

libcppa

Designing an Actor Semantic for C++11

Dominik Charousset* and Thomas C. Schmidt*
dcharousset@acm.org, t.schmidt@ieee.org

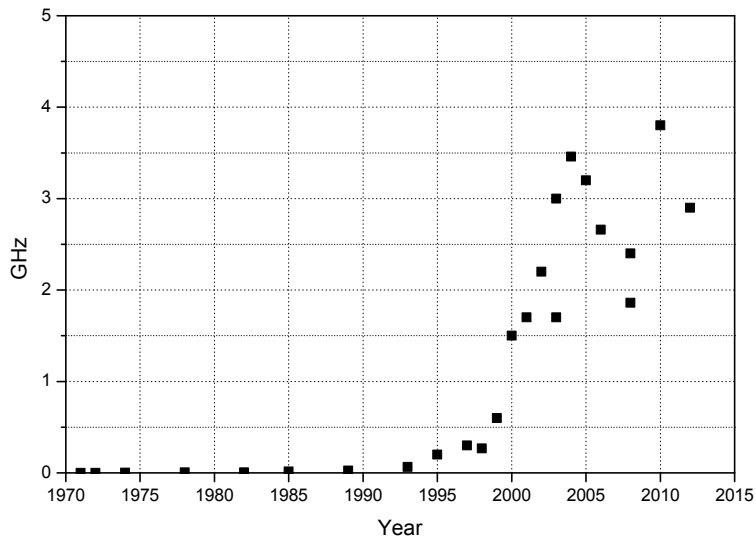
*iNET RG, Department of Computer Science
Hamburg University of Applied Sciences

May 2013

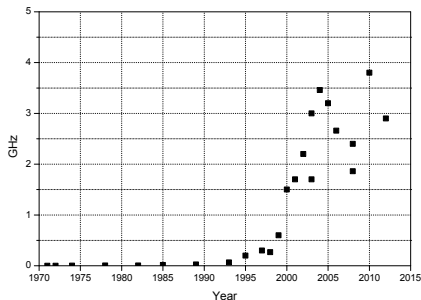
Agenda

- 1 Why focus on concurrency & distribution?
- 2 The problem with implicit sharing
- 3 The actor model & libcppa:
 - Concurrency without threads
 - Transparent inclusion of OpenCL
 - Pattern matching
 - Network transparency
- 4 Limitations Induced by C++11

"The Free Lunch Is Over"¹



"The Free Lunch Is Over"¹



Moore's law still remains true (for now), but ...

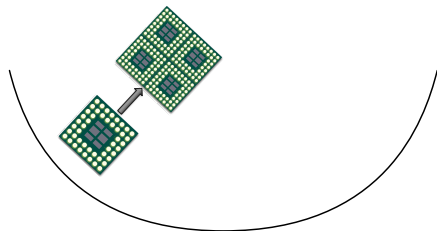
- More transistors \neq more clock speed
- "Old", i.e., single-threaded, SW no longer benefits from new HW

¹ Herb Sutter. Dr. Dobb's Journal 30(3):202-210 (2005)

Challenges of Modern Systems

Developers face not one, but multiple trends:

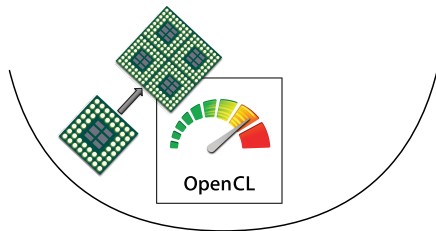
- More cores on both desktop & mobile platforms



Challenges of Modern Systems

Developers face not one, but multiple trends:

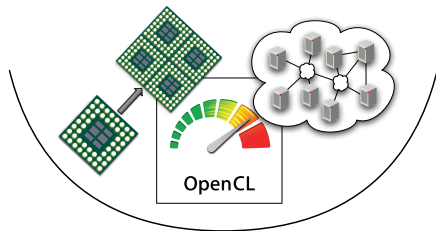
- More cores on both desktop & mobile platforms
- GPGPU programming: GPUs can vastly outperform CPUs



Challenges of Modern Systems

Developers face not one, but multiple trends:

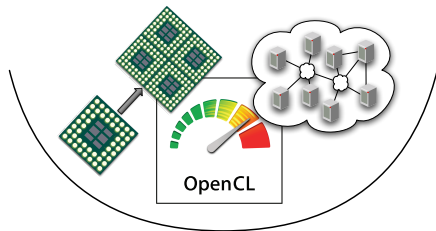
- More cores on both desktop & mobile platforms
- GPGPU programming: GPUs can vastly outperform CPUs
- Cloud computing, a.k.a. “infrastructure as a service”



