

CAF

The C++ Actor Framework

User Manual

version 0.15.0+exp.sha.3d3176d

Dominik Charousset

June 7, 2016

Contents

I	Core Library	1
1	Introduction	2
1.1	Actor Model	2
1.2	Terminology	2
1.2.1	Dynamically Typed Actor	2
1.2.2	Statically Typed Actor	2
1.2.3	Actor References	3
1.2.4	Spawning	3
1.2.5	Monitor	3
1.2.6	Link	3
1.2.7	Experimental Features	4
2	Overview	5
2.1	Features	5
2.2	Minimal Compiler Versions	5
2.3	Supported Operating Systems	5
2.4	Hello World Example	6
3	Message Handlers	7
3.1	Definition and Composition	7
3.2	Atoms	8
4	Actors	9
4.1	Environment / Actor Systems	9
4.2	Common Actor Base Type	10
4.3	Messaging Interfaces	11
4.4	Spawning Actors	12
4.5	Function-based Actors	12
4.6	Class-based Actors	14
4.7	Stateful Actors	15
4.8	Actors from Composable Behaviors <small>experimental</small>	16

4.9	Attaching Cleanup Code to Actors	17
4.10	Blocking Actors	18
4.10.1	Receiving Messages	18
4.10.2	Scoped Actors	19
5	Message Passing	20
5.1	Structure of Mailbox Elements	20
5.2	Default and System Message Handlers	20
5.2.1	Down Handler	21
5.2.2	Exit Handler	21
5.2.3	Error Handler	21
5.2.4	Default Handler	21
5.3	Requests	21
5.3.1	Sending Requests and Handling Responses	22
5.3.2	Error Handling in Requests	23
5.4	Delaying Messages	25
5.5	Delegating Messages	26
5.6	Response Promises	27
5.7	Message Priorities	28
6	Scheduler	29
6.1	Policies	29
6.2	Work Stealing	30
6.3	Work Sharing	31
7	Registry	32
8	Reference Counting	33
8.1	Shared Ownership in C++	33
8.2	Smart Pointers to Actors	34
8.3	Strong and Weak References	35
8.4	Converting Actor References with <code>actor_cast</code>	36
8.5	Breaking Cycles Manually	36
9	Errors	37

9.1	Class Interface	37
9.2	Add Custom Error Categories	37
9.3	System Error Codes	39
9.4	Default Exit Reasons	40
10	Configuring Actor Applications	41
10.1	Loading Modules	42
10.2	Command Line Options and INI Configuration Files	42
10.3	Adding Custom Message Types	44
10.4	Adding Custom Error Types	45
11	Type-Erased Tuples and Messages	46
11.1	RTTI and Type Numbers	46
11.2	Class <code>type_erased_tuple</code>	46
11.3	Class <code>message</code>	47
11.4	Class <code>message_builder</code>	48
11.5	Extracting	49
11.6	Extracting Command Line Options	50
12	Group Communication	51
12.1	Anonymous Groups	51
12.2	Local Groups	51
12.3	Remote Groups	51
13	Managing Groups of Workers <small>experimental</small>	52
13.1	Dispatching Policies	52
II	I/O Library	54
14	Middleman	55
14.1	Class <code>middleman</code>	55
14.2	Publishing and Connecting	55
15	Network I/O with Brokers	57
15.1	Spawning Brokers	57

15.2 Class <code>broker</code>	57
15.3 Broker-related Message Types	59

III Appendix 60

16 Frequently Asked Questions 61

16.1 Can I Create Messages Dynamically?	61
---	----

17 Common Pitfalls 62

17.1 Defining Message Handlers	62
17.2 Event-Based API	62
17.3 Requests	62
17.4 Sharing	63

18 Using `about` – A Concurrency-safe Wrapper for `cout` 64

19 Migration Guides 65

19.1 0.8 to 0.9	65
19.2 0.9 to 0.10 (<code>libcppa</code> to CAF)	66
19.3 0.10 to 0.11	67
19.4 0.11 to 0.12	67
19.5 0.12 to 0.13	67
19.6 0.13 to 0.14	67
19.7 0.14 to 0.15	68

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. CAF assumes addresses and handles to be *non-null*.

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 § 8).

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 § 8).

Pointer In a few instances, CAF uses `strong_actor_ptr` to refer to an actor using *strong reference semantics* (see § 8) without knowing the proper handle type. Pointers must be converted to a handle via `actor_cast` (see § ??) 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 § 5.2.1) 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 § 5.2.2) 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.

1.2.7 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 2

2.3 Supported Operating Systems

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

2.4 Hello World Example

```
1 #include <string>
2 #include <iostream>
3
4 #include "caf/all.hpp"
5
6 using std::endl;
7 using std::string;
8
9 using namespace caf;
10
11 behavior mirror(event_based_actor* self) {
12     // return the (initial) actor behavior
13     return {
14         // a handler for messages containing a single string
15         // that replies with a string
16         [=](const string& what) -> string {
17             // prints "Hello World!" via aout (thread-safe cout wrapper)
18             aout(self) << what << endl;
19             // reply "!dlroW olleH"
20             return string(what.rbegin(), what.rend());
21         }
22     };
23 }
24
25 void hello_world(event_based_actor* self, const actor& buddy) {
26     // send "Hello World!" to our buddy ...
27     self->request(buddy, std::chrono::seconds(10), "Hello World!").then(
28         // ... wait up to 10s for a response ...
29         [=](const string& what) {
30             // ... and print it
31             aout(self) << what << endl;
32         }
33     );
34 }
35
36 int main() {
37     // our CAF environment
38     actor_system system;
39     // create a new actor that calls 'mirror()'
40     auto mirror_actor = system.spawn(mirror);
41     // create another actor that calls 'hello_world(mirror_actor)';
42     system.spawn(hello_world, mirror_actor);
43     // system will wait until both actors are destroyed before leaving main
44 }
```

3 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.

3.1 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.

```
1 message_handler x1{
2   [] (int i) { /*...*/ },
3   [] (double db) { /*...*/ },
4   [] (int a, int b, int c) { /*...*/ }
5 };
```

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 § 5.2.4).

```
1 message_handler x2{
2   [] (double db) { /*...*/ },
3   [] (double db) { /* - unreachable - */ }
4 };
```

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.

```
1 message_handler x3 = x1.or_else(x2);
2 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`.

3.2 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.

```
1 atom_value a1 = atom("add");
2 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*.

