`libcppa`

An actor library for C++ with extensible group semantic

`libcppa`

Dominik Charousset

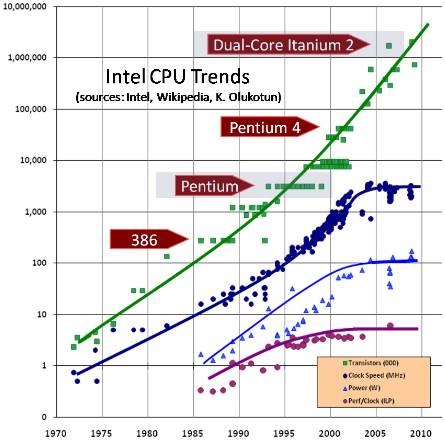
July 2011

Agenda

1. Motivation
2. Concurrency Approaches
3. The Actor Model
4. libcppa
5. libcppa
    - Architecture
6. Questions & Answers

Motivation

Herb Sutter: "The Free Lunch Is Over"

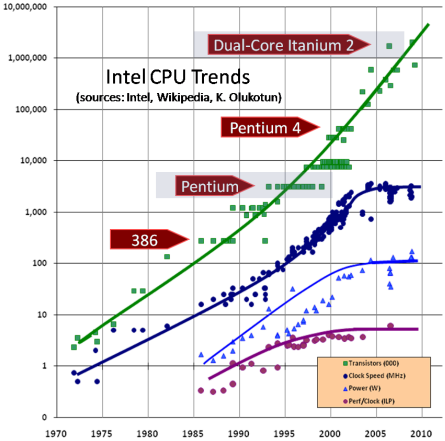


http://www.gotw.ca/publications/concurrency-ddj.htm

- CPU clock speed stagnates
- More cores instead of more clock speed

Motivation

Herb Sutter: “The Free Lunch Is Over”



http://www.gotw.ca/publications/concurrency-ddj.htm

- CPU clock speed stagnates
- More cores instead of more clock speed

⇒ Single-threaded Software doesn’t benefit from new hardware

Motivation

Herb Sutter: “The Free Lunch Is Over” – Consequence

“Software has to double the amount of parallelism
that it can support every two years.”
– Shekhar Y. Borkar (Intel)

Motivation

Multithreading In C-like Languages

- A multithreaded environment requires, that each object (in the *shared memory*) has to be **thread safe**
- Immutable objects are always thread-safe (if initialization is done)
- *Stateful objects* need synchronization

Motivation

Multithreading In C-like Languages

- A multithreaded environment requires, that each object (in the *shared memory*) has to be **thread safe**
- Immutable objects are always thread-safe (if initialization is done)
- *Stateful objects* need synchronization
  ⇒ The developer is responsible for thread safety! Errors lead to ...
    - Race conditions
    - Deadlocks/Lifelocks
    - Poor scalability due to queueing (Coarse-Grained Locking)

Motivation

Multithreading In C-like Languages

- A multithreaded environment requires, that each object (in the *shared memory*) has to be **thread safe**
- Immutable objects are always thread-safe (if initialization is done)
- *Stateful objects* need synchronization
  ⇒ The developer is responsible for thread safety! Errors lead to ...
  - Race conditions
  - Deadlocks/Lifelocks
  - Poor scalability due to queueing (Coarse-Grained Locking)

"Mutable stateful objects are the new spaghetti code" – Rich Hickey

Motivation

Multithreading in C-like languages – Example 1

```
class Subject {
    private int value; private List<Listener> listeners = ...;
    public interface Listener {
        public void stateChanged(int newValue);
    }
    public synchronized void addListener(Listener listener) {
        listeners.add(listener);
    }
    public synchronized void setValue(int newValue) {
        value = newValue;
        for (Listener l :  listeners) {
            l.stateChanged(newValue);
        }
    }
}
```

