-response communication requires timeout.
(`std::chrono::duration`)

```
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`.



Application Logic
Actor Runtime
Operating System
Hardware
Message Passing Abstraction
Cooperative Scheduler
Middleman / Broker
GPU Module
Threads
Sockets
PCIe
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
Accelerator

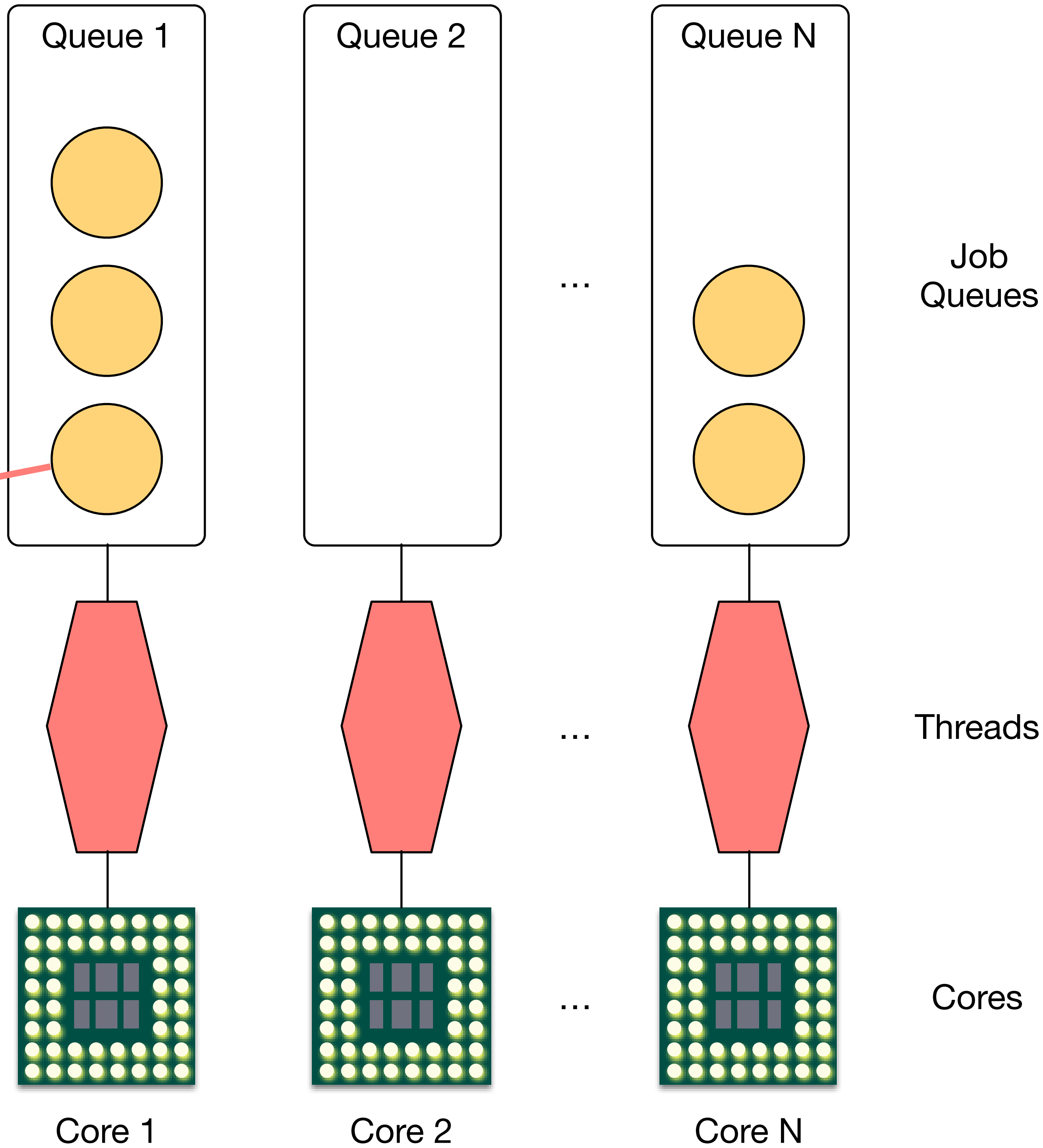

Application Logic
Actor Runtime
Operating System
Hardware
Message Passing Abstraction
Cooperative Scheduler
Middleman / Broker
GPU Module
Threads
Sockets
PCIe
CAF
C++ Actor Framework
Core 0
Core 1
Core 2
Core 3
L1 cache
L2 cache
L1 cache
L2 cache
L1 cache
L2 cache
L1 cache
L2 cache
Network I/O
Accelerator

Scheduler

Work Stealing*

- **Decentralized**: one job queue and worker thread per core

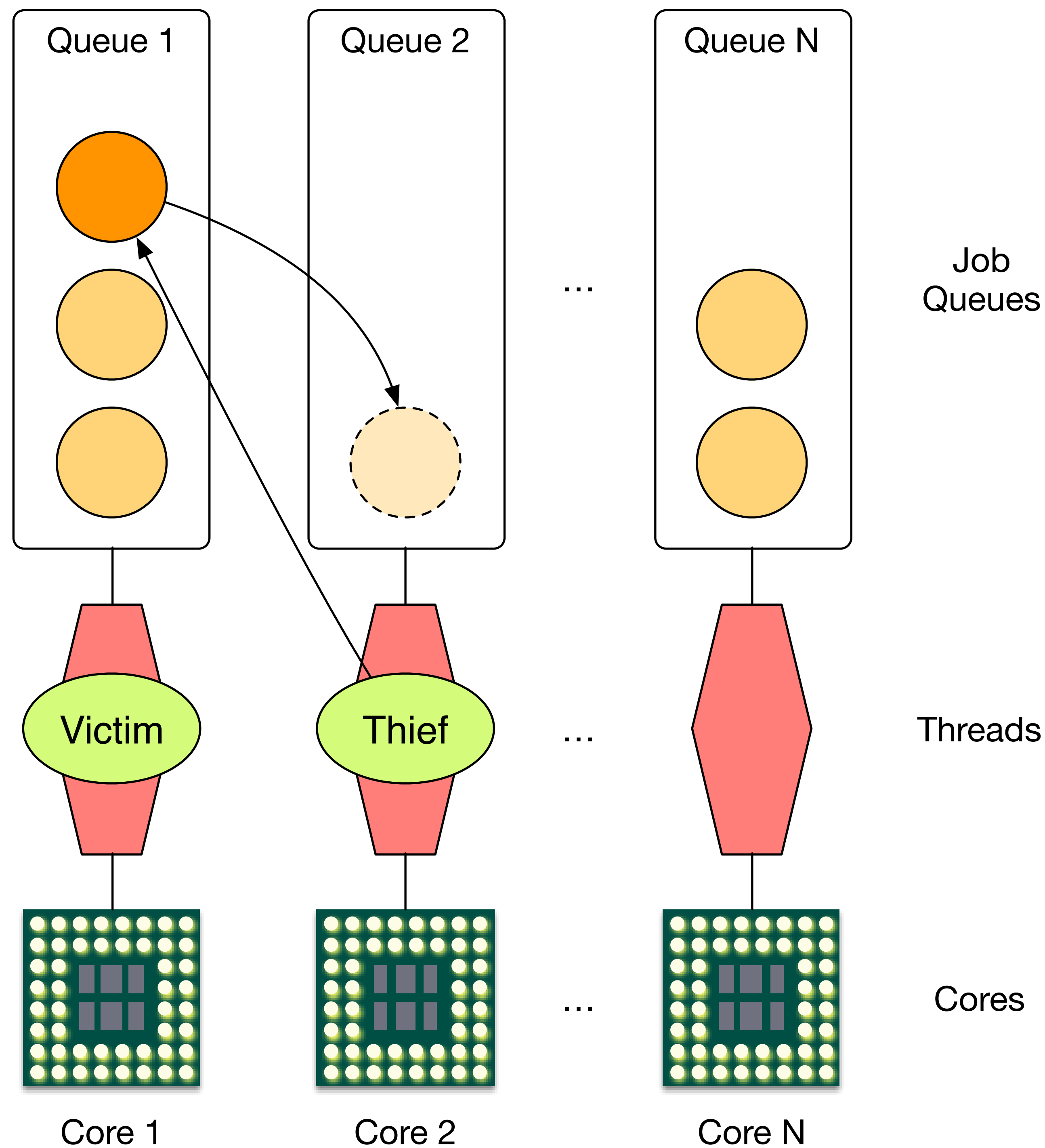
```
behavior adder() {
  return {
    [](int x, int y) {
      return x + y;
    },
    ...
```



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

Work Stealing*

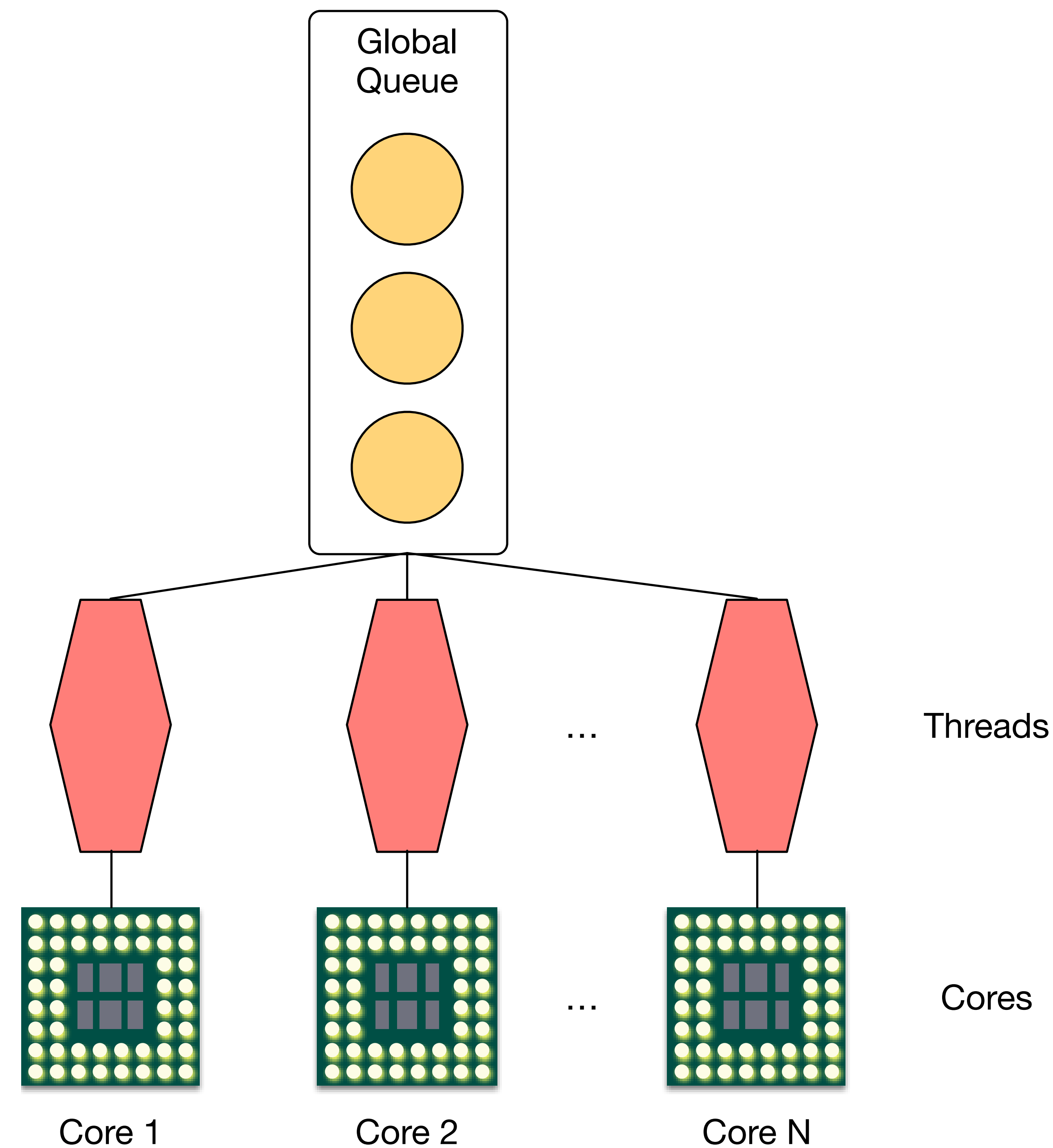
- **Decentralized**: one job queue and worker thread per core
- On empty queue, **steal** from other thread
- Efficient if stealing is a rare event
- Implementation: deque with two spinlocks



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

Work Sharing

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



Copy-On-Write

- **`caf::message`** = intrusive, ref-counted typed tuple
  - **Immutable access** permitted
  - **Mutable access** with ref count > 1 invokes copy constructor
- **Constness deduced** from message handlers

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

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

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

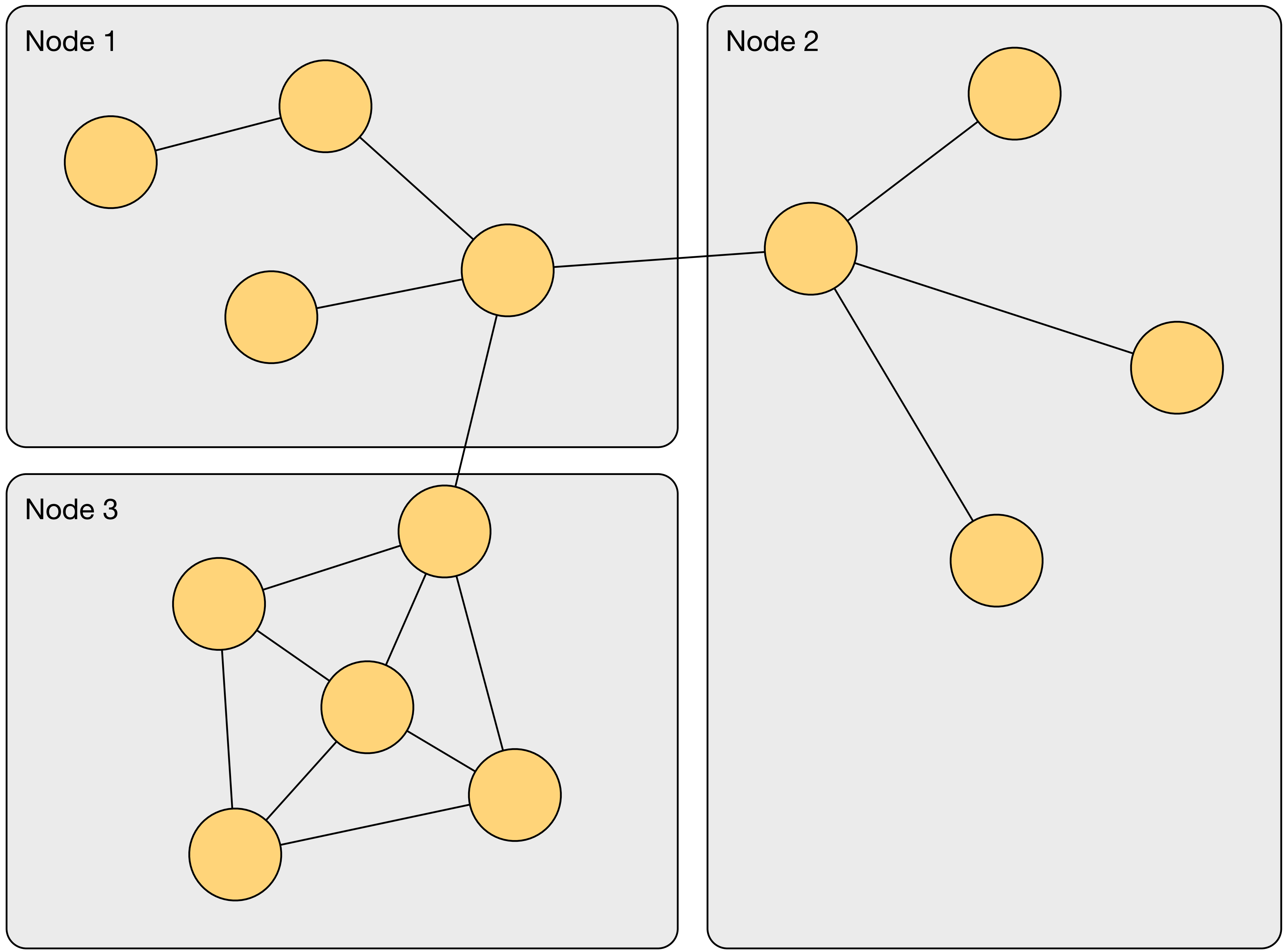
`const` access enables efficient sharing of messages

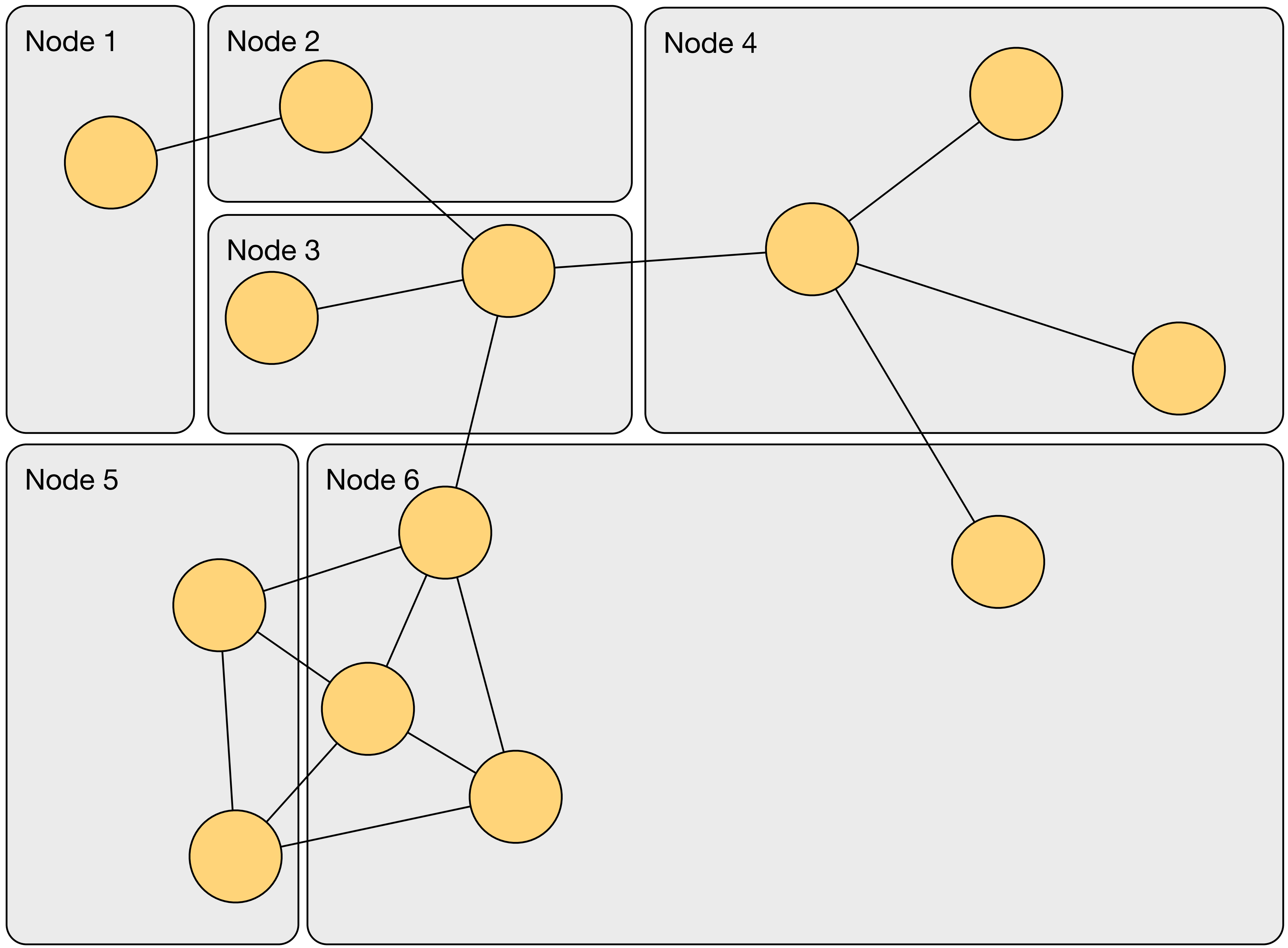
non-`const` access copies message contents **if ref count > 1**

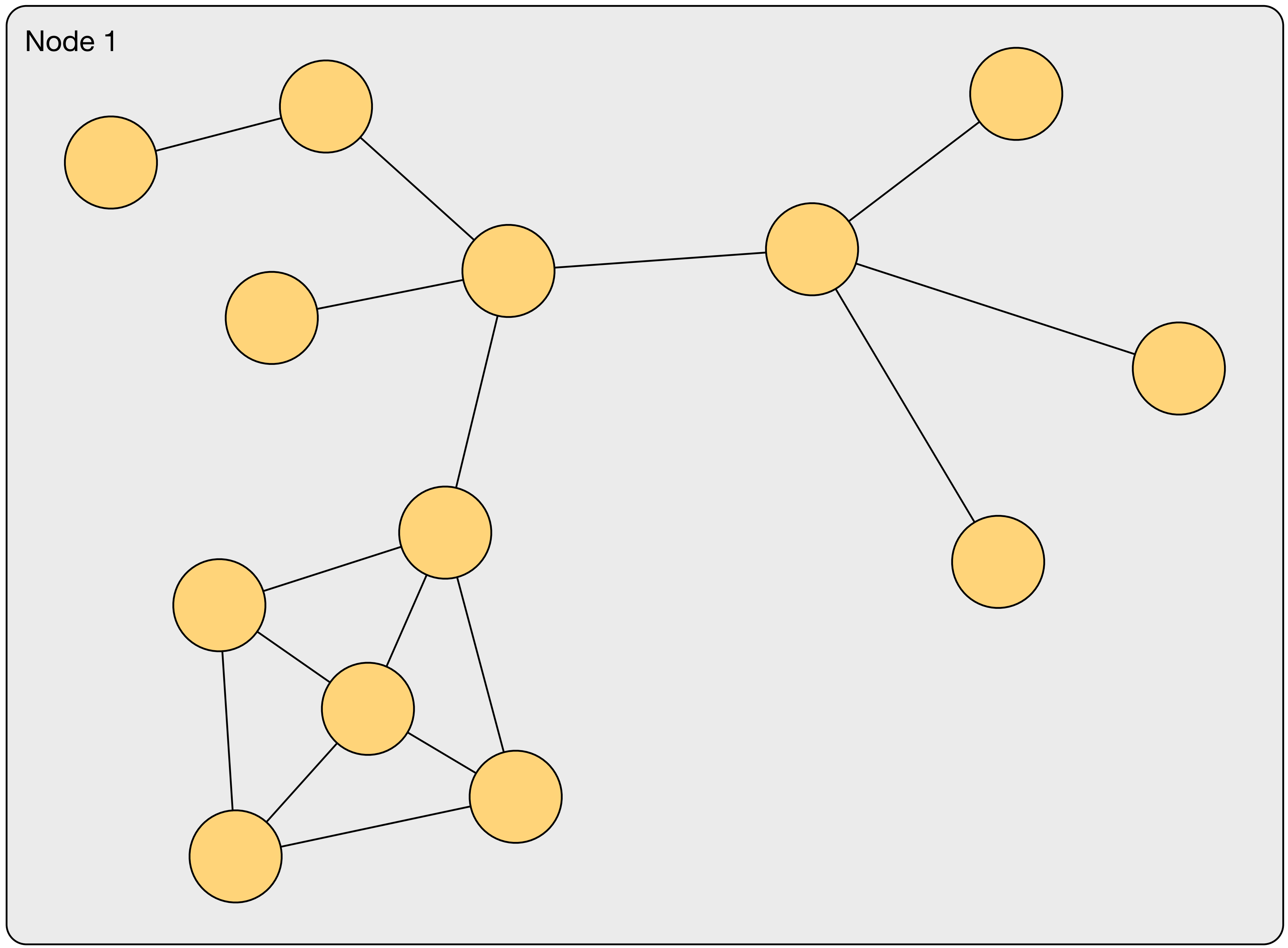
- **`caf::message`** = intrusive, ref-counted typed tuple
  - **Immutable access** permitted
  - **Mutable access** with ref count > 1 invokes copy constructor
- **Constness deduced** from message handlers
- **No data races** by design
- **Value semantics**, no complex lifetime management

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

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

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

Network Transparency


Node 1
Node 2
Node 3


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


Node 1

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

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

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.

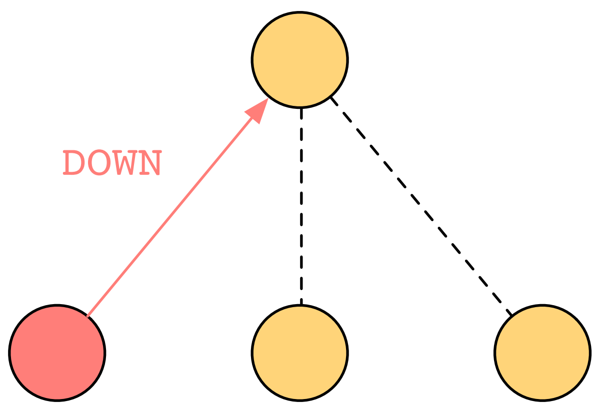
Connect to published actor at TCP endpoint.
Returns `expected<actor>`.

Failures

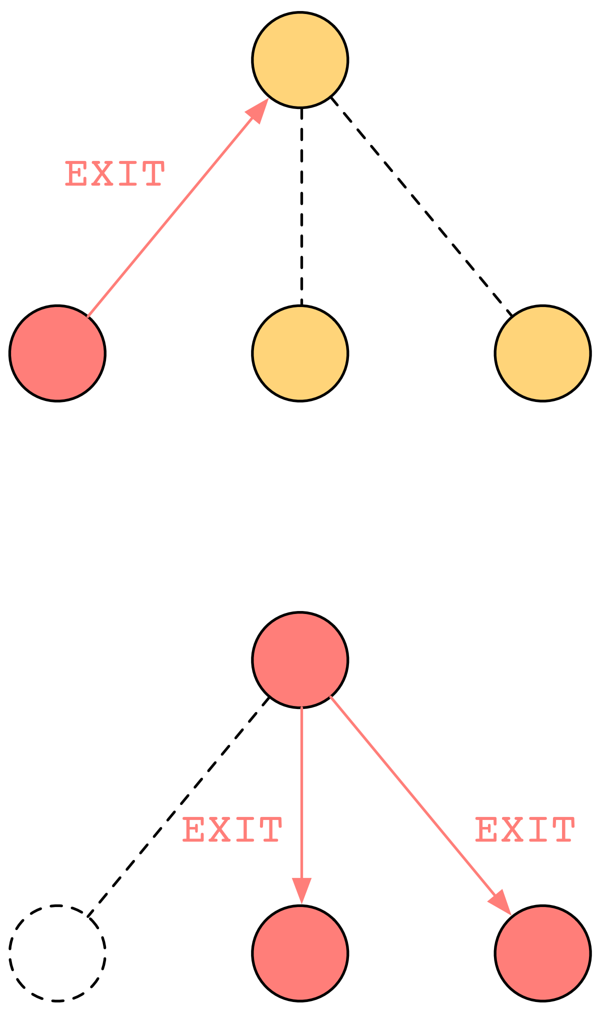
Components fail regularly in large-scale systems

- Actor model provides **monitors** and **links**
  - **Monitor**: subscribe to exit of actor (**unidirectional**)
  - **Link**: bind own lifetime to other actor (**bidirectional**)
- No side effects (unlike exception propagation)
- Explicit error control via message passing

Monitors



Links



Monitor Example

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

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

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

Link Example

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

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

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

Evaluation

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

Benchmark #1:
Actors vs. Threads

Matrix Multiplication

- Example for scaling computation
- Large number of independent tasks
- Can use C++11's **`std::async`**
- Simple to port to GPU

Matrix Class