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

Using these constants, we can now define message passing interfaces in a convenient way:

```
1 behavior do_math{
2     [](add_atom, int a, int b) {
3         return a + b;
4     },
5     [](multiply_atom, int a, int b) {
6         return a * b;
7     }
8 };
9
10 // 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.

4 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 (see § ??).

4.1 Environment / Actor Systems

All actors live in an `actor_system` representing an actor environment including scheduler (see § 6), registry (see § 7), and optional components such as a middleman (see § 14). 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 § 10. 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*.

4.2 Common Actor Base Type

The class `local_actor` is the base of all user-defined actors in CAF and 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`.

Types

<code>mailbox_type</code>	A concurrent, many-writers-single-reader queue type.
<code>error_handler</code>	<code>std::function<void (error*)></code>

Constructors

<code>(actor_config&)</code>	Constructs the actor using a config.
----------------------------------	--------------------------------------

Observers

<code>actor_addr address()</code>	Returns the address of this actor
<code>bool trap_exit()</code>	Checks whether this actor traps exit messages
<code>vector<group> joined_groups()</code>	Returns all subscribed groups

Modifiers

<code>quit(uint32_t reason = normal)</code>	Finishes execution of this actor
<code>void trap_exit(bool enabled)</code>	Enables or disables trapping of exit messages
<code>void join(const group&)</code>	Subscribes to a group
<code>void leave(const group&)</code>	Unsubscribes from a group
<code>void monitor(actor_addr)</code>	Unidirectionally monitors an actor (see § ??)
<code>void demonitor(actor_addr whom)</code>	Removes a monitor from <code>whom</code>
<code>void set_exception_handler(F)</code>	Sets a custom handler for uncaught exceptions
<code>void on_exit()</code>	Can be overridden for performing cleanup code

4.3 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 \rightarrow 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.

```
1 using add_atom = atom_constant<atom("add")>;
2 using sub_atom = atom_constant<atom("sub")>;
3
4 using calculator_actor = typed_actor<replies_to<add_atom, int, int>::with<int>,
5                                     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:

```
1 using i1 = typed_actor<replies_to<A>::with<B>, replies_to<C>::with<D>>;
2 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

<code>behavior_type</code>	A statically typed set of message handlers.
<code>base</code>	Base type for actors, i.e., <code>typed_event_based_actor<...></code> .
<code>pointer</code>	A pointer of type <code>base*</code> .
<code>stateful_base<T></code>	See § 4.7.
<code>stateful_pointer<T></code>	A pointer of type <code>stateful_base<T>*</code> .
<code>extend<Ts...></code>	Extend this typed actor with <code>Ts...</code>
<code>extend_with<Other></code>	Extend this typed actor with all cases from <code>Other</code> .

4.4 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.

```
1 behavior calculator_fun(event_based_actor* self);
2 void blocking_calculator_fun(blocking_actor* self);
3 calculator_actor::behavior_type typed_calculator_fun();
4 class calculator;
5 class blocking_calculator;
6 class typed_calculator;
```

Spawning an actor for each implementation is illustrated below.

```
1 auto a1 = system.spawn(blocking_calculator_fun);
2 auto a2 = system.spawn(calculator_fun);
3 auto a3 = system.spawn(typed_calculator_fun);
4 auto a4 = system.spawn<blocking_calculator>();
5 auto a5 = system.spawn<calculator>();
6 auto a6 = system.spawn<typed_calculator>();
```

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.

4.5 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 § 4.3).

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 § 3) 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 § 4.7).

The following three functions implement the prototypes shown in § 4.4 and illustrate one blocking actor and two event-based actors (statically and dynamically typed).

```

1 // function-based, dynamically typed, event-based API
2 behavior calculator_fun(event_based_actor*) {
3     return behavior{
4         [] (add_atom, int a, int b) {
5             return a + b;
6         },
7         [] (sub_atom, int a, int b) {
8             return a - b;
9         }
10    };
11 }
12
13 // function-based, dynamically typed, blocking API
14 void blocking_calculator_fun(blocking_actor* self) {
15     self->receive_loop (
16         [] (add_atom, int a, int b) {
17             return a + b;
18         },
19         [] (sub_atom, int a, int b) {
20             return a - b;
21         }
22    );
23 }
24
25 // function-based, statically typed, event-based API
26 calculator_actor::behavior_type typed_calculator_fun() {
27     return {
28         [] (add_atom, int a, int b) {
29             return a + b;
30         },
31         [] (sub_atom, int a, int b) {
32             return a - b;
33         }
34    };
35 }

```

4.6 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 § 4.8) that works well with inheritance. The following three examples implement the forward declarations shown in § 4.4.

```
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 };
```

4.7 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.

```
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;  
6 };  
7  
8 cell::behavior_type type_checked_cell(cell::stateful_pointer<cell_state> self) {  
9     return {  
10         [=](put_atom, int val) {  
11             self->state.value = val;  
12         },  
13         [=](get_atom) {  
14             return self->state.value;  
15         }  
16     };  
17 }  
18  
19 behavior unchecked_cell(stateful_actor<cell_state>* self) {  
20     return {  
21         [=](put_atom, int val) {  
22             self->state.value = val;  
23         },  
24         [=](get_atom) {  
25             return self->state.value;  
26         }  
27     };  
28 }
```

Stateful actors are spawned in the same way as any other function-based actor (see § 4.5).

```
1 auto cell2 = system.spawn(unchecked_cell);  
2 auto f = make_function_view(cell1);
```

4.8 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 § 4.3).

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, param<C>)`.

Note that `operator()` will take integral types as well as atom constants by value instead of by reference. A `result<T>` accepts either a value of type `T`, a `skip_t` (see § 5.2.4), an `error` (see § 9), a `delegated<T>` (see § 5.5), or a `response_promise<T>` (see § 5.6). 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. In this step, CAF will set any `operator()` to pure virtual again that occurs in more than one base class. This ensures that all conflicts are properly resolved by the combining class. Any composable (or composed) state with no pure virtual member functions can be spawned directly through an actor system by calling `system.spawn<...>()`, as shown below.

```
1 namespace {
2
3 // using add_atom = atom_constant<atom("add")>; (defined in atom.hpp)
4 using multiply_atom = atom_constant<atom("multiply")>;
5
6 using adder = typed_actor<replies_to<add_atom, int, int>::with<int>>;
7 using multiplier = typed_actor<replies_to<multiply_atom, int, int>::with<int>>;
8
9 class adder_bhvr : public composable_behavior<adder> {
10 public:
11     result<int> operator()(add_atom, int x, int y) override {
12         return x + y;
13     }
14 };
15
16 class multiplier_bhvr : public composable_behavior<multiplier> {
17 public:
18     result<int> operator()(multiply_atom, int x, int y) override {
19         return x * y;
20     }
21 };
22
23 // calculator_bhvr can be inherited from or composed further
24 using calculator_bhvr = composed_behavior<adder_bhvr, multiplier_bhvr>;
25
26 } // namespace <anonymous>
27
28 void caf_main(actor_system& system) {
29     auto f = make_function_view(system.spawn<calculator_bhvr>());
30     cout << "10 + 20 = " << f(add_atom::value, 10, 20) << endl;
31     cout << "7 * 9 = " << f(multiply_atom::value, 7, 9) << endl;
32 }
```

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 `const T&`, 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 exists. This copy-on-write optimization avoids race conditions by design, while keeping copy operations to a minimum. 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.