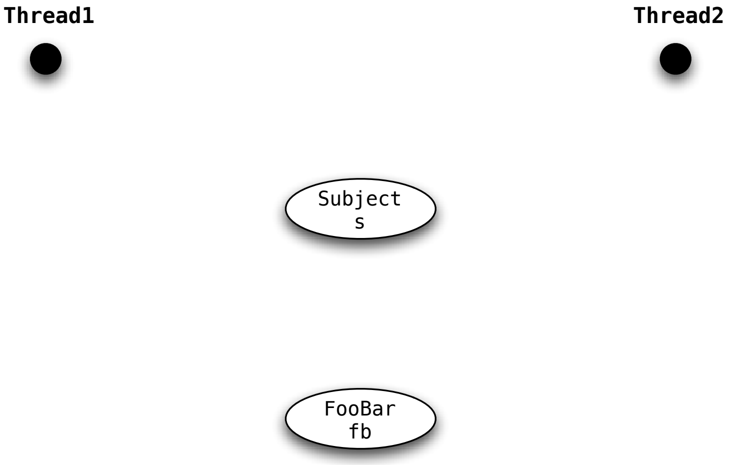
Motivation

Multithreading in C-like languages – Example 1

```
class FooBar {
    private Subject s;
    public synchronized void foo() {
        ...
        s.addListener(...);
        ...
    }
    public synchronized void bar() {
        ...
    }
}
```

Motivation

Multithreading in C-like languages – Example 1



Thread1

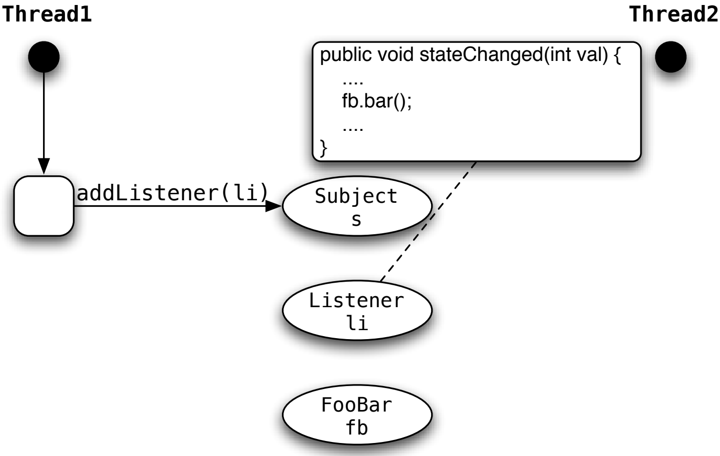
Thread2

Subject
s

FooBar
fb

Motivation

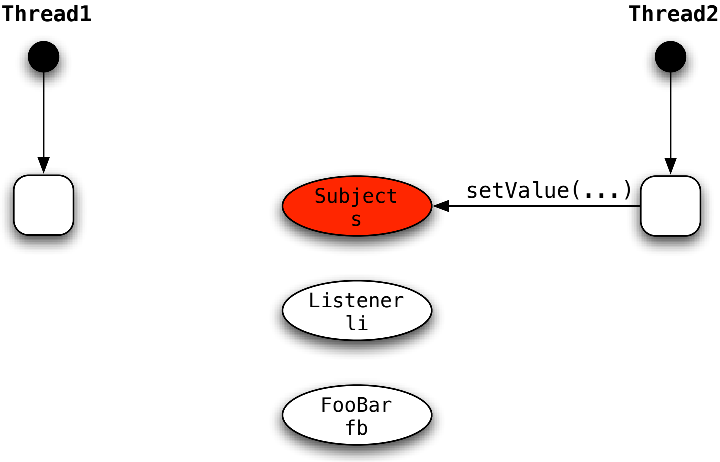
Multithreading in C-like languages – Example 1



```
public void stateChanged(int val) {
    ....
    fb.bar();
    ....
}
```

