CAF

A C++ framework for actor programming

User Manual

CAF version 0.13.2

Dominik Charousset

May 13, 2015

Contents

1 Introduction **1**

1.1 Actor Model 1

1.2 Terminology 1

1.2.1 Actor Address 1

1.2.2 Actor Handle 2

1.2.3 Untyped Actors 2

1.2.4 Typed Actor 2

1.2.5 Spawning 2

1.2.6 Monitoring 2

1.2.7 Links 2

2 First Steps **3**

2.1 Features Overview 3

2.2 Supported Compilers 3

2.3 Supported Operating Systems 3

2.4 Hello World Example 4

3 Pattern Matching **5**

3.1 Basics 5

3.2 Atoms 5

3.3 Advanced Match Cases 7

3.4 Wildcards 8

3.5 Projections 8

3.6 Dynamically Building Messages 9

4 Actors **10**

4.1 Implicit `self` Pointer 10

4.2 Interface 11

5 Sending Messages **12**

5.1 Replying to Messages 12

5.2 Delaying Messages . . . 12
5.3 Forwarding Messages in Untyped Actors . . . 13

6 Receiving Messages 14
6.1 Class-based actors . . . 14
6.2 Nesting Receives Using become/unbecome . . . 16
6.3 Timeouts . . . 17
6.4 Skipping Messages . . . 18

7 Synchronous Communication 19
7.1 Additional Error Messages . . . 19
7.2 Receive Response Messages . . . 19
7.3 Synchronous Failures and Error Handlers . . . 20

8 Management & Error Detection 21
8.1 Links . . . 21
8.2 Monitors . . . 21
8.3 Error Codes . . . 22
8.4 Attach Cleanup Code to an Actor . . . 22

9 Spawning Actors 23

10 Message Priorities 24

11 Network Transparency 25
11.1 Publishing of Actors . . . 25
11.2 Connecting to Remote Actors . . . 26

12 Network IO 27
12.1 Spawning Brokers . . . 27
12.2 Broker Interface . . . 28
12.3 Broker-related Message Types . . . 29

13 Group Communication 30
13.1 Anonymous Groups . . . 30

13.2 Local Groups . . . 30
13.3 Remote Groups . . . 30
13.4 Spawning Actors in Groups . . . 31

14 Managing Groups of Workers **32**
14.1 Predefined Dispatching Policies . . . 32
14.1.1 actor_pool::round_robin . . . 32
14.1.2 actor_pool::broadcast . . . 32
14.1.3 actor_pool::random . . . 32
14.2 Example . . . 32

15 Platform-Independent Type System **34**
15.1 User-Defined Data Types in Messages . . . 34

16 Blocking API **35**
16.1 Receiving Messages . . . 35
16.2 Receiving Synchronous Responses . . . 37
16.3 Mixing Actors and Threads with Scoped Actors . . . 37

17 Strongly Typed Actors **38**
17.1 Spawning Typed Actors . . . 38
17.2 Class-based Typed Actors . . . 39

18 Messages **41**
18.1 Class `message` . . . 41
18.2 Class `message_builder` . . . 42
18.3 Extracting . . . 43
18.4 Extracting Command Line Options . . . 44

19 Common Pitfalls **45**
19.1 Defining Patterns . . . 45
19.2 Event-Based API . . . 45
19.3 Synchronous Messages . . . 45
19.4 Sharing . . . 46

19.5 Constructors of Class-based Actors 46

20 Appendix **47**

20.1 Class `option` 47

20.2 Using `aout` – A Concurrency-safe Wrapper for `cout` 48

20.3 Migration Guides 49

20.3.1 0.8 ⇒ 0.9 49

20.3.2 0.9 ⇒ 0.10 (`libcppa` ⇒ CAF) 50

20.3.3 0.10 ⇒ 0.11 51

20.3.4 0.11 ⇒ 0.12 52

20.3.5 0.12 ⇒ 0.13 52

20.3.6 0.13 ⇒ 0.14 52

1 Introduction

Before diving into the API of CAF, we would like to take the opportunity to discuss the concepts behind CAF and to explain the terminology used in this manual.

1.1 Actor Model

The actor model describes concurrent entities—actors—that do not share state and communicate only via message passing. By decoupling concurrently running software components via message passing, the actor model avoids race conditions by design. Actors can create—"spawn"—new actors and monitor each other to build fault-tolerant, hierarchical systems. Since message passing is network transparent, the actor model applies to both concurrency and distribution.

When dealing with dozens of cores, mutexes, semaphores and other threading primitives are the wrong level of abstraction. Implementing applications on top of those primitives has proven challenging and error-prone. Additionally, mutex-based implementations can cause queueing and unmindful access to (even distinct) data from separate threads in parallel can lead to false sharing: both decreasing performance significantly, up to the point that an application actually runs slower when adding more cores.

The actor model has gained momentum over the last decade due to its high level of abstraction and its ability to make efficient use of multicore and multiprocessor machines. However, the actor model has not yet been widely adopted in the native programming domain. With CAF, we contribute a library for actor programming in C++ as open source software to ease native development of concurrent as well as distributed systems. In this regard, CAF follows the C++ philosophy "building the highest abstraction possible without sacrificing performance".

1.2 Terminology

You will find that CAF has not simply adopted exiting implementations based on the actor model such as Erlang or the Akka library. Instead, CAF aims to provide a modern C++ API allowing for type-safe as well as dynamically typed messaging. Hence, most aspects of our system are familiar to developers having experience with other actor systems, but there are also slight differences in terminology. However, neither CAF nor this manual require any foreknowledge.

1.2.1 Actor Address

In CAF, each actor has a (network-wide) unique logical address that can be used to identify and monitor it. However, the address can *not* be used to send a message to an actor. This limitation is due to the fact that the address does not contain any type information about the actor. Hence, it would not be safe to send it any message, because the actor might use a strictly typed messaging interface not accepting the given message.

1.2.2 Actor Handle

An actor handle contains the address of an actor along with its type information. In order to send an actor a message, one needs to have a handle to it – the address alone is not sufficient. The distinction between handles and addresses – which is unique to CAF when comparing it to other actor systems – is a consequence of the design decision to support both untyped and typed actors.

1.2.3 Untyped Actors

An untyped actor does not constrain the type of messages it receives, i.e., a handle to an untyped actor accepts any kind of message. That does of course not mean that untyped actors must handle all possible types of messages. Choosing typed vs untyped actors is mostly a matter of taste. Untyped actors allow developers to build prototypes faster, while typed actors allow the compiler to fetch more errors at compile time.

1.2.4 Typed Actor

A typed actor defines its messaging interface, i.e., both input and output types, in its type. This allows the compiler to check message types statically.

1.2.5 Spawning

“Spawning” an actor means to create and run a new actor.

1.2.6 Monitoring

A monitored actor sends a “down message” to all actors monitoring it as part of its termination. This allows actors to supervise other actors and to take measures when one of the supervised actors failed, i.e., terminated with a non-normal exit reason.

1.2.7 Links

A link is bidirectional connection between two actors. Each actor sends an “exit message” to all of its links as part of its termination. Unlike down messages (cf. 1.2.6), the default behavior for received exit messages causes the receiving actor to terminate for the same reason if the link has failed, i.e., terminated with a non-normal exit reason. This allows developers to create a set of actors with the guarantee that either all or no actors are alive. The default behavior can be overridden, i.e., exit message can be “trapped”. When trapping exit messages, they are received as any other ordinary message and can be handled by the actor.

2 First Steps

To compile CAF, you will need CMake and a C++11 compiler. To get and compile the sources, open a terminal (on Linux or Mac OS X) and type:

```
git clone https://github.com/actor-framework/actor-framework
cd actor-framework
./configure
make
make install [as root, optional]
```

It is recommended to run the unit tests as well:

```
make test
```

Please submit a bug report that includes (a) your compiler version, (b) your OS, and (c) the content of the file `build/Testing/Temporary/LastTest.log` if an error occurs.