```

1  using dict = typed_actor<reacts_to<put_atom, string, string>,
2                                replies_to<get_atom, string>::with<string>>;
3
4  class dict_behavior : public composable_behavior<dict> {
5  public:
6      result<string> operator()(get_atom, param<string> key) override {
7          auto i = values_.find(key);
8          if (i == values_.end())
9              return "";
10         return i->second;
11     }
12
13     result<void> operator()(put_atom, param<string> key,
14                             param<string> value) override {
15         if (values_.count(key) != 0)
16             return unit;
17         values_.emplace(key.move(), value.move());
18         return unit;
19     }
20
21 protected:
22     std::unordered_map<string, string> values_;
23 };
24
25 } // namespace <anonymous>

```

4.9 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.

```

1  void print_on_exit(const actor& hdl, const std::string& name) {
2      hdl->attach_functor( [=](const error& reason) {
3          cout << name << " exited: " << to_string(reason) << endl;
4      });
5  }

```

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

4.10 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.

4.10.1 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.

```
1 self->receive (
2     [](int x) { /* ... */ }
3 );
```

The code snippet above illustrates the use of `receive`. Note that the message handler passed to `receive` is a temporary object at runtime. Hence, calling `receive` inside a loop creates an unnecessary amount of short-lived objects. CAF provides three predefined receive loops to provide a more efficient way of defining loops.

```
1 // BAD
2 for (;;) {
3     receive (
4         // ...
5     );
6 }
7
8 // GOOD
9 receive_loop (
10    // ...
11 );
```

```
1 // BAD
2 std::vector<int> results;
3 for (size_t i = 0; i < 10; ++i) {
4     receive (
5         [&](int value) {
6             results.push_back(value);
7         }
8     );
9 }
10
11 // GOOD
12 std::vector<int> results;
13 size_t i = 0;
14 receive_for(i, 10) (
15     [&](int value) {
16         results.push_back(value);
17     }
18 );
```

```

1 // BAD
2 size_t received = 0;
3 do {
4     receive (
5         [&](int) {
6             ++received;
7         }
8     );
9 } while (received < 10);
10
11 // GOOD
12 size_t received = 0;
13 do_receive (
14     [&](int) {
15         ++received;
16     }
17 ).until([&] { return received >= 10; });

```

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

```

1 self->receive_loop (
2     [&](int value1) {
3         self->receive (
4             [&](float value2) {
5                 aout(self) << value1 << " ==> " << value2 << endl;
6             }
7         );
8     }
9 );

```

4.10.2 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.

```

1 void test(actor_system& system) {
2     scoped_actor self{system};
3     // spawn some actor
4     auto aut = self->spawn(my_actor_impl);
5     self->send(aut, "hi there");
6     // self will be destroyed automatically here; any
7     // actor monitoring it will receive down messages etc.
8 }

```

Note that `scoped_actor` throws an `actor_exited` exception when forced to quit for some reason, e.g., after receiving an `exit_msg` (see § 5.2.2).

5 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 § 5.3 and § 5.5.

5.1 Structure of Mailbox Elements

When enqueueing 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.

```
+-----+
| mailbox_element |
+-----+
| sender          |
| message ID      |
| stages          |
| content         |
+-----+
```

The sender is stored as a `strong_actor_ptr` (see § 5) 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` is a message object (see § 11) storing a type-erased tuple.

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.

5.2 Default and System Message Handlers

CAF has three system-level message types that all actor must handle regardless of there current state. Consequently, system messages are not handled by regular message handlers. The fourth special-purpose message handler is used as fallback for unmatched messages.

5.2.1 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_message&)` or `void (local_actor*, down_message&)`. The latter signature allows users to implement down message handlers as free function.

5.2.2 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 (local_actor*, exit_message&)`.

5.2.3 Error Handler

Actors send error messages to others by returning an `error` (see § 9) 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 (local_actor*, error&)`. Additionally, `request` accepts an error handler as second argument to handle errors for a particular request (see § 5.3.2). The default handler is used as fallback if `request` is used without error handler.

5.2.4 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> (local_actor*, const type_erased_tuple*)`, 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. The default implementations for this handler are: `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. Event-based actors use `print_and_drop` as default, while blocking actors use `skip`.

5.3 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.

5.3.1 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 § 9.3) error message (see § 5.2.3) 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;
6 };
7
8 cell::behavior_type cell_impl(cell::stateful_pointer<cell_state> self, int x0) {
9     self->state.value = x0;
10    return {
11        [=](put_atom, int val) {
12            self->state.value = val;
13        },
14        [=](get_atom) {
15            return self->state.value;
16        }
17    };
18 }
```

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

```
1 void waiting_testee(event_based_actor* self, vector<cell> cells) {
2     for (auto& x : cells)
3         self->request(x, seconds(1), get_atom::value).await([=](int y) {
4             aout(self) << "cell #" << x.id() << " -> " << y << endl;
5         });
6 }
7
8 void multiplexed_testee(event_based_actor* self, vector<cell> cells) {
9     for (auto& x : cells)
10        self->request(x, seconds(1), get_atom::value).then([=](int y) {
11            aout(self) << "cell #" << x.id() << " -> " << y << endl;
12        });
13 }
```

The second half of the example shows a blocking actor making use of `receive`.

```
1 void blocking_testee(blocking_actor* self, vector<cell> cells) {
2     for (auto& x : cells)
3         self->request(x, seconds(1), get_atom::value).receive([&](int y) {
4             aout(self) << "cell #" << x.id() << " -> " << y << endl;
5         });
6 }
```

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

```
1 for (auto i = 0; i < 5; ++i)
2     cells.emplace_back(system.spawn(cell_impl, i * i));
3 scoped_actor self{system};
```

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 predictable 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.

5.3.2 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`, `await`, or `receive`. If no such handler is defined, the default error handler (see § 5.2.3) is used as a fallback.

