CAF

The C++ Actor Framework

User Manual

Version 0.17.3+exp.sha.c6050d9

SHA

Dominik Charousset

November 25, 2019

Contents

Part I

Core Library

1 Introduction

Before diving into the API of CAF, we discuss the concepts behind it and 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 asynchronous message passing. Decoupling concurrently running software components via message passing 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.

Implementing applications on top of low-level primitives such as mutexes and semaphores has proven challenging and error-prone. In particular when trying to implement applications that scale up to many CPU cores. Queueing, starvation, priority inversion, and false sharing are only a few of the issues that can decrease performance significantly in mutex-based concurrency models. In the extreme, an application written with the standard toolkit can run 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 scale dynamically from one core to many cores and from one node to many nodes. 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

CAF is inspired by other implementations based on the actor model such as Erlang or Akka. It aims to provide a modern C++ API allowing for type-safe as well as dynamically typed messaging. While there are similarities to other implementations, we made many different design decisions that lead to slight differences when comparing CAF to other actor frameworks.

1.2.1 Dynamically Typed Actor

A dynamically typed actor accepts any kind of message and dispatches on its content dynamically at the receiver. This is the “traditional” messaging style found in implementations like Erlang or Akka. The upside of this approach is (usually) faster prototyping and less code. This comes at the cost of requiring excessive testing.

1.2.2 Statically Typed Actor

CAF achieves static type-checking for actors by defining abstract messaging interfaces. Since interfaces define both input and output types, CAF is able to verify messaging protocols statically. The upside of this approach is much higher robustness to code changes and fewer possible runtime

errors. This comes at an increase in required source code, as developers have to define and use messaging interfaces.

1.2.3 Actor References

CAF uses reference counting for actors. The three ways to store a reference to an actor are addresses, handles, and pointers. Note that *address* does not refer to a *memory region* in this context.

Address Each actor has a (network-wide) unique logical address. This identifier is represented by `actor_addr`, which allows to identify and monitor an actor. Unlike other actor frameworks, CAF does *not* allow users to send messages to addresses. This limitation is due to the fact that the address does not contain any type information. Hence, it would not be safe to send it a message, because the receiving actor might use a statically typed interface that does not accept the given message. Because an `actor_addr` fills the role of an identifier, it has *weak reference semantics* (see § 9).

Handle An actor handle contains the address of an actor along with its type information and is required for sending messages to actors. 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 enforce static type checking for all messages. Dynamically typed actors use `actor` handles, while statically typed actors use `typed_actor<...>` handles. Both types have *strong reference semantics* (see § 9).

Pointer In a few instances, CAF uses `strong_actor_ptr` to refer to an actor using *strong reference semantics* (see § 9) without knowing the proper handle type. Pointers must be converted to a handle via `actor_cast` (see § 9.4) prior to sending messages. A `strong_actor_ptr` can be *null*.

1.2.4 Spawning

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

1.2.5 Monitor

A monitored actor sends a down message (see § 6.4.1) to all actors monitoring it as part of its termination. This allows actors to supervise other actors and to take actions when one of the supervised actors fails, i.e., terminates with a non-normal exit reason.

1.2.6 Link

A link is a bidirectional connection between two actors. Each actor sends an exit message (see § 6.4.2) to all of its links as part of its termination. Unlike down messages, exit messages cause the receiving actor to terminate as well when receiving a non-normal exit reason

per default. This allows developers to create a set of actors with the guarantee that either all or no actors are alive. Actors can override the default handler to implement error recovery strategies.

1.3 Experimental Features

Sections that discuss experimental features are highlighted with experimental. The API of such features is not stable. This means even minor updates to CAF can come with breaking changes to the API or even remove a feature completely. However, we encourage developers to extensively test such features and to start discussions to uncover flaws, report bugs, or tweaking the API in order to improve a feature or streamline it to cover certain use cases.

2 Overview

Compiling CAF requires CMake and a C++11-compatible compiler. To get and compile the sources on UNIX-like systems, type the following in a terminal:

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

We recommended to run the unit tests as well:

```
make test
```

If the output indicates an error, 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`.

2.1 Features

- 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 Minimal Compiler Versions

- GCC 4.8
- Clang 3.4
- Visual Studio 2015, Update 3

2.3 Supported Operating Systems

- Linux
- Mac OS X
- Windows (static library only)

2.4 Hello World Example

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

#include "caf/all.hpp"

using std::endl;
using std::string;

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;
      // 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->request(buddy, std::chrono::seconds(10), "Hello World!").then(
    // ... wait up to 10s for a response ...
    [=](const string& what) {
      // ... and print it
      aout(self) << what << endl;
    }
  );
}

void caf_main(actor_system& system) {
  // create a new actor that calls 'mirror()'
  auto mirror_actor = system.spawn(mirror);
  // create another actor that calls 'hello_world(mirror_actor)';
  system.spawn(hello_world, mirror_actor);
  // system will wait until both actors are destroyed before leaving main
}

// creates a main function for us that calls our caf_main
CAF_MAIN()
```

3 Type Inspection (Serialization and String Conversion)

CAF is designed with distributed systems in mind. Hence, all message types must be serializable and need a platform-neutral, unique name that is configured at startup (see § 11.3). Using a message type that is not serializable causes a compiler error (see § 3.4). CAF serializes individual elements of a message by using the inspection API. This API allows users to provide code for serialization as well as string conversion with a single free function. The signature for a class `my_class` is always as follows:

```
template <class Inspector>
typename Inspector::result_type inspect(Inspector& f, my_class& x) {
  return f(...);
}
```

The function `inspect` passes meta information and data fields to the variadic call operator of the inspector. The following example illustrates an implementation for `inspect` for a simple POD struct.

```
// POD struct foo
struct foo {
  std::vector<int> a;
  int b;
};

// foo needs to be serializable
template <class Inspector>
typename Inspector::result_type inspect(Inspector& f, foo& x) {
  return f(meta::type_name("foo"), x.a, x.b);
}
```

The inspector recursively inspects all data fields and has builtin support for (1) `std::tuple`, (2) `std::pair`, (3) C arrays, (4) any container type with `x.size()`, `x.empty()`, `x.begin()` and `x.end()`.

We consciously made the inspect API as generic as possible to allow for extensibility. This allows users to use CAF's types in other contexts, to implement parsers, etc.

3.1 Inspector Concept

The following concept class shows the requirements for inspectors. The placeholder `T` represents any user-defined type. For example, `error` when performing I/O operations or some integer type when implementing a hash function.

```
Inspector {
  using result_type = T;

  if (inspector only requires read access to the state of T)
    static constexpr bool reads_state = true;
  else
    static constexpr bool writes_state = true;

  template <class... Ts>
  result_type operator()(Ts&&...);
}
```

A saving `Inspector` is required to handle constant lvalue and rvalue references. A loading `Inspector`

must only accept mutable lvalue references to data fields, but still allow for constant lvalue references and rvalue references to annotations.

3.2 Annotations

Annotations allow users to fine-tune the behavior of inspectors by providing addition meta information about a type. All annotations live in the namespace `caf::meta` and derive from `caf::meta::annotation`. An inspector can query whether a type `T` is an annotation with `caf::meta::is_annotation<T>::value`. Annotations are passed to the call operator of the inspector along with data fields. The following list shows all annotations supported by CAF:

- `type_name(n)`: Display type name as `n` in human-friendly output (position before data fields).
- `hex_formatted()`: Format the following data field in hex format.
- `omittable()`: Omit the following data field in human-friendly output.
- `omittable_if_empty()`: Omit the following data field if it is empty in human-friendly output.
- `omittable_if_none()`: Omit the following data field if it equals `none` in human-friendly output.
- `save_callback(f)`: Call `f` when serializing (position after data fields).
- `load_callback(f)`: Call `f` after deserializing all data fields (position after data fields).

3.3 Backwards and Third-party Compatibility

CAF evaluates common free function other than `inspect` in order to simplify users to integrate CAF into existing code bases.

Serializers and deserializers call user-defined `serialize` functions. Both types support `operator&` as well as `operator()` for individual data fields. A `serialize` function has priority over `inspect`.

When converting a user-defined type to a string, CAF calls user-defined `to_string` functions and prefers those over `inspect`.

3.4 Whitelisting Unsafe Message Types

Message types that are not serializable cause compile time errors when used in actor communication. When using CAF for concurrency only, this errors can be suppressed by whitelisting types with `CAF_ALLOW_UNSAFE_MESSAGE_TYPE`. The macro is defined as follows.

3.5 Splitting Save and Load Operations

If loading and storing cannot be implemented in a single function, users can query whether the inspector is loading or storing. For example, consider the following class `foo` with getter and setter functions and no public access to its members.

```
// no friend access for ‘inspect‘
class foo {
public:
  foo(int a0 = 0, int b0 = 0) : a_(a0), b_(b0) {
    // nop
  }

  foo(const foo&) = default;
  foo& operator=(const foo&) = default;

  int a() const {
    return a_;
  }

  void set_a(int val) {
    a_ = val;
  }

  int b() const {
    return b_;
  }

  void set_b(int val) {
    b_ = val;
  }

private:
  int a_;
  int b_;
};
```

Since there is no access to the data fields `a_` and `b_` (and assuming no changes to `foo` are possible), we need to split our implementation of `inspect` as shown below.

```
template <class Inspector>
typename std::enable_if<Inspector::reads_state,
                        typename Inspector::result_type>::type
inspect(Inspector& f, foo& x) {
  return f(meta::type_name("foo"), x.a(), x.b());
}

template <class Inspector>
typename std::enable_if<Inspector::writes_state,
                        typename Inspector::result_type>::type
inspect(Inspector& f, foo& x) {
  int a;
  int b;
  // write back to x at scope exit
  auto g = make_scope_guard([&] {
    x.set_a(a);
    x.set_b(b);
  });
  return f(meta::type_name("foo"), a, b);
}

behavior testee(event_based_actor* self) {
  return {
    [=](const foo& x) {
      aout(self) << to_string(x) << endl;
    }
  };
}
```

The purpose of the scope guard in the example above is to write the content of the temporaries back to `foo` at scope exit automatically. Storing the result of `f(...)` in a temporary first and then writing the changes to `foo` is not possible, because `f(...)` can return `void`.

4 Message Handlers

Actors can store a set of callbacks—usually implemented as lambda expressions—using either `behavior` or `message_handler`. The former stores an optional timeout, while the latter is composable.

4.1 Definition and Composition

As the name implies, a `behavior` defines the response of an actor to messages it receives. The optional timeout allows an actor to dynamically change its behavior when not receiving message after a certain amount of time.

```
message_handler x1{
  [](int i) { /*...*/ },
  [](double db) { /*...*/ },
  [](int a, int b, int c) { /*...*/ }
};
```

In our first example, `x1` models a behavior accepting messages that consist of either exactly one `int`, or one `double`, or three `int` values. Any other message is not matched and gets forwarded to the default handler (see § 6.4.4).

```
message_handler x2{
  [](double db) { /*...*/ },
  [](double db) { /* - unreachable - */ }
};
```

Our second example illustrates an important characteristic of the matching mechanism. Each message is matched against the callbacks in the order they are defined. The algorithm stops at the first match. Hence, the second callback in `x2` is unreachable.

```
message_handler x3 = x1.or_else(x2);
message_handler x4 = x2.or_else(x1);
```

Message handlers can be combined using `or_else`. This composition is not commutative, as our third examples illustrates. The resulting message handler will first try to handle a message using the left-hand operand and will fall back to the right-hand operand if the former did not match. Thus, `x3` behaves exactly like `x1`. This is because the second callback in `x1` will consume any message with a single `double` and both callbacks in `x2` are thus unreachable. The handler `x4` will consume messages with a single `double` using the first callback in `x2`, essentially overriding the second callback in `x1`.

4.2 Atoms

Defining message handlers in terms of callbacks is convenient, but requires a simple way to annotate messages with meta data. Imagine an actor that provides a mathematical service for integers. It receives two integers, performs a user-defined operation and returns the result. Without additional context, the actor cannot decide whether it should multiply or add the integers. 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 in CAF are mapped to integer values at compile time. 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 translates to a whitespace character.

While the `atom_value` is computed at compile time, it is not uniquely typed and thus cannot be used in the signature of a callback. To accomplish this, CAF offers compile-time *atom constants*.

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

Using these 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;
  }
};

// caller side: send(math_actor, add_atom::value, 1, 2)
```

Atom constants define a static member `value`. Please note that this static `value` member does *not* have the type `atom_value`, unlike `std::integral_constant` for example.

5 Actors

Actors in CAF are a lightweight abstraction for units of computations. They are active objects in the sense that they own their state and do not allow others to access it. The only way to modify the state of an actor is sending messages to it.

CAF provides several actor implementations, each covering a particular use case. The available implementations differ in three characteristics: (1) dynamically or statically typed, (2) class-based or function-based, and (3) using asynchronous event handlers or blocking receives. These three characteristics can be combined freely, with one exception: statically typed actors are always event-based. For example, an actor can have dynamically typed messaging, implement a class, and use blocking receives. The common base class for all user-defined actors is called `local_actor`.

Dynamically typed actors are more familiar to developers coming from Erlang or Akka. They (usually) enable faster prototyping but require extensive unit testing. Statically typed actors require more source code but enable the compiler to verify communication between actors. Since CAF supports both, developers can freely mix both kinds of actors to get the best of both worlds. A good rule of thumb is to make use of static type checking for actors that are visible across multiple translation units.

Actors that utilize the blocking receive API always require an exclusive thread of execution. Event-based actors, on the other hand, are usually scheduled cooperatively and are very lightweight with a memory footprint of only few hundred bytes. Developers can exclude—detach—event-based actors that potentially starve others from the cooperative scheduling while spawning it. A detached actor lives in its own thread of execution.

5.1 Environment / Actor Systems

All actors live in an `actor_system` representing an actor environment including scheduler (see § 7), registry (see § 8), and optional components such as a middleman (see § 17). A single process can have multiple `actor_system` instances, but this is usually not recommended (a use case for multiple systems is to strictly separate two or more sets of actors by running them in different schedulers). For configuration and fine-tuning options of actor systems see § 11. A distributed CAF application consists of two or more connected actor systems. We also refer to interconnected `actor_system` instances as a *distributed actor system*.

5.2 Common Actor Base Types

The following pseudo-UML depicts the class diagram for actors in CAF. Irrelevant member functions and classes as well as mixins are omitted for brevity. Selected individual classes are presented in more detail in the following sections.

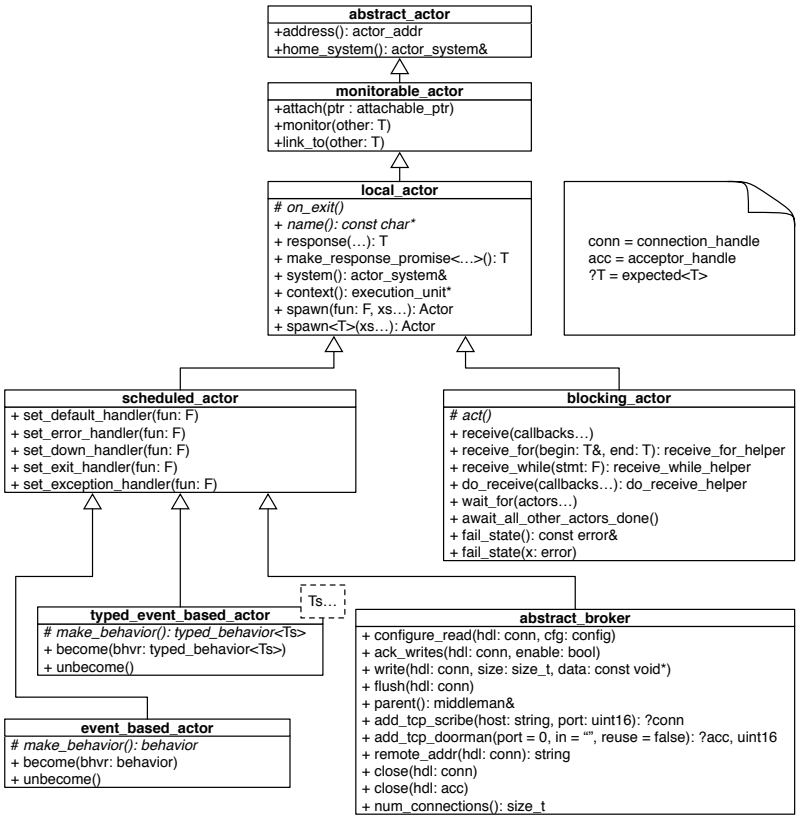


Figure 1: Actor Types in CAF

5.2.1 Class `local_actor`

The class `local_actor` is the root type for all user-defined actors in CAF. It defines all common operations. However, users of the library usually do not inherit from this class directly. Proper base classes for user-defined actors are `event_based_actor` or `blocking_actor`. The following table also includes member function inherited from `monitorable_actor` and `abstract_actor`.

