CAF
C++ Actor Framework

Matthias Vallentin
UC Berkeley

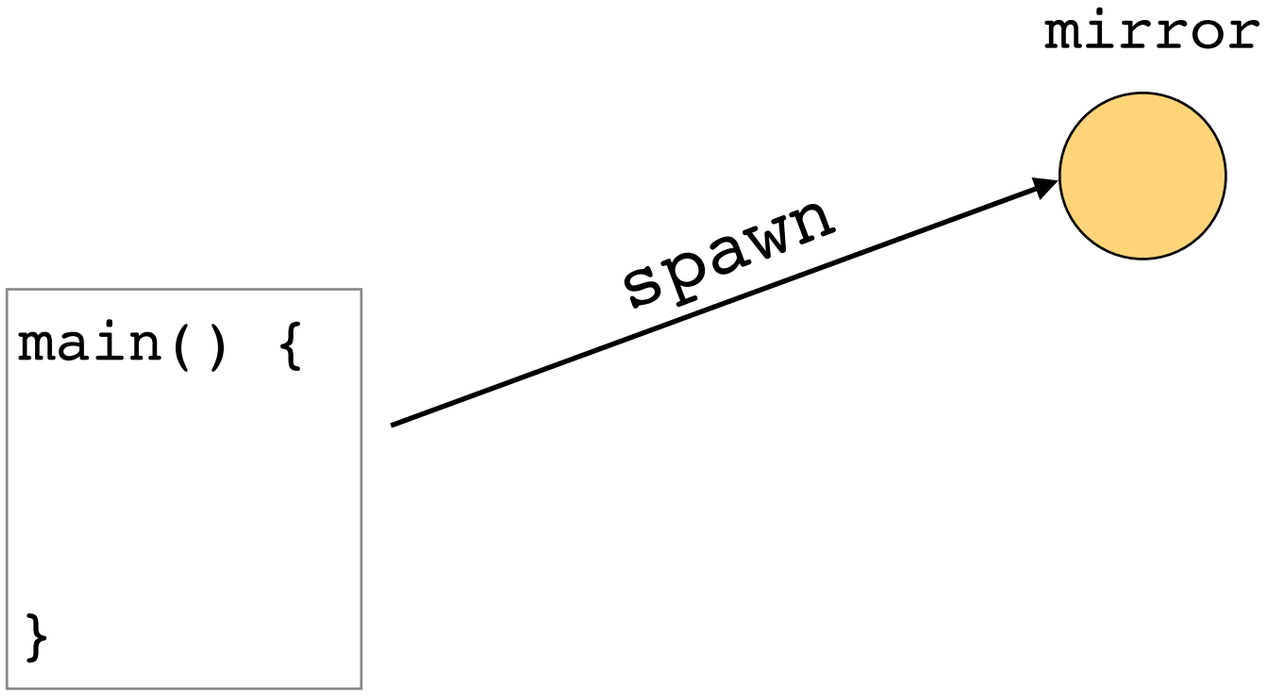
2016 Berkeley C++ Summit

Hello World

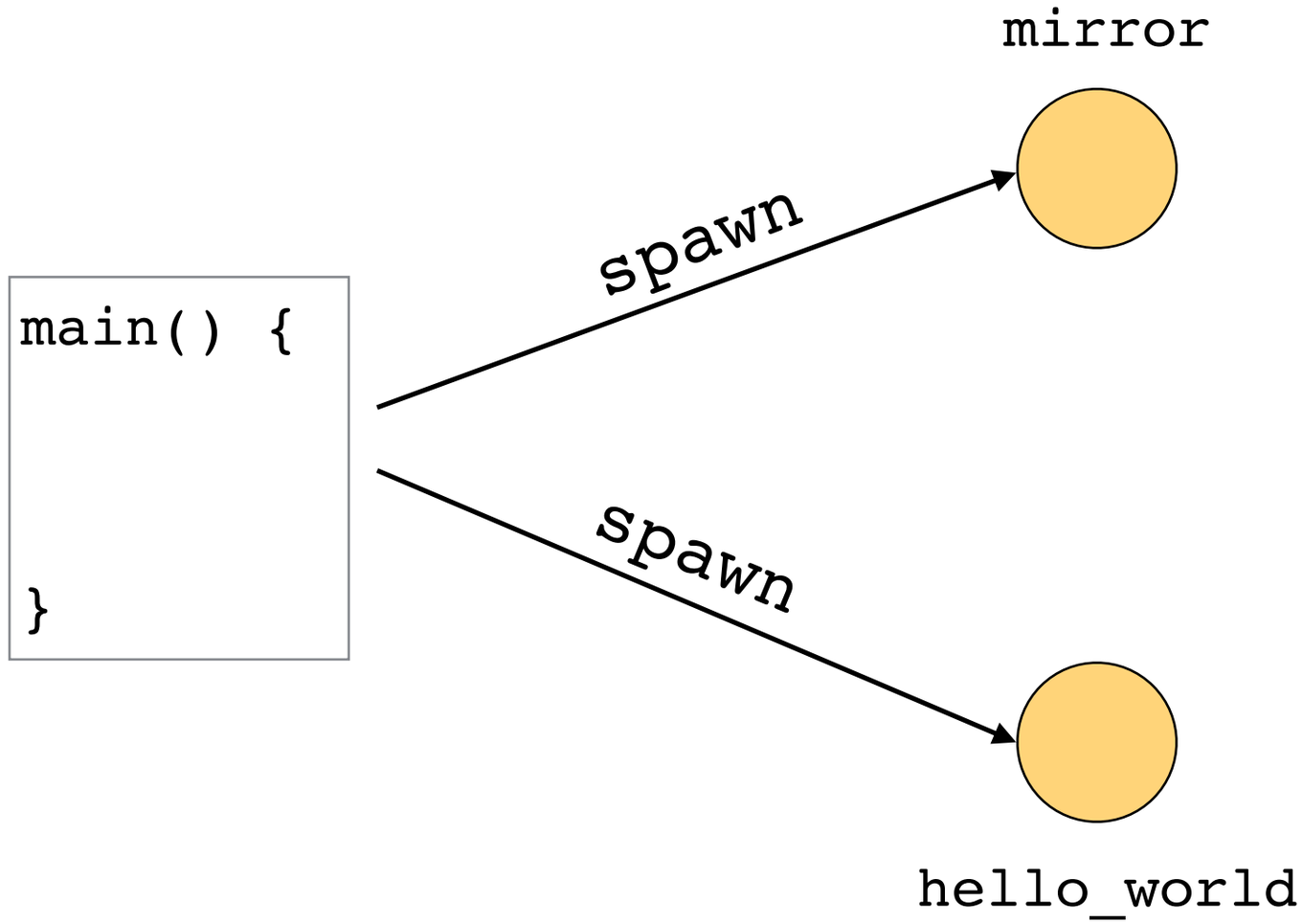
Hello World

```
main() {



}
```

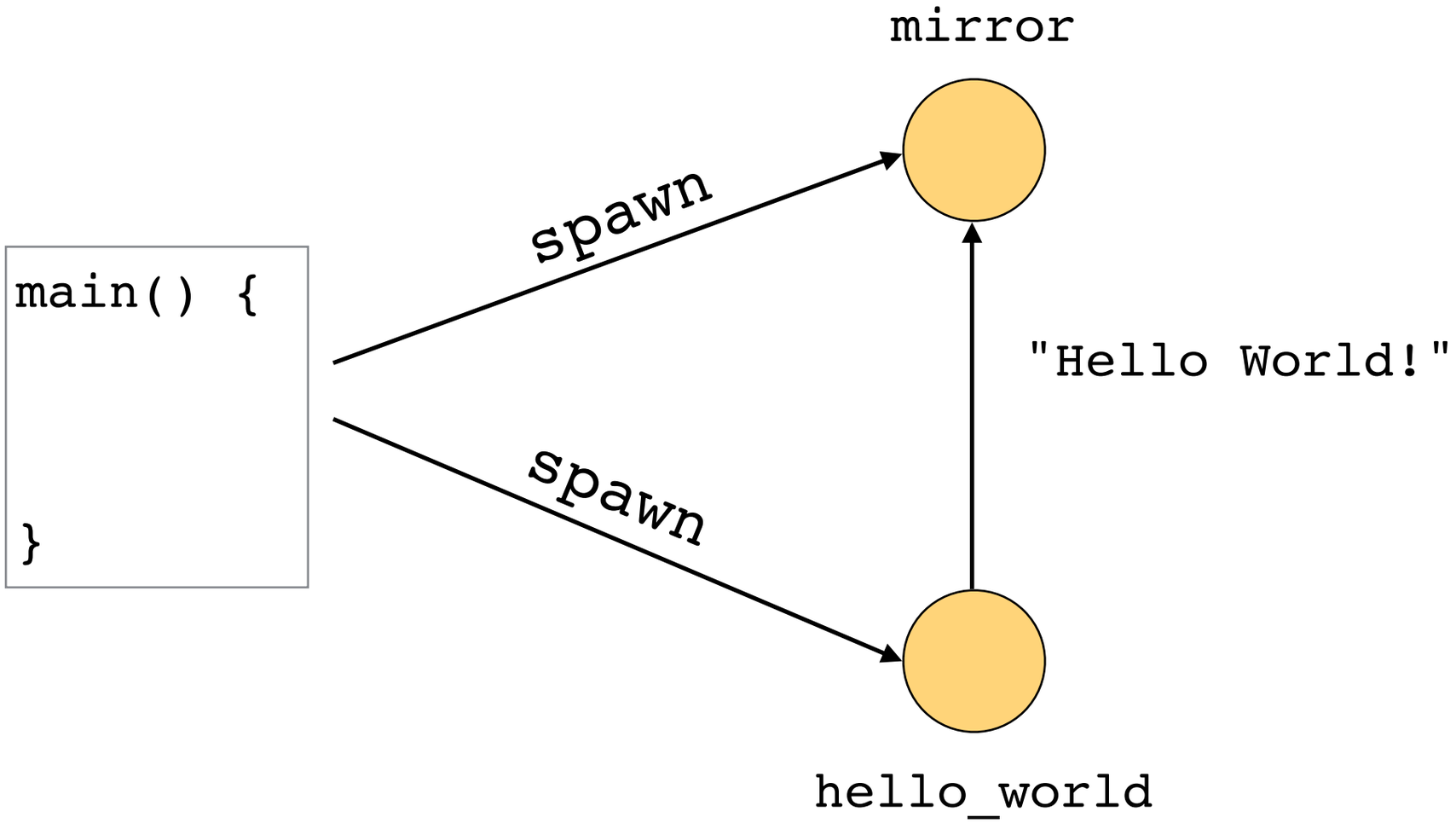
Hello World



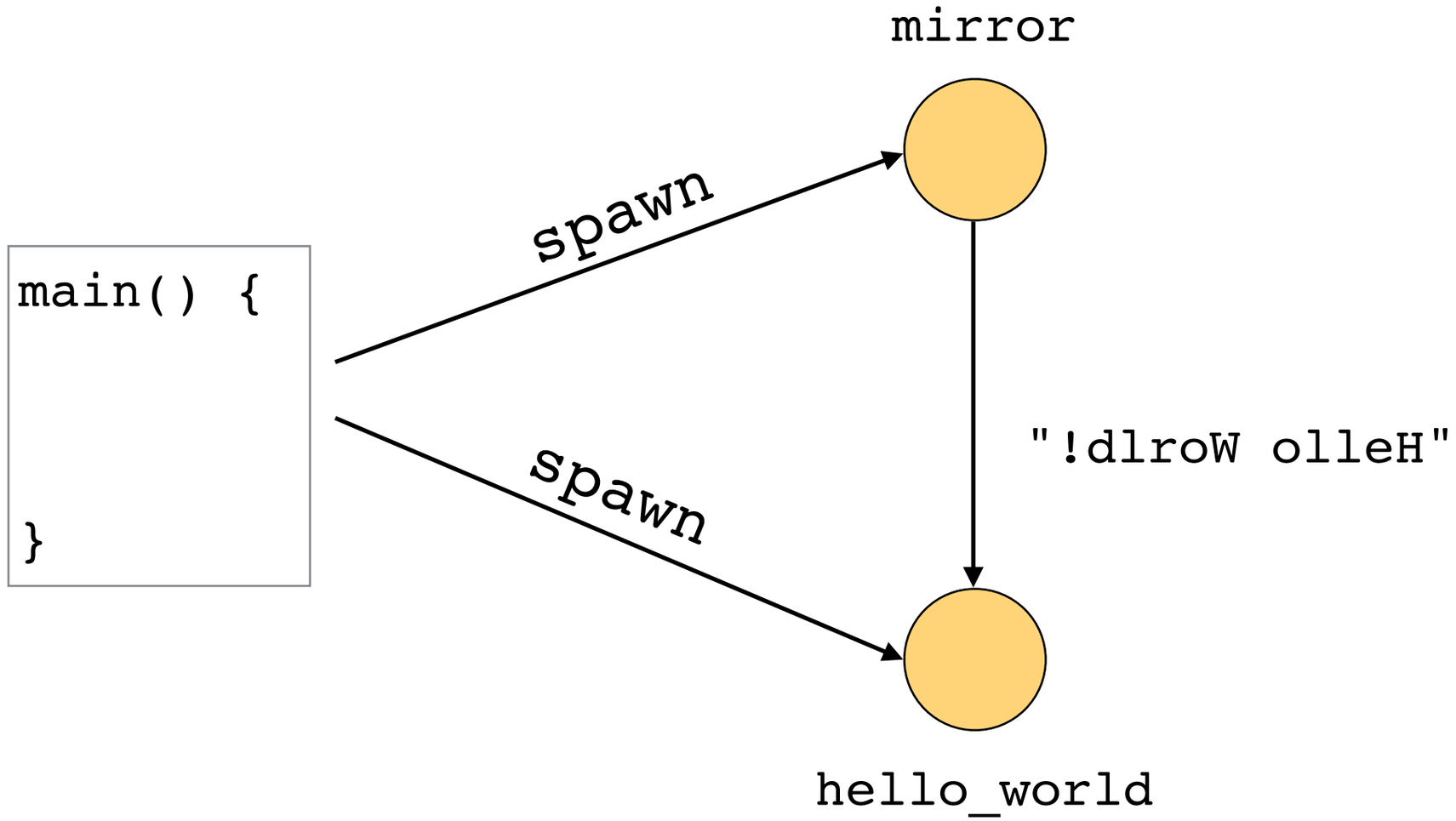
Hello World



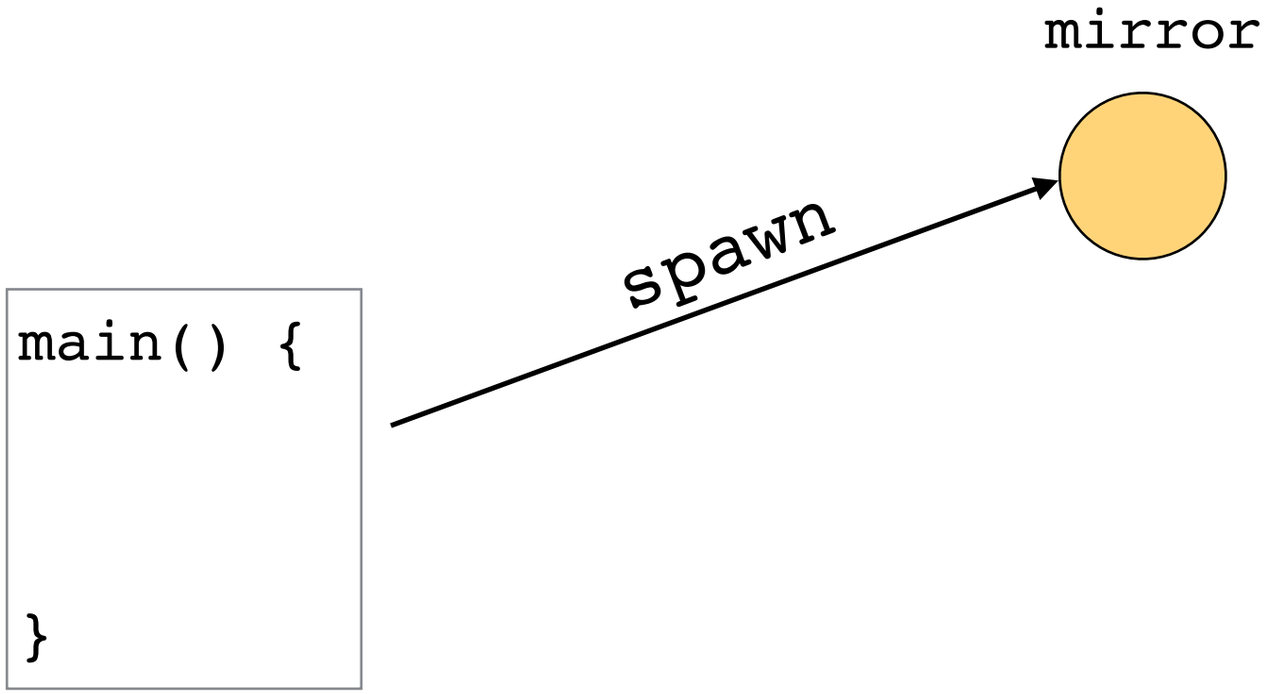
Hello World



Hello World



Hello World



Hello World

```
main() {



}
```

Hello World

```cpp
#include <string>
#include <iostream>

#include "caf/all.hpp"

using namespace caf;
using namespace std;

behavior mirror() {
  return {
    [=](const string& str) {
      return string(str.rbegin(), str.rend());
    }
  };
}

void hello_world(event_based_actor* self, const actor& buddy) {
  self->request(buddy, chrono::seconds(10), "Hello World!").then(
    [=](const string& str) {
      cout << str << endl;
    }
  );
}

int main() {
  actor_system_config cfg;
  actor_system system{cfg};
  auto m = system.spawn(mirror);
  system.spawn(hello_world, m);
}
```

```cpp
#include <string>
#include <iostream>

#include "caf/all.hpp"

using namespace caf;
using namespace std;

behavior mirror() {
  return {
    [=](const string& str) {
      return string(str.rbegin(), str.rend());
    }
  };
}
```

Function returning **behavior** is-a actor

```cpp
void hello_world(event_based_actor* self, const actor& buddy) {
  self->request(buddy, chrono::seconds(10), "Hello World!").then(
    [=](const string& str) {
      cout << str << endl;
    }
  );
}

int main() {
  actor_system_config cfg;
  actor_system system{cfg};
  auto m = system.spawn(mirror);
  system.spawn(hello_world, m);
}
```

```cpp
#include <string>
#include <iostream>

#include "caf/all.hpp"

using namespace caf;
using namespace std;

behavior mirror() {
  return {
    [=](const string& str) {
      return string(str.rbegin(), str.rend());
    }
  };
}
```

Function returning **behavior** is-a actor

Function with **event_based_actor** as first argument is-a actor

```cpp
void hello_world(event_based_actor* self, const actor& buddy) {
  self->request(buddy, chrono::seconds(10), "Hello World!").then(
    [=](const string& str) {
      cout << str << endl;
    }
  );
}

int main() {
  actor_system_config cfg;
  actor_system system{cfg};
  auto m = system.spawn(mirror);
  system.spawn(hello_world, m);
}
```

```cpp
#include <string>
#include <iostream>

#include "caf/all.hpp"

using namespace caf;
using namespace std;

behavior mirror() {
  return {
    [=](const string& str) {
      return string(str.rbegin(), str.rend());
    }
  };
}
```

Function returning **behavior** is-a actor

Function with **event_based_actor** as first argument is-a actor

```cpp
void hello_world(event_based_actor* self, const actor& buddy) {
  self->request(buddy, chrono::seconds(10), "Hello World!").then(
    [=](const string& str) {
      cout << str << endl;
    }
  );
}
```

Actor implicitly terminates at end of scope

```cpp
int main() {
  actor_system_config cfg;
  actor_system system{cfg};
  auto m = system.spawn(mirror);
  system.spawn(hello_world, m);
}
```

```cpp
#include <string>
#include <iostream>

#include "caf/all.hpp"

using namespace caf;
using namespace std;

behavior mirror() {
  return {
    [=](const string& str) {
      return string(str.rbegin(), str.rend());
    }
  };
}
```

Function returning **behavior** is-a actor

Function with **event_based_actor** as first argument is-a actor

```cpp
void hello_world(event_based_actor* self, const actor& buddy) {
  self->request(buddy, chrono::seconds(10), "Hello World!").then(
    [=](const string& str) {
      cout << str << endl;
    }
  );
}

int main() {
  actor_system_config cfg;
  actor_system system{cfg};
  auto m = system.spawn(mirror);
  system.spawn(hello_world, m);
}
```

Actor implicitly terminates at end of scope

Encapsulates state: worker threads, actors, etc.

```cpp
#include <string>
#include <iostream>

#include "caf/all.hpp"

using namespace caf;
using namespace std;

behavior mirror() {
  return {
    [=](const string& str) {
      return string(str.rbegin(), str.rend());
    }
  };
}
```

Function returning **`behavior`** is-a actor

Function with **`event_based_actor`** as first argument is-a actor

```cpp
void hello_world(event_based_actor* self, const actor& buddy) {
  self->request(buddy, chrono::seconds(10), "Hello World!").then(
    [=](const string& str) {
      cout << str << endl;
    }
  );
}

int main() {
  actor_system_config cfg;
  actor_system system{cfg};
  auto m = system.spawn(mirror);
  system.spawn(hello_world, m);
}
```

Actor implicitly terminates at end of scope

Encapsulates state: worker threads, actors, etc.