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 § 9).

```

1  enum class math_error : uint8_t {
2      division_by_zero = 1
3  };
4
5  error make_error(math_error x) {
6      return {static_cast<uint8_t>(x), atom("math")};
7  }
8
9  using div_atom = atom_constant<atom("add")>;
10
11 using divider = typed_actor<replies_to<div_atom, double, double>::with<double>>;
12
13 divider::behavior_type divider_impl() {
14     return {
15         [] (div_atom, double x, double y) -> result<double> {
16             if (y == 0.0)
17                 return math_error::division_by_zero;
18             return x / y;
19         }
20     };
21 }

```

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

```

1  cout << "y: " << flush;
2  std::cin >> y;
3  auto div = system.spawn(divider_impl);
4  scoped_actor self{system};
5  self->request(div, std::chrono::seconds(10), div_atom::value, x, y).receive(
6      [&](double z) {
7          aout(self) << x << " / " << y << " = " << z << endl;
8      },
9      [&](const error& err) {

```

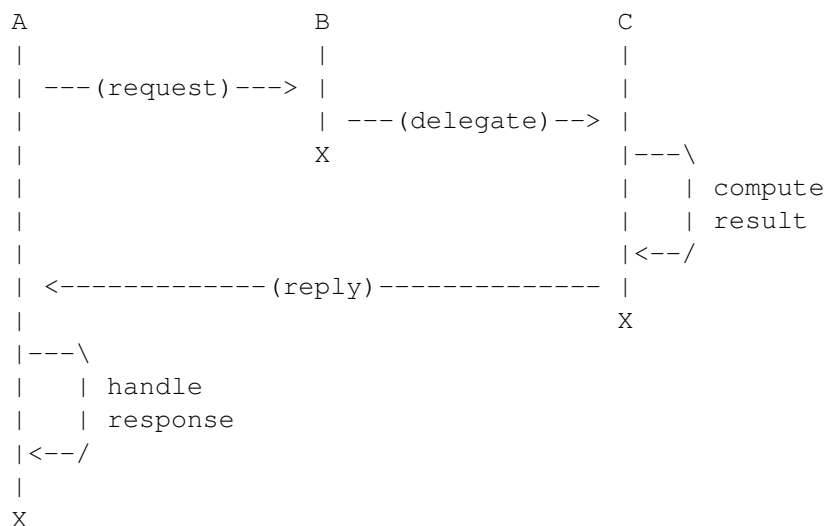
5.4 Delaying Messages

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

```
1 // uses a message-based loop to iterate over all animation steps
2 void dancing_kirby(event_based_actor* self) {
3     // let's get it started
4     self->send(self, step_atom::value, size_t{0});
5     self->become (
6         [=](step_atom, size_t step) {
7             if (step == sizeof(animation_step)) {
8                 // we've printed all animation steps (done)
9                 cout << endl;
10                self->quit();
11                return;
12            }
13            // print given step
14            draw_kirby(animation_steps[step]);
15            // animate next step in 150ms
16            self->delayed_send(self, std::chrono::milliseconds(150),
17                               step_atom::value, step + 1);
18        }
19    );
20 }
```

5.5 Delegating Messages

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



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.

```

1 void actor_a(event_based_actor* self, calc worker) {
2     self->request(worker, std::chrono::seconds(10), add_atom::value, 1, 2).then(
3         [=](int result) {
4             aout(self) << "1 + 2 = " << result << endl;
5         }
6     );
7 }
8
9 calc::behavior_type actor_b(calc::pointer self, calc worker) {
10     return {
11         [=](add_atom add, int x, int y) {
12             return self->delegate(worker, add, x, y);
13         }
14     };
15 }
16
17 calc::behavior_type actor_c() {
18     return {
19         [](add_atom, int x, int y) {
20             return x + y;
21         }
22     };
23 }
24
25 void caf_main(actor_system& system) {
26     system.spawn(actor_a, system.spawn(actor_b, system.spawn(actor_c)));
27 }
  
```

5.6 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 }
```


5.7 Message Priorities

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