Types	
`mailbox_type`	A concurrent, many-writers-single-reader queue type.

Constructors	
`(actor_config&)`	Constructs the actor using a config.

Observers	
`actor_addr address()`	Returns the address of this actor.
`actor_system& system()`	Returns `context()->system()`.
`actor_system& home_system()`	Returns the system that spawned this actor.
`execution_unit* context()`	Returns underlying thread or current scheduler worker.

Customization Points	
`on_exit()`	Can be overridden to perform cleanup code.
`const char* name()`	Returns a debug name for this actor type.

Actor Management	
`link_to(other)`	Link to an actor (see § 1.2.6).
`unlink_from(other)`	Remove link to an actor (see § 1.2.6).
`monitor(other)`	Unidirectionally monitors an actor (see § 1.2.5).
`demonitor(other)`	Removes a monitor from `whom`.
`spawn(F fun, xs...)`	Spawns a new actor from `fun`.
`spawn<T>(xs...)`	Spawns a new actor of type `T`.

Message Processing	
`T make_response_promise<Ts...>()`	Allows an actor to delay its response message.
`T response(xs...)`	Convenience function for creating fulfilled promises.

5.2.2 Class `scheduled_actor`

All scheduled actors inherit from `scheduled_actor`. This includes statically and dynamically typed event-based actors as well as brokers (see § 18).

Types	
`pointer`	`scheduled_actor*`
`exception_handler`	`function<error (pointer, std::exception_ptr&)>`
`default_handler`	`function<result<message> (pointer, message_view&)>`
`error_handler`	`function<void (pointer, error&)>`
`down_handler`	`function<void (pointer, down_msg&)>`
`exit_handler`	`function<void (pointer, exit_msg&)>`

Constructors	
`(actor_config&)`	Constructs the actor using a config.

Termination	
`quit()`	Stops this actor with normal exit reason.
`quit(error x)`	Stops this actor with error `x`.

Special-purpose Handlers	
`set_exception_handler(F f)`	Installs `f` for converting exceptions to errors (see § 10).
`set_down_handler(F f)`	Installs `f` to handle down messages (see § 6.4.1).
`set_exit_handler(F f)`	Installs `f` to handle exit messages (see § 6.4.2).
`set_error_handler(F f)`	Installs `f` to handle error messages (see § 6.4.3 and § 10).
`set_default_handler(F f)`	Installs `f` as fallback message handler (see § 6.4.4).

5.2.3 Class `blocking_actor`

A blocking actor always lives in its own thread of execution. They are not as lightweight as event-based actors and thus do not scale up to large numbers. The primary use case for blocking actors is to use a `scoped_actor` for ad-hoc communication to selected actors. Unlike scheduled actors, CAF does **not** dispatch system messages to special-purpose handlers. A blocking actors receives *all* messages regularly through its mailbox. A blocking actor is considered *done* only after it returned from `act` (or from the implementation in function-based actors). A `scoped_actor` sends its exit messages as part of its destruction.

Constructors	
`(actor_config&)`	Constructs the actor using a config.

Customization Points	
`void act()`	Implements the behavior of the actor.

Termination	
`const error& fail_state()`	Returns the current exit reason.
`fail_state(error x)`	Sets the current exit reason.

Actor Management	
`wait_for(Ts... xs)`	Blocks until all actors `xs...` are done.
`await_all_other_actors_done()`	Blocks until all other actors are done.

Message Handling	
`receive(Ts... xs)`	Receives a message using the callbacks `xs...`.
`receive_for(T& begin, T end)`	See § 5.10.3.
`receive_while(F stmt)`	See § 5.10.3.
`do_receive(Ts... xs)`	See § 5.10.3.

5.3 Messaging Interfaces

Statically typed actors require abstract messaging interfaces to allow the compiler to type-check actor communication. Interfaces in CAF are defined using the variadic template `typed_actor<...>`, which defines the proper actor handle at the same time. Each template parameter defines one `input/output` pair via `replies_to<X1,...,Xn>::with<Y1,...,Yn>`. For inputs that do not generate outputs, `reacts_to<X1,...,Xn>` can be used as shortcut for `replies_to<X1,...,Xn>::with<void>`. In the same way functions cannot be overloaded only by their return type, interfaces cannot accept one input twice (possibly mapping it to different outputs). The example below defines a messaging interface for a simple calculator.

```
using add_atom = atom_constant<atom("add")>;
using sub_atom = atom_constant<atom("sub")>;

using calculator_actor = typed_actor<replies_to<add_atom, int, int>::with<int>,
                                     replies_to<sub_atom, int, int>::with<int>>;
```

It is not required to create a type alias such as `calculator_actor`, but it makes dealing with statically typed actors much easier. Also, a central alias definition eases refactoring later on.

Interfaces have set semantics. This means the following two type aliases `i1` and `i2` are equal:

```
using i1 = typed_actor<replies_to<A>::with<B>, replies_to<C>::with<D>>;
using i2 = typed_actor<replies_to<C>::with<D>, replies_to<A>::with<B>>;
```

Further, actor handles of type `A` are assignable to handles of type `B` as long as `B` is a subset of `A`.

For convenience, the class `typed_actor<...>` defines the member types shown below to grant access to derived types.

Types	
`behavior_type`	A statically typed set of message handlers.
`base`	Base type for actors, i.e., `typed_event_based_actor<...>`.
`pointer`	A pointer of type `base*`.
`stateful_base<T>`	See § 5.7.
`stateful_pointer<T>`	A pointer of type `stateful_base<T>*`.
`extend<Ts...>`	Extend this typed actor with `Ts...`.
`extend_with<Other>`	Extend this typed actor with all cases from `Other`.

5.4 Spawning Actors

Both statically and dynamically typed actors are spawned from an `actor_system` using the member function `spawn`. The function either takes a function as first argument or a class as first template parameter. For example, the following functions and classes represent actors.

```
behavior calculator_fun(event_based_actor* self);
void blocking_calculator_fun(blocking_actor* self);
calculator_actor::behavior_type typed_calculator_fun();
class calculator;
class blocking_calculator;
class typed_calculator;
```

Spawning an actor for each implementation is illustrated below.

```
  auto a2 = system.spawn(calculator_fun);
  auto a3 = system.spawn(typed_calculator_fun);
  auto a4 = system.spawn<blocking_calculator>();
  auto a5 = system.spawn<calculator>();
  auto a6 = system.spawn<typed_calculator>();
  scoped_actor self{system};
```

Additional arguments to `spawn` are passed to the constructor of a class or used as additional function arguments, respectively. In the example above, none of the three functions takes any argument other than the implicit but optional `self` pointer.

5.5 Function-based Actors

When using a function or function object to implement an actor, the first argument *can* be used to capture a pointer to the actor itself. The type of this pointer is usually `event_based_actor*` or `blocking_actor*`. The proper pointer type for any `typed_actor` handle `T` can be obtained via `T::pointer` (see § 5.3).

Blocking actors simply implement their behavior in the function body. The actor is done once it returns from that function.

Event-based actors can either return a `behavior` (see § 4) that is used to initialize the actor or explicitly set the initial behavior by calling `self->become(...)`. Due to the asynchronous, event-based nature of this kind of actor, the function usually returns immediately after setting a behavior (message handler) for the *next* incoming message. Hence, variables on the stack will be out of scope once a message arrives. Managing state in function-based actors can be done either via rebinding state with `become`, using heap-located data referenced via `std::shared_ptr` or by using the "stateful actor" abstraction (see § 5.7).

The following three functions implement the prototypes shown in § 5.4 and illustrate one blocking actor and two event-based actors (statically and dynamically typed).

```
// function-based, dynamically typed, event-based API
behavior calculator_fun(event_based_actor*) {
  return {
    [](add_atom, int a, int b) {
      return a + b;
    },
    [](sub_atom, int a, int b) {
      return a - b;
    }
  };
}

// function-based, dynamically typed, blocking API
void blocking_calculator_fun(blocking_actor* self) {
  bool running = true;
  self->receive_while(running) (
    [](add_atom, int a, int b) {
      return a + b;
    },
    [](sub_atom, int a, int b) {
      return a - b;
    },
    [&](exit_msg& em) {
      if (em.reason) {
        self->fail_state(std::move(em.reason));
        running = false;
      }
    }
  );
}

// function-based, statically typed, event-based API
calculator_actor::behavior_type typed_calculator_fun() {
  return {
    [](add_atom, int a, int b) {
      return a + b;
    },
    [](sub_atom, int a, int b) {
      return a - b;
    }
  };
}
```

5.6 Class-based Actors

Implementing an actor using a class requires the following:

- Provide a constructor taking a reference of type `actor_config&` as first argument, which is forwarded to the base class. The config is passed implicitly to the constructor when calling `spawn`, which also forwards any number of additional arguments to the constructor.
- Override `make_behavior` for event-based actors and `act` for blocking actors.

Implementing actors with classes works for all kinds of actors and allows simple management of state via member variables. However, composing states via inheritance can get quite tedious. For dynamically typed actors, composing states is particularly hard, because the compiler cannot provide much help. For statically typed actors, CAF also provides an API for composable behaviors (see § 5.8) that works well with inheritance. The following three examples implement the forward declarations shown in § 5.4.

```
1  // class-based, dynamically typed, event-based API
2  class calculator : public event_based_actor {
3  public:
4    calculator(actor_config& cfg) : event_based_actor(cfg) {
5      // nop
6    }
7
8    behavior make_behavior() override {
9      return calculator_fun(this);
10   }
11 };
12
13 // class-based, dynamically typed, blocking API
14 class blocking_calculator : public blocking_actor {
15 public:
16   blocking_calculator(actor_config& cfg) : blocking_actor(cfg) {
17     // nop
18   }
19
20   void act() override {
21     blocking_calculator_fun(this);
22   }
23 };
24
25 // class-based, statically typed, event-based API
26 class typed_calculator : public calculator_actor::base {
27 public:
28   typed_calculator(actor_config& cfg) : calculator_actor::base(cfg) {
29     // nop
30   }
31
32   behavior_type make_behavior() override {
33     return typed_calculator_fun();
34   }
35 };
```

5.7 Stateful Actors

The stateful actor API makes it easy to maintain state in function-based actors. It is also safer than putting state in member variables, because the state ceases to exist after an actor is done and is not delayed until the destructor runs. For example, if two actors hold a reference to each other via member variables, they produce a cycle and neither will get destroyed. Using stateful actors instead breaks the cycle, because references are destroyed when an actor calls `self->quit()` (or is killed externally). The following example illustrates how to implement stateful actors with static typing as well as with dynamic typing.

```
using cell = typed_actor<reacts_to<put_atom, int>,
                         replies_to<get_atom>::with<int>>;

struct cell_state {
  int value = 0;
};

cell::behavior_type type_checked_cell(cell::stateful_pointer<cell_state> self) {
  return {
    [=](put_atom, int val) {
      self->state.value = val;
    },
    [=](get_atom) {
      return self->state.value;
    }
  };
}

behavior unchecked_cell(stateful_actor<cell_state>* self) {
  return {
    [=](put_atom, int val) {
      self->state.value = val;
    },
    [=](get_atom) {
      return self->state.value;
    }
  };
```

Stateful actors are spawned in the same way as any other function-based actor (see § 5.5).

```
  auto cell1 = system.spawn(type_checked_cell);
  auto cell2 = system.spawn(unchecked_cell);
```

5.8 Actors from Composable Behaviors experimental

When building larger systems, it is often useful to implement the behavior of an actor in terms of other, existing behaviors. The composable behaviors in CAF allow developers to generate a behavior class from a messaging interface (see § 5.3).

The base type for composable behaviors is `composable_behavior<T>`, where `T` is a `typed_actor<...>`. CAF maps each `replies_to<A,B,C>::with<D,E,F>` in `T` to a pure virtual member function with signature:

```
result<D, E, F> operator()(param<A>, param<B>, param<C>);.
```

Note that `operator()` will take integral types as well as atom constants simply by value. A `result<T>` accepts either a value of type `T`, a `skip_t` (see § 6.4.4), an `error` (see § 10), a `delegated<T>` (see § 6.7), or a `response_promise<T>` (see § 6.8). A `result<void>` is constructed by returning `unit`.

A behavior that combines the behaviors `X`, `Y`, and `Z` must inherit from `composed_behavior<X,Y,Z>` instead of inheriting from the three classes directly. The class `composed_behavior` ensures that the behaviors are concatenated correctly. In case one message handler is defined in multiple base types, the *first* type in declaration order "wins". For example, if `X` and `Y` both implement the interface `replies_to<int,int>::with<int>`, only the handler implemented in `X` is active.

Any composable (or composed) behavior with no pure virtual member functions can be spawned directly through an actor system by calling `system.spawn<...>()`, as shown below.

```
// using add_atom = atom_constant<atom("add")>; (defined in atom.hpp)
using multiply_atom = atom_constant<atom("multiply")>;

using adder = typed_actor<replies_to<add_atom, int, int>::with<int>>;
using multiplier = typed_actor<replies_to<multiply_atom, int, int>::with<int>>;

class adder_bhvr : public composable_behavior<adder> {
public:
  result<int> operator()(add_atom, int x, int y) override {
    return x + y;
  }
};

class multiplier_bhvr : public composable_behavior<multiplier> {
public:
  result<int> operator()(multiply_atom, int x, int y) override {
    return x * y;
  }
};

// calculator_bhvr can be inherited from or composed further
using calculator_bhvr = composed_behavior<adder_bhvr, multiplier_bhvr>;

} // namespace

void caf_main(actor_system& system) {
  auto f = make_function_view(system.spawn<calculator_bhvr>());
  cout << "10 + 20 = " << f(add_atom::value, 10, 20) << endl;
  cout << "7 * 9 = " << f(multiply_atom::value, 7, 9) << endl;
}

CAF_MAIN()
```

The second example illustrates how to use non-primitive values that are wrapped in a `param<T>` when working with composable behaviors. The purpose of `param<T>` is to provide a single interface for both constant and non-constant access. Constant access is modeled with the implicit conversion operator to a const reference, the member function `get()`, and `operator->`.

