`libcppa`

An implementation of the actor model for C++

User Manual

Dominik Charousset

June 15, 2012

Contents

1 First Steps

To compile `libcppa`, you will need Automake, Libtool, the Boost Library and a C++11 compiler. To get and compile the sources, open a terminal and type:

```
git clone git://github.com/Neverlord/libcppa.git
cd libcppa
autoreconf -i
./configure
make
make install [as root, optional]
```

It is recommended to run the unit tests as well.

```
./unit_testing/unit_tests
```

Please submit a bug report that includes (a) your compiler version, (b) your OS, and (c) the output of the unit tests if an error occurs.

1.1 Features Overview

- Lightweight, fast and efficient actor implementations
- Network transparent messaging
- Error handling based on Erlang's failure model
- Pattern matching for messages as internal DSL to ease development
- Thread-mapped actors and on-the-fly conversions for soft migration of existing applications
- Group communication based on Publish/Subscribe

1.2 Supported Compilers

- GCC $\geq$ 4.7
- Clang $\geq$ 3.2

1.3 Supported Operating Systems

- Linux
- Mac OS X

1.4 Hello World Example

```
#include <string>
#include <iostream>
#include "cppa/cppa.hpp"

using namespace cppa;

void echo_actor() {
    // wait for a message
    receive (
        // invoke this lambda expression if we receive a string
        on<std::string>() >> [](const std::string& what) {
            // prints "Hello World!"
            std::cout << what << std::endl;
            // replies "!dlroW olleH"
            reply(std::string(what.rbegin(), what.rend()));
        }
    );
}

int main() {
    // create a new actor that invokes the function echo_actor
    auto hello_actor = spawn(echo_actor);
    // send "Hello World!" to our new actor
    // note: libcppa converts string literals to std::string
    send(hello_actor, "Hello World!");
    // wait for a response and print it
    receive (
        on<std::string>() >> [](const std::string& what) {
            // prints "!dlroW olleH"
            std::cout << what << std::endl;
        }
    );
    // wait until all other actors we've spawned are done
    await_all_others_done();
    // done
    return 0;
}
```

2 Copy-On-Write Tuples

The message passing implementation of libcppa uses tuples with call-by-value semantic. Hence, it is not necessary to declare message types, though, libcppa allows users to use user-defined types in messages (see Section 9.1). A call-by-value semantic would cause multiple copies of a tuple if it is send to multiple actors. To avoid unnecessary copying overhead, libcppa uses a copy-on-write tuple implementation. A tuple is implicitly shared between any number of actors, as long as all actors demand only read access. Whenever an actor demands write access, it has to copy the data first if more than one reference to it exist. Thus, race conditions cannot occur and each tuple is copied only if necessary.

The interface of cow_tuple strictly distinguishes between const and non-const access. The template function get returns an element as immutable value, while get_ref explicitly returns a mutable *reference* to the required value and detaches the tuple if needed. We do not provide a const overload for get, because this would cause to unintended, and thus unnecessary, copying overhead.

```
auto x1 = make_cow_tuple(1, 2, 3);      // cow_tuple<int, int, int>
auto x2 = x1;                           // cow_tuple<int, int, int>
assert(&get<0>(x1) == &get<0>(x2));     // point to the same data
get_ref<0>(x1) = 10;                    // detaches x1 from x2
//get<0>(x1) = 10;                      // compiler error
assert(get<0>(x1) == 10);               // x1 is now {10, 2, 3}
assert(get<0>(x2) == 1);                // x2 is still {1, 2, 3}
assert(&get<0>(x1) != &get<0>(x2));     // no longer the same
```

2.1 Dynamically Typed Tuples

The class any_tuple represents a tuple without static type information. All messages send between actors use this tuple type. The type information can be either explicitly accessed for each element or the original tuple, or a subtuple of it, can be restored using tuple_cast. Users of libcppa usually do not need to know about any_tuple, since it is used “behind the scenes”. However, any_tuple can be created from a cow_tuple or by using make_any_tuple, as shown below.

```
auto x1 = make_cow_tuple(1, 2, 3);      // cow_tuple<int, int, int>
any_tuple x2 = x1;                      // any_tuple
any_tuple x3 = make_cow_tuple(10, 20); // any_tuple
auto x4 = make_any_tuple(42);           // any_tuple
```

