Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Master Thesis

Dominik Charousset

libcppa – An actor library for C++ with transparent and extensible group semantic

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Dominik Charousset

libcppa – An actor library for C++ with transparent and extensible group semantic

im Studiengang Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Thomas C. Schmidt
Zweitgutachter: Prof. Dr. rer. nat. Friedrich Esser

Abgegeben am 16.01.2012

Dominik Charousset

Thema der Masterarbeit
libcppa – An actor library for C++ with transparent and extensible group semantic
Stichworte
Aktormodell, C++, Publish/Subscribe, Nebenläufigkeit, Verteilte Systeme
Kurzzusammenfassung
Eine effiziente Nutzung paralleler Hardware setzt eine nebenläufige Ausführbarkeit von Programmen zwingend voraus. Nebenläufige Software mit Hardware-nahen Primitiven wie Threads und Mutexen zu implementieren ist komplex und fehleranfällig. Das Aktormodell ersetzt solche Kommunikation, die von Shared Memory Segmenten manipuliert durch explizite, nachrichtenbasierte Kommunikation. Dabei eignet es sich sowohl zur Implementierung nebenläufiger, als auch verteilter Software. Eine leichtgewichtige Aktormodell-Implementierung, die alle Aktoren in einem ausreichend dimensionierten Thread-Pool ausführt, kann dabei deutlich effizienter sein als eine äquivalente, Thread-basierte Anwendungen. Wir präsentieren in dieser Arbeit `libcppa`, eine Aktormodell-Implentierung für `C++`, die das Aktormodell um eine Semantik für Publish/Subscribe orientierte Gruppenkommunikation erweitert und damit die Entwicklung nebenläufiger und verteilter Anwendungen auf einem hohen Abstraktionslevel unterstützt. Unsere Ergebnisse zeigen, dass das Skalierungsverhalten von `libcppa` vergleichbar mit etablierten Implementierungen des Aktormodells ist.

Dominik Charousset

Title of Master Thesis
libcppa – An actor library for C++ with transparent and extensible group semantic
Keywords
C++, actor model, publish/subscribe, concurrency, distributed systems
Abstract
Parallel hardware makes concurrency mandatory for efficient program execution. However, writing concurrent software is challenging, especially with low-level synchronization primitives such as threads and locks in shared memory environments. The actor model replaces implicit communication by sharing with an explicit message passing mechanism. It applies to concurrency as well as distribution, and a lightweight actor model implementation that schedules all actors in a properly pre-dimensioned thread pool can outperform equivalent thread-based approaches. We build `libcppa`, an actor library with modular support for group semantics that is compliant to the new `C++` standard. By adding a publish/subscribe oriented group communication to the actor model, we support the development of scalable and efficient concurrent as well as distributed systems at a very high level of abstraction. Results indicate that `libcppa` competes mature implementations of the actor model.

Contents

1 Introduction **1**
- 1.1 The Actor Model . . . 3
- 1.2 Overview of this Work . . . 3
- 1.3 Organization of the Thesis . . . 4

2 Message-Oriented Programming **5**
- 2.1 Message Passing . . . 5
  - 2.1.1 Synchronous Message Passing . . . 5
  - 2.1.2 Asynchronous Message Passing . . . 6
  - 2.1.3 Inversion of Control . . . 6
- 2.2 Publish/Subscribe . . . 6
  - 2.2.1 Related Software Patterns . . . 7
  - 2.2.2 IP Multicast . . . 7
- 2.3 Messages and Patterns . . . 8
- 2.4 Actor Systems . . . 9
  - 2.4.1 Message Processing . . . 10
  - 2.4.2 Behavior of Actors . . . 10
  - 2.4.3 Monitoring of Actors and Fault Propagation . . . 10
  - 2.4.4 Group Communication . . . 11

3 Related Work **13**
- 3.1 Erlang . . . 13
  - 3.1.1 Actor Creation . . . 13
  - 3.1.2 Message Processing . . . 14
  - 3.1.3 Name Service for Actors . . . 14
  - 3.1.4 Fault Tolerance and Process Management . . . 15
- 3.2 Scala . . . 16
  - 3.2.1 Scala Actors Library . . . 17
  - 3.2.2 Akka . . . 18
- 3.3 Kilim . . . 18
- 3.4 Retlang . . . 19
- 3.5 Theron . . . 19

4 Design of `libcppa` **20**
- 4.1 Design Goals . . . 20
  - 4.1.1 Ease of Use . . . 20
  - 4.1.2 Scalability . . . 20
  - 4.1.3 Distribution Transparency . . . 21

4.2 Reference Counting Garbage Collection Using Smart Pointers 21
4.2.1 Base Class for Reference Counted Objects 23
4.2.2 Copy-On-Write 23
4.3 Designing an Actor Semantic for C++ 25
4.3.1 Keywords and Operators 25
4.3.2 Syntax Extension 26
4.3.3 Semantic of Send and Receive Statements 27
4.4 Unified Messaging for Groups and Actors 29
4.5 Actors 30
4.5.1 actor Interface 31
4.5.2 local_actor Interface 32
4.5.3 Implicit Conversion of Threads to Actors 33
4.5.4 Cooperative Scheduling of Actors 33
4.6 Event-Based Actors 34
4.6.1 Stacked and Non-Stacked Actor Behavior 34
4.6.2 Actors as Finite-State Machines 35
4.7 Messages 36
4.7.1 Copy-On-Write Tuples 37
4.7.2 Atoms 38
4.8 Group Interface 40
4.9 Serialization 41
4.9.1 Uniform Type Information 41
4.9.2 Announcing User-Defined Types 42
4.10 Network Transparency 43
4.10.1 Actor Addressing 43
4.10.2 Middle Men and Actor Proxies 43
4.10.3 Publishing Actors and Connect to Remote Actors 45

5 Implementation of libcppa 46
5.1 Actor Semantic as Internal Domain-Specific Language for C++ 46
5.1.1 Atoms 46
5.1.2 Receive Statement and Pattern Matching 47
5.1.3 Receive Loops 58
5.1.4 Send Statement 59
5.1.5 Emulating The Keyword self 60
5.2 Mailbox Implementation 62
5.2.1 Spinlock Queue 63
5.2.2 Lock-Free Queue 64
5.2.3 Cached Stack 65
5.2.4 Choosing an Algorithm 66

5.3 Actors . . . 67
5.3.1 Spawning Actors . . . 67
5.3.2 Abstract Actor . . . 67
5.3.3 Thread-Mapped Actors . . . 70
5.3.4 Cooperatively Scheduled Actors . . . 70
5.3.5 Event-Based Actors . . . 72
5.4 Groups . . . 74
5.4.1 Local Group Module . . . 74
5.4.2 A Use Case Example . . . 76
5.5 Serialization . . . 77
5.5.1 Uniform Type Name . . . 77
5.5.2 Announcement of User-Defined Types . . . 78
5.6 Middle Man . . . 81
5.6.1 Addresses of Middle Men . . . 81
5.6.2 Post Office . . . 81
5.6.3 Mailman . . . 81

6 Measurements and Evaluation **83**
6.1 Measuring the Overhead of Actor Creation . . . 84
6.2 Measuring Mailbox Performance in N:1 Communication Scenario . . . 86
6.3 Measuring Performance in a Mixed Scenario . . . 88
6.4 Measurement Summary . . . 90

7 Conclusion & Outlook **91**

1 Introduction

Sequential programming languages such as `C`, `C++` or Java do not provide concurrency semantics. They were developed before the "Multi-Core Revolution" started and thus originally aimed at single-core processor machines. Because all mainstream operating system allow starting multiple *threads of execution* within a process, threading libraries were developed on top of existing languages. One motivation to implement threading was to keep user interfaces responsive at ongoing background work, another to implement multi-client servers without starting multiple processes with `fork`. Real hardware concurrency makes threading mandatory: “(...) multi-core processors make concurrency an essential ingredient of efficient program execution” (Haller and Odersky, 2009).

Dealing with concurrency is challenging, especially in shared memory environments, where all threads share the same process-wide memory. This easily leads to *race conditions*, if two or more threads access a memory location in parallel. The performance and scalability of hand-written synchronization to avoid race conditions depends on the implementation strategy; coarse-grained locking is simple, but could lead to queuing and scalability issues, whereas fine-grained locking increases scalability but also complexity and error-proneness (e.g., due to lock order[1]). Furthermore, increased complexity and the many sources of errors in multithreaded applications increase the complexity of testing as well: “The main difficulty of multiprogramming is that concurrent activities can interact in a time-dependent manner which makes it practically impossible to locate programming errors by systematic testing” (Hansen, 1973).

Referential transparency, pure functions and immutable values ease parallelization, because race conditions cannot occur and isolated computations can be freely spread among processors to scale in a multi-core environment. Pure functions that do not have internal state increase locality of data, because such functions primarily accesses their own stack and local, temporary objects. Locality of data increases the possibility of cache hits because the accessed data is not spread among many memory locations. Sharing immutable data only reduces the possibility of “false sharing” and uses system resources more efficiently. Processors always fetch a *cache line*, e.g., 64 byte on an Intel i7 processor, to minimize access to the RAM module since such access is time-expensive compared to access a local cache. This optimization strategy yields good performance results as long as a program iterates over a memory location or accesses memory locations within a certain range. But this also can

[1] Lock order is a mutex-based synchronization protocol for a Fine-Grained Locking strategy. Assuming a program has two shared objects A and B, each guarded by its own mutex. If a thread wants to atomically modify both of them, it has to acquire both locks. This locking order must be equal for all threads. A deadlock occurs, if any thread locks B before A, while other threads lock A first. This is simple in examples with only a few shared objects, but easily leads to complex source code that is hard to maintain.

lead to so-called “false sharing” whenever two threads are accessing memory that fits into one cache line. The cache must be updated every time another processor modified data in the cached memory region. Thus, two threads running on separate cores mutually invalidate their cache on each single write operation, even if both threads access distinct data. Such false sharing causes a serious performance impact, because a processor has to frequently access the RAM rather than its local cache.

Still, concurrent software components often need to share (or exchange) information, resp. data. In shared memory environments, this communication is implicit (shared states) and implemented with low-level primitives (locks, threads, condition variables, etc.), which is error-prone and requires expert knowledge. One needs to know about compiler and processor reordering as well as language internals to implement efficient concurrency with low-level synchronization primitives. Optimization strategies such as double-checked locking can lead to memory corruptions, if one does not have the required expert knowledge for a correct implementation (Meyers and Alexandrescu, 2004). If information needs to be shared among a group of participants, it is even more difficult to implement a solution that is both thread-safe and scalable. There are basically two approaches to raise the level of abstraction to hide unsafe, shared memory: Make shared memory transactional, either in software (like in the `Clojure` programming language (Hickey, 2011)) or hardware, or use explicit communication facilities, such as message passing instead of shared memory.

Independent of the selected approach, we need high-level concepts and libraries to describe concurrency and to enable average programmers to write both safe[2] and scalable code. Functional programming languages always propagated stateless functions, free of side effects, operating on immutable data structures. Thus, functional programming languages are easier to execute in parallel. However, the five most popular programming languages in practice are the imperative programming languages Java, `C`, `C++`, `C#` and `Objective-C` (TIOBE software, 2012). The problem these languages face to make use of hardware concurrency, we believe, is not the imperative programming style. It is the wrong level of abstraction and sharing of mutable stateful objects. In this work, we address the `C++` programming language. It is one of the most relevant languages, especially in the UNIX and open source communities, with large available code bases, but lacks a sufficient level of abstraction for concurrency and distribution.

[2] Free of race conditions, deadlocks and lifelocks.

1.1 The Actor Model

The Actor model is a formalism describing concurrent entities, “actors”, that communicate by asynchronous message passing (Hewitt et al., 1973). An actor can send messages to addresses of other actors it knows and can create new actors. Actors do not share state and are executed concurrently.

Because Actors are self-contained and do not rely on shared resources, race conditions are avoided by design. The message passing communication style also allows network transparency and thus applies to both concurrency, if actors run on the same host on different cores/processors, and distribution, if actors run on different hosts, connected via the network.

The traditional actor model does not address group communication. However, it seems natural to combine the actor model with publish/subscribe-based group communication. Allowing actors to join and leave groups allows an extensible and technology-transparent group communication semantic. A group can be implemented as a local list of actors to generalize in-process event handling, or as distributed system using a network layer transport technology such as IP multicast.

1.2 Overview of this Work

The Actor model inspired several implementations in computer science, either as basis for languages, e.g., Erlang, or as libraries/frameworks, e.g., Scala Actor Library, to ease development of concurrent software. However, low-level primitives are still widely used in `C++` and other mainstream languages because there are no implementations of a high-level concurrency abstraction available. In `C++`, the standardization committee added threading facilities and synchronization primitives to the next standard (ISO, 2011), but did neither address the general issues of multiprogramming, nor distribution, nor message-oriented group communication mechanisms.

This work targets at concurrency, distribution and message-oriented group communication in `C++`. We design and implement a library for `C++`, called `libcppa`, with an internal Domain-Specific Language (DSL) approach to provide high-level abstraction. This library is benchmark tested against other implementations and offers an extensible group communication API. Results show, that we are able to compete established actor model implementations.

1.3 Organization of the Thesis

This thesis is organized as follows. We will introduce message-oriented programming in general, as well as publish/subscribe first. Then, we will give an introduction to actor systems and definitions of essential features. The related work section will focus on other implementations of the actor model, e.g., in the functional programming languages Erlang and Scala. Our design section will start with design goals and our approach to reference counting garbage collection. Then, we will augment the `C++` programming language with an abstract syntax extension. We will use this syntax to specify the semantic and behavior of our actor model implementation. This abstract syntax is used in our implementation section as reference for our internal DSL. The implementation section is divided in subsections, one for each software component of `libcppa`. We will further measure and evaluate our implementation by comparing our implementation with other, matured actor model implementations. A conclusion section will summarize evaluation and implementation results and finally will give an outlook on future work.

2 Message-Oriented Programming

Message-oriented programming emphasizes interaction between software components (objects) while object-oriented programming tends to focus on the objects themselves. Mutable stateful objects accessed in parallel need to synchronize access to member functions and internal state, causing serious issues and complexity in concurrent systems. By using messaging rather than granting access to member functions, concurrency issues such as race conditions are avoided by design since only the object itself has access to member function implementations.

2.1 Message Passing

The message passing paradigm follows the assumption that software components are self-consistent and isolated. The state of an independent software component cannot be manipulated from the outside. It changes its state or behavior based on messages it receives and it is free to ignore messages or to process other messages first. In this way, the message passing paradigm leads to loosely coupled software systems.

The characteristics of message passing systems depend on the message transfer technology in use and how messages are handled by receivers. Transfer technologies vary in reliability and message ordering and could be synchronous or asynchronous. Receive methods include blocking, non-blocking and callback-based approaches.

2.1.1 Synchronous Message Passing

Synchronous messaging is a joint process of sender and receiver. The sender of the message is blocked until the receiver has processed the given message. Results can be passed directly to the sender of a message, since there is a feedback channel between the two peers.

Synchronous communication requires reliable transmission of messages, since an undetected loss would always lead to a deadlock at the sender. A deadlock also occurs, if a peer sends a message to itself if the communication uses a mutex-based synchronization protocol, because a software component cannot call `wait()` and `notify()` at the same time.

2.1.2 Asynchronous Message Passing

Asynchronous communication decouples senders and receivers and requires buffering. Usually, a FIFO ordered message queue is used. As there is no implicit feedback channel between sender and receiver, an answer by the receiver must be passed in an explicit response message. Alternatively, synchronization primitives such as futures and promises could be used in shared memory environment.

Asynchronous communication can be unreliable and messages might arrive out of order, depending on the transport technology in use.

2.1.3 Inversion of Control

Callback-based message handling often has the issue of *Inversion of Control* (IoC). The callback, a registered function or object that is called whenever a new message arrives, has to implement all of the receiver's functionality and the receiver is executed in the context of the sender. Thus, the classical control flow of procedural programming is inverted from a receiver's point of view. IoC is common in event-driven software architectures such as GUI frameworks.

2.2 Publish/Subscribe

The publish/subscribe paradigm describes message passing systems that decouple senders and receivers. Messages are *published* to a *channel* rather than sent directly to receivers. Thus, a sender (*publisher*) does not know addresses of receivers (*subscribers*). Subscribers may be *notified* by the channel if a message was published.

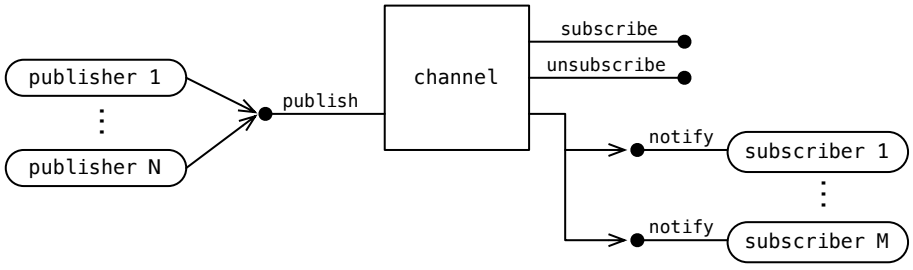

Figure 1: Simple publish/subscribe system

Figure 1 illustrates a simple publish/subscribe system. The channel is an N:M group com-

munication interface between N publishers and M subscribers. Each published message is replicated M times.

Many variants of the publish/subscribe paradigm exist, some allow subscribers to receive only a pre-filtered subset of messages such as topic-based, content-based and type-based publish/subscribe (Eugster et al., 2003).

2.2.1 Related Software Patterns

The publish/subscribe paradigm gave rise to software design patterns such as the observer pattern (Gamma et al., 1995) and the signal/slot model provided by various `C++` libraries, e.g., `Qt` or `boost`.

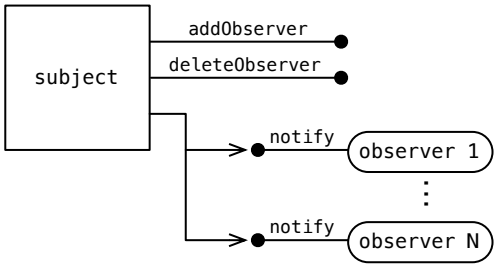

Figure 2: Observer pattern

Figure 2 shows a software design based on the observer pattern. The subject merges the role of publisher and channel. Thus, it models an 1:N communication. Illustrations of similar patterns such as the signal/slot model are analogous.

2.2.2 IP Multicast

IP multicast is a real-world example for a fully distributed publish/subscribe system. There is no well-known host serving as channel as in client/server based approaches. Instead, the routing algorithms establish network paths between publishers and subscriber. A *join* (subscription) to a *multicast address* (the address of a distributed channel) signals the network to add the sender of such a subscription to the network path. A *leave* (unsubscription) removes the sending host from the network path.

IP Any Source Multicast (ASM) (Deering, 1989) is a very general publish/subscribe system as shown in Figure 1. A host receives all messages sent to channels (groups) it has subscribed to.

IP Source-Specific Multicast (SSM) (Holbrook and Cain, 2006) adds a source filtering mechanism to the publish/subscribe system. A host subscribes to a channel for a particular sender S and does not receive messages from the channel that are not published by S.

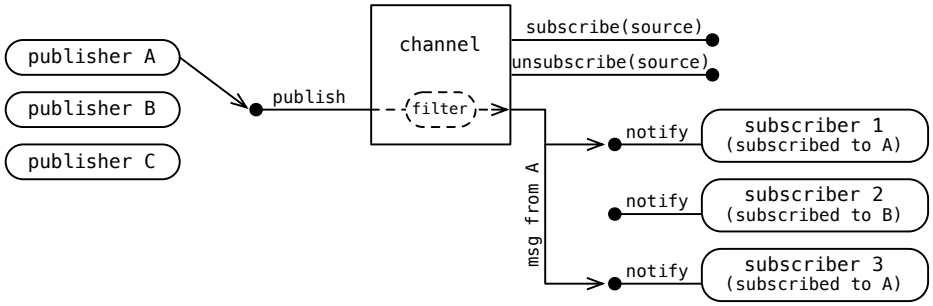

Figure 3: Source-specific publish/subscribe system

Figure 3 illustrates the observable behavior of IP SSM. A subscription includes a channel and a source. A subscriber then receives only messages from the channel with the specified source address. Thus, the channel models 1:N communication between a source and any number of subscribers.

2.3 Messages and Patterns

A message is an object of communication consisting of *data* and *metadata*. Metadata is mandatory for message processing. The type of message data is an inevitable metadata. There are use cases for messages with empty data, e.g., signaling of timeouts, but a receiver always has to evaluate the metadata of a message first in order to process it. However, metadata is not necessarily transferred along with the data, e.g., a *typed channel* could transmit only the raw data since all participants of a communication know the metadata beforehand.

A *message pattern*, henceforth referred to as *pattern*, is an expression querying both data and metadata of messages that evaluates to either *true* or *false* for any given message. Patterns ease message processing for participants of a message passing system to use untyped communication channels. An untyped communication channel does not restrict messages to particular types. Patterns are a convenient way for a participant to specify what messages it accepts. Furthermore, patterns also can bind parts of the message data to variables for further computations.

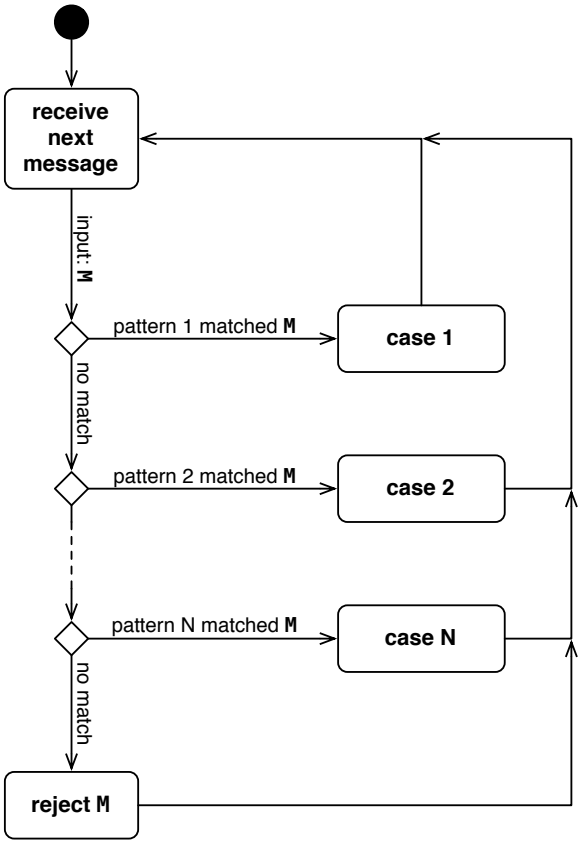

Figure 4: Receive loop using patterns

Figure 4 shows a typical receive loop using pattern matching. An incoming message M is accepted if one of the predefined patterns does match M, otherwise, M is rejected. M is processed at most once since no further patterns are evaluated after the first match.