2.1 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 for soft migration of existing applications
- Publish/subscribe group communication

2.2 Supported Compilers

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

2.3 Supported Operating Systems

- Linux
- Mac OS X
- *Note for MS Windows*: CAF relies on C++11 features such as unrestricted unions. We will support this platform as soon as Microsoft's compiler implements all required C++11 features. In the meantime, CAF can be used with MinGW.

2.4 Hello World Example

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

#include "caf/all.hpp"

using namespace std;
using namespace caf;

behavior mirror(event_based_actor* self) {
    // return the (initial) actor behavior
    return {
        // a handler for messages containing a single string
        // that replies with a string
        [=](const string& what) -> string {
            // prints "Hello World!" via aout
            // (thread-safe cout wrapper)
            aout(self) << what << endl;
            // terminates this actor
            // ('become' otherwise loops forever)
            self->quit();
            // reply "!dlroW olleH"
            return string(what.rbegin(), what.rend());
        }
    };
}

void hello_world(event_based_actor* self, const actor& buddy) {
    // send "Hello World!" to our buddy ...
    self->sync_send(buddy, "Hello World!").then(
        // ... wait for a response ...
        [=](const string& what) {
            // ... and print it
            aout(self) << what << endl;
        }
    );
}

int main() {
    // create a new actor that calls 'mirror()'
    auto mirror_actor = spawn(mirror);
    // create another actor that calls 'hello_world(mirror_actor)';
    spawn(hello_world, mirror_actor);
    // wait until all other actors we have spawned are done
    await_all_actors_done();
    // run cleanup code before exiting main
    shutdown();
}
```

3 Pattern Matching

Actor programming implies a message passing paradigm. This means that defining message handlers is a recurring task. The easiest and most natural way to specify such message handlers is pattern matching. Unfortunately, C++ does not provide any pattern matching facilities. Hence, we provide an internal domain-specific language to match incoming messages.

3.1 Basics

Actors can store a set of message callbacks using either `behavior` or `message_handler`. The difference between the two is that the former stores an optional timeout. The most basic way to define a pattern is to store a set of lambda expressions using one of the two container types.

```
behavior bhvr1{
  [](int i) { /*...*/ },
  [](int i, float f) { /*...*/ },
  [](int a, int b, int c) { /*...*/ }
};
```

In our first example, `bhvr1` models a pattern accepting messages that consist of either exactly one `int`, or one `int` followed by a `float`, or three `int`s. Any other message is not matched and will remain in the mailbox until it is consumed eventually. This caching mechanism allows actors to ignore messages until a state change replaces its message handler. However, this can lead to a memory leak if an actor receives messages it handles in no state. To allow actors to specify a default message handlers for otherwise unmatched messages, CAF provides `others`.

```
behavior bhvr2{
  [](int i) { /*...*/ },
  [](int i, float f) { /*...*/ },
  [](int a, int b, int c) { /*...*/ },
  others >> [] { /*...*/ }
};
```

Please note the change in syntax for the default case. The lambda expression passed to the constructor of `behavior` is prefixed by a ”match expression” and the operator `>>`.

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 CAF. This mapping is guaranteed to be collision-free and invertible, 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.

```
atom_value a1 = atom("add");
atom_value a2 = atom("multiply");
// ...
```

Warning: The compiler cannot enforce the restrictions at compile time, except for a length check. The assertion `atom("!?") != atom("?!")` is not true, because each invalid character is mapped to the whitespace character.

An `atom_value` alone does not help us statically annotate function handlers. To accomplish this, CAF offers compile-time *atom constants*.

```
using add_atom = atom_constant<atom("add")>;
using multiply_atom = atom_constant<atom("multiply")>;
```

Using the constants, we can now define message passing interfaces in a convenient way.

```
behavior do_math{
  [](add_atom, int a, int b) {
    return a + b;
  },
  [](multiply_atom, int a, int b) {
    return a * b;
  }
};
```

Atom constants define a static member `value` that can be used on the caller side (see Section 5), e.g., `send(math_actor, add_atom::value, 1, 2)`. Please note that the static `value` member does *not* have the type `atom_value`, unlike `std::integral_constant` for example.

3.3 Advanced Match Cases

Match cases are an advanced feature of CAF and allow you to match on values and to transform data while matching. A match case begins with a call to the function `on`, which returns an intermediate object providing `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`,

When using the basic syntax, CAF generates the match case automatically. A verbose version of the `bhvr1` from 3.1 is shown below.

```
behavior verbose_bhvr1{
  on<int>() >> [](int i) { /*...*/ },
  on<int, float>() >> [](int i, float f) { /*...*/ },
  on<int, int, int>() >> [](int a, int b, int c) { /*...*/ }
};
```

It is worth mentioning that passing the lambdas directly is more efficient, since it allows CAF to select a special-purpose implementation. 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. To match for any value of a given type, the template `val<T>` can be used, as shown in the following example.

```
behavior bhvr3{
  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 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
```

To avoid redundancy when working with match expressions, `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 any given callback.

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

Note that `arg_match` must be passed as last parameter. If all types should be deduced from the callback signature, `on_arg_match` can be used, which is a faster alternative for `on(arg_match)`. However, `on_arg_match` is used implicitly whenever a callback is used without preceding match expression.

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 Projections

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 must not have side-effects and must not throw exceptions. A failed projection is not an error, it simply indicates that a pattern is not matched. 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 {};
};
message_handler fun {
  on(intproj) >> [](int i) {
    // case 1: successfully converted a string
  },
  [](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.

3.6 Dynamically Building Messages

Usually, messages are created implicitly when sending messages but can also be created explicitly using `make_message`. In both cases, types and number of elements are known at compile time. To allow for fully dynamic message generation, CAF also offers a third option to create messages by using a `message_builder`:

```
message_builder mb;
// prefix message with some atom
mb.append(strings_atom::value);
// fill message with some strings
std::vector<std::string> strings{/*...*/};
for (auto& str : strings) {
  mb.append(str);
}
// create the message
message msg = mb.to_message();
```

4 Actors

CAF provides several actor implementations, each covering a particular use case. The class `local_actor` is the base class for all implementations, except for (remote) proxy actors. Hence, `local_actor` provides a common interface for actor operations like trapping exit messages or finishing execution. The default actor implementation in CAF is event-based. Event-based actors have a very small memory footprint and are thus very lightweight and scalable. Context-switching actors are used for actors that make use of the blocking API (see Section 16), but do not need to run in a separate thread. Context-switching and event-based actors are scheduled cooperatively in a thread pool. Thread-mapped actors can be used to opt-out of this cooperative scheduling.

4.1 Implicit `self` Pointer

When using a function or functor to implement an actor, the first argument *can* be used to capture a pointer to the actor itself. The type of this pointer is `event_based_actor*` per default and `blocking_actor*` when using the `blocking_api` flag. When dealing with typed actors, the types are `typed_event_based_actor<...>*` and `typed_blocking_actor<...>*`.

4.2 Interface

```
class local_actor;
```

Member functions

Observers

`actor_addr address()`	Returns the address of this actor
`bool trap_exit()`	Checks whether this actor traps exit messages
`message& current_message()`	Returns the currently processed message **Warning**: Only set during callback invocation; calling this function after forwarding the message or while not in a callback is undefined behavior
`actor_addr& current_sender()`	Returns the sender of the current message **Warning**: Only set during callback invocation; calling this function after forwarding the message or while not in a callback is undefined behavior
`vector<group> joined_groups()`	Returns all subscribed groups

Modifiers

`quit(uint32_t reason = normal)`	Finishes execution of this actor
`void trap_exit(bool enabled)`	Enables or disables trapping of exit messages
`void join(const group& g)`	Subscribes to group `g`
`void leave(const group& g)`	Unsubscribes group `g`
`void on_sync_failure(auto fun)`	Sets a handler, i.e., a functor taking no arguments, for unexpected synchronous response messages (default action is to kill the actor for reason `unhandled_sync_failure`)
`void monitor(actor whom)`	Unidirectionally monitors `whom` (see Section 8.2)
`void demonitor(actor whom)`	Removes a monitor from `whom`
`bool has_sync_failure_handler()`	Checks wheter this actor has a user-defined sync failure handler
`template <class F>` `void set_exception_handler(F f)`	Sets a custom handler for uncaught exceptions

5 Sending Messages

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

Messages can be sent by using the member function `send`. The variadic template parameter pack `what...` is converted to a message and then enqueued to the mailbox of `whom`.