```
static constexpr size_t matrix_size = /*...*/;

// square matrix: rows == columns == matrix_size
class matrix {
public:
  float& operator()(size_t row, size_t column);
  const vector <float>& data() const;
  // ...
private:
  vector <float> data_;
};
```

Simple Loop

```
matrix simple_multiply(const matrix& lhs,
                       const matrix& rhs) {
  matrix result;
  for (size_t r = 0; r < matrix_size; ++r)
    for (size_t c = 0; c < matrix_size; ++c)
      result(r, c) = dot_product(lhs, rhs, r, c);
  return result;
}
```

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

std::async

```
matrix async_multiply(const matrix& lhs,
                      const matrix& rhs) {
  matrix result;
  vector<future<void>> futures;
  futures.reserve(matrix_size * matrix_size);
    for (size_t r = 0; r < matrix_size; ++r)
      for (size_t c = 0; c < matrix_size; ++c)
        futures.push_back(async(launch::async, [&,r,c] {
          result(r, c) = dot_product(lhs, rhs, r, c);
        }));
  for (auto& f : futures)
    f.wait();
  return result;
}
```

Actors

```
matrix actor_multiply(const matrix& lhs,
                      const matrix& rhs) {
  matrix result;
  actor_system_config cfg;
  actor_system sys{cfg};
  for (size_t r = 0; r < matrix_size; ++r)
    for (size_t c = 0; c < matrix_size; ++c)
      sys.spawn([&,r,c] {
        result(r, c) = dot_product(lhs, rhs, r, c);
      });
  return result;
}
```

OpenCL Actors