When acquiring mutable access to the represented value, CAF copies the value before allowing mutable access to it if more than one reference to the value exists. This copy-on-write optimization avoids race conditions by design, while minimizing copy operations (see § 6.2). A mutable reference is returned from the member functions `get_mutable()` and `move()`. The latter is a convenience function for `std::move(x.get_mutable())`. The following example illustrates how to use `param<std::string>` when implementing a simple dictionary.

```
using dict = typed_actor<reacts_to<put_atom, string, string>,
                         replies_to<get_atom, string>::with<string>>;

class dict_behavior : public composable_behavior<dict> {
public:
  result<string> operator()(get_atom, param<string> key) override {
    auto i = values_.find(key);
    if (i == values_.end())
      return "";
    return i->second;
  }

  result<void> operator()(put_atom, param<string> key,
                          param<string> value) override {
    if (values_.count(key) != 0)
      return unit;
    values_.emplace(key.move(), value.move());
    return unit;
  }

protected:
  std::unordered_map<string, string> values_;
};
```

5.9 Attaching Cleanup Code to Actors

Users can attach cleanup code to actors. This code is executed immediately if the actor has already exited. Otherwise, the actor will execute it as part of its termination. The following example attaches a function object to actors for printing a custom string on exit.

```
void print_on_exit(const actor& hdl, const std::string& name) {
  hdl->attach_functor([=](const error& reason) {
    cout << name << " exited: " << to_string(reason) << endl;
  });
}
```

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

5.10 Blocking Actors

Blocking actors always run in a separate thread and are not scheduled by CAF. Unlike event-based actors, blocking actors have explicit, blocking *receive* functions. Further, blocking actors do not handle system messages automatically via special-purpose callbacks (see § 6.4). This gives users full control over the behavior of blocking actors. However, blocking actors still should follow conventions of the actor system. For example, actors should unconditionally terminate after receiving an `exit_msg` with reason `exit_reason::kill`.

5.10.1 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. Messages that are not matched by the behavior are automatically skipped and remain in the mailbox.

```
self->receive (
  [](int x) { /* ... */ }
);
```

5.10.2 Catch-all Receive Statements

Blocking actors can use inline catch-all callbacks instead of setting a default handler (see § 6.4.4). A catch-all case must be the last callback before the optional timeout, as shown in the example below.

```
self->receive(
  [&](float x) {
    // ...
  },
  [&](const down_msg& x) {
    // ...
  },
  [&](const exit_msg& x) {
    // ...
  },
  others >> [](message_view& x) -> result<message> {
    // report unexpected message back to client
    return sec::unexpected_message;
  }
);
```

5.10.3 Receive Loops

Message handler passed to `receive` are temporary object at runtime. Hence, calling `receive` inside a loop creates an unnecessary amount of short-lived objects. CAF provides predefined receive loops to allow for more efficient code.

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

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

```
// BAD
size_t received = 0;
while (received < 10) {
  receive (
    [&](int) {
      ++received;
    }
  );
} ;

// GOOD
size_t received = 0;
receive_while([&] { return received < 10; }) (
  [&](int) {
    ++received;
  }
);
```

```
// BAD
size_t received = 0;
do {
  receive (
    [&](int) {
      ++received;
    }
  );
} while (received < 10);

// GOOD
size_t received = 0;
do_receive (
  [&](int) {
    ++received;
  }
).until([&] { return received >= 10; });
```

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

```
bool running = true;
self->receive_while([&] { return running; }) (
  [&](int value1) {
    self->receive (
      [&](float value2) {
        aout(self) << value1 << " => " << value2 << endl;
      }
    );
  },
  // ...
);
```

5.10.4 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(actor_system& system) {
  scoped_actor self{system};
  // spawn some actor
  auto aut = self->spawn(my_actor_impl);
  self->send(aut, "hi there");
  // self will be destroyed automatically here; any
  // actor monitoring it will receive down messages etc.
}
```

6 Message Passing

Message passing in CAF is always asynchronous. Further, CAF neither guarantees message delivery nor message ordering in a distributed setting. CAF uses TCP per default, but also enables nodes to send messages to other nodes without having a direct connection. In this case, messages are forwarded by intermediate nodes and can get lost if one of the forwarding nodes fails. Likewise, forwarding paths can change dynamically and thus cause messages to arrive out of order.

The messaging layer of CAF has three primitives for sending messages: `send`, `request`, and `delegate`. The former simply enqueues a message to the mailbox the receiver. The latter two are discussed in more detail in § 6.5 and § 6.7.

6.1 Structure of Mailbox Elements

When enqueuing a message to the mailbox of an actor, CAF wraps the content of the message into a `mailbox_element` (shown below) to add meta data and processing paths.

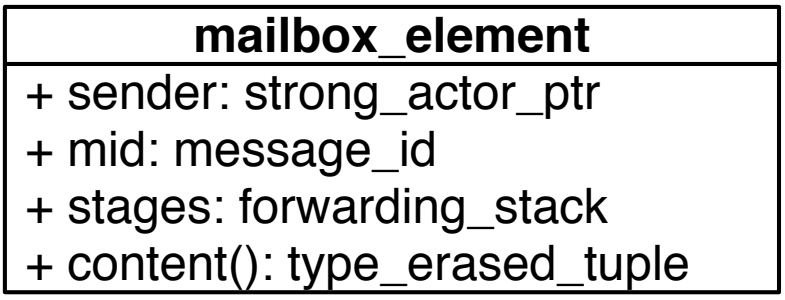


Figure 2: UML class diagram for `mailbox_element`

The sender is stored as a `strong_actor_ptr` (see § 7) and denotes the origin of the message. The message ID is either 0—invalid—or a positive integer value that allows the sender to match a response to its request. The `stages` vector stores the path of the message. Response messages, i.e., the returned values of a message handler, are sent to `stages.back()` after calling `stages.pop_back()`. This allows CAF to build pipelines of arbitrary size. If no more stage is left, the response reaches the sender. Finally, `content()` grants access to the type-erased tuple storing the message itself.

Mailbox elements are created by CAF automatically and are usually invisible to the programmer. However, understanding how messages are processed internally helps understanding the behavior of the message passing layer.

It is worth mentioning that CAF usually wraps the mailbox element and its content into a single object in order to reduce the number of memory allocations.

6.2 Copy on Write

CAF allows multiple actors to implicitly share message contents, as long as no actor performs writes. This allows groups (see § 13) to send the same content to all subscribed actors without any copying overhead.

Actors copy message contents whenever other actors hold references to it and if one or more arguments of a message handler take a mutable reference.

6.3 Requirements for Message Types

Message types in CAF must meet the following requirements:

1. Serializable or inspectable (see § 3)
2. Default constructible
3. Copy constructible

A type is serializable if it provides free function `serialize(Serializer&, T&)` or `serialize(Serializer&, T&, const unsigned int)`. Accordingly, a type is inspectable if it provides a free function `inspect(Inspector&, T&)`.

Requirement 2 is a consequence of requirement 1, because CAF needs to be able to create an object of a type before it can call `serialize` or `inspect` on it. Requirement 3 allows CAF to implement Copy on Write (see § 6.2).

6.4 Default and System Message Handlers

CAF has three system-level message types (`down_msg`, `exit_msg`, and `error`) that all actor should handle regardless of there current state. Consequently, event-based actors handle such messages in special-purpose message handlers. Additionally, event-based actors have a fallback handler for unmatched messages. Note that blocking actors have neither of those special-purpose handlers (see § 5.10).

6.4.1 Down Handler

Actors can monitor the lifetime of other actors by calling `self->monitor(other)`. This will cause the runtime system of CAF to send a `down_msg` for `other` if it dies. Actors drop down messages unless they provide a custom handler via `set_down_handler(f)`, where `f` is a function object with signature `void (down_msg&)` or `void (scheduled_actor*, down_msg&)`. The latter signature allows users to implement down message handlers as free function.

6.4.2 Exit Handler

Bidirectional monitoring with a strong lifetime coupling is established by calling `self->link_to(other)`. This will cause the runtime to send an `exit_msg` if either `this` or `other` dies. Per default, actors terminate after receiving an `exit_msg` unless the exit reason is `exit_reason::normal`. This mechanism propagates failure states in an actor system. Linked actors form a sub system in which an error causes all actors to fail collectively. Actors can override the default handler via `set_exit_handler(f)`, where `f` is a function object with signature `void (exit_message&)` or `void (scheduled_actor*, exit_message&)`.

6.4.3 Error Handler

Actors send error messages to others by returning an `error` (see § 10) from a message handler. Similar to exit messages, error messages usually cause the receiving actor to terminate, unless a custom handler was installed via `set_error_handler(f)`, where `f` is a function object with signature `void (error&)` or `void (scheduled_actor*, error&)`. Additionally, `request` accepts an error handler as second argument to handle errors for a particular request (see § 6.5.3). The default handler is used as fallback if `request` is used without error handler.

6.4.4 Default Handler

The default handler is called whenever the behavior of an actor did not match the input. Actors can change the default handler by calling `set_default_handler`. The expected signature of the function object is `result<message> (scheduled_actor*, message_view&)`, whereas the `self` pointer can again be omitted. The default handler can return a response message or cause the runtime to *skip* the input message to allow an actor to handle it in a later state. CAF provides the following built-in implementations: `reflect`, `reflect_and_quit`, `print_and_drop`, `drop`, and `skip`. The former two are meant for debugging and testing purposes and allow an actor to simply return an input. The next two functions drop unexpected messages with or without printing a warning beforehand. Finally, `skip` leaves the input message in the mailbox. The default is `print_and_drop`.

6.5 Requests

A main feature of CAF is its ability to couple input and output types via the type system. For example, a `typed_actor<replies_to<int>::with<int>>` essentially behaves like a function. It receives a single `int` as input and responds with another `int`. CAF embraces this functional take on actors by simply creating response messages from the result of message handlers. This allows CAF to match *request* to *response* messages and to provide a convenient API for this style of communication.

6.5.1 Sending Requests and Handling Responses

Actors send request messages by calling `request(receiver, timeout, content...)`. This function returns an intermediate object that allows an actor to set a one-shot handler for the response message. Event-based actors can use either `request(...).then` or `request(...).await`. The former multiplexes the one-shot handler with the regular actor behavior and handles requests as they arrive. The latter suspends the regular actor behavior until all awaited responses arrive and handles requests in LIFO order. Blocking actors always use `request(...).receive`, which blocks until the one-shot handler was called. Actors receive a `sec::request_timeout` (see § 10.3) error message (see § 6.4.3) if a timeout occurs. Users can set the timeout to `infinite` for unbound operations. This is only recommended if the receiver is running locally.

In our following example, we use the simple cell actors shown below as communication endpoints.

```
1 using cell = typed_actor<reacts_to<put_atom, int>,
2                          replies_to<get_atom>::with<int>>;
3
4 struct cell_state {
5   int value = 0;
```

```
};

cell::behavior_type cell_impl(cell::stateful_pointer<cell_state> self, int x0) {
  self->state.value = x0;
  return {
    [=](put_atom, int val) {
      self->state.value = val;
    },
    [=](get_atom) {
      return self->state.value;
    }
  };
}
```

The first part of the example illustrates how event-based actors can use either `then` or `await`.

```
void waiting_testee(event_based_actor* self, vector<cell> cells) {
  for (auto& x : cells)
    self->request(x, seconds(1), get_atom::value).await([=](int y) {
      aout(self) << "cell #" << x.id() << " -> " << y << endl;
    });
}

void multiplexed_testee(event_based_actor* self, vector<cell> cells) {
  for (auto& x : cells)
    self->request(x, seconds(1), get_atom::value).then([=](int y) {
      aout(self) << "cell #" << x.id() << " -> " << y << endl;
    });
}
```

The second half of the example shows a blocking actor making use of `receive`. Note that blocking actors have no special-purpose handler for error messages and therefore are required to pass a callback for error messages when handling response messages.

```
void blocking_testee(blocking_actor* self, vector<cell> cells) {
  for (auto& x : cells)
    self->request(x, seconds(1), get_atom::value).receive(
      [&](int y) {
        aout(self) << "cell #" << x.id() << " -> " << y << endl;
      },
      [&](error& err) {
        aout(self) << "cell #" << x.id()
                   << " -> " << self->system().render(err) << endl;
      }
    );
}
```

We spawn five cells and assign the values 0, 1, 4, 9, and 16.

```
  vector<cell> cells;
  for (auto i = 0; i < 5; ++i)
    cells.emplace_back(system.spawn(cell_impl, i * i));
```

When passing the `cells` vector to our three different implementations, we observe three outputs. Our `waiting_testee` actor will always print:

```
cell #9 -> 16
cell #8 -> 9
cell #7 -> 4
cell #6 -> 1
cell #5 -> 0
```

This is because `await` puts the one-shots handlers onto a stack and enforces LIFO order by re-ordering incoming response messages.

The `multiplexed_testee` implementation does not print its results in a predicable order. Response messages arrive in arbitrary order and are handled immediately.

Finally, the `blocking_testee` implementation will always print:

```
cell #5 -> 0
cell #6 -> 1
cell #7 -> 4
cell #8 -> 9
cell #9 -> 16
```

Both event-based approaches send all requests, install a series of one-shot handlers, and then return from the implementing function. In contrast, the blocking function waits for a response before sending another request.

6.5.2 Sending Multiple Requests

Sending the same message to a group of workers is a common work flow in actor applications. Usually, a manager maintains a set of workers. On request, the manager fans-out the request to all

of its workers and then collects the results. The function `fan_out_request` combined with the merge policy `fan_in_responses` streamlines this exact use case.

In the following snippet, we have a matrix actor (`self`) that stores worker actors for each cell (each simply storing an integer). For computing the average over a row, we use `fan_out_request`. The result handler passed to `then` now gets called only once with a `vector` holding all collected results. Using a response promise (see § 6.8) further allows us to delay responding to the client until we have collected all worker results.

```
1     [=](get_atom get, average_atom, row_atom, int row) {
2       assert(row < rows);
3       auto rp = self->make_response_promise<double>();
4       auto& row_vec = self->state.rows[row];
5       self->fan_out_request<policy::fan_in_responses>(row_vec, infinite, get)
6         .then(
7           [=](std::vector<int> xs) mutable {
8             assert(xs.size() == static_cast<size_t>(columns));
9             rp.deliver(std::accumulate(xs.begin(), xs.end(), 0.0) / columns);
10          },
11          [=](error& err) mutable { rp.deliver(std::move(err)); });
12      return rp;
13    },
```

6.5.3 Error Handling in Requests

Requests allow CAF to unambiguously correlate request and response messages. This is also true if the response is an error message. Hence, CAF allows to add an error handler as optional second parameter to `then` and `await` (this parameter is mandatory for `receive`). If no such handler is defined, the default error handler (see § 6.4.3) is used as a fallback in scheduled actors.

As an example, we consider a simple divider that returns an error on a division by zero. This examples uses a custom error category (see § 10).

```
enum class math_error : uint8_t {
  division_by_zero = 1
};

error make_error(math_error x) {
  return {static_cast<uint8_t>(x), atom("math")};
}

using div_atom = atom_constant<atom("div")>;

using divider = typed_actor<replies_to<div_atom, double, double>::with<double>>;

divider::behavior_type divider_impl() {
  return {
    [](div_atom, double x, double y) -> result<double> {
      if (y == 0.0)
        return math_error::division_by_zero;
      return x / y;
    }
  };
}
```

When sending requests to the divider, we use a custom error handlers to report errors to the user.

```
  scoped_actor self{system};
  self->request(div, std::chrono::seconds(10), div_atom::value, x, y).receive(
    [&](double z) {
      aout(self) << x << " / " << y << " = " << z << endl;
    },
    [&](const error& err) {
      aout(self) << "*** cannot compute " << x << " / " << y << " => "
                 << system.render(err) << endl;
    }
  );
```

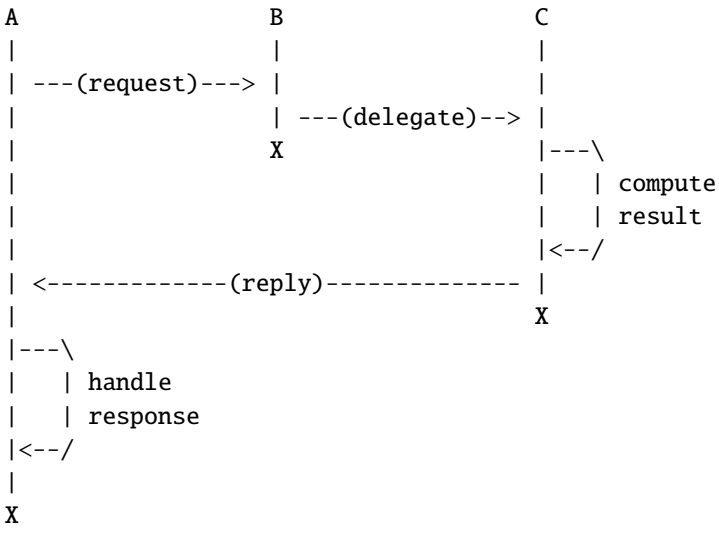
6.6 Delaying Messages

Messages can be delayed by using the function `delayed_send`, as illustrated in the following time-based loop example.