2.4 Actor Systems

The actor model is a refinement of the message passing paradigm. It is explicitly designed for both parallel and distributed systems using independent software entities responding to messages they receive. Actors communicate by asynchronous messaging. Pattern matching is not mandatory to implement actor systems, but it has proven useful and very effective for message processing (Armstrong, 1996).

2.4.1 Message Processing

A distinguishing feature of an actor system is its message processing implementation. There are two main categories.

Mailbox-based message processing
: Incoming messages are buffered at the actor in a *mailbox* in FIFO order. The message processing, however, is not necessarily FIFO ordered. During receive, an actor iterates over messages in its mailbox, beginning with the first. An actor is free to skip messages in its mailbox. A message remains in the mailbox until it is eventually processed and removed from the mailbox as part of its consumption. An actor will be blocked until a new message arrives or an optional timeout occurs after it reached the end of its mailbox or it is empty.

Event-based message processing
: Actors register a function or function object as callback to the runtime system. The runtime system calls the receive callback of an actor with the messages in arrival order. Thus, an actor cannot prioritize messages since it is executed in an event loop.

2.4.2 Behavior of Actors

The *behavior* of an actor denotes its response to the *next* incoming message. This response includes sending messages to other actors, creation of new actors and defining the subsequent behavior (Agha, 1986, p. 12). Actor systems using mailbox-based message processing and pattern matching can define a behavior as a partial function (Haller and Odersky, 2009). This enables the runtime system to skip messages automatically during receive, if the behavior of an actor is undefined for a given message.

2.4.3 Monitoring of Actors and Fault Propagation

The actor model allows actors to monitor other actors (Hewitt et al., 1973). This is in particular necessary for building fault-tolerant distributed systems since errors are more likely in distributed than in single-processed software. Distributed systems inherently include more sources of errors.

Complete actor systems provide monitoring and linking for actors (Armstrong, 2003). A terminating actor sends a *down* message to all its monitors and an *exit* message to all its

links. Both message types contain the exit reason of the terminating actor. An actor receiving an exit message will per default terminate with the same exit reason unless it is a *normal* exit reason. An actor terminated with exit reason set to *normal* finished execution without error. This default can be explicitly overridden by an actor to *trap* exit messages and to treat them manually. *Down* messages are processed like any other message.

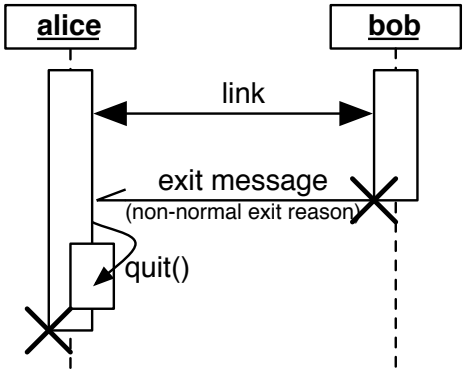

Figure 5: Linking of actors

Figure 5 shows two linked actors *Alice* and *Bob*. *Bob* sends an exit message to all of its links as part of its termination with non-normal exit reason, causing *Alice* to finish execution as well.

Links allow for strong coupling of actors. In fact, linked actors form a subsystem in which errors are propagated through exit messages.

2.4.4 Group Communication

Group communication combines several semantics such as *Anycast* for load balanced or replicated services, *(selective) Broadcast* for rendezvous processes or contacting unknowns, *Con(verge)cast* for data aggregation or scalable many-to-one communication and publish/-subscribe, resp. *Multicast*, for scalable (m)any-to-many communication.

We focus on the publish/subscribe paradigm, since it is most widely used for group communication in practice. However, existing publish/subscribe implementations such as IP multicast and `D-Bus`[3] individually require specific code access not applicable for message oriented programming as intended in actor systems.

[3] `D-Bus` is an inter-process communication system for UNIX with support for system-wide events and services: `http://dbus.freedesktop.org`

The lack of a general publish/subscribe API makes it difficult for a developer to implement future-proof software since his decision on technologies is based on deployment at coding time. Furthermore, the lack of an embedded and extensible group semantic for actor systems hinders the use of group communication technologies for distributed actors.

3 Related Work

In this section, we will focus on Erlang and Scala, because they provide established and practice-proven implementations of the actor model. Afterwards, we will introduce a few libraries providing message-oriented concurrency available in Java, `C#` and `C++`.

3.1 Erlang

Erlang is a functional, dynamically typed programming language that was designed for programming large-scale distributed systems. It is developed since 1985 (Armstrong, 1997). Its concurrency abstraction is based on lightweight processes and is an integral part of the language rather than implemented as a library. Erlang processes are actors, because they are isolated computational entities communicating only via asynchronous, network-transparent message passing with each other. As such, Erlang is a de facto implementation of the actor model and can be looked upon as reference for upcoming implementations including `libcppa`.

3.1.1 Actor Creation

New processes, resp. actors, are created with the function `spawn` that takes another function or lambda expression[4] as argument and returns the identifier of the newly created process. Process identifiers are addresses of actors that are needed for message passing. The function `spawn` could also create the process on any connected host with an optional argument. After creation, there is no difference between local and remote actors. All operations on processes are network transparent.

[4] A lambda expression is an anonymous function that can be stored in variables or passed to other functions as argument.

3.1.2 Message Processing

Erlang implements mailbox-based message processing as described in Section 2.4.1. Messages are send using the operator "!" followed by a tuple. The following example sends the tuple $(1,2,3)$ to the actor identified by `Pid`. This operation is network transparent. `Pid` is either the address of a local actor or of a remote actor.

```
1 Pid ! { 1, 2, 3}
```

The blocking *receive statement* is a selection control mechanism that evaluates a partial function defining the actors behavior as described in 2.4.2. The syntax of a receive statement reads

```
1 receive
2   Pattern1 [when Guard1] -> Expression1;
3   Pattern2 [when Guard2] -> Expression2;
4   ...
5   after Time -> Expression3
6 end
```

A *guard* is an optional part of a pattern. The receive statement blocks the calling actor either until an optional timeout occurs, or until a message was matched by one of the given patterns.

The following example shows a receive with three patterns. The first matches only messages with a single integer value. The second only matches messages with two elements, whereby the first element is "hello". The third pattern does match any message. If no message is received within 50 milliseconds, "Received nothing" will be printed.

```
1 receive
2   X when is_integer(X) -> io:format("Received an integer~n");
3   {hello, What} -> io:format("Received: hello ~s~n", [What]);
4   Y -> io:format("Received: ~p~n",[Y])
5   after 50 -> io:format("Received nothing~n")
6 end
```

3.1.3 Name Service for Actors

Actors need addresses of other actors to address messages. An actor could learn new addresses from messages it receives, or obtained a list of addresses during creation. Erlang

also provides a name service for actors to address actors providing system-wide services, e.g., printing, with a well-known names.

However, it is impracticable to give all actors addresses of all "well-known actors" providing system-wide services such as printing. Erlang provides a name service to register well-known actors system wide.

The name service is implemented by the two functions `register` and `whereis`. It is also possible to use names rather than process identifiers to send a message. Thus, `whereis` is implicitly called when needed as the following example shows.

```
1 Pid = spawn(...),
2 register(my_actor, Pid),
3 my_actor ! "Hello Actor".
```

3.1.4 Fault Tolerance and Process Management

Erlang was intended to build fault-tolerant, distributed systems and therefore provides built-in capabilities to monitor processes corresponding to Section 2.4.3:

> "A frequent assumption made when writing Erlang software is that any Erlang process may unexpectedly die (...). The runtime system provides a mechanism to notify selected processes of the fact that a certain other process has terminated; this is realized by a special message that arrives in the mailbox of processes that are specified to monitor the vanished process." (Earle et al., 2005, p. 27)

Besides monitoring of processes, Erlang provides hierarchical process management with *supervision trees*.

> "A supervision tree is a tree of processes. The upper processes (supervisors) in the tree monitor the lower processes (workers) in the tree and restart the lower processes if they fail." (Armstrong, 2007, p. 351)

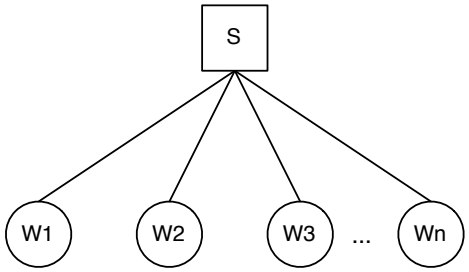

Figure 6: Supervision tree

Fig. 6 shows a simple supervision tree S with workers $W1$ to Wn. Workers are numbered consecutively in order of creation. Supervision trees have three restart strategies: *one_for_one*, *one_for_all* and *rest_for_one*.

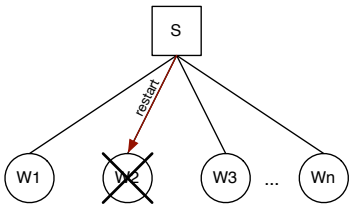

Figure 7: *one_for_one* strategy

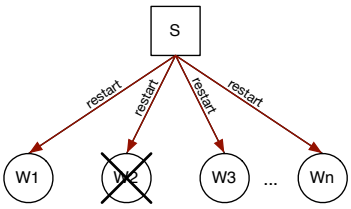

Figure 8: *one_for_all* strategy

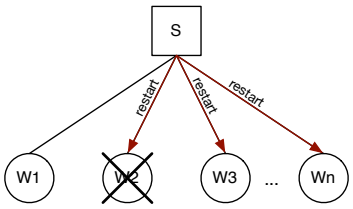

Figure 9: *rest_for_one* strategy

Fig. 7 illustrates the *one_for_one* strategy. A worker, $W2$ in this particular case, is restarted if it fails. Other workers are unaffected. The *one_for_all* strategy shown in Fig. 8 restarts all workers if a worker terminates unexpectedly. Fig. 9 visualizes the *rest_for_one* strategy. Only the failing worker and all workers created after it are restarted if a worker terminates unexpectedly.

Supervision trees are also capable of loop detection, e.g., if workers are continuously restarted and always terminate with the same error (Armstrong, 2003).

3.2 Scala

Scala is a multi paradigm programming language developed since 2001. It combines functional and object-oriented programming (Odersky, 2011) and compiles to Java bytecode. Scala is fully compatible to the Java programming language. It is possible to use and inherit Java classes and interfaces in Scala and vice versa. Scala does not implement the actor model as a language feature, but the language provides pattern matching and its syntax is explicitly designed to ease development of domain-specific languages. Thus, a well de-

signed library implementing the actor model could provide a syntax as good as a language extension.

Though, there is a downside of library solutions providing the actor model. It is impossible to ensure isolation of actors. A programmer could easily share states among actors since Scala does provide access to shared memory, leading to issues such as race conditions. It is recommended, though, by Scala programming guides to use immutable messages and not to share state among concurrent entities. This issue is inevitable in library solutions for languages allowing shared memory access, since the compiler cannot enforce isolation of actors.

3.2.1 Scala Actors Library

The Scala actors library is part of the standard library distribution since version 2.1.7 (2006). Actors are defined by inheriting the class `scala.actors.Actor` and overriding the method `act()`. An actor calls either `react` or `receive` with a partial function using pattern matching. Both functions implement a mailbox-based message processing. The difference between the two functions is that `react` uses an event-based implementation and never returns to the caller. The partial function argument of `react` is used as event handler. This event handler can be replaced by nesting `react` statements to change the behavior of the actor. Thus, `react` implements event-based programming without inversion of control (Haller and Odersky, 2006). An actor using the function `receive` is running in its own thread. A threaded actor needs more system resources compared to an event-based actor. It does have its own representation in the kernel of the operating system and allocates its own stack. Thus, event-based actors usually yield better performance due to fewer context switches and less allocated system resources.

The following example shows a singleton actor using `react`:

```
1  case class IntMessage(x: Int)
2
3  object ReactActor extends Actor {
4    override def act() {
5      loop {
6        react {
7          case i : Int => println("Received integer: " + i)
8          case IntMessage(x) => println("Received IntMessage: " + x)
9          case "quit" => exit() // terminate execution of this actor
10       }
11     }
```

```
12      println("This line is never executed (loop never returns)")
13    }
14  }
15
16  ReactActor.start()
```

The function `loop` recursively calls the following `react` statement. Such a loop also could be achieved by recursively call `act` from inside the `react` block. Line 12 is unreachable since neither `loop` nor `react` returns. Without using `loop` statement in line 5, this example would receive exactly one message.

3.2.2 Akka

Akka is an event-driven implementation of the actor model (Typesafe Inc., 2011). It provides an event-based message processing as described in 2.4.1. Akka also includes a Java API for actor programming.

In addition, Akka implements software transactional memory to safely share a datastructure across actors with begin/commit/rollback semantics. Transactions are atomic, consistent and isolated. This deviates from other actor model implementations that either prohibit shared states (Erlang) or at least recommend isolation of actors (Scala actors library).

3.3 Kilim

Kilim is an actor framework for Java (Srinivasan, 2011). It implements its own, lightweight threads for actors and enforces actor isolation by static analysis. Kilim uses a post compiler called *weaver* to modify the java bytecode based on annotations[5] as shown in Figure 10. The weaver ensures that a message does have at most one owner at a time and that an actor cannot access stack or instance fields of another actor (Srinivasan and Mycroft, 2008).

[5] An annotations is metadata that can be added to classes, methods, variables, parameters and packages in Java (Gosling et al., 2005). This metadata can be embedded in bytecode as part of the compiled class file and is accessible at run-time using Java reflections.

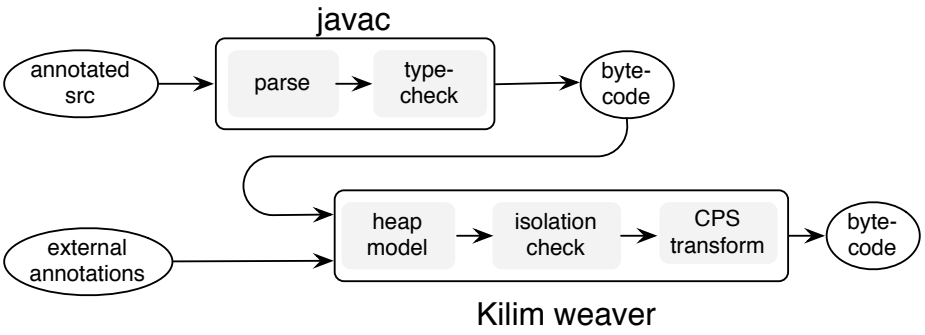

Figure 10: `javac` output post-processed by Kilim `weaver` (Srinivasan and Mycroft, 2008)

The weaver also performs a continuation passing style (CPS) transformation for all functions of an actor. CPS allows to intercept a function at any point and resume its computation later on (Reynolds, 1972). Thus, Kilim is able to perform preemptive scheduling of actors in its own userspace scheduler.

3.4 Retlang

Retlang is a `C#` library for message-based concurrency in the style of the actor model. It uses cooperatively scheduled *fibers* (Microsoft, 2011). Fibers are lightweight units of execution scheduled in userspace. Fibers are an implementation of coroutines (Conway, 1963) that allow suspending and resuming on predefined locations. Consequently, scheduling is controlled by Retlang rather than by the operation system.

3.5 Theron

Theron (Mason, 2011) is currently the only existing actor library for `C++` besides `libcppa`. It is a rudimentary implementation that neither does supports group communication, nor provides network transparency, thus, does not support distribution. Theron implements event-based message processing without mailbox, using registered member functions as callbacks, as discussed in Section 2.4.1. Actors are scheduled in a thread pool and defined by inheriting from the class `Theron::Actor`. The class `Theron::Receiver` allows threads to receive messages from actors. Monitoring or linking of actors is not supported.

4 Design of libcppa

Before presenting the actual software design of libcppa, we will start by outlining our general design goals and present the software concept for the reference counting garbage collection used in our library.

4.1 Design Goals

Our aim is to add an actor semantic to C++ that enables developers to build efficient concurrent and distributed software. Though libcppa is influenced by functional programming languages such as Erlang and Scala, it should not mimic a functional programming style and provide an API that looks familiar to C++ developers.

4.1.1 Ease of Use

A user of libcppa ideally does not need to know about any internals of the library implementation. We decided to use an internal Domain-Specific Language approach, because we want to raise the level of abstraction in C++, while requiring as little glue code[6] as possible. Unusual function names, code conventions and semantics are best avoided. libcppa should adopt concepts and best practices of the standard template library (STL) to reduce both complexity and learning effort for developers using it. In particular, this includes iterator interfaces and functor[7] arguments whenever functionality should be added to libcppa.

4.1.2 Scalability

All modern operating systems provide threading libraries to allow parallel execution of independent computations within a process. Threads are scheduled in the system kernel. This makes creation and destruction of threads heavyweight because they rely on system calls and acquire system resources such as thread state in the OS scheduler, stack and signal

[6] Glue code is source code that does not implement any functionality of an application, but serves the purpose to fulfill interface requirements of a library. A common example is to define a class D extending a class or interface C to override one virtual member function with the intended functionality F and then to create a single object of D because a function of a library L requires an instance of C.

[7] A functor is anything providing the operator "()". This includes function pointers, lambda expressions and hand-written classes using operator overloading.

stack. Thus, short-living threads do not scale well since the effort for creation and destruction outweighs the benefit of parallelization.

Scalability in the context of multi-core processors requires splitting application logic into many independent tasks that could be executed in parallel. An actor is a representation of an independent task. This makes lightweight creation and destruction of actors mandatory.

4.1.3 Distribution Transparency

The Actor model applies to both concurrency and distribution. Thus, the network layer of `libcppa` should manage all network connections and hide complexity and implementation details for serialization and deserialization of messages to make network communication transparent for developers. In particular, the representation of an address of an actor should use a common interface for both local running and remote actors.

4.2 Reference Counting Garbage Collection Using Smart Pointers

`C++` does not provide a garbage collector. Nevertheless, garbage collection can be implemented by using reference counting and smart pointers to make development easier and less error-prone.

A smart pointer is an object that behaves like a pointer in `C++` by providing the operator "`->`" and "`*`" with the semantics of raw pointers except that the pointed object is destroyed in the destructor of a smart pointer if there is no more reference to it.

A reference count is *intrusive* if it is stored in the object itself. Thus, a class has to provide a reference count to be used with an *intrusive smart pointer*.

There are several implementations of smart pointers available for `C++`. The most basic kind is the `unique_ptr` implementation of the standard template library. A `unique_ptr` object *owns* the object it points to, meaning that no other pointer to it is allowed. Therefore, copying a unique pointer is prohibited and only move semantics[8] are allowed. Moving the content of one unique pointer to another transfers ownership atomically and guarantees that

[8] A move statement shifts the content of one object to another, e.g., after the statement `a=b` is `a` a copy of `b` (`a==b` returns `true`), while the statement `a=std::move(b)` transfers state and content of `b` to `a`. Thus, `a` represents the old value of `b` and `b` is empty.

only one unique pointer at a time is pointing to a particular object. The pointed object is destroyed in the destructor of `unique_ptr` if the internal pointer is not `nullptr`.