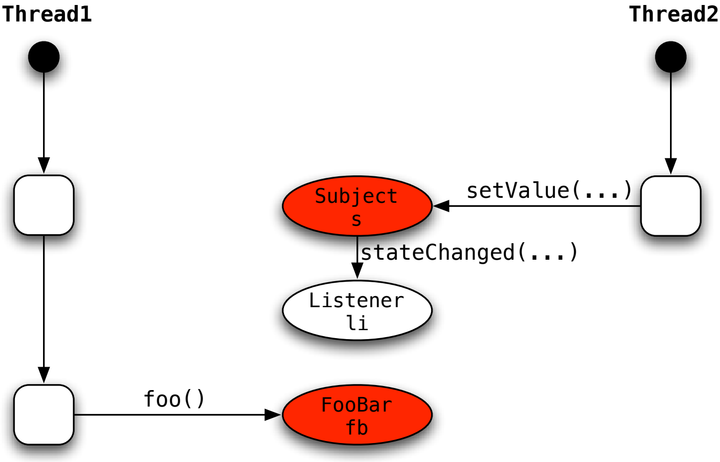
Motivation

Multithreading in C-like languages – Example 1



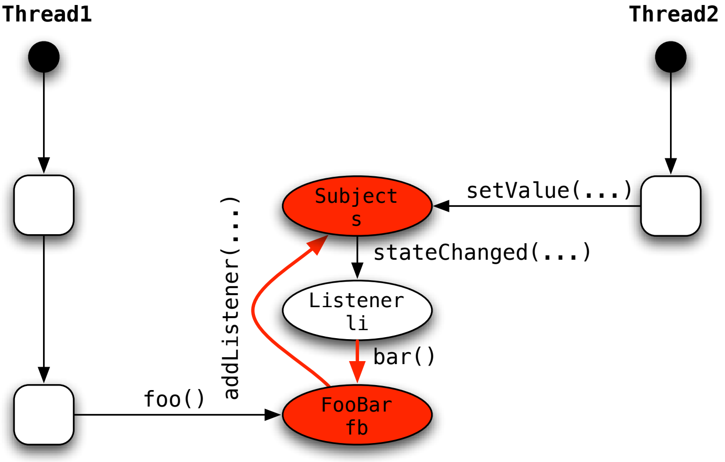
Motivation

Multithreading in C-like languages – Example 1



Motivation

Multithreading in C-like languages – Example 1



Motivation

Multithreading in C-like languages – Example 1

Programming with locks increases complexity and error-proneness.

- Libraries (objects) with locks are no longer black boxes
- The user have to know about implementation details (“which method uses which lock?”)

Motivation

Multithreading in C-like languages – Example 2

```
class Foo { // immutable
  static Foo* ptr;
  static Foo* instance() {
    // 1st test
    if (ptr == nullptr) {
      Lock lock;
      // 2nd test
      if (ptr == nullptr)
        ptr = new Foo;
    }
    return ptr;
  }
  // ...
}
```

Adapted from: *"C++ and the Perils of Double-Checked Locking"* (Meyers & Alexandrescu, 2004)

Motivation

Multithreading in C-like languages – Example 2

```
class Foo { // immutable
  static Foo* ptr;
  static Foo* instance() {
    // 1st test
    if (ptr == nullptr) {
      Lock lock;
      // 2nd test
      if (ptr == nullptr)
        ptr = new Foo;
    }
    return ptr;
  }
  // ...
}
```

Problem:

"`ptr = new Foo`" is **not** atomic:

1. Allocate memory
2. Call constructor of `Foo`
3. Assign memory address to `ptr`

Adapted from: *"C++ and the Perils of Double-Checked Locking"* (Meyers & Alexandrescu, 2004)

Motivation

Multithreading in C-like languages – Example 2

```
class Foo { // immutable
  static Foo* ptr;
  static Foo* instance() {
    // 1st test
    if (ptr == nullptr) {
      Lock lock;
      // 2nd test
      if (ptr == nullptr)
        ptr = new Foo;
    }
    return ptr;
  }
  // ...
}
```

Problem:

"`ptr = new Foo`" is **not** atomic:

1. Allocate memory
2. Call constructor of `Foo`
3. Assign memory address to `ptr`

If 3 happens before 2, a second thread might deallocate `ptr` *before* the constructor was called (undefined behavior).

Adapted from: *"C++ and the Perils of Double-Checked Locking"* (Meyers & Alexandrescu, 2004)

Motivation

Multithreading in C-like languages – Example 2

Concurrency with low-level primitives requires a lot of expert knowledge.