```
// uses a message-based loop to iterate over all animation steps
void dancing_kirby(event_based_actor* self) {
  // let's get it started
  self->send(self, step_atom::value, size_t{0});
  self->become (
    [=](step_atom, size_t step) {
      if (step == sizeof(animation_step)) {
        // we've printed all animation steps (done)
        cout << endl;
        self->quit();
        return;
      }
      // print given step
      draw_kirby(animation_steps[step]);
      // animate next step in 150ms
      self->delayed_send(self, std::chrono::milliseconds(150),
                         step_atom::value, step + 1);
    }
  );
}
```

6.7 Delegating Messages

Actors can transfer responsibility for a request by using `delegate`. This enables the receiver of the delegated message to reply as usual—simply by returning a value from its message handler—and the original sender of the message will receive the response. The following diagram illustrates request delegation from actor B to actor C.



Returning the result of `delegate(...)` from a message handler, as shown in the example below, suppresses the implicit response message and allows the compiler to check the result type when using statically typed actors.

```
void actor_a(event_based_actor* self, const calc& worker) {
  self->request(worker, std::chrono::seconds(10), add_atom::value, 1, 2).then(
    [=](int result) {
      aout(self) << "1 + 2 = " << result << endl;
    }
  );
}

calc::behavior_type actor_b(calc::pointer self, const calc& worker) {
  return {
    [=](add_atom add, int x, int y) {
      return self->delegate(worker, add, x, y);
    }
  };
}

calc::behavior_type actor_c() {
  return {
    [](add_atom, int x, int y) {
      return x + y;
    }
  };
}

void caf_main(actor_system& system) {
  system.spawn(actor_a, system.spawn(actor_b, system.spawn(actor_c)));
}
```

6.8 Response Promises

Response promises allow an actor to send and receive other messages prior to replying to a particular request. Actors create a response promise using `self->make_response_promise<Ts...>()`, where `Ts` is a template parameter pack describing the promised return type. Dynamically typed actors simply call `self->make_response_promise()`. After retrieving a promise, an actor can fulfill it by calling the member function `deliver(...)`, as shown in the following example.

```
1  // using add_atom = atom_constant<atom("add")>; (defined in atom.hpp)
2
3  using adder = typed_actor<replies_to<add_atom, int, int>::with<int>>;
4
5  // function-based, statically typed, event-based API
6  adder::behavior_type worker() {
7    return {
8      [](add_atom, int a, int b) {
9        return a + b;
10     }
11   };
12 }
13
14 // function-based, statically typed, event-based API
15 adder::behavior_type calculator_master(adder::pointer self) {
16   auto w = self->spawn(worker);
17   return {
18     [=](add_atom x, int y, int z) -> result<int> {
19       auto rp = self->make_response_promise<int>();
20       self->request(w, infinite, x, y, z).then([=](int result) mutable {
21         rp.deliver(result);
22       });
23       return rp;
24     }
25   };
26 }
```

6.9 Message Priorities

By default, all messages have the default priority, i.e., `message_priority::normal`. Actors can send urgent messages by setting the priority explicitly: `send<message_priority::high>(dst,...)`. Urgent messages are put into a different queue of the receiver's mailbox. Hence, long wait delays can be avoided for urgent communication.

7 Scheduler

The CAF runtime maps N actors to M threads on the local machine. Applications build with CAF scale by decomposing tasks into many independent steps that are spawned as actors. In this way, sequential computations performed by individual actors are small compared to the total runtime of the application, and the attainable speedup on multi-core hardware is maximized in agreement with Amdahl's law.

Decomposing tasks implies that actors are often short-lived. Assigning a dedicated thread to each actor would not scale well. Instead, CAF includes a scheduler that dynamically assigns actors to a pre-dimensioned set of worker threads. Actors are modeled as lightweight state machines. Whenever a *waiting* actor receives a message, it changes its state to *ready* and is scheduled for execution. CAF cannot interrupt running actors because it is implemented in user space. Consequently, actors that use blocking system calls such as I/O functions can suspend threads and create an imbalance or lead to starvation. Such "uncooperative" actors can be explicitly detached by the programmer by using the `detached` spawn option, e.g., `system.spawn<detached>(my_actor_fun)`.

The performance of actor-based applications depends on the scheduling algorithm in use and its configuration. Different application scenarios require different trade-offs. For example, interactive applications such as shells or GUIs want to stay responsive to user input at all times, while batch processing applications demand only to perform a given task in the shortest possible time.

Aside from managing actors, the scheduler bridges actor and non-actor code. For this reason, the scheduler distinguishes between external and internal events. An external event occurs whenever an actor is spawned from a non-actor context or an actor receives a message from a thread that is not under the control of the scheduler. Internal events are send and spawn operations from scheduled actors.

7.1 Policies

The scheduler consists of a single coordinator and a set of workers. The coordinator is needed by the public API to bridge actor and non-actor contexts, but is not necessarily an active software entity.

The scheduler of CAF is fully customizable by using a policy-based design. The following class shows a *concept* class that lists all required member types and member functions. A policy provides the two data structures `coordinator_data` and `worker_data` that add additional data members to the coordinator and its workers respectively, e.g., work queues. This grants developers full control over the state of the scheduler.

```
1  struct scheduler_policy {
2    struct coordinator_data;
3    struct worker_data;
4    void central_enqueue(Coordinator* self, resumable* job);
5    void external_enqueue(Worker* self, resumable* job);
6    void internal_enqueue(Worker* self, resumable* job);
7    void resume_job_later(Worker* self, resumable* job);
8    resumable* dequeue(Worker* self);
9    void before_resume(Worker* self, resumable* job);
10   void after_resume(Worker* self, resumable* job);
11   void after_completion(Worker* self, resumable* job);
12 };
```

Whenever a new work item is scheduled—usually by sending a message to an idle actor—, one of the functions `central_enqueue`, `external_enqueue`, and `internal_enqueue` is called. The first function is called whenever non-actor code interacts with the actor system. For example when spawning an actor from `main`. Its first argument is a pointer to the coordinator singleton and the second argument is the new work item—usually an actor that became ready. The function `external_enqueue` is never called directly by CAF. It models the transfer of a task to a worker by the coordinator or another worker. Its first argument is the worker receiving the new task referenced in the second argument. The third function, `internal_enqueue`, is called whenever an actor interacts with other actors in the system. Its first argument is the current worker and the second argument is the new work item.

Actors reaching the maximum number of messages per run are re-scheduled with `resume_job_later` and workers acquire new work by calling `dequeue`. The two functions `before_resume` and `after_resume` allow programmers to measure individual actor runtime, while `after_completion` allows to execute custom code whenever a work item has finished execution by changing its state to *done*, but before it is destroyed. In this way, the last three functions enable developers to gain fine-grained insight into the scheduling order and individual execution times.

7.2 Work Stealing

The default policy in CAF is work stealing. The key idea of this algorithm is to remove the bottleneck of a single, global work queue. The original algorithm was developed for fully strict computations by Blumofe et al in 1994. It schedules any number of tasks to `P` workers, where `P` is the number of processors available.

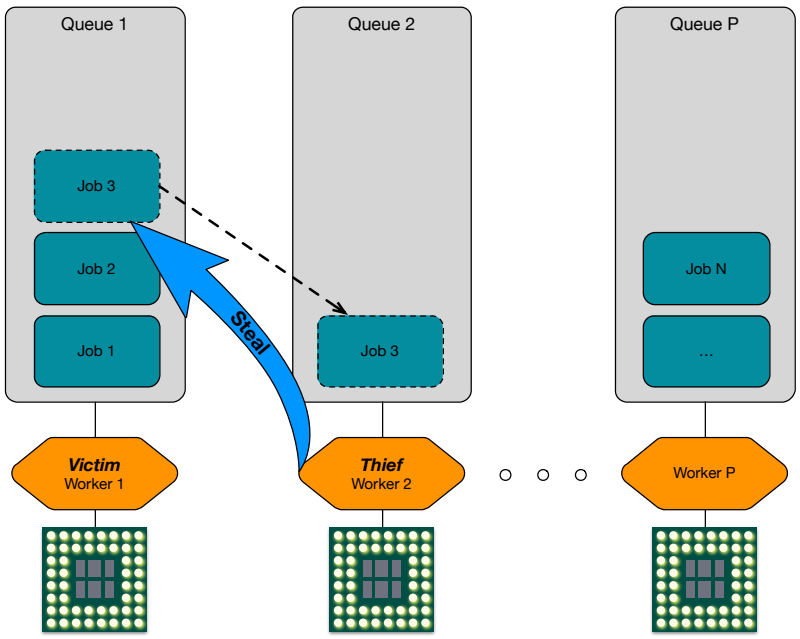


Figure 3: Stealing of work items

Each worker dequeues work items from an individual queue until it is drained. Once this happens, the worker becomes a *thief*. It picks one of the other workers—usually at random—as a *victim* and tries to *steal* a work item. As a consequence, tasks (actors) are bound to workers by default and only migrate between threads as a result of stealing. This strategy minimizes communication between threads and maximizes cache locality. Work stealing has become the algorithm of choice

for many frameworks. For example, Java's Fork-Join (which is used by Akka), Intel's Threading Building Blocks, several OpenMP implementations, etc.

CAF uses a double-ended queue for its workers, which is synchronized with two spinlocks. One downside of a decentralized algorithm such as work stealing is, that idle states are hard to detect. Did only one worker run out of work items or all? Since each worker has only local knowledge, it cannot decide when it could safely suspend itself. Likewise, workers cannot resume if new job items arrived at one or more workers. For this reason, CAF uses three polling intervals. Once a worker runs out of work items, it tries to steal items from others. First, it uses the *aggressive* polling interval. It falls back to a *moderate* interval after a predefined number of trials. After another predefined number of trials, it will finally use a *relaxed* interval.

Per default, the *aggressive* strategy performs 100 steal attempts with no sleep interval in between. The *moderate* strategy tries to steal 500 times with 50 microseconds sleep between two steal attempts. Finally, the *relaxed* strategy runs indefinitely but sleeps for 10 milliseconds between two attempts. These defaults can be overridden via system config at startup (see § 11).

7.3 Work Sharing

Work sharing is an alternative scheduler policy in CAF that uses a single, global work queue. This policy uses a mutex and a condition variable on the central queue. Thus, the policy supports only limited concurrency but does not need to poll. Using this policy can be a good fit for low-end devices where power consumption is an important metric.

8 Registry

The actor registry in CAF keeps track of the number of running actors and allows to map actors to their ID or a custom atom (see § 4.2) representing a name. The registry does *not* contain all actors. Actors have to be stored in the registry explicitly. Users can access the registry through an actor system by calling `system.registry()`. The registry stores actors using `strong_actor_ptr` (see § 7).

Users can use the registry to make actors system-wide available by name. The middleman (see § 17) uses the registry to keep track of all actors known to remote nodes in order to serialize and deserialize them. Actors are removed automatically when they terminate.

It is worth mentioning that the registry is not synchronized between connected actor system. Each actor system has its own, local registry in a distributed setting.

Types

`name_map`	`unordered_map<atom_value, strong_actor_ptr>`

Observers

`strong_actor_ptr get(actor_id)`	Returns the actor associated to given ID.
`strong_actor_ptr get(atom_value)`	Returns the actor associated to given name.
`name_map named_actors()`	Returns all name mappings.
`size_t running()`	Returns the number of currently running actors.

Modifiers

`void put(actor_id, strong_actor_ptr)`	Maps an actor to its ID.
`void erase(actor_id)`	Removes an ID mapping from the registry.
`void put(atom_value, strong_actor_ptr)`	Maps an actor to a name.
`void erase(atom_value)`	Removes a name mapping from the registry.

9 Reference Counting

Actors systems can span complex communication graphs that make it hard to decide when actors are no longer needed. As a result, manually managing lifetime of actors is merely impossible. For this reason, CAF implements a garbage collection strategy for actors based on weak and strong reference counts.

9.1 Shared Ownership in C++

The C++ standard library already offers `shared_ptr` and `weak_ptr` to manage objects with complex shared ownership. The standard implementation is a solid general purpose design that covers most use cases. Weak and strong references to an object are stored in a *control block*. However, CAF uses a slightly different design. The reason for this is twofold. First, we need the control block to store the identity of an actor. Second, we wanted a design that requires less indirections, because actor handles are used extensively copied for messaging, and this overhead adds up.

Before discussing the approach to shared ownership in CAF, we look at the design of shared pointers in the C++ standard library.

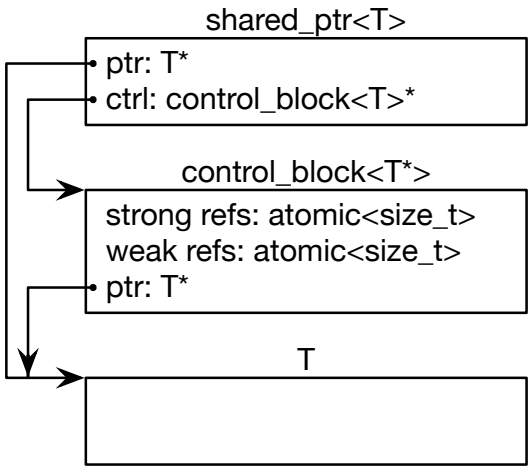


Figure 4: Shared pointer design in the C++ standard library

The figure above depicts the default memory layout when using shared pointers. The control block is allocated separately from the data and thus stores a pointer to the data. This is when using manually-allocated objects, for example `shared_ptr<int> iptr{new int}`. The benefit of this design is that one can destroy `T` independently from its control block. While irrelevant for small objects, it can become an issue for large objects. Notably, the shared pointer stores two pointers internally. Otherwise, dereferencing it would require to get the data location from the control block first.

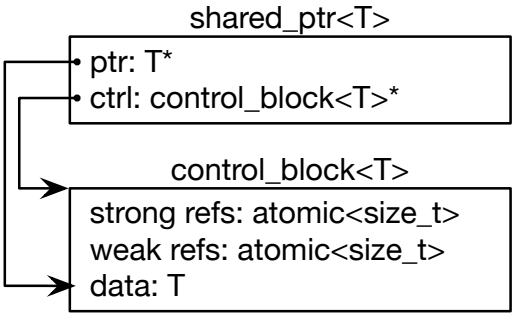


Figure 5: Memory layout when using `std::make_shared`

When using `make_shared` or `allocate_shared`, the standard library can store reference count and data in a single memory block as shown above. However, `shared_ptr` still has to store two pointers, because it is unaware where the data is allocated.

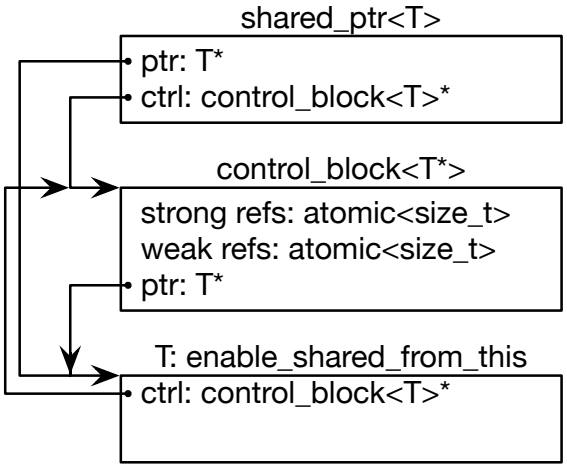


Figure 6: Memory layout with `std::enable_shared_from_this`

Finally, the design of the standard library becomes convoluted when an object should be able to hand out a `shared_ptr` to itself. Classes must inherit from `std::enable_shared_from_this` to navigate from an object to its control block. This additional navigation path is required, because `std::shared_ptr` needs two pointers. One to the data and one to the control block. Programmers can still use `make_shared` for such objects, in which case the object is again stored along with the control block.

9.2 Smart Pointers to Actors

In CAF, we use a different approach than the standard library because (1) we always allocate actors along with their control block, (2) we need additional information in the control block, and (3) we can store only a single raw pointer internally instead of the two raw pointers `std::shared_ptr` needs. The following figure summarizes the design of smart pointers to actors.

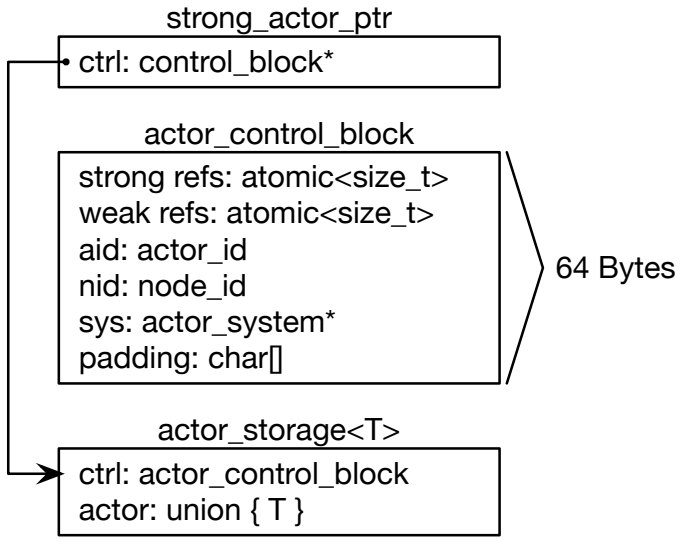


Figure 7: Shared pointer design in CAF

CAF uses `strong_actor_ptr` instead of `std::shared_ptr<...>` and `weak_actor_ptr` instead of `std::weak_ptr<...>`. Unlike the counterparts from the standard library, both smart pointer types only store a single pointer.

Also, the control block in CAF is not a template and stores the identity of an actor (`actor_id` plus `node_id`). This allows CAF to access this information even after an actor died. The control block fits exactly into a single cache line (64 Bytes). This makes sure no *false sharing* occurs between an actor and other actors that have references to it. Since the size of the control block is fixed and CAF *guarantees* the memory layout enforced by `actor_storage`, CAF can compute the address of an actor from the pointer to its control block by offsetting it by 64 Bytes. Likewise, an actor can compute the address of its control block.

The smart pointer design in CAF relies on a few assumptions about actor types. Most notably, the actor object is placed 64 Bytes after the control block. This starting address is cast to `abstract_actor*`. Hence, `T*` must be convertible to `abstract_actor*` via `reinterpret_cast`. In practice, this means actor subclasses must not use virtual inheritance, which is enforced in CAF with a `static_assert`.

9.3 Strong and Weak References

A *strong* reference manipulates the `strong refs` counter as shown above. An actor is destroyed if there are *zero* strong references to it. If two actors keep strong references to each other via member variable, neither actor can ever be destroyed because they produce a cycle (see § 9.5). Strong references are formed by `strong_actor_ptr`, `actor`, and `typed_actor<...>` (see § 1.2.3).

A *weak* reference manipulates the `weak refs` counter. This counter keeps track of how many references to the control block exist. The control block is destroyed if there are *zero* weak references to an actor (which cannot occur before `strong refs` reached *zero* as well). No cycle occurs if two actors keep weak references to each other, because the actor objects themselves can get destroyed independently from their control block. A weak reference is only formed by `actor_addr` (see § 1.2.3).

9.4 Converting Actor References with `actor_cast`

The function `actor_cast` converts between actor pointers and handles. The first common use case is to convert a `strong_actor_ptr` to either `actor` or `typed_actor<...>` before being able to send messages to an actor. The second common use case is to convert `actor_addr` to `strong_actor_ptr` to upgrade a weak reference to a strong reference. Note that casting `actor_addr` to a strong actor pointer or handle can result in invalid handles. The syntax for `actor_cast` resembles builtin C++ casts. For example, `actor_cast<actor>(x)` converts `x` to an handle of type `actor`.

9.5 Breaking Cycles Manually

Cycles can occur only when using class-based actors when storing references to other actors via member variable. Stateful actors (see § 5.7) break cycles by destroying the state when an actor terminates, *before* the destructor of the actor itself runs. This means an actor releases all references to others automatically after calling `quit`. However, class-based actors have to break cycles manually, because references to others are not released until the destructor of an actor runs. Two actors storing references to each other via member variable produce a cycle and neither destructor can ever be called.

Class-based actors can break cycles manually by overriding `on_exit()` and calling `destroy(x)` on each handle (see § 1.2.3). Using a handle after destroying it is undefined behavior, but it is safe to assign a new value to the handle.

10 Errors

Errors in CAF have a code and a category, similar to `std::error_code` and `std::error_condition`. Unlike its counterparts from the C++ standard library, `error` is plattform-neutral and serializable. Instead of using category singletons, CAF stores categories as atoms (see § 4.2). Errors can also include a message to provide additional context information.

10.1 Class Interface

Constructors	
`(Enum x)`	Construct error by calling `make_error(x)`
`(uint8_t x, atom_value y)`	Construct error with code `x` and category `y`
`(uint8_t x, atom_value y, message z)`	Construct error with code `x`, category `y`, and context `z`

Observers	
`uint8_t code()`	Returns the error code
`atom_value category()`	Returns the error category
`message context()`	Returns additional context information
`explicit operator bool()`	Returns `code() != 0`

10.2 Add Custom Error Categories

Adding custom error categories requires three steps: (1) declare an enum class of type `uint8_t` with the first value starting at 1, (2) implement a free function `make_error` that converts the enum to an `error` object, (3) add the custom category to the actor system with a render function. The last step is optional to allow users to retrieve a better string representation from `system.render(x)` than `to_string(x)` can offer. Note that any error code with value 0 is interpreted as *not-an-error*. The following example adds a custom error category by performing the first two steps.