Challenges of Modern Systems

Developers face not one, but multiple trends:

- More cores on both desktop & mobile platforms
 - GPGPU programming: GPUs can vastly outperform CPUs
 - Cloud computing, a.k.a. “infrastructure as a service”
- ⇒ Parallelization, specialization & distribution



Agenda

- 1 Why focus on concurrency & distribution?
- 2 The problem with implicit sharing
- 3 The actor model & libcppa:
 - Concurrency without threads
 - Transparent inclusion of OpenCL
 - Pattern matching
 - Network transparency
- 4 Limitations Induced by C++11

The Problem With Implicit Sharing

When writing concurrent programs:

- Stateful objects need to be synchronized (if shared)
- Developer is responsible for thread-safety
- Challenges are ...
 - Race conditions (“solved” by locks)
 - Deadlocks/Livelocks (caused by locks)
 - Poor scalability due to queueing (Coarse-Grained Locking)
 - Very high complexity (Fine-Grained Locking)
- Time-dependent errors make testing (almost) impossible

The Problem With Implicit Sharing

When writing concurrent programs:

- Stateful objects need to be synchronized (if shared)
 - Developer is responsible for thread-safety
 - Challenges are ...
 - Race conditions (“solved” by locks)
 - Deadlocks/Livelocks (caused by locks)
 - Poor scalability due to queueing (Coarse-Grained Locking)
 - Very high complexity (Fine-Grained Locking)
 - Time-dependent errors make testing (almost) impossible
- ⇒ Expert knowledge & experience required

Compose Synchronized Classes

```
class Subject {
public:
    void subscribe(function<void(int)> fun) {
        unique_lock<mutex> guard{m_mtx};
        m_subscribers.push_back(move(fun));
    }
    void broadcast(int value) {
        unique_lock<mutex> guard{m_mtx};
        for (auto& s : m_subscribers) s(value);
    }
private:
    mutex m_mtx;
    vector<function<void(int)>> m_subscribers;
};
```

Compose Synchronized Classes

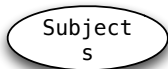
```
class FooBar {
public:
    void foo() {
        unique_lock<mutex> guard{m_mtx};
        m_subject->subscribe( [=](int v) {
            /*...*/ bar(v); /*...*/
        });
        // ...
    }
    void bar(int value) {
        unique_lock<mutex> guard{m_mtx};
        // ...
    }
private:
    Subject* m_subject;
    mutex m_mtx;
};
```

Compose Synchronized Classes

Thread1

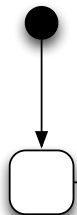


Thread2



Compose Synchronized Classes

Thread1



subscribe(f)

Subject
s

Functor
f

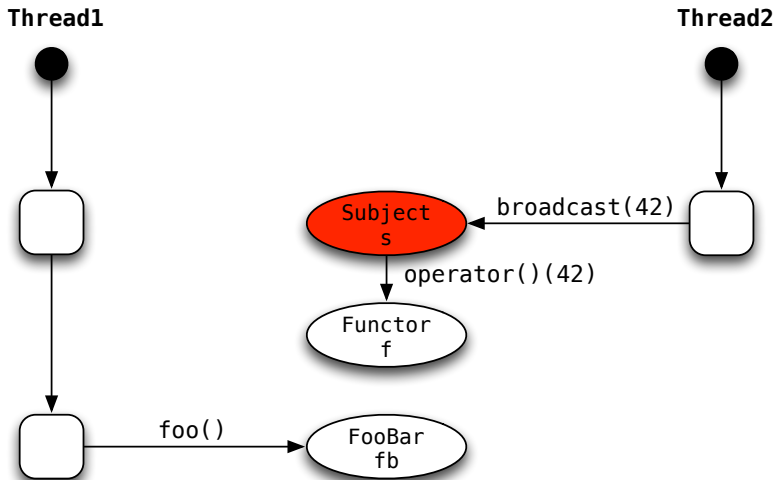
FooBar
fb

```
[(int val) {  
  ...  
  fb.bar();  
  ...  
}]
```