- Seemingly correct code *can* lead to undefined behavior
- Almost impossible to verify by testing
- An implementation can be thread-safe on a uniprocessor machine (“timeslice-based parallelism”) but can lead to race conditions on a multiprocessor machine (true hardware concurrency)

Concurrency Approaches

Transactional Memory

- Race condition free shared memory
- Reads & writes are atomic and transactional
- "all or nothing" writes
- Readers don't interfere writers and vice versa
- In hardware or software (e.g. `Clojure`)

Concurrency Approaches

Join-Calculus (JoCaml)

1. `def fruit(f) & cake(c) = print_endline (f^‘‘ ’’^ c) ; 0`
   `val fruit :  string Join.chan = <abstr>`
   `val cake :  string Join.chan = <abstr>`

2. `spawn fruit ‘‘apple’’ & cake ‘‘pie’’`

3. `spawn fruit ‘‘apple’’ & fruit ‘‘lime’’ & cake ‘‘pie’’ & cake ‘‘torte’’`

- Join-calculus is a member of the π calculus family
- Processes communicate (synchronize) via ports

Concurrency Approaches

Join-Calculus (JoCaml)

1.
   ```
   def fruit(f) & cake(c) = print_endline (f^“ ”^ c) ; 0
   val fruit :  string Join.chan = <abstr>
   val cake :  string Join.chan = <abstr>
   ```

2.
   ```
   spawn fruit “apple” & cake “pie”
   ```

3.
   ```
   spawn fruit “apple” & fruit “lime” & cake “pie” & cake “torte”
   ```

- Join-calculus is a member of the π calculus family
- Processes communicate (synchronize) via ports
- Source code example:
  1. Define two ports and the guarded process `print_endline ...`
  2. Prints “`apple pie`”
  3. Prints “`apple pie`”, “`lime torte`” *or* “`apple torte`”, “`lime pie`”

Concurrency Approaches

Summary

There are basically two approaches:

- Provide a safe (free of race conditions) shared memory
- Model concurrent tasks/processes as independent components, communicating via messages/channels/ports

Concurrency Approaches

Summary

There are basically two approaches:

- Provide a safe (free of race conditions) shared memory
    - `Clojure`
    - Intel `C++` STM Compiler
    - ...
- Model concurrent tasks/processes as independent components, communicating via messages/channels/ports

Concurrency Approaches

Summary

There are basically two approaches:

- Provide a safe (free of race conditions) shared memory
  - `Clojure`
  - Intel `C++` STM Compiler
  - ...
- Model concurrent tasks/processes as independent components, communicating via messages/channels/ports
  - `Erlang` (resp. the Actor Model in general)
  - `Google Go` (channel based communication)
  - ...

Concurrency Approaches

Summary

We have to enable "average programmers" to write both (multiprocessor) safe and scalable applications.

- No shared memory *or* transactional memory
- Explicit communication of independent software components (channels, ports, ...) instead of implicit communication via shared memory segments and locks
- High-level concepts with reasonable metaphors

The Actor Model

Definition

Actors are self-contained, concurrent computation entities, that ...

- Communicate only via (asynchronous) message passing
- Don't share memory
- Can create ("spawn") new Actors

The Actor Model

Benefits

- Race conditions are avoided by design (no shared memory comm.)

The Actor Model

Benefits

- Race conditions are avoided by design (no shared memory comm.)
- High-level, explicit communication

The Actor Model

Benefits

- Race conditions are avoided by design (no shared memory comm.)
- High-level, explicit communication
- Applies to both concurrency *and* distribution
  (network transparency thanks to message passing)

The Actor Model

Benefits

- Race conditions are avoided by design (no shared memory comm.)
- High-level, explicit communication
- Applies to both concurrency *and* distribution (network transparency thanks to message passing)
- Inspired several implementations either as basis for languages (`Erlang`) or as library/framework (`Scala`, `Kilim`, `Retlang`, ...)

Why C++?

- Thousands of active developers and huge, existing code bases
- Still no high-level concurrency abstraction in `C++11`
- New language features (lambda expression, variadic templates, ...) ease development of libraries as internal DSL

libcppa

Goals

- An actor library for `C++` as internal DSL

- An actor library for `C++` as internal DSL
- Network transparency (ease distribution)

libcppa

Goals

- An actor library for `C++` as internal DSL
- Network transparency (ease distribution)
- Pattern matching for messages

`libcppa`

Goals

- An actor library for `C++` as internal DSL
- Network transparency (ease distribution)
- Pattern matching for messages
- Extensible group (N:M) communication API
  - In-process (event handling)
  - Inter-process (services, system-wide events)
  - Network layer multicast (IP, Overlay, H∀Mcast, ...)