```
static constexpr const char* source = R"__(
  __kernel void multiply(__global float* lhs,
                         __global float* rhs,
                         __global float* result) {
    size_t size = get_global_size(0);
    size_t r = get_global_id(0);
    size_t c = get_global_id(1);
    float dot_product = 0;
    for (size_t k = 0; k < size; ++k)
      dot_product += lhs[k+c*size] * rhs[r+k*size];
    result[r+c*size] = dot_product;
  }
)__";
```

OpenCL Actors

```
matrix opencl_multiply(const matrix& lhs,
                       const matrix& rhs) {
 auto worker = spawn_cl<float* (float* ,float*)>(
    source, "multiply", {matrix_size , matrix_size});
  actor_system_config cfg;
  actor_system sys{cfg};
  scoped_actor self{sys};
  self->send(worker, lhs.data(), rhs.data());
  matrix result;
  self->receive([&](vector<float>& xs) {
    result = move(xs);
  });
  return result;
}
```

Results

Setup: 12 cores, Linux, GCC 4.8, 1000 x 1000 matrices

```
time ./simple_multiply
```
0m9.029s

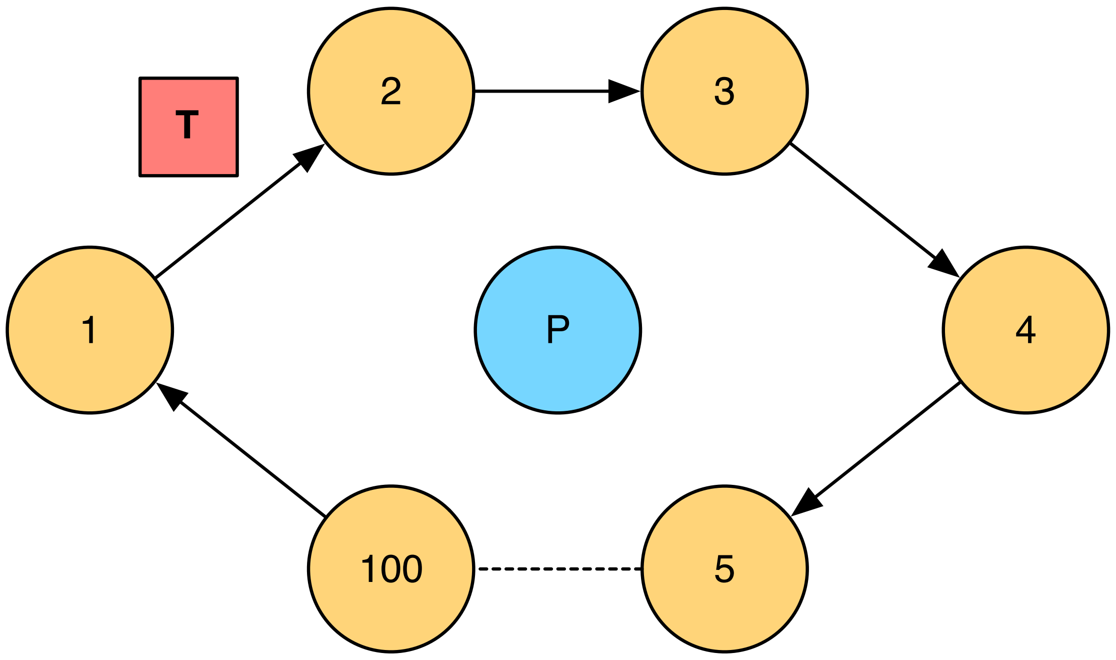
```
time ./actor_multiply
```
0m1.164s