```
1 #include "caf/all.hpp"
2
3 using std::endl;
4 using namespace caf;
5
6 behavior foo(event_based_actor* self) {
7     self->send(self, "world");
8     self->send<message_priority::high>(self, "hello");
9     // when spawning `foo` with priority_aware flag, it will print "hello" first
10    return {
11        [=](const std::string& str) {
12            aout(self) << str << endl;
13        }
14    };
15 }
16
17 void caf_main(actor_system& system) {
18     scoped_actor self{system};
19     aout(self) << "spawn foo" << endl;
20     self->spawn(foo);
21     self->await_all_other_actors_done();
22     aout(self) << "spawn foo again with priority_aware flag" << endl;
23     self->spawn<priority_aware>(foo);
24 }
25
26 CAF_MAIN()
```

6 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 `detach` spawn option, e.g., `system.spawn<detach>(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.

6.1 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.

6.2 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.

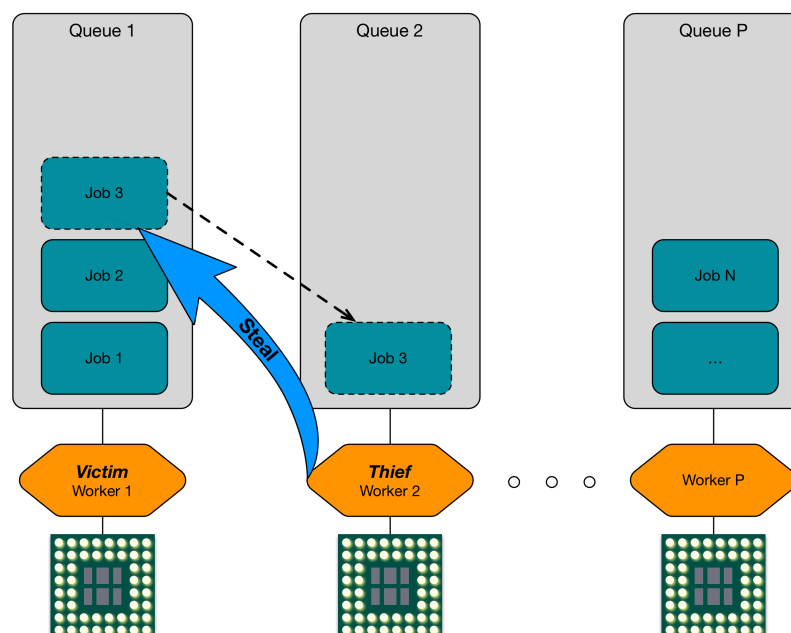


Figure 1: Stealing of work items

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 § 10).

6.3 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.

7 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 § 3.2) 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 § 5).

Users can use the registry to make actors system-wide available by name. The middleman (see § 14) 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

<code>name_map</code>	<code>unordered_map<atom_value, strong_actor_ptr></code>
-----------------------	--

Observers

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

Modifiers

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

8 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.

8.1 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.

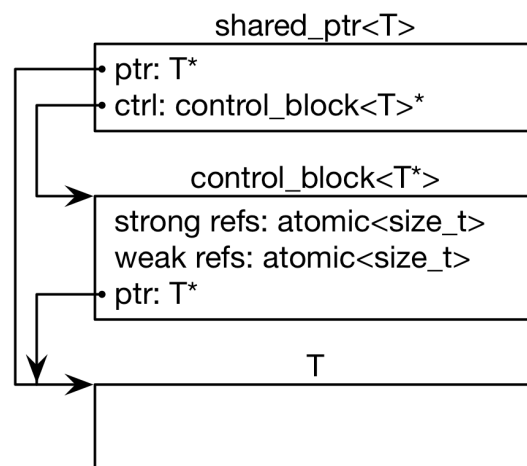


Figure 2: Shared pointer design in the C++ standard library

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.

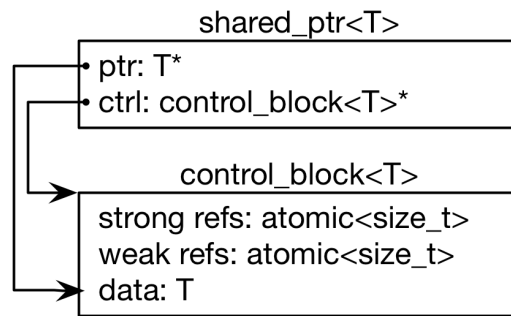


Figure 3: Memory layout when using `std::make_shared`

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.

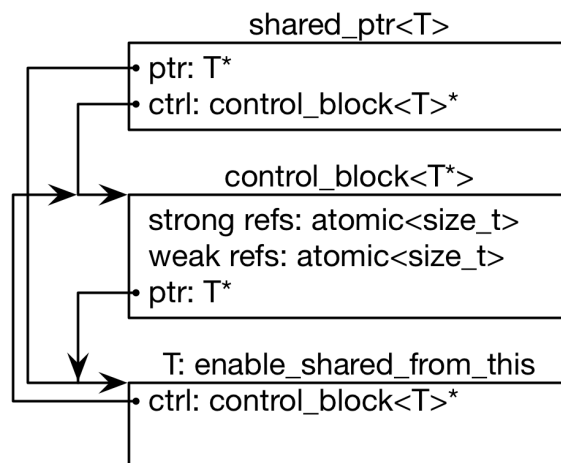


Figure 4: Memory layout with `std::enable_shared_from_this`

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.

8.2 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.

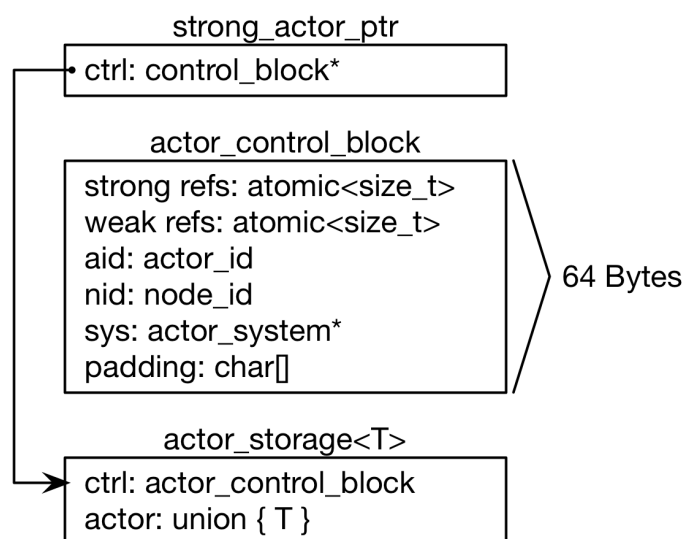


Figure 5: Shared pointer design in CAF

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 by using a `static_assert`.

8.3 Strong and Weak References

A *strong* reference manipulates the `strong refs` counter 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 § 8.5). 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).

8.4 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 `actor_addr` can *not* be converted to an actor handle directly, because handles are guaranteed to be non-null and the upgrade fails (returns null) if the actor already reached zero strong references and was destroyed.

The syntax for `actor_cast` resembles builtin C++ casts. For example, `actor_cast<actor>(x)` converts `x` to an handle of type `actor`. Users should never cast a `strong_actor_ptr` to a handle if it is null. Creating a “null handle” in this way causes undefined behavior (dereferencing null).

8.5 Breaking Cycles Manually

Cycles can occur only when using class-based actors when storing references to other actors via member variable. Stateful actors (see § 4.7) 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.

9 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 platform-neutral and serializable. Instead of using category singletons, CAF stores categories as atoms (see §3.2). Errors can also include a message to provide additional context information.

9.1 Class Interface

Constructors

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

Observers

<code>uint8_t code()</code>	Returns the error code
<code>atom_value category()</code>	Returns the error category
<code>message context()</code>	Returns additional context information
<code>explicit operator bool()</code>	Returns <code>code() != 0</code>

9.2 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.

```
1 enum class math_error : uint8_t {
2     division_by_zero = 1
3 };
4
5 error make_error(math_error x) {
6     return {static_cast<uint8_t>(x), atom("math")};
7 }
8
9 std::string to_string(math_error x) {
10     switch (x) {
11         case math_error::division_by_zero:
12             return "division_by_zero";
13         default:
14             return "-unknown-error-";
15     }
16 }
```

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.

```
1 public:
2     config() {
3         auto renderer = [](uint8_t x, atom_value, const message&) {
4             return "math_error" + deep_to_string_as_tuple(static_cast<math_error>(x));
5         };
6         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)"`.