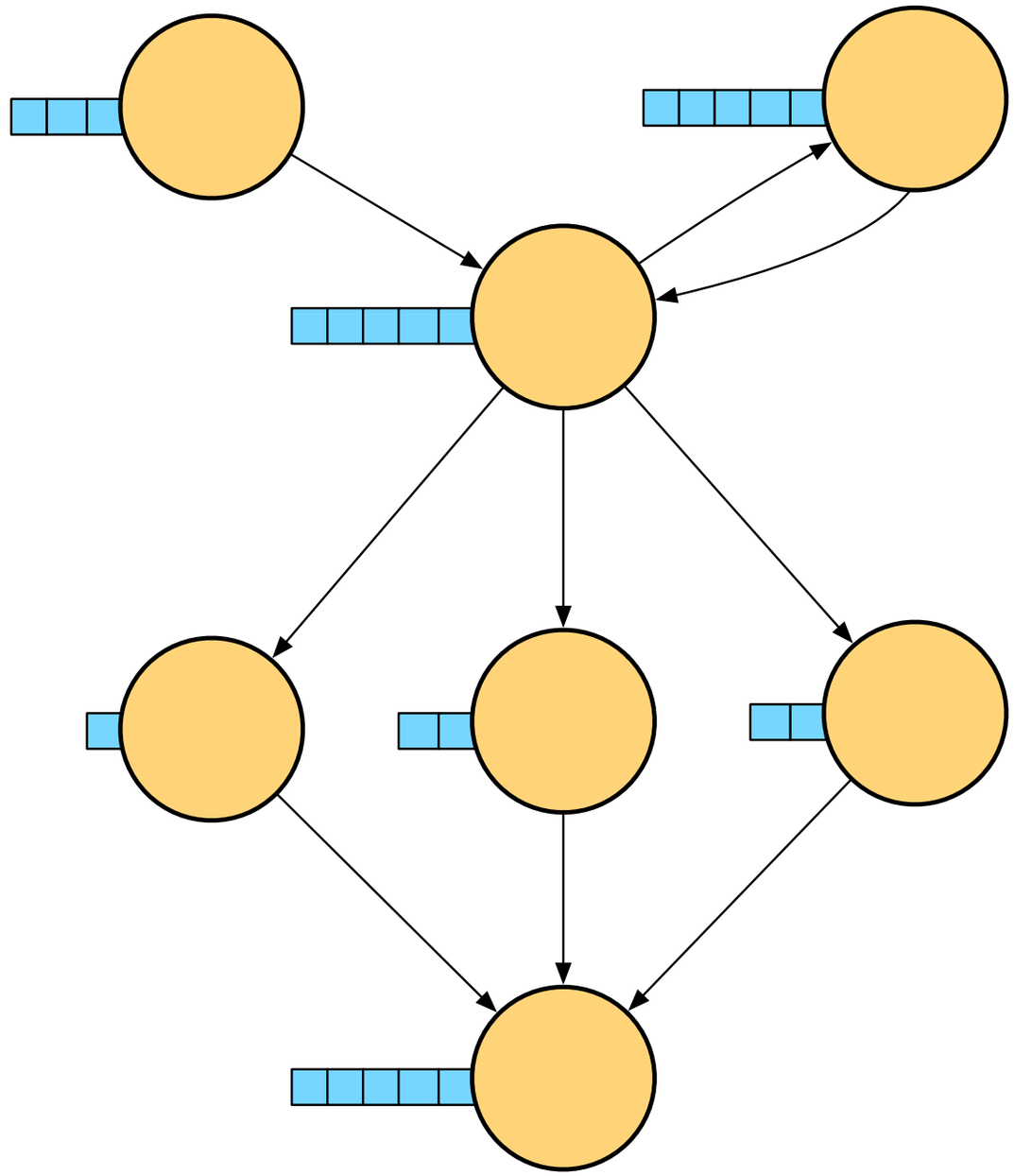
`~actor_system()` blocks until all actors have terminated / no more references

Outline

- Actor Model
- CAF
- Performance

Actor Model

- **Actor**: sequential unit of computation
- **Message**: n-ary typed tuple
- **Mailbox**: FIFO/queue of messages
- **Behavior**: function how to process next message



Actor Semantics

Actor Semantics

- Actors are **reactive**

Actor Semantics

- Actors are **reactive**
- In response to a message, an actor can do *any* of:

Actor Semantics

- Actors are **reactive**
- In response to a message, an actor can do *any* of:

1. Creating (*spawn*) new actors

Actor Semantics

- Actors are **reactive**
- In response to a message, an actor can do *any* of:

1. Creating (*spawn*) new actors
2. Sending messages to other actors

Actor Semantics

- Actors are **reactive**
- In response to a message, an actor can do *any* of:

1. Creating (*spawn*) new actors
2. Sending messages to other actors
3. Designating a behavior for the next message

Key Advantages

Key Advantages

- All actors execute **concurrently**

Key Advantages

- All actors execute **concurrently**
- **Asynchronous** (non-blocking) message passing

Key Advantages

- All actors execute **concurrently**
- **Asynchronous** (non-blocking) message passing
- **Network-transparent** communication

Key Advantages

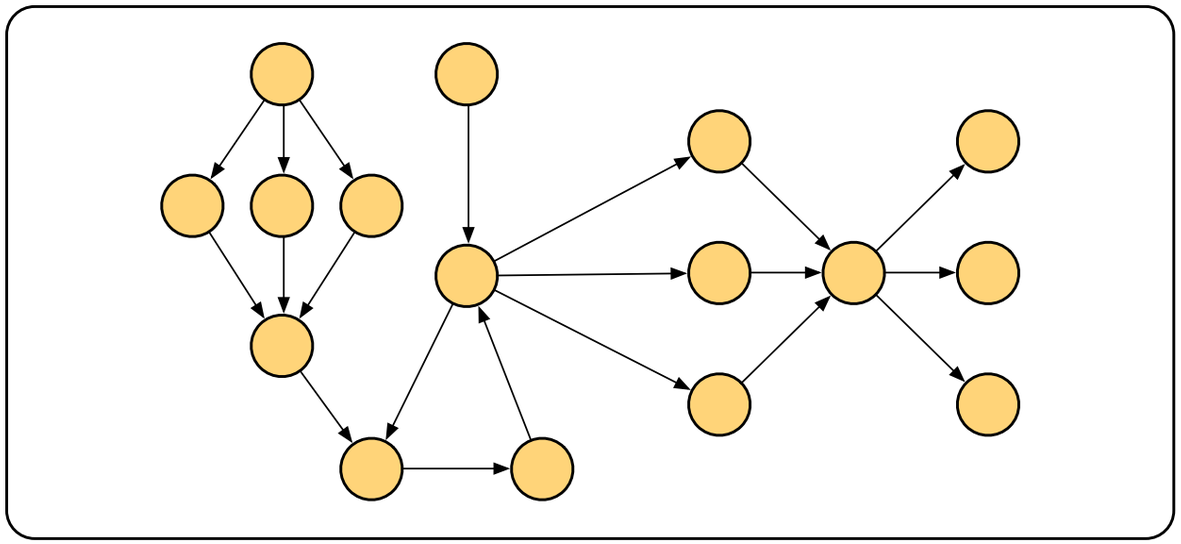
- All actors execute **concurrently**
- **Asynchronous** (non-blocking) message passing
- **Network-transparent** communication
- Hierarchical **fault propagation**

Key Advantages

- All actors execute **concurrently**
- **Asynchronous** (non-blocking) message passing
- **Network-transparent** communication
- Hierarchical **fault propagation**
- Sound semantics: **no data races** by design

CAF

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

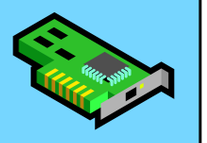
L2 cache

L1 cache

L2 cache

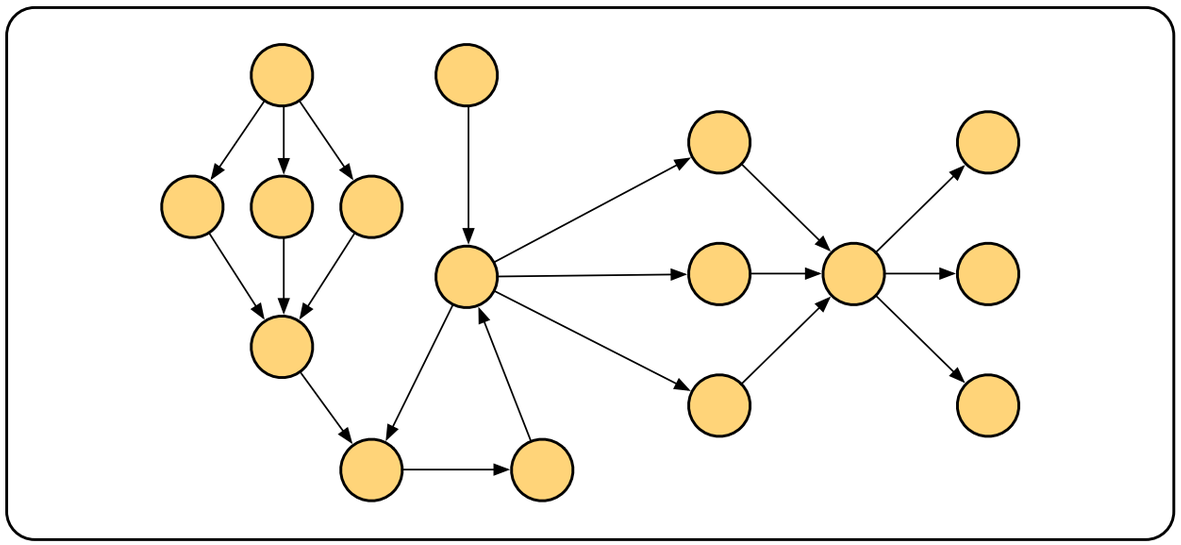
L1 cache

L2 cache



Network I/O

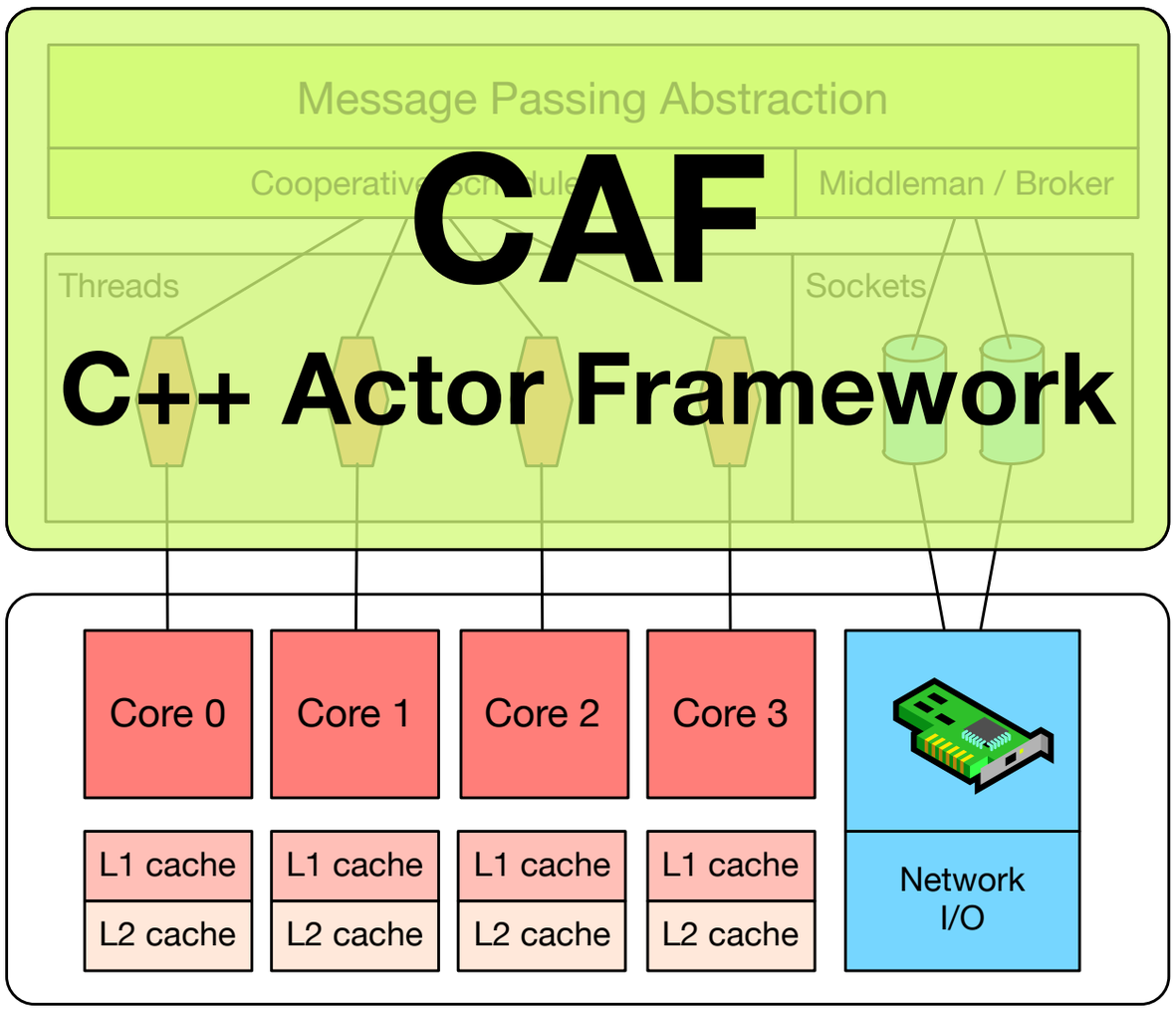
Application Logic



Actor Runtime

Operating System

Hardware



Scheduler

- Maps **N** **jobs** (= actors) to **M** **workers** (= threads)

- Maps **N** **jobs** (= actors) to **M** **workers** (= threads)
- Limitation: **cooperative multi-tasking** in user-space

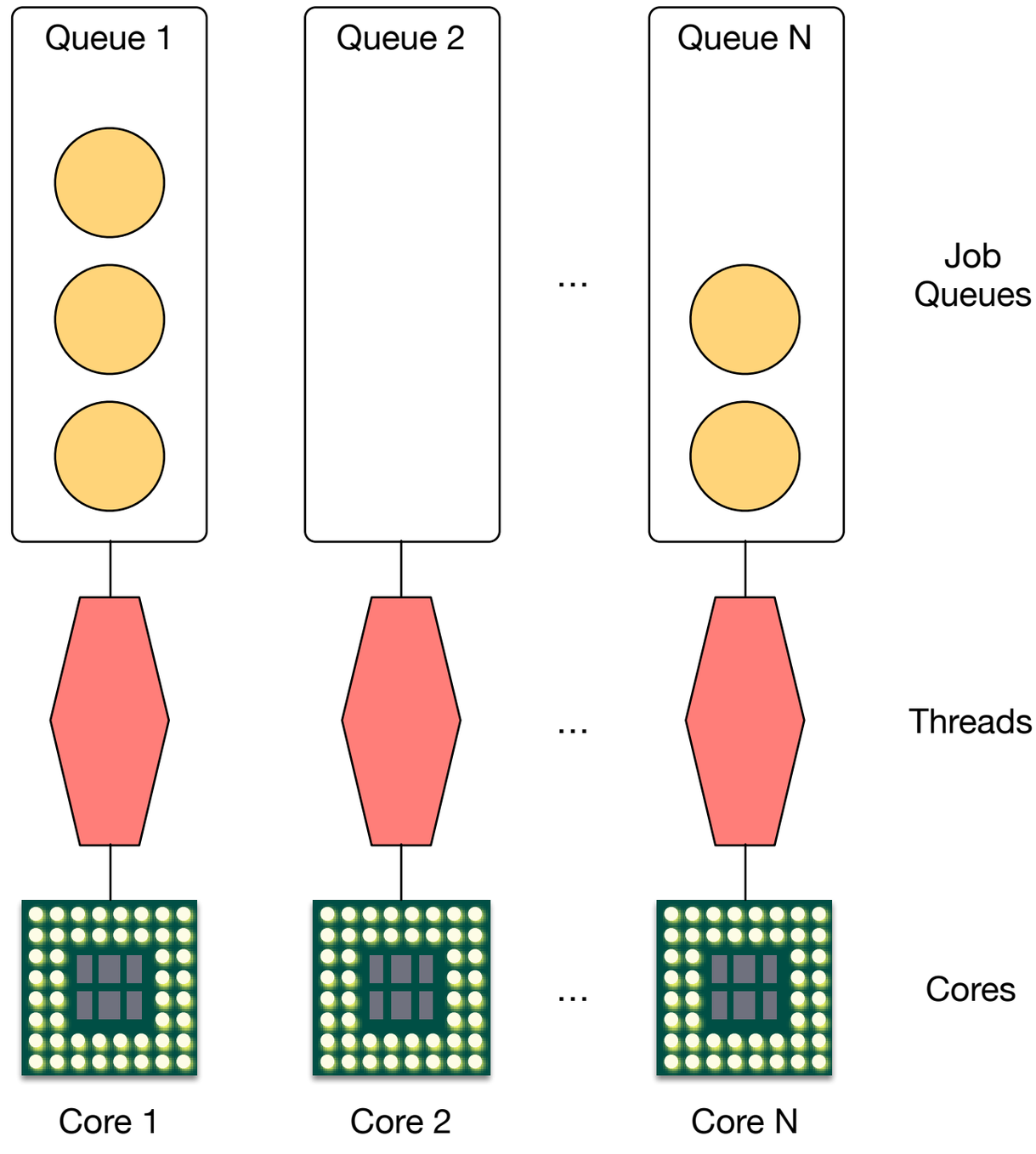
- Maps **N** **jobs** (= actors) to **M** **workers** (= threads)
- Limitation: **cooperative multi-tasking** in user-space
- **Issue**: actors that block

- 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 imbalance

- 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 imbalance
  - Not well-suited for **I/O-heavy tasks**

- 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 imbalance
  - Not well-suited for **I/O-heavy tasks**
  - Current solution: detach "uncooperative" actors into **separate thread**

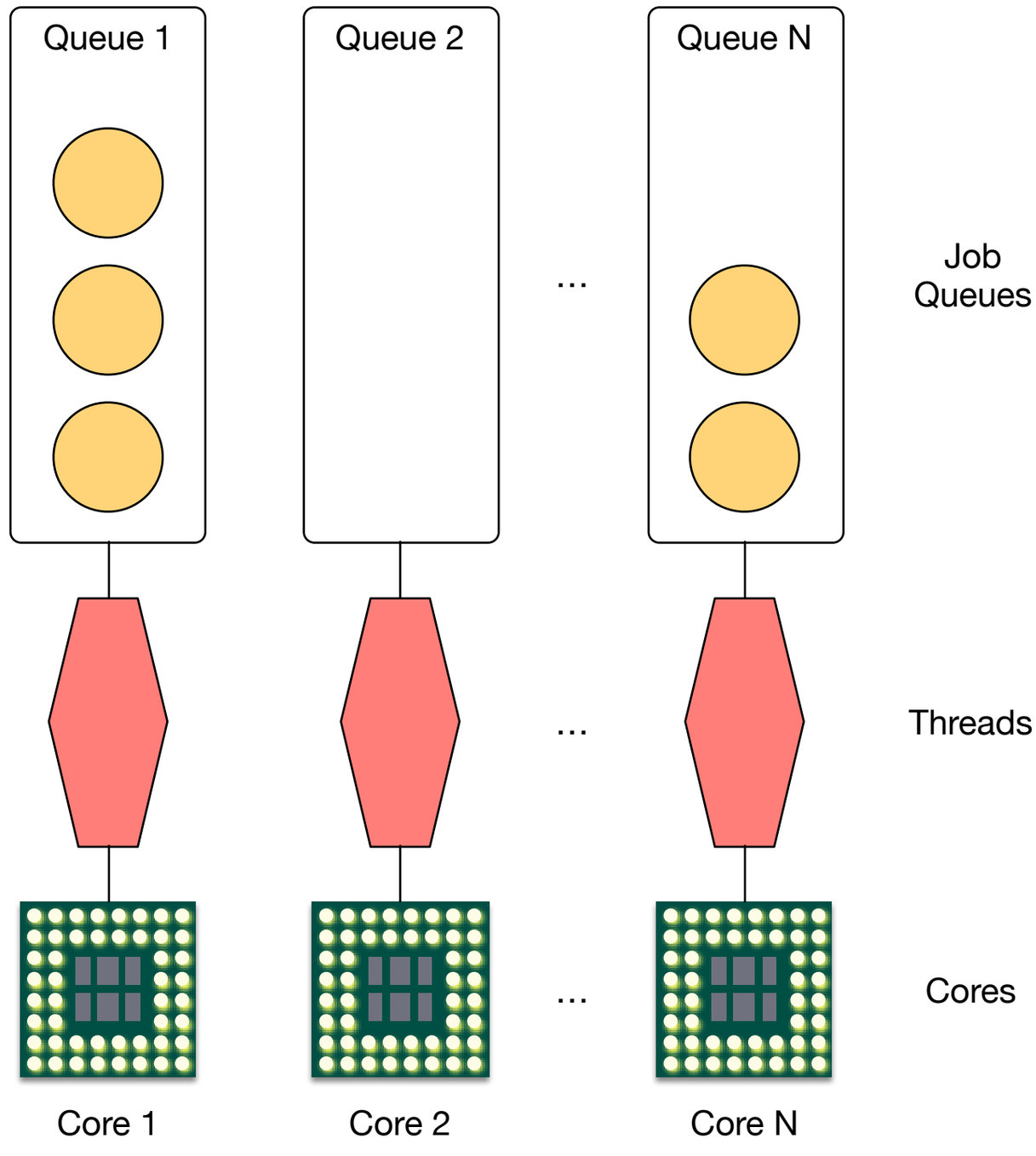
Work Stealing*



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

Work Stealing*

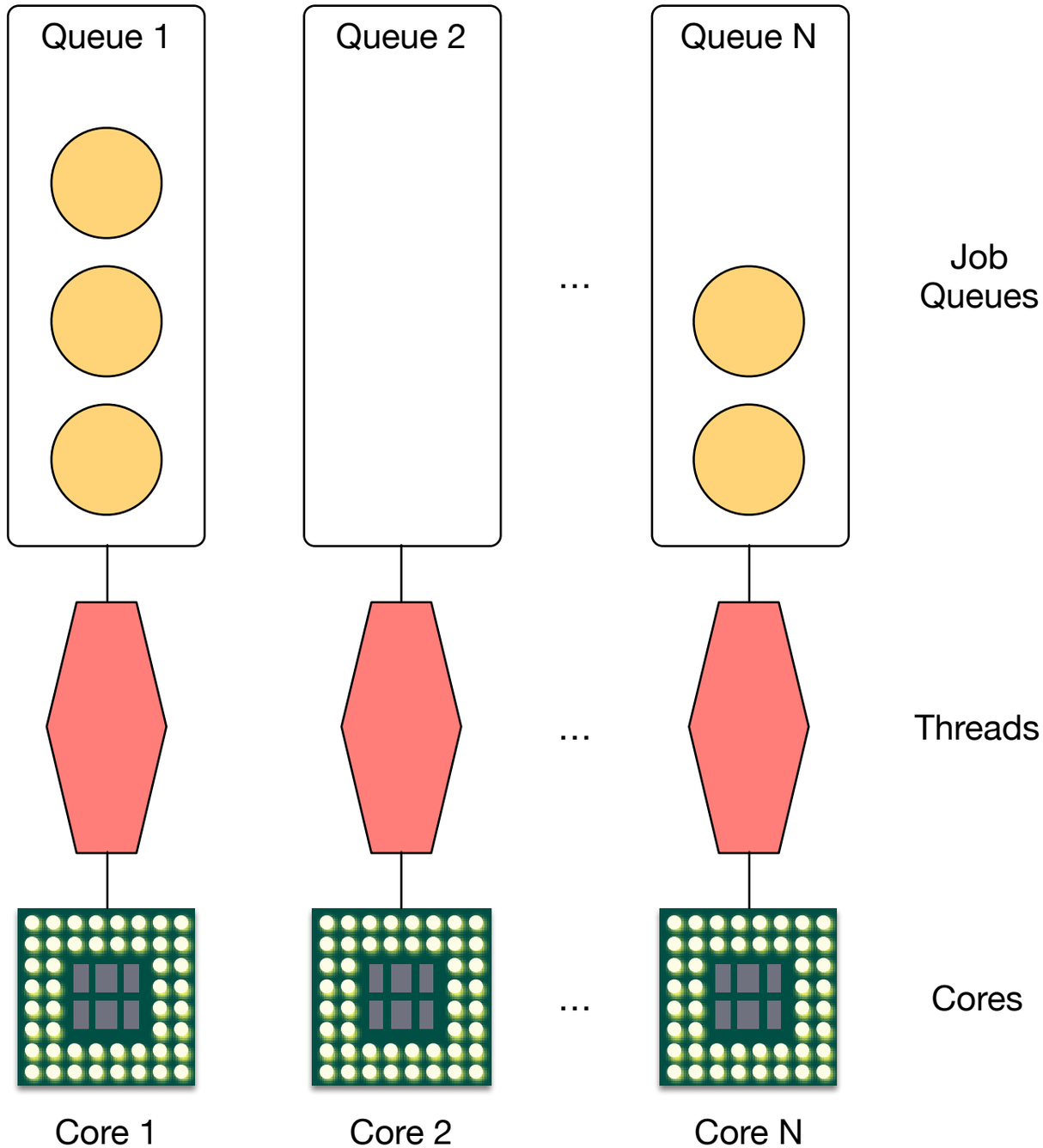
- **Decentralized**: one job queue and worker thread per core