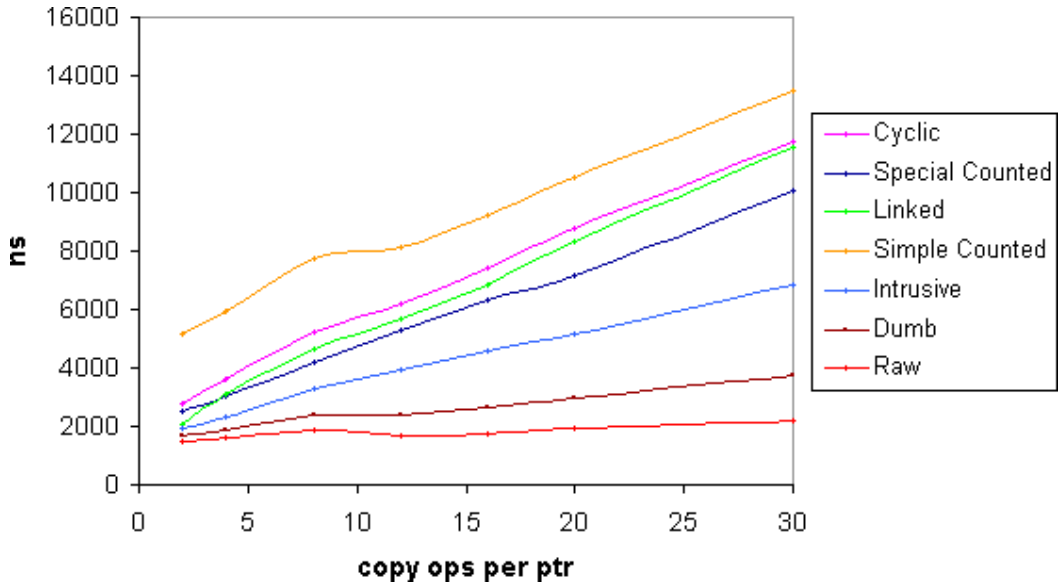

Figure 11: Smart pointer timings for GCC (Abrahams et al., 2001))

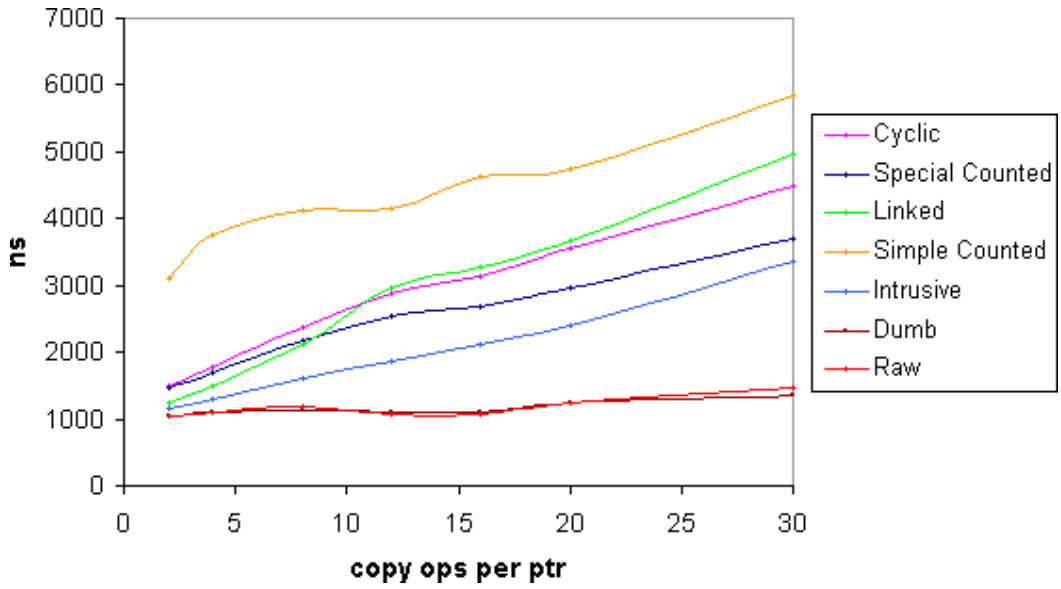

Figure 12: Smart pointer timings for MSVC (Abrahams et al., 2001))

Figures 11 and 12 compare several smart pointer implementations (Abrahams et al., 2001) compiled with GCC (GNU compiler collection) and MSVC (Microsoft® Visual C++). The results are compared to raw pointers as well as an implementation without reference counting

at all (*Dumb*). *Intrusive* uses a reference count stored in the object itself. *Simple Counted* uses a separate, heap allocated reference count. *Linked* is an implementation using an intrusive linked list in the object where all smart pointer instances pointing to it are stored. An object has no more references if the list is empty. *Special Counted* is a non-intrusive reference count similar to *Simple Counted* but uses a special purpose allocator. *Cyclic* uses an intrusive reference count and stores each pointed object in a list for cycle detection.

Intrusive pointers outperform any other smart pointer implementation. The downside of this approach is, that it affects client code by forcing classes to provide a reference count. This downside is negligible if the decision to use intrusive reference counting is made at the beginning of the design phase and all classes of a library are designed with reference counting garbage collection in mind.

4.2.1 Base Class for Reference Counted Objects

The class `ref_counted` is the base for all reference counted classes. It provides an atomic, lock-free reference count to allow a subclass to be used with intrusive smart pointers.

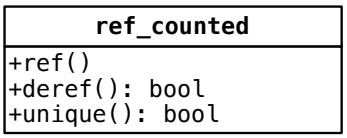

Figure 13: Base class for reference counting

Figure 13 shows the interface of `ref_counted`. The member function `ref` increases the reference count by one. `deref` decreases the reference count by one and returns *true* if there are still one or more references left to the object. The member function `unique` returns *true* if there is *exactly one* reference to the object.

`libcppa` provides two smart pointer implementations designed along with `ref_counted`: `intrusive_ptr` and `cow_ptr`. The first is a straightforward smart pointer implementation that mimics a pointer with automatic destruction but no additional functionality. The second implements a *copy-on-write* (CoW) strategy.

4.2.2 Copy-On-Write

Copy-on-write (CoW) is an optimization strategy to minimize copying overhead with call-by-value semantic. The message passing implementation of `libcppa` uses a call-by-value

semantic. This would cause multiple copies of a message if it is send to more than one actor. A message can be shared among several actors as long as all participants only demand read access. An actor copies the shared object when it demands write access and is only allowed to modify its own copy. Thus, race conditions cannot occur and each object is copied only if necessary.

The template class `cow_ptr` implements a smart pointer with CoW semantic. It requires an object O of type T to provide the same member functions `ref_counted` does and an addition `copy` member function returning a new instance with the same type and value as O. Using the copy constructor of T to create such a copy would fail if O is of a derived type. The member function `unique` of O is called whenever non-const dereferencing[9] is requested. A new copy using the corresponding member function is created, if an object is not unique. Afterwards, `cow_ptr` decreases the reference count of O and points to its copy O' that can be safely modified.

[9]Non-const dereferencing allows to modify an object, while const dereferencing grants read access only, e.g., assuming two pointer to an integer: `int i=0; int* pi=&i; int const* ci=&i;`, the expression `*pi` is a non-const dereferencing, allowing to modify the value of the pointed object: `*pi=2`. The expression `*ci` is a const dereferencing and the expression `*ci=2` would cause a compiler error.

4.3 Designing an Actor Semantic for C++

In this section, we extend the C++ programming language with message-passing capabilities conforming to the actor model. We will introduce new keywords as well as new statements to the syntax of C++ and discuss the semantics of the introduced extensions afterwards. This language extension is mapped to an internal Domain-Specific Language (DSL) later on.

4.3.1 Keywords and Operators

`self`
: Identifies the current actor, similar to *this* in an object-oriented context.

`receive`
: Begins a receive statement.

`after`
: Defines a timeout within a receive statement.

`anything`
: Wildcard for pattern expressions matching any number of types.

`<-`
: This binary operator sends a message given as right operand to an actor given as left operand.

4.3.2 Syntax Extension

The syntax rules type-id, identifier and literal are according to the C++ syntax definitions, base-stmt refers to the original statement definition (ISO, 2011, p. 1190, p. 22, p. 1178 and 1184). We extend[10] the C++ syntax with the following Extended Backus–Naur Form (EBNF) rules.

```
nondigit      = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h"
              | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p"
              | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x"
              | "y" | "z" | "_"
              | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H"
              | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P"
              | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X"
              | "Y" | "Z"
nonzero-digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
              | "9"
digit         = "0" | nonzero-digit
whitespace    = " " | <tab> | <newline>
_             = { whitespace }
pattern-brick = "anything"
              | ( type-id [ _ identifier ]
                          [ _ "=" _ ( identifier | literal ) ] )
              | ( identifier | literal )
pattern       = pattern-brick { _ "," _ pattern-brick }
rcv-case      = "(" _ pattern _  ")" _
                "{" _ { statement } _ "}"
duration      = nonzero-digit { digit } ( "sec" | "msec" )
timeout       = "after" _ duration _ "{" _ { statement } _ "}"
behavior-stmt = "{" _
                  ( ( rcv-case _ { rcv-case _ } [ timeout ] )
                    | timeout )
                _ "}"
receive-stmt  = "receive" _ behavior-stmt
send-stmt     = identifier _ "<-" _ "{" _
                       ( identifier | literal ) _
                       { "," _ ( identifier | literal ) _ }
                  "}" _ ";"
statement     = receive-stmt | send-stmt | base-stmt
```

[10] digit, nondigit and nonzero-digit are already identically defined and thus not added. They are defined here for reasons of clarity and comprehensibility.

4.3.3 Semantic of Send and Receive Statements

We introduced syntactic rules to extend the C++ programming language with a receive statement and a send statement. A message is a tuple of values with size ≥ 1. The send statement is basically a simplification and could be actually expressed by a function as well. The receive statement, on the other hand, adds the required message-passing semantic to the language. A receive statement is a selection control mechanism comparable to the switch statement. An incoming message is compared to the patterns *in declaration order* and the corresponding code of the *first* matching pattern is executed. Further patterns are not evaluated, since receive has an *at most once* semantic. This is a mailbox-based message processing approach as described in Section 2.4.1. An example:

```
1  receive {
2    (int = 1, double val) {
3      // ... case 0 ...
4    }
5    (int val1, double val2) {
6      // ... case 1 ...
7    }
8    after 0sec {
9      // ... timeout ...
10   }
11 }
```

This receive statement evaluates a `behavior-stmt` that creates a partial function f as discussed in Section 2.4.2. f is undefined for a given message x if none of the given patterns matches x. The partial function type of f behaves according to the following pseudo code.

```
1  class f:
2    is_defined_for(x):
3      return    x.size() == 2
4             && x.typeAt(0) == typeid(int)
5             && x.typeAt(1) == typeid(double)
6    handle_message(x):
7      assert(is_defined_for(x))
8      if x.at(0) == 1:
9        // ... case 0 ...
10     else:
11       // ... case 1 ...
12   handle_timeout():
13     // ... timeout ...
```

Note that `case 0` would never be evaluated if we would swap the declaration order.

Receive statements block the calling actor until it received a message matched by one of the given patterns or an optional timeout occurs. An empty receive block would always be a deadlock and is thus prohibited by the syntax definition. A timeout with a zero-time duration as in line 11 of the example above can be used to obtain a non-blocking receive. An actor immediately executes the timeout if no message in its mailbox was matched.

It is possible to nest receive statements, as illustrated by the following example program.

```
1  #include <iostream>
2  using std::cout;
3  using std::endl;
4  int main()
5  {
6    self <- { 0.0 };         // msg1
7    self <- { 0.0f, 'a' }; // msg2
8    self <- { 'a', 0.0 };   // msg3
9    self <- { 1, 1.0 };     // msg4
10   self <- { 2, 2.0 };     // msg5
11   self <- { 3.0 };        // msg6
12   for (;;)
13     receive {
14       (int = 1, double val) { // alternative: (1, double val)
15         cout << "case1: " << val << endl;
16       }
17       (int val1, anything, double val2) {
18         cout << "case2: " << val1 << ",...," << val2 << endl;
19       }
20       (double val1) {
21         receive { // nested receive
22           (double val2) {
23             cout << "case3: " << val1 << "," << val2 << endl;
24           }
25         }
26       }
27       (float val, anything) {
28         cout << "case4: " << val << ",..." << endl;
29       }
30       (anything) { cout << "case5" << endl; }
31       after 1sec {
32         cout << "timeout" << endl;
33         return 0;
34       }
35     }
36 }
```

The output of the example program will be:

```
case3: 0.0,3.0
case4: 0.0f,...
case5
case1: 1.0
case2: 2,...,2.0
timeout
```

The first message `msg1` is a tuple with a single double element. Thus, it is matched by the pattern in line 20. The nested receive statement accepts single double messages only. Thus, all messages before `msg6` are skipped. `msg2` is a tuple with a `float` and a `char` element. It is matched by the pattern in line 27 that matches any message with a `float` as first element (the wildcard expression `anything` matches any number of additional elements). `msg3` has a `char` as first element, followed by a `double`. It is **not** matched by the pattern in line 17 because a `char` is not an `int` and the runtime is not allowed to implicit convert elements. Thus, it is matched by the pattern in line 30 that matches any message. The two remaining messages `msg4` and `msg5` are both matched by the pattern in line 17 (an `int`, followed by any number of elements and a `double` as last element), but `msg4` is handled by the pattern defined in line 14 because it is declared first.

4.4 Unified Messaging for Groups and Actors

Sending a message to a group of actors should be equal to sending a message to a single actor. In fact, a single actor is nothing else but a list of actors with one element. Thus, the abstract class `group` and the abstract class `actor` are both inheriting the same interface: `channel`.

A channel is a software entity that can receive asynchronous messages. More precisely, messages are *enqueued* to a channel since the sender can assume that its message is stored in the buffer of the asynchronous communication system.

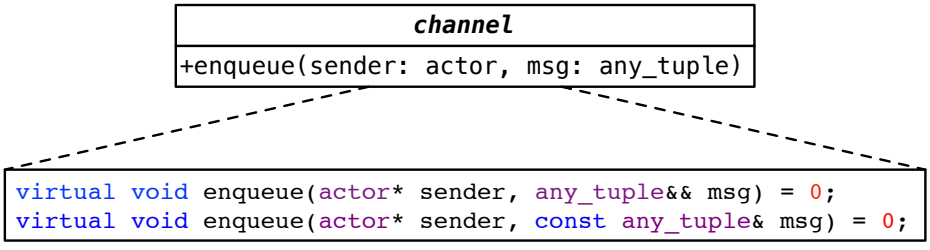

Figure 14: `channel` interface

Figure 14 shows the `channel` interface along with the actual `C++` signatures. The member function `enqueue` is overloaded to allow for either copying or moving a tuple (see Section 4.7.1) to the receiver. The *sender* argument is optional, thus, can be set to `nullptr`. If set, it allows the receiver to reply to a message.

4.5 Actors

An actor is the most fundamental communication primitive. It provides capabilities to receive and send messages. However, there are several kinds of actors as well as several access levels to actors.

The actor interface is essentially split in two. The first interface called `actor` provides all operations needed for linking, monitoring and group subscriptions. The second interface is `local_actor`. It represents the *private* access to an actor since only the actor itself is allowed to dequeue a message from its mailbox. The keyword `self` that we introduced earlier in Section 4.3.1 does have the type `local_actor*`.

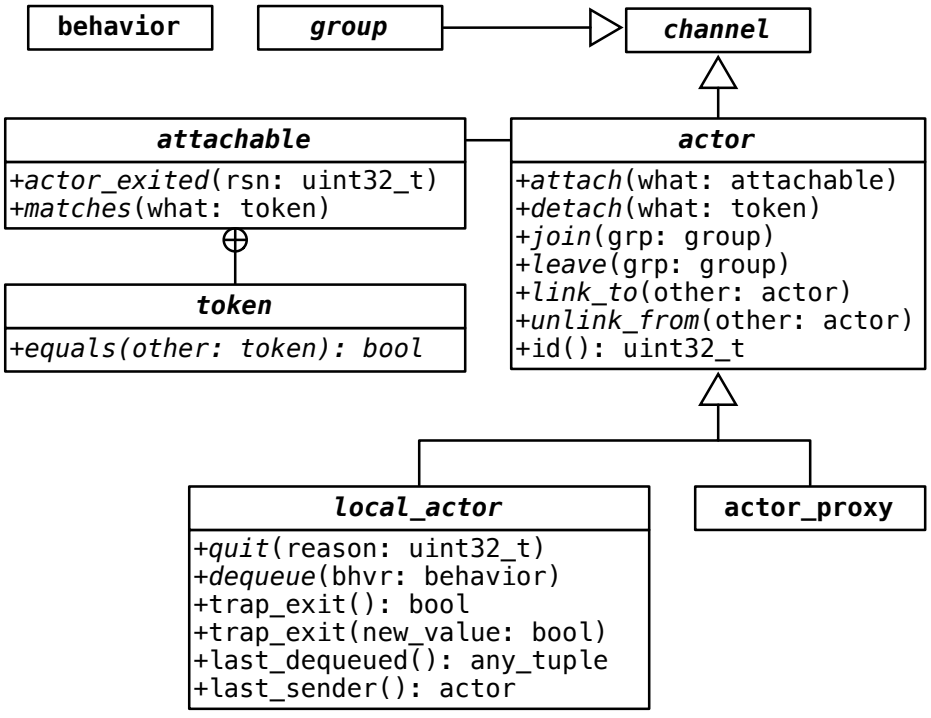

Figure 15: Actor interfaces with inheritance

Figure 15 shows a class diagram containing `channel` and `local_actor` as well as

their inheritance relation. In addition, it shows the class `actor_proxy` that represents an actor running on a different process or node. The interface `attachable` is a utility class describing a handler that is called by the actor it is attached to if it finishes execution. This is particularly useful to monitor actors. It is also possible to *detach* such a handler by using an appropriate implementation of `token`.

4.5.1 `actor` Interface

An implementation of the `actor` interface shall implement all member functions according to the definitions below. In addition, all member functions need to be implemented in a thread-safe manner since other actors are allowed to call any of those member functions in parallel.

`attach(what: attachable)`
Adds atomically *what* to an internal set of $attachable$ instances if the actor did not finished execution yet. Otherwise, $what.actor\_exited(reason)$ is called immediately with *reason* set to the exit reason of the actor. The actor takes ownership of *what*. Thus, it destroys *what* after calling $what.actor\_exited()$. An actor calls all handlers during its exit phase and empties its internal set of $attachable$ instances afterwards.

`detach(what: token)`
Removes each a from the internal set of $attachable$ instances with $a.match(what) = true$ without calling $actor\_exited()$.

`join(grp: group)`
Causes the actor to subscribe the group grp if it did not finished execution yet. Does nothing if the actor already subscribed to grp.

`leave(grp: group)`
Causes the actor to unsubscribe the group grp. Does nothing if the actor does not have a subscription to grp.

`link_to(other: actor)`
Causes this actor to enable a link between itself and $other$. The actor sends an exit message to $other$ immediately if he did finished execution. This member function has no effect if this actor is already linked to $other$.

`unlink_from(other: actor)`
Causes this actor to remove a link between itself and $other$. This member function has no effect if this actor is not linked to $other$.

id(): uint32_t
Returns the identifier of this actor. This identifier is guaranteed to be unique in the actors process.

4.5.2 local_actor Interface

The local_actor interface is only accessible from an actor itself. Thus, all member functions can safely assume that they will never be called in parallel.

quit(reason: uint32_t)
Finishes execution of this actor by throwing an exception of the type actor_exited with *reason*. Before throwing the exception, the actor has to call and destroy all $attachable$ instances in its local set as well as leaving all joined groups and send exit messages to all of its links.

dequeue(bhvr: behavior)
The argument *bhvr* is a representation of a partial function with optional timeout. This member function performs a receive operation according to the pseudo code in Section 4.3.3 with $P = bhvr$.

trap_exit(): bool
Returns $true$ if this actor treats exit messages as ordinary messages that could be received and handled manually. Otherwise, $false$ is returned if this actor finishes execution as soon as it receives an exit message with non-normal exit reason.

trap_exit(new_value: bool)
Changes the way an actor handles exit messages as described for trap_exit(): bool.

last_dequeued(): any_tuple
Returns the message that was last recently dequeued from this actor's mailbox.

last_sender(): actor)
Returns the sender of the last recently dequeued message. It is optional to define a sender for a message, thus, the result could be $nullptr$.

4.5.3 Implicit Conversion of Threads to Actors

An actor semantic needs to be consistent. Therefore, the introduced keyword `self` is not allowed to be invalid or to return $nullptr$. Otherwise, it could not be guaranteed that a receive or send statement never fails. This implies that a non-actor caller, e.g., a thread, is converted to an actor if needed. The example in Section 4.3.3 assumed `main` to be an actor without calling any function beforehand. Thus, the keyword `self` cannot be implemented as a well-known local, thread-local, or global variable similar to `this`. On each access, the runtime system has to verify the value of `self` and must create an instance of `local_actor` if needed.

4.5.4 Cooperative Scheduling of Actors

An actor library needs to schedule actors in an efficient way, as mentioned in Section 4.1.2. An ideal way to ensure *fairness* in an actor system requires preemptive scheduling. A fair system would guarantee that no actor could starve other actors by occupying system resources. However, one needs unrestricted access to hardware or kernel space to implement preemptive scheduling, since it needs hardware interrupts to switch between two running tasks. No operating system can allow unrestricted hardware or kernel space access to a userspace application for obvious reasons.

In general, userspace schedulers cannot implement anything but cooperative scheduling. But there is some design space to make context switching implicit. The two operations each actor uses frequently are sending and receiving of messages. Thus, the library could switch the actor's context back to the scheduler, whenever it sends or receives a message. An ordinary workflow of an actor is receiving messages in a loop and sending messages either as part of its message processing or after computing some result. Thus, interrupting an actor during send would probably interrupt an actor during its message processing, while interrupting it during receive seems natural in this workflow. Furthermore, we need to support interruption during receive since an actor is not allowed to block while its mailbox is empty. Instead, an actor returns to scheduler and is re-scheduled again after a new message arrives to its mailbox.

A thread that uses the keyword `self` to send and receive messages is not affected by the cooperative scheduling. Thus, developers can choose to execute an actor in its own thread if it needs blocking system calls because, this would possibly starve other actors in a cooperative scheduling.

4.6 Event-Based Actors

We introduced an approach to extend C++ with a general actor semantic in Section 4.5. This extension fits seamlessly into C++ concerning program execution and flow control. An actor is *blocked*, or at least allows the scheduler to interrupt its computation during receive and continues execution from the receive block on afterwards. This is the usual behavior of any language construct in a procedural programming language using a *stack*. Therefore, each actor needs to have its own stack in our presented approach. This is not much of an issue in systems consisting of a few hundreds or even a few thousand actors. However, it does become an issue in large-scale actor systems or on platforms with limited memory.

As an example for a current mainstream system: Mac OS X Lion defines the two constants $SIGSTKSZ = 131072$ and $MINSIGSTKSZ = 32768$ in its system headers. $SIGSTKSZ$ is the recommended stack size in bytes and $MINSIGSTKSZ$ is the minimum allowed stack size in bytes. Assuming a system with 500,000 actors, one would require a memory usage of *at least* 15 GB of RAM for stack space only. This would rise up to 61 with the recommended stack size instead in use. This clearly does not scale well for large systems. To reduce memory usage and scale up to large systems, we provide a second, event-based actor approach that is not callback-based.

4.6.1 Stacked and Non-Stacked Actor Behavior

Callback-based message processing cannot prioritize messages since the callback is invoked by the runtime system usually in arrival order, and thus has a semantic different from our mailbox-based actors using the receive statement. Therefore, we use a behavior-based API with mailbox-based message processing.