```
enum class math_error : uint8_t {
  division_by_zero = 1
};

error make_error(math_error x) {
  return {static_cast<uint8_t>(x), atom("math")};
}

std::string to_string(math_error x) {
  switch (x) {
    case math_error::division_by_zero:
      return "division_by_zero";
    default:
      return "-unknown-error-";
  }
}
```

The implementation of `to_string(error)` is unable to call string conversions for custom error categories. Hence, `to_string(make_error(math_error::division_by_zero))` returns `"error(1, math)"`.

The following code adds a rendering function to the actor system to provide a more satisfactory

string conversion.

```
class config : public actor_system_config {
public:
  config() {
    auto renderer = [](uint8_t x, atom_value, const message&) {
      return "math_error" + deep_to_string_as_tuple(static_cast<math_error>(x));
    };
    add_error_category(atom("math"), renderer);
  }
};
```

With the custom rendering function, `system.render(make_error(math_error::division_by_zero))` returns `"math_error(division_by_zero)"`.

10.3 System Error Codes

System Error Codes (SECs) use the error category `"system"`. They represent errors in the actor system or one of its modules and are defined as follows.

```
/// ::actor_system and its modules.
enum class sec : uint8_t {
  /// No error.
  none = 0,
  /// Indicates that an actor dropped an unexpected message.
  unexpected_message = 1,
  /// Indicates that a response message did not match the provided handler.
  unexpected_response,
  /// Indicates that the receiver of a request is no longer alive.
  request_receiver_down,
  /// Indicates that a request message timed out.
  request_timeout,
  /// Indicates that requested group module does not exist.
  no_such_group_module = 5,
  /// Unpublishing or connecting failed: no actor bound to given port.
  no_actor_published_at_port,
  /// Connecting failed because a remote actor had an unexpected interface.
  unexpected_actor_messaging_interface,
  /// Migration failed because the state of an actor is not serializable.
  state_not_serializable,
  /// An actor received an unsupported key for `('sys', 'get', key)` messages.
  unsupported_sys_key,
  /// An actor received an unsupported system message.
  unsupported_sys_message = 10,
  /// A remote node disconnected during CAF handshake.
  disconnect_during_handshake,
  /// Tried to forward a message via BASP to an invalid actor handle.
  cannot_forward_to_invalid_actor,
  /// Tried to forward a message via BASP to an unknown node ID.
  no_route_to_receiving_node,
  /// Middleman could not assign a connection handle to a broker.
  failed_to_assign_scribe_from_handle,
  /// Middleman could not assign an acceptor handle to a broker.
  failed_to_assign_doorman_from_handle = 15,
  /// User requested to close port 0 or to close a port not managed by CAF.
  cannot_close_invalid_port,
  /// Middleman could not connect to a remote node.
  cannot_connect_to_node,
  /// Middleman could not open requested port.
  cannot_open_port,
  /// A C system call in the middleman failed.
  network_syscall_failed,
  /// A function received one or more invalid arguments.
  invalid_argument = 20,
  /// A network socket reported an invalid network protocol family.
  invalid_protocol_family,
  /// Middleman could not publish an actor because it was invalid.
  cannot_publish_invalid_actor,
  /// A remote spawn failed because the provided types did not match.
  cannot_spawn_actor_from_arguments,
  /// Serialization failed because there was not enough data to read.
  end_of_stream,
  /// Serialization failed because no CAF context is available.
```

```
54    no_context = 25,
55    /// Serialization failed because CAF misses run-time type information.
56    unknown_type,
57    /// Serialization of actors failed because no proxy registry is available.
58    no_proxy_registry,
59    /// An exception was thrown during message handling.
60    runtime_error,
61    /// Linking to a remote actor failed because actor no longer exists.
62    remote_linking_failed,
63    /// Adding an upstream to a stream failed.
64    cannot_add_upstream = 30,
65    /// Adding an upstream to a stream failed because it already exists.
66    upstream_already_exists,
67    /// Unable to process upstream messages because upstream is invalid.
68    invalid_upstream,
69    /// Adding a downstream to a stream failed.
70    cannot_add_downstream,
71    /// Adding a downstream to a stream failed because it already exists.
72    downstream_already_exists,
73    /// Unable to process downstream messages because downstream is invalid.
74    invalid_downstream = 35,
75    /// Cannot start streaming without next stage.
76    no_downstream_stages_defined,
77    /// Actor failed to initialize state after receiving a stream handshake.
78    stream_init_failed,
79    /// Unable to process a stream since due to missing state.
80    invalid_stream_state,
81    /// Stream aborted due to unexpected error.
82    unhandled_stream_error,
83    /// A function view was called without assigning an actor first.
84    bad_function_call = 40,
85    /// Feature is disabled in the actor system config.
86    feature_disabled,
```

10.4 Default Exit Reasons

CAF uses the error category `"exit"` for default exit reasons. These errors are usually fail states set by the actor system itself. The two exceptions are `exit_reason::user_shutdown` and `exit_reason::kill`. The former is used in CAF to signalize orderly, user-requested shutdown and can be used by programmers in the same way. The latter terminates an actor unconditionally when used in `send_exit`, even if the default handler for exit messages (see § 6.4.2) is overridden.

```
1  namespace caf {
2
3  /// This error category represents fail conditions for actors.
4  enum class exit_reason : uint8_t {
5    /// Indicates that an actor finished execution without error.
6    normal = 0,
7    /// Indicates that an actor died because of an unhandled exception.
8    unhandled_exception,
9    /// Indicates that the exit reason for this actor is unknown, i.e.,
10   /// the actor has been terminated and no longer exists.
11   unknown,
12   /// Indicates that an actor pool unexpectedly ran out of workers.
13   out_of_workers,
14   /// Indicates that an actor was forced to shutdown by a user-generated event.
```

```
15    user_shutdown,
16    /// Indicates that an actor was killed unconditionally.
17    kill,
18    /// Indicates that an actor finishied execution because a connection
19    /// to a remote link was closed unexpectedly.
20    remote_link_unreachable,
21    /// Indicates that an actor was killed because it became unreachable.
```

11 Configuring Actor Applications

CAF configures applications at startup using an `actor_system_config` or a user-defined subclass of that type. The config objects allow users to add custom types, to load modules, and to fine-tune the behavior of loaded modules with command line options or configuration files (see § 11.2).

The following code example is a minimal CAF application with a middleman (see § 17) but without any custom configuration options.

```
void caf_main(actor_system& system) {
  // ...
}
CAF_MAIN(io::middleman)
```

The compiler expands this example code to the following.

```
void caf_main(actor_system& system) {
  // ...
}
int main(int argc, char** argv) {
  return exec_main<io::middleman>(caf_main, argc, argv);
}
```

The function `exec_main` creates a config object, loads all modules requested in `CAF_MAIN` and then calls `caf_main`. A minimal implementation for `main` performing all these steps manually is shown in the next example for the sake of completeness.

```
int main(int argc, char** argv) {
  actor_system_config cfg;
  // read CLI options
  cfg.parse(argc, argv);
  // return immediately if a help text was printed
  if (cfg.cli_helptext_printed)
    return 0;
  // load modules
  cfg.load<io::middleman>();
  // create actor system and call caf_main
  actor_system system{cfg};
  caf_main(system);
}
```

However, setting up config objects by hand is usually not necessary. CAF automatically selects user-defined subclasses of `actor_system_config` if `caf_main` takes a second parameter by reference, as shown in the minimal example below.

```
class my_config : public actor_system_config {
public:
  my_config() {
    // ...
  }
};

void caf_main(actor_system& system, const my_config& cfg) {
  // ...
}

CAF_MAIN()
```

Users can perform additional initialization, add custom program options, etc. simply by implementing a default constructor.

11.1 Loading Modules

The simplest way to load modules is to use the macro `CAF_MAIN` and to pass a list of all requested modules, as shown below.

```
void caf_main(actor_system& system) {
  // ...
}
CAF_MAIN(mod1, mod2, ...)
```

Alternatively, users can load modules in user-defined config classes.

```
class my_config : public actor_system_config {
public:
  my_config() {
    load<mod1>();
    load<mod2>();
    // ...
  }
};
```

The third option is to simply call `x.load<mod1>()` on a config object *before* initializing an actor system with it.

11.2 Command Line Options and INI Configuration Files

CAF organizes program options in categories and parses CLI arguments as well as INI files. CLI arguments override values in the INI file which override hard-coded defaults. Users can add any number of custom program options by implementing a subtype of `actor_system_config`. The example below adds three options to the "global" category.

```
public:
  uint16_t port = 0;
  std::string host = "localhost";
  bool server_mode = false;

  config() {
    opt_group{custom_options_, "global"}
    .add(port, "port,p", "set port")
    .add(host, "host,H", "set host (ignored in server mode)")
    .add(server_mode, "server-mode,s", "enable server mode");
  }
};
```

We create a new "global" category in `custom_options_}`. Each following call to `add` then appends individual options to the category. The first argument to `add` is the associated variable. The second argument is the name for the parameter, optionally suffixed with a comma-separated single-character short name. The short name is only considered for CLI parsing and allows users to abbreviate commonly used option names. The third and final argument to `add` is a help text.

The custom `config` class allows end users to set the port for the application to 42 with either `--port=42` (long name) or `-p 42` (short name). The long option name is prefixed by the category when using a different category than "global". For example, adding the port option to the category "foo" means end users have to type `--foo.port=42` when using the long name. Short names are unaffected by the category, but have to be unique.

Boolean options do not require arguments. The member variable `server_mode` is set to `true` if the command line contains either `--server-mode` or `-s`.

The example uses member variables for capturing user-provided settings for simplicity. However, this is not required. For example, `add<bool>(...)` allows omitting the first argument entirely. All values of the configuration are accessible with `get_or`. Note that all global options can omit the `"global."` prefix.

CAF adds the program options "help" (with short names `-h` and `-?`) as well as "long-help" to the "global" category.

The default name for the INI file is `caf-application.ini`. Users can change the file name and path by passing `--config-file=<path>` on the command line.

INI files are organized in categories. No value is allowed outside of a category (no implicit "global" category). The parses uses the following syntax:

`key=true`	is a boolean
`key=1`	is an integer
`key=1.0`	is an floating point number
`key=1ms`	is an timespan
`key='foo'`	is an atom
`key="foo"`	is a string
`key=[0, 1, ...]`	is as a list
`key={a=1, b=2, ...}`	is a dictionary (map)

The following example INI file lists all standard options in CAF and their default value. Note that some options such as `scheduler.max-threads` are usually detected at runtime and thus have no hard-coded default.