2.2 Casting Tuples

The function `tuple_cast` restores static type information from an `any_tuple` object. It returns an `option` (see Section 10.1) for a `cow_tuple` of the requested types.

```
auto x1 = make_any_tuple(1, 2, 3);
auto x2_opt = tuple_cast<int, int, int>(x1);
assert(x2_opt.valid());
auto x2 = *x2_opt;
assert(get<0>(x2) == 1);
assert(get<1>(x2) == 2);
assert(get<2>(x2) == 3);
```

The function `tuple_cast` can be used with wildcards (see Section 3.4) to create a view to a subset of the original data. No elements are copied, unless the tuple becomes detached.

```
auto x1 = make_cow_tuple(1, 2, 3);
any_tuple x2 = x1;
auto x3_opt = tuple_cast<int, anything, int>(x2);
assert(x3_opt.valid());
auto x3 = *x3_opt;
assert(get<0>(x3) == 1);
assert(get<1>(x3) == 3);
assert(&get<0>(x3) == &get<0>(x1));
assert(&get<1>(x3) == &get<2>(x1));
```

3 Pattern Matching

C++ does not provide pattern matching facilities. A general pattern matching solution for arbitrary data structures would require a language extension. Hence, we decided to restrict our implementation to tuples, to be able to use an internal domain-specific language approach.

3.1 Basics

A match expression begins with a call to the function `on`, which returns an intermediate object providing the member function `when` and `operator>>`. The right-hand side of the operator denotes a callback, usually a lambda expression, that should be invoked if a tuple matches the types given to `on`, as shown in the example below.

```
on<int>() >> [](int i) { /*...*/ }
on<int, float>() >> [](int i, float f) { /*...*/ }
on<int, int, int>() >> [](int a, int b, int c) { /*...*/ }
```

The result of `operator>>` is a partial function that is defined for the types given to `on`. A comma separated list of partial functions results in a single partial function that sequentially evaluates its subfunctions. At most one callback is invoked, since the evaluation stops at the first match.

```
auto fun = (
  on<int>() >> [](int i) {
    // case1
  },
  on<int>() >> [](int i) {
    // is never invoked, since case1 always matches first
  }
);
```

Note: A list of partial function definitions must be enclosed in brackets if assigned to a variable. Otherwise, the compiler assumes commas to separate variable definitions.

The function “`on`” can be used in two ways. Either with template parameters only or with function parameters only. The latter version deduces all types from its arguments and matches for both type and value. The template “`val`” can be used to match only the type of a parameter.

```
on(42) >> [](int i) { assert(i == 42); }
on("hello world") >> []() { /* ... */ }
on("print", val<std::string>) >> [](const std::string& what) {
  // ...
}
```

Note: The given callback can have less arguments than given to the pattern. But it is only allowed to skip arguments from left to right.

```
on<int, float, double>() >> [](double) { /*...*/ }                // ok
on<int, float, double>() >> [](float, double) { /*...*/ }         // ok
on<int, float, double>() >> [](int, float, double) { /*...*/ } // ok

on<int, float, double>() >> [](int i) { /*...*/ } // compiler error
```

3.2 Atoms

Assume an actor provides a mathematical service for integers. It takes two arguments, performs a predefined operation and returns the result. It cannot determine an operation, such as multiply or add, by receiving two operands. Thus, the operation must be encoded into the message. The Erlang programming language introduced an approach to use non-numerical constants, so-called *atoms*, which have an unambiguous, special-purpose type and do not have the runtime overhead of string constants. Atoms are mapped to integer values at compile time in `libcppa`. This mapping is guaranteed to be collision-free but limits atom literals to ten characters and prohibits special characters. Legal characters are "`_0-9A-Za-z`" and the whitespace character. Atoms are created using the `constexpr` function `atom`, as the following example illustrates.

```
on<atom("add"), int, int>() >> [](int a, int b) { /*...*/ },
on<atom("multiply"), int, int>() >> [](int a, int b) { /*...*/ },
// ...
```

Note: The current implementation cannot enforce the restrictions at compile time, except for a length check. Each invalid character is mapped to the whitespace character, why the assertion `atom("!?") != atom("?!")` is not true. However, this issue will fade away after user-defined literals become available in mainstream compilers, because it is then possible to raise a compiler error for invalid characters.