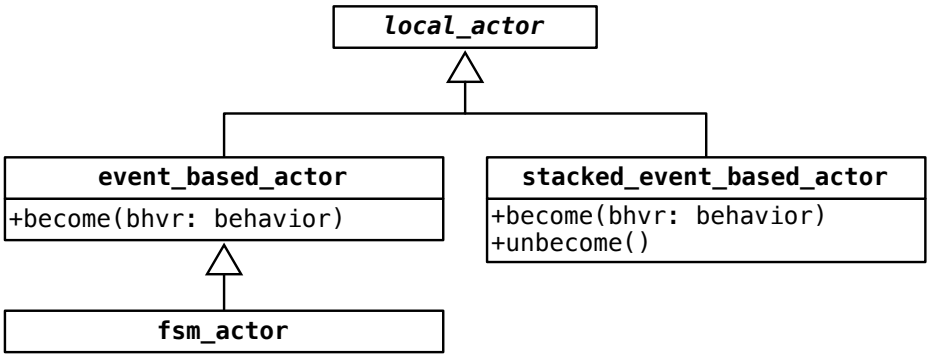

Figure 16: Event-based actor classes

An event-based actor sets its required behavior as a partial function using the `become` member function. This partial function is used until the actor replaces it by invoking `become` again. Thus, both actor implementations in `libcppa` use a behavior-based approach as described in Section 2.4.2.

The class `stacked_event_based_actor` stores previously used partial functions in LIFO order, allowing an actor to return to its previous behavior by using the `unbecome` member function.

4.6.2 Actors as Finite-State Machines

With the `become` semantic, one could model an actor as a special case of finite-state machines by using member variables storing a particular behavior acting as *states*. The class `fsm_actor` should encourage developers to do so. It does not provide any member function but calls `become(init_state)` in its constructor where the member variable *init_state* must be defined by any class derived from `fsm_actor`.

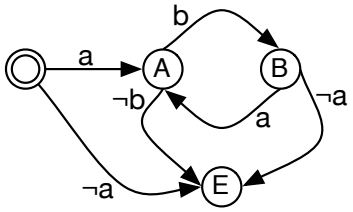

Figure 17: Simple finite-state machine

Figure 17 shows a simple finite-state machine (FSM) providing the states `A`, `B`, and `E` with the transition conditions `a` and `b`. `E` is used as error state if the FSM receives an unexpected token. This FSM is implemented by the following example class.

```
1  class simple_fsm : public fsm_actor {
2      behavior state_a = {
3          ('b') { become(state_b); }
4          (anything) { become(error_state); }
5      };
6      behavior state_b = {
7          ('a') { become(state_a); }
8          (anything) { become(error_state); }
9       };
10     behavior init_state = {
11         ('a') { become(state_a); }
12         (anything) { become(error_state); }
13     };
14     behavior error_state = {
15         (anything) { /* ... */ }
16     };
17 };
```

Providing an individual constructor is not necessary if a class is derived from `fsm_actor`. All states could be declared with non-static member initialization introduced by `C++11` as our example illustrates.

Our approach does not cover non-deterministic state machines. Furthermore, asynchronous message-passing differs from transition-based processing of the mathematical model of finite-state machines, e.g., mailbox-based message processing allows prioritizing of messages. However, finite-state machines are well-known in computer science and have proven useful in designing software systems, since this design pattern leads to small, very well testable software components.

4.7 Messages

The statement `x <- { 1, 2, 3.0 }` enqueues a tuple with three elements to `x`. Usually, a developer does not need to access the underlying tuple implementation. Elements are accessed by pattern matching implicitly, e.g., the statement `receive { (int arg0, int arg1, double arg2) { /*bhvr1*/} }` matches tuples with three elements consisting of two `int` values followed by a `double` value and invokes `bhvr1` with the three tuple elements as arguments. A developer may access the received tuple by calling `self->last_dequeued()`, which returns an object of type `any_tuple`, representing a tuple with no static type information. This allows developers to access elements matched by `anything` that are not accessible otherwise.

4.7.1 Copy-On-Write Tuples

A message could be send to multiple receivers, e.g., the statement `x <- { 1, 2, 3.0 }` has any number of receivers, as described in Section 4.4. Since messages always follow call-by-value semantic, sending a message to multiple actors would require multiple copies of the message. We use a copy-on-write implementation as presented in sec. 4.2.2 to avoid both unnecessary copies and race conditions.

namespace cppa

any_tuple

```
bool empty() const;

const void* at(size_t p) const;

const uniform_type_info& utype_info_at(size_t p) const;

void* mutable_at(size_t p);

template<typename... Args>
any_tuple(const tuple<Args...>& tup);

template<typename... Args>
any_tuple(const tuple_view<Args...>& tup);
```

template<typename...> tuple_view

template<typename...> tuple

```
template<typename... Types>
tuple<Types...> make_tuple(const Types&... args);

template<size_t N, typename... Types>
const Types[N]& get(const tuple<Types...>& tup);

template<size_t N, typename... Types>
Types[N]& get_ref(tuple<Types...>& tup);

template<typename... Types>
tuple_view<Types...> tuple_cast(const any_tuple& tup);

template<size_t N, typename... Types>
const Types[N]& get(const tuple_view<Types...>& tup);

template<size_t N, typename... Types>
Types[N]& get_ref(tuple_view<Types...>& tup);
```

Figure 18: Copy-on-write tuples with C++ signatures

Fig. 18 shows all representations of tuples with C++ signatures. Each non-const access to a tuple representation *detaches* its value if it is shared. All three tuple classes have a copy-on-write pointer to the data but provide different functionality. Note: *Types[N]* is not legal C++

syntax. It is used here as simplification for representing the *N*-th element of the template parameter pack *Types*.

tuple
: Is a one-to-one representation of the internal data. Values are accessed via the free functions `get` and `get_ref`. The difference between those two functions is that `get_ref` grants non-const access. Thus, one has always a local copy that could be mutated after calling `get_ref`.

tuple_view
: Does provide the same code access via `get` and `get_ref` but is not a one-to-one representation of the internal data. A view could hide values of the internal data. For example, `receive { (int arg0, anything, double arg2) {/*bhvr*/} }` accesses the first and the last element of a tuple only. `bhvr` is invoked with a `tuple_view<int,double>` that hides all elements of the original message matched by `anything`.

any_tuple
: Represents a tuple without static types. It can be queried for the number of values as well as the type of each individual value. It grants access to the values via raw, untyped pointers. One can either cast each single element after a runtime type check at runtime or create a `tuple_view` to get a representation of the same data with static type information. A view can access either all data elements of the original `any_tuple` using `tuple_cast` or could be created using a pattern to hide some values.

4.7.2 Atoms

Assume an actor provides a mathematical service for integers. It takes two arguments, performs a predefined operation and returns the result. It cannot determine an operation, such as *multiply* or *add*, by receiving two operands. Thus, the operation must be encoded into the message. The following example compares different approaches to do this.

```
1  #define ADD_REQ 0
2  #define MULTIPLY_REQ 1
3  enum class req { add, multiply };
4  struct add_req { int lhs; int rhs; };
5  struct multiply_req { int lhs; int rhs; };
6  // ...
7  void math_actor() {
8    for (;;) {
9      receive {
10       // approach 1
11       (add_req req) { reply(req.lhs + req.rhs); }
12       (multiply_req req) { reply(req.lhs * req.rhs); }
13       // approach 2
14       (ADD_REQ, int lhs, int rhs) { reply(lhs + rhs); }
15       (MULTIPLY_REQ, int lhs, int rhs) { reply(lhs * rhs); }
16       // approach 3
17       (req::add, int lhs, int rhs) { reply(lhs + rhs); }
18       (req::multiply, int lhs, int rhs) { reply(lhs * rhs); }
19       // approach 4
20       ("add", int lhs, int rhs) { reply(lhs + rhs); }
21       ("multiply", int lhs, int rhs) { reply(lhs * rhs); }
22     }
23   }
24 }
```

The first approach uses *message types*, types that are used for messaging and do not have any other purpose than encapsulating values. Using message types is verbose since one has to create one type for each operation of an actor. Furthermore, the definition of the message types must be available to all participants of a communication. Therefore, developers have to maintain several header files containing message types. The second approach uses symbolic, C-like constants and is obviously the worst solution since the implicit type of the constants is `int`, which easily could lead to misinterpretation. An enumeration type, as used in the third approach, solves the issue of ambiguous type, but a developer would have a similar maintaining overhead as in the first approach, since the enum definition needs to be available to each participant of the communication. The last approach uses string constants. Strings are not a special purpose type, thus, there is still some possibility of misinterpretation. But string constants come with a cost since strings are always allocated on heap and string comparison is more expensive than integer comparison.

The Erlang programming language introduced an approach to use non-numerical constants, so-called *atoms* (Armstrong, 2007), which have an unambiguous, special-purpose type and do not have the overhead of string constants. Atoms fill the gap between fast but inflexi-

ble enums, causing maintaining overhead and expensive but flexible string constants. We decided to not include atoms to our syntax extension since it could be done as well with a user-defined literal in C++. Therefore, we introduced the suffix _atom as shown in the following code snippet.

```
1 receive {
2     (add_atom, int lhs, int rhs) { /*...*/ }
3     (multiply_atom, int lhs, int rhs) { /*...*/ }
4 }
```

Atoms are mapped to integer values at compile time. An implementation shall use a collision-free mapping. Thus, atoms combine performance of integer constants with the flexibility of string constants.

4.8 Group Interface

The class group is a *factory* managing user-defined group modules. Each module implements a group communication technology.

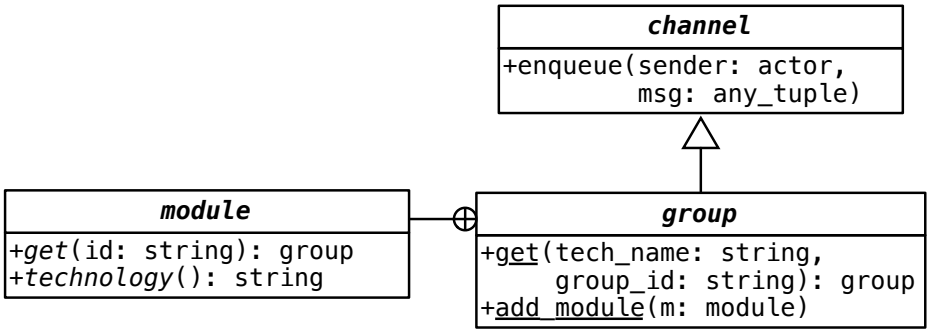

Figure 19: Group and related classes

Figure 19 shows a UML class diagram for the class group and its nested class module which one has to implement in order to add a new group communication technology to a libcppa application. A module has to provide a factory member function group get(string id) that returns a technology-specific group instance for the group identified by id. A group implementation using IP multicast for example would expect a valid IP group address such as ”239.1.2.3“ for IPv4 or ”ff05::1:3“ for IPv6 as id parameter. An implementation using a locator/ID split probably would expect an URI like ”sip://news@cnn.com“. However, groups always behave similar after getting an instance via the factory function get of class group. Actors can join/leave such groups without any knowledge about the underlying technology.

4.9 Serialization

Serialization is a requirement for network transparency. C++ does neither support platform-independent serialization by itself, nor does it support the required runtime type representation, such as reflections to implement serialization. Thus, we need a minimal runtime-type information (RTTI) system that allowing us to implement serialization on top of it.

4.9.1 Uniform Type Information

The C++ programming language does have very minimalist RTTI support. It provides the operator typeid that returns a reference to a type_info object. That object could be compared to other type_info objects and has the two member functions before and name. The first one evaluates the *collating order* for types. The latter returns a string identifying the represented type. However, the format of the returned string is not specified in the C++ standard specification and is thus platform-dependent.

libcppa implements a runtime type information class of its own, since type_info does not provide required functionality such as creation of new instances of the represented type or support for user-defined serialization.

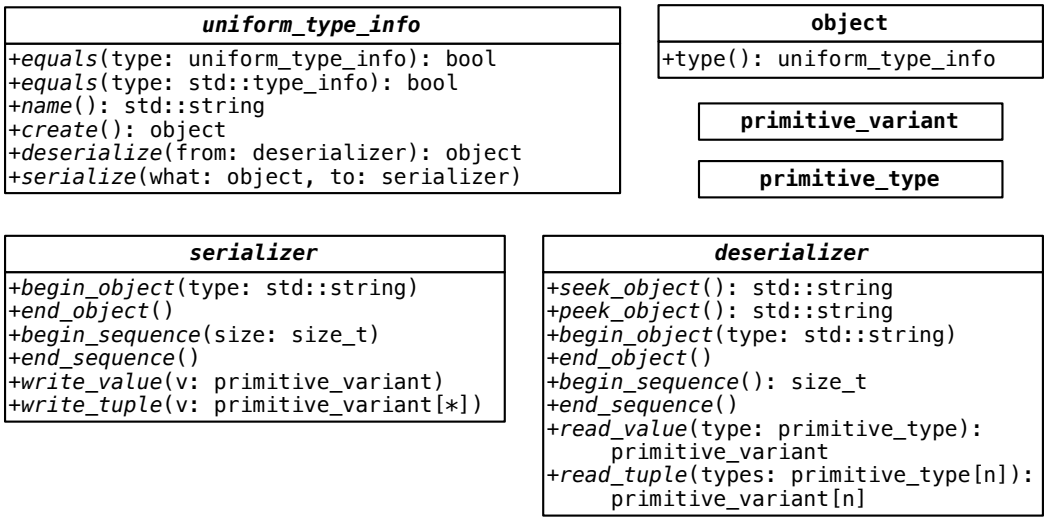

Figure 20: Class diagram containing all serialization related classes

Figure 20 shows the interface uniform_type_info as well as its related classes for serialization and deserialization. The complementary class object represents an instance of any type. It provides access to its value through get or get_ref as shows in Figure 21.

```
namespace cppa

const uniform_type_info* uniform_typeid(const std::type_info&);

template<typename T>
const uniform_type_info* uniform_typeid();

template<typename T>
T& get_ref(object& obj);

template<typename T>
const T& get(const object& obj);
```

Figure 21: Related functions for `unfiorm_type_info` and `object`

Figure 21 also shows the accessor functions for the `uniform_type_info` singletons. Each `type_info` instance does have at most one corresponding `uniform_type_info` instance.

The interfaces `serializer` and `deserializer` have a begin/end protocol to allow for XML-like data representations as well as a binary data format. The `deserializer` interface has two member functions to get the type name of the next object in the underlying data source. The `seek_object` member function returns the type name and modifies the position in the data source to handle a following call to `begin_object`, whereas `peek_object` does not manipulate the read position. The class `primitive_variant` encapsulates any primitive value such as UTF-8, UTF-16 and UTF-32 strings, signed and unsigned integers as well as floating point values. The enumeration `primitive_type` denotes the type of such a primitive value.

4.9.2 Announcing User-Defined Types

The compiler creates a `type_info` instance for each type. However, it is not possible to create the corresponding `uniform_type_info` automatically, neither at compile-time, nor at runtime due to the lack of reflections. Thus, a user has to *announce* all user-defined types for messaging to enable serialization for the particular type by creating a `uniform_type_info` instance for each user-defined type. For trivial types, this can be done semi-automatically by giving pointers to all members of a class, as the following example illustrates.

```
1 struct my_msg_type { int a; float b; };
2 int main() {
3     announce<my_msg_type>(&my_msg_type::a, &my_msg_type::b);
4     // ...
5 };
```

A pair of getter and setter member function pointers could be used, if a type does not allow public access to its members. However, this approach is limited to trivial types. A more complex data structure, such as a tree, still requires provisioning of a hand-written `uniform_type_info` implementation by the developer.

4.10 Network Transparency

Network transparency hides from an actor whether another actor runs on the same process or not. Each operation like linking and sending messages behaves as if the other actor ran in the same process. The only difference between a remote actor and a local actor is its exit reason on a connection error.

4.10.1 Actor Addressing

Each actor needs a unique address so that other actors can send messages to it. This address is not needed for in-process communication, since actors use the `actor` interface to communicate with each other. However, the address is inevitable in distributed systems. We designed the address of an actor to have three parts. The first part uniquely identifies the network node. The second part is the process id given by the operating system. The last part is the ID of an actor itself returned by the member function `uint32_t actor::id() const`.

The node ID is a 160 bit hash of the Ethernet address of the first network device and the UUID (*Universally Unique Identifier*, Leach et al. (2005)) of the root partition of the file system. This ID is always the same for a given node as long as the hardware of the system remains unchanged. A random number can be used instead of the Ethernet address if the host system uses hardware interfaces without Ethernet addresses.

4.10.2 Middle Men and Actor Proxies

The class `actor_proxy` was introduced in Section 4.5. It implements the `actor` interface and forwards all messages it receives to the *middle man* (MM) that created it. The MM is a software component that encapsulates network communication and manages all connections and sockets. Furthermore, it creates instances of `actor_proxy` for known remote actors.

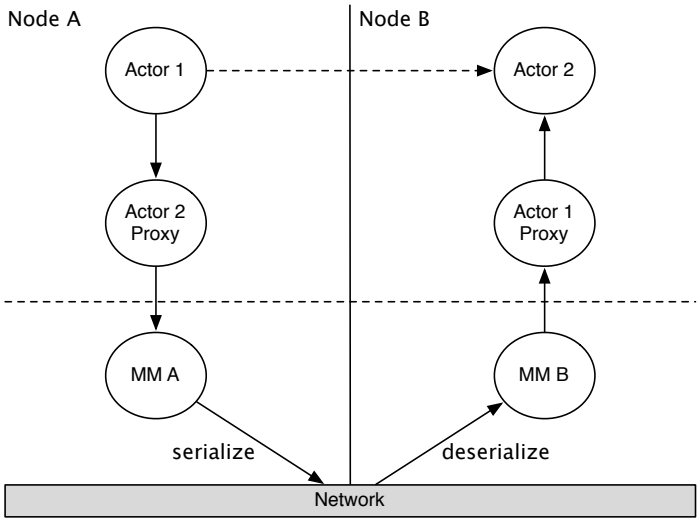

Figure 22: Communication to a remote actor

Figure 22 shows a communication scenario for two actors running on different network nodes. `Actor1` (A1) sends a message to `Actor2` (A2) represented by `Actor2 Proxy` (A2P). A2P forwards the received message to its MM, which serializes the message and sends the serialized byte stream to the MM of node B. The MM of B then creates a proxy for A1 if needed and sends the deserialized message to A2 with sender set to `Actor1 Proxy`. Both middle men are invisible to A1 and A2. The reversed path for the message will be taken if A2 replies to the received message.

Linking of actors slightly differs for an actor (A1) linked to a remote actor (A2). The proxy of A2 sends a special message to its MM whenever it becomes linked to another actor. The MM then sends this message to the MM of the original A2 instance. Finally, A2 becomes linked to the proxy of A1, too.

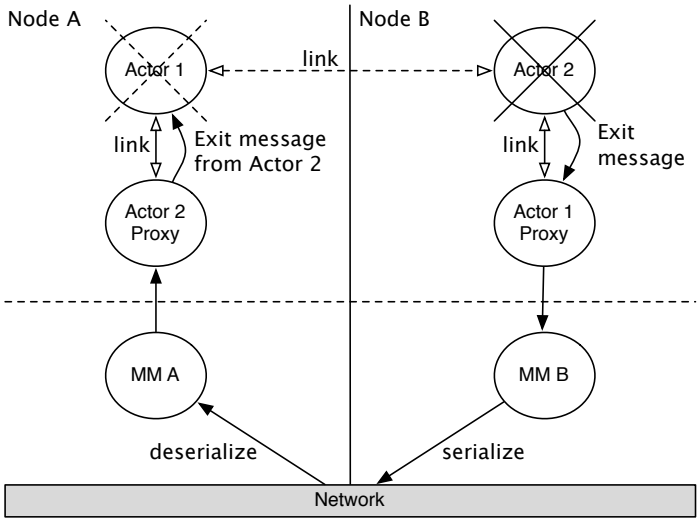

Figure 23: Links in a distributed system

Figure 23 shows a scenario for links in a distributed system. `Actor1` is linked to the proxy of `Actor2` and vice versa. Thus, whenever an actor finishes execution, it sends an exit message to all of its links including proxies of remote actors it is linked to. The exit messages send to proxies then are forwarded like any other message.

4.10.3 Publishing Actors and Connect to Remote Actors

The function `void publish(actor_ptr whom, uint16_t where)`[11] causes the MM to create a socket listening on port `where`. It throws an exception if the given port is already in use by another socket.

The function `actor_ptr remote_actor(string host, uint16_t port)` could be used to get a proxy representation of the actor running on the network node `host` published at port `port`.

[11] The type `actor_ptr` is a smart pointer to an actor.

5 Implementation of libcppa

This section presents our implementation of the software design introduced in Section 4. We will transform the language extensions to an internal domain-specific language (DSL) for C++, and discuss trade-offs we had to decide about during this process. After the DSL, we will take a closer look at implementations of individual software components including class definitions and source code as well as arguments for particular choices of algorithms.

5.1 Actor Semantic as Internal Domain-Specific Language for C++

This section covers the mapping of our syntax extension to an internal domain-specific language, as well as our implementation of atoms. Each subsection starts with a conceptual discussion from a user's point of view, followed by the actual implementation its discussion.

5.1.1 Atoms

We introduced the suffix `_atom` in Section 4.7.2. An implementation would be rather simple and straight forward by defining a mapping of strings to integer values. Unfortunately, the C++ ISO standard is yet not fully implemented by compiler vendors. Currently, no compiler vendor did implement user-defined literals in a non-beta release. However, the value of an atom should be an integer value computed at compile time.

We decided to use a `constexpr` function that converts a string constant to a 6-bit encoding stored in a `uint64_t`. The conversion is done at compile time due to `constexpr`. This is guaranteed to be collision-free though it limits atom literals to ten characters and prevents special characters. Legal characters are "`_0-9A-Za-z`" and the whitespace character. The value of an atom is represented by `enum class atom_value : uint64_t { unused = 37337 }`. An `enum class` is a strongly typed enumeration and thus not implicitly convertible to any other type. The implementation of the converting function `atom` reads:

```
1  constexpr uint64_t next_val(uint64_t val, size_t char_code) {
2    return (val << 6) | encoding_table[char_code];
3  }
4  constexpr uint64_t atom_val(char const* cstr, uint64_t val = 0) {
5    return (*cstr == '\0')
6           ? val
7           : atom_val(cstr + 1, next_val(val, *cstr));
8  }
9  template<size_t Size>
10 constexpr atom_value atom(char const (&str) [Size]) {
11   // last character is the null terminator
12   static_assert(Size <= 11, "only 10 characters are allowed");
13   return static_cast<atom_value>(detail::atom_val(str, 0));
14 }
```

The `constexpr` array `encoding_table` maps ASCII encoded characters to our 6-bit encoding. The calculated value is explicitly casted to the enum type in line 13. Still, the internal representation of `atom_value` is an integer and two values could be compared as usual, even though the values are not declared in the enum definition.