9.3 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.

```
1  /// SEC stands for "System Error Code". This enum contains
2  /// error codes used internally by CAF.
3  enum class sec : uint8_t {
4      /// Indicates that an actor dropped an unexpected message.
5      unexpected_message = 1,
6      /// Indicates that a response message did not match the provided handler.
7      unexpected_response,
8      /// Indicates that the receiver of a request is no longer alive.
9      request_receiver_down,
10     /// Indicates that a request message timed out.
11     request_timeout,
12     /// Unpublishing failed because the actor is not bound to given port.
13     no_actor_published_at_port,
14     /// Migration failed because the state of an actor is not serializable.
15     state_not_serializable,
16     /// An actor received an unsupported key for `('sys', 'get', key)` messages.
17     unsupported_sys_key,
18     /// An actor received an unsupported system message.
19     unsupported_sys_message,
20     /// A remote node disconnected during CAF handshake.
21     disconnect_during_handshake,
22     /// Tried to forward a message via BASP to an invalid actor handle.
23     cannot_forward_to_invalid_actor,
24     /// Tried to forward a message via BASP to an unknown node ID.
25     no_route_to_receiving_node,
26     /// Middleman could not assign a connected handle to a broker.
27     failed_to_assign_scribe_from_handle,
28     /// User requested to close port 0 or to close a port not managed by CAF.
29     cannot_close_invalid_port,
30     /// Middleman could not connect to a remote node.
31     cannot_connect_to_node,
32     /// Middleman could not open requested port.
33     cannot_open_port,
34     /// A remote spawn failed because the provided types did not match.
35     cannot_spawn_actor_from_arguments,
36     /// Requested RIAC information about a node that does not exist.
37     no_such_riac_node
38 };
```

9.4 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 § 5.2.2) is overridden.

```
1  /// This error category represents fail conditions for actors.
2  enum class exit_reason : uint8_t {
3      /// Indicates that an actor finished execution without error.
4      normal = 0,
5      /// Indicates that an actor died because of an unhandled exception.
6      unhandled_exception,
7      /// Indicates that the exit reason for this actor is unknown, i.e.,
8      /// the actor has been terminated and no longer exists.
9      unknown,
10     /// Indicates that an actor pool unexpectedly ran out of workers.
11     out_of_workers,
12     /// Indicates that an actor was forced to shutdown by a user-generated event.
13     user_shutdown,
14     /// Indicates that an actor was killed unconditionally.
15     kill,
16     /// Indicates that an actor finished execution because a connection
17     /// to a remote link was closed unexpectedly.
18     remote_link_unreachable,
19     /// Indicates that an actor was killed because it became unreachable.
20     unreachable
21 };
```

10 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 § 10.2).

The following code example is a minimal CAF application with a middleman (see § 14) but without any custom configuration options.

```
1 void caf_main(actor_system& system) {
2     // ...
3 }
4 CAF_MAIN(io::middleman)
```

The compiler expands this example code to the following.

```
1 void caf_main(actor_system& system) {
2     // ...
3 }
4 int main(int argc, char** argv) {
5     return exec_main<io::middleman>(caf_main, argc, argv);
6 }
```

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.

```
1 int main(int argc, char** argv) {
2     actor_system_config cfg;
3     // read CLI options
4     cfg.parse(argc, argv);
5     // return immediately if a help text was printed
6     if (cfg.cli helptext_printed)
7         return 0;
8     // load modules
9     cfg.load<io::middleman>();
10    // create actor system and call caf_main
11    actor_system system{cfg};
12    caf_main(system);
13 }
```

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.

```
1 class my_config : public actor_system_config {
2 public:
3     my_config() {
4         // ...
5     }
6 };
7
8 void caf_main(actor_system& system, const my_config& cfg) {
9     // ...
10 }
11
12 CAF_MAIN()
```

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

10.1 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.

```
1 void caf_main(actor_system& system) {
2     // ...
3 }
4 CAF_MAIN(mod1, mod2, ...)
```

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

```
1 class my_config : public actor_system_config {
2 public:
3     my_config() {
4         load<mod1>();
5         load<mod2>();
6         // ...
7     }
8 };
```

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

10.2 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.

```
1 class config : public actor_system_config {
2 public:
3     uint16_t port = 0;
4     std::string host = "localhost";
5     bool server_mode = false;
6
7     config() {
8         opt_group{custom_options_, "global"}
9         .add(port, "port,p", "set port")
10        .add(host, "host,H", "set host (ignored in server mode)")
11        .add(server_mode, "server-mode,s", "enable server mode");
12    }
13 };
```

The line `opt_group{custom_options_, "global"}` adds the “global” category to the config parser. The following calls to `add` then append 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`.

CAF prefixes all of its default CLI options with `caf#`, except for “help” (`--help`, `-h`, or `-?`). The default name for the INI file is `caf-application.ini`. Users can change the file name and path by passing `--caf#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). CAF reads `true` and `false` as boolean, numbers as (signed) integers or `double`, “-”-enclosed characters as strings, and ‘-’-enclosed characters as atoms (see §3.2). 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.

```
1 ; This file shows all possible parameters with defaults.
2 ; Values enclosed in <> are detected at runtime unless defined by the user.
3
4 ; when using the default scheduler
5 [scheduler]
6 ; accepted alternative: 'sharing'
7 policy='stealing'
8 ; configures whether the scheduler generates profiling output
9 enable-profiling=false
10 ; can be overridden to force a fixed number of threads
11 max-threads=<number of cores>
12 ; configures the maximum number of messages actors can consume in one run
13 max-throughput=<infinite>
14 ; measurement resolution in milliseconds (only if profiling is enabled)
15 profiling-ms-resolution=100
16 ; output file for profiler data (only if profiling is enabled)
17 profiling-output-file="/dev/null"
18
19 ; when loading io::middleman
20 [middleman]
21 ; configures whether MMs try to span a full mesh
22 enable-automatic-connections=false
23 ; accepted alternative: 'asio' (only when compiling CAF with ASIO)
24 network-backend='default'
25 ; sets the maximum number of consecutive I/O reads per broker
26 max-consecutive-reads=50
27 ; heartbeat message interval in ms (0 disables heartbeating)
28 heartbeat-interval=0
29
30 ; when loading riac::probe
31 [probe]
32 ; denotes the hostname or IP address to reach the Nexus
33 nexus-host=""
34 ; denotes the port where the Nexus actor is published
35 nexus-port=0
```


10.3 Adding Custom Message Types

CAF is designed with distributed systems in mind. Hence, all message types must be serializable and need a platform-neutral, unique name. Using a message type that is not serializable via a free function `serialize` causes a compiler error. Developers that use CAF for concurrency only can suppress this error by whitelisting non-serializable message types using the macro `CAF_ALLOW_UNSAFE_MESSAGE_TYPE`:

```
1 #define CAF_ALLOW_UNSAFE_MESSAGE_TYPE(type_name) \
2     namespace caf { \
3     template <> \
4     struct allowed_unsafe_message_type<type_name> : std::true_type {}; \
5 }
```

CAF serializes objects by calling `serialize(proc, x, 0)`, where the data processor `proc` is either a serializer or a deserializer. The third parameter is a `const unsigned int`, which is never evaluated by CAF. The parameter exists for source compatibility with `Boost.Serialize`. As an introductory example, we use the following POD type `foo`.

```
1 struct foo {
2     std::vector<int> a;
3     int b;
4 };
```

To make `foo` serializable, we implement a free function `serialize`. Serializers provide `operator<<`, while deserializers provide `operator>>`. Both types also allow `operator&` to allow users to write a single function covering loading and storing, as shown below.

```
1 template <class Processor>
2 void serialize(Processor& proc, foo& x, const unsigned int) {
3     proc & x.a;
4     proc & x.b;
5 }
```

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

```
1 class config : public actor_system_config {
2 public:
3     config() {
4         add_message_type<foo>("foo");
5     }
6 };
7
8 void caf_main(actor_system& system, const config&) {
```

If loading and storing cannot be implemented in a single function, users can query whether the processor is loading or storing as shown below.

```
1 template <class T>
2 typename std::enable_if<T::is_saving::value>::type
3 serialize(T& out, const foo& x, const unsigned int) {
4 template <class T>
5 typename std::enable_if<T::is_loading::value>::type
6 serialize(T& in, foo& x, const unsigned int) {
```

10.4 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 § 9.2.