3.3 Reducing Redundancy with "`arg_match`" and "`on_arg_match`"

Our previous example is quite verbose and redundant, since you have to type the types twice – as template parameter and as argument type for the lambda. To avoid such redundancy, `arg_match` can be used as last argument to the function `on`. This causes the compiler to deduce all further types from the signature of the given callback.

```
on<atom("add"), int, int>() >> [](int a, int b) { /*...*/ }
// is equal to:
on(atom("add"), arg_match) >> [](int a, int b) { /*...*/ }
```

Note that the second version does call `on` without template parameters. Furthermore, `arg_match` must be passed as last parameter. If all types should be deduced from the callback signature, `on_arg_match` can be used. It is equal to `on(arg_match)`.

```
on_arg_match >> [](const std::string& str) { /*...*/ }
```

3.4 Wildcards

The type `anything` can be used as wildcard to match any number of any types. A pattern created by `on<anything>()` or its alias `others()` is useful to define a default case. For patterns defined without template parameters, the `constexpr` value `any_vals` can be used as function argument. The constant `any_vals` is of type `anything` and is nothing but syntactic sugar for defining patterns.

```
on<int, anything>() >> [](int i) {
  // tuple with int as first element
},
on(any_vals, arg_match) >> [](int i) {
  // tuple with int as last element
  // "on(any_vals, arg_match)" is equal to "on(anything{}, arg_match)"
},
others() >> {
  // everything else (default handler)
  // "others()" is equal to "on<anything>()" and "on(any_vals)"
}
```

3.5 Guards

Guards can be used to constrain a given match statement by using placeholders, as the following example illustrates.

```
using namespace cppa::placeholders; // contains _x1 - _x9

on<int>().when(_x1 % 2 == 0) >> []() {
  // int is even
},
on<int>() >> []() {
  // int is odd
}
```

Guard expressions are a lazy evaluation technique. The placeholder `_x1` is substituted with the first value of a given tuple. All binary comparison and arithmetic operators are supported, as well as `&&` and `||`. In addition, there are two functions designed to be used in guard expressions: `gref` ("guard reference") and `gcall` ("guard function call"). The function `gref` creates a reference wrapper. It is similar to `std::ref` but it is always `const` and "lazy", i.e., evaluated when a tuple arrives. A few examples to illustrate some pitfalls:

```
int val = 42;

on<int>().when(_x1 == val)          // (1) matches if _x1 == 42
on<int>().when(_x1 == gref(val))    // (2) matches if _x1 == val
on<int>().when(_x1 == std::ref(val)) // (3) ok, because of placeholder
others().when(gref(val) == 42)      // (4) matches everything
                                    //     as long as val == 42
others().when(std::ref(val) == 42)  // (5) compiler error
```

Statement `(5)` is evaluated immediately and returns a boolean, whereas statement `(4)` creates a valid guard expression. Thus, you should always use `gref` instead of `std::ref` to avoid subtle errors.

The second function, `gcall`, encapsulates a function call. Its usage is similar to `std::bind`, but there is also a short version for unary functions: `gcall(fun, _x1)` is equal to `_x1(fun)`.

```
auto vec_sorted = [](std::vector<int> const& vec) {
  return std::is_sorted(vec.begin(), vec.end());
};

on<std::vector<int>>().when(gcall(vec_sorted, _x1)) // is equal to:
on<std::vector<int>>().when(_x1(vec_sorted)))
```

3.5.1 Placeholder Interface

```
template<int X>
struct guard_placeholder;
```

Member functions (`x` represents the value at runtime, `y` represents an iterable container)