`libcppa`

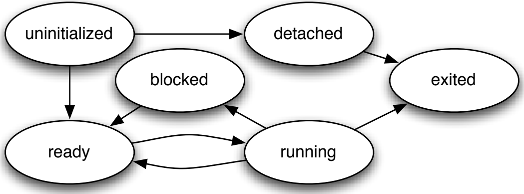
Goals

- An actor library for `C++` as internal DSL
- Network transparency (ease distribution)
- Pattern matching for messages
- Extensible group (N:M) communication API
  - In-process (event handling)
  - Inter-process (services, system-wide events)
  - Network layer multicast (IP, Overlay, H∀Mcast, ...)
- Lightweight, scheduled Actors

libcppa

Actor Scheduling

- Actors are lighweight tasks, scheduled in a thread pool



- Small overhead for spawn/delete operations

libcppa

Group Communication

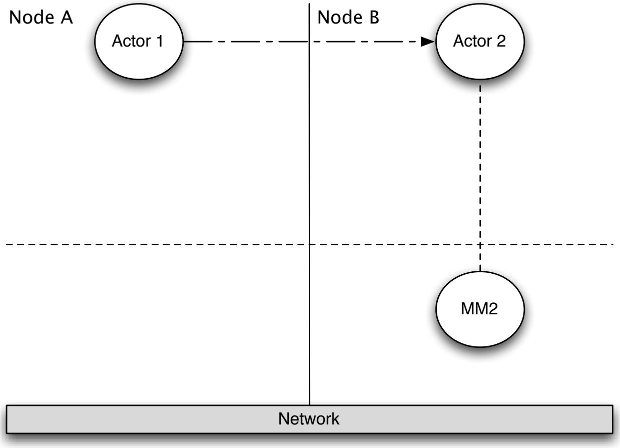
- Actors can join and leave groups
- A group is identified by `module name` + `group identifier`
- Users can add new modules (e.g. for “ip”, “H∀Mcast”, ...)

```
class group : // ...
{
  // ... virtual member functions ...
  static group* get(const std::string& module_name,
                    const std::string& group_identifier);
  static void add_module(module*);
};
```

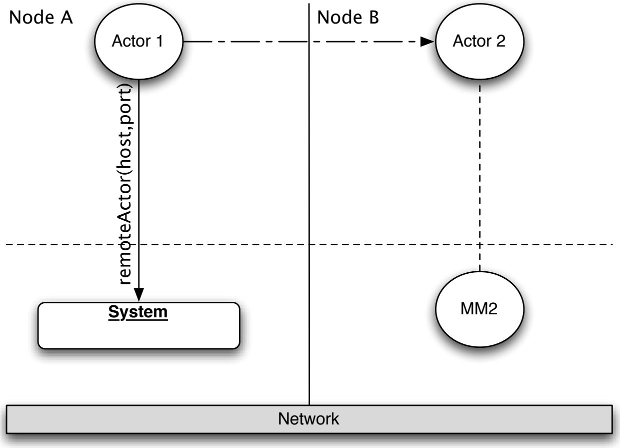
- Actors can join and leave groups
- A group is identified by `module name` + `group identifier`
- Users can add new modules (e.g. for “ip”, “H∀Mcast”, ...)

libcppa

Network Transparency



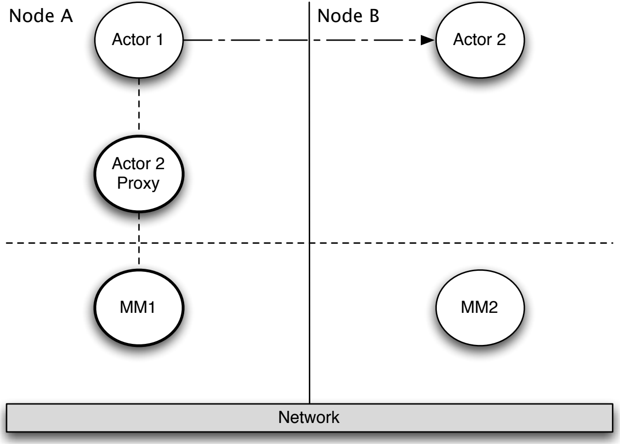
Send a message to a remote (“published”) actor