11 Type-Erased Tuples and Messages

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

11.1 RTTI and Type Numbers

All builtin types in CAF have a non-zero 16-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 `std::pair<uint16_t, const std::type_info*>`. The first value is the 16-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.

11.2 Class `type_erased_tuple`

Types

<code>rtti_pair</code>	<code>std::pair<uint16_t, const std::type_info*></code>
------------------------	---

Observers

<code>bool empty()</code>	Returns whether this message is empty.
<code>size_t size()</code>	Returns the size of this message.
<code>rtti_pair type(size_t pos)</code>	Returns run-time type information for the nth element.
<code>void save(serializer& x)</code>	Writes the tuple to x.
<code>void save(size_t n, serializer& x)</code>	Writes the nth element to x.
<code>const void* get(size_t n)</code>	Returns a const pointer to the nth element.
<code>std::string stringify()</code>	Returns a string representation of the tuple.
<code>std::string stringify(size_t n)</code>	Returns a string representation of the nth element.
<code>bool matches(size_t n, rtti_pair)</code>	Checks whether the nth element has given type.

Modifiers

<code>void* get_mutable(size_t n)</code>	Returns a mutable pointer to the nth element.
<code>void load(deserializer& x)</code>	Reads the tuple from x.

11.3 Class message

Observers

<code>bool empty()</code>	Returns whether this message is empty.
<code>size_t size()</code>	Returns the size of this message.
<code>const void* at(size_t p)</code>	Returns a const pointer to the element at position <code>p</code> .
<code>const T& get_as<T>(size_t p)</code>	Returns a const ref. to the element at position <code>p</code> .
<code>bool match_element<T>(size_t p)</code>	Returns whether the element at position <code>p</code> has type <code>T</code> .
<code>bool match_elements<Ts...>()</code>	Returns whether this message has the types <code>Ts...</code> .
<code>message drop(size_t n)</code>	Creates a new message with all but the first <code>n</code> values.
<code>message drop_right(size_t n)</code>	Creates a new message with all but the last <code>n</code> values.
<code>message take(size_t n)</code>	Creates a new message from the first <code>n</code> values.
<code>message take_right(size_t n)</code>	Creates a new message from the last <code>n</code> values.
<code>message slice(size_t p, size_t n)</code>	Creates a new message from <code>[p, p + n)</code> .
<code>message slice(size_t p, size_t n)</code>	Creates a new message from <code>[p, p + n)</code> .
<code>message extract(message_handler)</code>	See § 11.5.
<code>message extract_opts(...)</code>	See § 11.6.

Modifiers

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

11.4 Class `message_builder`

Constructors

<code>(void)</code>	Creates an empty message builder.
<code>(Iter first, Iter last)</code>	Adds all elements from range <code>[first, last)</code> .

Observers

<code>bool empty()</code>	Returns whether this message is empty.
<code>size_t size()</code>	Returns the size of this message.
<code>message to_message()</code>	Converts the buffer to an actual message object.
<code>append(T val)</code>	Adds <code>val</code> to the buffer.
<code>append(Iter first, Iter last)</code>	Adds all elements from range <code>[first, last)</code> .
<code>message extract(message_handler)</code>	See § 11.5.
<code>message extract_opts(...)</code>	See § 11.6.

Modifiers

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

11.5 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:

```
1 auto msg = make_message(1, 2.f, 3.f, 4);
2 // remove float and integer pairs
3 auto msg2 = msg.extract({
4     [](float, float) { },
5     [](int, int) { }
6 });
7 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.

11.6 Extracting Command Line Options

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

```
1 int main(int argc, char** argv) {
2     uint16_t port;
3     string host = "localhost";
4     auto res = message_builder(argv + 1, argv + argc).extract_opts({
5         {"port,p", "set port", port},
6         {"host,H", "set host (default: localhost)", host},
7         {"verbose,v", "enable verbose mode"}
8     });
9     if (! res.error.empty()) {
10         // read invalid CLI arguments
11         cerr << res.error << endl;
12         return 1;
13     }
14     if (res.opts.count("help") > 0) {
15         // CLI arguments contained "-h", "--help", or "-?" (builtin);
16         cout << res.helptext << endl;
17         return 0;
18     }
19     if (! res.remainder.empty()) {
20         // res.remainder stores all extra arguments that weren't consumed
21     }
22     if (res.opts.count("verbose") > 0) {
23         // enable verbose mode
24     }
25     // ...
26 }
27
28 /*
29 Output of ./program_name -h:
30
31 Allowed options:
32   -p [--port] arg   : set port
33   -H [--host] arg   : set host (default: localhost)
34   -v [--verbose]    : enable verbose mode
35 */
```

12 Group Communication

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

```
1 actor_system system;
2 std::string group_module = ...;
3 std::string group_id = ...;
4 auto grp = system.groups().get("local", "foo");
5 scoped_actor self{system};
6 self->join(grp);
7 self->send(grp, "test");
8 self->leave(grp);
```

12.1 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.

12.2 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"`.

12.3 Remote Groups

Remote groups are available only if a middleman (see § 14) is loaded.

Calling `system.middleman().publish_local_groups(port, addr)` makes a group 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, other nodes can get a handle for that group by using the “remote” module: `system.middleman().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.

13 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')`.

For example, `send(my_pool, sys_atom::value, put_atom::value, worker)` adds `worker` to `my_pool`.

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.

13.1 Dispatching Policies

A dispatching policy is a functor with the following signature:

```
1 using uplock = upgrade_lock<detail::shared_spinlock>;
2 using policy = std::function<void (uplock& guard,
3                                   const actor_vec& workers,
4                                   mailbox_element_ptr& ptr,
5                                   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.

```
1 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.

```
1 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.

```
1 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 redispach messages.