`size()`	Returns `x.size()`
`empty()`	Returns `x.empty()`
`not_empty()`	Returns `!x.empty()`
`front()`	Returns an `option` (see Section 10.1) to `x.front()`
`in(y)`	Returns `true` if `y` contains `x`, `false` otherwise
`not_in(y)`	Returns `!in(y)`

3.5.2 Examples

```
using namespace std;
typedef vector<int> ivec;

vector<string> strings{"abc", "def"};

on<ivec>().when(_x1.front() == 0) >> [](const ivec& v) {
  // note: we don't have to check whether _x1 is empty in our guard,
  //       because '_x1.front()' returns an option for a
  //       reference to the first element
  assert(v.size() >= 1);
  assert(v.front() == 0);
},
on<int>().when(_x1.in({10, 20, 30})) >> [](int i) {
  assert(i == 10 || i == 20 || i == 30);
},
on<string>().when(_x1.not_in(strings)) >> [](const string& str) {
  assert(str != "abc" && str != "def");
},
on<string>().when(_x1.size() == 10) >> [](const string& str) {
```

```
  // ...
}
```

3.6 Projections and Extractors

Projections perform type conversions or extract data from a given input. If a callback expects an integer but the received message contains a string, a projection can be used to perform a type conversion on-the-fly. This conversion should be free of side-effects and, in particular, shall not throw exceptions, because a failed projection is not an error. A pattern simply does not match if a projection failed. Let us have a look at a simple example.

```
auto intproj = [](const string& str) -> option<int> {
  char* endptr = nullptr;
  int result = static_cast<int>(strtol(str.c_str(), &endptr, 10));
  if (endptr != nullptr && *endptr == '\0') return result;
  return {};
};
auto fun = (
  on(intproj) >> [](int i) {
    // case 1, successfully converted a string
  },
  on_arg_match >> [](const string& str) {
    // case 2, str is not an integer
  }
);
```

The lambda `intproj` is a `string` ⇒ `int` projection, but note that it does not return an integer. It returns `option<int>`, because the projection is not guaranteed to always succeed. An empty `option` indicates, that a value does not have a valid mapping to an integer. A pattern does not match if a projection failed.

Note: Functors used as projection must take exactly one argument and must return a value. The types for the pattern are deduced from the functor's signature. If the functor returns an `option<T>`, then `T` is deduced.

4 Actors

`libcppa` provides three actor implementations, each covering a particular use case. The class `local_actor` is the base class for all implementations, except for (remote) proxy actors.

4.1 Local Actors

The class `local_actor` describes a local running actor. It provides a common interface for actor operations like trapping exit messages or finishing execution.

4.1.1 "Keyword" `self`

The `self` pointer is an essential ingredient of our design. It identifies the running actor similar to the implicit `this` pointer identifying an object within a member function. Unlike `this`, though, `self` is not limited to a particular scope. The `self` pointer is used implicitly, whenever an actor calls functions like `send` or `receive`, but can be accessed to use more advanced actor operations such as linking to another actor, e.g., by calling `self->link_to(other)`.

A thread that accesses `self` is converted on-the-fly to an actor if needed. Hence, "everything is an actor" in `libcppa`.

4.1.2 Interface

```
class local_actor;
```

Member functions

Observers

`bool trap_exit()`	Checks whether this actor traps exit messages
`bool chaining()`	Checks whether this actor uses the "chained send" optimization (see Section 5.2)
`any_tuple last_dequeued()`	Returns the last message that was dequeued from the actor's mailbox **Note**: Only set during callback invocation
`actor_ptr last_sender()`	Returns the sender of the last dequeued message **Note**$_1$: Only set during callback invocation **Note**$_2$: Used by the function `reply` (see Section 5.1)

Modifiers

`void trap_exit(bool enabled)`	Enables or disables trapping of exit messages
`void chaining(bool enabled)`	Enables or disables chained send

4.2 Types of Actors

We have already shown the differences of context-switching and event-based actors in Section 6. Context-switching and event-based actors are scheduled cooperatively in a thread pool. Developers can opt-out of this cooperative scheduling by using thread-mapped actors.