This approach provides an invertible mapping of string constants with ten or less characters to a 64-bit integers. But it cannot ensure that users only use legal characters. Each invalid character is mapped to the whitespace character, why the assertion `atom("!?") != atom("?!")` is not true. However, this issue will fade away after user-defined literals become available in mainstream compilers.

5.1.2 Receive Statement and Pattern Matching

We introduced our receive statement for our fictional extension of the `C++` programming language in Section 4.3.3 with the example below.

```
1 receive {
2   (int =1, double v2) { /* ... */ }
3   (int v1, double v2) { /* ... */ }
4   after 0sec { /* ... */ }
5 }
```

This short example shows all important characteristics of our pattern matching approach. The optional timeout is always the final statement in a receive block. It is possible to match for type and value as shown in line 2. However, we have to implement `receive` as a

function since we cannot introduce a new statement to the hosting language. Thus, we need a representation for the matching rules as shown in line 2 and 3. We could use lambda expressions rather than function bodies as a first attempt as the following example illustrates.

```
1 receive (
2   [](int =1, double v2) { /* ... */ },
3   [](int v1, double v2) { /* ... */ }
4 );
```

This approach excludes the timeout but looks promisingly close to our design at first glance. However, this is neither legal C++ code, since default parameters are not allowed if the argument has successors with non-default parameters, nor is it possible to query such default parameters. To work around, we have to split our behavior statements in two to implement the required functionality. The first part creates a *pattern* that matches incoming messages. The second part is a lambda expression that is invoked after the pattern matched a message. The first statement needs to create an intermediate object that provides a binary operator taking a lambda expression as argument. The operator then returns an *invoke rule* object that consists of the pattern of the intermediate object and the given lambda expression. The following example uses a template function to create the intermediate object for visualizing this idea.

```
1 receive (
2   on<int =1, double>() >> [](int, double v2) { /* ... */ },
3   on<int, double>() >> [](int v1, double v2) { /* ... */ }
4 );
```

Choosing an operator in C++ should always follow careful justification. One cannot introduce a new operator to the C++ programming language. We decided to use the *stream* operator “>>” (also called *bit shift* operator) for this purpose, thus, line 2 and 3 of the example above could be read as ”*an incoming message is streamed to the lambda expression on a match*”. Besides, operator “->” could not be used since it is unary, and choosing one of the other binary operators would be misleading, since they have arithmetic, logical, comparison or assigning semantic.

Still, the example code above is not legal C++ syntax, because template parameters cannot have a value. Furthermore, the first parameter of the lambda expression in line 2 is useless since we already know its value. The pattern in line 3 is redundant as we are basically matching the parameter types of the lambda expression. Thus, the pattern could be deduced from the given lambda expression. Our last example fixes all issues and is syntactically correct.

```
1 receive (
2   on(1, val<double>) >> [](double v2) { },
3   on<float,anything,double>() >> [](float v1, double v2) { },
4   on_arg_match >> [](int v1, double v2) { },
5   after(std::chrono::seconds(1)) >> [] () { }
6 );
```

There are two ways of using the variadic template function `on`, either without arguments and template parameters only, or the reverse. The function matches types only if used with template parameters. The default for using it with arguments is to match for values. The template function `val` creates a wrapper object to cause `on` to match for the type only for this particular argument. The literal "`1`" has the implicit type `int` in `C++`.

The lambda expression following the intermediate object returned by `on` can have fewer arguments. Any number of arguments can be skipped from left to right as shown in line 2. This avoids redundant arguments as long as the elements in a message follow the convention to put elements that are matched for their values to the beginning, respectively, left-hand side.

The type `anything` matches any number of elements of a message and is thus ignored in the parameter list of the lambda expression as shown in line 3.

Line 4 demonstrates how to reduce unnecessary repetitions. The special-purpose constant `on_arg_match` creates a pattern from the parameter list of the following lambda expression. Thus, line 4 is equal to `on<int,double>() >> [](int v1, double v2) { }`.

Finally, line 5 shows the use of `after` in declaring a timeout with duration definition of the `C++` standard template library (STL). The following lambda expression must not have parameters and there is no further statement allowed after a timeout definition.

The code line `on<int,double>() >> [](int v1, double v2) { }` creates an object of class `invoke_rule`. Thus, `on<...>()` has to return an intermediate object that provides operator `>>`. The type `invoke_rule_builder` of this intermediate object is transparent to the user and is thus put into the namespace `detail`. The code listing below shows the shortened class definition of `invoke_rule_builder`.

```
1  template<typename... TypeList>
2  class invoke_rule_builder
3  {
4    // ...
5   public:
6    template<typename... Args>
7    invoke_rule_builder(const Args&... args) { /*...*/ }
8    template<typename F>
9    invoke_rules operator>>(F&& f) { return /*...*/ }
10 };
```

The class `invoke_rule_builder` is basically a wrapper around an instance of the template class `pattern` and provides operator `>>` to build an `invoke_rule` object. The definition of `pattern` is as follows.

```
1  template<typename T0, typename... Tn>
2  class pattern<T0, Tn...>
3  {
4    detail::tdata<T0, Tn...> m_data;
5    const cppa::uniform_type_info* m_utis[size];
6    const void* m_data_ptr[size];
7   public:
8    static constexpr size_t size = sizeof...(Tn) + 1;
9    typedef util::type_list<T0, Tn...> tpl_args;
10   typedef typename util::filter_type_list<anything,tpl_args>::type
11           filtered_tpl_args;
12   typedef typename tuple_view_type_from_type_list
13                    <filtered_tpl_args>::type
14           tuple_view_type;
15   typedef typename tuple_view_type::mapping_vector mapping_vector;
16   pattern() {
17     const cppa::uniform_type_info** iter = m_utis;
18     detail::fill_uti_vec<decltype(iter), T0, Tn...>(iter);
19     for (size_t i = 0; i < size; ++i)
20     {
21       m_data_ptr[i] = nullptr;
22     }
23   }
24   template<typename Arg0, typename... Args>
25   pattern(const Arg0& arg0, const Args&... args)
26     : m_data(arg0, args...)
27   {
28     bool invalid_args[] = { detail::is_boxed<Arg0>::value,
29                             detail::is_boxed<Args>::value... };
30     detail::fill_vecs<decltype(m_data), T0, Tn...>(
31           0, sizeof...(Args) + 1,
32           invalid_args, m_data,
33           m_utis, m_data_ptr);
34   }
35   bool operator()(const cppa::any_tuple& msg,
36                   mapping_vector* mapping) const {
37     detail::pattern_arg arg0(size, m_data_ptr, m_utis);
38     detail::tuple_iterator_arg<mapping_vector> arg1(msg, mapping);
39     return detail::do_match(arg0, arg1);
40   }
41 };
```

A pattern has the two arrays m_utis and m_data_ptr as well as a tuple m_data holding values. The array m_utis holds each type given as template parameter with one exception. Each template parameter set to anything results in a nullptr at the corresponding position in the array m_utis. The tuple m_data can be initialized with a subset of values. The class detail::tdata<...> then initializes all unspecified values with their default constructor. Furthermore, initializing a value of type T with a boxed<T> also uses default construction. The array m_data_ptr is set to nullptr wherever the default constructor was used to create the corresponding value in m_data.

The typedef mapping_vector is a fixed-size vector holding at most N elements, where N is the number of template parameters passed to pattern. The actual pattern matching is implemented in the function detail::do_match called in line 39. It returns true if a tuple was matched by the pattern, otherwise it returns false. Furthermore, the given vector mapping contains a valid mapping as a side effect on match. This mapping could be used to create a tuple view as introduced in Section 4.7.1 from the any_tuple argument msg in line 35. This vector is in fact used as a map with the index as key. For example, a vector containing the elements $(1,3,4)$ is interpreted as $((1,1),(2,3),(3,4))$ meaning that the first element of the view is equal to the first element of msg, the second element of the view is equal to the third element of msg, and the third element of the view is equal to the fourth element of msg. The resulting view then could be used to invoke a lambda expression, since we could use the static type information of the view to use its elements as arguments. The function detail::do_match takes two parameters. The first parameter is a pattern iterator. The second parameter is an iterable representation of the incoming message including the mapping vector. The function is implemented as follows.

```
1  template<typename VectorType>
2  bool do_match(pattern_iterator& iter,
3                tuple_iterator_arg<VectorType>& targ)
4  {
5    for (; !(iter.at_end()&&targ.at_end()); iter.next(),targ.next())
6    {
7      if (iter.at_end()) return false;
8      // nullptr == anything; perform submatching
9      else if (iter.type() == nullptr) {
10       iter.next();
11       if (iter.at_end()) return true;
12       VectorType mv;
13       auto mv_ptr = (targ.mapping) ? &mv : nullptr;
14       for (; !targ.at_end(); mv.clear(),targ.next()) {
15         auto arg0 = iter;
16         tuple_iterator_arg<VectorType> arg1(targ, mv_ptr);
17         if (do_match(arg0, arg1)) {
18           targ.push_mapping(mv);
19           return true;
20         }
21       }
22       return false; // no submatch found
23     }
24     // compare types
25     else if (!targ.at_end() && iter.type() == targ.type())
26     {
27       // compare values if needed
28       if (   iter.has_value() == false
29           || iter.type()->equals(iter.value(), targ.value())) {
30         targ.push_mapping();
31       }
32       else return false; // values did not match
33     }
34     else return false; // no match
35   }
36   return true; // iter.at_end() && targ.at_end()
37 }
```

The parameter `iter` is a pattern iterator and provides the member functions `next`, `at_end`, `has_value`, `value` and `type`. The parameter `targ` iterates over an `any_tuple` that should be matched by the given pattern. It has a similar interface as `iter`, but adds a `push_mapping` that pushes the current position to the mapping vector. Passing another

vector to push_mapping, as shown in line 18, adds all elements from the passed vector to the internal vector.

A tuple is matched by a pattern if each element is equal in type and value, if specified in the pattern. Those comparisons are done in line 25 to 34. The member functions type return a pointer to a uniform type information object. Those pointers can be compared without dereferencing them, since all uniform_type_info instances are singletons.

The type anything needs special treatment as it matches zero ore more elements in the tuple. A tuple, or subtuple is always matched if anything appears as the last or only element of the pattern. Thus, the function always returns true in this case (line 11). Otherwise, the function calls itself recursively with the pattern iterator at the next position and increases the position in the tuple in each iteration (line 14 to 22) until it reaches the end of the tuple without match. A user of libcppa will barely ever need to use the pattern class. However, a pattern could be used to create a view from a tuple as shown in the following example.

```
1 pattern<int,anything,double> p(1);
2 decltype(p)::mapping_vector mv;
3 any_tuple t = make_tuple(1, 2, 3, 4, 5, 6.0);
4 if (p(t, &mv)) {
5   tuple_view<int,double> tv(d, mv);
6   // or: decltype(p)::tuple_view_type tv(d, mv);
7   assert(get<0>(tv) == 1);
8   assert(get<1>(tv) == 6.0);
9 }
```

The template class invoke_rule_builder creates a pattern in its constructor from given parameters and template parameters. The function on that we introduced earlier to define the receive statement is an overloaded function defined as follows.

```
1  template<typename T>
2  constexpr typename detail::boxed<T>::type val() {
3      return typename detail::boxed<T>::type();
4  }
5  constexpr anything any_vals = anything();
6  template<typename... TypeList>
7  detail::invoke_rule_builder<TypeList...> on() {
8      return { };
9  }
10 template<typename Arg0, typename... Args>
11 detail::invoke_rule_builder
12 <typename detail::unboxed<Arg0>::type,
13  typename detail::unboxed<Args>::type...>
14 on(Arg0 const& arg0, Args const&... args) {
15     return { arg0, args... };
16 }
17 template<atom_value A0, typename... TypeList>
18 detail::invoke_rule_builder<atom_value, TypeList...> on() {
19     return { A0 };
20 }
21 // ...
22 template<atom_value A0, atom_value A1,
23          atom_value A2, atom_value A3,
24          typename... TypeList>
25 detail::invoke_rule_builder<atom_value, atom_value, atom_value,
26                             atom_value, TypeList...> on() {
27     return { A0, A1, A2, A3 };
28 }
```

The first overload in line 6-9 is the simple case that matches types only. The second overload in line 10-16 is the case that matches for values. A user can use the function `val` to match for types only on a particular element and `any_vals` as wildcard expression. The last cases are convenience overloads that allow up to four[12] atoms to be used in front of a type-only match as shown in line four and five of the usage examples below.

```
1 on<int,double>()
2 on(1,val<double>)
3 on(any_vals, val<double>) // ... equal to: on<anything,double>()
4 on<atom("foo"),int>() // ... equal to: on(atom("foo"),val<int>)
5 on<atom("one"),atom("two"),anything,int>()
```

[12] It is not possible to use two variadic template parameter arguments. For example, `template<int...,class...>` will cause a compiler error. Thus, we decided to overload `on` for up to four leading atoms. There should be seldom a use for more than one.

As mentioned earlier, an "on(...) >> ..." expression returns an invoke_rules object. Such an object encapsulates a pattern and a lambda expression. There is also a class timed_invoke_rules that encapsulates a timeout instead of a pattern and a lambda expression. Objects of that type are created by operator >> of timed_invoke_rule_builder as shown below.

```
1 template<class Rep, class Period>
2 constexpr detail::timed_invoke_rule_builder
3 after(const std::chrono::duration<Rep, Period>& d)
4 {
5     return { util::duration(d) };
6 }
```

Class duration in namespace util used in line 5 is a simple wrapper for timeout durations. It accepts seconds, milliseconds and microseconds. Statements beginning with after have to have a return type different from statements beginning with on in order to ensure at most one timeout specification. Finally, the function receive will cause a compile-time error if used with more than one timeout specification. It is implemented as follows.

```
1  inline void receive(invoke_rules& rules) {
2    self->dequeue(rules);
3  }
4  inline void receive(timed_invoke_rules&& rules) {
5    timed_invoke_rules tmp(std::move(rules));
6    self->dequeue(tmp);
7  }
8  inline void receive(timed_invoke_rules& rules) {
9    self->dequeue(rules);
10 }
11 inline void receive(invoke_rules&& rules) {
12   invoke_rules tmp(std::move(rules));
13   self->dequeue(tmp);
14 }
15 template<typename Head, typename... Tail>
16 void receive(invoke_rules&& rules, Head&& head, Tail&&... tail) {
17   invoke_rules tmp(std::move(rules));
18   receive(tmp.splice(std::forward<Head>(head)),
19           std::forward<Tail>(tail)...);
20 }
21 template<typename Head, typename... Tail>
22 void receive(invoke_rules& rules, Head&& head, Tail&&... tail) {
23   receive(rules.splice(std::forward<Head>(head)),
24           std::forward<Tail>(tail)...);
25 }
```

The first two declarations in line 1 and 4 cover use cases with either a variable or an rvalue[13] passed to `receive`. Declaration three and four in line 8 and 11 are analogues for usage with timeout. The recursively implemented overloads in line 15 and 21 cover the use case of `receive` with multiple parameters. The member function `splice` of `invoke_rules` moves the content of the right-hand operand to the left-hand operand. It returns a reference to the left-hand operand unless the right-hand operand was a `timed_invoke_rules` object. In this case, a new `timed_invoke_rules` object is returned with the content of both operands moved to it. The class `timed_invoke_rules` does neither provide a `slice` member function nor is a function overload of `receive` available with `timed_invoke_rules` as first argument followed by any other argument. Thus, the recursion ends on a `timed_invoke_rules` argument. Any additional argument after a `timed_invoke_rules` object will cause a compiler error as shown in line 5 of the usage example below.

```
1 receive(
2   on<int,double>() >> []() { /*...*/ },
3   on("hello world") >> []() { /*...*/ },
4   after(std::chrono::seconds(1)) >> []() { /*...*/ }
5   //, on<int>() >> []() { } [would cause a compiler error]
6 );
```

It is also possible to store a behavior in a variable, since `invoke_rules` provides the operator "`,`" to concatenate objects. Such concatenations have to appear in parentheses as shown in the following example. The commas would separate variable declarations in `C++` if used without parentheses.

```
1 auto sample_behavior = (
2   on("hello world") >> []() { /*...*/ },
3   after(std::chrono::seconds(1)) >> []() { /*...*/ }
4 );
```

The type of `sample_behavior` is `timed_invoke_rules` since concatenation uses the same recursive implementation as receive does. Thus, the type depends on the initialization. This might be an issue if a user wants to store a behavior in a member variable that should carry both rules with and without timeout. Therefore, we provide the class `behavior` that stores either an `invoke_rules` instance or a `timed_invoke_rules` and can be used wherever a behavior is expected. The following example code illustrates such use.

[13] An *rvalue* identifies a temporary object or an operand of a move assignment.

```
1 behavior b = (
2   on("hello world") >> []() { /*...*/ },
3   after(std::chrono::seconds(1)) >> []() { /*...*/ }
4 );
5 receive(b); // b stores a timed_invoke_rule
6 b = (
7   on<int,double>() >> []() { /*...*/ }
8 );
9 receive(b); // b stores an invoke_rule
```

5.1.3 Receive Loops

The function `receive` is explicitly designed to be used with temporary objects. A receive statement shall contain all patterns and lambda expression. Functionality should not be spread, but written exclusively at the position of the receive statement. However, this approach comes at a performance cost if used in a loop as the following example illustrates.

```
1 for (;;) {
2   receive(
3     on("hello world") >> []() { /*...*/ },
4     after(std::chrono::seconds(1)) >> []() { /*...*/ }
5   );
6 }
```

The receive statement in line 2 is nested in a loop. Thus, it creates identical temporary objects in each iteration again and again. This may have serious performance impacts for large receive statements. Therefore, we provide the three functions implementing a receive loop without redundant object creation issues shown below.

```
1 receive_loop(/*behavior*/);
2 receive_while(/*lambda returning bool*/)(/*behavior*/);
3 do_receive(/*behavior*/).until(/*lambda returning bool*/);
```

The first function never returns. It is an endless loop receiving messages. The second loop receives messages as long as the given lambda expression returns `true`. This is equal to receive used in a while loop. The last loop receives messages until the given lambda returns `true`. It is equal to a do..while loop in C++ except that it loops until the condition becomes `true` while a `do..while` loop continues until the condition is `false`.

The following example compares two implementations of the same loop using `do_receive` and a `do..while` loop.

```
1  int hello_world_count = 0;
2  do_receive (
3    on(atom("HelloWorld")) >> [&]() { ++hello_world_count; }
4  )
5  .until([&]() { return hello_world_count == 10 });
6  // -----
7  int hello_world_count = 0;
8  do {
9    receive (
10     on(atom("HelloWorld")) >> [&]() { ++hello_world_count; }
11   );
12 }
13 while (hello_world_count < 10);
```

The `do_receive` loop closely approximates a native language construct. However, especially the lambda expression used as condition pollutes the source code with brackets and a `return` statement, making it harder to read than its native counterpart. The `C++` programming language was not specified with internal or embedded domain-specific language support in mind. Thus, user-defined statements will always be distinguishable from native statements.

5.1.4 Send Statement

We discussed the operator `<-` in our design section, even though this operator is not available in `C++`. Instead, we could use the stream operator `<<` for the purpose of sending messages, which still does not accept our preferred syntax `self << { val1, val2, ... }`[14]. We are required to convert the message elements to a tuple first: `self << make_tuple(val1, val2, ...)`. This is more verbose than originally intended. Accordingly, we also added the function `send` that is less verbose: `send(self, val1, val2, ...)`. Both implementations are straightforward.

[14] GCC rejects such code with: `expected primary-expression before '{' token`.

```
1  template<class C, typename Arg0, typename... Args>
2  void send(intrusive_ptr<C>& whom,
3            Arg0 const& arg0,
4            Args const&... args)
5  {
6      static_assert(std::is_base_of<channel,C>::value,
7                    "C is not a channel");
8      if (whom) whom->enqueue(self, make_tuple(arg0, args...));
9  }
10
11 template<class C>
12 typename util::enable_if<std::is_base_of<channel,C>,
13                          intrusive_ptr<C>&         >::type
14 operator<<(intrusive_ptr<C>& whom, any_tuple const& what)
15 {
16     if (whom) whom->enqueue(self, what);
17     return whom;
18 }
```

The static assertion in line 6 gives users a clear hint whether `send` was used incorrectly. The operator `<<` uses a technique known as SFINAE (*Substitution Failure Is Not An Error*, Vandevoorde and Josuttis (2002)) by using the template `enable_if`. The typedef `enable_if<...>::type` is undefined if `channel`[15] is not a base of `C`. The compiler will ignore the function definition in such, case since the function does not have a valid return type due to a substitution failure.

5.1.5 Emulating The Keyword `self`

The keyword `self` is an essential ingredient of our design. From a user's pointer of view, the keyword identifies an actor similar to the implicit `this` pointer identifying an object within a member function. Unlike `this`, though, `self` is not limited to a particular scope. Furthermore, it is not just a pointer, but it needs to perform implicit conversions on demand as defined in Section 4.5.3. Consequently, `self` requires a type allowing implicit conversion to `local_actor*`, where the conversion function returns a thread-local pointer. Our approach shown below uses a global `constexpr` variable with a type that behaves like a pointer.