```
time ./opencl_multiply
```
0m0.288s

```
time ./async_multiply
terminate called after throwing an instance of 'std::system_error'
  what():  Resource temporarily unavailable
```

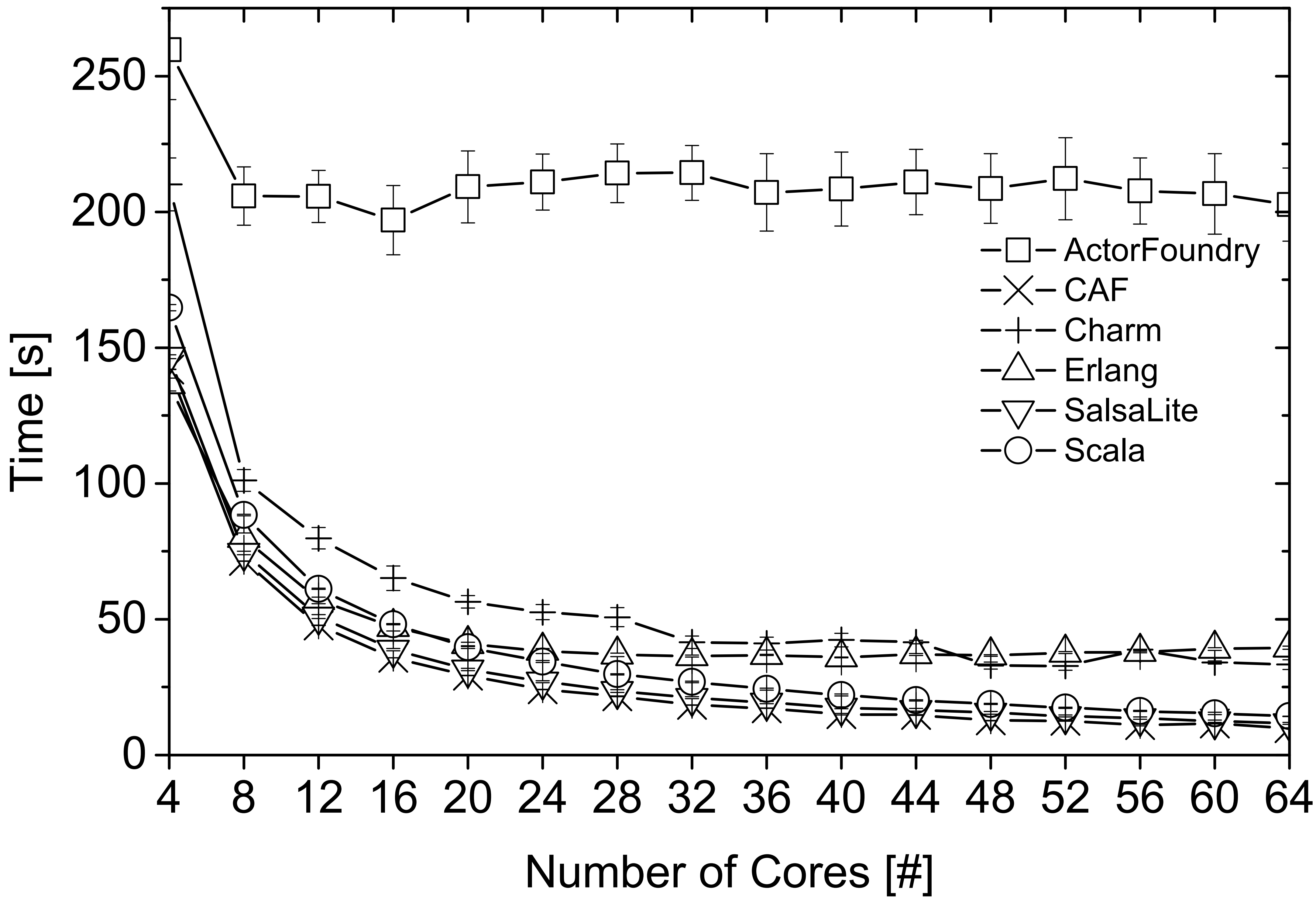
Benchmark #1

Setup #1

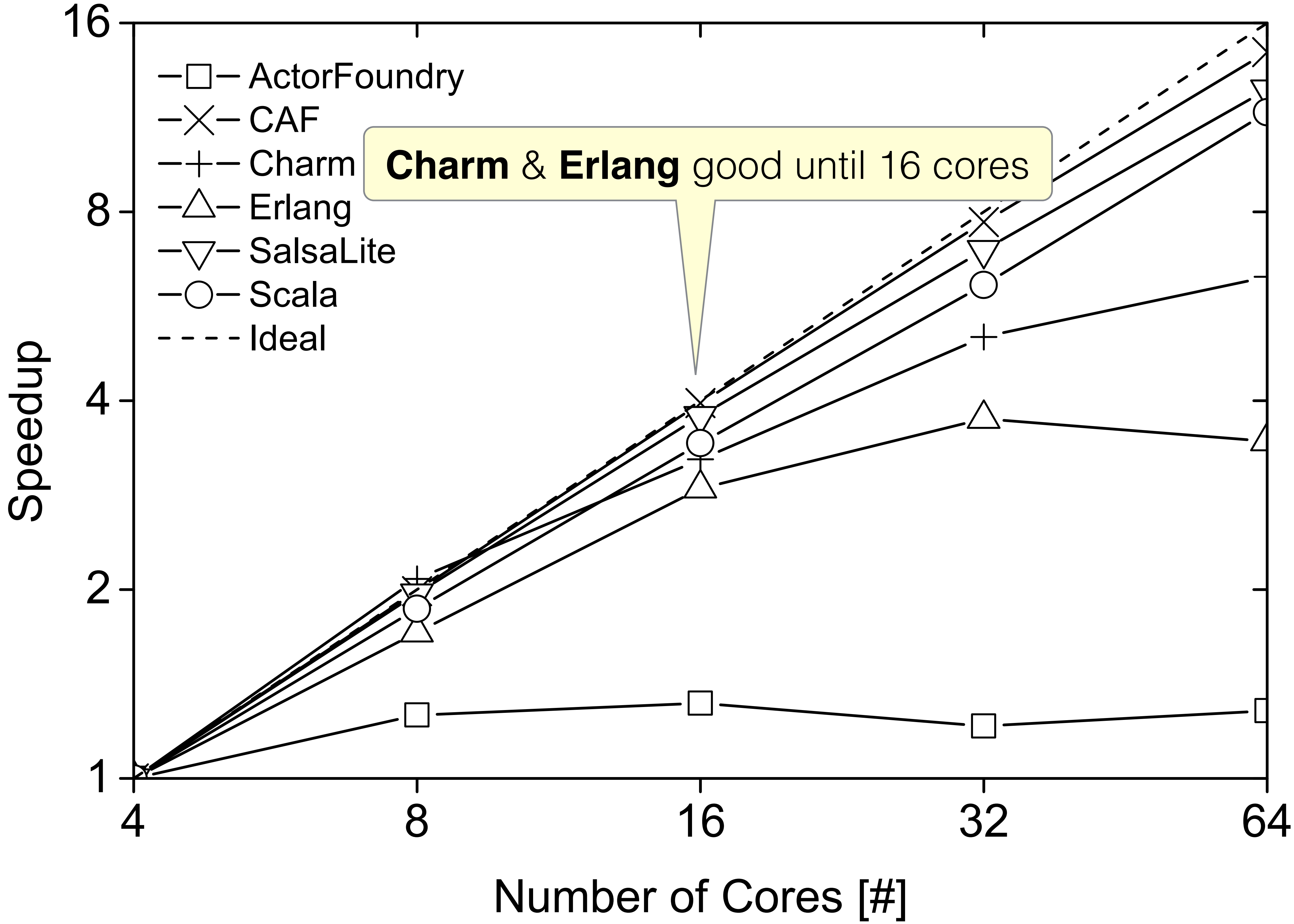


- 100 rings of 100 actors each
- Actors forward single token 1K times, then terminate
- 4 re-creations per ring
- One actor per ring performs *prime factorization*
- Resulting workload: **high message & CPU pressure**
- Ideal: 2 x cores ⟹ 0.5 x runtime

Performance

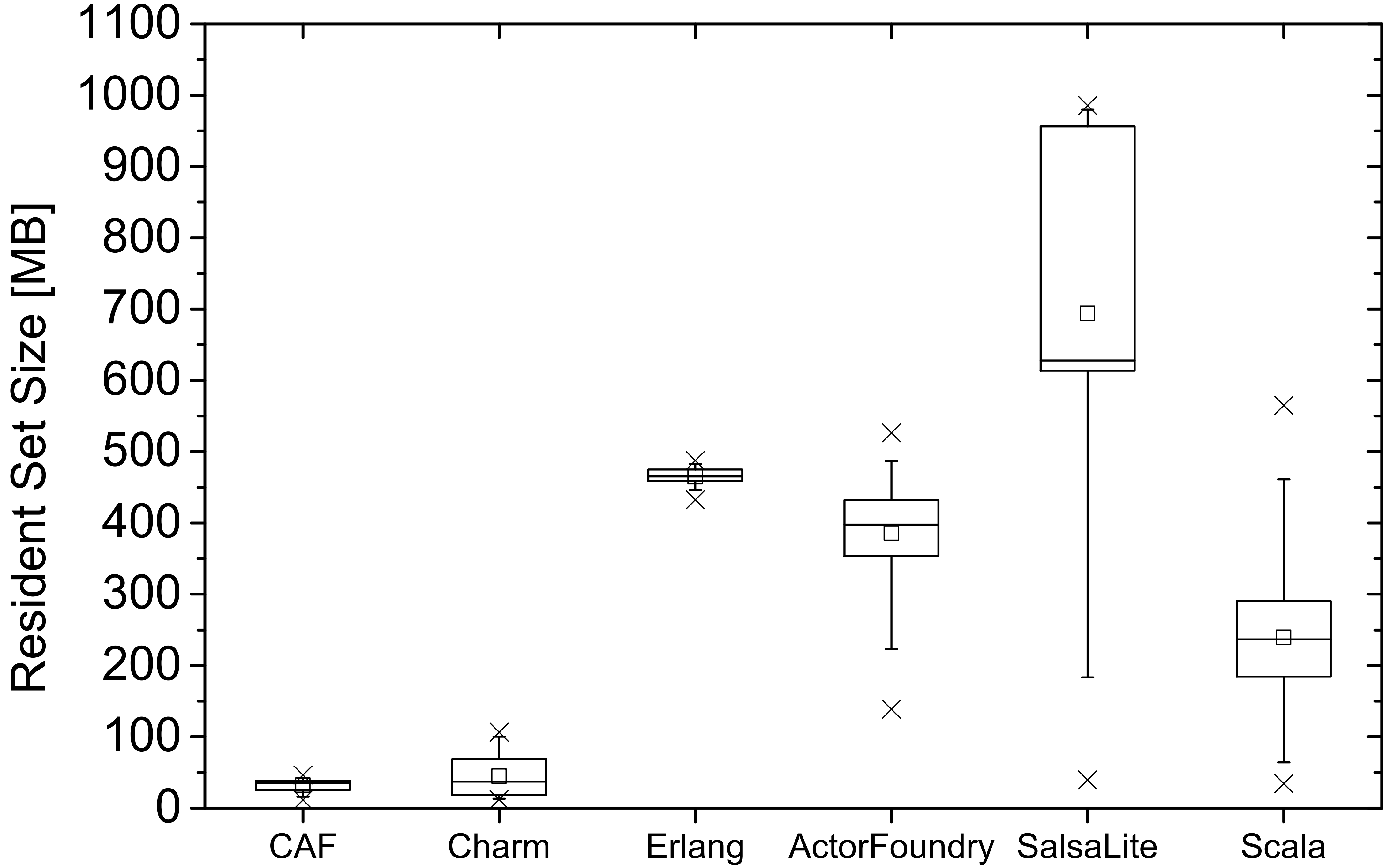


(normalized)



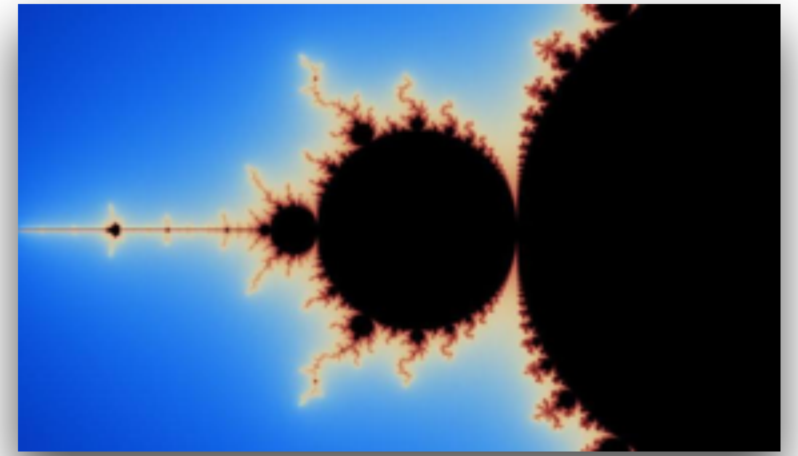
16
8
4
2
1
Speedup
ActorFoundry
CAF
Charm
Erlang
SalsaLite
Scala
Ideal
Charm & **Erlang** good until 16 cores
4
8
16
32
64
Number of Cores [#]

Memory Overhead



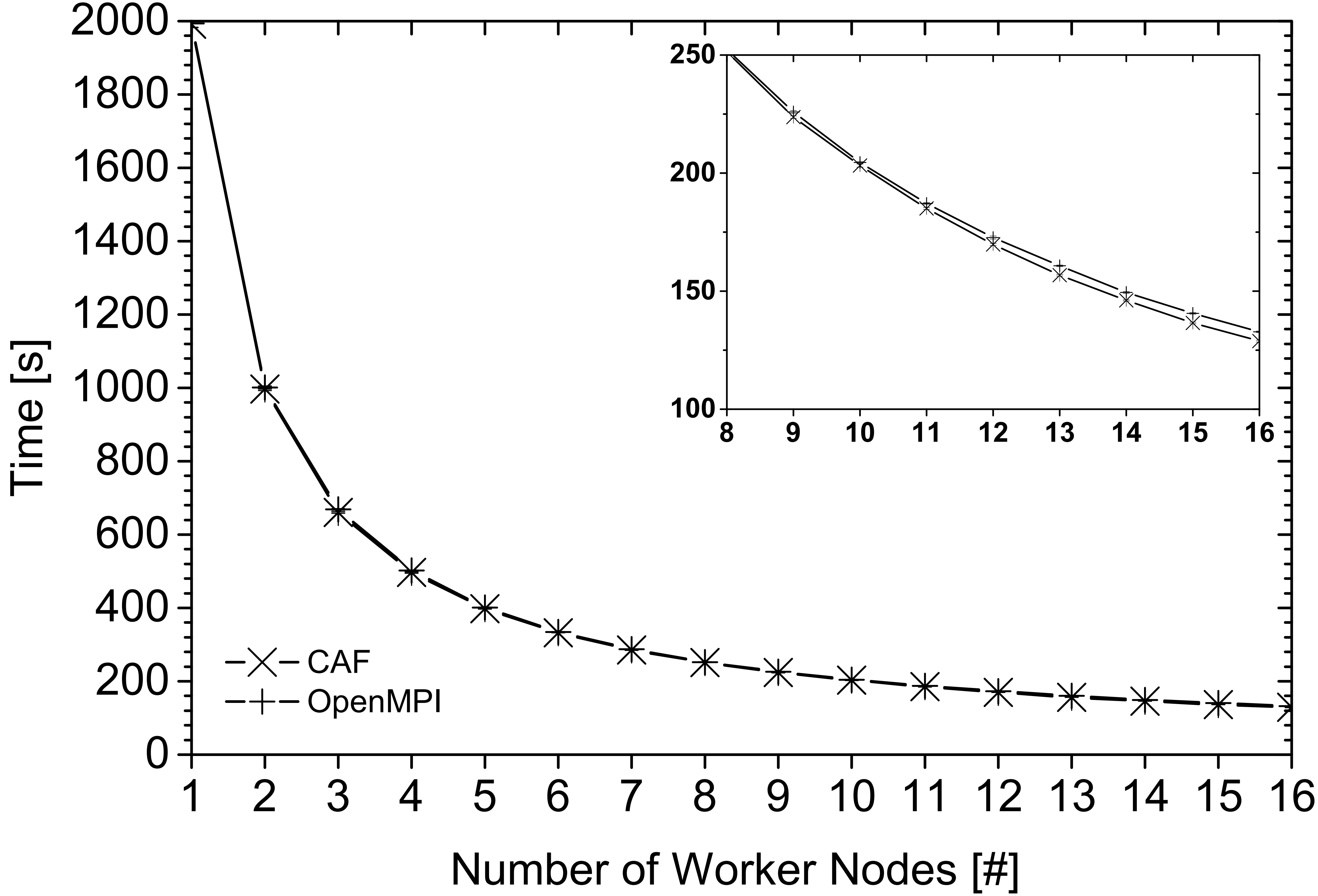
Benchmark #2

CAF vs. MPI



- 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

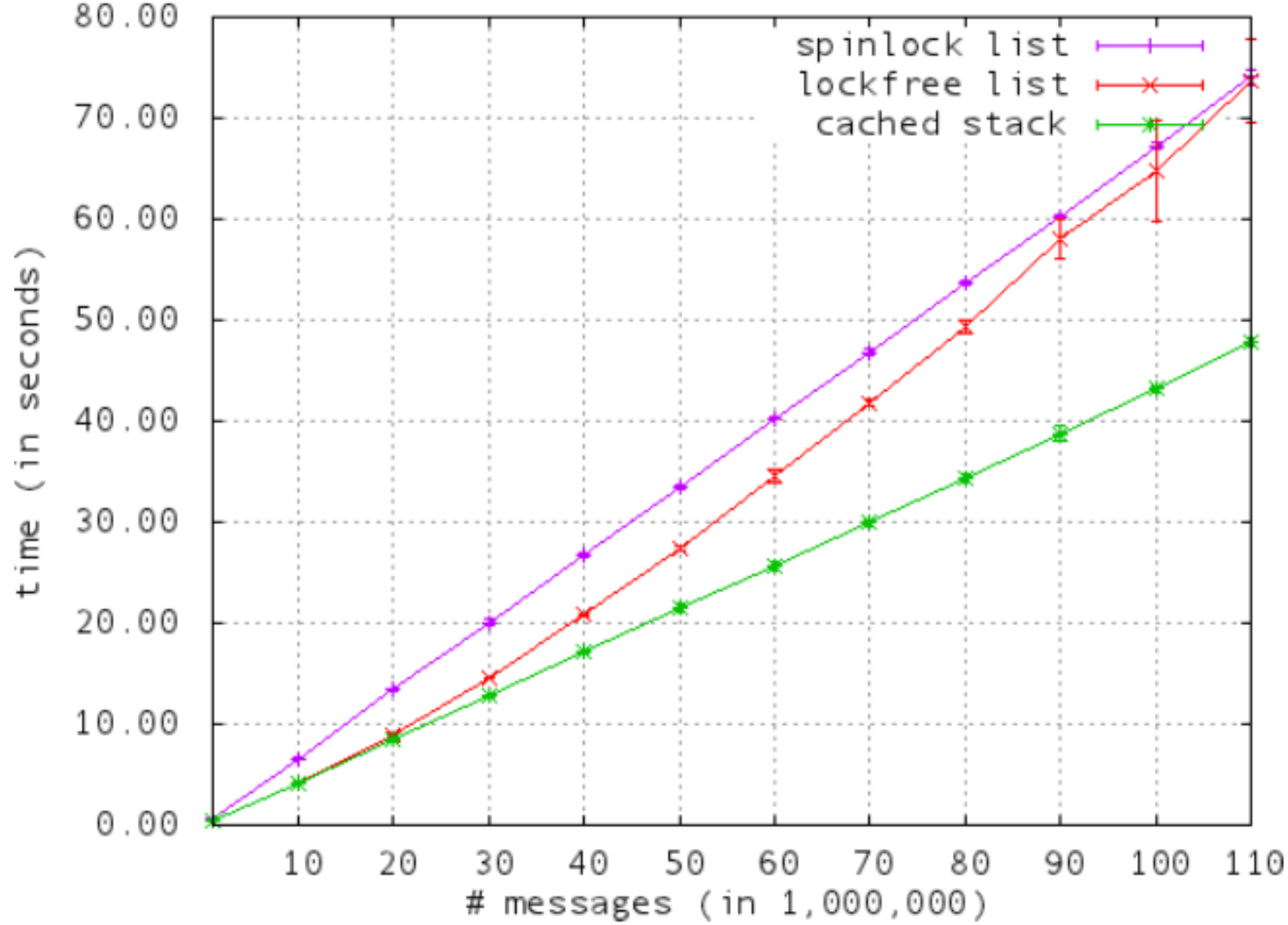


Benchmark #3

Mailbox Performance

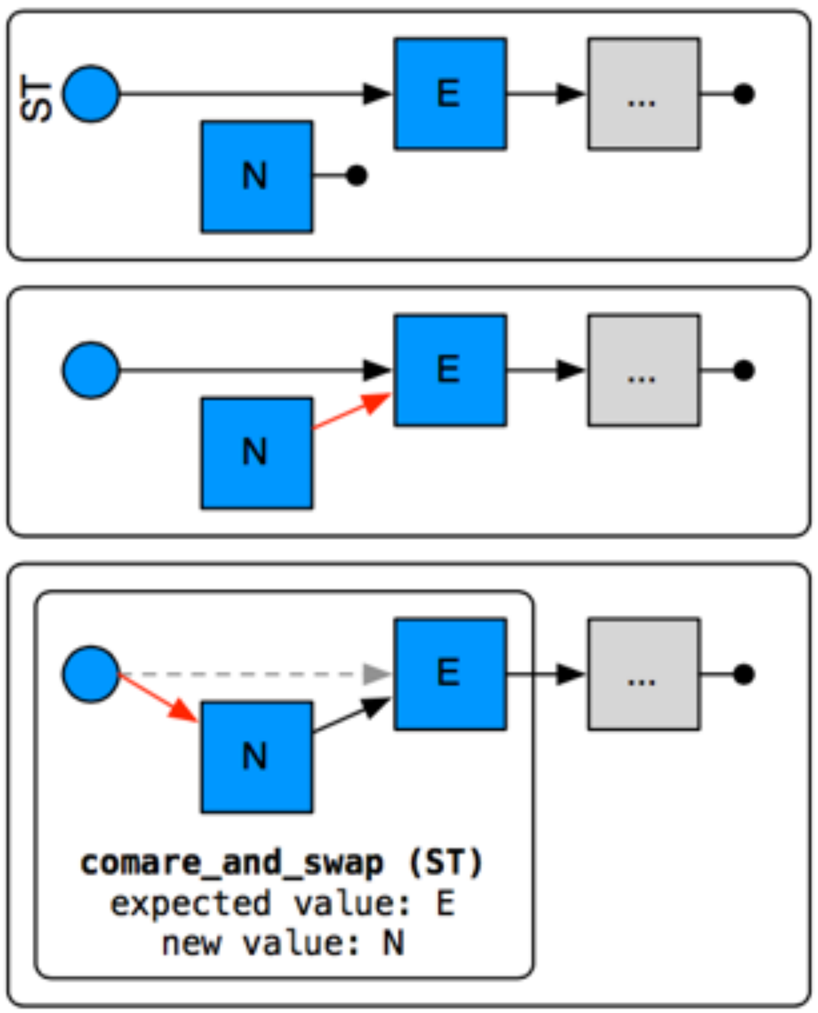
- Mailbox implementation is critical to performance
- **Single-reader-many-writer** queue
- Test only queues with atomic CAS operations

  1. Spinlock queue
  2. Lock-free queue
  3. **Cached stack**



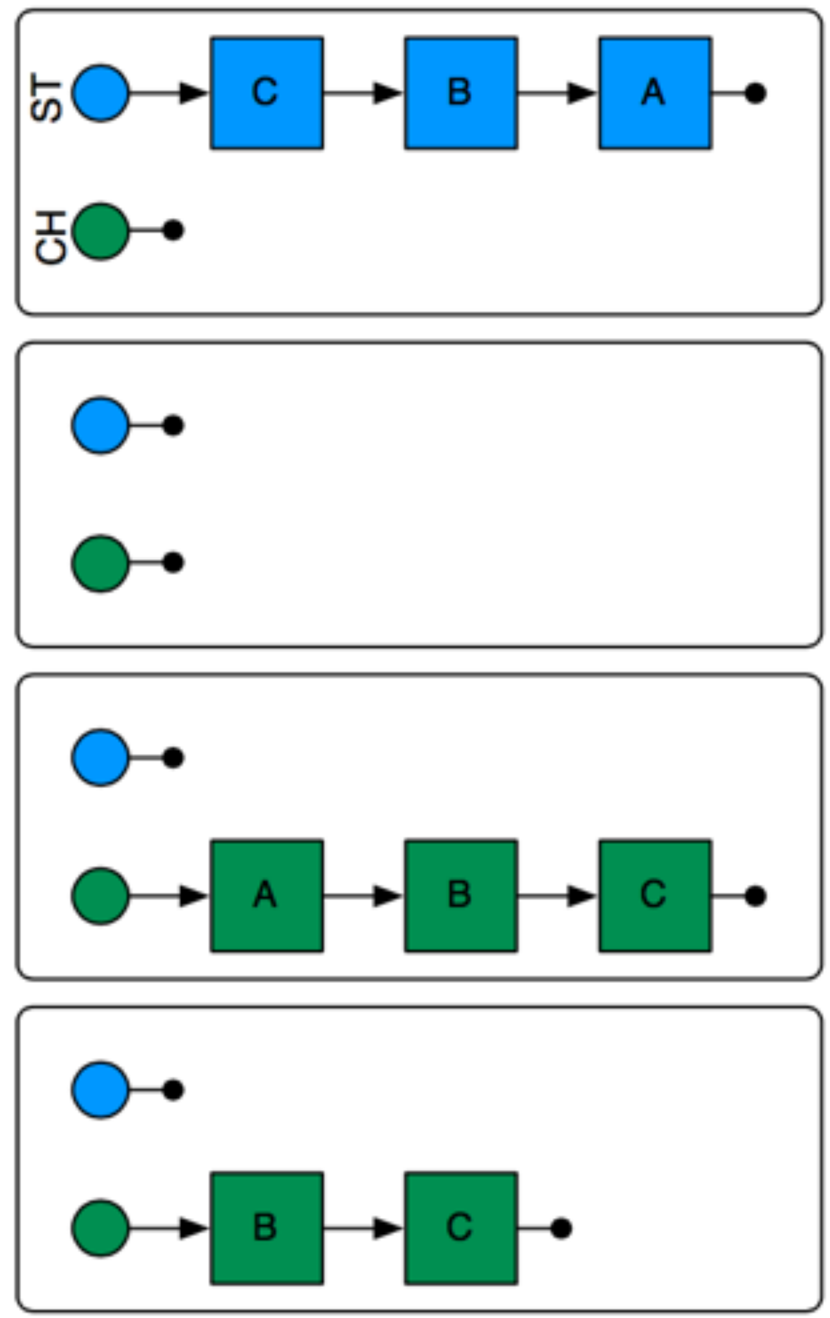
spinlock list
lockfree list
cached stack
time (in seconds)
messages (in 1,000,000)
80.00
70.00
60.00
50.00
40.00
30.00
20.00
10.00
0.00
10
20
30
40
50
60
70
80
90
100
110

Cached Stack