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

libcppa

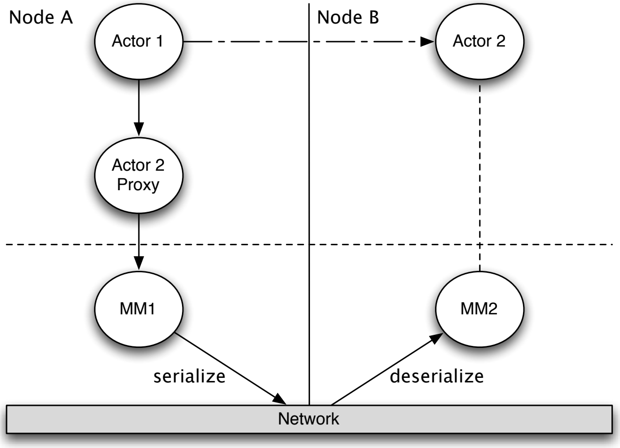
Network Transparency



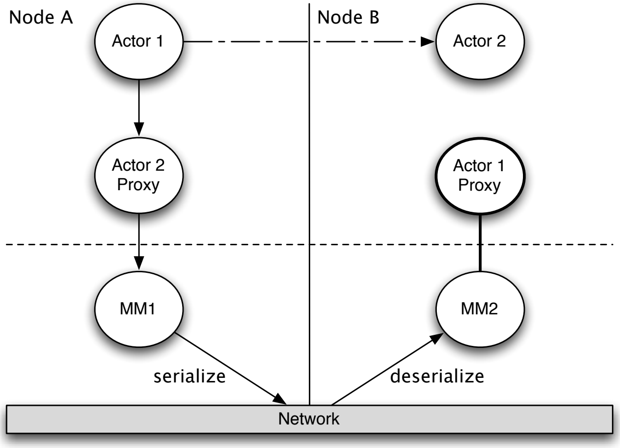
Send a message to a remote (“published”) actor

`libcppa`

Network Transparency



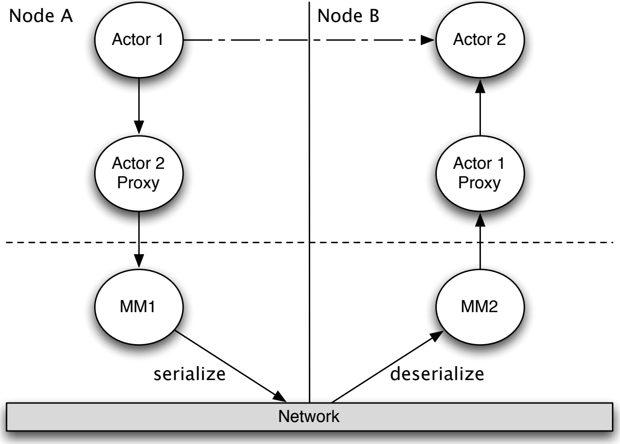
Send a message to a remote ("published") actor



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

libcppa

Network Transparency



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

libcppa

Example

```
#include "cppa/cppa.hpp"
using namespace cppa;

void ping();
void pong(actor_ptr ping_actor);

int main(int, char**)
{
  spawn(pong, spawn(ping));
  await_all_others_done();
  return 0;
}
```

libcppa

Example

```
void ping()
{
    receive_loop
    (
        on<atom("Pong"), int>() >> [](int value)
        {
            reply(atom("Ping"), value + 1);
        }
    );
}
```

libcppa

Example

```
void pong(actor_ptr ping_actor)
{
    link(ping_actor);
    // kickoff
    ping_actor << make_tuple(atom("Pong"), 0);
    // or: send(ping_actor, atom("Pong"), 0);
    receive_loop
    (
        on<atom("Ping"), int>(9) >> []()
        {
            // terminate with non-normal exit reason
            quit(exit_reason::user_defined);
        },
        on<atom("Ping"), int>() >> [](int value)
        {
            reply(atom("Pong"), value + 1);
        }
    );
}
```

Questions & Answers

Thank you for your attention!

Questions?