```
void some_fun(event_based_actor* self) {
  actor other = spawn(...);
  self->send(other, 1, 2, 3);
  // sending a message directly is also ok:
  auto msg = make_message(1, 2, 3);
  self->send(other, msg);
}
```

5.1 Replying to Messages

The return value of a message handler is used as response message. Actors can also use the result of a `sync_send` to answer to a request, as shown below.

```
behavior client(event_based_actor* self, const actor& master) {
  return {
    [=](const string& request) {
      return self->sync_send(master, request).then(
        [=](const std::string& response) {
          return response;
        }
      );
    }
  };
};
```

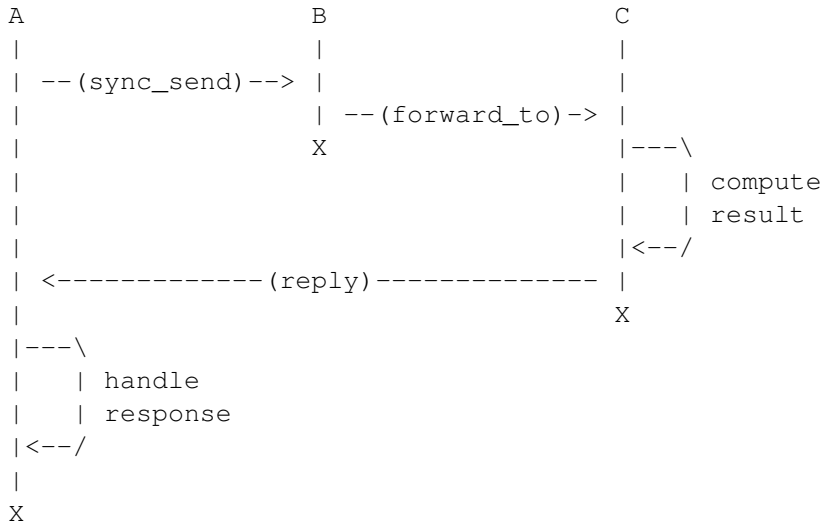
5.2 Delaying Messages

Messages can be delayed by using the function `delayed_send`.

```
using poll_atom = atom_constant<atom("poll")>;
behavior poller(event_based_actor* self) {
  using std::chrono::seconds;
  self->delayed_send(self, seconds(1), poll_atom::value);
  return {
    [](poll_atom) {
      // poll a resource
      // ...
      // schedule next polling
      self->delayed_send(self, seconds(1), poll_atom::value);
    }
  };
}
```

5.3 Forwarding Messages in Untyped Actors

The member function `forward_to` forwards the last dequeued message to an other actor. Forwarding a synchronous message will also transfer responsibility for the request, i.e., the receiver of the forwarded message can reply as usual and the original sender of the message will receive the response. The following diagram illustrates forwarding of a synchronous message from actor `B` to actor `C`.



The forwarding is completely transparent to actor `C`, since it will see actor `A` as sender of the message. However, actor `A` will see actor `C` as sender of the response message instead of actor `B` and thus could recognize the forwarding by evaluating `self->last_sender()`.

6 Receiving Messages

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.

An event-based actor, i.e., the default implementation in CAF, 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.

6.1 Class-based actors

A class-based actor is a subtype of `event_based_actor` and must implement the pure virtual member function `make_behavior` returning the initial behavior.

```
class printer : public event_based_actor {
  behavior make_behavior() override {
    return {
      others >> [] {
        cout << to_string(last_dequeued()) << endl;
      }
    };
  }
};
```

Another way to implement class-based actors is provided by the class `sb_actor` ("State-Based Actor"). This base class simply returns `init_state` (defined in the subclass) from its implementation for `make_behavior`.

```
struct printer : sb_actor<printer> {
  behavior init_state {
    others >> [] {
      cout << to_string(last_dequeued()) << endl;
    }
  };
};
```

Note that `sb_actor` uses the Curiously Recurring Template Pattern. Thus, the derived class must be given as template parameter. This technique allows `sb_actor` to access the `init_state` member of a derived class. The following example illustrates a more advanced state-based actor that implements a stack with a fixed maximum number of elements.

```
using pop_atom = atom_constant<atom("pop")>;
using push_atom = atom_constant<atom("push")>;

class fixed_stack : public sb_actor<fixed_stack> {
 public:
  fixed_stack(size_t max) : max_size(max)  {
    full.assign(
      [=](push_atom, int) {
        /* discard */
      },
      [=](pop_atom) -> message {
        auto result = data.back();
        data.pop_back();
        become(filled);
        return make_message(ok_atom::value, result);
      }
    );
    filled.assign(
      [=](push_atom, int what) {
        data.push_back(what);
        if (data.size() == max_size) {
          become(full);
        }
      },
      [=](pop_atom) -> message {
        auto result = data.back();
        data.pop_back();
        if (data.empty()) {
          become(empty);
        }
        return make_message(ok_atom::value, result);
      }
    );
    empty.assign(
      [=](push_atom, int what) {
        data.push_back(what);
        become(filled);
      },
      [=](pop_atom) {
        return error_atom::value;
      }
    );
  }

  size_t max_size;
  std::vector<int> data;
  behavior full;
  behavior filled;
  behavior empty;
  behavior& init_state = empty;
};
```

6.2 Nesting Receives Using `become/unbecome`

Since `become` does not block, an actor has to manipulate its behavior stack to achieve nested receive operations. An actor can set a new behavior by calling `become` with the `keep_behavior` policy to be able to return to its previous behavior later on by calling `unbecome`, as shown in the example below.

```
// receives {int, float} sequences
behavior testee(event_based_actor* self) {
  return {
    [=](int value1) {
      self->become (
        // the keep_behavior policy stores the current behavior
        // on the behavior stack to be able to return to this
        // behavior later on by calling unbecome()
        keep_behavior,
        [=](float value2) {
          cout << value1 << " => " << value2 << endl;
          // restore previous behavior
          self->unbecome();
        }
      );
    }
  };
}
```

An event-based actor finishes execution with normal exit reason if the behavior stack is empty after calling `unbecome`. The default policy of `become` is `discard_behavior` that causes an actor to override its current behavior. The policy flag must be the first argument of `become`.

Note: the message handling in CAF is consistent among all actor implementations: unmatched messages are *never* implicitly discarded if no suitable handler was found. Hence, the order of arrival is not important in the example above. This is unlike other event-based implementations of the actor model such as Akka for instance.

6.3 Timeouts

A behavior set by `become` is invoked whenever a new messages arrives. If no message ever arrives, the actor would wait 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.

```
behavior eager_actor(event_based_actor* self) {
  return {
    [](int i) { /* ... */ },
    [](float i) { /* ... */ },
    others >> [] { /* ... */ },
    after(std::chrono::seconds(10)) >> [] {
      aout(self) << "received nothing within 10 seconds..." << 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 the actor to scan its mailbox once and then invoke the timeout immediately if no matching message was found.

CAF supports timeouts using `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.

6.4 Skipping Messages

Unmatched messages are skipped automatically by CAF's runtime system. This is true for *all* actor implementations. To allow actors to skip messages manually, `skip_message` can be used. This is in particular useful whenever an actor switches between behaviors, but wants to use a default rule created by `others` to catch messages that are not handled by any of its behaviors.