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

Work Stealing*

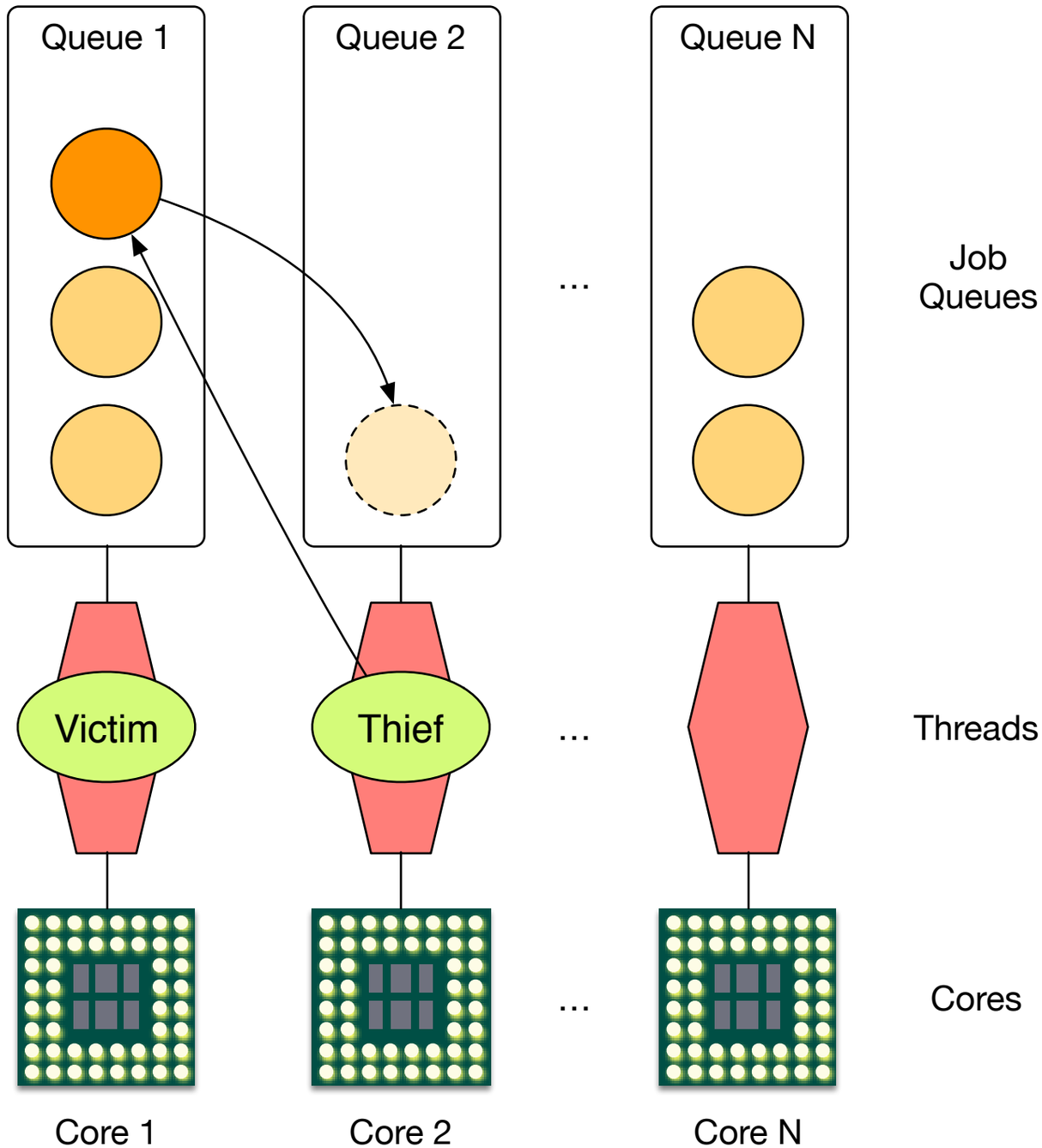
- **Decentralized**: one job queue and worker thread per core
- On empty queue, **steal** from other thread



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

Work Stealing*

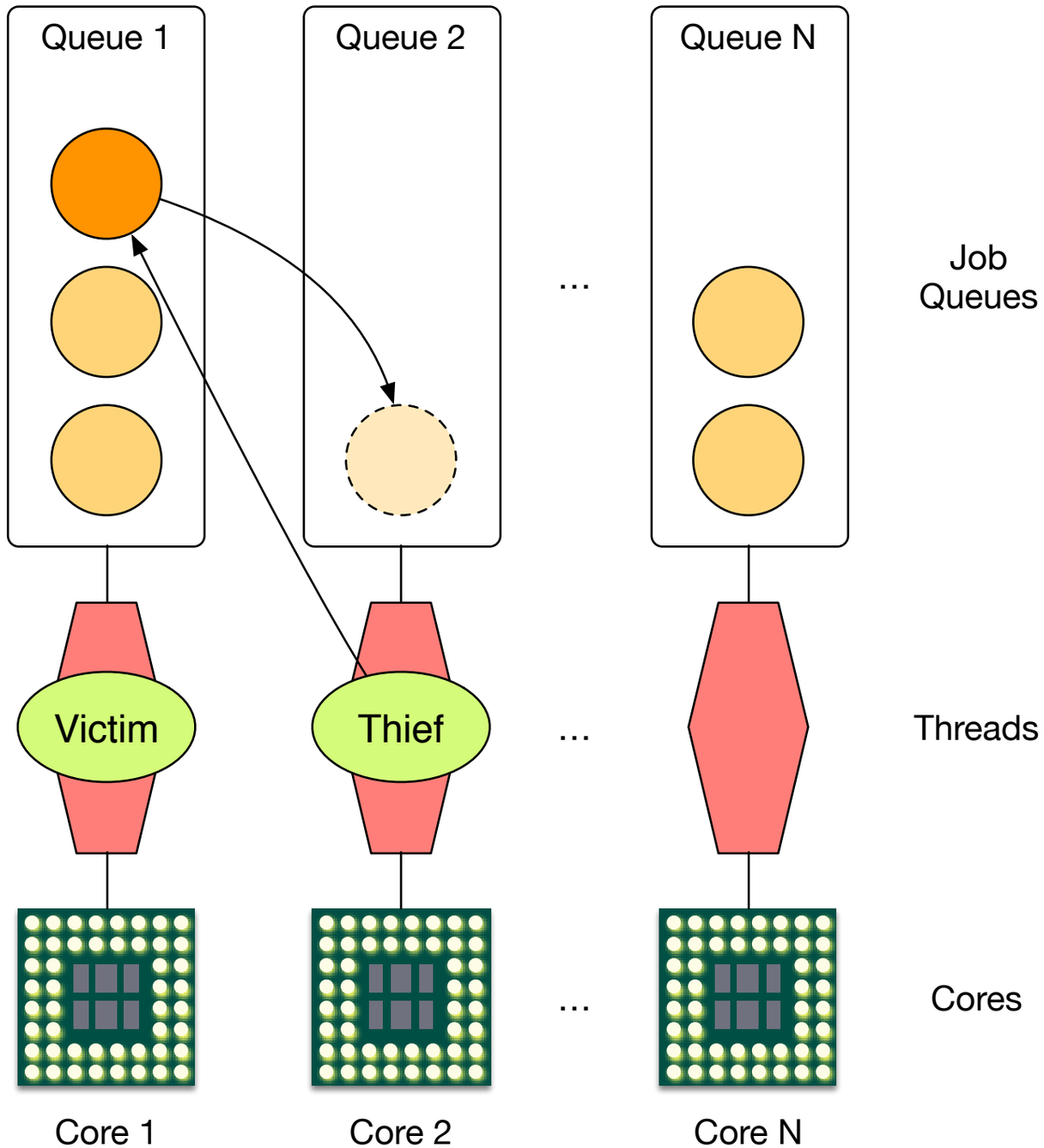
- **Decentralized**: one job queue and worker thread per core
- On empty queue, **steal** from other thread



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

Work Stealing*

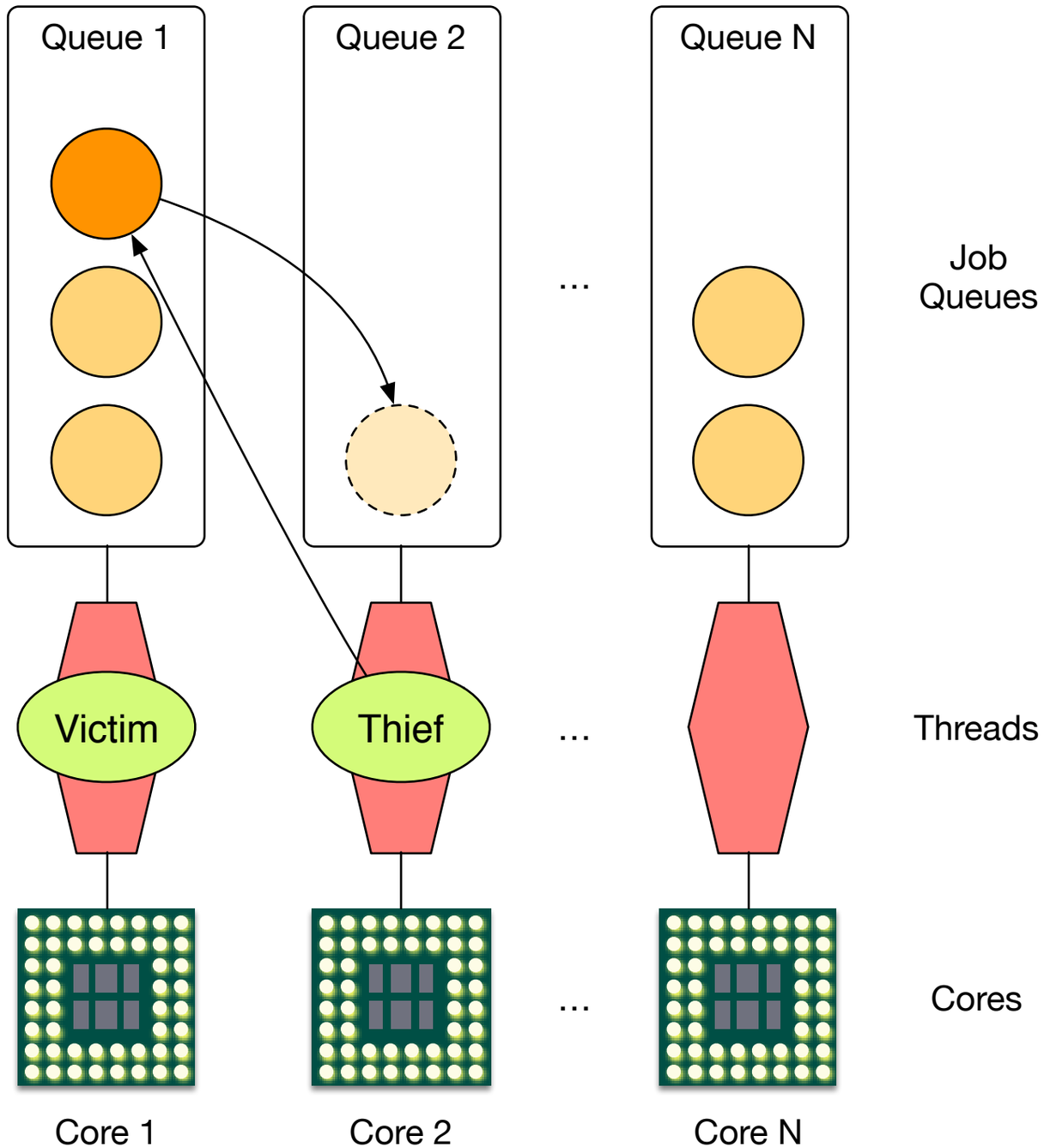
- **Decentralized**: one job queue and worker thread per core
- On empty queue, **steal** from other thread
- Efficient if stealing is a rare event



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

Work Stealing*

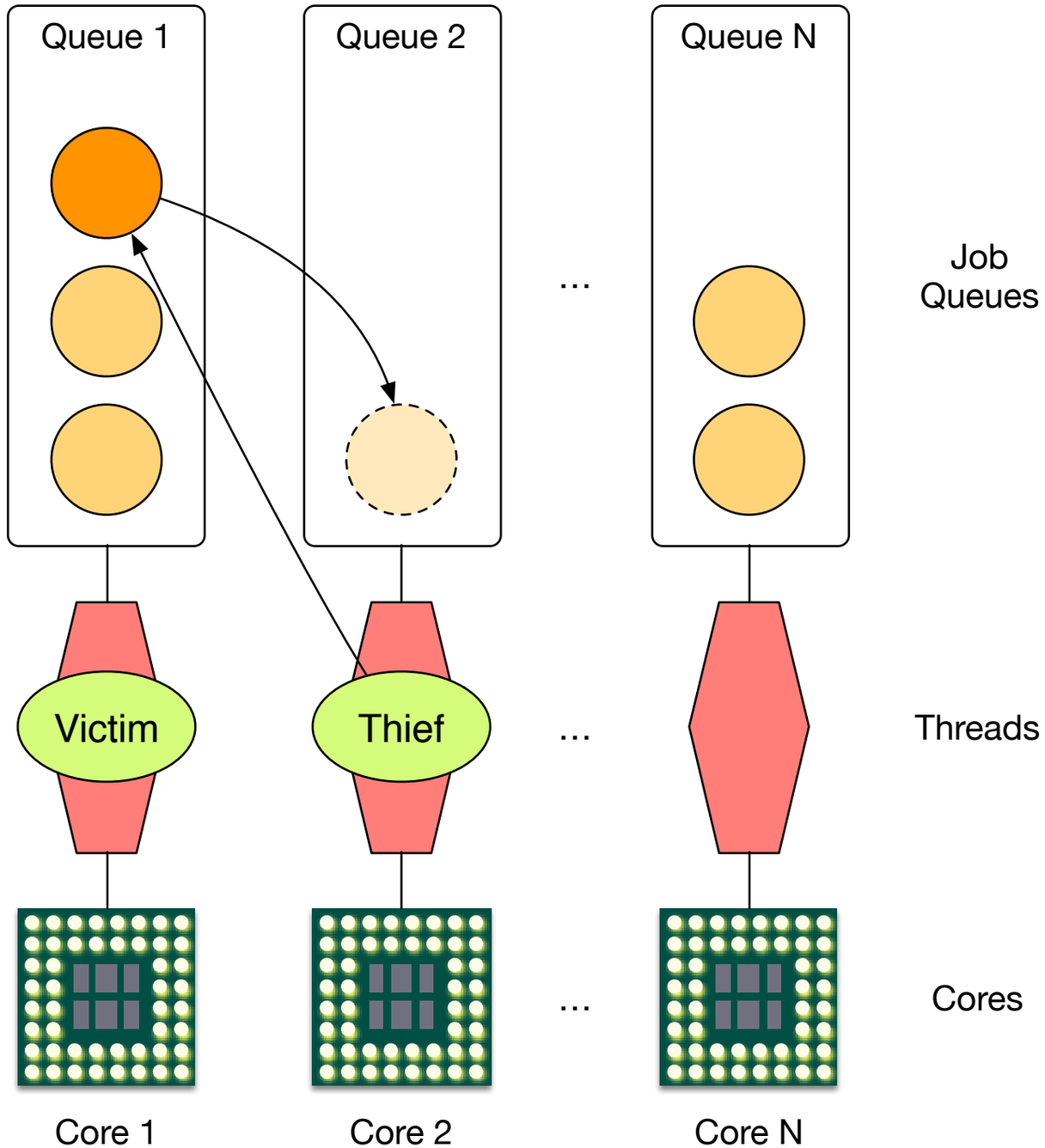
- **Decentralized**: one job queue and worker thread per core
- On empty queue, **steal** from other thread
- Efficient if stealing is a rare event
- Implementation: deque with two spinlocks



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

Work 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
- **Graceful downscaling** of polling
  - **100** (0) → **500** (10µs) → **∞** (10ms)

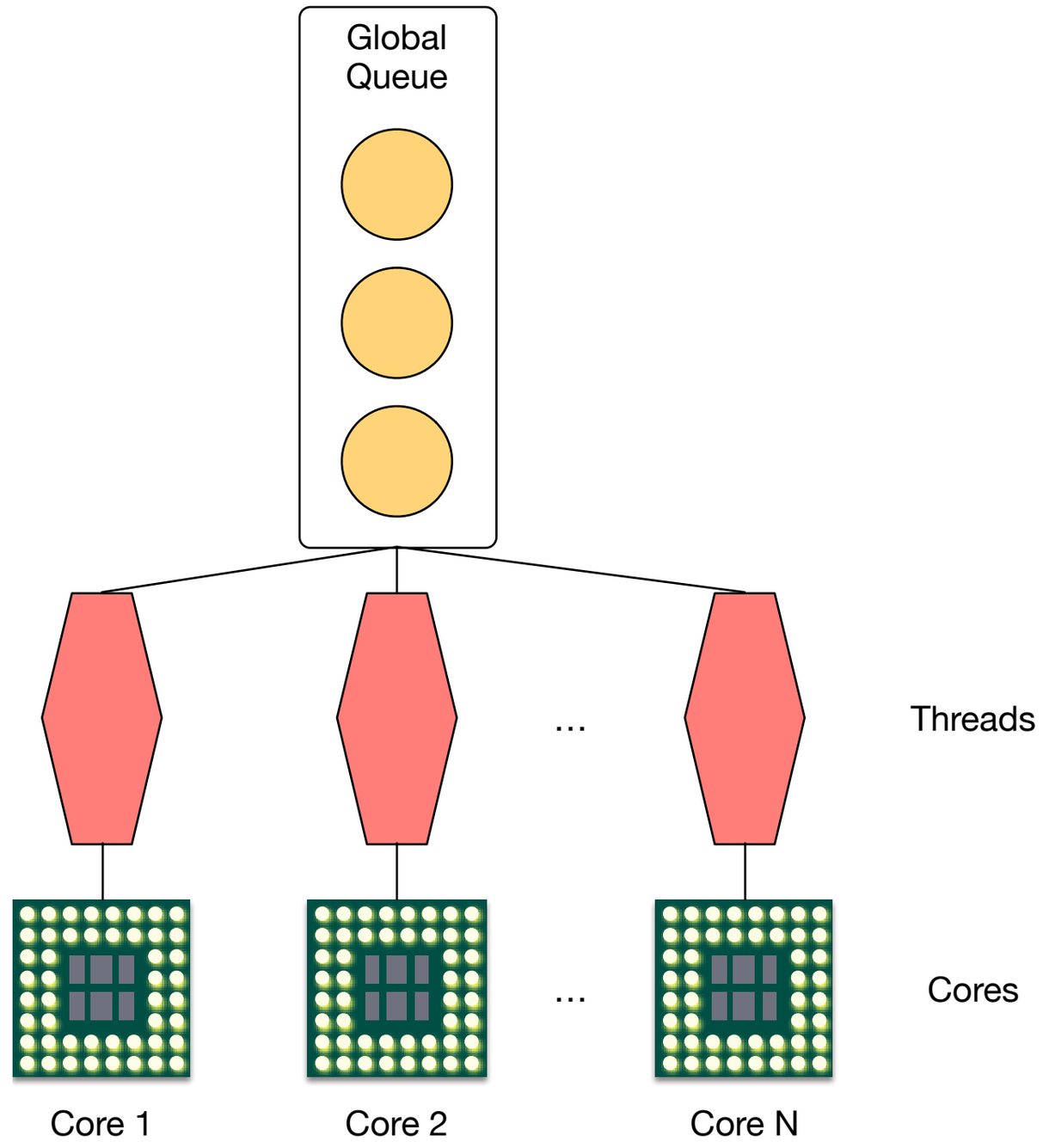


*Robert D. Blumofe and Charles E. Leiserson. *Scheduling Multithreaded Computations by Work Stealing*. J. ACM, 46(5):720–748, September 1999.