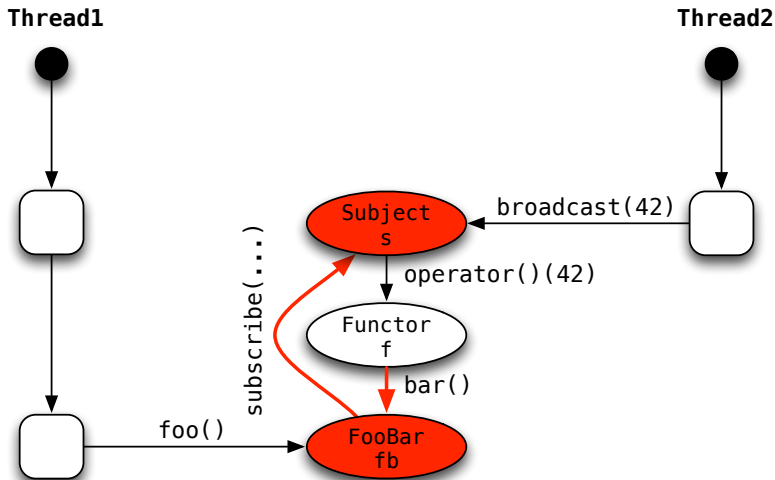
Thread2



Compose Synchronized Classes



Compose Synchronized Classes



Locks Are Not Composable

“Mutable, stateful objects are the new spaghetti code.”
– Rich Hickey

Locks Are Not Composable

“Mutable, stateful objects are the new spaghetti code.”
– Rich Hickey

- Libraries with threads & locks are no longer black boxes
- Composition of two thread-safe classes not necessarily thread-safe
- User has to know about implementation details:
 - Which code runs asynchronously/where?
 - Which functions are “thread-safe”?
 - Which function uses which lock?

Locks Are Not Composable

“Mutable, stateful objects are the new spaghetti code.”
– Rich Hickey

- Libraries with threads & locks are no longer black boxes
- Composition of two thread-safe classes not necessarily thread-safe
- User has to know about implementation details:
 - Which code runs asynchronously/where?
 - Which functions are “thread-safe”?
 - Which function uses which lock?

⇒ Wrong level of abstraction

The “Right” Level of Abstraction

A programming paradigm should enable us to ...

- Easily split application logic into as many tasks as needed
 - Avoid race conditions by design (no locks!)
 - Keep interfaces between two software components stable:
 - Whether or not they run on the same host
 - Whether or not they run on specialized hardware
- ⇒ Flexible composition

Agenda

- 1 Why focus on concurrency & distribution?
- 2 The problem with implicit sharing
- 3 The actor model & libcppa:
 - Concurrency without threads
 - Transparent inclusion of OpenCL
 - Pattern matching
 - Network transparency
- 4 Limitations Induced by C++11

The Actor Model

Actors are concurrent entities, that ...

- Communicate via message passing
- Do not share state
- Can create (“spawn”) new actors
- Can monitor other actors

Benefits of the Actor Model

- High-level, explicit communication: no locks, no implicit sharing
- A lightweight implementation allows millions of active actors
- Applies to both concurrency *and* distribution
 - Divide workload by spawning actors
 - Network-transparent messaging

libcppa – A C++11 Actor Library

libcppa provides an actor semantic for C++11

- Raises the level of abstraction (ease of development)
- Implements lightweight actors (ease of concurrency)
- Offers transparent OpenCL layer (ease of composition)
- Operates network transparent (ease of distribution)

Concurrency with libcppa

In order to make use of increasingly parallel hardware, we need ...

- to split application logic into many tasks, i.e., actors
- minimal overhead for launching actors and collecting results
- support to collect results from specialized HW in the same way

Multiply Matrices

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

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

Multiply Matrices – 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) {
            // each calculation can run independently
            result(r, c) = dot_product(lhs, rhs, r, c);
        }
    }
    return move(result);
}
```

Multiply Matrices – 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 move(result);
}
```

Multiply Matrices – libcppa Actors

```
matrix actor_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) {
            spawn([&,r,c] {
                result(r, c) = dot_product(lhs, rhs, r, c);
            });
        }
    }
    await_all_others_done();
    return move(result);
}
```

Multiply Matrices – 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;
    }
)__;
```

Multiply Matrices – OpenCL Actors