[15] See Section 4.4 for the definition of `channel`.

```
1  class self_type {
2    static local_actor* get_impl();
3    static voidset_impl(local_actor* ptr);
4   public:
5    constexpr self_type() { }
6    inline operator local_actor*() const {
7      return get_impl();
8    }
9    inline local_actor* operator->() const {
10     return get_impl();
11   }
12   inline void set(local_actor* ptr) const {
13     set_impl(ptr);
14   }
15 };
16 constexpr self_type self;
```

The `constexpr` variable `self` provides access to the implicit conversion operator as well as the dereference operator "`->`". From a user's point of view, `self` is not distinguishable from a pointer of type `local_actor`. The static member functions are implemented as follows.

```
1 thread_local local_actor* t_self = nullptr;
2 local_actor* self_type::get_impl() {
3   if (t_self == nullptr) t_self = convert_thread_to_actor();
4   return t_self;
5 }
6 void self_type::set_impl(local_actor* ptr) {
7   t_self = ptr;
8 }
```

Our approach adds little, if any, overhead to an application. In fact, `self` is nothing but syntactic sugar and the compiler could easily optimize away the overhead of using member functions. A `constexpr` variable does not cause a dynamic initialization at runtime, why the global variable `self` does not cause any overhead since it provides an empty `constexpr` constructor. Furthermore, all member functions are declared `inline`, allowing the compiler to replace each occurrence of a member function with a call to `self_type::get_impl`. `self.set()` is intended for in-library use only. The latter is needed to implement cooperative scheduling.

5.2 Mailbox Implementation

The message queue, or mailbox, implementation is an important component of message passing systems. In fact, the overall system performance including its scalability depends on the chosen algorithm. A mailbox is a single-reader-many-writer queue. Everyone is allowed to enqueue a message to a mailbox, but only the owning actor is allowed to dequeue a message from it. Hence, the dequeue operation does not need to support parallel access.

Mutex-based concurrency does not scale well, especially in multi-core environments. Therefore, lock-free queue implementations became an independent research topic. Scalable solutions use atomic operations to avoid expensive locking. However, such atomic operations require hardware support. Furthermore, a lock-free implementation has to solve the so-called ABA problem. The ABA problem occurs whenever a thread T_1 reads a value A from a shared memory segment and gets suspended. Another thread T_2 then changes the value of the shared memory segment from A to B and back to A again. When T_1 finally gets resumed, it has no way to detect the modification (I.B.M. Corporation, 1983). This may corrupt states in CAS[16]-based systems (Dechev et al., 2006).

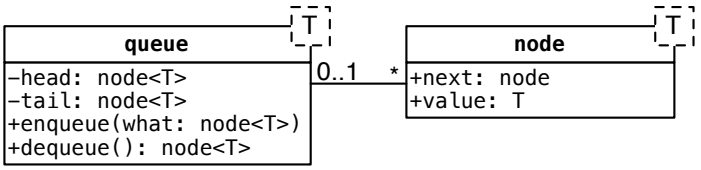

Figure 24: Generic queue interface

Figure 24 shows a simple interface for a queue. The `enqueue` member function could be implemented as in the pseudo code below.

```
1 void enqueue(node<T>: what) {
2   tail.next = what;
3   tail = what;
4 }
```

In this example, `enqueue` manipulates two memory locations, `tail.next` first and afterwards `tail` itself. Even if we assume that setting a value is atomic, a race condition is very likely to occur. Consider that thread T_1 manipulates `tail.next` and gets interrupted by thread T_2. T_2 overrides the change to `tail.next` and writes `tail`. T_1 then also writes `tail` resulting in an inconsistent state.

[16] CAS stands for *compare-and-swap* and denotes an atomic operation with three parameters: $Address$, $ExpectedValue$ and $NewValue$. It compares the value stored at the memory location $Address$ and replaces it with $NewValue$ if $ExpectedValue$ was read.

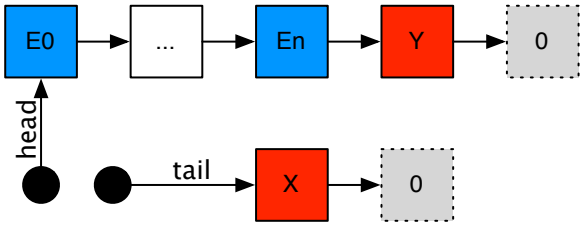

Figure 25: Inconsistent queue state with unreachable tail

Figure 25 shows an inconsistent queue state due to the race condition in the previous example. X is the element enqueued by T_1, Y is the element enqueued by T_2. X has no predecessor in the queue since $En.next$ was overwritten by T_2 with Y. Thus, X is unreachable from $head$. To avoid such inconsistencies we need to set both pointer atomically like in the following pseudo code, or we need to choose a more complex algorithm that excludes race-conditions and still is reasonably fast.

```
1  void enqueue(node<T>: what) {
2    atomic { // the holy grail of concurrency
3      tail.next = what;
4      tail = what;
5    }
6  }
```

5.2.1 Spinlock Queue

A *spinlock* (Mellor-Crummey and Scott, 1991) is an integer value guarding a critical section. It is initialized with 0 and set atomically to 1 using a CAS operation. It is called *spin*lock because a thread *spins* in a loop waiting for the CAS operation to succeed. The following example adds an atomic integer variable $lock$ to our queue.

```
1  void enqueue(node<T>: what) {
2    for (;;) {
3      if (CAS(&lock, 0, 1)) { // lock acquired
4        tail.next = what;
5        tail = what;
6        lock = 0; // release lock
7        return;
8      }
9    }
10 }
```

The spinlock is acquired in line 3 by trying to set the value of lock from 0 to 1. The lock is released to 0 after the critical section is left. We assume this to be atomic in our example. Furthermore, we assume that neither the compiler nor the runtime environment reorder write operations in the critical section. Spinlocks are lightweight locks that do not involve system calls and thus outperform mutex based implementations in practice (Sutter, 2008). Spinlocks are reasonably fast for short critical sections, but the waiting loop becomes wasteful otherwise.

5.2.2 Lock-Free Queue

So far, we assumed a queue always to have a consistent state. There are inconsistent states from which no recovery exists (see Figure 25). However, the following example is an optimistic approach allowing temporarily inconsistent states.

```
1 void enqueue(node<T>: what) {
2   for (;;) {
3     node<T> old_tail = tail;
4     if (CAS(&tail, old_tail, what)) {
5       old_tail.next = what;
6       return;
7     }
8   }
9 }
```

This implementation first sets the *tail* pointer in an atomic operation. The predecessor is connected to the new element thereafter.

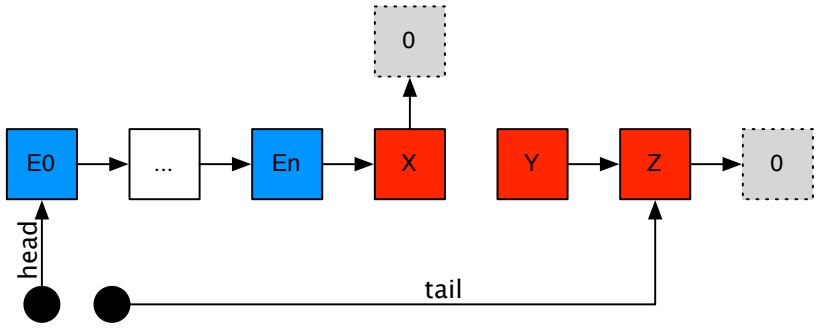

Figure 26: Temporary inconsistent queue state

Figure 26 shows a possible state of this algorithm. Y and Z are enqueued after X but are not yet reachable from $head$, because the thread T_1 that enqueued Y did yet not updated

$X.next$. However, this is only a temporary inconsistency. The queue returns to a consistent state after T_1 updates $X.next$.

5.2.3 Cached Stack

There is a data structure that does not exhibit such consistency issues since it requires only one CAS operation for its enqueue operation: the *stack*. A stack does not have a $head$ pointer since it is a LIFO container. However, a mailbox provides FIFO ordering of messages. Thus, a dequeue operation would have complexity $O(n)$ since it would have to traverse the whole stack to reach the oldest element. We combined a stack with a FIFO ordered, non-thread-safe cache to achieve a fast and efficient enqueue operation offered by a stack as well as FIFO ordered dequeue operations.

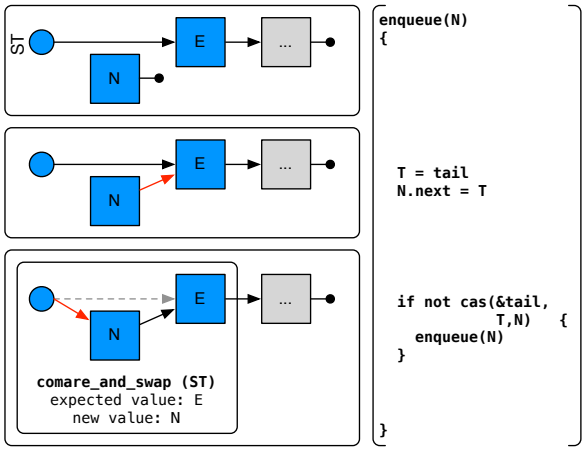

Figure 27: Enqueue operation in a cached stack

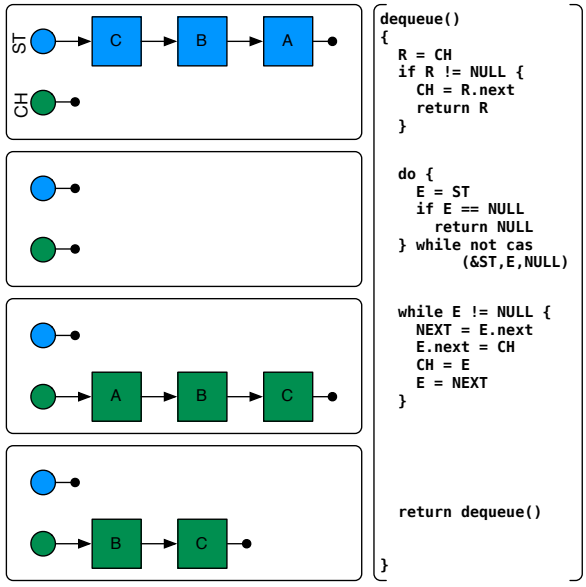

Figure 28: Dequeue operation in a cached stack

Figure 27 shows the enqueue operation. It only needs one CAS operation and always maintains a consistent state. Figure 28 shows the dequeue implementation. It always dequeues elements from the FIFO ordered cache (CH). The stack (ST) is emptied and its elements are moved in reverse order to the cache if there was no element in the cache. Emptying the stack is also done with one CAS operation because it just has to set ST to NULL.

Enqueue has the complexity $O(1)$ while dequeue has an *average* of $O(1)$ but a worst case of $O(n)$. However, concurrent access to the cached stack is reduced to a minimum.

5.2.4 Choosing an Algorithm

We introduced three possible single-reader-many-writer queue implementations. We only introduced algorithms that could be implemented with atomic CAS operations. As mentioned earlier, atomic operations need hardware support. CAS is widely supported by mainstream vendors and is part of the C++ standard library. There are other algorithms available that require DCAS (double-word CAS that could update a memory location of two words length rather than one) or other atomic operations (Michael and Scott, 1996). However, we have tested our three queue implementations in a performance benchmark to obtain reliable decision guidance.

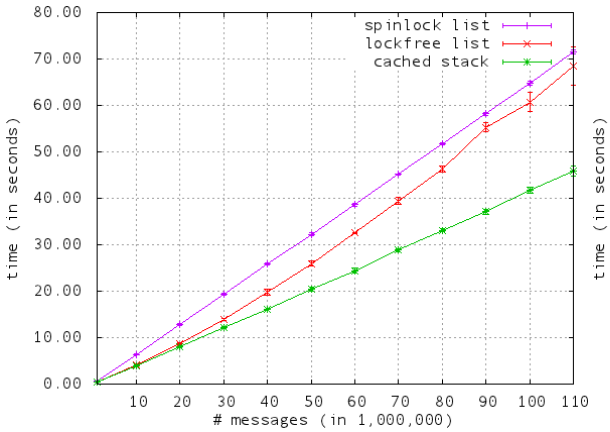

Figure 29: Queue benchmark using 4 threads

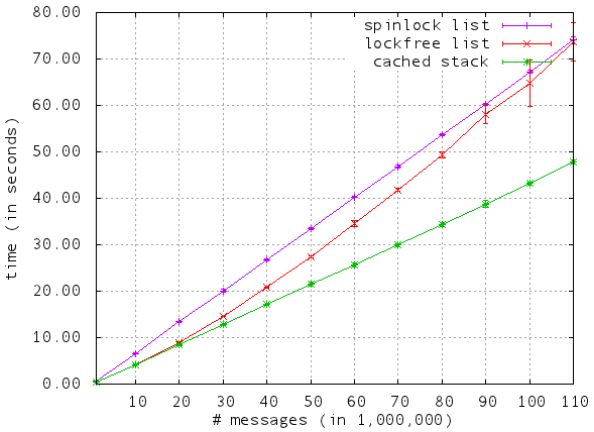

Figure 30: Queue benchmark using 8 threads

In Figures 29 and 30, benchmark results for all three queue algorithms are shown for a 2.66 GHz dual core i7 processor. All tests used a polling consumer thread without synchronization overhead other than caused by the queue algorithms. More threads increase the probability of collisions during enqueue. Thus, the performance decreases with increased concurrency. However, the spinlock queue and the cached stack scale linearly with the number of messages. The performance of the lockfree list depends heavily on timing. If a thread sets the $tail$ pointer and gets suspended thereafter, the consumer has to wait until that particular worker resumes. The lockfree queue approximates the cached stack performance if workers seldom get suspended during an enqueue, but approximates the performance of the spinlock queue for heavy work load.

Our implementation of the lockfree queue did not solve the ABA problem. However, we implemented our mailbox using the cached stack algorithm since it yields the best performance. The cached stack avoids the ABA problem since its enqueue operation does not manipulate preceding elements.

5.3 Actors

We introduced three kinds of actors in Section 4.5: converted threads, cooperatively scheduled actors and event-based actors. This section discusses our actor implementations as well as the ways users create actors. In addition, appropriate use case examples are given.

5.3.1 Spawning Actors

The function `spawn` starts a new actor and returns an `actor_ptr` to its creator. We have implemented an opt-out cooperative scheduling using a thread pool as discussed in Section 4.5.4. An actor that should not be cooperatively scheduled can be spawned *detached* to run in an own thread. The prototypes of the function `spawn` read as follows.

```
1  enum scheduling_hint {
2      scheduled,
3      detached
4  };
5  template<scheduling_hint Hint, typename F, typename... Args>
6  actor_ptr spawn(F&& what, Args const&... args);
7  template<typename F, typename... Args>
8  inline actor_ptr spawn(F&& what, Args const&... args) {
9      return spawn<scheduled>(std::forward<F>(what), args...);
10 }
11 actor_ptr spawn(abstract_event_based_actor* what);
```

The enumeration `scheduling_hint` denotes whether a user wants an actor to take part in cooperative scheduling or not. The value `detached` can be passed as a template parameter to the function overload of `spawn` in line 5. The template parameter `F` identifies a function pointer or functor. The parameters `args` are passed to that function or functor on execution. The function `spawn` could be used without declaring a `scheduling_hint` (see line 8) in which case the default is `scheduled`. The last function overload in line 11 executes the given instance of an event-based actor. Event-based actors are always scheduled in the thread pool.

5.3.2 Abstract Actor

The class `abstract_actor` implements all pure virtual member functions of the class `actor` as defined in Section 4.5. The two member functions `join` and `leave` were im-

plemented using attach and detach. The member function join *attaches* a group subscription to the actor that could be *detached* by using leave. The C++ declaration of the abstract class actor, restricted to its pure virtual functions, looks as follows.

```
1  class actor : public channel {
2   public:
3    virtual void attach(attachable* ptr) = 0;
4    virtual void detach(attachable::token const& what) = 0;
5    virtual void link_to(actor_ptr& other) = 0;
6    virtual void unlink_from(actor_ptr& other) = 0;
7    virtual bool establish_backlink(actor_ptr& other) = 0;
8    virtual void remove_backlink(actor_ptr& other) = 0;
9  };
```

We have added the two member functions establish_backlink and remove_backlink to our initial design. Links must be bidirectional[17], why an actor has to establish a backlink from the actor it becomes linked to. The class abstract_actor uses the following member variables.

```
1  class abstract_actor : public actor {
2    typedef std::lock_guard<std::mutex> guard_type;
3    typedef std::unique_ptr<attachable> attachable_ptr;
4    uint32_t m_exit_reason;
5    std::mutex m_mtx;
6    std::list<actor_ptr> m_links;
7    std::list<attachable_ptr> m_attachables;
8    inline bool exited() const {
9      return m_exit_reason != exit_reason::not_exited;
10   }
11 };
```

Each access to one of the lists or to the exit reason is guarded by a mutex. The typedef guard_type is a RAII (*Resource Acquisition Is Initialization*, Stroustrup (1995)) style class that locks a mutex in its constructor and automatically unlocks it in its destructor if the variable leaves its scope. The utility member function exited returns true after the actor already finished execution. The namespace exit_reasons contains the following predefined constants.

[17] As shown in Section 2.4.3, a link couples the lifetime of actors.

```
1 namespace exit_reason {
2   constexpr uint32_t not_exited = 0x00000;
3   constexpr uint32_t normal = 0x00001;
4   constexpr uint32_t unhandled_exception = 0x00002;
5   constexpr uint32_t remote_link_unreachable = 0x00101;
6   constexpr uint32_t user_defined = 0x10000;
7 }
```

An actor is alive as long as its exit reason is `not_exited`. The constant `normal` identifies an actor did finish execution without any error, while `unhandled_exception` is the exit reason for an actor that died unexpectedly. The error code `remote_link_unreachable` is used only if the middle man lost connection to remote actors. Then each proxy will send this error code to all of its links. A user also can define its own exit reasons, if he wishes to, but has to use a constant greater or equal `user_defined` to ensure that its own error codes do not collide with internal error codes.

The implementations of the four link-related member functions in `abstract_actor` are as follows.

```
1  void abstract_actor::link_to(actor_ptr& other) {
2    guard_type guard(m_mtx);
3    if (exited()) send(other, atom("exit"), m_exit_reason);
4    else if (other->establish_backlink(this))
5      m_links.push_back(other);
6  }
7  bool establish_backlink(actor_ptr& other) {
8    guard_type guard(m_mtx);
9    if (exited()) {
10     send(other, atom("exit"), m_exit_reason);
11     return false;
12   }
13   m_links.push_back(other);
14   return true;
15 }
16 void unlink_from(actor_ptr& other) {
17   guard_type guard(m_mtx);
18   other.remove_backlink(this);
19   m_links.remove_if([&](actor_ptr& ptr) { return ptr == other });
20 }
21 void remove_backlink(actor_ptr& other) {
22   guard_type guard(m_mtx);
23   m_links.remove_if([&](actor_ptr& ptr) { return ptr == other });
24 }
```

The member functions attach and detach are analogues but do not send exit messages nor manipulate the member variable m_attachables instead of m_links.

5.3.3 Thread-Mapped Actors

A thread-mapped actor is of the type converted_thread_context that represents either threads that are implicit converted to actors or actors spawned with scheduling hint set to detached. The class converted_thread_context implements the member function dequeue inherited from local_actor, see Section 4.5.

5.3.4 Cooperatively Scheduled Actors

Cooperatively scheduled actors are executed in a thread pool. This thread pool is managed by a supervisor that offers two queues. The first queue is the *worker queue* that contains all idle threads of the pool. The second queue is the *job queue* that contains all actors in *ready* state. An actor is ready if it has at least one message in its mailbox. The supervisor increases the size of the thread pool dynamically, if all workers are blocked for a certain amount of time, as illustrated by the pseudo code below.

```
supervisor()
  job = job_queue.dequeue()
  if (worker_count < max_worker_count)
    worker = worker_queue.try_dequeue(500ms)
  else
    worker = worker_queue.dequeue()
  if (worker == NULL) worker = new_worker()
  worker.execute(job)
  supervisor()
```

To prevent starvation of other actors in the job queue, the supervisor waits at most 500 milliseconds for a worker to become unblocked again. It then starts a new worker, unless the thread pool reached its maximum size. We have chosen a long timeout since we assume actors to behave well.

Cooperatively scheduled actor implementations inherit from abstract_scheduled_actor and override the member function resume that is defined as follows.

```
1 struct resume_callback {
2   virtual void exec_done() = 0;
3 };
4 class abstract_scheduled_actor : public abstract_actor {
5  public:
6   virtual void resume(fiber* from, resume_callback* callback) = 0;
7 };
```

The class `fiber` is a platform-independent context switching implementation. The parameter `from` denotes the context of the calling worker. The resumed actor calls `callback->exec_done()` to indicate that it finished execution.

Our default implementation used for scheduled actors is the class `yielding_actor`. It uses the following context switching API.

```
1 enum class yield_state {
2   blocked,
3   done
4 };
5 yield_state call(util::fiber* callee, util::fiber* caller);
6 void yield(yield_state state);
```

The function `call` switches to the context `callee`. A call to `yield` then switches back to `caller`. The parameter `state` passed to `yield` becomes the return value of `call`. The return value `blocked` indicates that it is waiting for an event, e.g., a new message, and `done` indicates that the callee finished execution. The resume member function of `yielding_actor` is shown below.

```
1  void yielding_actor::resume(fiber* from, resume_callback* callback)
2  {
3    self.set(this);
4    for (;;) {
5      switch (call(&m_fiber, from)) {
6        case yield_state::done:
7          callback->exec_done();
8          return;
9        case yield_state::blocked:
10         return;
11     }
12   }
13 }
```

An actor calls `yield` whenever its mailbox is empty. A receive statement with a timeout causes the actor to send a delayed timeout message, a so-called *future message*, to itself rather than block the thread it is executed in.

5.3.5 Event-Based Actors

Event-based actors are scheduled in the same thread pool as cooperatively scheduled actors. The `resume` member function does not perform context switching, but dequeues the next message from the actor's mailbox and executes the currently active behavior if defined for the dequeued message. However, programming event-based actors is slightly different since one has to implement the actor as a class. Users can choose one of the three classes `event_based_actor`, `stacked_event_based_actor` and `fsm_actor`.