The following example illustrates a simple server actor that dispatches requests to workers. After receiving an `'idle'` message, it awaits a request that is then forwarded to the idle worker. Afterwards, the server returns to its initial behavior, i.e., awaits the next `'idle'` message. The server actor will exit for reason `user_defined` whenever it receives a message that is neither a request, nor an idle message.

```
using idle_atom = atom_constant<atom("idle")>;
using request_atom = atom_constant<atom("request")>;

behavior server(event_based_actor* self) {
  auto die = [=] { self->quit(exit_reason::user_defined); };
  return {
    [=](idle_atom) {
      auto worker = last_sender();
      self->become (
        keep_behavior,
        [=](request_atom) {
          // forward request to idle worker
          self->forward_to(worker);
          // await next idle message
          self->unbecome();
        },
        [=](idle_atom) {
          return skip_message();
        },
        others >> die
      );
    },
    [=](request_atom) {
      return skip_message();
    },
    others >> die
  };
}
```

7 Synchronous Communication

CAF supports both asynchronous and synchronous communication. The latter is provided by the member function `sync_send`.

```
template<typename... Args>
__unspecified__ sync_send(actor whom, Args&&... what);
```

A synchronous message is sent to the receiving actor's mailbox like any other (asynchronous) message. Only the response message is treated separately.

7.1 Additional Error Messages

```
struct sync_exited_msg {
  actor_addr source;
  uint32_t reason;
};
```

When using synchronous messaging, CAF's runtime will send a `sync_exited_msg` message if the receiver is not alive. This is in addition to exit and down messages caused by linking or monitoring.

7.2 Receive Response Messages

When sending a synchronous message, the response handler can be passed by either using `then` (event-based actors) or `await` (blocking actors).

```
void foo(event_based_actor* self, actor testee) {
  // testee replies with a string to 'get'
  self->sync_send(testee, get_atom::value).then(
    [=](const std::string& str) {
      // handle str
    },
    after(std::chrono::seconds(30)) >> [=]() {
      // handle error
    }
  );
);
```

Similar to `become`, the `then` function modifies an actor's behavior stack. However, it is used as "one-shot handler" and automatically returns to the previous behavior afterwards.

7.3 Synchronous Failures and Error Handlers

An unexpected response message, i.e., a message that is not handled by the “one-shot-handler”, is considered as an error. The runtime will invoke the actor’s `on_sync_failure`, which kills the actor by calling `self->quit(exit_reason::unhandled_sync_failure)` per default. The error handler can be overridden by calling `self->on_sync_failure(...)` as shown below.

```
void foo(event_based_actor* self, actor testee) {
  // set handler for unexpected messages
  self->on_sync_failure([=] {
    aout(self) << "received unexpected synchronous response: "
               << to_string(self->last_dequeued()) << endl;
  });
  // set response handler by using "then"
  sync_send(testee, get_atom::value).then(
    [=](const std::string& str) {
      /* handle str */
    }
    // any other result will call the on_sync_failure handler
  );
}
```

8 Management & Error Detection

CAF 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.

8.1 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.

```
actor worker = ...;
// receive exit messages as regular messages
self->trap_exit(true);
// monitor spawned actor
self->link_to(worker);
// wait until worker exited
self->become (
  [=](const exit_msg& e) {
    if (e.reason == exit_reason::normal) {
      // worker finished computation
    else {
      // worker died unexpectedly
    }
  }
);
```

8.2 Monitors

A monitor observes the lifetime of an actor. Monitored actors send a down message to all observers as part of their termination. Unlike exit messages, down messages are always treated like any other ordinary message. An actor will receive one down message for each time it called `self->monitor(...)`, even if it adds a monitor to the same actor multiple times.

```
actor worker = ...;
// monitor spawned actor
self->monitor(worker);
// wait until worker exited
self->become (
  [](const down_msg& d) {
    if (d.reason == exit_reason::normal) {
      // worker finished computation
    } else {
      // worker died unexpectedly
    }
  }
);
```

8.3 Error Codes

All error codes are defined in the namespace `caf::exit_reason`. To obtain a string representation of an error code, use `caf::exit_reason::as_string(uint32_t)`.

`normal`	1	Actor finished execution without error
`unhandled_exception`	2	Actor was killed due to an unhandled exception
`unhandled_sync_failure`	4	Actor was killed due to an unexpected synchronous response message
`unhandled_sync_timeout`	5	Actor was killed, because no timeout handler was set and a synchronous message timed out
`unknown`	6	Indicates that an actor has been exited and its state is no longer known
`user_shutdown`	16	Actor was killed by a user-generated event
`remote_link_unreachable`	257	Indicates that a remote actor became unreachable, e.g., due to connection error
`user_defined`	65536	Minimum value for user-defined exit codes

8.4 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.

```
using done_atom = atom_constant<atom("done")>;

behavior supervisor(event_based_actor* self, actor worker) {
  actor observer = self;
  // "monitor" spawned actor
  worker->attach_functor([observer](std::uint32_t reason) {
    // this callback is invoked from worker
    anon_send(observer, done_atom::value);
  });
  // wait until worker exited
  return {
    [](done_atom) {
      // ... worker terminated ...
    }
  };
}
```

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

9 Spawning Actors

Actors are created using the function `spawn`. The easiest way to implement actors is to use functors, e.g., a free function or a lambda expression. The arguments to the functor are passed to `spawn` as additional arguments. The function `spawn` also takes optional flags as template parameter. The flag `detached` causes `spawn` to assign a dedicated thread to the actor, i.e., to opt-out of the cooperative scheduling. Convenience flags like `linked` or `monitored` automatically link or monitor to the newly created actor. Naturally, these two flags are not available on "top-level" spawns. Actors that make use of the blocking API—see Section 16—must be spawned using the flag `blocking_api`. Flags are concatenated using the operator `+`, as shown in the examples below.

```
#include "caf/all.hpp"
using namespace caf;

void my_actor1();
void my_actor2(event_based_actor*, int arg1, const std::string& arg2);
void ugly_duckling(blocking_actor*);

class my_actor3 : public event_based_actor { /* ... */ };
class my_actor4 : public event_based_actor {
  public: my_actor4(int some_value) { /* ... */ }
  /* ... */
};

// whenever we want to link to or monitor a spawned actor,
// we have to spawn it using the self pointer, otherwise
// we can use the free function 'spawn' (top-level spawn)
void server(event_based_actor* self) {
  // spawn functor-based actors
  auto a0 = self->spawn(my_actor1);
  auto a1 = self->spawn<linked>(my_actor2, 42, "hello actor");
  auto a2 = self->spawn<monitored>([] { /* ... */ });
  auto a3 = self->spawn([](int) { /* ... */ }, 42);
  // spawn thread-mapped actors
  auto a4 = self->spawn<detached>(my_actor1);
  auto a5 = self->spawn<detached + linked>([] { /* ... */ });
  auto a6 = self->spawn<detached>(my_actor2, 0, "zero");
  // spawn class-based actors
  auto a7 = self->spawn<my_actor3>();
  auto a8 = self->spawn<my_actor4, monitored>(42);
  // spawn and detach class-based actors
  auto a9 = self->spawn<my_actor4, detached>(42);
  // spawn actors that need access to the blocking API
  auto aa = self->spawn<blocking_api>(ugly_duckling);
  // compiler error: my_actor2 captures the implicit
  // self pointer as event_based_actor* and thus cannot
  // be spawned using the `blocking_api` flag
  // --- auto ab = self->spawn<blocking_api>(my_actor2);
}
```

10 Message Priorities

By default, all messages have the same priority and actors ignore priority flags. Actors that should evaluate priorities must be spawned using the `priority_aware` flag. This flag causes the actor to use a priority-aware mailbox implementation. It is not possible to change this implementation dynamically at runtime.

```
using a_atom = atom_constant<atom("a")>;
using b_atom = atom_constant<atom("b")>;

behavior testee(event_based_actor* self) {
  // send 'b' with normal priority
  self->send(self, b_atom::value);
  // send 'a' with high priority
  self->send(message_priority::high, self, a_atom::value);
  // terminate after receiving a 'b'
  return {
    [=](b_atom) {
      aout(self) << "received 'b' => quit" << endl;
      self->quit();
    },
    [=](a_atom) {
      aout(self) << "received 'a'" << endl;
    },
  };
}

int main() {
  // will print "received 'b' => quit"
  spawn(testee);
  await_all_actors_done();
  // will print "received 'a'" and then "received 'b' => quit"
  spawn<priority_aware>(testee);
  await_all_actors_done();
  shutdown();
}
```

11 Network Transparency

All actor operations as well as sending messages are network transparent. Remote actors are represented by actor proxies that forward all messages. All functions shown in this section can be accessed by including the header `"caf/io/all.hpp"` and live in the namespace `caf::io`.