```
enqueue(N)
{


  T = tail
  N.next = T


  if not cas(&tail,
             T,N)   {
    enqueue(N)
  }

}
```

Figure 27: Enqueue operation in a cached stack



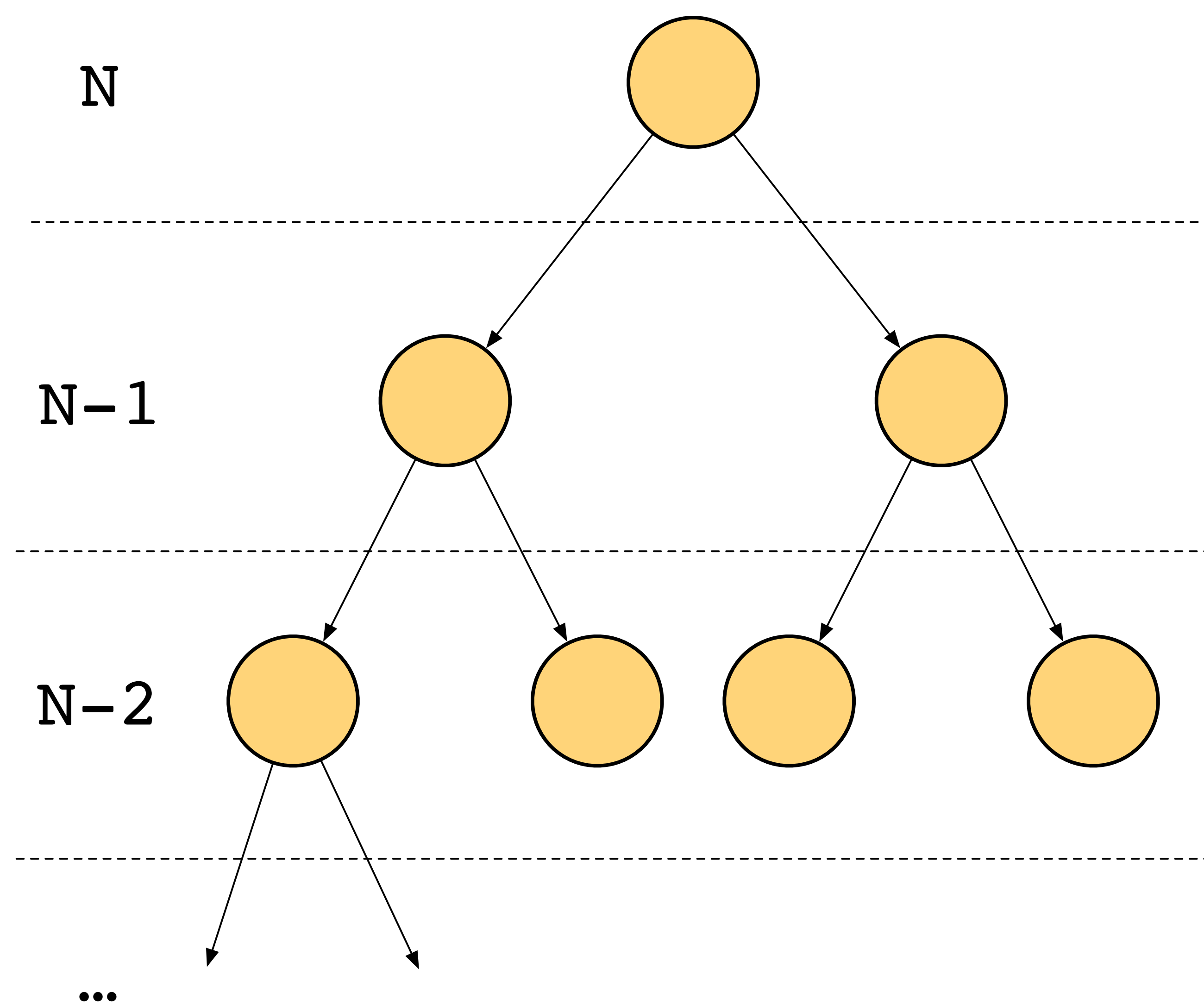

```
dequeue()
{
  R = CH
  if R != NULL {
    CH = R.next
    return R
  }

  do {
    E = ST
    if E == NULL
      return NULL
  } while not cas
       (&ST,E,NULL)

  while E != NULL {
    NEXT = E.next
    E.next = CH
    CH = E
    E = NEXT
  }


  return dequeue()