Implementation

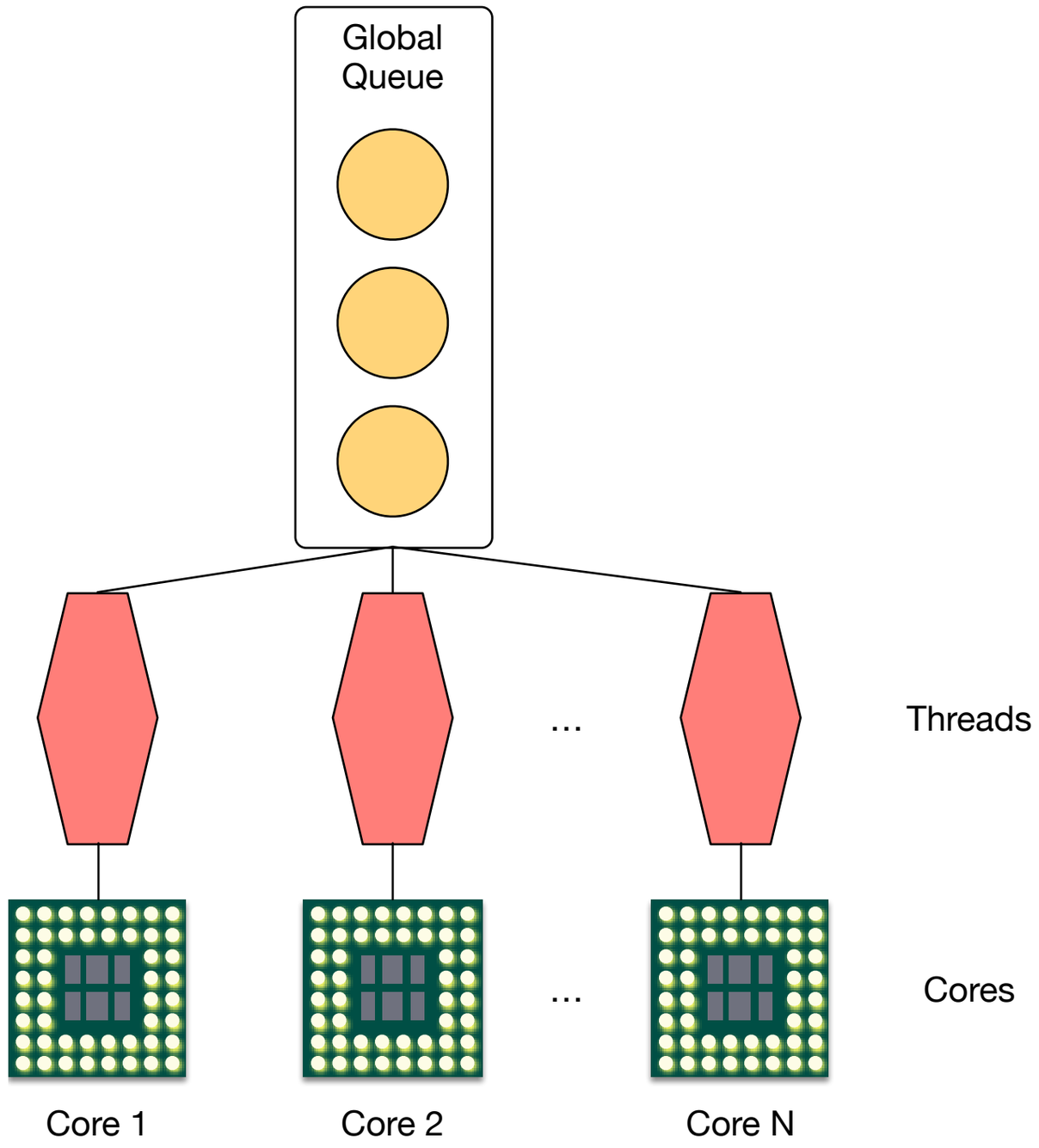
```
template <class Worker>
resumable* dequeue(Worker* self) {
  auto& strategies = self->data().strategies;
  resumable* job = nullptr;
  for (auto& strat : strategies) {
    for (size_t i = 0; i < strat.attempts; i += strat.step_size) {
      // try to grab a job from the front of the queue
      job = 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



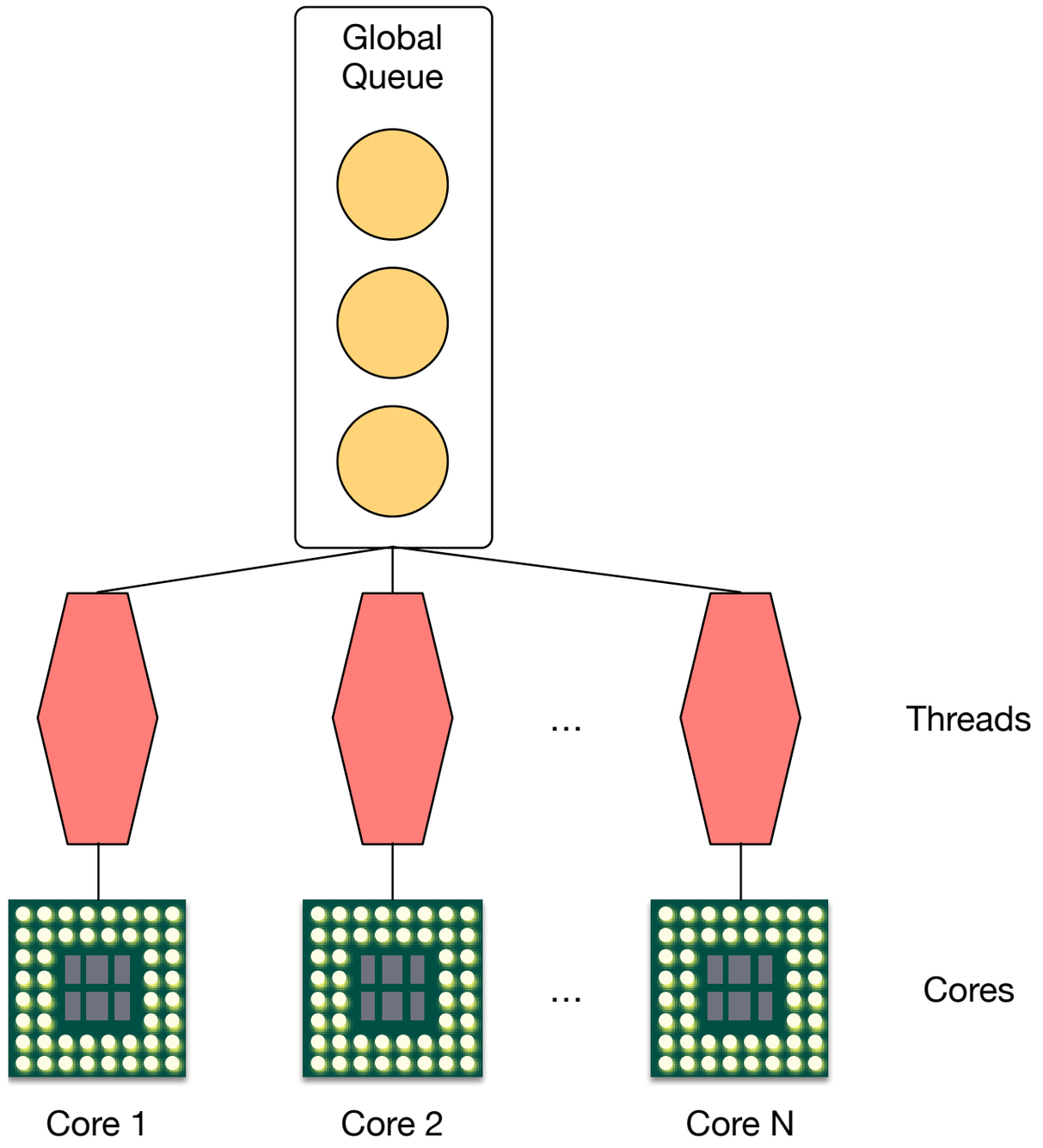
Work Sharing

- **Centralized**: one shared global queue



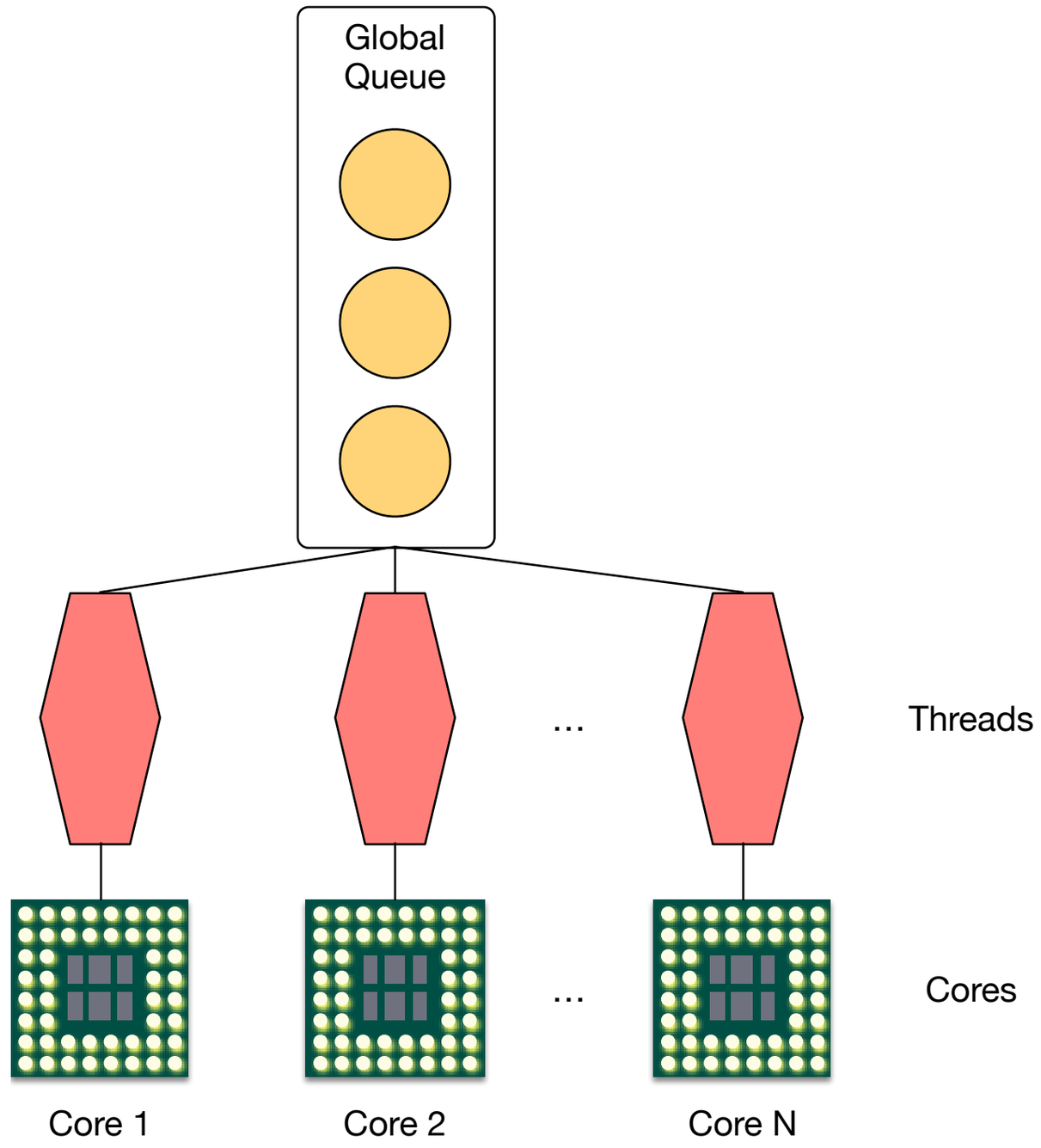
Work Sharing

- **Centralized**: one shared global queue
- Synchronization: **mutex & CV**



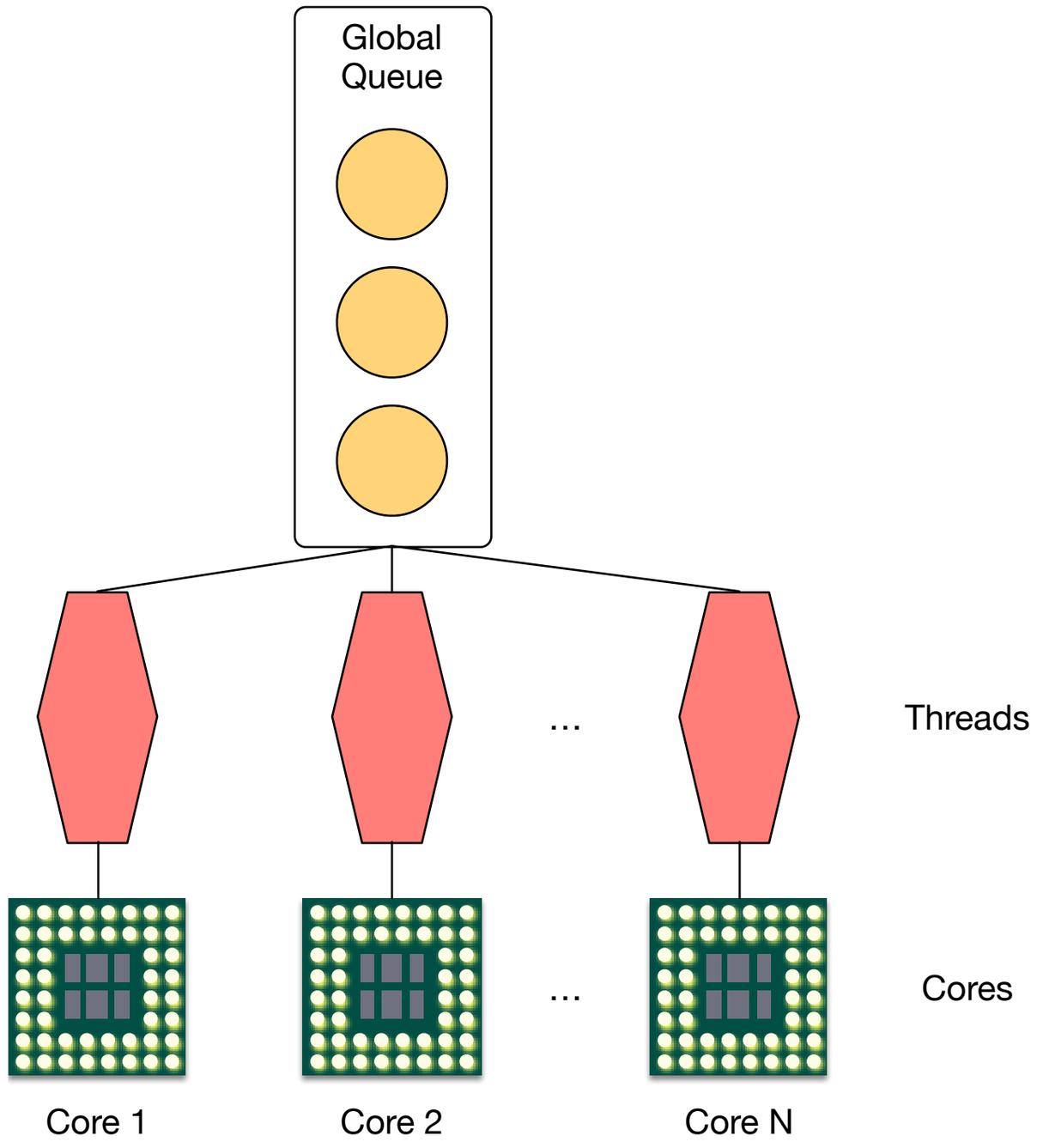
Work Sharing

- **Centralized**: one shared global queue
- Synchronization: **mutex & CV**
- **No polling**



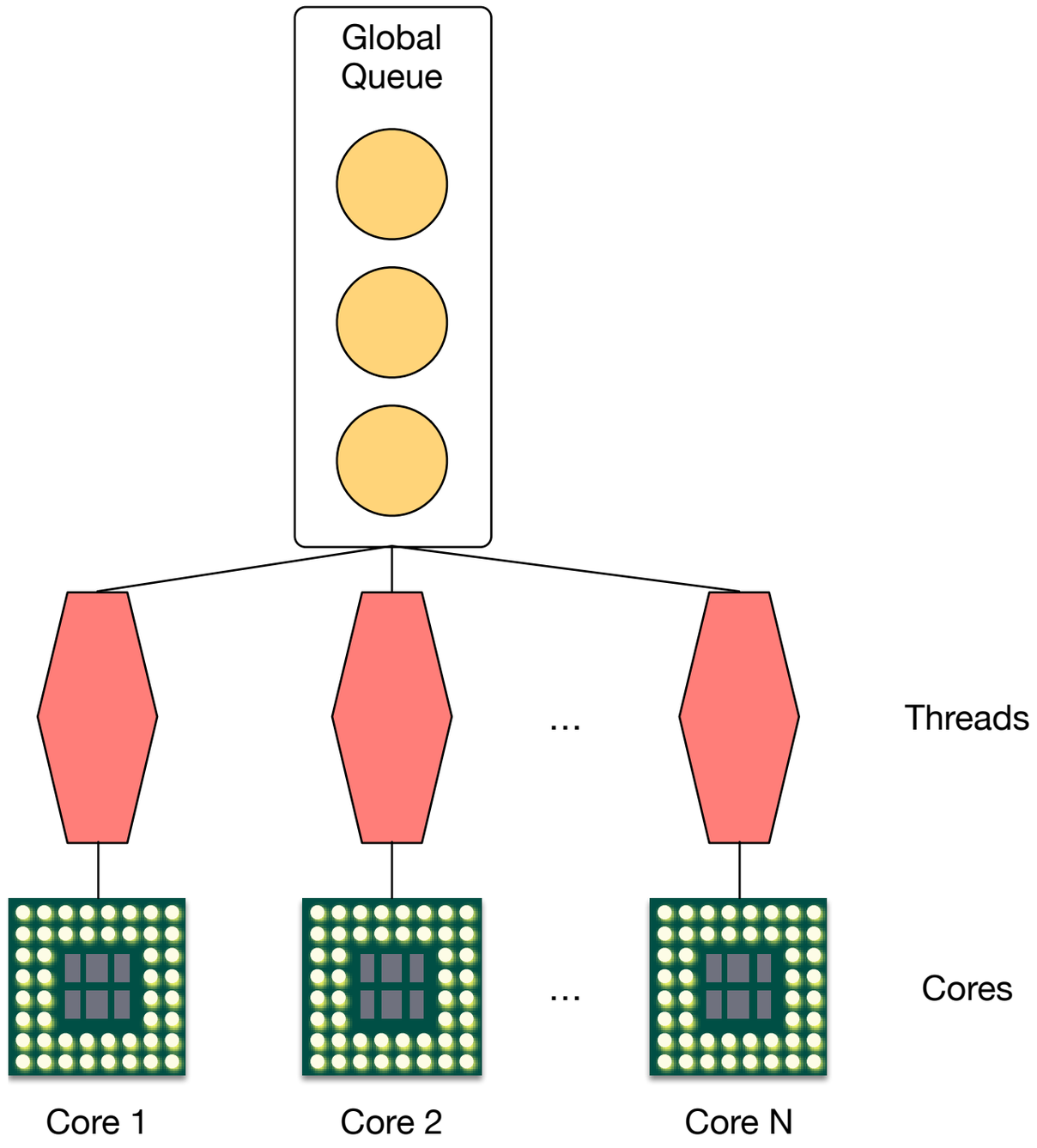
Work Sharing

- **Centralized**: one shared global queue
- Synchronization: **mutex & CV**
- **No polling**
  - less CPU usage
  - lower throughouput



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

- **Message** = atomic, intrusive reference-counted tuple

- **Message** = atomic, intrusive reference-counted tuple

- **Immutable access** permitted

- **Message** = atomic, intrusive reference-counted tuple
- **Immutable access** permitted
- **Mutable access** with reference count > 1 invokes copy constructor

- **Message** = atomic, intrusive reference-counted tuple
- **Immutable access** permitted
- **Mutable access** with reference count > 1 invokes copy constructor
- **Constness deduced** from message handlers

- **Message** = atomic, intrusive reference-counted tuple
- **Immutable access** permitted
- **Mutable access** with reference count > 1 invokes copy constructor

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

- **Constness deduced** from message handlers

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

- **Message** = atomic, intrusive reference-counted tuple
- **Immutable access** permitted
- **Mutable access** with reference 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)
  send(r, msg);

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

const access enables efficient sharing of messages

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

- **Message** = atomic, intrusive reference-counted tuple
- **Immutable access** permitted
- **Mutable access** with reference 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)
  send(r, msg);

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

`const` access enables efficient sharing of messages

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

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

- **Message** = atomic, intrusive reference-counted tuple
- **Immutable access** permitted
- **Mutable access** with reference count > 1 invokes copy constructor

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

- **Constness deduced** from message handlers
- **No data races** by design

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

- **Message** = atomic, intrusive reference-counted tuple
- **Immutable access** permitted
- **Mutable access** with reference count > 1 invokes copy constructor

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

- **Constness deduced** from message handlers
- **No data races** by design
- **Value semantics**, no complex lifetime management

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

Type Safety

- CAF has **statically** and **dynamically typed** actors

- CAF has **statically** and **dynamically typed** actors
- **Dynamic**
  - Type-erased `caf::message` hides tuple types
  - Good for rapid prototyping

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

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

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

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);
    }
  };
}
```

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) {

  }
);
```

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

Class-Based API

Class-Based API

Dynamic