4.2.1 Thread-Mapped

This is the implicit type of all threads that were converted to actors implicitly. Furthermore, this type is used for actors created with `spawn<detached>`. It is recommended to use detached actors whenever an actor could starve other actors, e.g., by calling time-expensive, blocking system calls. Detached actors also could be used for actors that need to stay responsive, independent of the current work load. However, threads to not scale well. Hence, detached actors should be used only in small numbers for long-lived actors.

4.2.2 Context-Switching

Context-switching actors have an own control flow allow to spawn arbitrary functions as actors. The downside of context-switching actors is that each actor needs to allocate its own stack. This seriously impacts the performance for short-lived actors and is not applicable for large-scale actor systems. This implementations allows for an easy migration of previously threaded application, but a system should not contain more than a few hundred context-switching actors.

4.2.3 Event-Based

This is the recommended implementation for most use cases. Event-based actors have a small memory footprint and are thus very lightweight. The behavior-based API makes it harder to nest receives, but this implementation clearly scales best. See Section 6.2 for a few examples.

4.3 Management

`libcppa` adapts Erlang's well-established fault propagation model. It allows to build actor subsystem in which either all actors are alive or have collectively failed.

4.3.1 Links

Linked actors monitor each other. An actor sends an exit message to all of its links as part of its termination. The default behavior for actors receiving such an exit message is to die for the same reason, if the exit reason is non-normal. Actors can *trap* exit messages to handle them manually.

```
auto worker = spawn(...);
// receive exit messages as regular messages
self->trap_exit(true);
// monitor spawned actor
self->link_to(worker);
// wait until worker exited
receive (
  on(atom("EXIT"), exit_reason::normal) >> []() {
    // worker finished computation
  },
  on<atom("EXIT"), std::uint32_t>() >> [](std::uint32_t reason) {
    // worker died unexpectedly
  }
);
```

4.3.2 Monitors

A monitor observes the lifetime of an actor. Monitored actors send a down message to all monitors as part of their termination. Unlike exit messages, down messages are always treated like any other ordinary message.

```
auto worker = spawn(...);
// monitor spawned actor
self->monitor(worker);
// wait until worker exited
receive (
  on(atom("DOWN"), exit_reason::normal) >> []() {
    // worker finished computation
  },
  on(atom("DOWN"), arg_match) >> [](std::uint32_t reason) {
    // worker died unexpectedly
  }
);
```

Monitors are redundant. Hence, actors will receive one down message for each monitor.

4.3.3 Error Codes

All error codes are defined in the namespace `cppa::exit_reason`.

`normal`	Actor finished execution without error
`unhandled_exception`	Actor was killed due to an unhandled exception
`unallowed_function_call`	Indicates that an event-based actor tried to use blocking receive calls
`remote_link_unreachable`	Indicates that a remote actor became unreachable, e.g., due to connection error
`user_defined`	Minimum value for user-defined exit codes

4.3.4 Attach Cleanup Code to an Actor

Actors can attach cleanup code to other actors. This code is executed immediately if the actor has already exited. Keep in mind that `self` refers to the currently running actor. Thus, `self` refers to the terminating actor and not to the actor that attached a functor to it.

```
auto worker = spawn(...);
actor_ptr observer = self;
// "monitor" spawned actor
worker->attach_functor([observer](std::uint32_t reason) {
  // this callback is invoked from worker => self == worker
  send(observer, atom("DONE"));
});
// wait until worker exited
receive (
  on(atom("DONE")) >> []() {
    // worker terminated
  }
);
```

Note: It is possible to attach code to remote actors, but the cleanup code will run on the local machine.