```
matrix opengl_multiply(const matrix& lhs,
                       const matrix& rhs) {
    typedef vector<float> fvec;
    typedef const fvec& fvec_cref;
        // function signature
    auto worker = spawn_cl<fvec(fvec_cref, fvec_cref)>(
        // code, kernel name & dimensions
        source, "multiply",
        {matrix_size, matrix_size});
    // ordinary message passing
    send(worker, lhs.data(), rhs.data());
    matrix result;
    receive(on_arg_match >> [&](fvec& res_vec) {
        result = move(res_vec);
    });
    return move(result);
}
```


Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.7, 1000x1000 matrices

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

Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.7, 1000x1000 matrices

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

```
time ./actor_multiply  
0m2.428s
```

Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.7, 1000x1000 matrices

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

```
time ./actor_multiply  
0m2.428s
```

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

Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.7, 1000x1000 matrices

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

```
time ./actor_multiply  
0m2.428s
```

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

```
time ./async_multiply
```

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

Multiply Matrices – Runtimes

Setup: 12 cores, Linux, GCC 4.7, 1000x1000 matrices

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

```
time ./actor_multiply  
0m2.428s
```

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

```
time ./async_multiply
```

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

... apparently, one cannot start 1,000,000 threads

Multiply Matrices – Summary

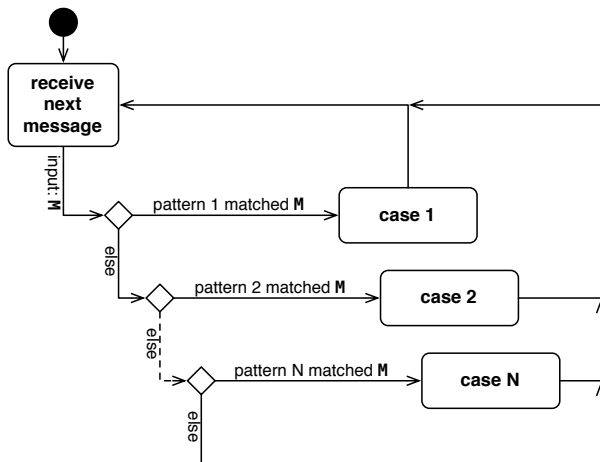
Performance of `actor_multiply` is suboptimal:

- We spawn considerably more actors than cores are available
- When spawning 1,000 actors instead, runtime drops to 0.8 s

However:

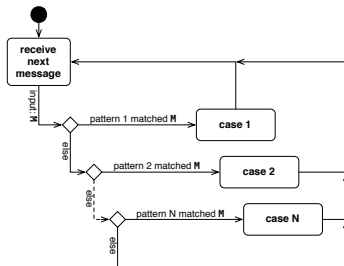
- Spawning actors is fast (a million actors in ≤ 2.4 s)
- Threads do not scale up to large numbers, actors do

Message Processing



Typical actor loop

Message Processing



- Messages are copy-on-write tuples of any size
- Messages are buffered at the actor in a FIFO-ordered *mailbox*
- Actors set a partial function f as (replaceable) message handler
- Runtime skips each message M if $f(M)$ is undefined
- Unmatched (skipped) messages remain in the actor's mailbox
- Each receive operation begins with the oldest element

Partial Functions in libcppa

```
partial_function f {
    on("hello") >> [] {
        cout << "hello!" << endl;
    },
    on(atom("hello")) >> [] {
        cout << "atom(hello)!" << endl;
    },
    on_arg_match >> [](int a, int b) {
        cout << a << ", " << b << endl;
    },
    on("hello", arg_match) >> [](const string& name) {
        cout << "hello " << name << "!" << endl;
    }
};

assert(not f(make_any_tuple(42)));
assert(f(make_any_tuple("hello")));
```

Partial Functions in libcppa

```
partial_function f {  
  on("hello") >> [] {  
    cout << "hello!" << endl;  
  },  
  on_arg_match >> [] (int a, int b) {  
    cout << a << ", " << b << endl;  
  },  
  on("hello", arg_match) >> [] (const string& name) {  
    cout << "hello " << name << "!" << endl;  
  }  
};  
  
assert(not f(make_any_tuple(42)));  
assert(f(make_any_tuple("hello")));
```

matches tuples with one (string) element of value "hello"

callback that should be invoked on a match; could take a string as argument

Partial Functions in libcppa

```
partial_function f {
  on("hello") >> [] {
    cout << "hello!" << endl;
  },
  on(atom("hello")) >> [] {
    cout << "atom(hello)!" << endl;
  },
  on(int b) {
    cout << b << endl;
  },
  on("hello ", arg_name) >> [](const string& name) {
    cout << "hello " << name << "!" << endl;
  }
};

assert(not f(make_any_tuple(42)));
assert(f(make_any_tuple("hello")));
```

atoms are constants, calculated at compile time from short strings (max 10 characters)

Partial Functions in libcppa