```
class math : public event_based_actor {
public:
  behavior make_behavior() override {
    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);
      }
    };
  }
};
```

Class-Based API

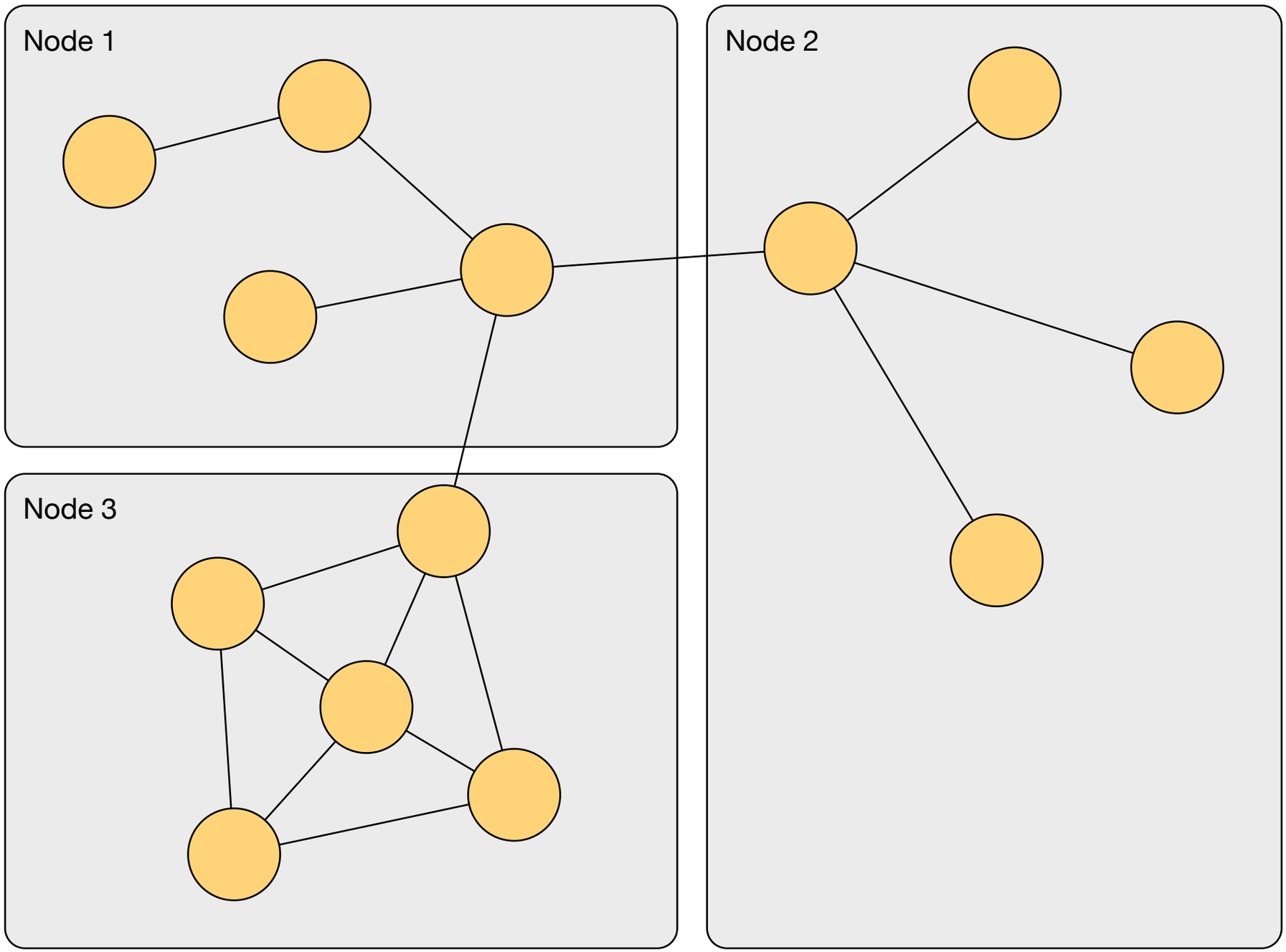
Dynamic

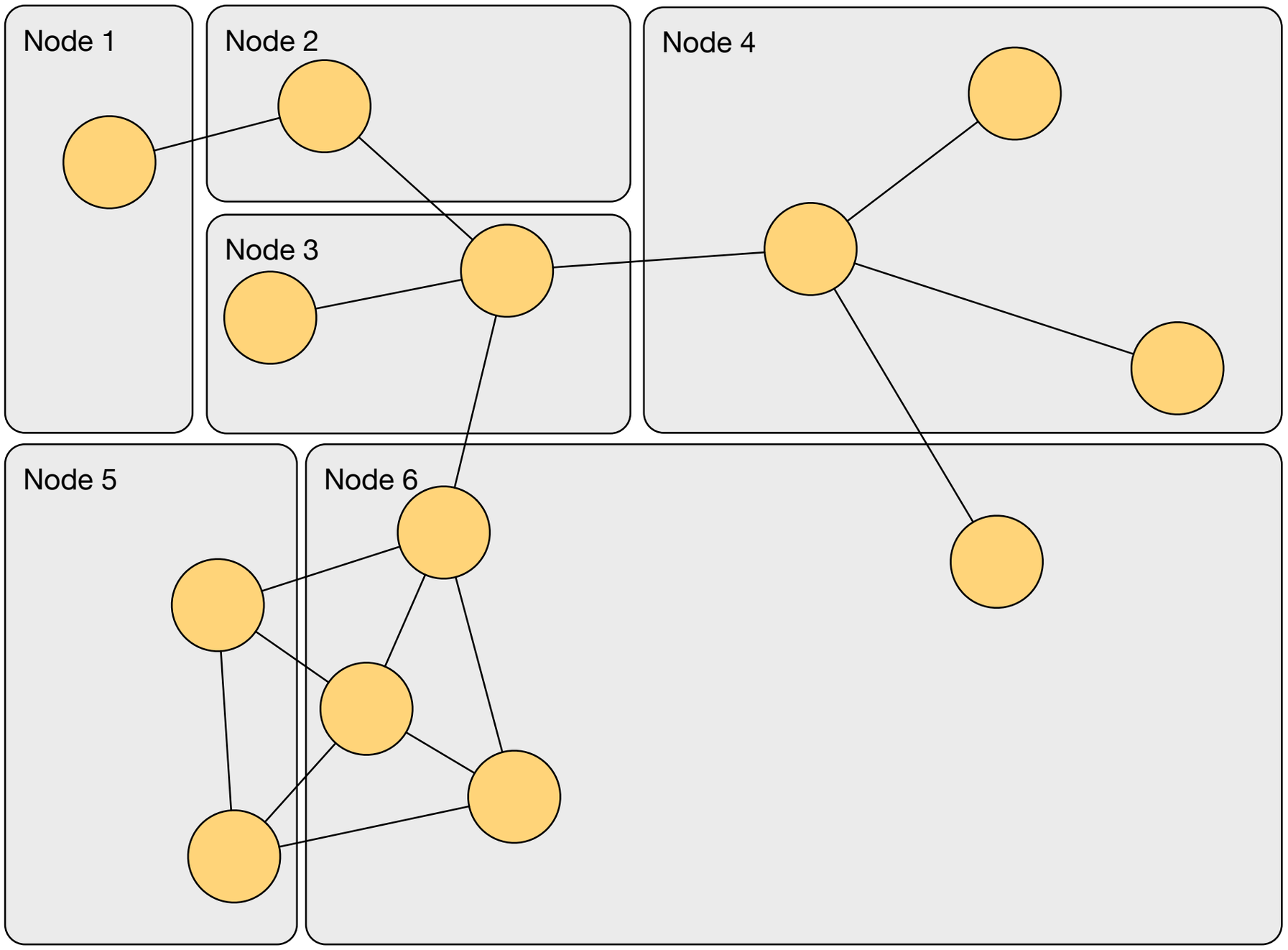
```
class math : public event_based_actor {
public:
  behavior make_behavior() override {
    return {
      [](plus_atom, int a, int b) {
        return make_tuple(result_atom::value, a + b);
      },
      [](minus_atom, int a, int b) {
        return make_tuple(result_atom::value, a - b);
      }
    };
  }
};
```

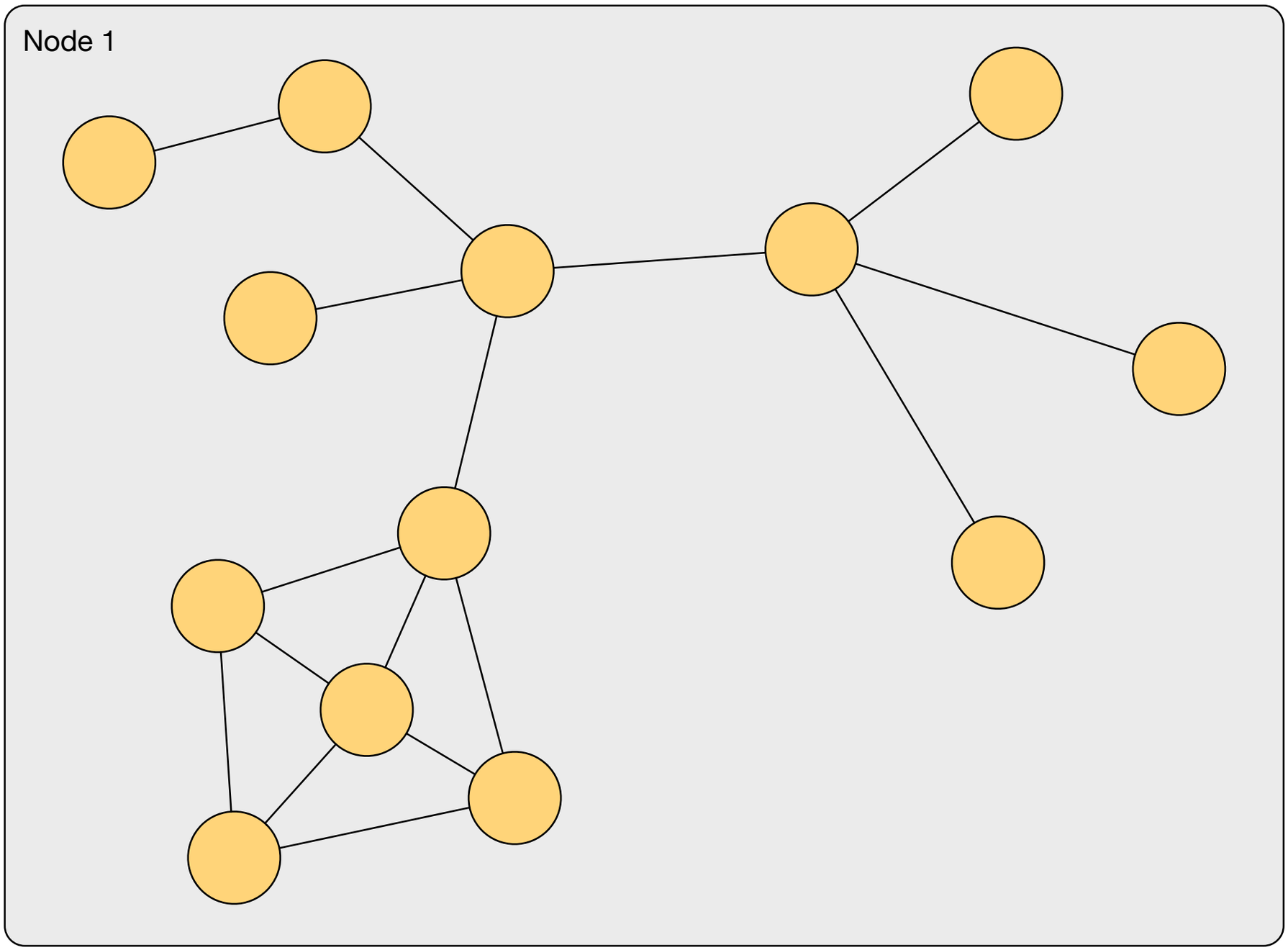
Static

```
class typed_math : public math_actor::base {
public:
  behavior_type make_behavior() override {
    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);
      }
    };
  }
};
```

Network Transparency


Node 1
Node 2
Node 3


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


Node 1

Network Transparency

Network Transparency

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

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**

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
}
```

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.

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.

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

Failure Management

Failure Management

Components fail regularly in large-scale systems

Failure Management

Components fail regularly in large-scale systems

- Actor model provides **monitors** and **links**

Failure Management

Components fail regularly in large-scale systems

- Actor model provides **monitors** and **links**
  - **Monitor**: subscribe to exit of actor (**unidirectional**)

Failure Management

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**)

Failure Management

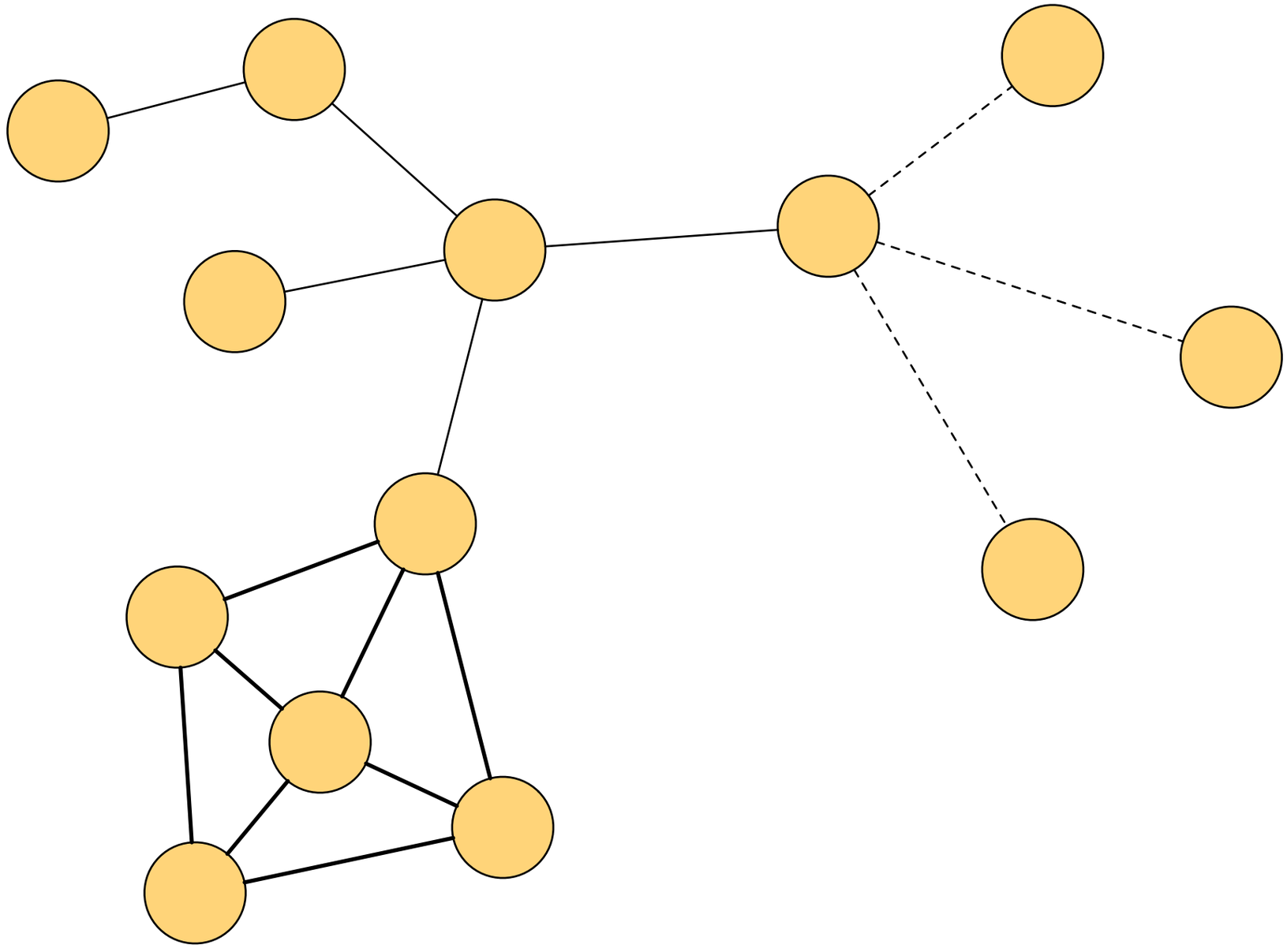
Components fail regularly in large-scale systems

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

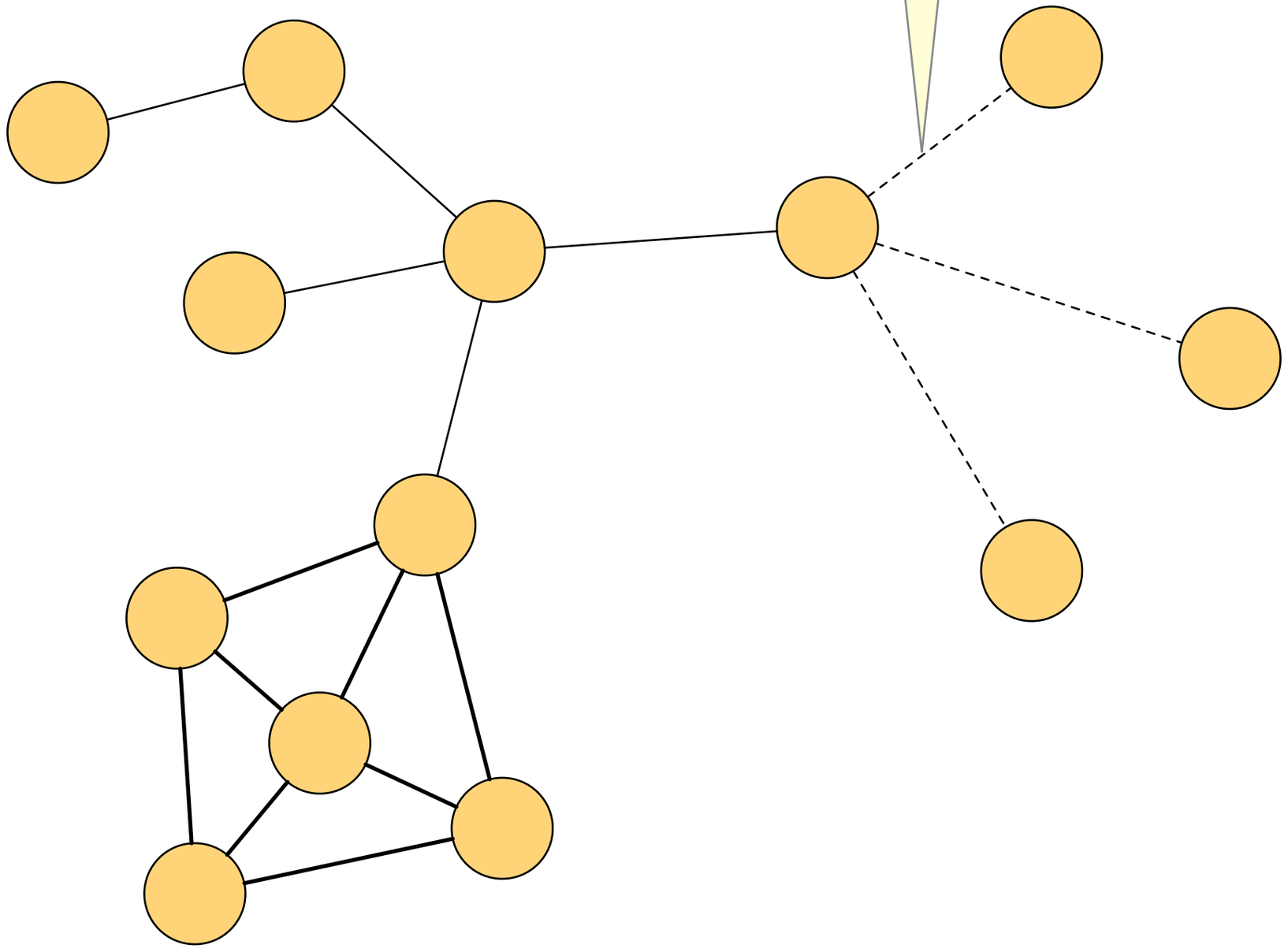
Failure Management

Components fail regularly in large-scale systems

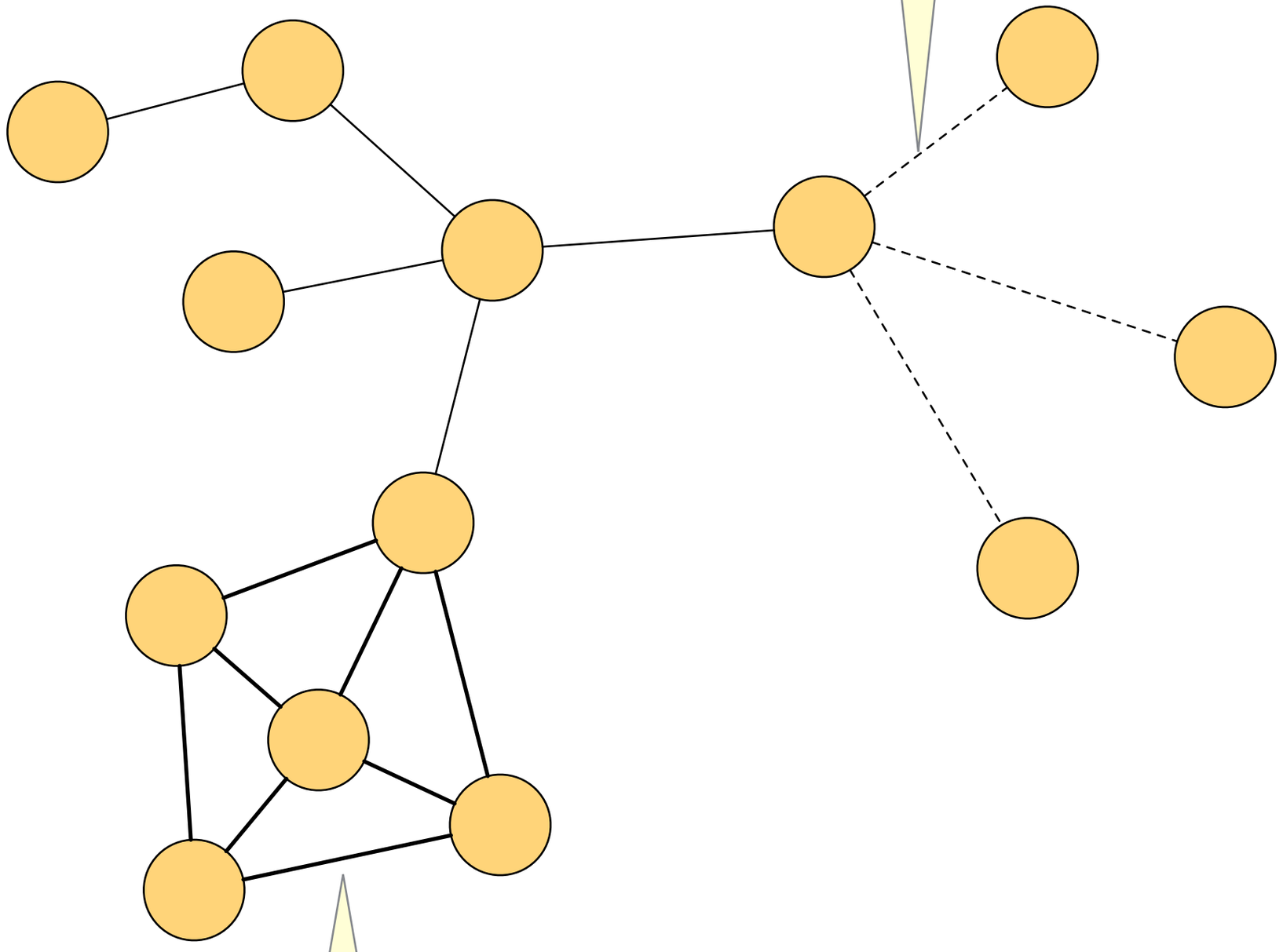
- Actor model provides **monitors** and **links**
  - **Monitor**: subscribe to exit of actor (**unidirectional**)
  - **Link**: bind own lifetime to other actor (**bidirectional**)
- Enables **hierarchical propagation** of failures
- Can define **local fault isolation** domains



Monitored actors: subscribe to notification if actor terminates



Monitored actors: subscribe to notification if actor terminates



Linked actors: either **all alive** or **collectively failing**

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;
  }
);
```

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;
  }
);
```

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);`**

Project Summary

Project Summary

- Lead: **Dominik Charousset** (HAW Hamburg)
  - Started CAF as Master's thesis
  - Active development as part of his Ph.D.