```
; This file shows all possible parameters with defaults.
; Values enclosed in <> are detected at runtime unless defined by the user.

; when using the default scheduler
[scheduler]
; accepted alternative: 'sharing'
policy='stealing'
; configures whether the scheduler generates profiling output
enable-profiling=false
; forces a fixed number of threads if set
max-threads=<number of cores>
; maximum number of messages actors can consume in one run
max-throughput=<infinite>
; measurement resolution in milliseconds (only if profiling is enabled)
profiling-resolution=100ms
; output file for profiler data (only if profiling is enabled)
profiling-output-file="/dev/null"

; when using 'stealing' as scheduler policy
[work-stealing]
; number of zero-sleep-interval polling attempts
aggressive-poll-attempts=100
; frequency of steal attempts during aggressive polling
aggressive-steal-interval=10
; number of moderately aggressive polling attempts
moderate-poll-attempts=500
; frequency of steal attempts during moderate polling
moderate-steal-interval=5
; sleep interval between poll attempts
moderate-sleep-duration=50us
; frequency of steal attempts during relaxed polling
relaxed-steal-interval=1
; sleep interval between poll attempts
relaxed-sleep-duration=10ms

; when loading io::middleman
[middleman]
; configures whether MMs try to span a full mesh
enable-automatic-connections=false
; application identifier of this node, prevents connection to other CAF
; instances with different identifier
app-identifier=""
; maximum number of consecutive I/O reads per broker
max-consecutive-reads=50
; heartbeat message interval in ms (0 disables heartbeating)
heartbeat-interval=0ms
; configures whether the MM attaches its internal utility actors to the
; scheduler instead of dedicating individual threads (needed only for
; deterministic testing)
attach-utility-actors=false
; configures whether the MM starts a background thread for I/O activity,
; setting this to true allows fully deterministic execution in unit test and
; requires the user to trigger I/O manually
manual-multiplexing=false
; disables communication via TCP
disable-tcp=false
; enable communication via UDP
enable-udp=false
```

```
; configures how many background workers are spawned for deserialization,
; by default CAF uses 1-4 workers depending on the number of cores
workers=<min(3, number of cores / 4) + 1>

; when compiling with logging enabled
[logger]
; file name template for output log file files (empty string disables logging)
file-name="actor_log_[PID]_[TIMESTAMP]_[NODE].log"
; format for rendering individual log file entries
file-format="%r %c %p %a %t %C %M %F:%L %m%n"
; configures the minimum severity of messages that are written to the log file
; (quiet|error|warning|info|debug|trace)
file-verbosity='trace'
; mode for console log output generation (none|colored|uncolored)
console='none'
; format for printing individual log entries to the console
console-format="%m"
; configures the minimum severity of messages that are written to the console
; (quiet|error|warning|info|debug|trace)
console-verbosity='trace'
; excludes listed components from logging (list of atoms)
component-blacklist=[]
```

11.3 Adding Custom Message Types

CAF requires serialization support for all of its message types (see § 3). However, CAF also needs a mapping of unique type names to user-defined types at runtime. This is required to deserialize arbitrary messages from the network.

As an introductory example, we (again) use the following POD type `foo`.

```
struct foo {
  std::vector<int> a;
  int b;
};
```

To make `foo` serializable, we make it inspectable (see § 3):

```
template <class Inspector>
typename Inspector::result_type inspect(Inspector& f, foo& x) {
  return f(meta::type_name("foo"), x.a, x.b);
}
```

Finally, we give `foo` a platform-neutral name and add it to the list of serializable types by using a custom config class.

```
class config : public actor_system_config {
public:
  config() {
    add_message_type<foo>("foo");
  }
};

void caf_main(actor_system& system, const config&) {
```

11.4 Adding Custom Error Types

Adding a custom error type to the system is a convenience feature to allow improve the string representation. Error types can be added by implementing a render function and passing it to `add_error_category`, as shown in § 10.2.

11.5 Adding Custom Actor Types experimental

Adding actor types to the configuration allows users to spawn actors by their name. In particular, this enables spawning of actors on a different node (see § 19). For our example configuration, we consider the following simple `calculator` actor.

```
using add_atom = atom_constant<atom("add")>;
using sub_atom = atom_constant<atom("sub")>;

using calculator = typed_actor<replies_to<add_atom, int, int>::with<int>,
                               replies_to<sub_atom, int, int>::with<int>>;

calculator::behavior_type calculator_fun(calculator::pointer self) {
```

Adding the calculator actor type to our config is achieved by calling `add_actor_type<T>`. Note that adding an actor type in this way implicitly calls `add_message_type<T>` for typed actors (see § 11.3). This makes our `calculator` actor type serializable and also enables remote nodes to spawn calculators anywhere in the distributed actor system (assuming all nodes use the same config).

```
struct config : actor_system_config {
  config() {
    add_actor_type("calculator", calculator_fun);
  }
};
```

Our final example illustrates how to spawn a `calculator` locally by using its type name. Because the dynamic type name lookup can fail and the construction arguments passed as message can mismatch, this version of `spawn` returns `expected<T>`.

```
auto x = system.spawn<calculator>("calculator", make_message());
if (! x) {
  std::cerr << "*** unable to spawn calculator: "
            << system.render(x.error()) << std::endl;
  return;
}
calculator c = std::move(*x);
```

Adding dynamically typed actors to the config is achieved in the same way. When spawning a dynamically typed actor in this way, the template parameter is simply `actor`. For example, spawning an actor ”foo” which requires one string is created with:

```
auto worker = system.spawn<actor>("foo", make_message("bar"));
```

Because constructor (or function) arguments for spawning the actor are stored in a `message`, only actors with appropriate input types are allowed. For example, pointer types are illegal. Hence users need to replace C-strings with `std::string`.

11.6 Log Output

Logging is disabled in CAF per default. It can be enabled by setting the `--with-log-level=` option of the `configure` script to one of “error”, “warning”, “info”, “debug”, or “trace” (from least output to most). Alternatively, setting the CMake variable `CAF_LOG_LEVEL` to 0, 1, 2, 3, or 4 (from least output to most) has the same effect.

All logger-related configuration options listed here and in § 11.2 are silently ignored if logging is disabled.

11.6.1 File Name

The output file is generated from the template configured by `logger-file-name`. This template supports the following variables.

Variable	Output
`[PID]`	The OS-specific process ID.
`[TIMESTAMP]`	The UNIX timestamp on startup.
`[NODE]`	The node ID of the CAF system.

11.6.2 Console

Console output is disabled per default. Setting `logger-console` to either `"uncolored"` or `"colored"` prints log events to `std::clog`. Using the `"colored"` option will print the log events in different colors depending on the severity level.

11.6.3 Format Strings

CAF uses log4j-like format strings for configuring printing of individual events via `logger-file-format` and `logger-console-format`. Note that format modifiers are not supported at the moment. The recognized field identifiers are:

Character	Output
c	The category/component. This name is defined by the macro CAF_LOG_COMPONENT. Set this macro before including any CAF header.
C	The full qualifier of the current function. For example, the qualifier of void ns::foo::bar() is printed as ns.foo.
d	The date in ISO 8601 format, i.e., "YYYY-MM-DD hh:mm:ss".
F	The file name.
L	The line number.
m	The user-defined log message.
M	The name of the current function. For example, the name of void ns::foo::bar() is printed as bar.
n	A newline.
p	The priority (severity level).
r	Elapsed time since starting the application in milliseconds.
t	ID of the current thread.
a	ID of the current actor (or "actor0" when not logging inside an actor).
%	A single percent sign.

11.6.4 Filtering

The two configuration options logger-component-filter and logger-verbosity reduce the amount of generated log events. The former is a list of excluded component names and the latter can increase the reported severity level (but not decrease it beyond the level defined at compile time).

12 Type-Erased Tuples, Messages and Message Views

Messages in CAF are stored in type-erased 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 use cases than only sending data from one actor to another.

The interface `type_erased_tuple` encapsulates access to arbitrary data. This data can be stored on the heap or on the stack. A `message` is a type-erased tuple that is always heap-allocated and uses copy-on-write semantics. When dealing with ”plain” type-erased tuples, users are required to check if a tuple is referenced by others via `type_erased_tuple::shared` before modifying its content.

The convenience class `message_view` holds a reference to either a stack-located `type_erased_tuple` or a `message`. The content of the data can be access via `message_view::content` in both cases, which returns a `type_erased_tuple&`. The content of the view can be forced into a message object by calling `message_view::move_content_to_message`. This member function either returns the stored message object or moves the content of a stack-allocated tuple into a new message.

12.1 RTTI and Type Numbers

All builtin types in CAF have a non-zero 6-bit *type number*. All user-defined types are mapped to 0. When querying the run-time type information (RTTI) for individual message or tuple elements, CAF returns a pair consisting of an integer and a pointer to `std::type_info`. The first value is the 6-bit type number. If the type number is non-zero, the second value is a pointer to the C++ type info, otherwise the second value is null. Additionally, CAF generates 32 bit *type tokens*. These tokens are *type hints* that summarizes all types in a type-erased tuple. Two type-erased tuples are of different type if they have different type tokens (the reverse is not true).

12.2 Class `type_erased_tuple`

Note: Calling modifiers on a shared type-erased tuple is undefined behavior.

Observers	
`bool empty()`	Returns whether this message is empty.
`size_t size()`	Returns the size of this message.
`rtti_pair type(size_t pos)`	Returns run-time type information for the nth element.
`error save(serializer& x)`	Writes the tuple to `x`.
`error save(size_t n, serializer& x)`	Writes the nth element to `x`.
`const void* get(size_t n)`	Returns a const pointer to the nth element.
`std::string stringify()`	Returns a string representation of the tuple.
`std::string stringify(size_t n)`	Returns a string representation of the nth element.
`bool matches(size_t n, rtti_pair)`	Checks whether the nth element has given type.
`bool shared()`	Checks whether more than one reference to the data exists.
`bool match_element<T>(size_t n)`	Checks whether element `n` has type `T`.
`bool match_elements<Ts...>()`	Checks whether this message has the types `Ts...`.
`const T& get_as<T>(size_t n)`	Returns a const reference to the nth element.

Modifiers	
`void* get_mutable(size_t n)`	Returns a mutable pointer to the nth element.
`T& get_mutable_as<T>(size_t n)`	Returns a mutable reference to the nth element.
`void load(deserializer& x)`	Reads the tuple from `x`.

12.3 Class `message`

The class `message` includes all member functions of `type_erased_tuple`. However, calling modifiers is always guaranteed to be safe. A `message` automatically detaches its content by copying it from the shared data on mutable access. The class further adds the following member functions over `type_erased_tuple`. Note that `apply` only detaches the content if a callback takes mutable references as arguments.

Observers

`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 extract(message_handler)`	See § 12.5.
`message extract_opts(...)`	See § 12.6.

Modifiers

`optional<message> apply(message_handler f)`	Returns `f(*this)`.

Operators

`message operator+(message x, message y)`	Concatenates `x` and `y`.
`message& operator+=(message& x, message y)`	Concatenates `x` and `y`.

12.4 Class `message_builder`

Constructors

`(void)`	Creates an empty message builder.
`(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.
`append(T val)`	Adds `val` to the buffer.
`append(Iter first, Iter last)`	Adds all elements from range `[first, last)`.
`message extract(message_handler)`	See § 12.5.
`message extract_opts(...)`	See § 12.6.

Modifiers

`optional<message> apply(message_handler f)`	Returns `f(*this)`.
`message move_to_message()`	Transfers ownership of its data to the new message.

12.5 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.

12.6 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.error.empty()) {
    // read invalid CLI arguments
    cerr << res.error << endl;
    return 1;
  }
  if (res.opts.count("help") > 0) {
    // CLI arguments contained "-h", "--help", or "-?" (builtin);
    cout << res.helptext << endl;
    return 0;
  }
  if (! res.remainder.empty()) {
    // res.remainder stors all extra arguments that weren't consumed
  }
  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
*/
```

13 Group Communication

CAF supports publish/subscribe-based group communication. Dynamically typed actors can join and leave groups and send messages to groups. The following example showcases the basic API for retrieving a group from a module by its name, joining, and leaving.

```
std::string module = "local";
std::string id = "foo";
auto expected_grp = system.groups().get(module, id);
if (! expected_grp) {
  std::cerr << "*** cannot load group: "
            << system.render(expected_grp.error()) << std::endl;
  return;
}
auto grp = std::move(*expected_grp);
scoped_actor self{system};
self->join(grp);
self->send(grp, "test");
self->receive(
  [](const std::string& str) {
    assert(str == "test");
  }
);
self->leave(grp);
```

It is worth mentioning that the module `"local"` is guaranteed to never return an error. The example above uses the general API for retrieving the group. However, local modules can be easier accessed by calling `system.groups().get_local(id)`, which returns `group` instead of `expected<group>`.

13.1 Anonymous Groups

Groups created on-the-fly with `system.groups().anonymous()` can be used to coordinate a set of workers. Each call to this function 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 `system.groups().get_local("GUI events")`. The group ID `"GUI events"` uniquely identifies a singleton group instance of the module `"local"`.

13.3 Remote Groups

Calling`system.middleman().publish_local_groups(port, addr)` makes all local groups available to other nodes in the network. The first argument denotes the port, while the second (optional) parameter can be used to whitelist IP addresses.

After publishing the group at one node (the server), other nodes (the clients) can get a handle for that group by using the “remote” module: `system.groups().get("remote", "<group>@<host>:<port>")`. This implementation uses N-times unicast underneath and the group is only available as long as the hosting server is alive.

14 Managing Groups of Workers experimental

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. Unlike groups, pools usually own their workers.

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')`.

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 do not cache messages, 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 Dispatching Policies

A dispatching policy is a functor with the following signature:

```
using uplock = upgrade_lock<detail::shared_spinlock>;
using policy = std::function<void (actor_system& sys,
                                   uplock& guard,
                                   const actor_vec& workers,
                                   mailbox_element_ptr& ptr,
                                   execution_unit* host)>;
```

The argument `guard` is a shared lock that can be upgraded for unique access if the policy includes a critical section. The second argument is a vector containing all workers managed by the pool. The argument `ptr` contains the full message as received by the pool. Finally, `host` is the current scheduler context that can be used to enqueue workers into the corresponding job queue.

The actor pool class comes with a set predefined policies, accessible via factory functions, for convenience.

```
actor_pool::policy 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.

```
actor_pool::policy 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.

```
actor_pool::policy 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.