5 Sending Messages

Messages can be sent by using either the function `send` or `operator<<`. The variadic template function `send` has the following signature.

```
template<typename... Args>
void send(actor_ptr whom, Args&&... what);
```

The variadic template pack `what...` is converted to a dynamically typed tuple (see Section 2.1) and then enqueued to the mailbox of `whom`. The following example shows two equal sends, one using `send` and the other using `operator<<`.

```
actor_ptr other = spawn(...);
send(other, 1, 2, 3);
other << make_any_tuple(1, 2, 3);
```

Using the function `send` is more compact, but does not have any other benefit. However, note that you should not use `send` if you already have an instance of `any_tuple`, because it creates a new tuple containing the old one.

```
actor_ptr other = spawn(...);
auto msg = make_any_tuple(1, 2, 3);
send(other, msg); // oops, creates a new tuple that contains msg
other << msg; // ok
```

5.1 Replying to Messages

During callback invokation, `self->last_sender()` is set. This identifies the sender of the received message and is used implicitly by the function `reply`.

```
template<typename... Args>
void reply(Args&&... what) {
  send(self->last_sender(), std::forward<Args>(what)...);
}
```

As you can see, `reply` is nothing but syntactic sugar for `send`.

However, one could also use `self->last_sender()` directly for replying to messages. For example, `self->last_sender() << self->last_dequeued()` sends the received message back to the sender.

5.2 Chained Send

Sending a message to a cooperatively scheduled actor usually causes the receiving actor to be put into the scheduler's job queue if it is currently blocked, i.e., is waiting for a new message. This job queue is accessed by worker threads. The *chaining* optimization does not cause the receiver to be put into the scheduler's job queue if it is currently blocked. The receiver is stored as successor of the currently running actor instead. Hence, the active worker thread does not need to access the job queue, which significantly speeds up execution. However, this optimization can be inefficient if an actor does first sends a message and then starts computation.

```
void foo(actor_ptr other) {
  send(other, ...);
  very_long_computation();
  // ...
}

int main() {
  // ...
  auto a = spawn(...);
  auto b = spawn(foo, a);
  // ...
}
```

The example above illustrates an inefficient work flow. The actor `other` is marked as successor of the `foo` actor but its execution is delayed until `very_long_computation()` is done. In general, actors should follow the work flow `receive` ⇒`compute` ⇒ `send results`. However, this optimization can be disabled by calling `self->chaining(false)` if an actor does not match this work flow.

```
void foo(actor_ptr other) {
  self->chaining(false);   // disable chaining optimization
  send(other, ...);        // not delayed by very_long_compuation
  very_long_computation();
  // ...
}
```

5.3 Delay Messages

Messages can be delayed, e.g., to implement time-based polling strategies, by using the two functions `delayed_send` and `delayed_reply`.

```
delayed_send(self, std::chrono::seconds(1), atom("poll"));
receive_loop (
  on(atom("poll")) >> []() {
    // poll a resource...
    // schedule next polling
    delayed_send(self, std::chrono::seconds(1), atom("poll"));
  }
);
```

6 Receiving Messages

Event-based actors differ in receiving messages from context-switching and thread-mapped actors: the former define their behavior as a message handler that is invoked whenever a new messages arrives in the actor's mailbox, whereas the latter use an explicit receive function. The current *behavior* of an actor is its response to the *next* incoming message and includes (a) sending messages to other actors, (b) creation of more actors, and (c) setting a new behavior.

6.1 Context-Switching and Thread-Mapped Actors

The function `receive` sequentially iterates over all elements in the mailbox beginning with the first. It takes a partial function that is applied to the elements in the mailbox until an element was matched by the partial function. An actor calling `receive` is blocked until it successfully dequeued a message from its mailbox or an optional timeout occurs.

```
receive (
  on<int>().when(_x1 > 0) >> // ...
);
```

The code snippet above illustrates the use of `receive`. Note that the partial function passed to `receive` is a temporary object at runtime. Hence, using receive inside a loop would cause creation of a new partial function on each iteration. `libcppa` provides three predefined receive loops to provide a more efficient but yet convenient way of defining receive loops.

```
//DON'T

for (;;) {
  receive (
    // ...
  );
}
```

```
//DO

receive_loop (
  // ...
);
```

```
std::vector<int> results;
for (size_t i = 0; i < 10; ++i) {
  receive (
    on<int>() >> [&](int value) {
      results.push_back(value);
    }
  );
}
```

```
std::vector<int> results;
size_t i = 0;
receive_for(i, 10) (
  on<int>() >> [&](int value) {
    results.push_back(value);
  }
);
```

```
size_t received = 0;
do {
  receive (
    others() >> [&]() {
      ++received;
    }
  );
} while (received < 10);
```

```
size_t received = 0;
do_receive (
  others() >> [&]() {
    ++received;
  }
).until(gref(received) >= 10);
```