11.1 Publishing of Actors

```
uint16_t publish(actor whom, uint16_t port,
                 const char* addr = nullptr,
                 bool reuse_addr = false)
```

The function `publish` binds an actor to a given port. To choose the next high-level port available for binding, one can specify `port == 0` and retrieves the bound port as return value. The return value is equal to `port` if `port != 0`. The function throws `network_error` if socket related errors occur or `bind_failure` if the specified port is already in use. The optional `addr` parameter can be used to listen only to the given address. Otherwise, the actor accepts all incoming connections (`INADDR_ANY`). The flag `reuse_addr` controls the behavior when binding an IP address to a port, with the same semantics as the BSD socket flag `SO_REUSEADDR`. For example, if `reuse_addr = false`, binding two sockets to 0.0.0.0:42 and 10.0.0.1:42 will fail with `EADDRINUSE` since 0.0.0.0 includes 10.0.0.1. With `reuse_addr = true` binding would succeed because 10.0.0.1 and 0.0.0.0 are not literally equal addresses.

```
publish(self, 4242);
self->become (
  [](ping_atom, int i) {
    return std::make_tuple(pong_atom::value, i);
  }
);
```

To close a socket, e.g., to allow other actors to be published at the port, the function `unpublish` can be used. This function is called implicitly if a published actor terminates.

```
void unpublish(caf::actor whom, uint16_t port)
```

11.2 Connecting to Remote Actors

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

The function `remote_actor` connects to the actor at given host and port. A `network_error` is thrown if the connection failed.

```
auto pong = remote_actor("localhost", 4242);
self->send(pong, ping_atom::value, 0);
self->become (
  [=](pong_value, int i) {
    if (i >= 10) {
      self->quit();
      return;
    }
    self->send(pong, ping_atom::value, i + 1);
  }
);
```

12 Network IO

When communicating to other services in the network, sometimes low-level socket IO is inevitable. For this reason, CAF provides *brokers*. A broker is an event-based actor running in the middleman that multiplexes socket IO. It can maintain any number of acceptors and connections. Since the broker runs in the middleman, implementations should be careful to consume as little time as possible in message handlers. Any considerable amount work should outsourced by spawning new actors (or maintaining worker actors). All functions shown in this section can be accessed by including the header `"caf/io/all.hpp"` and live in the namespace `caf::io`.

12.1 Spawning Brokers

Brokers are spawned using the function `spawn_io` and always use functor-based implementations capturing the self pointer of type `broker*`. For convenience, `spawn_io_server` can be used to spawn a new broker listening to a local port and `spawn_io_client` can be used to spawn a new broker that connects to given host and port or uses existing IO streams.

```
template<spawn_options Os = no_spawn_options,
         typename F = std::function<behavior (broker*)>,
         typename... Ts>
actor spawn_io(F fun, Ts&&... args);

template<spawn_options Os = no_spawn_options,
         typename F = std::function<behavior (broker*)>,
         typename... Ts>
actor spawn_io_client(F fun,
                      input_stream_ptr in,
                      output_stream_ptr out,
                      Ts&&... args);

template<spawn_options Os = no_spawn_options,
         typename F = std::function<behavior (broker*)>,
         typename... Ts>
actor spawn_io_client(F fun, string host, uint16_t port, Ts&&... args);

template<spawn_options Os = no_spawn_options,
         typename F = std::function<behavior (broker*)>,
         typename... Ts>
actor spawn_io_server(F fun, uint16_t port, Ts&&... args);
```

12.2 Broker Interface

```
class broker;
```

Member Functions

`void configure_read(connection_handle hdl, receive_policy::config config)`	Modifies the receive policy for the connection identified by `hdl`. This will cause the middleman to enqueue the next `new_data_msg` according to the given `config` created by `receive_policy::exactly(x)`, `receive_policy::at_most(x)`, or `receive_policy::at_least(x)` (with `x` denoting the number of bytes)
`void write(connection_handle hdl, size_t num_bytes, const void* buf)`	Writes data to the output buffer
`void flush(connection_handle hdl)`	Sends the data from the output buffer
`template <class F, class... Ts> actor fork(F fun, connection_handle hdl, Ts&&... args)`	Spawns a new broker that takes ownership of given connection
`size_t num_connections()`	Returns the number of open connections
`void close(connection_handle hdl)`	Closes a connection
`void close(accept_handle hdl)`	Closes an acceptor

12.3 Broker-related Message Types

Brokers receive system messages directly from the middleman whenever an event on one of it handles occurs.

```
struct new_connection_msg {
  accept_handle source;
  connection_handle handle;
};
```

Whenever a new incoming connection (identified by the `handle` field) has been accepted for one of the broker's accept handles, it will receive a `new_connection_msg`.

```
struct new_data_msg {
  connection_handle handle;
  std::vector<char> buf;
};
```

New incoming data is transmitted to the broker using messages of type `new_data_msg`. The raw bytes can be accessed as buffer object of type `std::vector<char>`. The amount of data, i.e., how often this message is received, can be controlled using `configure_read` (see 12.2). It is worth mentioning that the buffer is re-used whenever possible. This means, as long as the broker does not create any new references to the message by copying it, the middleman will always use only a single buffer per connection.

```
struct connection_closed_msg {
  connection_handle handle;
};

struct acceptor_closed_msg {
  accept_handle handle;
};
```

A `connection_closed_msg` or `acceptor_closed_msg` informs the broker that one of it handles is no longer valid.

13 Group Communication

CAF 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 = ...;
auto grp = group::get(group_module, group_id);
self->join(grp);
self->send(grp, "test");
self->leave(grp);
```

13.1 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, unique group instance.

13.2 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"`.

13.3 Remote Groups

To deploy groups in a network, one host can act as group server by publishing its local groups at any given port:

```
void publish_local_groups(std::uint16_t port, const char* addr)
```

By calling `group::get("remote", "<group>@<host>:<port>")`, other hosts are now able to connect to a remotely running group. Please note that the group communication is no longer available once the server disconnects. This implementation uses N-times unicast underneath. It is worth mentioning that user-implemented groups can be build on top of IP multicast or overlay technologies such as Scribe to achieve better performance or reliability.

13.4 Spawning Actors in Groups

The function `spawn_in_group` can be used to create actors as members of a group. The function causes the newly created actors to call `join(...)` immediately and before `spawn_in_group` returns. The usage of `spawn_in_group` is equal to `spawn`, except for an additional group argument. The group handle is always the first argument, as shown in the examples below.