Project Summary

- 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**

Project Summary

- 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)

Project Summary

- 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**

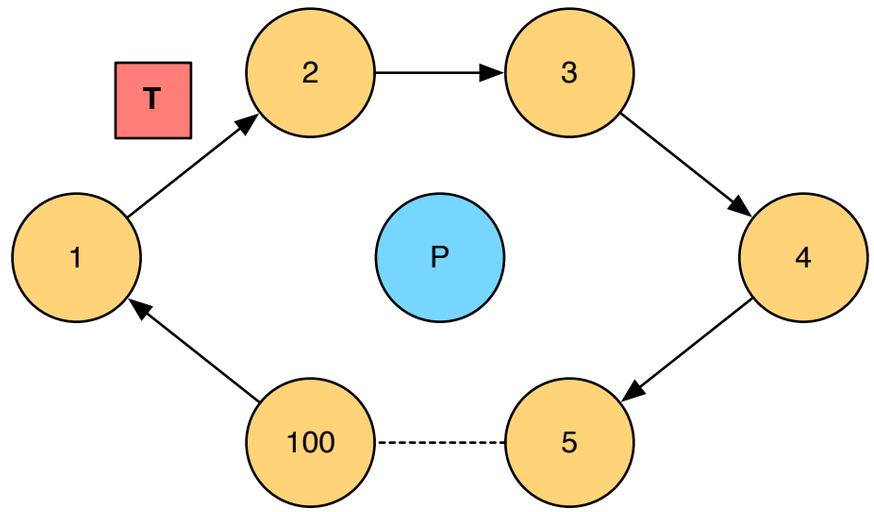
Project Summary

- 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

Evaluation

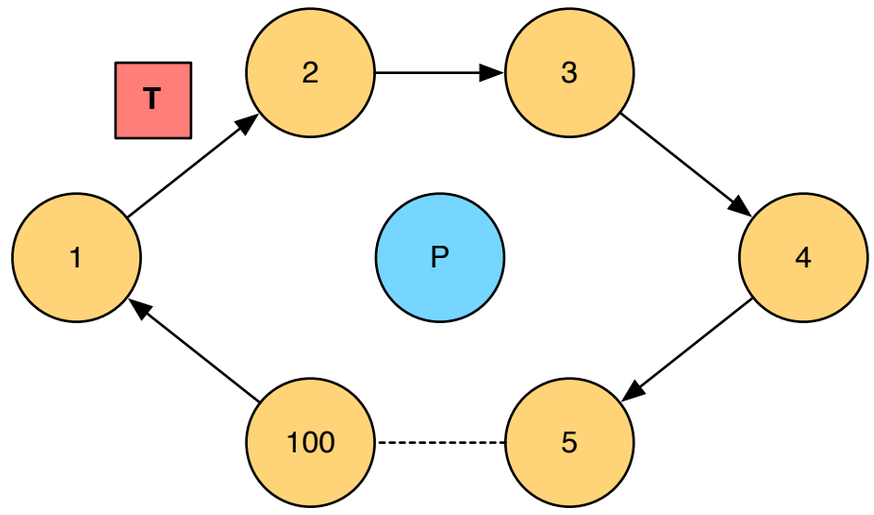
https://github.com/actor-framework/benchmarks

Setup #1

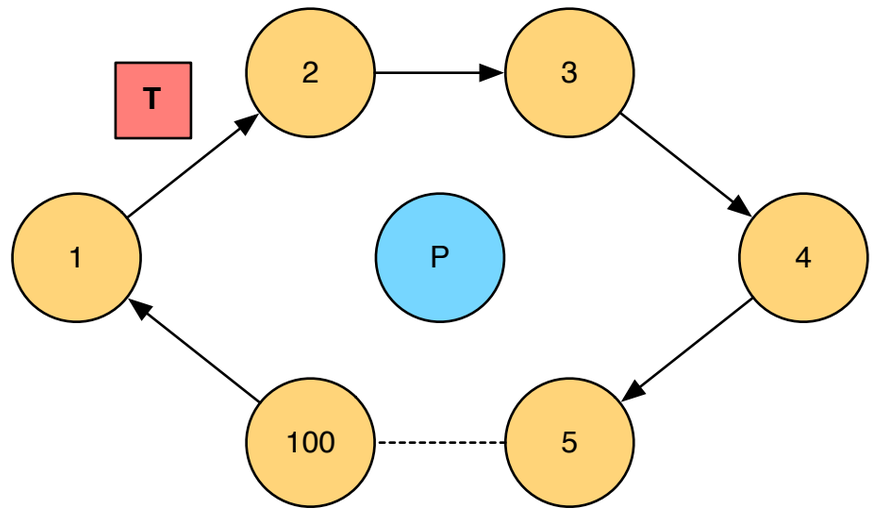


Setup #1

- 100 rings of 100 actors each

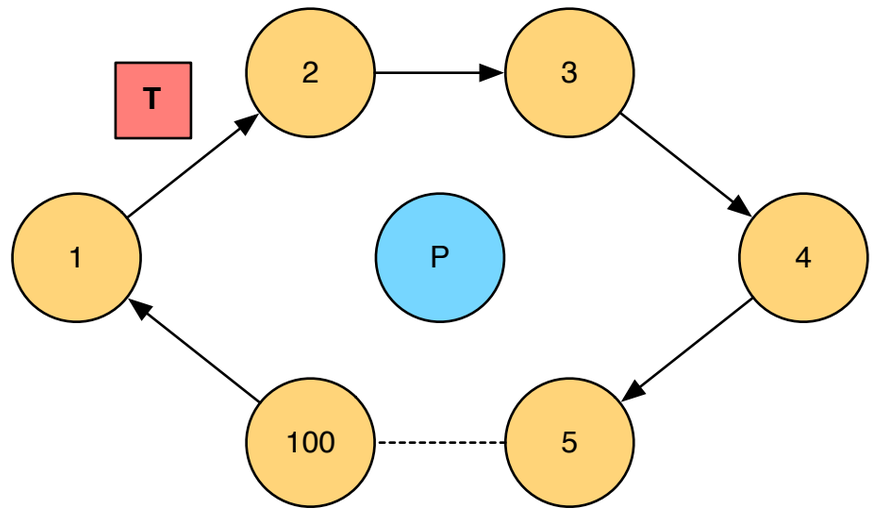


Setup #1



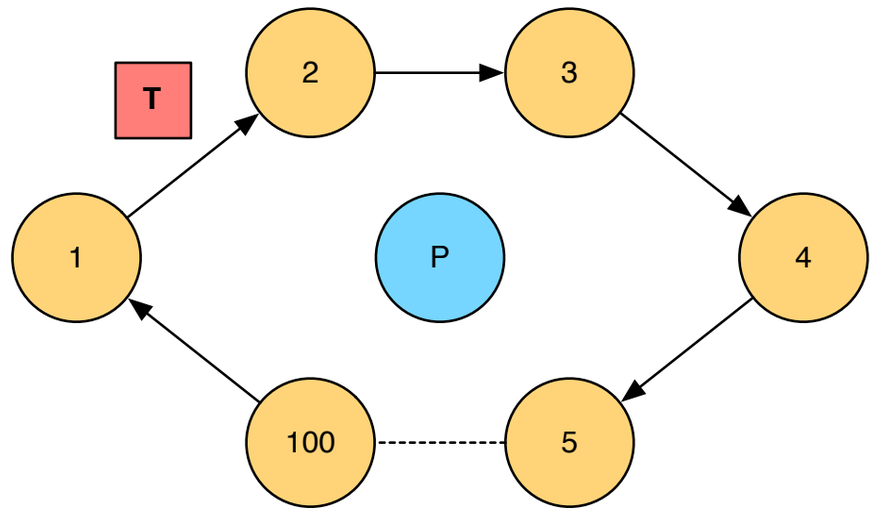
- 100 rings of 100 actors each
- Actors forward single token 1K times, then terminate

Setup #1



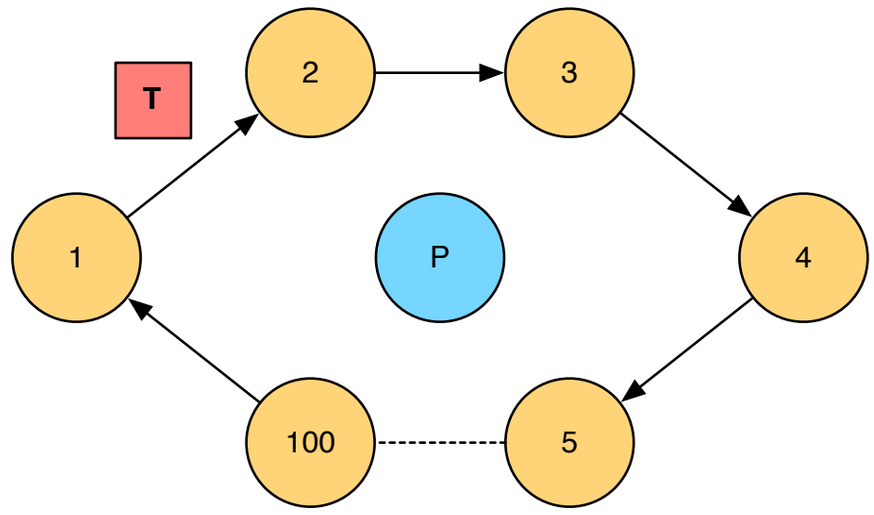
- 100 rings of 100 actors each
- Actors forward single token 1K times, then terminate
- 4 re-creations per ring

Setup #1



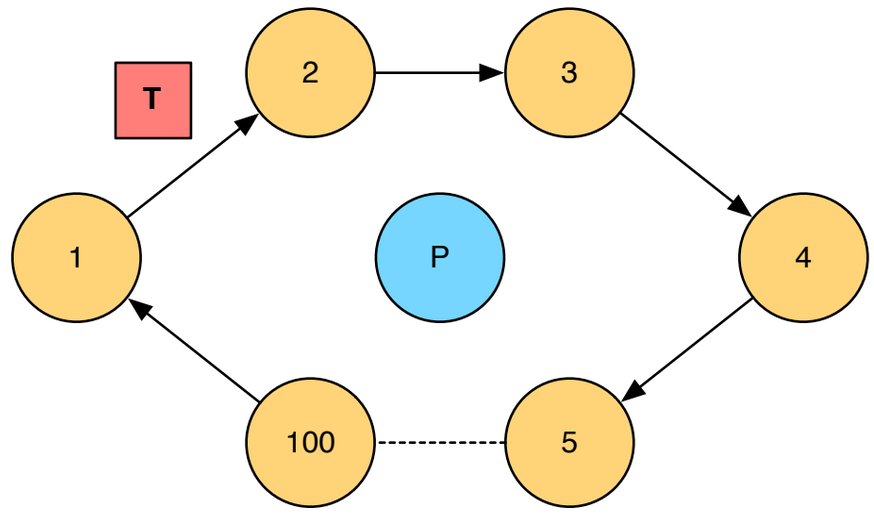
- 100 rings of 100 actors each
- Actors forward single token 1K times, then terminate
- 4 re-creations per ring
- One actor per ring performs *prime factorization*

Setup #1



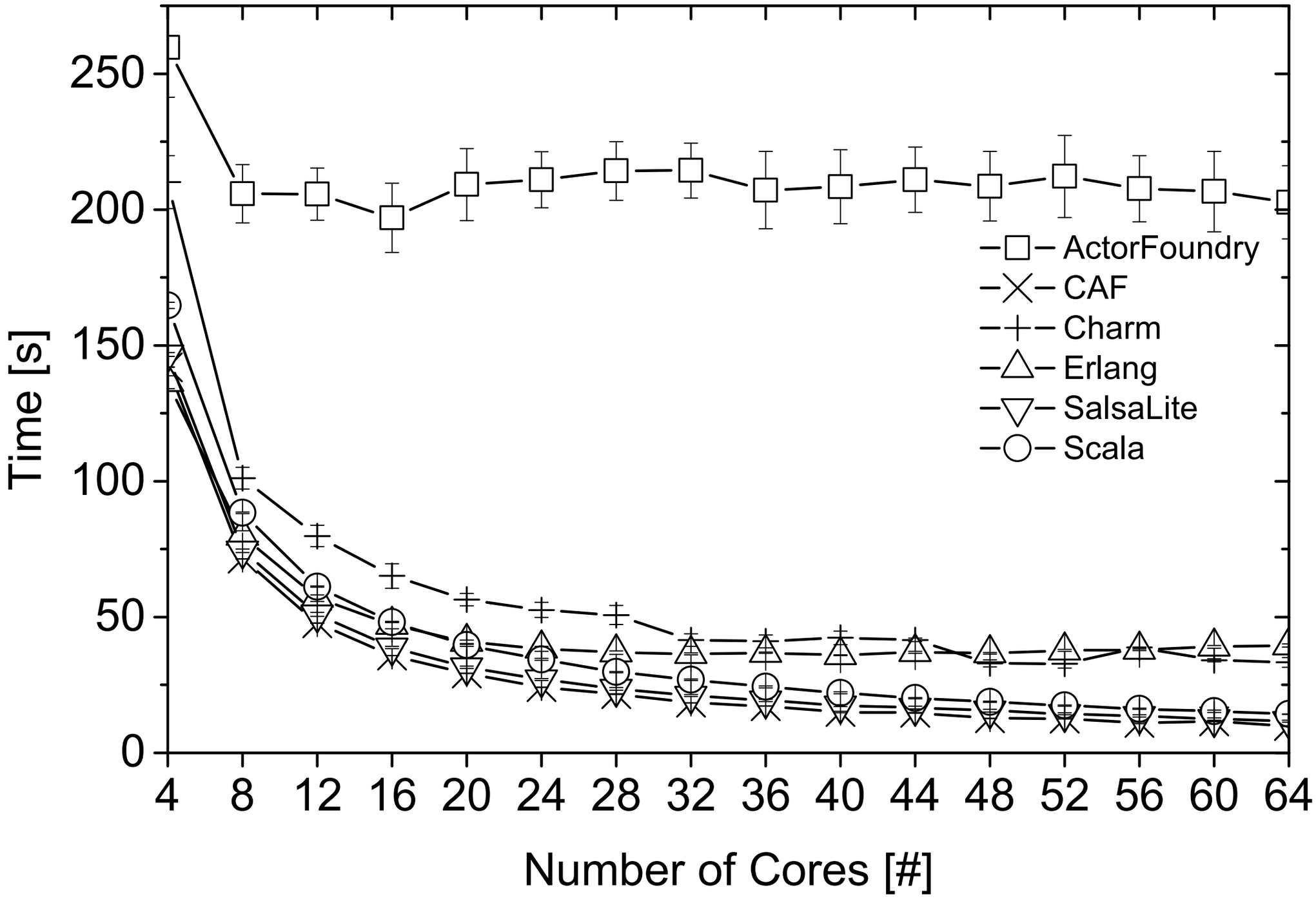
- 100 rings of 100 actors each
- Actors forward single token 1K times, then terminate
- 4 re-creations per ring
- One actor per ring performs *prime factorization*
- Resulting workload: high message & CPU pressure

Setup #1

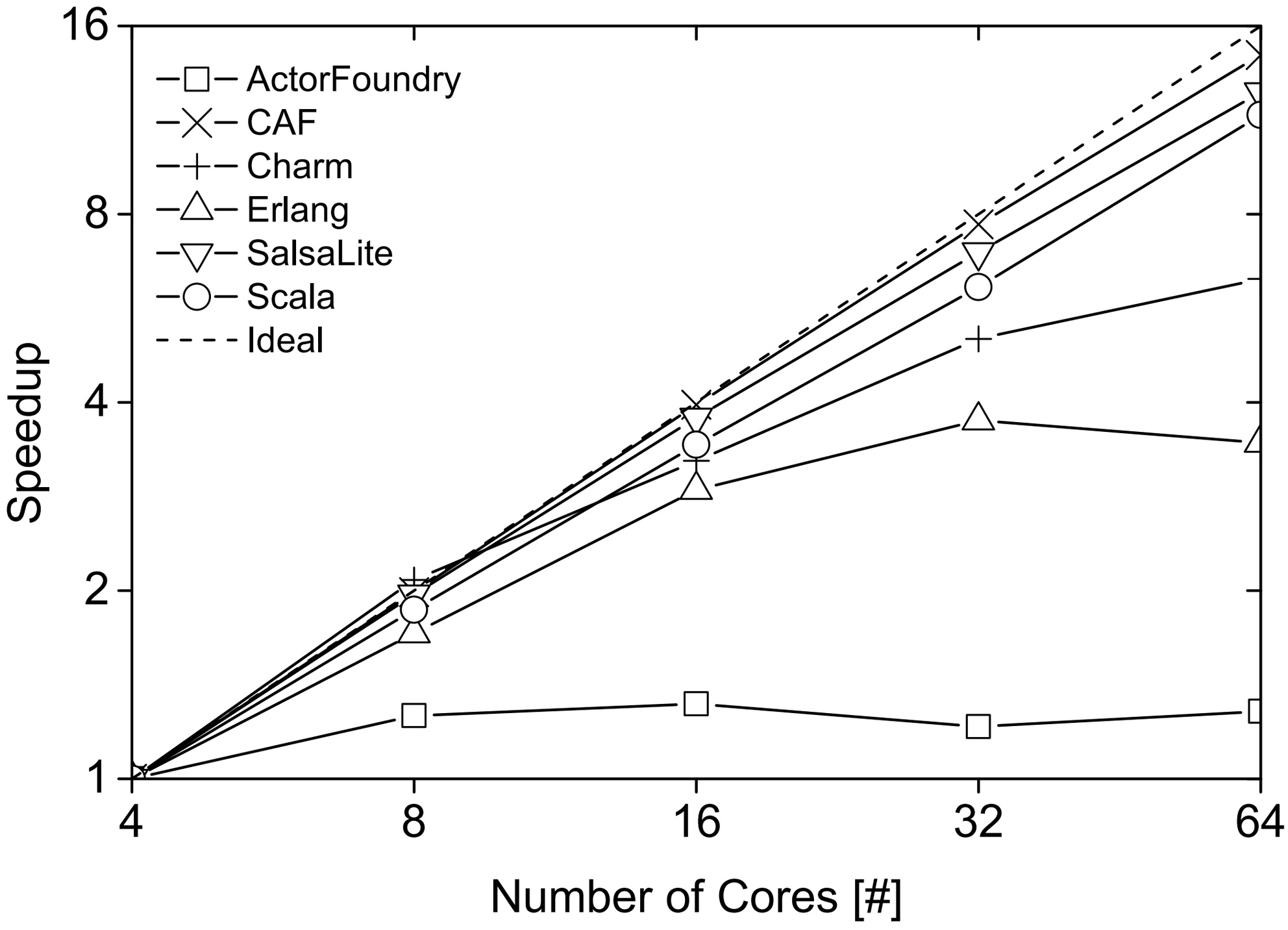


- 100 rings of 100 actors each
- Actors forward single token 1K times, then terminate
- 4 re-creations per ring
- One actor per ring performs *prime factorization*
- Resulting workload: high message & CPU pressure
- Ideal: 2 x cores ⟹ 0.5 x runtime

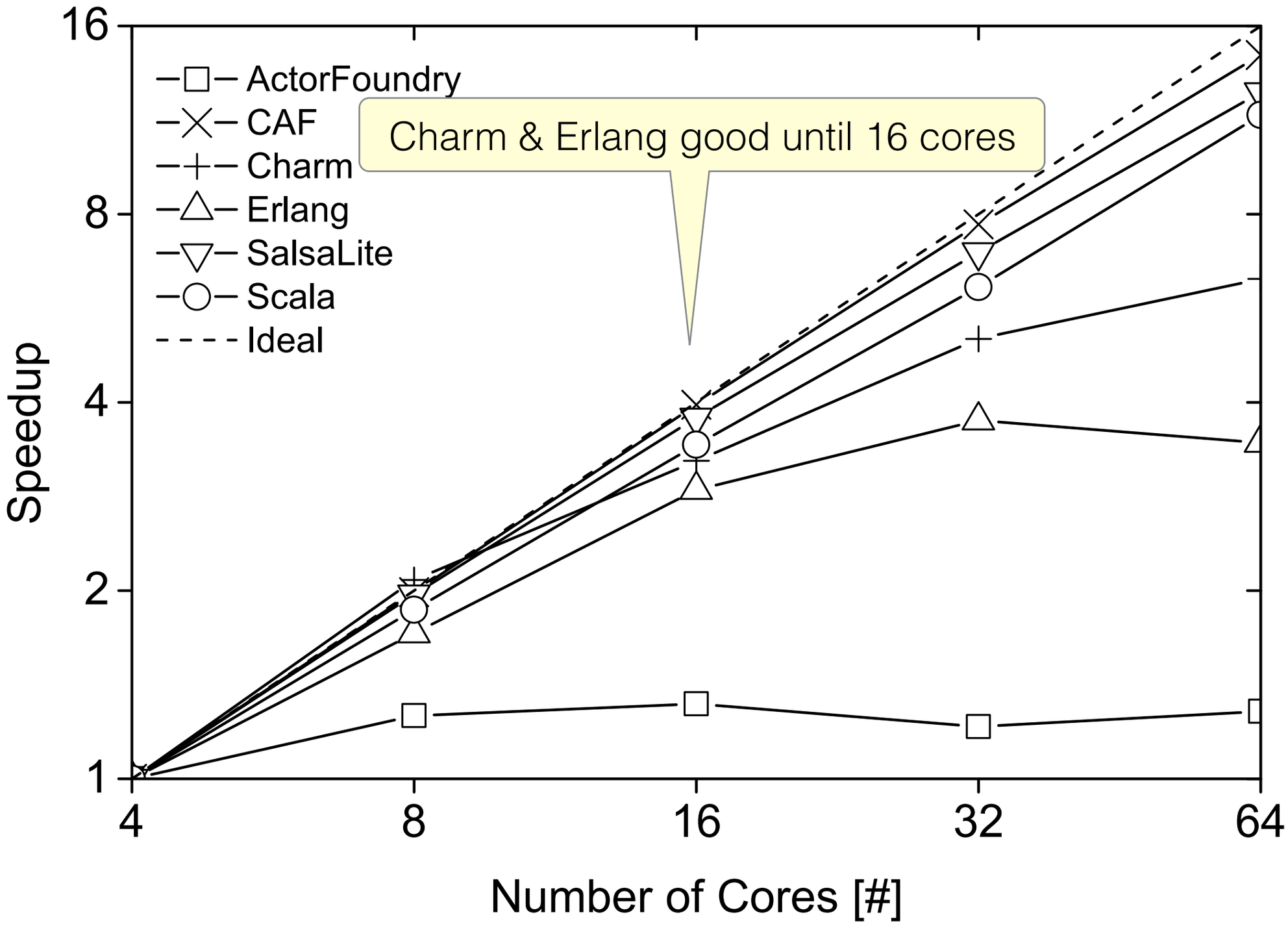
Performance



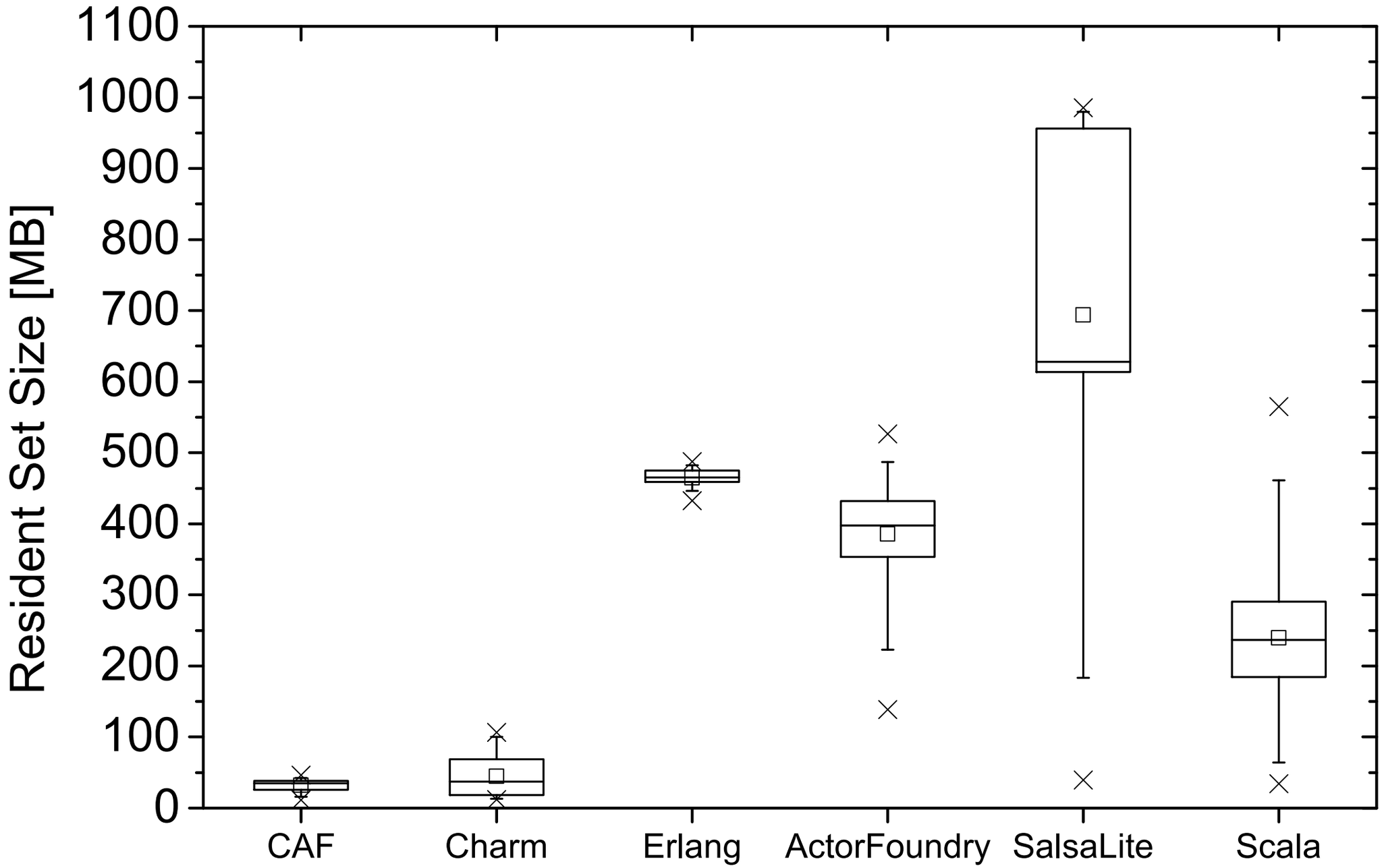
(normalized)