The examples above illustrate the usage of the three loops `receive_loop`, `receive_for` and `do_receive(...)`. It is possible to nest receives, and receive loops.

```
receive_loop (
  on<int>() >> [](int value1) {
    receive (
      on<float>() >> [&](float value2) {
        cout << value1 << " => " << value2 << endl;
      }
    );
  }
);
```

6.2 Event-Based Actors

An event-based actor defined as a class uses `become` to set its behavior. The given behavior is then executed until it is replaced by another call to `become` or the actor finishes execution.

```
class printer : public event_based_actor {
  void init() {
    become (
      others() >> []() {
        cout << to_string(self->last_received()) << endl;
      }
    );
  }
};
```

Actors inheriting from `event_based_actor` must implement the virtual member function `init`. Another way to implement event-based actors is provided by the class `fsm_actor`. This base class automatically calls `become(&init_state)` in its `init` member function. Hence, a subclass must only provide a member of type `behavior` named `init_state`.

```
struct fsm_printer : fsm_actor<fsm_printer> {
  behavior init_state = (
    others() >> []() {
      cout << to_string(self->last_received()) << endl;
    }
  );
};
```

Note that `fsm_actor` uses the curiously recurring template pattern. Thus, the derived class must be given as template parameter. This technique allows `fsm_actor` to access the `init_state` member of a derived class.

The third event-based base class for event-based actors is `stacked_event_based_actor` and provides the additional member function `unbecome`. As the name suggests, this implementation provides a *behavior stack*. A call to `become` pushes a new behavior to the stack, whereas a call to `unbecome` pops an element. An actor always executes the behavior on top of its stack. Thus, `unbecome` returns to the previously set behavior. This can be used to “nest” receives, but can lead to stack overflows.

```
// receives {int, float} sequences
struct testee : stacked_event_based_actor {
  void init() {
    become (
      on<int>() >> [=](int value1) {
        become (
          on<float>() >> [=](float value2) {
            cout << value1 << " => " << value2 << endl;
            unbecome();
          }
        );
      }
    );
  }
};
```

An actor finishes execution with normal exit reason if the behavior stack is empty after calling `unbecome`.

6.3 Common Pitfalls

- `become` does *not* block. It always returns immediately. Thus, you should *always* capture by value in event-based actors, because all references on the stack will cause undefined behavior if a lambda is executed.

6.4 Timeouts

During receive, an actor is blocked until it dequeues a message from its mailbox that matches the given pattern. If no such message ever arrives, the actor is blocked forever. This might be desirable if the actor only provides a service and should not do anything else. But often, we need to be able to recover if an expected messages does not arrive within a certain time period. The following examples illustrates the usage of `after` to define a timeout.

```
#include <chrono>
#include <iostream>

using std::cout;
using std::cerr;
using std::endl;

receive(
  on_arg_match >> [](int i) { /* ... */ },
  on_arg_match >> [](float i) { /* ... */ },
  others() >> []() { /* ... */ },
  after(std::chrono::seconds(10)) >> []() {
    cerr << "received nothing within 10 seconds..." << endl;
    // ...
  }
);

receive(
  after(std::chrono::milliseconds(50)) >> []() {
    cerr << "slept for 50ms" << endl;
  }
);

receive(
  on_arg_match >> [](int i) {
    cout << "found: " << i << endl;
  },
  after(std::chrono::seconds(0)) >> []() {
    cout << "no integer found in mailbox" << endl;
  }
);
```

Callbacks given as timeout handler must have zero arguments. Any number of patterns can precede the timeout definition, but "`after`" must always be the final statement. Using a zero-duration timeout causes `receive` to not block.

`libcppa` supports `minutes`, `seconds`, `milliseconds` and `microseconds`. However, note that the precision depends on the operating system and your local work load. Thus, you should not depend on a certain clock resolution.