}
```

Figure 28: Dequeue operation in a cached stack

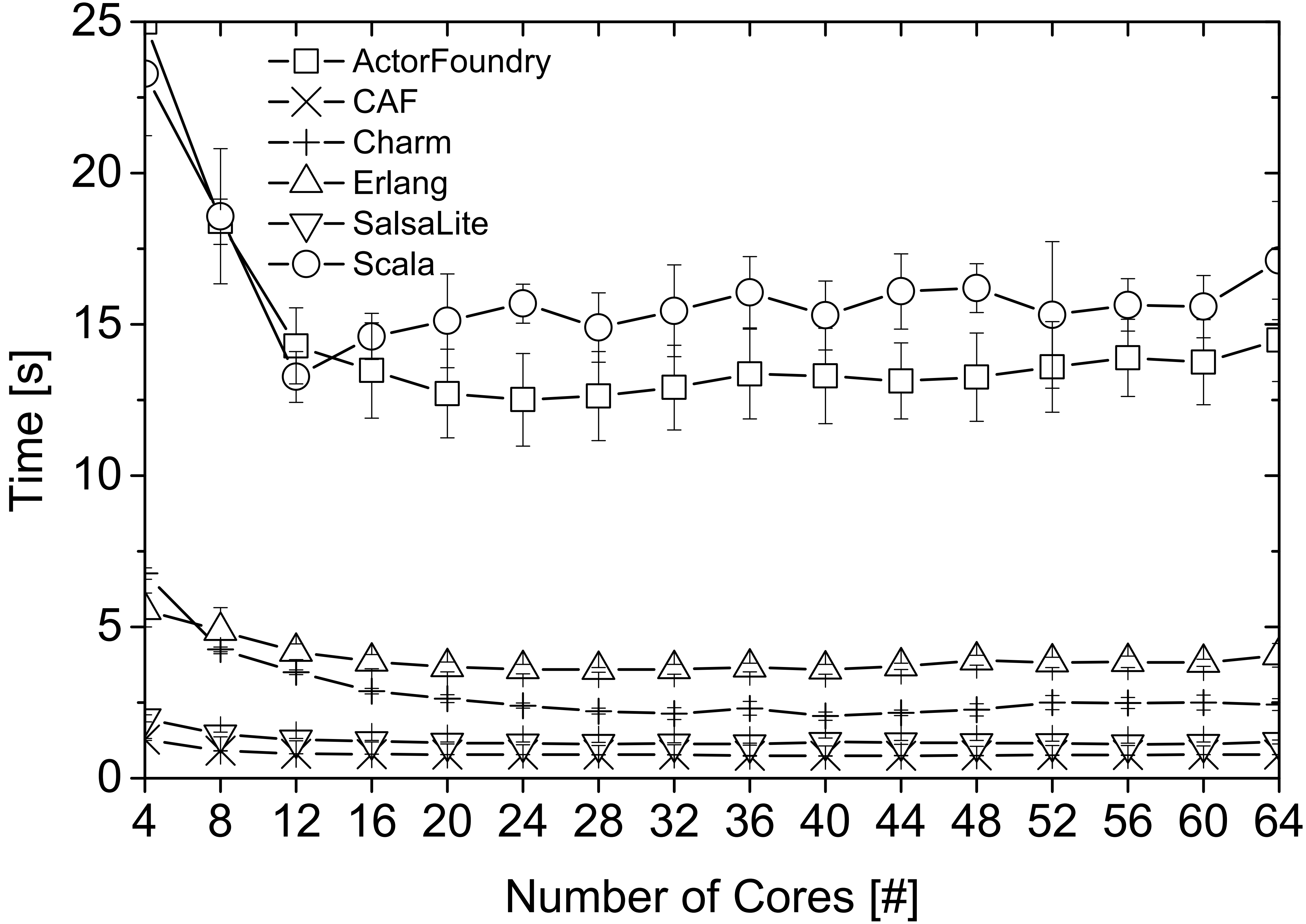
Benchmark #4

Actor Creation

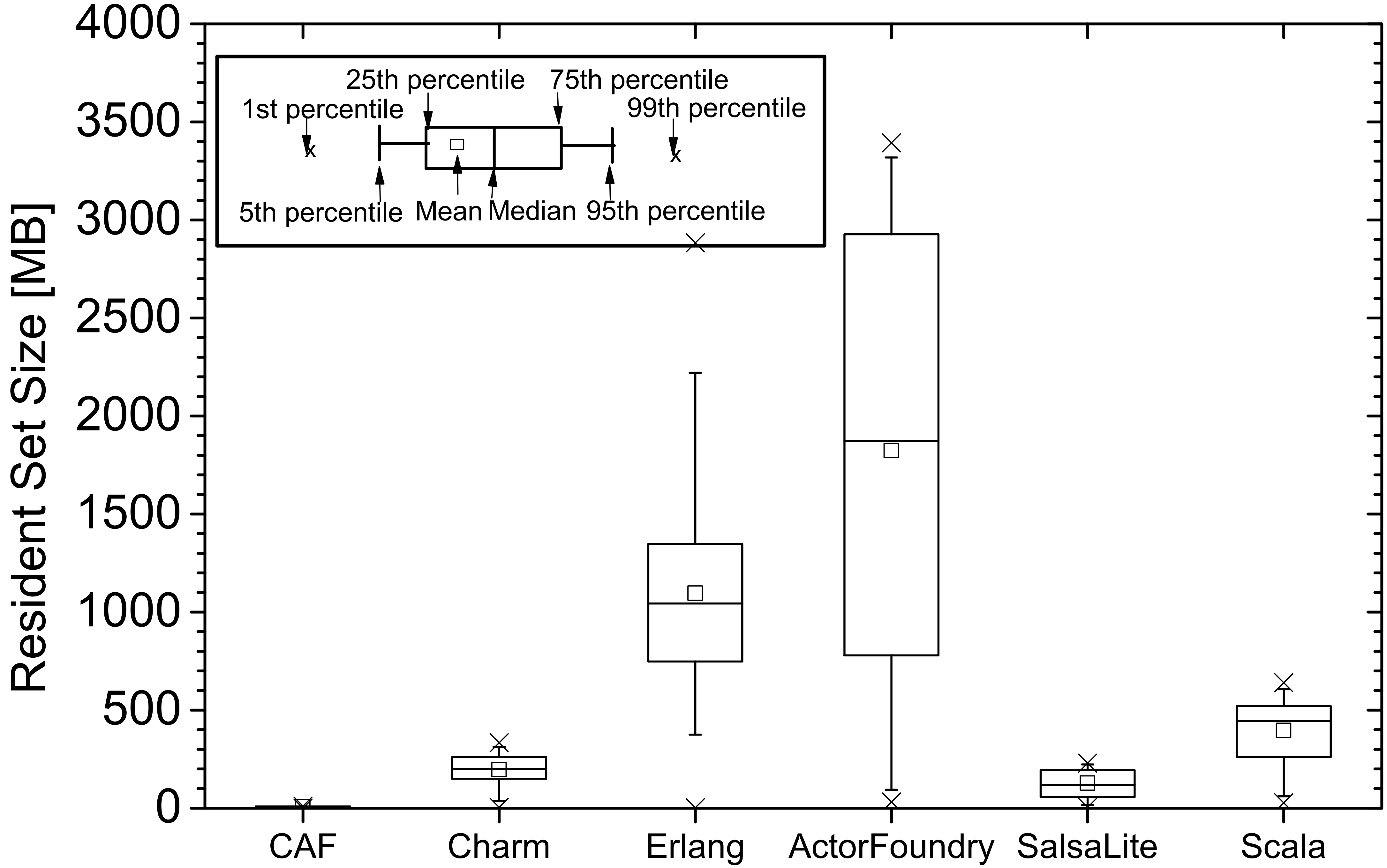
- Compute 2^{20} by **recursively spawning actors**
- Behavior: at step *N*, spawn 2 actors of recursion counter *N-1,* and wait for their results
- **Over 1M actors** created



Actor Creation



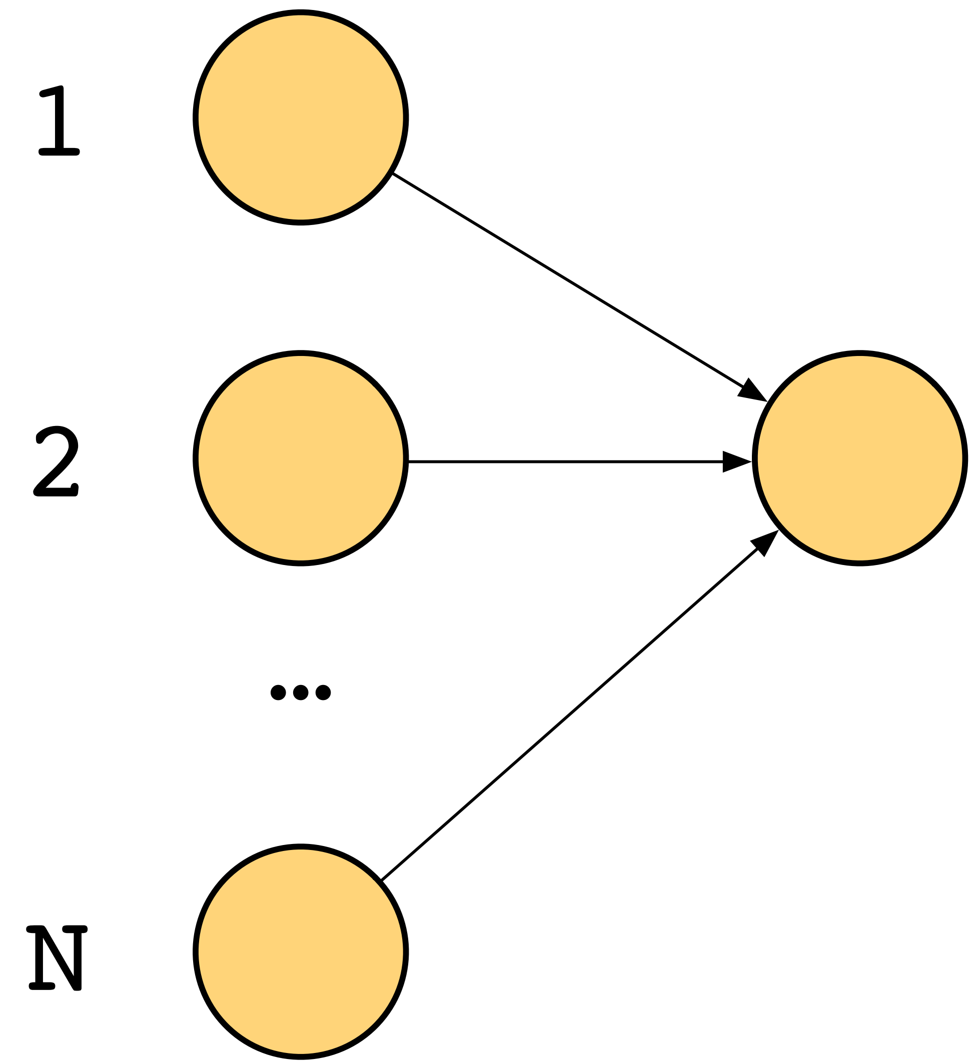
Actor Creation



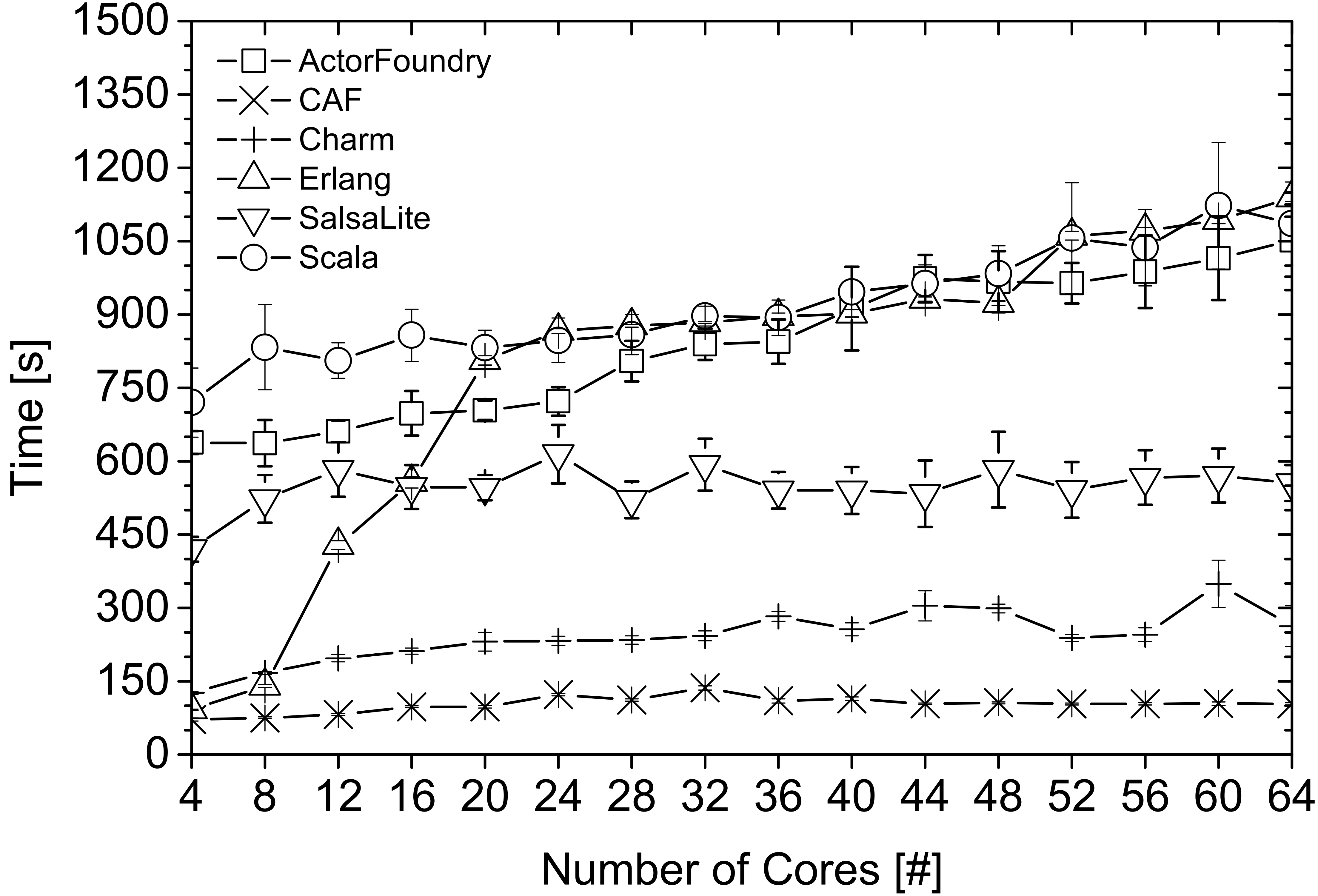
Benchmark #5

Incast

- **N:1 communication**
- 100 actors, each sending 1M messages to a single receiver
- Benchmark runtime := time it takes until receiver got all messages
- Expectation: adding more cores speeds up senders ⇒ higher runtime

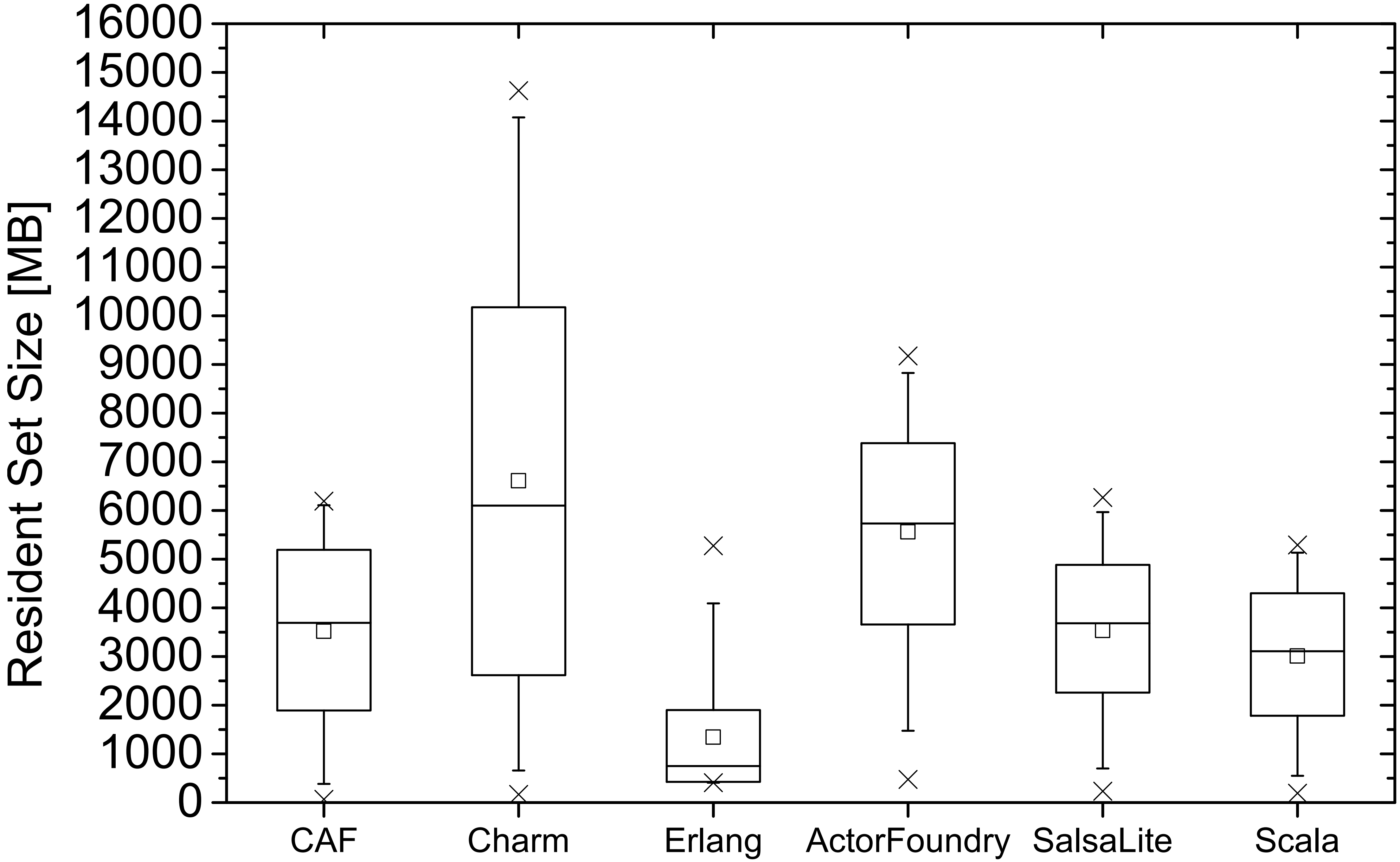


Mailbox - CPU



ActorFoundry
CAF
Charm
Erlang
SalsaLite
Scala
Time [s]
Number of Cores [#]
0
150
300
450
600
750
900
1050
1200
1350
1500
4
8
12
16
20
24
28
32
36
40
44
48
52
56
60
64

Mailbox - Memory

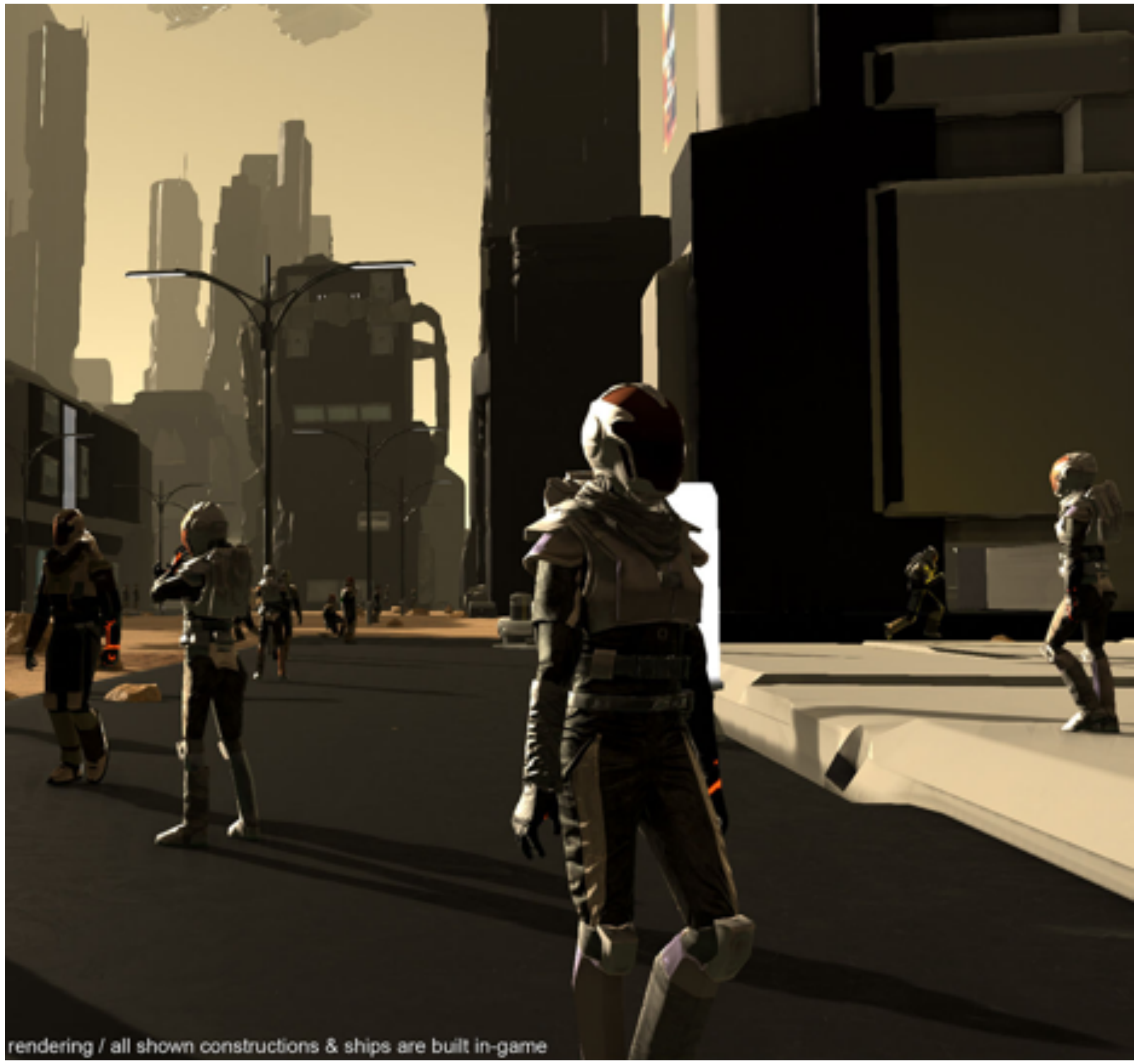


Project

- 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

CAF in MMOs

- **Dual Universe**
  - Single-shard sandbox MMO
  - Backend based on CAF
  - Pre-alpha
  - Developed at Novaquark (Paris)



http://www.dualthegame.com

CAF in Network Monitors

- Broker: Bro's messaging library
  - Hierarchical publish/subscribe communication
  - Distributed data stores
  - Used in Bro cluster deployments at +10 Gbps

http://bro.github.io/broker

CAF in Network Forensics

- **VAST**: Visibility Across Space and Time
  - Interactive, iterative data exploration
  - Actorized concurrent indexing & search
  - Scales from single-machine to cluster

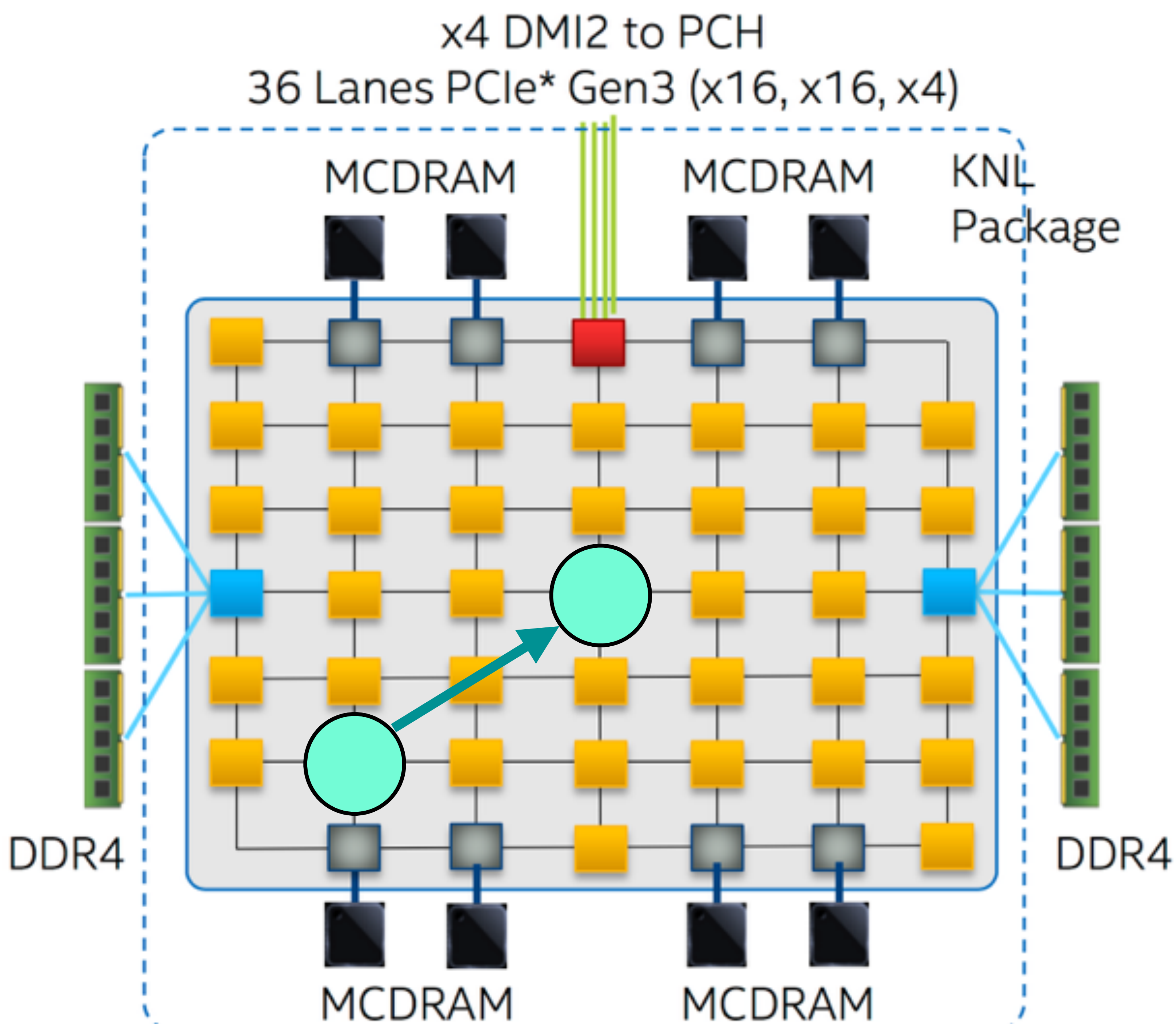
http://vast.io

Research Opportunities

Scheduler

- Improve **work stealing**
  - Stealing has no uniform cost
  - Account for cost of NUMA domain transfer

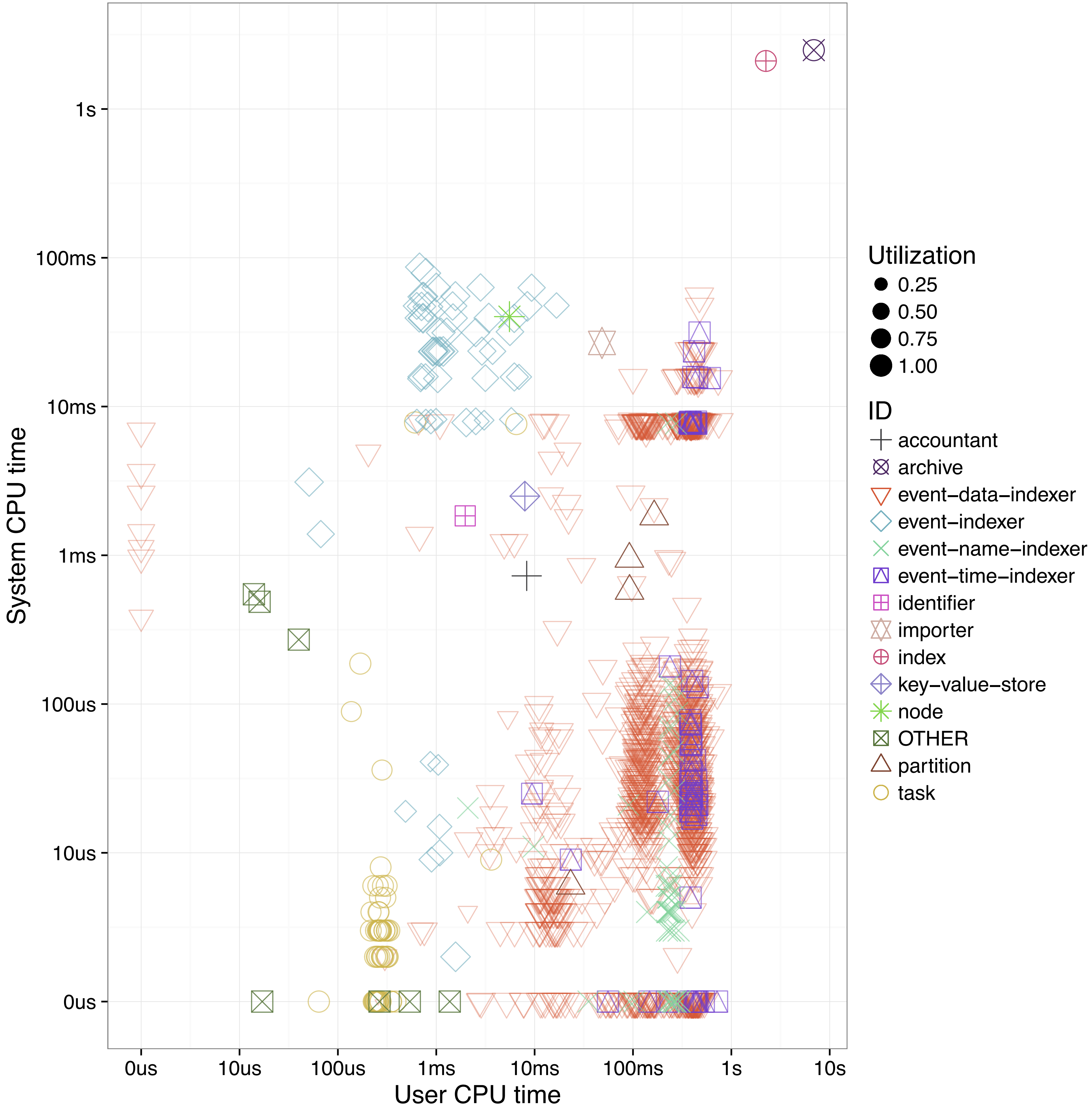
Scheduler



x4 DMI2 to PCH
36 Lanes PCIe* Gen3 (x16, x16, x4)
MCDRAM
MCDRAM
KNL
Package
DDR4
DDR4
MCDRAM
MCDRAM

Scheduler