```
using join = function<void (T&, message&)>;
using split = function<void (vector<pair<actor, message>>&, message&)>;
template <class T>
static policy split_join(join jf, split sf = ..., T init = T());
```

This policy models split/join or scatter/gather work flows, where a work item is split into as many tasks as workers are available and then the individuals results are joined together before sending the full result back to the client.

The join function is responsible for “glueing” all result messages together to create a single result. The function is called with the result object (initialed using `init`) and the current result messages from a worker.

The first argument of a split function is a mapping from actors (workers) to tasks (messages). The second argument is the input message. The default split function is a broadcast dispatching, sending each worker the original request.

15 Streaming experimental

Streams in CAF describe data flow between actors. We are not aiming to provide functionality similar to Apache projects like Spark, Flink or Storm. Likewise, we have different goals than APIs such as RxJava, Reactive Streams, etc. Streams complement asynchronous messages, request/response communication and publish/subscribe in CAF. In a sense, actor streams in CAF are a building block that users could leverage for building feature-complete stream computation engines or reactive high-level Big Data APIs.

A stream establishes a logical channel between two or more actors for exchanging a potentially unbound sequence of values. This channel uses demand signaling to guarantee that senders cannot overload receivers.

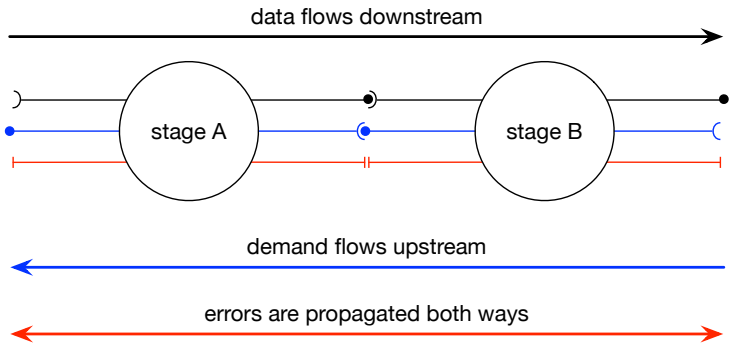


Figure 8: Streaming Concept

Streams are directed and data flows only *downstream*, i.e., from sender (source) to receiver (sink). Establishing a stream requires a handshake in order to initialize required state and signal initial demand.

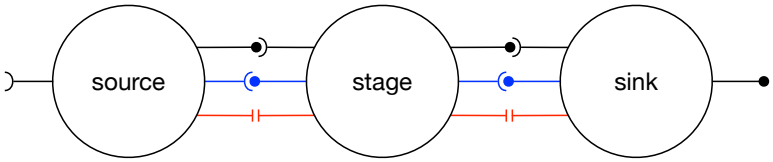


Figure 9: Streaming Roles

CAF distinguishes between three roles in a stream: (1) a *source* creates streams and generates data, (2) a *stage* transforms or filters data, and (3) a *sink* terminates streams by consuming data.

We usually draw streams as pipelines for simplicity. However, sources can have any number of outputs (downstream actors). Likewise, sinks can have any number of inputs (upstream actors) and stages can multiplex N inputs to M outputs. Hence, streaming topologies in CAF support arbitrary complexity with forks and joins.

15.1 Stream Managers

Streaming-related messages are handled separately. Under the hood, actors delegate to *stream managers* that in turn allow customization of their behavior with *drivers* and *downstream managers*.

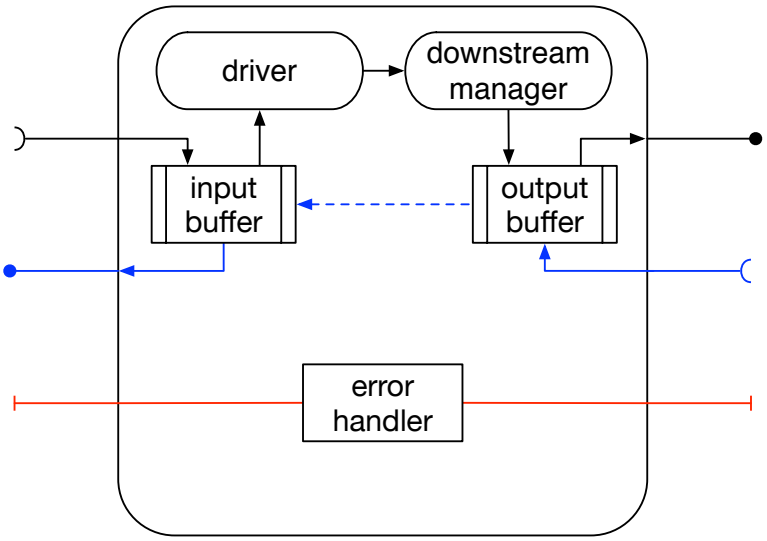


Figure 10: Internals of Stream Managers

Users usually can skip implementing driver classes and instead use the lambda-based interface showcased in the following sections. Drivers implement the streaming logic by taking inputs from upstream actors and pushing data to the downstream manager. A source has no input buffer. Hence, drivers only provide a *generator* function that downstream managers call according to demand.

A downstream manager is responsible for dispatching data to downstream actors. The default implementation broadcasts data, i.e., all downstream actors receive the same data. The downstream manager can also perform any sort multi- or anycast. For example, a load-balancer would use an anycast policy to dispatch data to the next available worker.

15.2 Defining Sources

```
// Simple source for generating a stream of integers from [0, n).
behavior int_source(event_based_actor* self) {
  return {[=](open_atom, int n) {
    // Produce at least one value.
    if (n <= 0)
      n = 1;
    // Create a stream manager for implementing a stream source. The
    // streaming logic requires three functions: initializer, generator, and
    // predicate.
    return attach_stream_source(
      self,
      // Initializer. The type of the first argument (state) is freely
      // chosen. If no state is required, `caf::unit_t` can be used here.
      [](int& x) { x = 0; },
      // Generator. This function is called by CAF to produce new stream
      // elements for downstream actors. The `x` argument is our state again
      // (with our freely chosen type). The second argument `out` points to
      // the output buffer. The template argument (here: int) determines what
      // elements downstream actors receive in this stream. Finally, `num` is
      // a hint from CAF how many elements we should ideally insert into
      // `out`. We can always insert fewer or more items.
      [n](int& x, downstream<int>& out, size_t num) {
        auto max_x = std::min(x + static_cast<int>(num), n);
        for (; x < max_x; ++x)
          out.push(x);
      },
      // Predicate. This function tells CAF when we reached the end.
      [n](const int& x) { return x == n; });
  }};
}
```

The simplest way to defining a source is to use the `attach_stream_source` function and pass it four arguments: a pointer to *self*, *initializer* for the state, *generator* for producing values, and *predicate* for signaling the end of the stream.

15.3 Defining Stages

```
// Simple stage that only selects even numbers.
behavior int_selector(event_based_actor* self) {
  return {[=](stream<int> in) {
    // Create a stream manager for implementing a stream stage. Similar to
    // `make_source`, we need three functions: initialzer, processor, and
    // finalizer.
    return attach_stream_stage(
      self,
      // Our input source.
      in,
      // Initializer. Here, we don't need any state and simply use unit_t.
      [](unit_t&) {
        // nop
      },
      // Processor. This function takes individual input elements as `val`
      // and forwards even integers to `out`.
      [](unit_t&, downstream<int>& out, int val) {
        if (val % 2 == 0)
          out.push(val);
      },
      // Finalizer. Allows us to run cleanup code once the stream terminates.
      [=](unit_t&, const error& err) {
        if (err) {
          aout(self) << "int_selector aborted with error: " << err << std::endl;
        } else {
          aout(self) << "int_selector finalized" << std::endl;
        }
        // else: regular stream shutdown
      });
  }};
}
```

The function `make_stage` also takes three lambdas but additionally the received input stream handshake as first argument. Instead of a predicate, `make_stage` only takes a finalizer, since the stage does not produce data on its own and a stream terminates if no more sources exist.

15.4 Defining Sinks

```
behavior int_sink(event_based_actor* self) {
  return {[=](stream<int> in) {
    // Create a stream manager for implementing a stream sink. Once more, we
    // have to provide three functions: Initializer, Consumer, Finalizer.
    return attach_stream_sink(
      self,
      // Our input source.
      in,
      // Initializer. Here, we store all values we receive. Note that streams
      // are potentially unbound, so this is usually a bad idea outside small
      // examples like this one.
      [](std::vector<int>&) {
        // nop
      },
      // Consumer. Takes individual input elements as `val` and stores them
      // in our history.
      [](std::vector<int>& xs, int val) { xs.emplace_back(val); },
      // Finalizer. Allows us to run cleanup code once the stream terminates.
      [=](std::vector<int>& xs, const error& err) {
        if (err) {
          aout(self) << "int_sink aborted with error: " << err << std::endl;
        } else {
          aout(self) << "int_sink finalized after receiving: " << xs
                     << std::endl;
        }
      });
  }};
```

The function `make_sink` is similar to `make_stage`, except that is does not produce outputs.

15.5 Initiating Streams

```
auto src = sys.spawn(int_source);
auto snk = sys.spawn(int_sink);
auto pipeline = cfg.with_stage ? snk * sys.spawn(int_selector) * src
                               : snk * src;
anon_send(pipeline, open_atom::value, cfg.n);
```

In our example, we always have a source `int_source` and a sink `int_sink` with an optional stage `int_selector`. Sending `open_atom` to the source initiates the stream and the source will respond with a stream handshake.

Using the actor composition in CAF (`snk * src` reads *sink after source*) allows us to redirect the stream handshake we send in `caf_main` to the sink (or to the stage and then from the stage to the sink).

16 Testing

CAF comes with built-in support for writing unit tests in a domain-specific language (DSL). The API looks similar to well-known testing frameworks such as Boost.Test and Catch but adds CAF-specific macros for testing messaging between actors.

Our design leverages four main concepts:

- **Checks** represent single boolean expressions.
- **Tests** contain one or more checks.
- **Fixtures** equip tests with a fixed data environment.
- **Suites** group tests together.

The following code illustrates a very basic test case that captures the four main concepts described above.

```
1  // Adds all tests in this compilation unit to the suite "math".
2  #define CAF_SUITE math
3
4  // Pulls in all the necessary macros.
5  #include "caf/test/dsl.hpp"
6
7  namespace {
8
9  struct fixture {};
10
11 } // namespace
12
13 // Makes all members of `fixture` available to tests in the scope.
14 CAF_TEST_FIXTURE_SCOPE(math_tests, fixture)
15
16 // Implements our first test.
17 CAF_TEST(divide) {
18   CAF_CHECK(1 / 1 == 0); // fails
19   CAF_CHECK(2 / 2 == 1); // passes
20   CAF_REQUIRE(3 + 3 == 5); // fails and aborts test execution
21   CAF_CHECK(4 - 4 == 0); // unreachable due to previous requirement error
22 }
23
24 CAF_TEST_FIXTURE_SCOPE_END()
```

The code above highlights the two basic macros `CAF_CHECK` and `CAF_REQUIRE`. The former reports failed checks, but allows the test to continue on error. The latter stops test execution if the boolean expression evaluates to false.

The third macro worth mentioning is `CAF_FAIL`. It unconditionally stops test execution with an error message. This is particularly useful for stopping program execution after receiving unexpected messages, as we will see later.

16.1 Testing Actors

The following example illustrates how to add an actor system as well as a scoped actor to fixtures. This allows spawning of and interacting with actors in a similar way regular programs would. Except that we are using macros such as `CAF_CHECK` and provide tests rather than implementing a `caf_main`.

```
namespace {

struct fixture {
  caf::actor_system_config cfg;
  caf::actor_system sys;
  caf::scoped_actor self;

  fixture() : sys(cfg), self(sys) {
    // nop
  }
};

caf::behavior adder() {
  return {
    [=](int x, int y) {
      return x + y;
    }
  };
}

} // namespace

CAF_TEST_FIXTURE_SCOPE(actor_tests, fixture)

CAF_TEST(simple actor test) {
  // Our Actor-Under-Test.
  auto aut = self->spawn(adder);
  self->request(aut, caf::infinite, 3, 4).receive(
    [=](int r) {
      CAF_CHECK(r == 7);
    },
    [&](caf::error& err) {
      // Must not happen, stop test.
      CAF_FAIL(sys.render(err));
    });
}

CAF_TEST_FIXTURE_SCOPE_END()
```

The example above works, but suffers from several issues:

- Significant amount of boilerplate code.
- Using a scoped actor as illustrated above can only test one actor at a time. However, messages between other actors are invisible to us.
- CAF runs actors in a thread pool by default. The resulting nondeterminism makes triggering reliable ordering of messages near impossible. Further, forcing timeouts to test error handling code is even harder.

16.2 Deterministic Testing

CAF provides a scheduler implementation specifically tailored for writing unit tests called `test_coordinator`. It does not start any threads and instead gives unit tests full control over message dispatching and timeout management.

To reduce boilerplate code, CAF also provides a fixture template called `test_coordinator_fixture` that comes with ready-to-use actor system (`sys`) and testing scheduler (`sched`). The optional template parameter allows unit tests to plugin custom actor system configuration classes.

Using this fixture unlocks three additional macros:

- `expect` checks for a single message. The macro verifies the content types of the message and invokes the necessary member functions on the test coordinator. Optionally, the macro checks the receiver of the message and its content. If the expected message does not exist, the test aborts.
- `allow` is similar to `expect`, but it does not abort the test if the expected message is missing. This macro returns `true` if the allowed message was delivered, `false` otherwise.
- `disallow` aborts the test if a particular message was delivered to an actor.

The following example implements two actors, `ping` and `pong`, that exchange a configurable amount of messages. The test *three pings* then checks the contents of each message with `expect` and verifies that no additional messages exist using `disallow`.

```
namespace {

using ping_atom = atom_constant<atom("ping")>;
using pong_atom = atom_constant<atom("pong")>;

behavior ping(event_based_actor* self, actor pong_actor, int n) {
  self->send(pong_actor, ping_atom::value, n);
  return {
    [=](pong_atom, int x) {
      if (x > 1)
        self->send(pong_actor, ping_atom::value, x - 1);
    }
  };
}

behavior pong() {
  return {
    [=](ping_atom, int x) {
      return std::make_tuple(pong_atom::value, x);
    }
  };
}

struct ping_pong_fixture : test_coordinator_fixture<> {
  actor pong_actor;

  ping_pong_fixture() {
    // Spawn the Pong actor.
    pong_actor = sys.spawn(pong);
```

```
30      // Run initialization code for Pong.
31      run();
32    }
33  };
34
35  } // namespace
36
37  CAF_TEST_FIXTURE_SCOPE(ping_pong_tests, ping_pong_fixture)
38
39  CAF_TEST(three pings) {
40    // Spawn the Ping actor and run its initialization code.
41    auto ping_actor = sys.spawn(ping, pong_actor, 3);
42    sched.run_once();
43    // Test communication between Ping and Pong.
44    expect((ping_atom, int), from(ping_actor).to(pong_actor).with(_, 3));
45    expect((pong_atom, int), from(pong_actor).to(ping_actor).with(_, 3));
46    expect((ping_atom, int), from(ping_actor).to(pong_actor).with(_, 2));
47    expect((pong_atom, int), from(pong_actor).to(ping_actor).with(_, 2));
48    expect((ping_atom, int), from(ping_actor).to(pong_actor).with(_, 1));
49    expect((pong_atom, int), from(pong_actor).to(ping_actor).with(_, 1));
50    // No further messages allowed.
51    disallow((ping_atom, int), from(ping_actor).to(pong_actor).with(_, 1));
52  }
53
54  CAF_TEST_FIXTURE_SCOPE_END()
```

Part II

I/O Library

17 Middleman

The middleman is the main component of the I/O module and enables distribution. It transparently manages proxy actor instances representing remote actors, maintains connections to other nodes, and takes care of serialization of messages. Applications install a middleman by loading `caf::io::middleman` as module (see § 11). Users can include `"caf/io/all.hpp"` to get access to all public classes of the I/O module.

17.1 Class `middleman`

Remoting	
`expected<uint16> open(uint16, const char*, bool)`	See § 17.2.
`expected<uint16> publish(T, uint16, const char*, bool)`	See § 17.2.
`expected<void> unpublish(T x, uint16)`	See § 17.2.
`expected<node_id> connect(std::string host, uint16_t port)`	See § 17.2.
`expected<T> remote_actor<T = actor>(string, uint16)`	See § 17.2.
`expected<T> spawn_broker(F fun, ...)`	See § 18.
`expected<T> spawn_client(F, string, uint16, ...)`	See § 18.
`expected<T> spawn_server(F, uint16, ...)`	See § 18.

17.2 Publishing and Connecting

The member function `publish` binds an actor to a given port, thereby allowing other nodes to access it over the network.