```
partial_function f {
  on("hello") >> [] {
    cout << "hello!" << endl;
  },
  on(atom("hello")) >> [] {
    cout << "atom(hello)!" << endl;
  },
  on_arg_match >> [](int a, int b) {
    cout << a << ", " << b << endl;
  },
  on(const string& name) >> [](const string& name) {
    cout << "!" << endl;
  }
};
```

deduce types from callback signature → match tuples with two integers

```
assert(not f(make_any_tuple(42)));
assert(f(make_any_tuple("hello")));
```

Partial Functions in libcppa

```
partial_function f {
  on("hello") >> [] {
    cout << "hello!" << endl;
  },
  on("hello", arg_match) >> [] (const string& name) {
    cout << a << ", " << b << endl;
  },
  on("hello", arg_match) >> [] (const string& name) {
    cout << "hello " << name << "!" << endl;
  }
};
```

deduce second half of types from
callback signature → match tuples with
two strings if first element is "hello"

```
assert(not f(make_any_tuple(42)));
assert(f(make_any_tuple("hello")));
```

Partial Functions in libcppa

```
partial_function f {  
  on("hello") >> [] {  
    cout << "hello!" << endl;  
  },  
  on(atom("hello")) >> [] {  
    cout << "atom(hello)!" << endl;  
  },  
  on_arg_match >> [](int a, int b) {  
    cout << a << ", " << b << endl;  
  }  
};
```

libcppa's pattern matching is defined only for any_tuple, because it requires runtime type information

```
const string& name) {  
  cout << endl;  
};
```

```
assert(not f(make_any_tuple(42)));  
assert(f(make_any_tuple("hello")));
```

Minimal Actor Example

```
void math_server() {
    become (
        on(atom("plus"), arg_match) >> [](int a, int b) {
            reply(atom("result"), a + b);
        }
    );
}

void math_client(actor_ptr ms) {
    sync_send(ms, atom("plus"), 40, 2).then(
        on(atom("result"), arg_match) >> [=](int result) {
            cout << "40 + 2 = " << result << endl;
        }
    );
}

int main() {
    spawn(math_client, spawn(math_server));
    // ...
}
```

Minimal Actor Example

```
void math_server() {  
    become (  
        on(atom("plus"), arg_match) >> [](int a, int b) {  
            reply(atom("result"), a + b);  
        }  
    );  
}  
void math_client(actor_ptr ms) {  
    sync_send(ms, atom("plus"), 40, 2).then(  
        on(atom("result"), arg_match) >> [=](int result) {  
            cout << "40 + 2 = " << result << endl;  
        }  
    );  
}  
int main() {  
    spawn(math_client, spawn(math_server));  
    // ...  
}
```

set partial function as message handler; handler is used until replaced or actor is done

Minimal Actor Example

```
void math_server() {
    become (
        on(atom("plus"), arg_match) >> [](int a, int b) {
            send a message and then
            wait for response
            (using a "one-shot handler")
        }
    );
}

void math_client(actor_ptr ms) {
    sync_send(ms, atom("plus"), 40, 2).then(
        on(atom("result"), arg_match) >> [=](int result) {
            cout << "40 + 2 = " << result << endl;
        }
    );
}

int main() {
    spawn(math_client, spawn(math_server));
    // ...
}
```

Minimal Actor Example

```
void math_server() {
    become (
        on(atom("plus"), arg_match) >> [](int a, int b) {
            a + b);
}

void math_client(actor_ptr ms) {
    sync_send(ms, atom("plus"), 40, 2).then(
        on(atom("result"), arg_match) >> [=](int result){
            cout << "40 + 2 = " << result << endl;
        }
    );
}

int main() {
    spawn(math_client, spawn(math_server));
    // ...
}
```

this actor "loops" forever
(or until it is forced to quit)

Minimal Actor Example

```
void math_server() {
    become (
        (ch) >> [](int a, int b) {
            a + b);
    );
}

void math_client(actor_ptr ms) {
    sync_send(ms, atom("plus"), 40, 2).then(
        on(atom("result"), arg_match) >> [=](int result) {
            cout << "40 + 2 = " << result << endl;
        }
    );
}

int main() {
    spawn(math_client, spawn(math_server));
    // ...
}
```

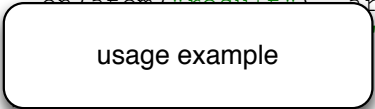
this actor sends one message and receives one messages

Minimal Actor Example