```
1 using join = function<void (T&, message&)>;  
2 using split = function<void (vector<pair<actor, message>>&, message&)>;  
3 template <class T>  
4 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.

Part II

I/O Library

14 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 § 10). Users can include `"caf/io/all.hpp"` to get access to all public classes of the I/O module.

14.1 Class `middleman`

Observers

<code>size_t heartbeat_interval()</code>	Returns the heartbeat interval in milliseconds.
<code>bool enable_automatic_connections()</code>	Returns whether middlemen try to establish a full mesh.

Modifiers

<code>uint16 publish(T, uint16, cchar*, bool)</code>	See § 14.2.
<code>void unpublish(T x, uint16)</code>	See § 14.2.
<code>actor remote_actor(string, uint16)</code>	See § 14.2.
<code>T typed_remote_actor(string, uint16)</code>	See § 14.2.
<code>T spawn_broker(F fun, ...)</code>	See § 15.
<code>T spawn_client(F, string, uint16, ...)</code>	See § 15.
<code>T spawn_server(F, uint16, ...)</code>	See § 15.

14.2 Publishing and Connecting

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

```
1 template <class T>
2 uint16_t publish(T x, uint16_t port,
3                 const char* in = nullptr,
4                 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.

```
1 template <class T>
2 uint16_t 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.

```
1 actor remote_actor(std::string host, uint16_t port);
```

```
2
3 template <class ActorHandle>
4 ActorHandle typed_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` or `typed_remote_actor`:

```
1 // node A
2 auto ping = spawn(ping);
3 system.middelman().publish(ping, 4242);
4
5 // node B
6 auto ping = system.middelman().remote_actor("node A", 4242);
```

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.

15 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.

15.1 Spawning Brokers

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

```
1  template <spawn_options Os = no_spawn_options,
2          class F = std::function<void(broker*)>, class... Ts>
3  typename infer_handle_from_fun<F>::type
4  spawn_broker(F fun, Ts&&... xs);
5
6  template <spawn_options Os = no_spawn_options,
7          class F = std::function<void(broker*)>, class... Ts>
8  typename infer_handle_from_fun<F>::type
9  spawn_client(F fun, const std::string& host, uint16_t port, Ts&&... xs);
10
11 template <spawn_options Os = no_spawn_options,
12         class F = std::function<void(broker*)>, class... Ts>
13 typename infer_handle_from_fun<F>::type
14 spawn_server(F fun, uint16_t port, Ts&&... xs);
```

The function `spawn_broker` simply spawns a broker. The convenience function `spawn_client` spawns a broker and immediately connects it to given host and port. Finally, `spawn_server` immediately adds an acceptor for the given port to the new broker.

15.2 Class broker

```
1 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).

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

Writes data to the output buffer.

```
1 void flush(connection_handle hdl)
```

Sends the data from the output buffer.

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

Spawns a new broker that takes ownership of given connection.

```
1 size_t num_connections()
```

Returns the number of open connections.

```
1 void close(connection_handle hdl)
2 void close(accept_handle hdl)
```

Closes a connection or acceptor.

15.3 Broker-related Message Types

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

```
1 struct new_connection_msg {
2     accept_handle source;
3     connection_handle handle;
4 };
```

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

```
1 struct new_data_msg {
2     connection_handle handle;
3     std::vector<char> buf;
4 };
```

New incoming data is transmitted to the broker using messages of type `new_data_msg`. The raw bytes can be accessed as buffer object of type `std::vector<char>`. The amount of data, i.e., how often this message is received, can be controlled using `configure_read` (see 15.2). It is worth mentioning that the buffer is re-used whenever possible. This means, as long as the broker does not create any new references to the message by copying it, the middleman will always use only a single buffer per connection.

```
1 struct connection_closed_msg {
2     connection_handle handle;
3 };
```

```
1 struct acceptor_closed_msg {
2     accept_handle handle;
3 };
```

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

Part III

Appendix

16 Frequently Asked Questions

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

16.1 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`:

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

17 Common Pitfalls

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

17.1 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:

```
1 message_handler wrong = (  
2     [](int i) { /*...*/ },  
3     [](float f) { /*...*/ }  
4 );
```

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

```
1 message_handler ok1{  
2     [](int i) { /*...*/ },  
3     [](float f) { /*...*/ }  
4 };  
5  
6 message_handler ok2;  
7 // some place later  
8 ok2.assign(  
9     [](int i) { /*...*/ },  
10    [](float f) { /*...*/ }  
11 );
```

17.2 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.

17.3 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.

17.4 Sharing

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

18 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).

```
1 #include <random>
2 #include <chrono>
3 #include <cstdlib>
4 #include <iostream>
5
6 #include "caf/all.hpp"
7 #include "caf/io/all.hpp"
8
9 using namespace caf;
10 using std::endl;
11
12 void caf_main(actor_system& system) {
13   for (int i = 1; i <= 50; ++i) {
14     system.spawn([i] (blocking_actor* self) {
15       aout(self) << "Hi there! This is actor nr. "
16         << i << "!" << endl;
17       std::random_device rd;
18       std::default_random_engine re(rd());
19       std::chrono::milliseconds tout{re() % 10};
20       self->delayed_send(self, tout, 42);
21       self->receive(
22         [i, self](int) {
23           aout(self) << "Actor nr. "
24             << i << " says goodbye!" << endl;
25         }
26       );
27     });
28   }
29 }
30
31 CAF_MAIN()
```

19 Migration Guides

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

19.1 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.

19.2 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, `#include "cppa/actor.hpp"` becomes `#include "caf/actor.hpp"`. Further, the convenience header to get all parts of the user API is now `"caf/all.hpp"`. The networking has been separated from the core library. To get the networking components, simply include `"caf/io/all.hpp"` and use the namespace `caf::io`, e.g., `caf::io::remote_actor`.

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

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

```
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.

```
1 accept_handle, connection_handle, publish, remote_actor,  
2 max_msg_size, typed_publish, typed_remote_actor, publish_local_groups,  
3 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`

19.3 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`.

19.4 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.

19.5 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 § 11.6).

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 § 4.3).

19.6 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.

19.7 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 § 4.1). Likewise, most functions in namespace `caf::io` are now member functions of `middleman` (see § 14). 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.

Actor handles now implement a *non-null* assumption. Therefore, `invalid_actor` and the default constructor for all actor handles have been removed. Invalid handles can still be created by using `unsafe_actor_handle_init` for delaying proper initialization. Calling `destroy(x)` on handle releases its reference and allows developers to break cycles manually. Passing an invalid handle (i.e. constructed unsafe or destroyed) to any function in CAF is undefined behavior.

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