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

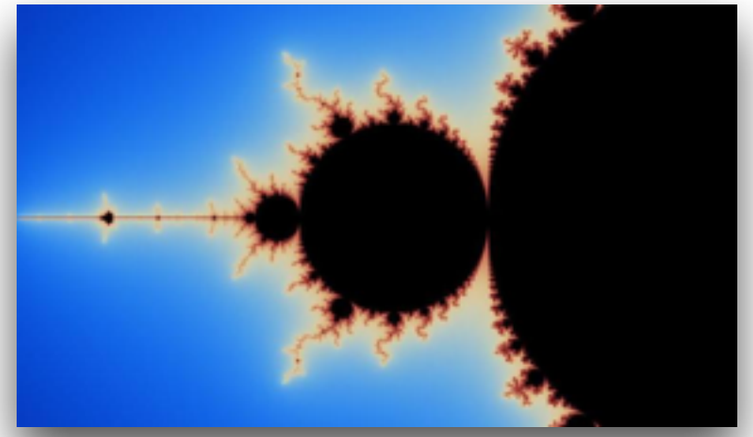
(normalized)


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

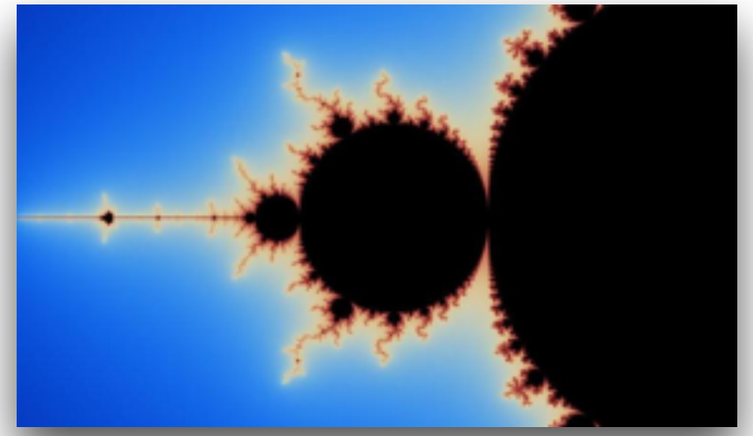


Setup #2

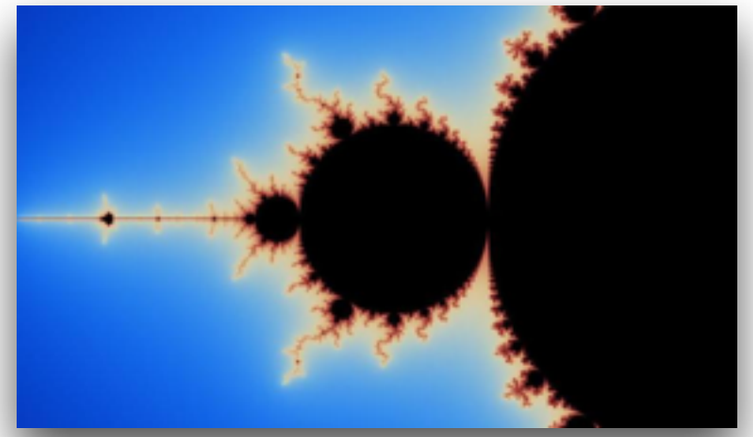


Setup #2

- Compute images of Mandelbrot set

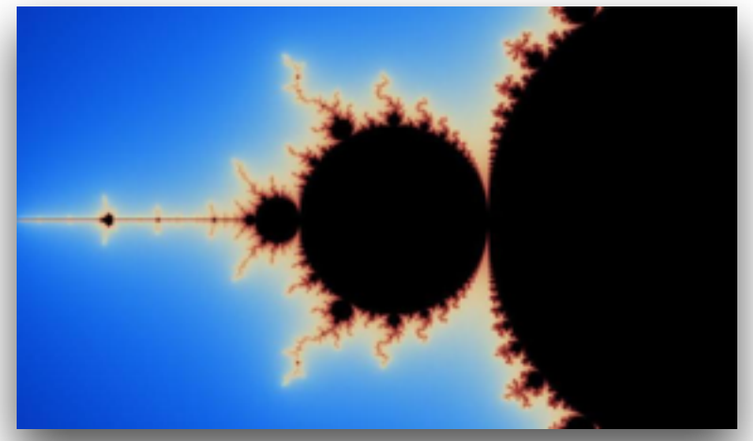


Setup #2



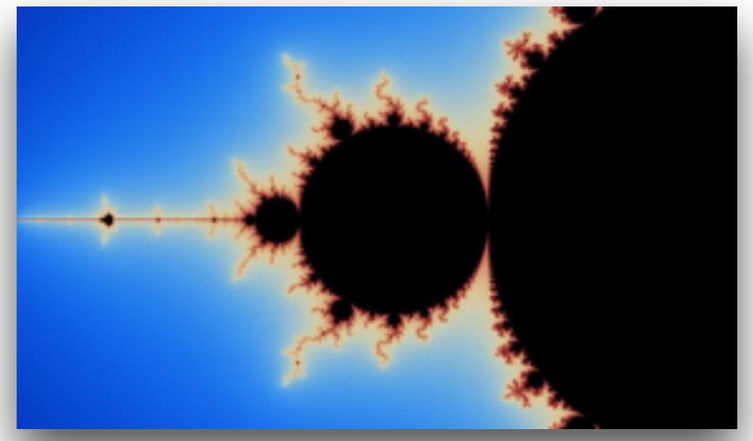
- Compute images of Mandelbrot set
- Divide & conquer algorithm

Setup #2



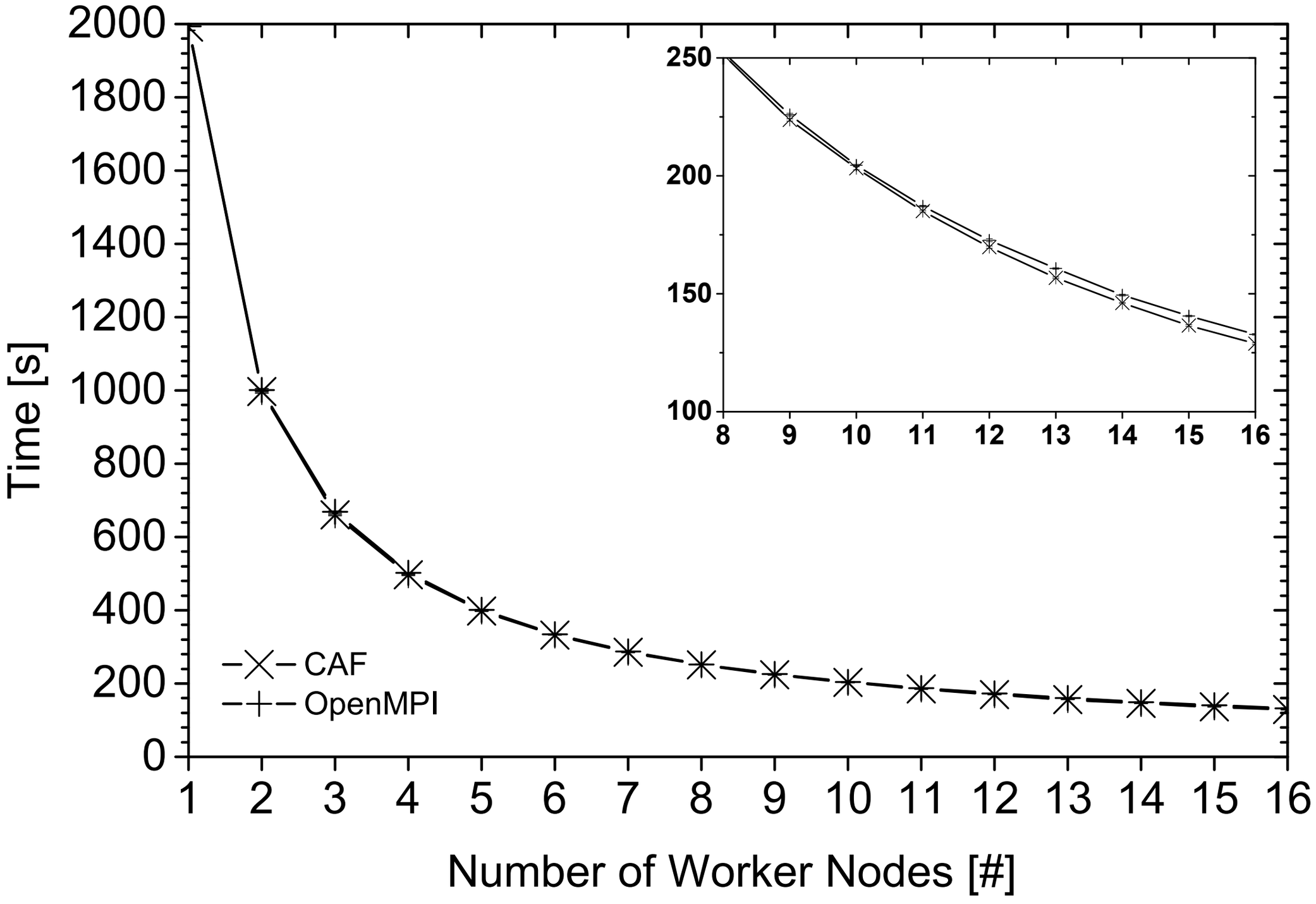
- Compute images of Mandelbrot set
- Divide & conquer algorithm
- Compare against OpenMPI (via Boost.MPI)
  - Only message passing layers differ

Setup #2

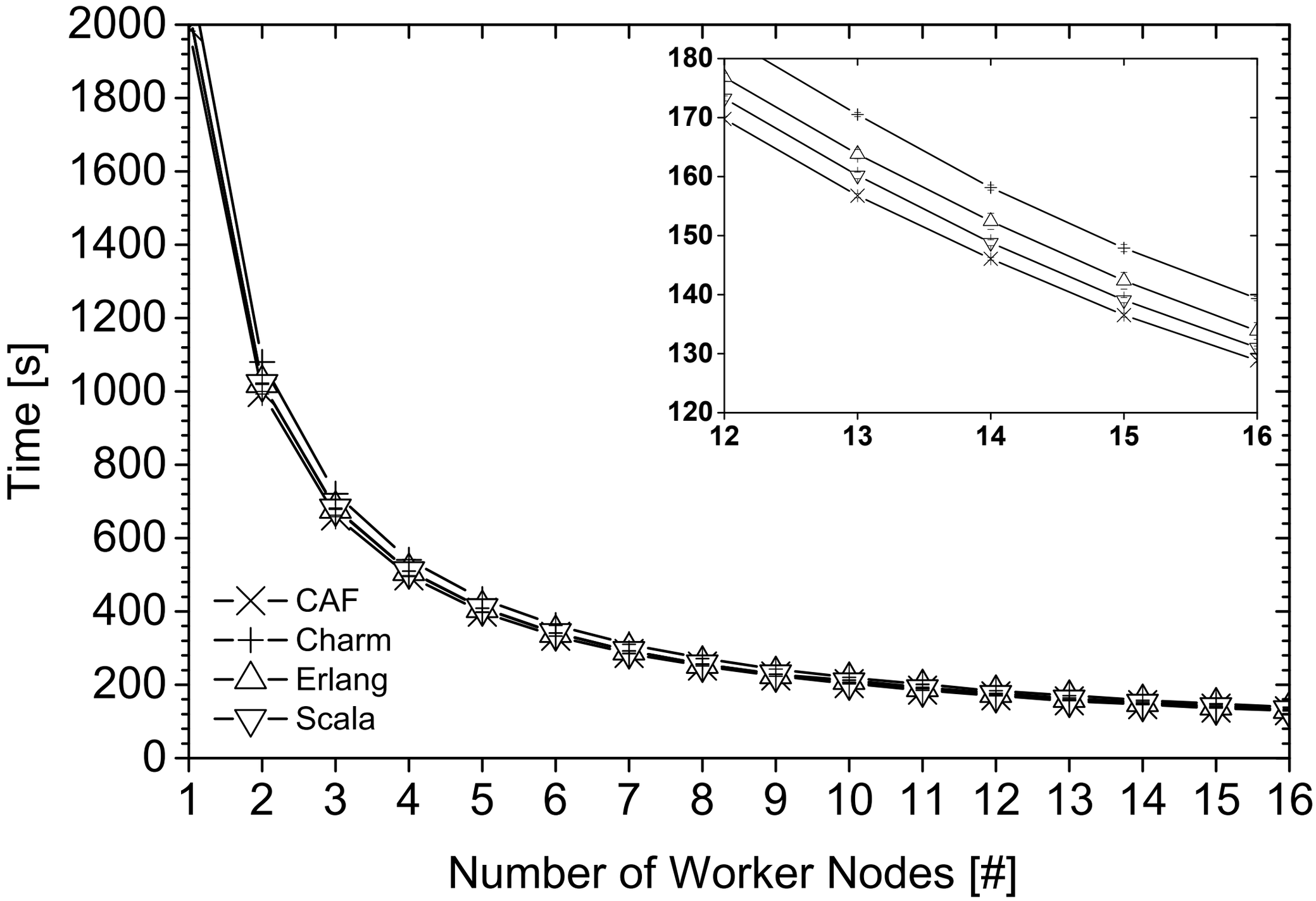


- Compute images of Mandelbrot set
- Divide & conquer algorithm
- Compare against OpenMPI (via Boost.MPI)
  - Only message passing layers differ
- 16-node cluster: quad-core Intel i7 3.4 GHz

CAF vs. OpenMPI



Scalability



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

Backup Slides

Sending Messages

- Asynchronous fire-and-forget

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

- Request-response with one-shot handler

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

- Transparent forwarding of message forwarding

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

A Simple Actor

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

A Simple Actor

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

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

A Simple Actor

Optional reference to the running actor, e.g., to send messages in response handlers.

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

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

A Simple Actor

An actor is typically implemented as function

Optional reference to the running actor, e.g., to send messages in response handlers.

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

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

A Simple Actor

An actor is typically implemented as function

Optional reference to the running actor, e.g., to send messages in response handlers.

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

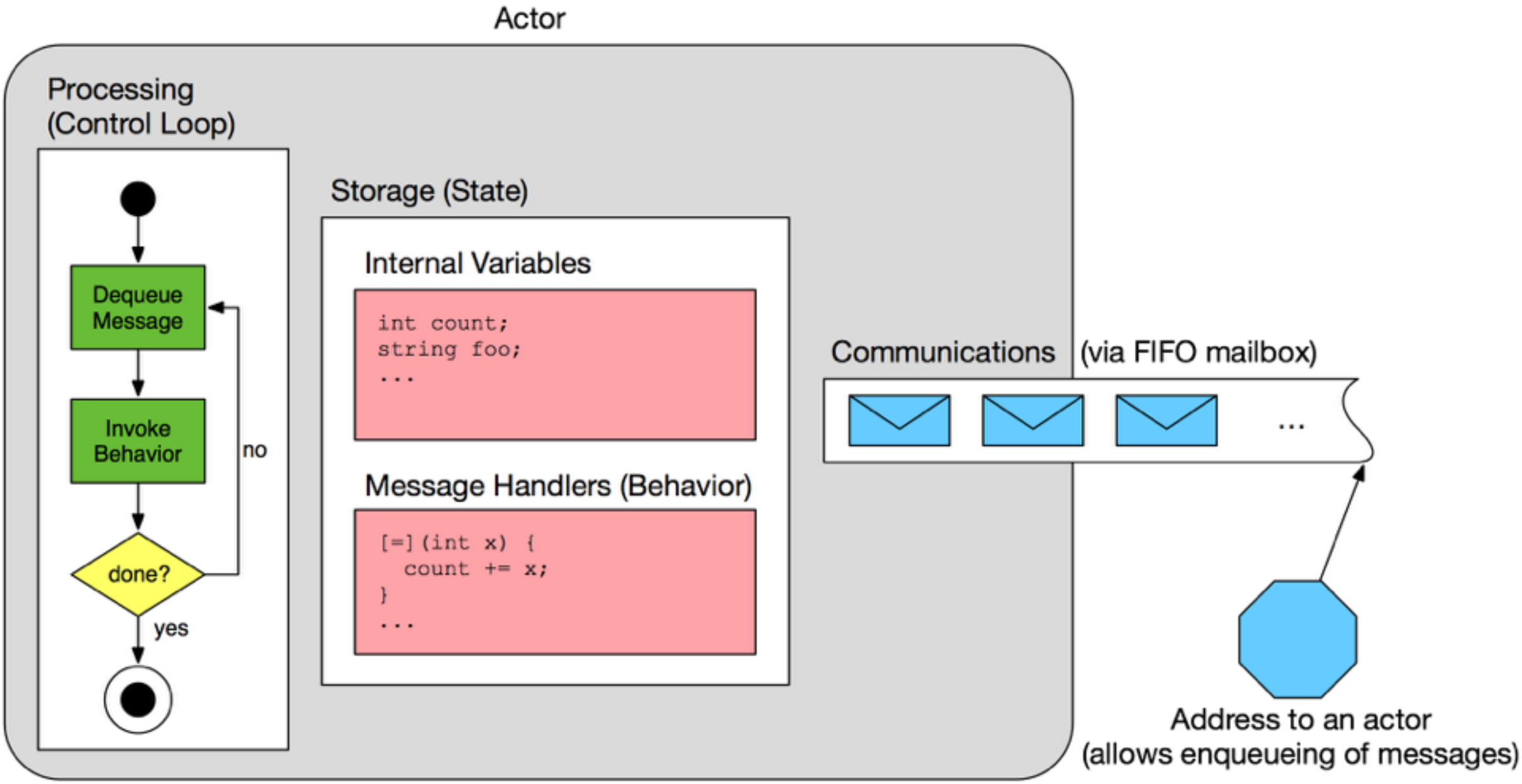
A list of lambda functions determines the message types an actor processes.