```
void fun1();
void fun2(int, float);
class my_actor1 : event_based_actor { /* ... */ };
class my_actor2 : event_based_actor {
  // ...
  my_actor2(const std::string& str) { /* ... */ }
};
// ...
auto grp = group::get(...);
auto a1 = spawn_in_group(grp, fun1);
auto a2 = spawn_in_group(grp, fun2, 1, 2.0f);
auto a3 = spawn_in_group<my_actor1>(grp);
auto a4 = spawn_in_group<my_actor2>(grp, "hello my_actor2!");
```

14 Managing Groups of Workers

When managing a set of workers, a central actor often dispatches requests to a set of workers. For this purpose, the class `actor_pool` implements a lightweight abstraction for managing a set of workers using a dispatching policy.

Pools are created using the static member function `make`, which takes either one argument (the policy) or three (number of workers, factory function for workers, and dispatching policy). After construction, one can add new workers via messages of the form $('SYS', 'PUT', worker)$, remove workers with $('SYS', 'DELETE', worker)$, and retrieve the set of workers as `vector<actor>` via $('SYS', 'GET')$. For example, adding `worker` to `my_pool` could be done using `send(my_pool, sys_ato`

An actor pool takes ownership of its workers. When forced to quit, it sends an exit messages to all of its workers, forcing them to quit as well. The pool also monitors all of its workers.

Pools does not cache messages by default, but enqueue them directly in a workers mailbox. Consequently, a terminating worker loses all unprocessed messages. For more advanced caching strategies, such as reliable message delivery, users can implement their own dispatching policies.

14.1 Predefined Dispatching Policies

The actor pool class comes with a set predefined policies for convenience.

14.1.1 `actor_pool::round_robin`

This policy forwards incoming requests in a round-robin manner to workers. There is no guarantee that messages are consumed, i.e., work items are lost if the worker exits before processing all of its messages.

14.1.2 `actor_pool::broadcast`

This policy forwards *each* message to *all* workers. Synchronous messages to the pool will be received by all workers, but the client will only recognize the first arriving response message—or error—and discard subsequent messages. Note that this is not caused by the policy itself, but a consequence of forwarding synchronous messages to more than one actor.

14.1.3 `actor_pool::random`

This policy forwards incoming requests to one worker from the pool chosen uniformly at random. Analogous to `round_robin`, this policy does not cache or redispatch messages.

14.2 Example

```
actor new_worker() {
  return spawn([]() -> behavior {
    return {
      [](int x, int y) {
        return x + y;
      }
    };
  });
}

void broadcast_example() {
  scoped_actor self;
  // spawns a pool with 5 workers
  auto pool5 = [] {
    return actor_pool::make(5, new_worker, actor_pool::broadcast{});
  };
  // spawns a pool with 5 pools with 5 workers each
  auto w = actor_pool::make(5, pool5, actor_pool::broadcast{});
  // will be broadcasted to 25 workers
  self->send(w, 1, 2);
  std::vector<int> results;
  int i = 0;
  self->receive_for(i, 25)(
    [&](int res) {
      results.push_back(res);
    }
  );
  assert(results.size(), 25);
  assert(std::all_of(results.begin(), results.end(),
                     [](int res) { return res == 3; }));
  // terminate pool(s) and all workers
  self->send_exit(w, exit_reason::user_shutdown);
}
```

15 Platform-Independent Type System

CAF provides a fully network transparent communication between actors. Thus, CAF needs to serialize and deserialize messages. Unfortunately, this is not possible using the RTTI system of C++. CAF 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
uniform_value i = uniform_typeid<int>()->create();
```

You should rarely, if ever, need to use `uniform_value` or `uniform_type_info`. The type `uniform_value` stores a type-erased pointer along with the associated `uniform_type_info`. The sole purpose of this simple abstraction is to enable the pattern matching engine of CAF to query the type information and then dispatch the value to a message handler. When using a `message_builder`, each element is stored as a `uniform_value`.

15.1 User-Defined Data Types in Messages

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

```
#include "caf/all.hpp"

struct foo { int a; int b; };

int main() {
  caf::announce<foo>("foo", &foo::a, &foo::b);
  // ... foo can now safely be used in messages ...
}
```

Without announcing `foo`, CAF is not able to (de)serialize instances of it. The function `announce()` takes the class as template parameter. The first argument to the function always is the type name followed by pointers to all members (or getter/setter pairs). 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.

16 Blocking API

Besides event-based actors (the default implementation), CAF also provides context-switching and thread-mapped actors that can make use of the blocking API. Those actor implementations are intended to ease migration of existing applications or to implement actors that need to have access to blocking receive primitives for other reasons.

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 (by using `become`), whereas the latter use an explicit, blocking receive function.

16.1 Receiving Messages

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

```
self->receive (
  on<int>() >> // ...
);
```

The code snippet above illustrates the use of `receive`. Note that the message handler passed to `receive` is a temporary object at runtime. Hence, using receive inside a loop would cause creation of a new handler on each iteration. CAF 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([&] { return received >= 10; });
```

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

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

16.2 Receiving Synchronous Responses

Analogous to `sync_send(...).then(...)` for event-based actors, blocking actors can use `sync_send(...).await(...)`.

```
void foo(blocking_actor* self, actor testee) {
  // testee replies with a string to 'get'
  self->sync_send(testee, get_atom::value).await(
    [&](const std::string& str) {
      // handle str
    },
    after(std::chrono::seconds(30)) >> [&]() {
      // handle error
    }
  );
}
```

16.3 Mixing Actors and Threads with Scoped Actors

The class `scoped_actor` offers a simple way of communicating with CAF actors from non-actor contexts. It overloads `operator->` to return a `blocking_actor*`. Hence, it behaves like the implicit `self` pointer in functor-based actors, only that it ceases to exist at scope end.

```
void test() {
  scoped_actor self;
  // spawn some monitored actor
  auto aut = self->spawn<monitored>(my_actor_impl);
  self->sync_send(aut, "hi there").await(
    ... // handle response
  );
  // self will be destroyed automatically here; any
  // actor monitoring it will receive down messages etc.
}
```

Note that `scoped_actor` throws an `actor_exited` exception when forced to quit for some reason, e.g., via an `exit_msg`. Whenever a `scoped_actor` might end up receiving an `exit_msg` (because it links itself to another actor for example), the caller either needs to handle the exception or the actor needs to process `exit_msg` manually via `self->trap_exit(true)`.

17 Strongly Typed Actors

Strongly typed actors provide a convenient way of defining type-safe messaging interfaces. Unlike untyped actorsd, typed actors are not allowed to use guard expressions. When calling `become` in a strongly typed actor, *all* message handlers from the typed interface must be set.

Typed actors use handles of type `typed_actor<...>` rather than `actor`, whereas the template parameters hold the messaging interface. For example, an actor responding to two integers with a dobule would use the type `typed_actor<replies_to<int, int>::with<double>>`. All functions for message passing, linking and monitoring are overloaded to accept both types of actors.

17.1 Spawning Typed Actors

Typed actors are spawned using the function `spawn_typed`. The argument to this function call *must* be a match expression as shown in the example below, because the runtime of CAF needs to evaluate the signature of each message handler.

```
auto p0 = spawn_typed(
  [](int a, int b) {
   return static_cast<double>(a) * b;
  },
  [](double a, double b) {
    return std::make_tuple(a * b, a / b);
  }
);
// assign to identical type
using full_type = typed_actor<
                    replies_to<int, int>::with<double>,
                    replies_to<double, double>::with<double, double>
                  >;
full_type p1 = p0;
// assign to subtype
using subtype1 = typed_actor<
                   replies_to<int, int>::with<double>
                 >;
subtype1 p2 = p0;
// assign to another subtype
using subtype2 = typed_actor<
                   replies_to<double, double>::with<double, double>
                 >;
subtype2 p3 = p0;
```

17.2 Class-based Typed Actors

Typed actors are spawned using the function `spawn_typed` and define their message passing interface as list of `replies_to<...>::with<...>` statements. This interface is used in (1) `typed_event_based_actor<...>`, which is the base class for typed actors, (2) the handle type `typed_actor<...>`, and (3) `typed_behavior<...>`, i.e., the behavior definition for typed actors. Since this is rather redundant, the actor handle provides definitions for the behavior as well as the base class, as shown in the example below. It is worth mentioning that all typed actors always use the event-based implementation, i.e., there is no typed actor implementation providing a blocking API.