```
void math_server() {
    become (
        on(atom("plus"), arg_match) >> [](int a, int b) {
            reply(atom("result"), a + b);
        }
    );
}

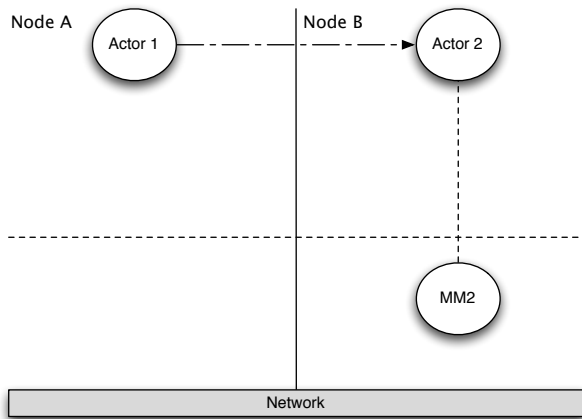
void math_client(actor_ptr ms) {
    sync_send(ms, atom("plus"), 40, 2).then(
        on(atom("result"), arg_match) >> [=](int result) {
            << result << endl;
        }
    );
}

int main() {
    spawn(math_client, spawn(math_server));
    // ...
}
```



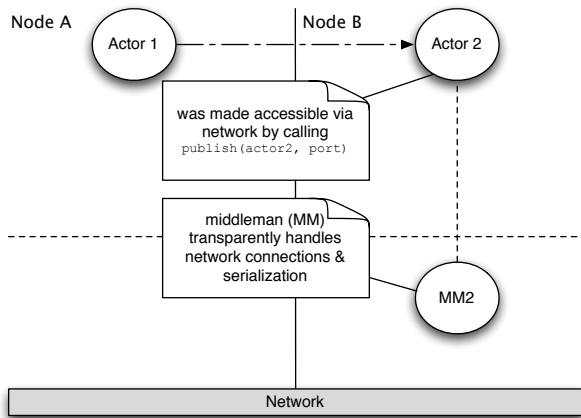
usage example

Network Transparency



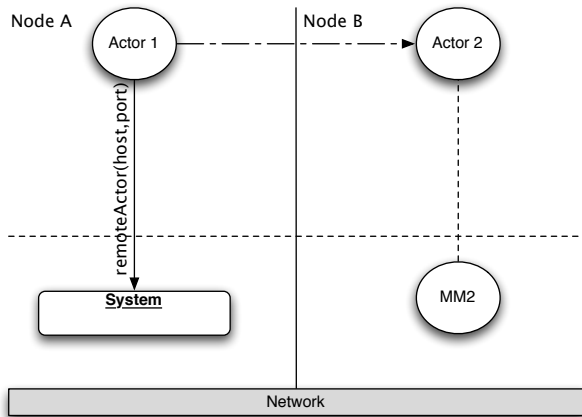
Send a message to a remote (“published”) actor

Network Transparency



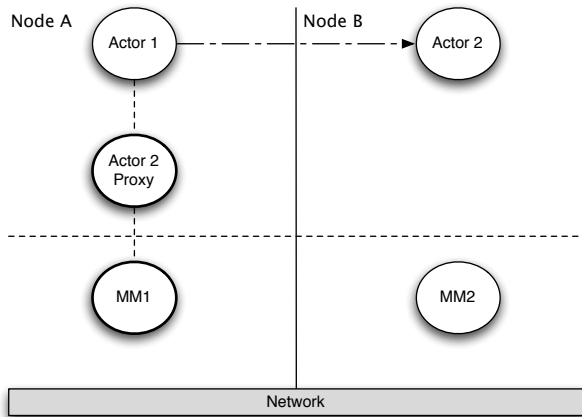
Send a message to a remote (“published”) actor

Network Transparency



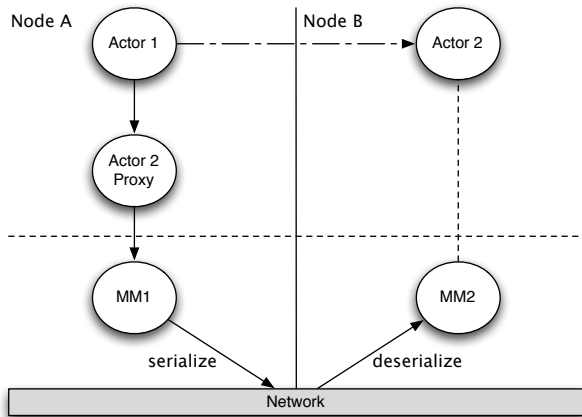
Send a message to a remote ("published") actor

Network Transparency



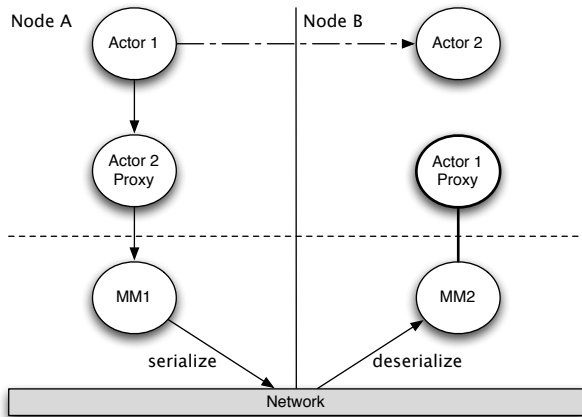
Send a message to a remote (“published”) actor

Network Transparency



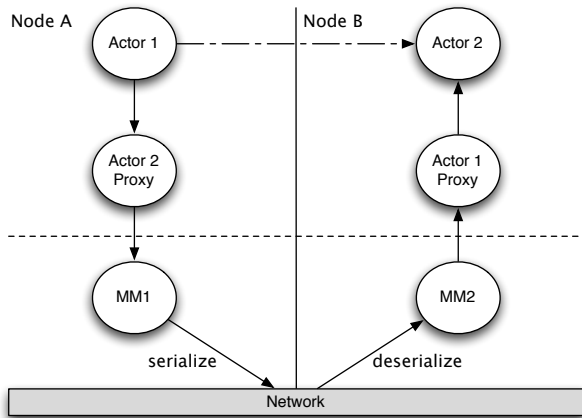
Send a message to a remote (“published”) actor

Network Transparency



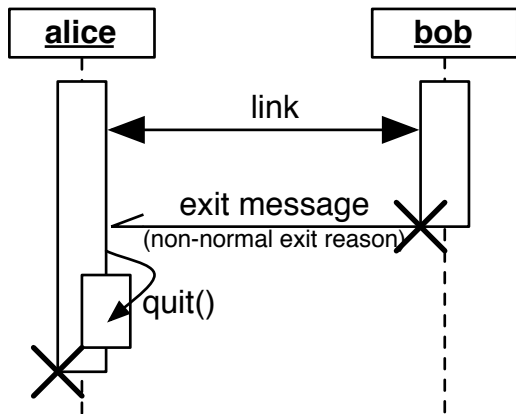
Send a message to a remote (“published”) actor

Network Transparency

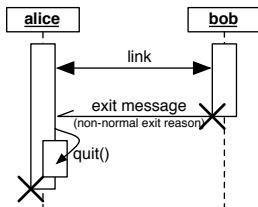


Send a message to a remote ("published") actor

Fault Tolerance – Linking Actors



Fault Tolerance – Linking Actors