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

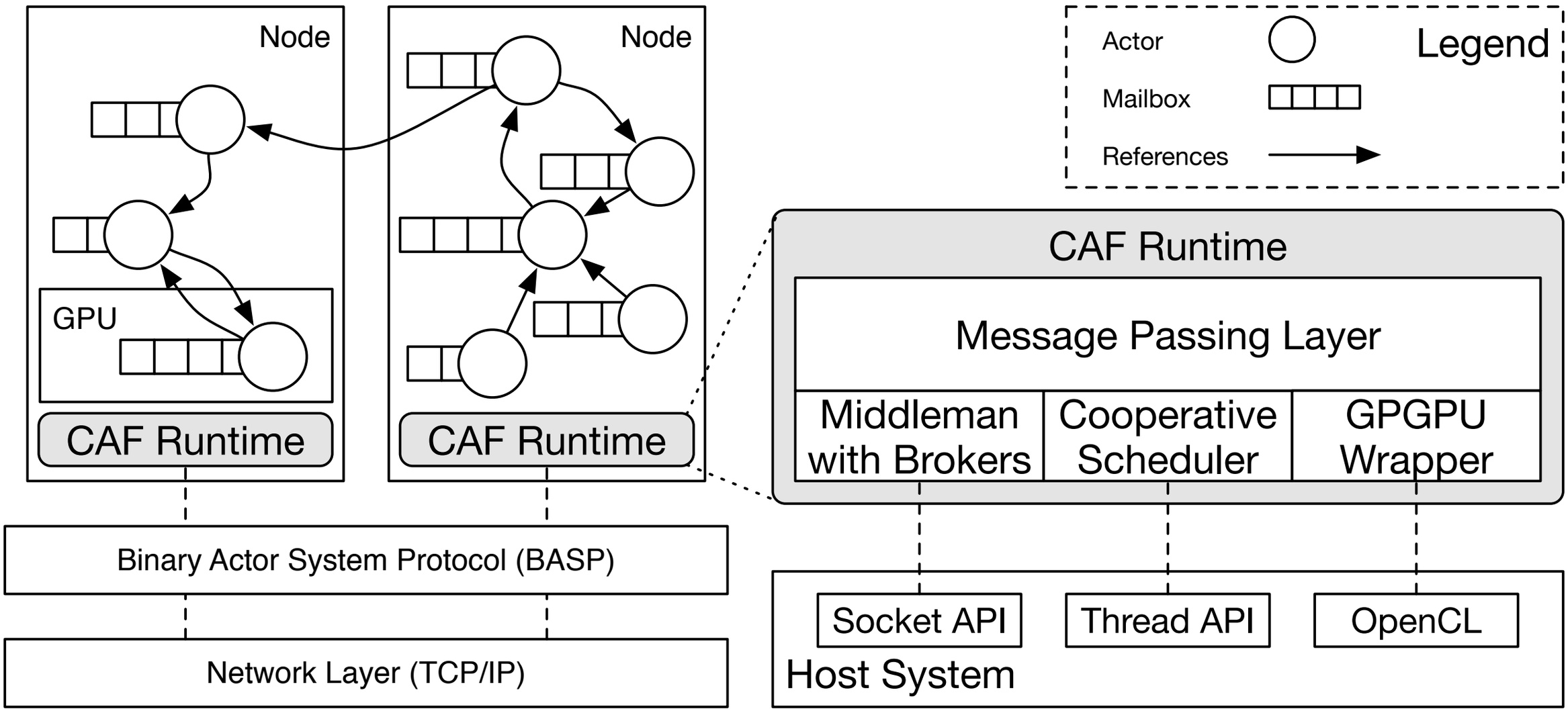
Actors as Function Objects

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

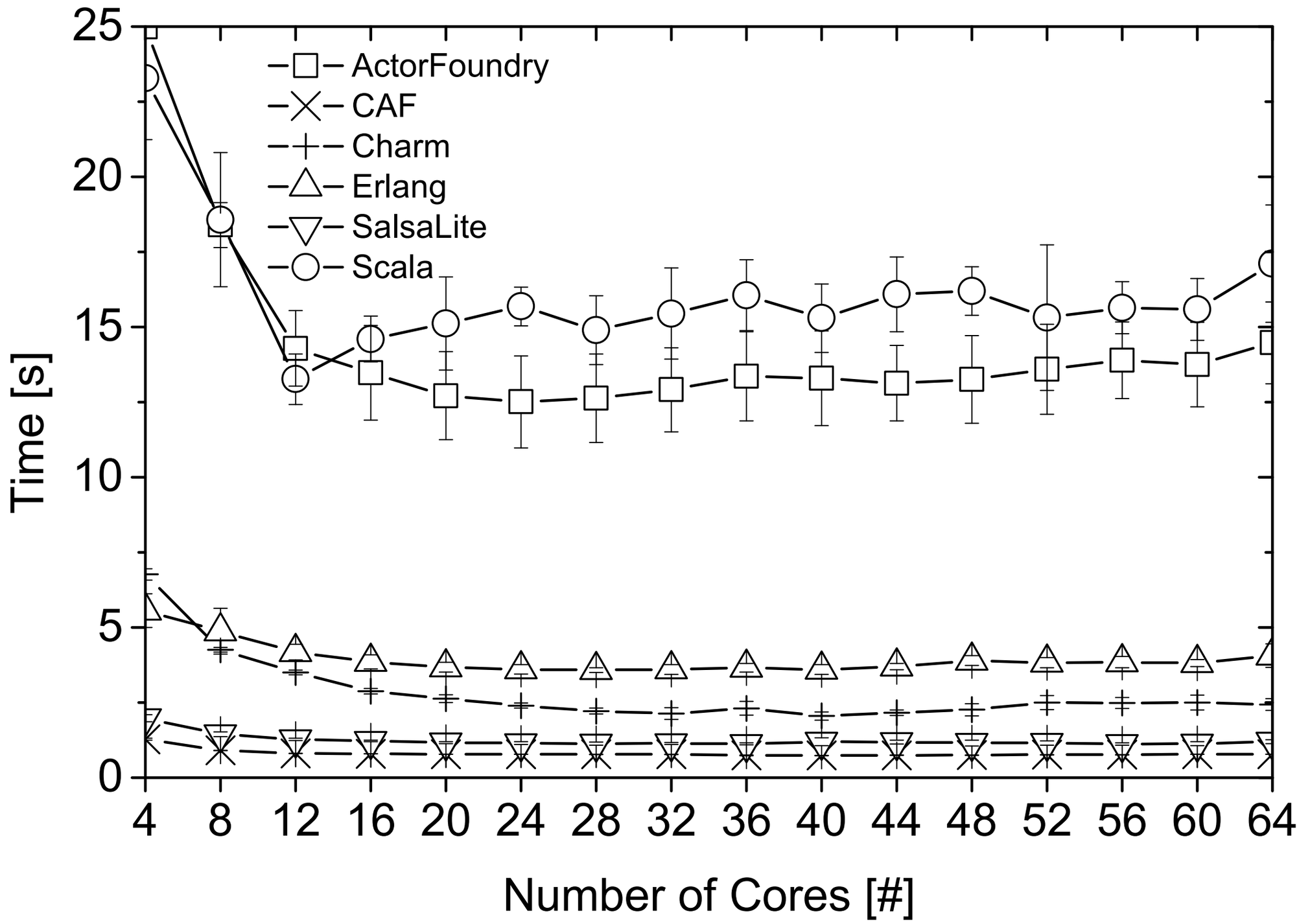
Anatomy of an Actor



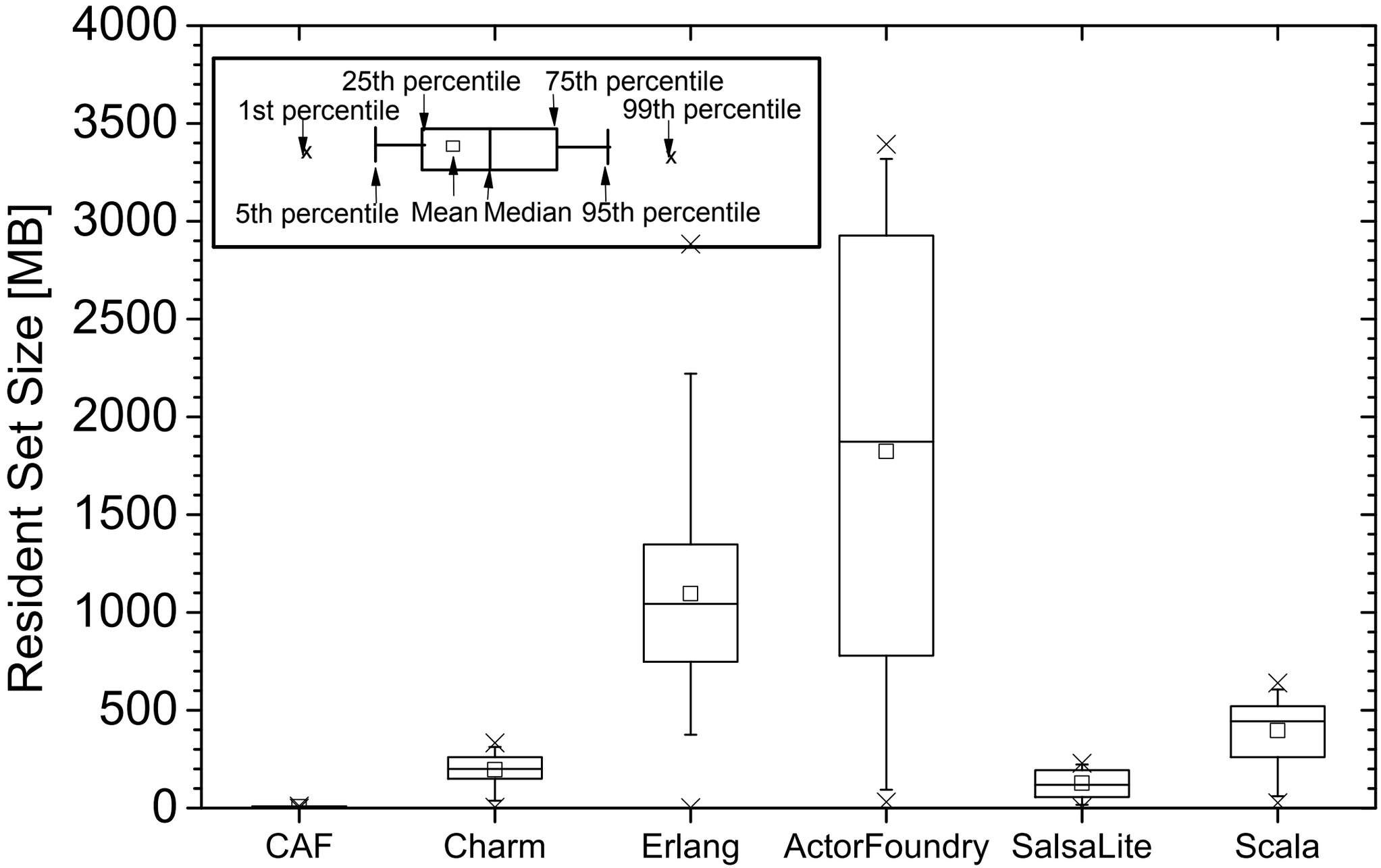
Architecture



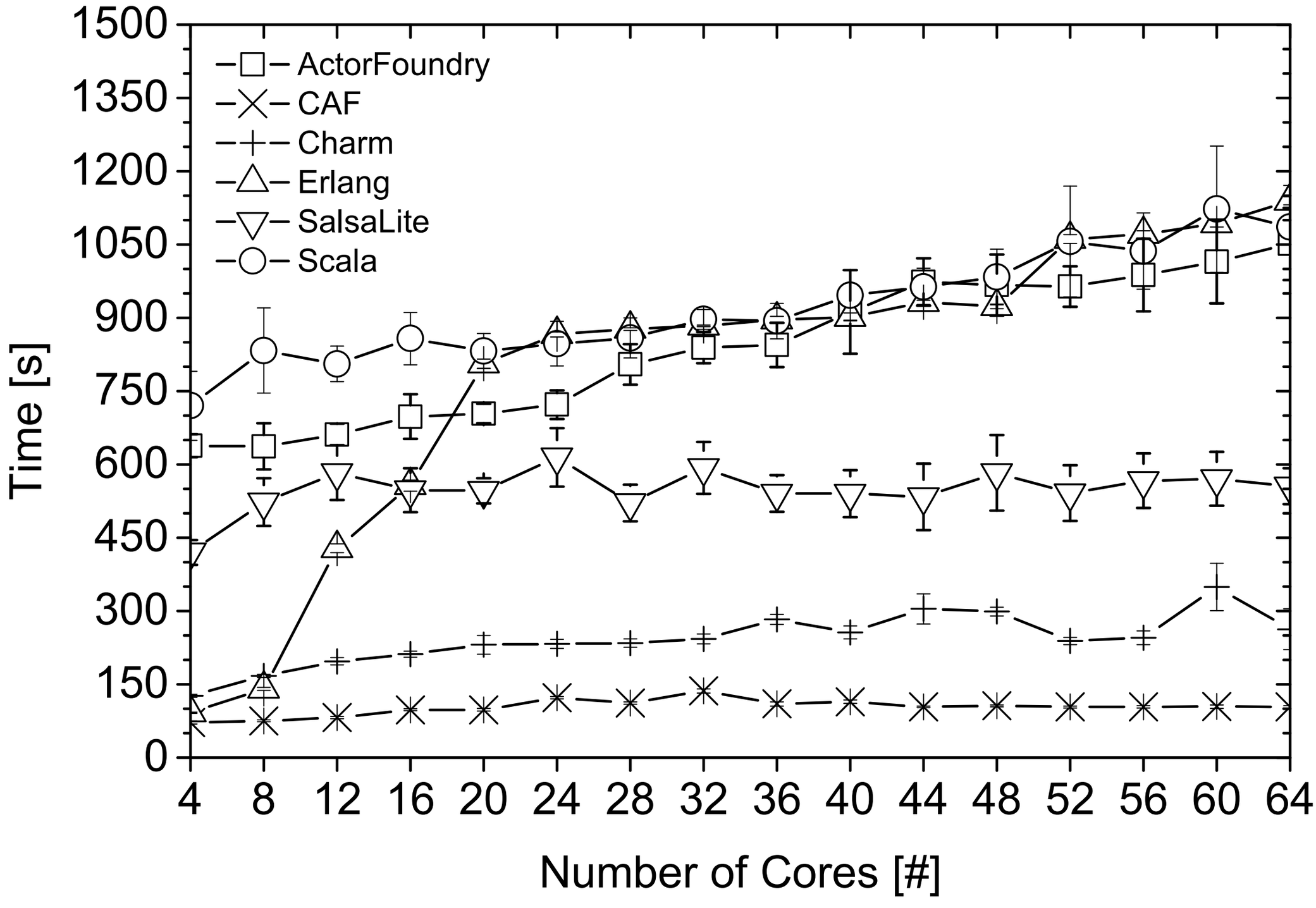
Actor Creation



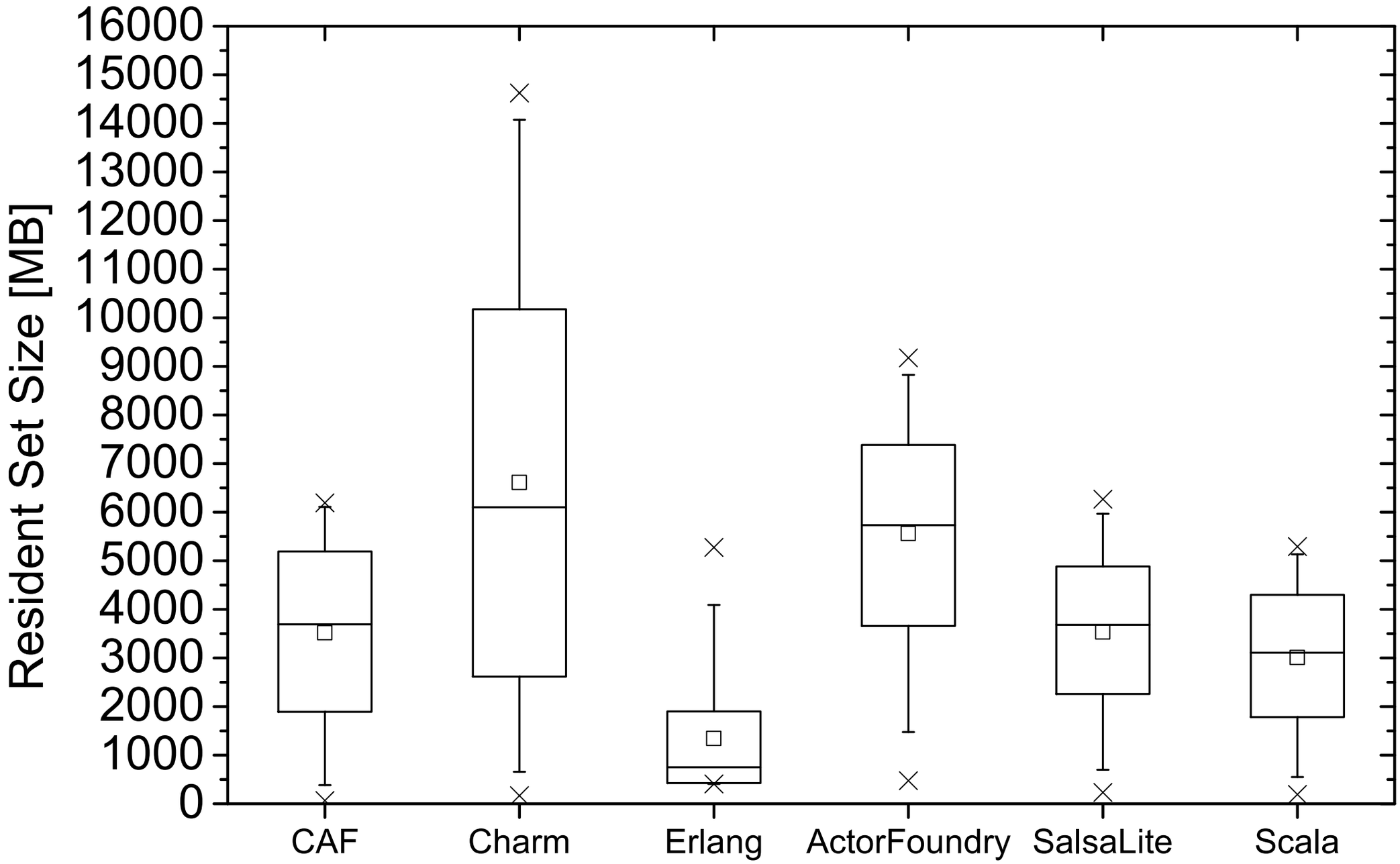
Actor Creation

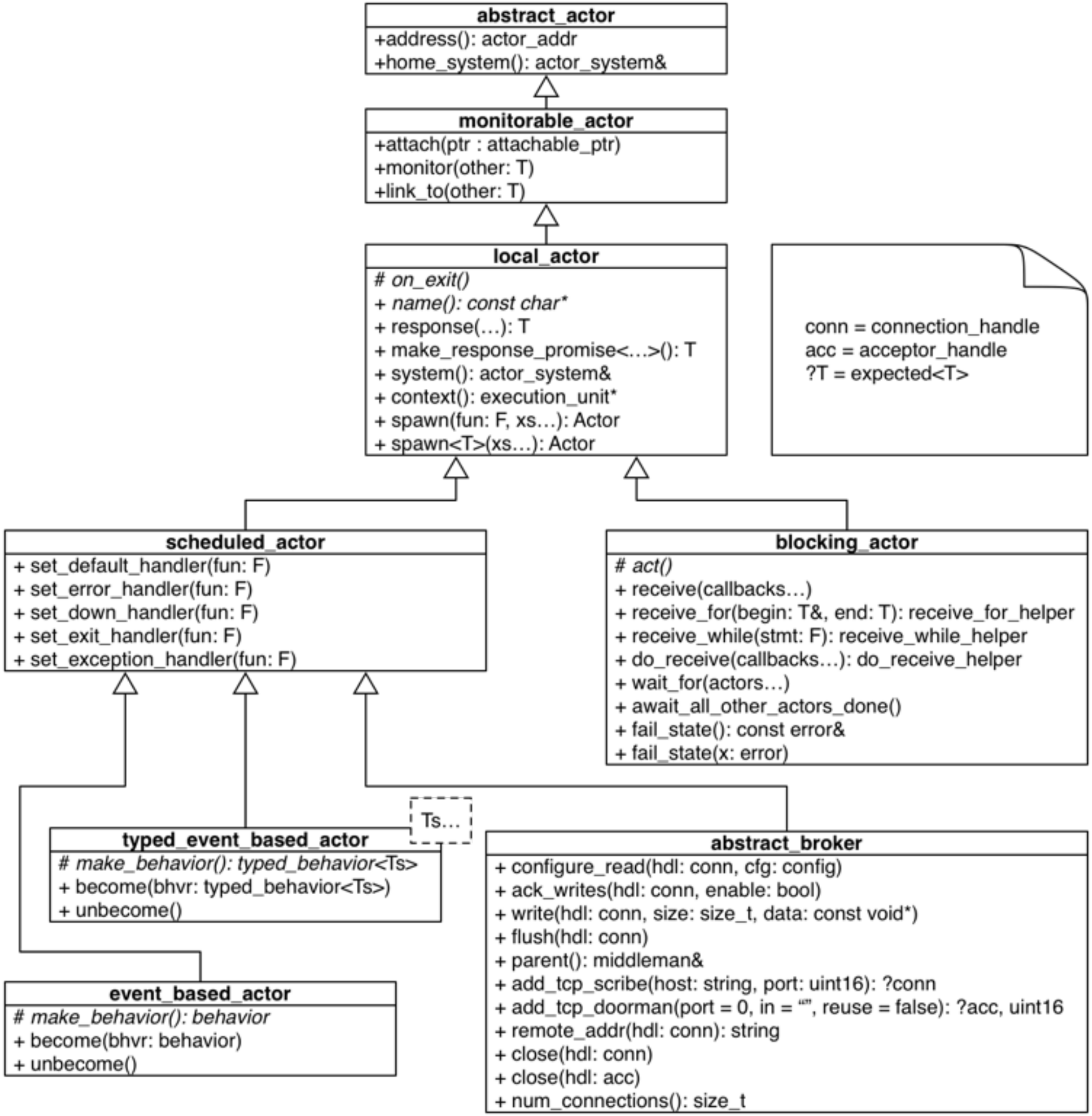


Mailbox - CPU


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

Mailbox - Memory




abstract_actor
+address(): actor_addr
+home_system(): actor_system&
monitorable_actor
+attach(ptr : attachable_ptr)
+monitor(other: T)
+link_to(other: T)
local_actor
on_exit()
+ name(): const char*
+ response(…): T
+ make_response_promise<…>(): T
+ system(): actor_system&
+ context(): execution_unit*
+ spawn(fun: F, xs…): Actor
+ spawn<T>(xs…): Actor
conn = connection_handle
acc = acceptor_handle
?T = expected<T>
scheduled_actor
+ set_default_handler(fun: F)
+ set_error_handler(fun: F)
+ set_down_handler(fun: F)
+ set_exit_handler(fun: F)
+ set_exception_handler(fun: F)
blocking_actor
act()
+ receive(callbacks…)
+ receive_for(begin: T&, end: T): receive_for_helper
+ receive_while(stmt: F): receive_while_helper
+ do_receive(callbacks…): do_receive_helper
+ wait_for(actors…)
+ await_all_other_actors_done()
+ fail_state(): const error&
+ fail_state(x: error)
Ts…
typed_event_based_actor
make_behavior(): typed_behavior<Ts>
+ become(bhvr: typed_behavior<Ts>)
+ unbecome()
abstract_broker
+ configure_read(hdl: conn, cfg: config)
+ ack_writes(hdl: conn, enable: bool)
+ write(hdl: conn, size: size_t, data: const void*)
+ flush(hdl: conn)
+ parent(): middleman&
+ add_tcp_scribe(host: string, port: uint16): ?conn
+ add_tcp_doorman(port = 0, in = "", reuse = false): ?acc, uint16
+ remote_addr(hdl: conn): string
+ close(hdl: conn)
+ close(hdl: acc)
+ num_connections(): size_t
event_based_actor
make_behavior(): behavior
+ become(bhvr: behavior)
+ unbecome()

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
	Features											
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.