```
struct shutdown_request { };
struct plus_request { int a; int b; };
struct minus_request { int a; int b; };

typedef typed_actor<replies_to<plus_request>::with<int>,
                    replies_to<minus_request>::with<int>,
                    replies_to<shutdown_request>::with<void>>
        calculator_type;

calculator_type::behavior_type
typed_calculator(calculator_type::pointer self) {
    return {
        [](const plus_request& pr) {
            return pr.a + pr.b;
        },
        [](const minus_request& pr) {
            return pr.a - pr.b;
        },
        [=](const shutdown_request&) {
            self->quit();
        }
    };
}

class typed_calculator_class : public calculator_type::base {
 protected: behavior_type make_behavior() override {
        return {
            [](const plus_request& pr) {
                return pr.a + pr.b;
            },
            [](const minus_request& pr) {
                return pr.a - pr.b;
            },
            [=](const shutdown_request&) {
                quit();
            }
        };
    }
};
```

```
void tester(event_based_actor* self, const calculator_type& testee) {
    self->link_to(testee);
    // will be invoked if we receive an unexpected response message
    self->on_sync_failure([=] {
        aout(self) << "AUT (actor under test) failed" << endl;
        self->quit(exit_reason::user_shutdown);
    });
    // first test: 2 + 1 = 3
    self->sync_send(testee, plus_request{2, 1}).then(
        [=](int r1) {
            assert(r1 == 3);
            // second test: 2 - 1 = 1
            self->sync_send(testee, minus_request{2, 1}).then(
                [=](int r2) {
                    assert(r2 == 1);
                    // both tests succeeded
                    aout(self) << "AUT (actor under test) "
                               << "seems to be ok"
                               << endl;
                    self->send(testee, shutdown_request{});
                }
            );
        }
    );
}

int main() {
    // announce custom message types
    announce<shutdown_request>("shutdown_request");
    announce<plus_request>("plus_request",
                           &plus_request::a, &plus_request::b);
    announce<minus_request>("minus_request",
                            &minus_request::a, &minus_request::b);
    // test function-based impl
    spawn(tester, spawn_typed(typed_calculator));
    await_all_actors_done();
    // test class-based impl
    spawn(tester, spawn_typed<typed_calculator_class>());
    await_all_actors_done();
    // done
    shutdown();
    return 0;
}
```

18 Messages

Messages in CAF are type-erased, copy-on-write tuples. The actual message type itself is usually hidden, as actors use pattern matching to decompose messages automatically. However, the classes `message` and `message_builder` allow more advanced usage scenarios than only sending data from one actor to another.

18.1 Class `message`

Member functions

Observers

`bool empty()`	Returns whether this message is empty
`size_t size()`	Returns the size of this message
`const void* at(size_t p)`	Returns a const pointer to the element at position `p`
`template <class T>` `const T& get_as(size_t p)`	Returns a const ref. to the element at position `p`
`template <class T>` `bool match_element(size_t p)`	Returns whether the element at position `p` has type `T`
`template <class... Ts>` `bool match_elements()`	Returns whether this message has the types `Ts...`
`message drop(size_t n)`	Creates a new message with all but the first `n` values
`message drop_right(size_t n)`	Creates a new message with all but the last `n` values
`message take(size_t n)`	Creates a new message from the first `n` values
`message take_right(size_t n)`	Creates a new message from the last `n` values
`message slice(size_t p, size_t n)`	Creates a new message from `[p, p + n)`
`message slice(size_t p, size_t n)`	Creates a new message from `[p, p + n)`
`message extract(message_handler)`	See 18.3
`message extract_opts(...)`	See 18.4

Modifiers

`optional<message>` `apply(message_handler f)`	Returns `f(*this)`
`void* mutable_at(size_t p)`	Returns a pointer to the element at position p
`template <class T>` `T& get_as_mutable(size_t p)`	Returns a reference to the element at position p

18.2 Class `message_builder`

Member functions

Constructors

`()`	Creates an empty message builder
`template <class Iter>` `(Iter first, Iter last)`	Adds all elements from range `[first, last)`

Observers

`bool empty()`	Returns whether this message is empty
`size_t size()`	Returns the size of this message
`message to_message()`	Converts the buffer to an actual message object
`template <class T> append(T val)`	Adds `val` to the buffer
`template <class Iter>` `append(Iter first, Iter last)`	Adds all elements from range `[first, last)`
`message extract(message_handler)`	See 18.3
`message extract_opts(...)`	See 18.4

Modifiers

`optional<message>` `apply(message_handler f)`	Returns `f(*this)`
`message move_to_message()`	Transfers ownership of its data to the new message **Warning**: this function leaves the builder in an *invalid state*, i.e., calling any member function on it afterwards is undefined behavior

18.3 Extracting

The member function `message::extract` removes matched elements from a message. x Messages are filtered by repeatedly applying a message handler to the greatest remaining slice, whereas slices are generated in the sequence `[0, size)`, `[0, size-1)`, `...`, `[1, size-1)`, `...`, `[size-1, size)`. Whenever a slice is matched, it is removed from the message and the next slice starts at the same index on the reduced message.

For example:

```
auto msg = make_message(1, 2.f, 3.f, 4);
// remove float and integer pairs
auto msg2 = msg.extract({
  [](float, float) { },
  [](int, int) { }
});
assert(msg2 == make_message(1, 4));
```

Step-by-step explanation:

- Slice 1: `(1, 2.f, 3.f, 4)`, no match
- Slice 2: `(1, 2.f, 3.f)`, no match
- Slice 3: `(1, 2.f)`, no match
- Slice 4: `(1)`, no match
- Slice 5: `(2.f, 3.f, 4)`, no match
- Slice 6: `(2.f, 3.f)`, *match*; new message is `(1, 4)`
- Slice 7: `(4)`, no match

Slice 7 is `(4)`, i.e., does not contain the first element, because the match on slice 6 occurred at index position 1. The function `extract` iterates a message only once, from left to right. The returned message contains the remaining, i.e., unmatched, elements.

18.4 Extracting Command Line Options

The class `message` also contains a convenience interface to `extract` for parsing command line options: the member function `extract_opts`.

```
int main(int argc, char** argv) {
  uint16_t port;
  string host = "localhost";
  auto res = message_builder(argv + 1, argv + argc).extract_opts({
    {"port,p", "set port", port},
    {"host,H", "set host (default: localhost)", host},
    {"verbose,v", "enable verbose mode"}
  });
  if (res.opts.count("help") > 0) {
    // CLI arguments contained "-h", "--help", or "-?" (builtin);
    // note: the help text has already been printed to stdout
    return 0;
  }
  if (!res.remainder.empty()) {
    // ... extract did not consume all CLI options ...
  }
  if (res.opts.count("verbose") > 0) {
    // ... enable verbose mode ...
  }
  // ...
}

/*
Output of ./program_name -h:

Allowed options:
  -p [--port] arg  : set port
  -H [--host] arg  : set host (default: localhost)
  -v [--verbose]   : enable verbose mode
*/
```

19 Common Pitfalls

19.1 Defining Patterns

- C++ evaluates comma-separated expressions from left-to-right, using only the last element as return type of the whole expression. This means that message handlers and behaviors must *not* be initialized like this:

```
message_handler wrong = (
  [](int i) { /*...*/ },
  [](float f) { /*...*/ }
);
```

The correct way to initialize message handlers and behaviors is to either use the constructor or the member function `assign`:

```
message_handler ok1{
  [](int i) { /*...*/ },
  [](float f) { /*...*/ }
};

message_handler ok2;
// some place later
ok2.assign(
  [](int i) { /*...*/ },
  [](float f) { /*...*/ }
);
```

19.2 Event-Based API

- The functions `become` and `handle_response` do not block, i.e., always return immediately. Thus, one should *always* capture by value in lambda expressions, because all references on the stack will cause undefined behavior if the lambda expression is executed.

19.3 Synchronous Messages

- A handle returned by `sync_send` represents *exactly one* response message. Therefore, it is not possible to receive more than one response message.
- The handle returned by `sync_send` is bound to the calling actor. It is not possible to transfer a handle to a response to another actor.

19.4 Sharing

- It is strongly recommended to **not** share states between actors. In particular, no actor shall ever access member variables or member functions of another actor. Accessing shared memory segments concurrently can cause undefined behavior that is incredibly hard to find and debug. However, sharing *data* between actors is fine, as long as the data is *immutable* and its lifetime is guaranteed to outlive all actors. The simplest way to meet the lifetime guarantee is by storing the data in smart pointers such as `std::shared_ptr`. Nevertheless, the recommended way of sharing informations is message passing. Sending the same message to multiple actors does not result in copying the data several times.

19.5 Constructors of Class-based Actors

- You should **not** try to send or receive messages in a constructor or destructor, because the actor is not fully initialized at this point.

20 Appendix

20.1 Class `option`

Defined in header `"caf/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` has a value
`bool empty()` `bool operator!()`	Returns `true` if `this` does **not** has a value
`const T& get()` `const T& operator*()`	Access stored value
`const T& get_or_else(const T& x)`	Returns `get()` if valid, `x` otherwise

Modifiers

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

20.2 Using `aout` – A Concurrency-safe Wrapper for `cout`

When using `cout` from multiple actors, output often appears interleaved. Moreover, using `cout` from multiple actors – and thus from multiple threads – in parallel should be avoided regardless, since the standard does not guarantee a thread-safe implementation.

By replacing `std::cout` with `caf::aout`, actors can achieve a concurrency-safe text output. The header `caf/all.hpp` also defines overloads for `std::endl` and `std::flush` for `aout`, but does not support the full range of ostream operations (yet). Each write operation to `aout` sends a message to a 'hidden' actor (keep in mind, sending messages from actor constructors is not safe). This actor only prints lines, unless output is forced using `flush`. The example below illustrates printing of lines of text from multiple actors (in random order).

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

#include "caf/all.hpp"

using namespace caf;
using std::endl;

using done_atom = atom_constant<atom("done")>;

int main() {
  std::srand(std::time(0));
  for (int i = 1; i <= 50; ++i) {
    spawn<blocking_api>([i](blocking_actor* self) {
      aout(self) << "Hi there! This is actor nr. "
                 << i << "!" << endl;
      std::chrono::milliseconds tout{std::rand() % 1000};
      self->delayed_send(self, tout, done_atom::value);
      self->receive(
        [i, self](done_atom) {
          aout(self) << "Actor nr. "
                     << i << " says goodbye!" << endl;
        }
      );
    });
  }
  // wait until all other actors we've spawned are done
  await_all_actors_done();
  shutdown();
}
```

20.3 Migration Guides

The guides in this section document all possibly breaking changes in the library for that last versions of CAF.

20.3.1 0.8 ⇒ 0.9

Version 0.9 included a lot of changes and improvements in its implementation, but it also made breaking changes to the API.

`self` has been removed

This is the biggest library change since the initial release. The major problem with this keyword-like identifier is that it must have a single type as it's implemented as a thread-local variable. Since there are so many different kinds of actors (event-based or blocking, untyped or typed), `self` needs to perform type erasure at some point, rendering it ultimately useless. Instead of a thread-local pointer, you can now use the first argument in functor-based actors to "catch" the self pointer with proper type information.

`actor_ptr` has been replaced

CAF now distinguishes between handles to actors, i.e., `typed_actor<...>` or simply `actor`, and *addresses* of actors, i.e., `actor_addr`. The reason for this change is that each actor has a logical, (network-wide) unique address, which is used by the networking layer of CAF. Furthermore, for monitoring or linking, the address is all you need. However, the address is not sufficient for sending messages, because it doesn't have any type information. The function `last_sender()` now returns the *address* of the sender. This means that previously valid code such as `send(last_sender(), ...)` will cause a compiler error. However, the recommended way of replying to messages is to return the result from the message handler.

The API for typed actors is now similar to the API for untyped actors

The APIs of typed and untyped actors have been harmonized. Typed actors can now be published in the network and also use all operations untyped actors can.

20.3.2 0.9 ⇒ 0.10 (`libcppa` ⇒ CAF)

The first release under the new name CAF is an overhaul of the entire library. Some classes have been renamed or relocated, others have been removed. The purpose of this refactoring was to make the library easier to grasp and to make its API more consistent. All classes now live in the namespace `caf` and all headers have the top level folder "caf" instead of "cppa". For example, `#include "cppa/actor.hpp"` becomes `#include "caf/actor.hpp"`. Further, the convenience header to get all parts of the user API is now `"caf/all.hpp"`. The networking has been separated from the core library. To get the networking components, simply include `"caf/io/all.hpp"` and use the namespace `caf::io`, e.g., `caf::io::remote_actor`.

Version 0.10 still includes the header `cppa/cppa.hpp` to make the transition process for users easier and to not break existing code right away. The header defines the namespace `cppa` as an alias for `caf`. Furthermore, it provides implementations or type aliases for renamed or removed classes such as `cow_tuple`. You won't get any warning about deprecated headers with 0.10. However, we will add this warnings in the next library version and remove deprecated code eventually.

Even when using the backwards compatibility header, the new library has breaking changes. For instance, guard expressions have been removed entirely. The reasoning behind this decision is that we already have projections to modify the outcome of a match. Guard expressions add little expressive power to the library but a whole lot of code that is hard to maintain in the long run due to its complexity. Using projections to not only perform type conversions but also to restrict values is the more natural choice.

The following table summarizes the changes made to the API.

Change	Explanation
`any_tuple => message`	This type is only being used to pass a message from one actor to another. Hence, `message` is the logical name.
`partial_function => message_handler`	Technically, it still is a partial function, but wanted to emphasize its use case in the library.
`cow_tuple => X`	We want to provide a streamlined, simple API. Shipping a full tuple abstraction with the library does not fit into this philosophy. The removal of `cow_tuple` implies the removal of related functions such as `tuple_cast`.
`cow_ptr => X`	This pointer class is an implementation detail of `message` and should not live in the global namespace in the first place. It also had the wrong name, because it is intrusive.
`X => message_builder`	This new class can be used to create messages dynamically. For example, the content of a vector can be used to create a message using a series of `append` calls.
`accept_handle`, `connection_handle`, `publish`, `remote_actor`, `max_msg_size`, `typed_publish`, `typed_remote_actor`, `publish_local_groups`, `new_connection_msg`, `new_data_msg`, `connection_closed_msg`, `acceptor_closed_msg`	These classes concern I/O functionality and have thus been moved to `caf::io`.

20.3.3 0.10 ⇒ 0.11

Version 0.11 introduced new, optional components. The core library itself, however, mainly received optimizations and bugfixes with one exception: the member function `on_exit` is no longer virtual. You can still provide it to define a custom exit handler, but you must not use `override`.

20.3.4 0.11 ⇒ 0.12

Version 0.12 removed two features:

- Type names are no longer demangled automatically. Hence, users must explicitly pass the type name as first argument when using `announce`, i.e., `announce<my_class>(...)` becomes `announce<my_class>("my_class", ...)`.
- Synchronous send blocks no longer support `continue_with`. This feature has been removed without substitution.

20.3.5 0.12 ⇒ 0.13

This release removes the (since 0.9 deprecated) `cppa` headers and deprecates all `*_send_tuple` versions (simply use the function without `_tuple` suffix).

In case you were using the old `cppa::options_description` API, you can migrate to the new API based on `extract` (cf. 18.4).

Most importantly, version 0.13 slightly changes `last_dequeued` and `last_sender`. Both functions will now cause undefined behavior (dereferencing a `nullptr`) instead of returning dummy values when accessed from outside a callback or after forwarding the current message. Besides, these function names were not a good choice in the first place, since "last" implies accessing data received in the past. As a result, both functions are now deprecated. Their replacements are named `current_message` and `current_sender` (cf. Section 4.2).

20.3.6 0.13 ⇒ 0.14

The function `timed_sync_send` has been removed. It offered an alternative way of defining message handlers, which is inconsistent with the rest of the API.