7 Network Transparency

All actor operations as well as sending messages are network transparent. Remote actors are represented by actor proxies that forward all messages.

7.1 Publishing of Actors

```
void publish(actor_ptr whom, std::uint16_t port)
```

The function `publish` binds an actor to a given port. It throws `network_error` if socket related errors occur or `bind_failure` if the specified port is already in use.

```
publish(self, 4242);
receive_loop (
  on(atom("ping"), arg_match) >> [](int i) {
    reply(atom("pong"), i);
  }
);
```

7.2 Connecting to Remote Actors

```
actor_ptr remote_actor(const char* host, std::uint16_t port)
```

The function `remote_actor` tries to connect to the actor at given host and port or throws `network_error` if the connection failed.

```
auto pong = remote_actor("localhost", 4242);
send(pong, atom("ping"), 0);
bool done = false;
do_receive (
  on(atom("pong"), 10) >> [&]() {
    done = true;
  },
  on<atom("pong"), int>() >> [](int i) {
    reply(atom("ping"), i+1);
  }
).until(gref(done));
```

8 Group Communication

`libcppa` supports publish/subscribe-based group communication. Actors can join and leave groups and send messages to groups.

```
std::string group_module = ...;
std::string group_id = ...;
group_ptr grp = group::get(group_module, group_id);
self->join(grp);
send(grp, atom("test"));
self->leave(grp);
```

8.1 Anonymous Groups

Groups created on-the-fly with `group::anonymous()` can be used to coordinate a set of workers. Each call to `group::anonymous()` returns a new group instance.

8.2 Local Groups

The `"local"` group module creates groups for in-process communication. For example, a group for GUI related events could be identified by `group::get("local", "GUI events")`. The group ID `"GUI events"` uniquely identifies a singleton group instance of the module `"local"`.

9 Platform-Independent Type System

libcppa provides a fully network transparent communication between actors. Thus, libcppa needs to serialize and deserialize messages. Unfortunately, this is not possible using the RTTI system of C++. libcppa uses its own RTTI based on the class uniform_type_info, since it is not possible to extend std::type_info.

Unlike std::type_info::name(), uniform_type_info::name() is guaranteed to return the same name on all supported platforms. Furthermore, it allows to create an instance of a type by name.

```
// creates a signed, 32 bit integer
cppa::object i = cppa::uniform_typeid<int>()->create();
```

However, you should rarely if ever need to use object or uniform_type_info.

9.1 User-Defined Data Types in Messages

All user-defined types must be explicitly "announced" so that libcppa can (de)serialize them correctly, as shown in the example below.

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

struct foo { int a; int b; };

int main() {
  announce<foo>(&foo::a, &foo::b);
  send(self, foo{1,2});
  return 0;
}
```

Without the announce function call, the example program would terminate with an exception, because libcppa rejects all types without available runtime type information.

announce() takes the class as template parameter and pointers to all members (or getter/setter pairs) as arguments. This works for all primitive data types and STL compliant containers. See the announce examples 1 – 4 of the standard distribution for more details.

Obviously, there are limitations. You have to implement serialize/deserialize by yourself if your class does implement an unsupported data structure. See announce_example_5.cpp in the examples folder.

10 Appendix

10.1 Class `option`

Defined in header `"cppa/option.hpp"`.

```
template<typename T>
class option;
```

Represents an optional value.

Member types

Member type	Definition
`type`	`T`

Member functions

`option()`	Constructs an empty option
`option(T value)`	Initializes `this` with `value`
`option(const option&)` `option(option&&)`	Copy/move construction
`option& operator=(const option&)` `option& operator=(option&&)`	Copy/move assignment

Observers

`bool valid()` `explicit operator bool()`	Returns `true` if `this` represents a value
`bool empty()` `bool operator!()`	Returns `true` if `this` does **not** represent a value
`const T& get()` `const T& operator*()`	Access the stored value
`const T& get_or_else(const T& x)`	Returns `get()` if valid, `x` otherwise

Modifiers

`T& get()` `T& operator*()`	Access the stored value