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**: sequential unit of computation

**Message**: typed tuple

**Mailbox**: message FIFO

**Behavior**: function how to process next message

**Actor**: sequential unit of computation
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
<table>
<thead>
<tr>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++98</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>2003</td>
<td>2004</td>
<td>2005</td>
<td>2006</td>
<td>2007</td>
<td>2008</td>
<td>2009</td>
<td>2010</td>
</tr>
<tr>
<td>C++0x → 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Library TR1
- Decimal TR (not merged)
- Library TR2 (deferred to post-C++0x, then replaced by File System TS)
- Special Math IS

<table>
<thead>
<tr>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aug</td>
<td>Feb</td>
<td>Oct</td>
<td>Apr</td>
<td>Sep</td>
<td>May</td>
<td>Oct</td>
<td>Mar</td>
<td>Jun</td>
<td></td>
</tr>
</tbody>
</table>

**IS: trunk**
- C++14
- C++17
- C++20

**TSeS: feature branches** for beta release & then merge
- File System
- Networking
- Lib Fundamentals 1
- Lib Fundamentals 2
- Parallelism 1
- Parallelism 2
- Concepts
- Ranges
- Modules
- Concurrency 1
- Coroutines
- 2D Graphics
- Concurrency 2

TS bars start and end where work on detailed specification starts (“adopt initial working draft”) and ends (“send to publication”). Future starts/ends are shaded to indicate that dates and TS branches are approximate and subject to change.

[https://isocpp.org/std/status](https://isocpp.org/std/status)
CAF
An **actor** is typically implemented as a **function**

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

A list of **lambda**s determines the **behavior** of the actor.

A non-void return value sends a **response message** back to the sender.
int main() {
    actor_system_config cfg;
    actor_system sys{cfg};
    // Create (spawn) our actor.
    auto a = sys.spawn(add); // Spawns an actor valid only for the current scope.
    // Send it a message.
    scoped_actor self{sys};
    self->send(a, 40, 2); // Block and wait for reply.
    // Encapsulates all global state (worker threads, actors, types, etc.)
    self->receive(
        [](int result) {
            cout << result << endl; // prints "42"
        }
    );
}
int main() {
    actor_system_config cfg;
    actor_system sys{cfg};
    // Create (spawn) our actor.
    auto a = sys.spawn(add){
        // 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"
            }
        );
    }
}
auto a = sys.spawn(add);  
  sys.spawn(  
      [=](event_based_actor* self) -> behavior {  
        self->send(a, 40, 2);  
        return {  
          [=](int result) {  
            cout << result << endl;  
            self->quit();  
          }  
        };  
      }  
  );

Example #3

Optional first argument to running actor.

Capture by value because spawn returns immediately.

Designate how to handle next message. (= set the actor behavior)
auto a = sys.spawn(add);  

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

Non-Blocking Example #3
Example #4

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

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

Continuation specified as behavior.
Hardware

- Core 0
  - L1 cache
  - L2 cache
- Core 1
  - L1 cache
  - L2 cache
- Core 2
  - L1 cache
  - L2 cache
- Core 3
  - L1 cache
  - L2 cache

Network

- I/O

Operating System

- Threads
- Sockets

Actor Runtime

- Message Passing Abstraction
  - Cooperative Scheduler
  - Middleman / Broker
  - GPU Module

Application Logic

- Logic

Accelerator

- PCIe
- Accelerator
Hardware
- Core 0
  - L1 cache
  - L2 cache
- Core 1
  - L1 cache
  - L2 cache
- Core 2
  - L1 cache
  - L2 cache
- Core 3
  - L1 cache
  - L2 cache

Network
- I/O

Operating System
- Middleman / Broker
- Cooperative Scheduler

Actor Runtime
- Message Passing Abstraction

Application Logic

Accelerator
- GPU Module

C++ Actor Framework
Work Stealing*

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

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

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

Work Sharing

- **Centralized**: one shared global queue
- **No polling**
  - less CPU usage
  - lower throughput
- **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

```cpp
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

```cpp
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
Separation of **application logic** from **deployment**

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

```cpp
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
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;
    }
);
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

\[
a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n
\]
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, [&] {
                result(r, c) = dot_product(lhs, rhs, r, c);
            }));
    for (auto& f : futures)
        f.wait();
    return result;
}
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;
}
)__(";
```
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 $\Rightarrow$ 0.5 x runtime
(normalized)

Charm & Erlang good until 16 cores

Number of Cores [#] vs Speedup

- ActorFoundry
- CAF
- Charm
- Erlang
- SalsaLite
- Scala
- Ideal
Memory Overhead

Resident Set Size [MB]
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

![Graph showing comparison between CAF and OpenMPI](image)

- **CAF**
- **OpenMPI**

**Time [s]** vs. **Number of Worker Nodes [#]**
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**
Cached Stack

**Figure 27:** Enqueue operation in a cached stack

**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

![Graph showing the time taken for actor creation across different numbers of cores. The x-axis represents the number of cores, ranging from 4 to 64, and the y-axis represents the time taken in seconds, ranging from 0 to 25. The graph compares five different systems: ActorFoundry, CAF, Charm, Erlang, SalsaLite, and Scala. Each system is represented by a different marker and line style.](image-url)
Actor Creation

![Actor Creation Diagram](image-url)
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 - Memory

Resident Set Size [MB]

CAF    Charm    Erlang    ActorFoundry    SalsaLite    Scala
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

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

Demand flows upstream

Errors are propagated both ways
Streaming

User-defined function for creating outputs
Fault Tolerance

Host 1: actor A
Host 2: actor B (crossed out)
Host 3: actor C
Host 4: actor B'
Debugging
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

```cpp
actor a = sys.spawn(addner);
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**
// 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>
    >;
// 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

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

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

* Via reference counting, as opposed to tracing garbage collection.
† Only in SALSA, not SALSA Lite.