- Actors can *link* their lifetime
- Errors are propagated through exit messages
- When receiving an exit message:
 - Actors fail for the same reason per default
 - Actors can *trap* exit messages to handle failure manually
- Build systems where all actors are alive or have collectively failed

Linking Actors in libcppa – Example

```
void bob_fun(); // will fail
void alice_fun() {
    auto bob = spawn<linked>(bob_fun);
    send(bob, "hello bob");
    become ( /* will bob ever call back? */ );
}
void carl() {
    self->trap_exit(true);
    auto alice = spawn<linked>(alice_fun);
    become (
        on(atom("EXIT"), arg_match) >> [](uint32_t r) {
            if (r != exit_reason::normal)
                cout << "something went wrong..." << endl;
        }
    );
}
```

Linking Actors in libcppa – Example

```
void bob_fun(); // will fail
void alice_fun() {
    auto bob = spawn<linked>(bob_fun);
    send(bob, "hello bob");
    become ( /* will bob ever call back? */ );
}
void alice_fun() {
    auto alice = spawn<linked>(alice_fun);
    become (
        on(atom("EXIT"), arg_match) >> [](uint32_t r) {
            if (r != exit_reason::normal)
                cout << "something went wrong..." << endl;
        }
    );
}
```

spawn bob with linked lifetime:
if bob fails, alice fails as well
(and vice versa)

Linking Actors in libcppa – Example