All event-based actor classes provide the member function `become` that takes either a behavior expression or a pointer to a behavior member variable. The class `stacked_event_based_actor` also provides an `unbecome` member function that restores its previous behavior. Thus, a user is able to nest behavior as the following example illustrates.

```
1  struct test_actor : stacked_event_based_actor {
2    void init() {
3      become(
4        on<int>() >> [=](int v1) {
5          become(
6            on<double>() >> [=](double v2) {
7              reply(v1 * v2);
8              unbecome();
9            }
10         );
11       }
12     );
13   }
14 };
```

Unlike `receive`, the behavior passed to `become` is executed until it will be replaced by another behavior. Furthermore, `become` does replace the actor's event handler with the specified behavior and returns immediately. An actor finishes execution for normal exit reasons if `unbecome` is called with an empty behavior stack. An actor inheriting from `event_based_actor` or `stacked_event_based_actor` must override the member function `init` that should set an initial behavior. The template class `fsm_actor` uses the

curiously recurring template pattern (Coplien, 1995) to set the initial behavior of an actor to the member variable `init_state` of the derived class. It is implemented as follows.

```
1  template<class Derived>
2  struct fsm_actor : event_based_actor
3  {
4    void init() {
5      become(&(static_cast<Derived*>(this)->init_state));
6    }
7  };
```

A class inheriting from `fsm_actor` passes itself as template parameter, illustrated in the example below.

```
1  struct test_actor : fsm_actor<test_actor> {
2    behavior init_state = (
3      on<int>() >> [=](int value) {
4        // ...
5        become(&wait4double);
6      }
7    );
8    behavior wait4double = (
9      on<double>() >> [=](double value) {
10       // ...
11       become(&init_state);
12     }
13   );
14 };
```

5.4 Groups

The technology-transparent group communication API, introduced in Section 4.8, allows developers to provide their own modules to extend libcppa. We implemented a simple module for local groups identified by user-defined names. Future releases of libcppa may include implementations for network technologies such as IP multicast. Our implementation of the group interface introduced in Section 4.8 in C++ is defined as follows.

```
1  class group : public channel {
2   protected:
3      group(std::string const& id, std::string const& mod_name);
4      virtual void unsubscribe(channel_ptr const& who) = 0;
5   public:
6      class subscription;
7      class module {
8       protected:
9        module(std::string const& module_technology);
10      public:
11       virtual intrusive_ptr<group> get(std::string const& id) = 0;
12     };
13     std::string const& identifier() const;
14     std::string const& technology() const;
15     virtual subscription* subscribe(channel_ptr const& who) = 0;
16     static intrusive_ptr<group> get(std::string const& technology,
17                                     std::string const& identifier);
18     static void add_module(module* impl);
19 };
```

There are three member functions a class inheriting from group needs to override: subscribe, unsubscribe and enqueue. The latter is inherited from channel. The class group::subscriber is an *attachable*, see Section 4.5.1, that automatically unsubscribes an actor after it finished execution. The factory class for such a user-defined group implementation needs to inherit from group::module and to override get, which creates a group object from a technology-dependent group identifier.

5.4.1 Local Group Module

A local group is a list of actors identified by some name. Its purpose is to provide a process-wide group communication. Our implementation of local_group as shown below uses a

shared_spinlock that allows either shared locking or exclusive locking. Thus, any number of actors can enqueue messages in parallel to a single group.

```
1  class local_group : public group {
2    shared_spinlock m_shared_mtx;
3    std::set<channel_ptr> m_subscribers;
4   public:
5    local_group(const std::string& id) : group(id, "local") { }
6    void enqueue(actor* sender, const any_tuple& msg) {
7      shared_guard guard(m_shared_mtx);
8      for (channel_ptr& cptr : m_subscriber) {
9        cptr->enqueue(sender, msg);
10     }
11   }
12   group::subscription* subscribe(const channel_ptr& who) {
13     exclusive_guard guard(m_shared_mtx);
14     if (m_subscribers.insert(who).second)
15       return new group::subscription(who, this);
16     return nullptr;
17   }
18   void unsubscribe(const channel_ptr& who) {
19     exclusive_guard guard(m_shared_mtx);
20     m_subscribers.erase(who);
21   }
22 };
```

The member variable m_subscriber is a set of actors. A set is an associative container that stores its elements uniquely. Its insert operation returns a pair consisting of an iterator to the element and a boolean value that is set to false, if the element was already inserted to the set. The local_group does create a subscription object only if an actor did not already subscribe to the group, see line 15. The module that manages local group instances is implemented as follows.

```
1  class local_group_module : public group::module {
2    shared_spinlock m_mtx;
3    std::map<std::string, group_ptr> m_instances;
4   public:
5    local_group_module() : group::module("local") { }
6    intrusive_ptr<group> get(const std::string& group_name) {
7      shared_guard guard(m_mtx);
8      auto i = m_instances.find(group_name);
9      if (i != m_instances.end())
10       return i->second;
11     else {
```

```
12        group_ptr tmp(new local_group(group_name));
13        upgrade_guard uguard(guard);
14        auto p = m_instances.insert(std::make_pair(group_name, tmp));
15        return p.first->second;
16      }
17    }
18  };
```

The member function `get` seeks for an element with a shared lock first. If the map does not contain an appropriate object, the lock becomes exclusive by using an upgrade guard, see line 13. The insert operation of the map could return a different object than `tmp` if another thread preemptively inserted another instance. Thus, the function returns always the instance found in the map, see line 15, to ensure that always the same object is returned for any given name.

5.4.2 A Use Case Example

The following use case example illustrates the use of a local group for in-process event handling. Our usage example implements a daemon actor that sends an *update* message every two seconds to the group “time check” and five listening actors finish execution after receiving five updates.

```
1  void daemon() {
2    auto g = group::get("local", "time check");
3    receive_loop(
4      after(seconds(2)) >> []() { send(g, atom("update")); }
5    );
6  }
7  void worker() {
8    self->join(group::get("local", "time check"));
9    int i = 0;
10   receive_while([&]() { return i < 5; }) (
11     on(atom("update")) >> [&]() { ++i; }
12   );
13 }
14 int main() {
15   spawn(daemon);
16   for (int i = 0; i < 5; ++i) spawn(worker);
17   await_all_others_done();
18   return 0;
19 }
```

The function await_all_others_done used in line 19 blocks the calling actor until all other actors finish execution. Thus, this example program never terminates since the daemon actor remains in its loop forever.

5.5 Serialization

This section discusses our implementation of the uniform type representation and semi-automated serialization of user-defined classes as introduced in Section 4.9.2.

5.5.1 Uniform Type Name

The C++ standard does not specify the content of a string returned by type_info::name. Thus, typeid(X).name() does not only depend on X. It also depends on the used compiler. This is not suitable for serialization, since we need a mapping from a type name to its runtime type for deserialization. Thus, uniform_type_info::name has to return an identical string on each platform. The MS Visual C++ compiler generates a human-readable name that can be easily transformed to the type name as declared in the source code. GCC contrary does return a mangled representation that can be demangled using GCCs ABI[18]. The function demangle must be implemented for each supported compiler. Its output shall be the type name as declared in source code without any whitespaces or qualifiers such as “class” or “struct”. The following implementation demangles a type name produced by GCC to our indented representation.

```
1  std::string demangle(char const* decorated) {
2    size_t size;
3    int status;
4    char* undecorated = abi::__cxa_demangle(decorated, nullptr,
5                                            &size, &status);
6    assert(status == 0);
7    std::string result = undecorated;
8    free(undecorated);
9    filter_whitespaces(result);
10   return result;
11 }
```

The argument decorated is the C-string returned by type_info::name. The function abi::__cxa_demangle in line 4 returns a C-string containing a demangled version of

[18] An **A**pplication **B**inary **I**nterface is a low-level interface for applications.

the type name. On success, status is set to 0. The C-string might contain needless whitespaces, e.g., "> >" rather than ">>" in nested template names. Those whitespaces are removed by filter_whitespaces called in line 10. It erases each whitespace in the string unless it separates two alphanumeric characters, such as in "unsigned int".

All uniform_type_info instances are singletons. They are stored in two maps, one that associates instances with their platform-dependent name, and one that associates instances with their uniform name. Thus, a lookup at runtime is always of the complexity $O(log\, n)$. The two maps increase the overall system performance since a deserializer requires the uniform name, but the function uniform_typeid, see Section 4.9.1, Figure 21, needs to lookup the name returned by type_info::name. Demangling the platform-dependent name on each lookup would add an expensive overhead.

5.5.2 Announcement of User-Defined Types

Our approach automatized serialization of member variables as long as all members are built-in data types, strings or STL compatible containers which element types follow the same rules. These constraints are checked with metaprogramming utility classes. The first utility class identifies primitive data types. Our utility classes use the same conventions as the type traits of the standard template library (STL). The first utility class is implemented as follows.

```
1 template<typename T>
2 struct is_primitive {
3   static const bool value =
4           std::is_arithmetic<T>::value
5       || std::is_same<T,std::string>::value
6       || std::is_same<T,std::u16string>::value
7       || std::is_same<T,std::u32string>::value;
8 };
```

A data type is primitive if and only if it is one of the built-in integer or floating point types. This is the case if std::is_arithmetic<T>::value is true, or if it is one of the STL string classes. We do not support std::wstring since it is platform-dependent. Ensuring STL compatibility is not that trivial and requires some metaprogramming utilities first.

```
1 template<typename T> struct rm_ref { typedef T type; };
2 template<typename T> struct rm_ref<T const&> { typedef T type; };
3 template<typename T> struct rm_ref<T&> { typedef T type; };
4 template<> struct rm_ref<void> { };
```

The class `rm_ref` removes references, both const and non-const, from a given type but is explicitly not defined for `void`. We need this utility class in our next trait which evaluates whether a given type behaves like a forward iterator.

```
1  template<typename T> class is_forward_iterator {
2    template<class C> static bool sfinae_fun(
3      C* iter,
4      typename rm_ref<decltype(*(*iter))>::type* = 0,
5      typename enable_if<
6        std::is_same<C&,decltype(++(*iter))>>::type* = 0,
7      typename enable_if<
8        std::is_same<bool,decltype(*iter == *iter)>>::type* = 0,
9      typename enable_if<
10       std::is_same<bool,decltype(*iter != *iter)>>::type* = 0
11   )
12   { return true; }
13   static void sfinae_fun(void*) { }
14   typedef decltype(sfinae_fun((T*) nullptr)) result_t;
15  public:
16     static const bool value = std::is_same<bool,result_t>::value;
17 };
```

The class defines an overloaded, static member function. The second overload at line 13 takes a void pointer and has a void return type. The other overload in line 2 is a template function that has a non-default first parameter of type `C`. The second parameter has an undefined type if `C::operator*` is undefined or has a void return type. The next parameters in line 6 evaluates if the member function `C::operator++` is defined. The last two parameters evaluate if `C` supports the comparison operations `==` and `!=`. All parameters, except the first, have default values. Thus, the template function could be called with one argument. The typedef `result_t` in line 14 is defined as the return type of the overloaded member function `sfninae_fun`. The compiler chooses the overload based on the type of the parameter. A template function taking a pointer of type `T*` is always a better match than a function taking a pointer of type `void*`, since this would require an implicit conversion. However, the first overload in line 2 is only available if and only if `T` provides all required operations. Otherwise, the first overload is unavailable due to substitution failure. Finally, we set the static member variable `value` in line 16 to `true` if `result_t` is defined as `bool`, which is the return type of the first overload, otherwise `value` is `false`.

Based on this trait, we declared a type trait `is_iterable` that evaluates if a type `T` defines the two member functions `ConstIterator T::begin() const` and `ConstIterator T::end() const` where `ConstIterator` is a forward iterator. All containers in the STL are iterable, thus, detected by the trait `is_iterable`.

STL containers divide into two categories: lists and maps. Lists, such as `std::vector` and `std::list`, provide the member function `push_back` to add new elements. Maps, such as `std::map` and `std::set`, provide the member function `insert` instead. Thus, we defined the two traits `is_stl_compatible_list` and `is_stl_compatible_map`. Now, we are able to provide default implementations for strings, built-in types and any container providing an STL compatible interface. The correct implementation is chosen with `enable_if` and our declared type traits. The following source code shows a pseudo code implementation for a STL compatible list. A map implementation would be similar except that it would use `insert` rather than `push_back`.

```
1  template<typename T>
2  enable_if<is_stl_compatible_list<T>>
3  serialize(T const& list, serializer& sink) {
4    sink << list.size();
5    for (auto i = list.begin(); i != list.end(); ++i)
6      serialize(*i);
7  }
8  template<typename T>
9  enable_if<is_stl_compatible_list<T>>
10 deserialize(T& list, deserializer& source) {
11   size_t size;
12   source >> size;
13   for (size_t i = 0; i < size; ++i) {
14     T::value_type element;
15     deserialize(element);
16     list.push_back(element);
17   }
18 }
```

The implementation is a one-to-one mapping of our design in Section 4.9.2. Recursive data structures, such as a vector of vectors, are detected as well as primitive data types as the following usage example illustrates.

```
1  struct my_struct { std::vector<std::vector<int>> a; float b; };
2  int main() {
3      announce<my_struct>(&my_struct::a, &my_struct::b);
4      send(self, my_struct { {{1,2},{3,4}}, 5 });
5      receive(
6        on<my_struct>() >> [](my_struct const& value) {
7          // ...
8        }
9      );
10 };
```

5.6 Middle Man

A middle man has to fulfill three tasks. It receives, deserializes and forwards messages from remote peers to local actors, serializes and sends messages to remote actors, and creates local proxies for remote actors. Therefore, we have split the middle man in two distinct components.

5.6.1 Addresses of Middle Men

Middle men are singletons, why each process has exactly one middle man. A process is identified by its `process_information` containing the $nodeID$ and the $processID$ as specified in Section 4.10.1. A `process_information` also identifies a middle man instance since each process has exactly one middle man.

An actor address is specified as $(nodeID, processID, actorID)$, where $nodeID$ and $processID$ identify a middle man instance MM. Thus, an actor address also could be read as $(MMID, actorID)$.

5.6.2 Post Office

A *post office* PO is a software entity that covers two of the three tasks of a middle man. It creates proxy instances for remote actors and it receives messages from other nodes in the network. More precisely, it receives messages from other middle men. A PO only has read access to all sockets it has to remote nodes and runs in its own thread. Sockets are multiplexed by using `select` and non-blocking receive operations.

All proxy instances are stored in a multimap, a map that can store more than one value per key, with the ID of the middle man it belongs to as key. The PO creates a new proxy instance whenever it deserializes an actor address with an unknown, new $actorID$. As a result, the PO implicitly learns new remote actors.

5.6.3 Mailman

The mailman implements the third task of a middle man: serializing and transmission of messages to remote actors. It runs in its own thread and shares all sockets with the PO but

does only have write access to the sockets. It has a map that stores all sockets as values with the middle man address as key.

Each proxy instance forwards all messages it receives to the mailman with its own address as receiver. The mailman then selects the corresponding socket, serializes the message and sends the byte stream to the socket.

6 Measurements and Evaluation

In this section, we compare the runtime behavior of our implementation with the matured actor model implementations of Erlang and Scala - its standard library, as well as the Akka library. For this purpose we identified and implemented three different use cases to measure actor creation overhead, mailbox performance in N:1 communication scenarios and a use case simulating a realistic application behavior.

In detail, we implemented and compared all three algorithms using

libcppa (event-based)
: `C++`[19] with `libcppa` based on event-based message processing

libcppa (stacked)
: `C++` [19] with `libcppa` based on context switching

erlang
: Erlang in version 5.8.3

scala (akka)
: Scala[20] with the Akka library (event-based message processing)

scala (react)
: Scala [20] with the event-based actor implementation of the standard library

scala (receive)
: Scala [20] with the thread-mapped actor implementation of the standard library

The benchmarks ran on a virtual machine with Linux using 2 to 12 cores of the host system comprised of two hexa-core Intel® Xeon® processors with 2.27GHz.

[19] Compiled using the GNU `C++` compiler (g++) in version 4.6.1 with optimization level O3.
[20] Scala in version 2.9.1 running on a JVM using 4GB of RAM.

6.1 Measuring the Overhead of Actor Creation

Our first benchmark measures the overhead of actor creation. It recursively creates 2^{19} actors, as the following pseudo code illustrates.

```
1  spreading_actor(Parent):
2    receive:
3      {spread, 0} =>
4        Parent ! {result, 1}
5      {spread, N} =>
6        spawn(spreading_actor, self()) ! {spread, N-1}
7        spawn(spreading_actor, self()) ! {spread, N-1}
8        receive:
9          {result, X1} =>
10           receive:
11             {result, X2} =>
12               Parent ! {result, X1+X2}
13
14 main(X):
15   spawn(spreading_actor, self()) ! {spread, X}
16   receive:
17     {result, Y} =>
18       assert(2^X == Y)
```

Each actor spawns two further actors after receiving a `{spread, N}` message unless `N` is 0, in which case the actor sends `{result, 1}`. After spawning two more actors, an actor waits for two result messages, sends an result message of its own to its parent, and finishes execution. Thus, our benchmark program creates 2^X actors, where X is the argument passed to `main`.

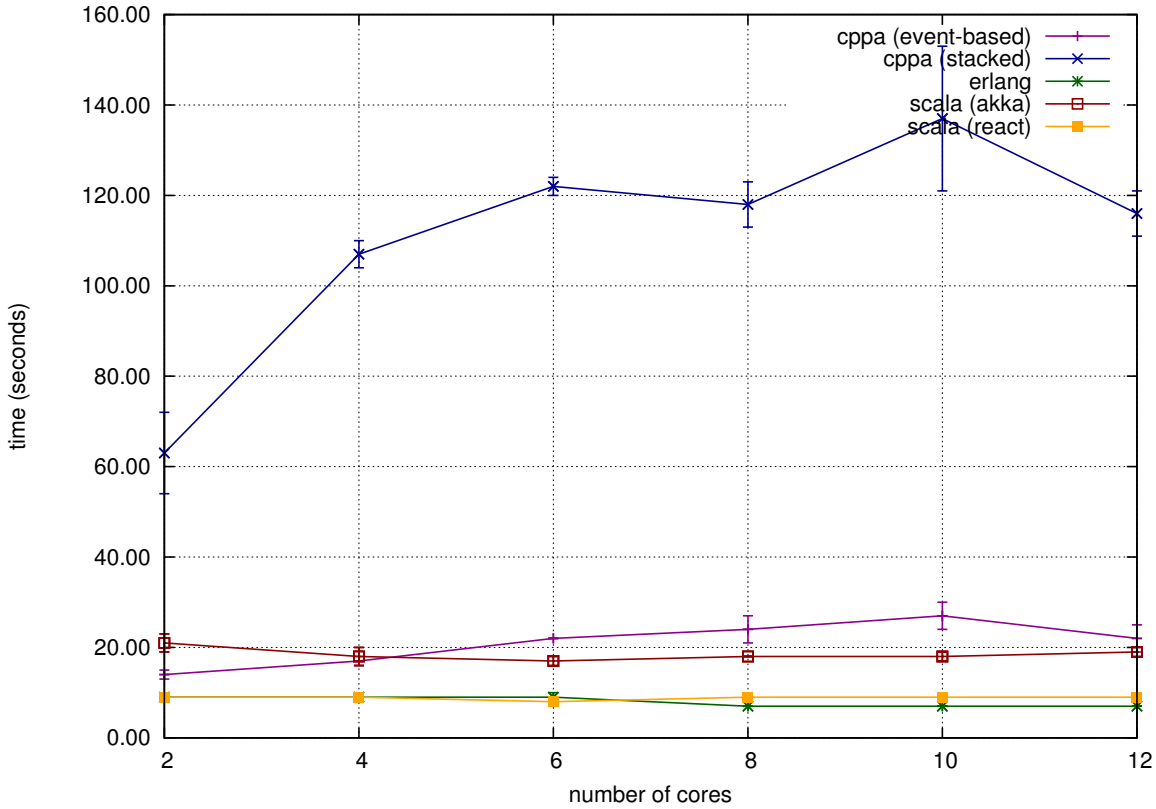

Figure 31: Actor creation performance for 2^{19} actors

In Figure 31, we display the actor creation time as a function of available CPU cores. This measurement tests how lightweight actor implementations are. The ideal behavior is a decreasing curve. We did not use test thread-mapped actor implementation of Scala, because the JVM cannot handle half a million threads. And neither could a native application.

In the outcome, two classes can be clearly identified. Both Scala implementations and Erlang have constant or decreasing time consumption, while `libcppa` has a more fluctuating time consumption that increases with concurrency.

It is not surprising that Erlang yields the best performance, as its virtual machine was build to efficiently handle actors. Furthermore, it is not surprising that the context-switching implementation of `libcppa` consumes more time than an event-based approach because of the overhead of stack allocation. Results indicate that the scheduling overhead caused by more hardware concurrency outweighs the benefit in `libcppa` for this scenario.

6.2 Measuring Mailbox Performance in N:1 Communication Scenario

Our second benchmark measures the mailbox performance in an N:1 communication scenario. We used 20 threads sending 1,000,000 messages each, except for Erlang which does not have a threading library. In Erlang, we spawned 20 actors instead. The minimal runtime of this benchmark is the time the receiving actor needs to process 20,000,000 messages and the overhead of passing the messages to the mailbox. More hardware concurrency leads to higher synchronization between the sending threads, since the mailbox acts as a shared resource.