- Optimize **job placement**
  - Avoid work stealing when possible
  - Measure resource utilization
  - Derive optimal schedule (cf. TetriSched)

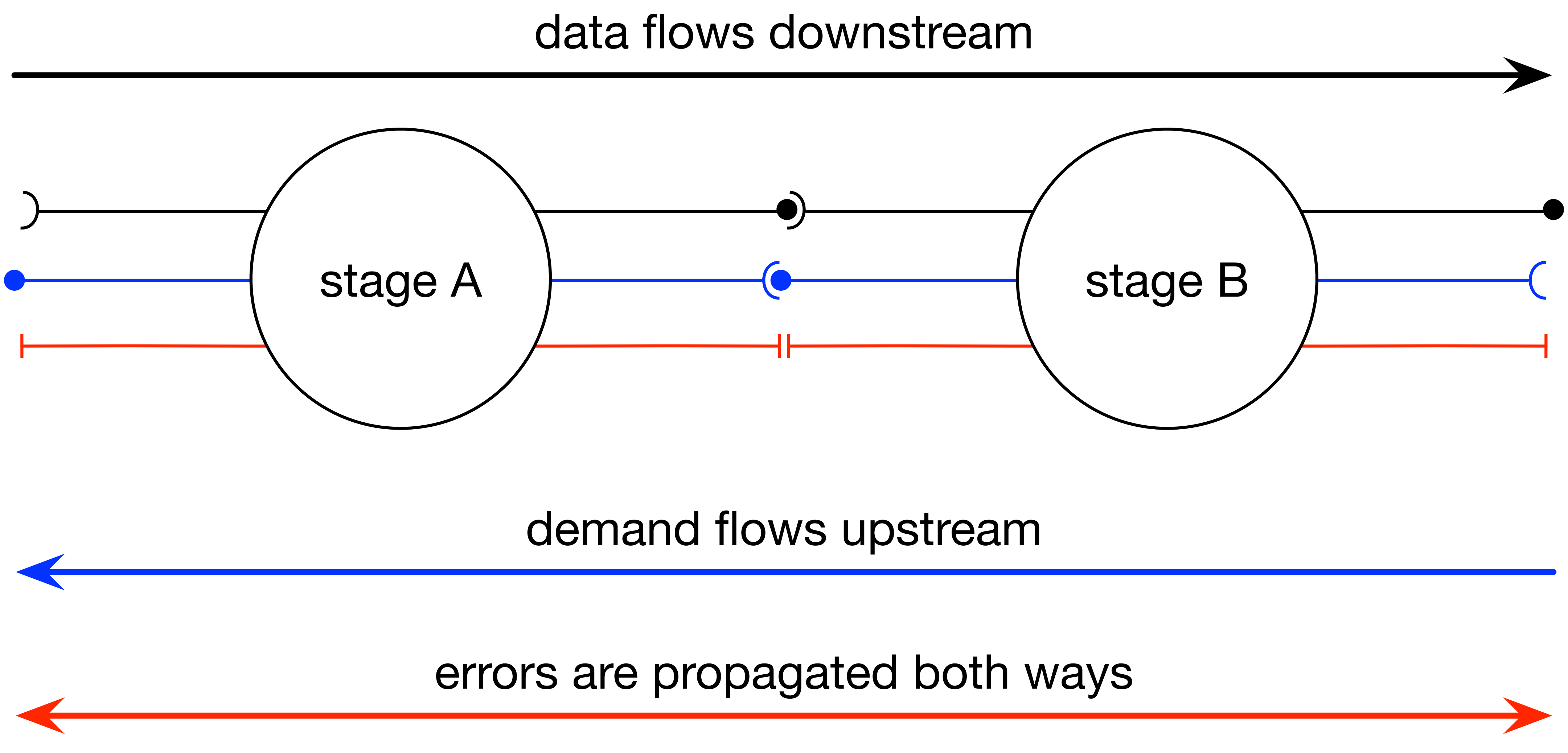


System CPU time
1s
100ms
10ms
1ms
100us
10us
0us
User CPU time
0us
10us
100us
1ms
10ms
100ms
1s
10s
Utilization
0.25
0.50
0.75
1.00
ID
accountant
archive
event-data-indexer
event-indexer
event-name-indexer
event-time-indexer
identifier
importer
index
key-value-store
node
OTHER
partition
task

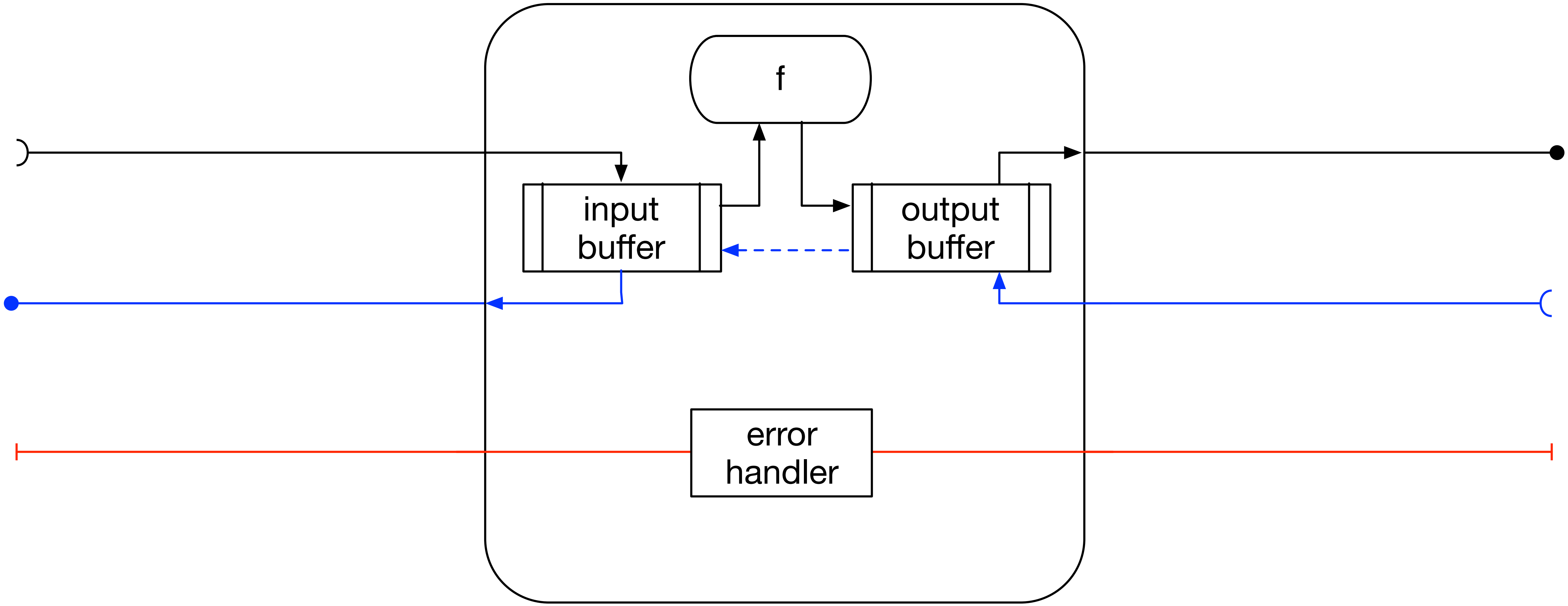
Streaming

- CAF as **building block** for streaming engines
- Existing systems exhibit vastly different semantics
  - SparkStreaming, Heron/Storm/Trident, *MQ, Samza, Flink, Beam/Dataflow, ...

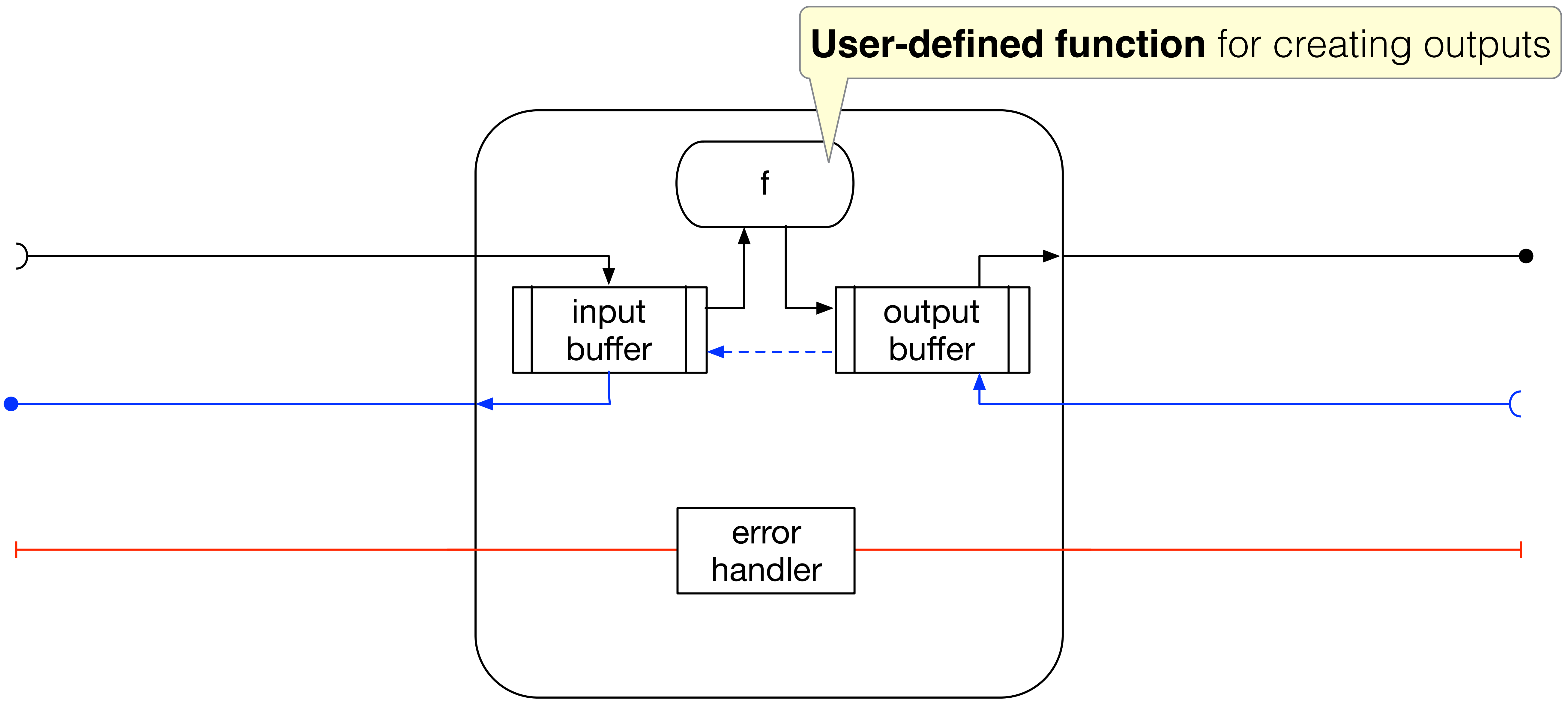
Streaming


data flows downstream
stage A
stage B
demand flows upstream
errors are propagated both ways

Streaming

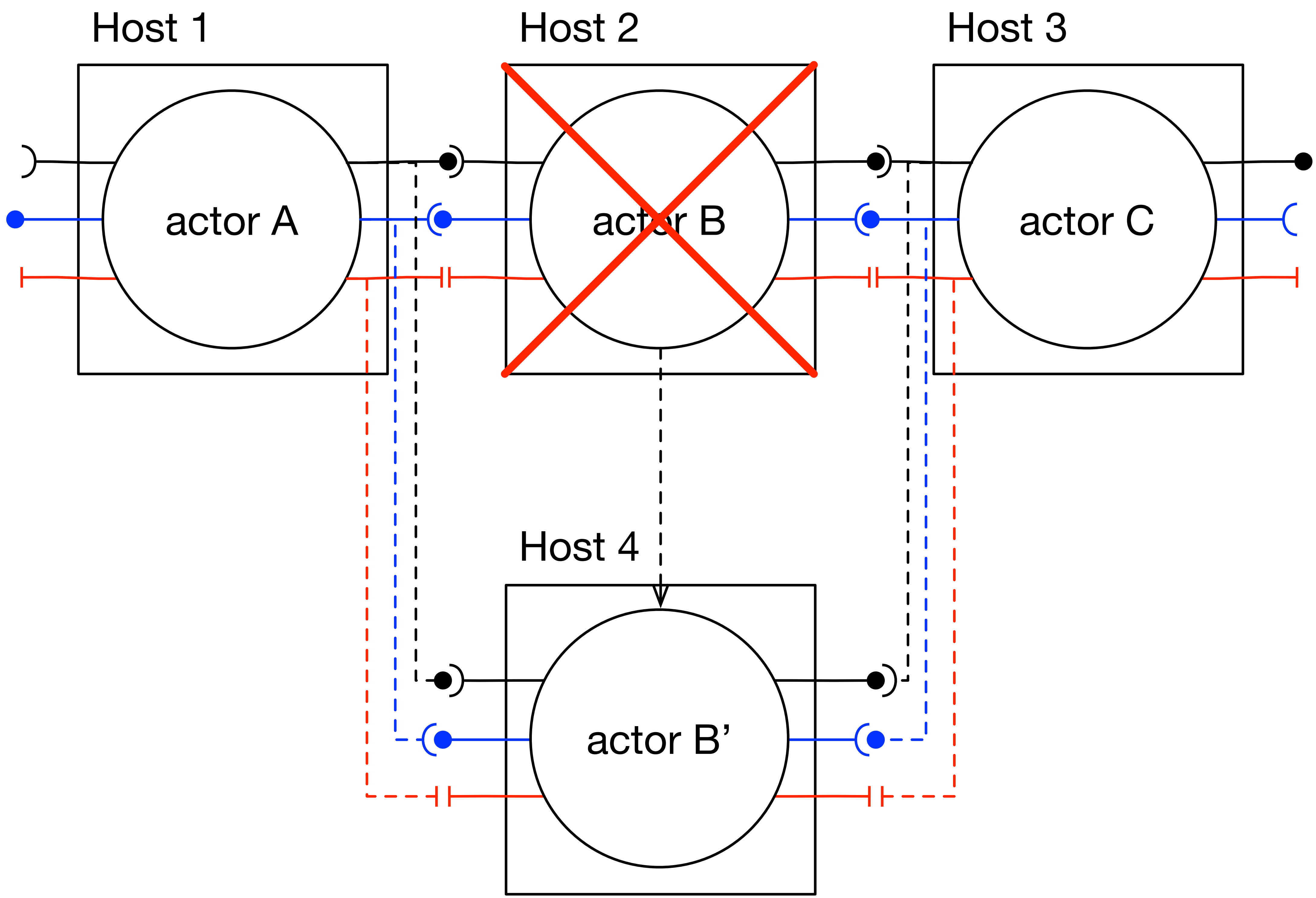


Streaming

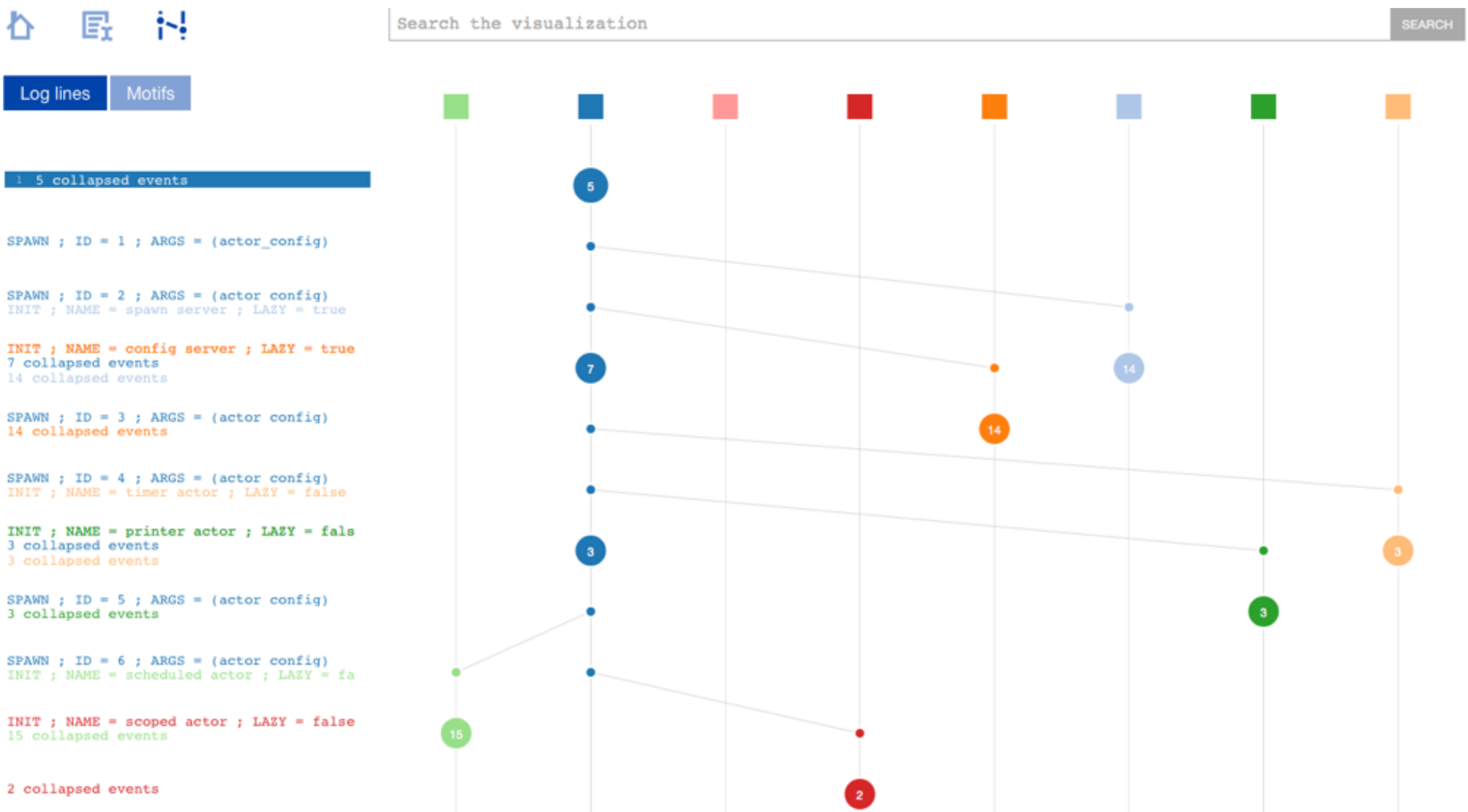


User-defined function for creating outputs
f
input buffer
output buffer
error handler

Fault Tolerance



Debugging


Search the visualization
SEARCH
Log lines
Motifs
1 5 collapsed events
SPAWN ; ID = 1 ; ARGS = (actor_config)
SPAWN ; ID = 2 ; ARGS = (actor config)
INIT ; NAME = spawn server ; LAZY = true
INIT ; NAME = config server ; LAZY = true
7 collapsed events
14 collapsed events
SPAWN ; ID = 3 ; ARGS = (actor config)
14 collapsed events
SPAWN ; ID = 4 ; ARGS = (actor config)
INIT ; NAME = timer actor ; LAZY = false
INIT ; NAME = printer actor ; LAZY = fals
3 collapsed events
3 collapsed events
SPAWN ; ID = 5 ; ARGS = (actor config)
3 collapsed events
SPAWN ; ID = 6 ; ARGS = (actor config)
INIT ; NAME = scheduled actor ; LAZY = fa
INIT ; NAME = scoped actor ; LAZY = false
15 collapsed events
2 collapsed events

Summary

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

Questions?

http://actor-framework.org

Our C++ chat:

https://gitter.im/vast-io/cpp

Backup Slides

API

Sending Messages

- Asynchronous fire-and-forget