```
void bob_fun(); // will fail
void alice_fun() {
    auto bob = spawn<linked>(bob_fun);
    send(bob, "hello bob");
    become ( /* will bob ever call back? */ );
}
void carl() {
    self->trap_exit(true);
    auto alice = spawn<linked>(alice_fun);
    become (
        [alice] {
            alice->send("hello alice");
            alice->send(wrong);
        }
    );
}
```

self always points to the running actor itself

Linking Actors in libcppa – Example

```
void bob_fun(); // will fail
void alice_fun() {
    auto bob = spawn<linked>(bob_fun);
    send(bob, "hello bob");
    become ( /* will bob ever call back? */ );
}
void carl() {
    self->trap_exit(true);
    auto alice = spawn<linked>(alice_fun);
    become (
        [alice] {
            catch ( ... ) {
                normal
                vent wrong..." << endl;
            }
        }
    );
}
```

receive exit messages as ordinary messages; overriding the default behavior

Linking Actors in libcppa – Example

```
void bob_fun(); // will fail
void alice_fun() {
    auto bob = spawn<linked>(bob_fun);
    send(bob, "hello bob");
    become ( /* will bob ever call back? */ );
}
void carl() {
    self->trap_exit(true);
    auto alice = spawn<linked>(alice_fun);
    become (
        on(atom("EXIT"), arg_match) >> [](uint32_t r) {
            if (r == 1)
                send(bob, "wrong..." << endl;
        )
    );
}
```

carl traps exit messages of alice,
alice would fail whenever carl
fails (default behavior)

Linking Actors in libcppa – Example

```
void bob_fun(); // will fail
void alice_fun() {
    auto bob = spawn<linked>(bob_fun);
    send(bob, "hello bob");
    // ... ( /t/ ... back? */ );
}
void ...
auto alice = spawn<linked>(alice_fun);
become (
    on(atom("EXIT"), arg_match) >> [](uint32_t r) {
        if (r != exit_reason::normal)
            cout << "something went wrong..." << endl;
    }
);
}
```

exit messages always consist of the atom 'EXIT' and the exit reason as uint32

Linking Actors in libcppa – Example

```
void bob_fun(); // will fail
void alice_fun() {
    auto bob = spawn<linked>(bob_fun);
    send(bob, "hello bob");
    become ( /* will bob ever call back? */ );
}
void main() {
    auto alice = spawn<linked>(alice_fun);
    become (
        on(atom("EXIT"), arg_match) >> [](uint32_t r) {
            if (r != exit_reason::normal)
                cout << "something went wrong..." << endl;
        }
    );
}
```

a normal exit reason would indicate that alice is done (no failure occurred)

libcppa Facts Sheet

- Open source (GPLv2) C++11 actor library
- Runs on GCC \geq 4.7, Clang \geq 3.2 (Linux + Mac)
- Will run on Windows as soon as MSVC supports required features
- Hosted on GitHub
- Feedback & contributions always welcome!
- Hot topics in the iNET group:
 - Actors on ARM / embedded systems
 - Actors & publish/subscribe (multicast)
 - Message routing & composability

Agenda

- 1 Why focus on concurrency & distribution?
- 2 The problem with implicit sharing
- 3 The actor model & libcppa:
 - Concurrency without threads
 - Transparent inclusion of OpenCL
 - Pattern matching
 - Network transparency
- 4 Limitations Induced by C++11

Limitations Induced by C++11

- C++ can not serialize functions or classes:
 - No migration of actors to other nodes at runtime
 - Function `spawn` can not create actors on remote nodes

Limitations Induced by C++11

- C++ can not serialize functions or classes:
 - No migration of actors to other nodes at runtime
 - Function `spawn` can not create actors on remote nodes
- No language support for pattern matching
 - More code noise compared to functional languages
 - Pattern matching is limited to `any_tuple`

Limitations Induced by C++11

- C++ can not serialize functions or classes:
 - No migration of actors to other nodes at runtime
 - Function `spawn` can not create actors on remote nodes
- No language support for pattern matching
 - More code noise compared to functional languages
 - Pattern matching is limited to `any_tuple`
- No reflection-like access to data types (C++1y?)
 - Serialization of messages managed by `libcoppa`'s type system
 - User-defined data types must be announced

Thank you for your attention!

Developer blog: <http://libcppa.org>

Sources: <https://github.com/Neverlord/libcppa>

iNET working group: <http://inet.cpt.haw-hamburg.de>