```
template <class T>
expected<uint16_t> middleman::publish(T x, uint16_t port,
                                      const char* in = nullptr,
                                      bool reuse_addr = false);
```

The first argument is a handle of type `actor` or `typed_actor<...>`. The second argument denotes the TCP port. The OS will pick a random high-level port when passing 0. The third parameter configures the listening address. Passing null will accept all incoming connections (`INADDR_ANY`). Finally, 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, with `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.

The member function returns the bound port on success. Otherwise, an `error` (see § 10) is returned.

```
template <class T>
expected<uint16_t> middleman::unpublish(T x, uint16_t port = 0);
```

The member function `unpublish` allows actors to close a port manually. This is performed automatically if the published actor terminates. Passing 0 as second argument closes all ports an actor is published to, otherwise only one specific port is closed.

The function returns an `error` (see § 10) if the actor was not bound to given port.

```
template<class T = actor>
expected<T> middleman::remote_actor(std::string host, uint16_t port);
```

After a server has published an actor with `publish`, clients can connect to the published actor by calling `remote_actor`:

```
// node A
auto ping = spawn(ping);
system.middleman().publish(ping, 4242);

// node B
auto ping = system.middleman().remote_actor("node A", 4242);
if (! ping) {
  cerr << "unable to connect to node A: "
       << system.render(ping.error()) << std::endl;
} else {
  self->send(*ping, ping_atom::value);
}
```

There is no difference between server and client after the connection phase. Remote actors use the same handle types as local actors and are thus fully transparent.

The function pair `open` and `connect` allows users to connect CAF instances without remote actor setup. The function `connect` returns a `node_id` that can be used for remote spawning (see § 19).

17.3 Free Functions

The following free functions in the namespace `caf::io` avoid calling the middleman directly. This enables users to easily switch between communication backends as long as the interfaces have the same signatures. For example, the (experimental) OpenSSL binding of CAF implements the same functions in the namespace `caf::openssl` to easily switch between encrypted and unencrypted communication.

`expected<uint16> open(actor_system&, uint16, const char*, bool)`	See § 17.2.
`expected<uint16> publish(T, uint16, const char*, bool)`	See § 17.2.
`expected<void> unpublish(T x, uint16)`	See § 17.2.
`expected<node_id> connect(actor_system&, std::string host, uint16_t port)`	See § 17.2.
`expected<T> remote_actor<T = actor>(actor_system&, string, uint16)`	See § 17.2.

17.4 Transport Protocols **experimental**

CAF communication uses TCP per default and thus the functions shown in the middleman API above are related to TCP. There are two alternatives to plain TCP: TLS via the OpenSSL module shortly discussed in § 17.3 and UDP.

UDP is integrated in the default multiplexer and BASP broker. Set the flag `middleman_enable_udp` to true to enable it (see § 11). This does not require you to disable TCP. Use `publish_udp` and `remote_actor_udp` to establish communication.

Communication via UDP is inherently unreliable and unordered. CAF reestablishes order and drops messages that arrive late. Messages that are sent via datagrams are limited to a maximum of 65.535

bytes which is used as a receive buffer size by CAF. Note that messages that exceed the MTU are fragmented by IP and are considered lost if a single fragment is lost. Optional reliability based on retransmissions and messages slicing on the application layer are planned for the future.

18 Network I/O with Brokers

When communicating to other services in the network, sometimes low-level socket I/O is inevitable. For this reason, CAF provides *brokers*. A broker is an event-based actor running in the middleman that multiplexes socket I/O. 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. Brokers should outsource any considerable amount of work by spawning new actors or maintaining worker actors.

Note that all UDP-related functionality is still experimental .

18.1 Spawning Brokers

Brokers are implemented as functions and are spawned by calling on of the three following member functions of the middleman.

```
template <spawn_options Os = no_spawn_options,
          class F = std::function<void(broker*)>, class... Ts>
typename infer_handle_from_fun<F>::type
spawn_broker(F fun, Ts&&... xs);

template <spawn_options Os = no_spawn_options,
          class F = std::function<void(broker*)>, class... Ts>
expected<typename infer_handle_from_fun<F>::type>
spawn_client(F fun, const std::string& host, uint16_t port, Ts&&... xs);

template <spawn_options Os = no_spawn_options,
          class F = std::function<void(broker*)>, class... Ts>
expected<typename infer_handle_from_fun<F>::type>
spawn_server(F fun, uint16_t port, Ts&&... xs);
```

The function `spawn_broker` simply spawns a broker. The convenience function `spawn_client` tries to connect to given host and port over TCP and returns a broker managing this connection on success. Finally, `spawn_server` opens a local TCP port and spawns a broker managing it on success. There are no convenience functions spawn a UDP-based client or server.

18.2 Class `broker`

```
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)
void write(datagram_handle hdl, size_t num_bytes, const void* buf)
```

Writes data to the output buffer.

```
void enqueue_datagram(datagram_handle hdl, std::vector<char> buf);
```

Enqueues a buffer to be sent as a datagram. Use of this function is encouraged over write as it allows reuse of the buffer which can be returned to the broker in a `datagram_sent_msg`.

```
void flush(connection_handle hdl);
void flush(datagram_handle hdl);
```

Sends the data from the output buffer.

```
template <class F, class... Ts>
actor fork(F fun, connection_handle hdl, Ts&&... xs);
```

Spawns a new broker that takes ownership of a given connection.

```
size_t num_connections();
```

Returns the number of open connections.

```
void close(connection_handle hdl);
void close(accept_handle hdl);
void close(datagram_handle hdl);
```

Closes the endpoint related to the handle.

```
expected<std::pair<accept_handle, uint16_t>>
add_tcp_doorman(uint16_t port = 0, const char* in = nullptr,
                bool reuse_addr = false);
```

Creates new doorman that accepts incoming connections on a given port and returns the handle to the doorman and the port in use or an error.

```
expected<connection_handle>
add_tcp_scribe(const std::string& host, uint16_t port);
```

Creates a new scribe to connect to host:port and returns handle to it or an error.

```
expected<std::pair<datagram_handle, uint16_t>>
add_udp_datagram_servant(uint16_t port = 0, const char* in = nullptr,
                         bool reuse_addr = false);
```

Creates a datagram servant to handle incoming datagrams on a given port. Returns the handle to the servant and the port in use or an error.

```
expected<datagram_handle>
add_udp_datagram_servant(const std::string& host, uint16_t port);
```

Creates a datagram servant to send datagrams to host:port and returns a handle to it or an error.

18.3 Broker-related Message Types

Brokers receive system messages directly from the middleman for connection and acceptor events.

Note: brokers are *required* to handle these messages immediately regardless of their current state. Not handling the system messages from the broker results in loss of data, because system messages are *not* delivered through the mailbox and thus cannot be skipped.

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

Indicates that `source` accepted a new TCP connection identified by `handle`.

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

Contains raw bytes received from `handle`. The amount of data received per event is controlled with `configure_read` (see 18.2). It is worth mentioning that the buffer is re-used whenever possible.

```
struct data_transferred_msg {
  connection_handle handle;
  uint64_t written;
  uint64_t remaining;
};
```

This message informs the broker that the `handle` sent `written` bytes with `remaining` bytes in the buffer. Note, that these messages are not sent per default but must be explicitly enabled via the member function `ack_writes`.

```
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 its handles is no longer valid.

```
struct connection_passivated_msg {
  connection_handle handle;
};

struct acceptor_passivated_msg {
  accept_handle handle;
};
```

A `connection_passivated_msg` or `acceptor_passivated_msg` informs the broker that one of its handles entered passive mode and no longer accepts new data or connections (see § 18.4).

The following messages are related to UDP communication (see § 17.4. Since UDP is not connection oriented, there is no equivalent to the `new_connection_msg` of TCP.

```
struct new_datagram_msg {
  datagram_handle handle;
  network::receive_buffer buf;
};
```

Contains the raw bytes from `handle`. The buffer always has a maximum size of 65k to receive all regular UDP messages. The amount of bytes can be queried via the `.size()` member function. Similar to TCP, the buffer is reused when possible—please do not resize it.

```
struct datagram_sent_msg {
  datagram_handle handle;
  uint64_t written;
  std::vector<char> buf;
};
```

This message informs the broker that the `handle` sent a datagram of `written` bytes. It includes the buffer that held the sent message to allow its reuse. Note, that these messages are not sent per default but must be explicitly enabled via the member function `ack_writes`.

```
struct datagram_servant_closed_msg {
  std::vector<datagram_handle> handles;
};
```

A `datagram_servant_closed_msg` informs the broker that one of its handles is no longer valid.

```
struct datagram_servant_passivated_msg {
  datagram_handle handle;
};
```

A `datagram_servant_closed_msg` informs the broker that one of its handles entered passive mode and no longer accepts new data (see § 18.4).

18.4 Manually Triggering Events **experimental**

Brokers receive new events as `new_connection_msg` and `new_data_msg` as soon and as often as they occur, per default. This means a fast peer can overwhelm a broker by sending it data faster than the broker can process it. In particular if the broker outsources work items to other actors, because work items can accumulate in the mailboxes of the workers.

Calling `self->trigger(x,y)`, where `x` is a connection or acceptor handle and `y` is a positive integer, allows brokers to halt activities after `y` additional events. Once a connection or acceptor stops accepting new data or connections, the broker receives a `connection_passivated_msg` or `acceptor_passivated_msg`.

Brokers can stop activities unconditionally with `self->halt(x)` and resume activities unconditionally with `self->trigger(x)`.

19 Remote Spawning of Actors experimental

Remote spawning is an extension of the dynamic spawn using run-time type names (see § 11.5). The following example assumes a typed actor handle named `calculator` with an actor implementing this messaging interface named ”calculator”.

```
void client(actor_system& system, const config& cfg) {
  auto node = system.middleman().connect(cfg.host, cfg.port);
  if (!node) {
    cerr << "*** connect failed: "
         << system.render(node.error()) << endl;
    return;
  }
  auto type = "calculator"; // type of the actor we wish to spawn
  auto args = make_message(); // arguments to construct the actor
  auto tout = std::chrono::seconds(30); // wait no longer than 30s
  auto worker = system.middleman().remote_spawn<calculator>(*node, type,
                                                            args, tout);
  if (!worker) {
    cerr << "*** remote spawn failed: "
         << system.render(worker.error()) << endl;
    return;
  }
  // start using worker in main loop
  client_repl(make_function_view(*worker));
```

We first connect to a CAF node with `middleman().connect(...)`. On success, `connect` returns the node ID we need for `remote_spawn`. This requires the server to open a port with `middleman().open(...)` or `middleman().publish(...)`. Alternatively, we can obtain the node ID from an already existing remote actor handle—returned from `remote_actor` for example—via `hdl->node()`. After connecting to the server, we can use `middleman().remote_spawn<...>(...)` to create actors remotely.

Part III

Appendix

20 Frequently Asked Questions

This Section is a compilation of the most common questions via GitHub, chat, and mailing list.

20.1 Can I Encrypt CAF Communication?

Yes, by using the OpenSSL module (see § 17.3).

20.2 Can I Create Messages Dynamically?

Yes.

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 `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();
```

20.3 What Debugging Tools Exist?

The `scripts/` and `tools/` directories contain some useful tools to aid in development and debugging.

`scripts/atom.py` converts integer atom values back into strings.

`scripts/demystify.py` replaces cryptic `typed_mpi<...>` templates with `replies_to<...>::with<...>` and `atom_constant<...>` with a human-readable representation of the actual atom.

`scripts/caf-prof` is an R script that generates plots from CAF profiler output.

`caf-vec` is a (highly) experimental tool that annotates CAF logs with vector timestamps. It gives you happens-before relations and a nice visualization via ShiViz.

21 Utility

CAF includes a few utility classes that are likely to be part of C++ eventually (or already are in newer versions of the standard). However, until these classes are part of the standard library on all supported compilers, we unfortunately have to maintain our own implementations.

21.1 Class `optional`

Represents a value that may or may not exist.

Constructors	
`(T value)`	Constructs an object with a value.
`(none_t = none)`	Constructs an object without a value.

Observers	
`explicit operator bool()`	Checks whether the object contains a value.
`T* operator->()`	Accesses the contained value.
`T& operator*()`	Accesses the contained value.

21.2 Class `expected`

Represents the result of a computation that *should* return a value. If no value could be produced, the `expected<T>` contains an `error` (see § 10).

Constructors	
`(T value)`	Constructs an object with a value.
`(error err)`	Constructs an object with an error.

Observers	
`explicit operator bool()`	Checks whether the object contains a value.
`T* operator->()`	Accesses the contained value.
`T& operator*()`	Accesses the contained value.
`error& error()`	Accesses the contained error.

21.3 Constant `unit`

The constant `unit` of type `unit_t` is the equivalent of `void` and can be used to initialize `optional<void>` and `expected<void>`.

21.4 Constant `none`

The constant `none` of type `none_t` can be used to initialize an `optional<T>` to represent “nothing”.

22 Common Pitfalls

This Section highlights common mistakes or C++ subtleties that can show up when programming in CAF.

22.1 Defining Message Handlers

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) { /*...*/ }
);
```

22.2 Event-Based API

The member function `become` does not block, i.e., always returns immediately. Thus, lambda expressions should *always* capture by value. Otherwise, all references on the stack will cause undefined behavior if the lambda expression is executed.

22.3 Requests

A handle returned by `request` represents *exactly one* response message. It is not possible to receive more than one response message.

The handle returned by `request` is bound to the calling actor. It is not possible to transfer a handle to a response to another actor.

22.4 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 information is message passing. Sending the same message to multiple actors does not result in copying the data several times.

23 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. 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 <random>
#include <chrono>
#include <cstdlib>
#include <iostream>

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

using namespace caf;
using std::endl;

void caf_main(actor_system& system) {
  for (int i = 1; i <= 50; ++i) {
    system.spawn([i](blocking_actor* self) {
      aout(self) << "Hi there! This is actor nr. "
                 << i << "!" << endl;
      std::random_device rd;
      std::default_random_engine re(rd());
      std::chrono::milliseconds tout{re() % 10};
      self->delayed_send(self, tout, 42);
      self->receive(
        [i, self](int) {
          aout(self) << "Actor nr. "
                     << i << " says goodbye!" << endl;
        }
      );
    });
  }
}

CAF_MAIN()
```

24 Migration Guides

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

24.1 0.8 to 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 `current_sender()` now returns the *address* of the sender. This means that previously valid code such as `send(current_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.

24.2 0.9 to 0.10 (libcppa to 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, `cppa/actor.hpp` becomes `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`.

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.

`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. However, we wanted to put emphasize on its use case.

`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`

24.3 0.10 to 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`.

24.4 0.11 to 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.

24.5 0.12 to 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). `local_actor::on_exit` once again became virtual.

In case you were using the old `cppa::options_description` API, you can migrate to the new API based on `extract` (see § 12.6).

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` (see § 5.3).

24.6 0.13 to 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.

The policy classes `broadcast`, `random`, and `round_robin` in `actor_pool` were removed and replaced by factory functions using the same name.

24.7 0.14 to 0.15

Version 0.15 replaces the singleton-based architecture with `actor_system`. Most of the free functions in namespace `caf` are now member functions of `actor_system` (see § 5.1). Likewise, most functions in namespace `caf::io` are now member functions of `middleman` (see § 17). The structure of CAF applications has changed fundamentally with a focus on configurability. Setting and fine-tuning the scheduler, changing parameters of the middleman, etc. is now bundled in the class `actor_system_config`. The new configuration mechanism is also easily extensible.

Patterns are now limited to the simple notation, because the advanced features (1) are not implementable for statically typed actors, (2) are not portable to Windows/MSVC, and (3) drastically impact compile times. Dropping this functionality also simplifies the implementation and improves performance.

The `blocking_api` flag has been removed. All variants of *spawn* now auto-detect blocking actors.