```
self->send(other, x, xs...);
```

- Timed request-response with one-shot continuation

```
self->request(other, timeout, x, xs...).then(
  [=](T response) {
  }
);
```

- Transparent forwarding of message authority

```
self->delegate(other, x, xs...);
```

Actors as Function Objects

```
actor a = sys.spawn(adder);
auto f = make_function_view(a);
cout << "f(1, 2) = "
     << to_string(f(1, 2))
     << "\n";
```

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>
  >;
```

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

Features

Implementations	Native Execution	Garbage Collection	Pattern Matching	Copy-On-Write Messaging	Failure Propagation	Dynamic Behaviors	Compile-Time Type Checking	Run-Time Type Checking	Exchangeable Scheduler	Network Actors	GPU Actors	Backend
Erlang [13]	✗	✓	✓	✗	✓	✓	✗	✓	✓	✓	✗	BEAM
Elixir [70]	✗	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	BEAM
Akka/Scala [7]	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓	✗	JVM
SALSA Lite [62]	✗	✓	✗	✗	✗	✗	✓	✓	✗	●†	✗	JVM
Actor Foundry [2]	✗	✓	✓	✗	✗	✓	✗	✓	✗	✓	✗	JVM
Pulsar [151]	✗	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	JVM
Pony [47]	✓	✓	✓	✗	✗	✗	✓	✓	✗	✓	✗	LLVM
Charm++ [109]	✓	●*	✗	✗	✗	✗	✓	✓	✗	✓	✗	C++
Theron [182]	✓	●*	✗	✗	✗	✓	✗	✓	✗	✓	✗	C++
libprocess [124]	✓	●*	✗	✗	✓	✗	✓	✗	✗	✓	✗	C++
CAF [44]	✓	●*	✓	✓	✓	✓	✓	✓	✓	✓	✓	C++

* Via reference counting, as opposed to tracing garbage collection.

† Only in SALSA, not SALSA Lite.