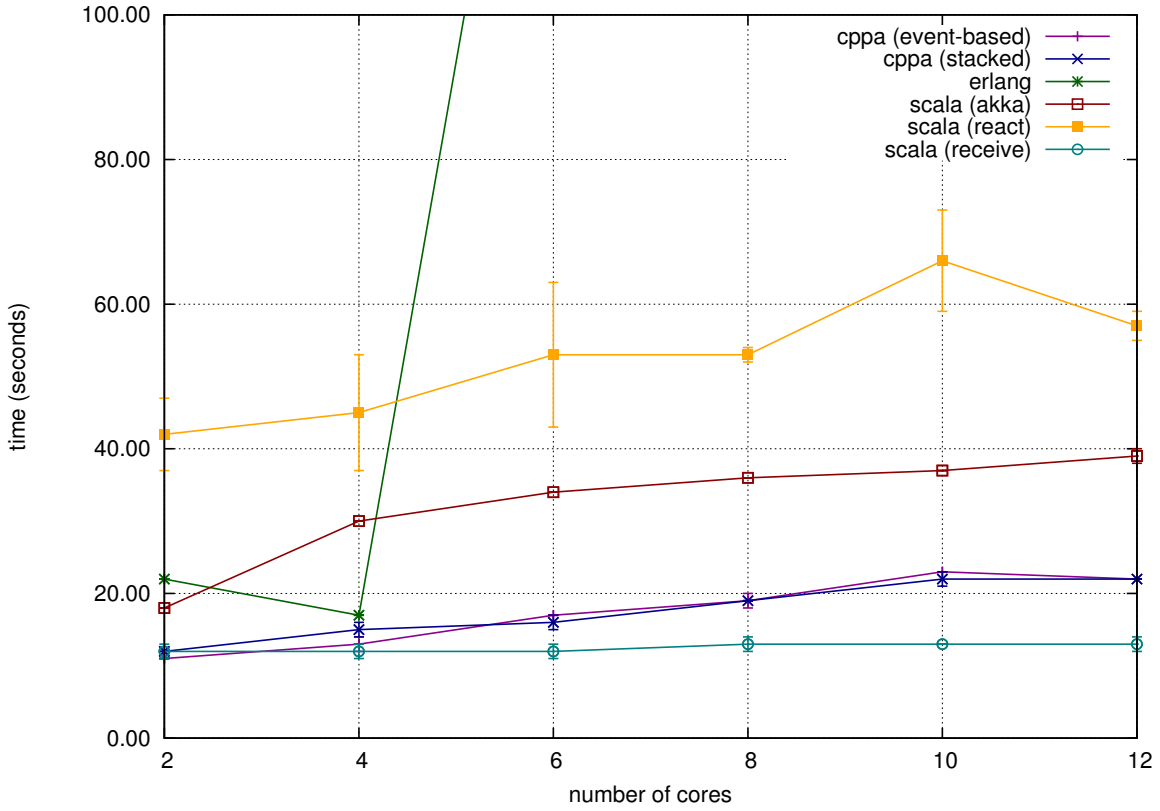

Figure 32: Mailbox performance in N:1 communication scenario

In figure 32, we display the time needed for the application to send and process 20,000,000 messages as a function of available CPU cores. The ideal behavior is a slowly increasing curve.

In the outcome, four classes can be identified. Scala (receive) approaches the optimal behavior, while the other Scala implementations show inferior performance. Both `libcppa` implementations show similar performance to Scala (receive) on two cores but have a faster

increasing curve. Erlang has an abrupt rise in runtime for more than 4 cores up to an average of 600 seconds on 12 cores. The results are clipped for visibility purposes in the graph.

Results indicate that our mailbox implementation using the cached stack algorithm, introduced in Section 5.2.3, scales very well though we could see the same increase in runtime due to the scheduling overhead as in our previous benchmark. The overhead of stack allocation is negligible in this use case. Thus, the run time of both `libcppa` implementations is almost identical. The message passing implementation of Erlang does not scale well for this use case. The more concurrency we add, the more time the Erlang program needs.

6.3 Measuring Performance in a Mixed Scenario

Our final benchmark simulates a more realistic use case with a mixture of operations. The continuous creation and termination of actors is simulated along with a total of more than 50,000,000 messages sent between actors and some expensive calculations are included to account for numerical work load. The test program creates 20 rings of 50 actors each. A token with initial value of 10,000 is passed along the ring and decremented once per iteration. A client receiving a token always forwards it to the next client and finishes execution whenever the value of the token was 0. The following pseudo code illustrates the implemented algorithm.

```
1  chain_link(Next):
2    receive:
3      {token, N} =>
4        next ! {token, N}
5        if (N > 0) chain_link(Next)
6
7  worker(MessageCollector):
8    receive:
9      {calc, X} =>
10       MessageCollector ! {result, prime_factorization(X)}
11
12 master(Worker, MessageCollector):
13   5 times:
14     Next = self
15     49 times: Next = spawn(chain_link, Next)
16     Next ! {token, 10000}
17     Done = false
18     while not Done:
19       receive:
20         {token, X} =>
21           if (X > 0): Next ! {token, X-1}
22           else: Done = true
23   MessageCollector ! {master_done}
```

Each ring consists of 49 `chain_link` actors and one `master`. The `master` recreates the terminated actors five times. Each `master` spawns a total of 245 actors $(5 * 49)$ and the program spawns 20 `master` actors. Additionally, there is one message collector and one worker per master. A total of 4921 actors $(20 + 20 * 245 + 1)$ are created but no more than 1021 $(20 + 20 + (20 * 49) + 1)$ are running concurrently. The message collector waits until it receives 100 $(20 * 5)$ prime factorization results and a *done* message from each master.

We calculated the prime factors of $28,350,160,440,309,881$ ($329,545,133$ and $86,028,157$) to simulate some work load. The calculation took about two seconds on the tested hardware in our loop-based `C++` implementation. Our tail recursive Scala implementation performed at the same speed, whereas Erlang needed almost seven seconds.

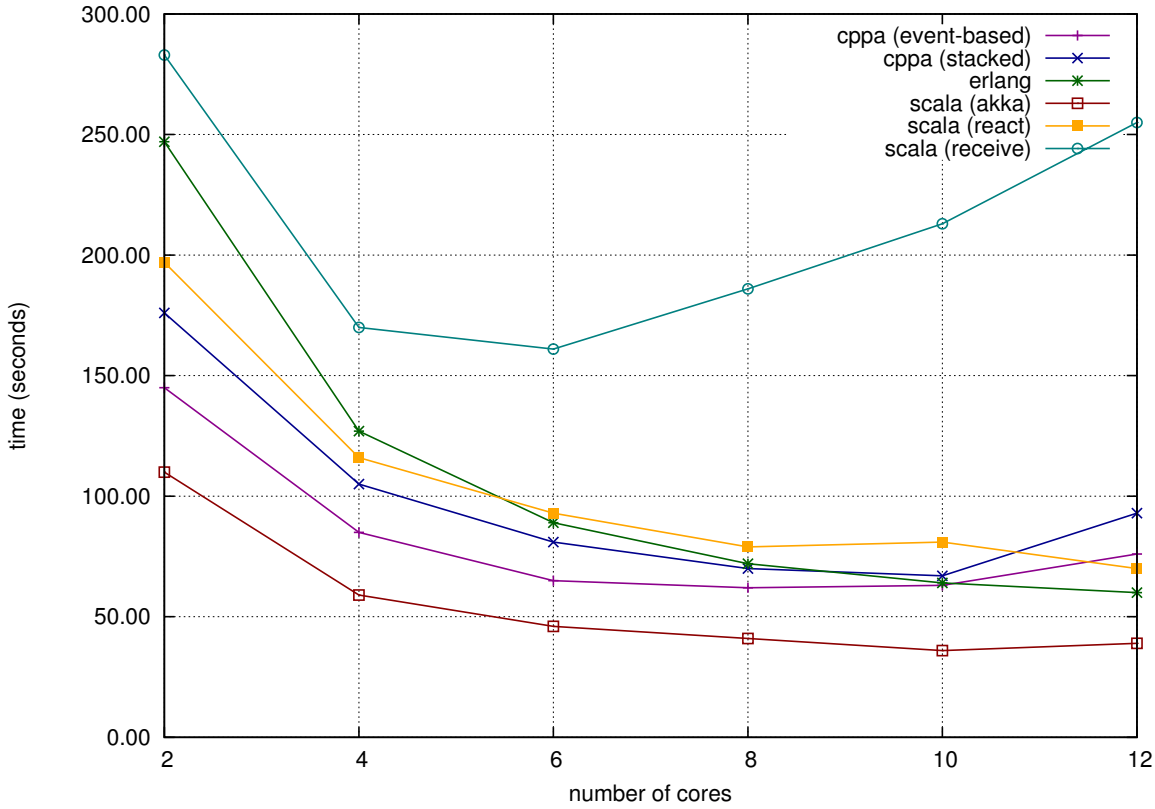

Figure 33: Performance in mixed use case

In figure 33, we display the runtime of the benchmark as a function of available CPU cores. The ideal behavior is a linear speed-up – doubling the number of cores should halve the runtime.

In the outcome, three classes can be identified. Erlang has an almost linear speed-up. Scala (receive), Scala (akka) and both `libcppa` implementations increase in runtime after reaching a global minimum. Scala (react) reaches a local minimum at 8 cores and accelerates again for 12 cores.

As expected, the thread-based Scala implementation yields the worst performance though the runtime increase for eight and more cores surprises. Akka is significantly faster than both standard library implementations of Scala and about 30% than `libcppa` (event-based) for up to six cores.

Erlang performs very well, given the fact that its prime factorization is more than three times slower. The very efficient scheduling of Erlang, which is the only implementation under test that performs preemptive scheduling, is best at utilizing hardware concurrency.

The overhead of the `libcppa` scheduler hinders better performance results. Especially the rise in runtime at 12 cores surprises. The overhead of stack allocation and context switching is about 10-20% in this benchmark for up to six cores where the scheduler is stretched to its limits.

6.4 Measurement Summary

We have shown that both `libcppa` implementations are competitive, though context-switching actors are not feasible for scenarios with a large number of actors. `libcppa` performs at comparable speed to well-tested and established actor model implementations. Nevertheless, its scheduling algorithm is not able to utilize more than six cores efficiently by now.

As expected, the event-based implementation is faster since it requires less scheduling and allocation overhead. Provided that the application does spawn a moderate number of actors, the context-switching actor implementation performs sufficiently.

7 Conclusion & Outlook

In this thesis, we have implemented a library that extends the C++ programming language with an actor semantic that provides an extensible, publish/subscribe-based group communication. C++ does not provide pattern matching as a language feature, but we were able to implement this feature in our library to provide a convenient way to implement actors. Actor communication is distribution transparent and, as first results indicate, fast. The cooperative scheduling is scalable and we were able to compete mature implementations of the actor model, though, the used scheduling algorithm hinders better results.

The pattern matching implementation of libcppa is not, and cannot be, as elegant as language based approaches, as the following comparison of two equivalent actor implementations illustrates. Both actors implement the spreading_actor of Section 6.1. The first implementation uses Erlang, the second context-switching actors of libcppa.

```
1  testee(Parent) ->
2    receive
3      {spread, 0} ->
4        Parent ! {result, 1};
5      {spread, X} ->
6        spawn(actor_creation, testee, [self()]) ! {spread, X-1},
7        spawn(actor_creation, testee, [self()]) ! {spread, X-1},
8        receive
9          {result, R1} ->
10           receive
11             {result, R2} ->
12               Parent ! {result, (R1+R2)}
13           end
14       end
15   end.
```

```
1  void stacked_testee(actor_ptr parent) {
2    receive (
3      on(atom("spread"), 0) >> [&]() {
4        send(parent, atom("result"), (uint32_t) 1);
5      },
6      on<atom("spread"), int>() >> [&](int x) {
7        any_tuple msg = make_tuple(atom("spread"), x-1);
8        spawn(stacked_testee, self) << msg;
9        spawn(stacked_testee, self) << msg;
10       receive (
11         on<atom("result"), uint32_t>() >> [&](uint32_t r1) {
12           receive (
13             on<atom("result"), uint32_t>() >> [&](uint32_t r2) {
14               send(parent, atom("result"), r1+r2);
15             }
16           );
17         }
18       );
19     }
20   );
21 }
```

Apart from C++ using more brackets, the receive statement in Erlang is more compact since it does not have to use lambda expressions. However, we come close to a language-based solution, though, `receive` is a trade-off. One the one hand, allocating a stack for each actor and performing context-switching causes a small runtime overhead, on the other hand, `receive`-based actor communication can be used in threads as well. This allows developers to easily extend existing, thread-based applications. From a developer point of view, spawning an actor differs from creating a thread only by the name of the function used[21], allowing developers to migrate seamlessly to more lightweight and scalable context-switching actors.

Applications aiming at hundreds of thousand actors surely should use event-based actors. We would recommend using event-based actors for any new application due to fewer overhead and better runtime results. Our event-based actors scale better and use less memory but have a slightly different API. However, a system can freely mix threads acting as actors, context-switching actors as well as event-based actors, since all actors use one single interface for sending messages, linking, monitoring and joining groups.

[21] The `boost` threading library as well as the C++ standard template library provide an API that is very similar to the `spawn` function of `libcppa`.

The following example is an event-based actor implementation of `spreading_actor` (see page 84) in Scala using the Akka library. The second example implements an equivalent actor in `C++` using the event-based API of `libcppa`.

```
1  import akka.actor._
2  case class Spread(value: Int)
3  case class Result(value: Int)
4  class AkkaTestee(parent: ActorRef) extends Actor {
5    def receive = {
6      case Spread(0) =>
7        parent ! Result(1)
8        self.stop
9      case Spread(x) =>
10       val msg = Spread(x-1)
11       actorOf(new AkkaTestee(self)).start ! msg
12       actorOf(new AkkaTestee(self)).start ! msg
13       become {
14         case Result(r1) =>
15           become {
16             case Result(r2) =>
17               parent ! Result(r1+r2)
18               self.exit
19           }
20       }
21   }
22 }
```

```
1  struct testee : fsm_actor<testee> {
2    actor_ptr parent;
3    behavior init_state = (
4      on(atom("spread"), 0) >> [=]() {
5        send(parent, atom("result"), (uint32_t) 1);
6        become_void();
7      },
8      on(atom("spread"), int) >> [=](int x) {
9        any_tuple msg = make_tuple(atom("spread"), x-1);
10       spawn(new testee(this)) << msg;
11       spawn(new testee(this)) << msg;
12       become (
13         on(atom("result"), uint32_t) >> [=](uint32_t r1) {
14           become (
15             on(atom("result"), uint32_t) >> [=](uint32_t r2) {
16               send(parent, atom("result"), r1+r2);
17               become_void();
18             }
19           );
20         }
21       );
22     }
23   );
24   testee(actor_ptr const& pptr) : parent(pptr) { }
25 };
```

Scala does have a pattern matching implementation similar to Erlang, making the source code more compact compared to our `on... >> lambda` approach. But besides that, we achieved our goal of providing an actor semantic for `C++`.

We provide a technology-independent group communication semantic in `libcppa`. So far, we implemented a technology module for in-process communication, see Section 5.4.2 for an example, but such a group interface aims at large-scale distributed applications based on the actor model. Furthermore, it allows developers to integrate existing publish/subscribe systems such as `D-Bus` seamlessly without dealing with technology-specific or platform-dependent APIs. We could not unfold the full potential of an “actor multicast model” yet. Our current development state is a solid basis for further implementations and application scenarios for such large-scale distribution that could scale up to globally distributed systems by using a group communication service layer for world-wide multicast such as HΛMcast (Waehlisch et al., 2011).

The actor model allows developers to implement scalable, concurrent applications of a high level of abstraction. However, currently available actor model implementations do not utilize the parallel processing power present in modern computer systems. SIMD[22] processing units, such as GPUs (graphical processing units), are known to outperform CPUs in computationally intensive applications such as encoding or decoding video streams and cryptography. OpenCL (Open Computing Language, Khronos OpenCL Working Group (2008)) is a programming standard for SIMD processing units that explicitly addresses high-performance computing using both GPU and CPU. OpenCL also could be used to access custom hardware operations (Jaskelainen et al., 2010). There is a `C++` API for developing OpenCL applications (Gaster, 2010). However, developing such applications requires specific code access and OpenCL-context management. A GPU has its own memory. Code and data are transferred to the GPU before executing an algorithm and the results must be read back after the computation is done. This management could be done by `libcppa`. A SIMD actor could define its behavior based on patterns but would have to provide an OpenCL compatible implementation. Such actors would be scheduled on the GPU rather than on the cooperatively used thread pool. Executing actors on a GPU would enable `libcppa` to address high-performance computation applications based on the actor model as well.

[22] SIMD (Single Instruction, Multiple Data) processing units perform operations on multiple data simultaneously.

References

Abrahams, D., Collings, G., Colvin, G., and Dawes, B. (2001). *Smart Pointer Timings*. Boost Library Documentation
URL http://www.boost.org/libs/smart_ptr/smarttests.htm.

Agha, G. (1986). Actors: a model of concurrent computation in distributed systems. Technical Report 844, MIT, Cambridge, MA, USA.

Armstrong, J. (1996). Erlang - A survey of the language and its industrial applications. In *Proceedings of the symposium on industrial applications of Prolog (INAP96)*, pages 16–18. Hino.

Armstrong, J. (1997). The development of Erlang. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 196–203, New York, NY, USA. ACM. ISBN: 0-89791-918-1.

Armstrong, J. (2003). *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden.
URL: http://erlang.org/download/armstrong_thesis_2003.pdf.

Armstrong, J. (2007). *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf. ISBN-10: 193435600X, ISBN-13: 978-1934356005.

Conway, M. E. (1963). Design of a separable transition-diagram compiler. *Commun. ACM*, 6:396–408. ISSN: 0001-0782.

Coplien, J. O. (1995). Curiously recurring template patterns. *C++ Report*, 7:24–27. ISSN: 1040-6042.

Dechev, D., Pirkelbauer, P., and Stroustrup, B. (2006). Lock-Free Dynamically Resizable Arrays. In Shvartsman, A. A., editor, *OPODIS*, volume 4305 of *Lecture Notes in Computer Science*, pages 142–156. Springer. ISBN: 3-540-49990-3.

Deering, S. (1989). Host extensions for IP multicasting. RFC 1112, IETF.

Earle, C. B., Fredlund, L.-A., and Derrick, J. (2005). Verifying fault-tolerant Erlang programs. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 26–34, New York, NY, USA. ACM. ISBN: 1-59593-066-3.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131. ISSN: 0360-0300.

Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley. ISBN-10: 0201633612 ISBN-13: 978-0201633610.

Gaster, B. R. (2010). *The OpenCL C++ Wrapper API*. Khronos Group. `http://www.khronos.org`.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition.

Haller, P. and Odersky, M. (2006). Event-Based Programming without Inversion of Control. In *Joint Modular Languages Conference*, Lecture Notes in Computer Science, pages 4–22. Springer-Verlag Berlin. ISBN: 3-540-40927-0.

Haller, P. and Odersky, M. (2009). Scala Actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220.

Hansen, P. B. (1973). *Operating system principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN: 0-13-637843-9.

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, pages 235–245, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Hickey, R. (2011). Clojure programming language. `URL http://clojure.org`.

Holbrook, H. and Cain, B. (2006). Source-Specific Multicast for IP. RFC 4607, IETF.

I.B.M. Corporation (1983). *IBM System/370 Extended Architecture, Principles of Operation*. IBM Publication No. SA22-7085.

ISO (2011). Programming languages - C++. Standard 14882:2011, ISO/IEC Information technology, Geneva, Switzerland.

Jaskelainen, P. O., de La Lama, C. S., Huerta, P., and Takala, J. H. (2010). OpenCL-based design methodology for application-specific processors. In *ICSAMOS*, pages 223–230. IEEE. ISBN: 978-1-4244-7936-8.

Khronos OpenCL Working Group (2008). *The OpenCL Specification, version 1.2*. `UTL: http://www.khronos.org/opencl`.

Leach, P. J., Mealling, M., and Salz, R. (2005). A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, IETF.

Mason, A. (2011). Theron homepage.
`http://www.theron-library.com/`.

Mellor-Crummey, J. M. and Scott, M. L. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65. ISSN: 0734-2071.

Meyers, S. and Alexandrescu, A. (2004). C++ and the Perils of Double-Checked Locking.
`URL http://drdobbs.com/cpp/184405726`.

Michael, M. M. and Scott, M. L. (1996). Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, USA. ACM. ISBN: 0-89791-800-2.

Microsoft (2011). Fibers.
`URL http://msdn.microsoft.com/en-us/windows/ms682661`.

Odersky, M. (2011). The Scala Language Specification Version 2.9.
`URL: http://scala.epfl.ch/docu/files/ScalaReference.pdf`.

Reynolds, J. C. (1972). Definitional interpreters for higher-order programming languages. In *Reprinted from the proceedings of the 25th ACM National Conference*, pages 717–740. ACM.

Srinivasan, S. (2011). Kilim homepage.
`URL http://www.malhar.net/sriram/kilim/`.

Srinivasan, S. and Mycroft, A. (2008). Kilim: Isolation-Typed Actors for Java. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 104–128, Berlin, Heidelberg. Springer-Verlag. ISBN: 978-3-540-70591-8.

Stroustrup, B. (1995). *The design and evolution of C++*. Addison-Wesley. ISBN: 978-0-201-54330-8.

Sutter, H. (2008). Writing a Generalized Concurrent Queue.
`URL http://drdobbs.com/cpp/211601363`.

TIOBE software (2012). Programming Community Index.
`URL http://www.tiobe.com`.

Typesafe Inc. (2011). Akka homepage. `URL http://akka.io/`.

Vandevoorde, D. and Josuttis, N. M. (2002). *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 1 edition. ISBN: 9780201734843.

Waehlisch, M., Schmidt, T., and Venaas, S. (2011). A Common API for Transparent Hybrid Multicast. Internet-Draft – work in progress 03, IETF.

List of Figures

1 Simple publish/subscribe system . . . 6
2 Observer pattern . . . 7
3 Source-specific publish/subscribe system . . . 8
4 Receive loop using patterns . . . 9
5 Linking of actors . . . 11
6 Supervision tree . . . 16
7 *one_for_one* strategy . . . 16
8 *one_for_all* strategy . . . 16
9 *rest_for_one* strategy . . . 16
10 `javac` output post-processed by Kilim `weaver` (Srinivasan and Mycroft, 2008) . . . 19
11 Smart pointer timings for GCC (Abrahams et al., 2001)) . . . 22
12 Smart pointer timings for MSVC (Abrahams et al., 2001)) . . . 22
13 Base class for reference counting . . . 23
14 `channel` interface . . . 29
15 Actor interfaces with inheritance . . . 30
16 Event-based actor classes . . . 34
17 Simple finite-state machine . . . 35
18 Copy-on-write tuples with `C++` signatures . . . 37
19 Group and related classes . . . 40
20 Class diagram containing all serialization related classes . . . 41
21 Related functions for `unfiorm_type_info` and `object` . . . 42
22 Communication to a remote actor . . . 44
23 Links in a distributed system . . . 45
24 Generic queue interface . . . 62
25 Inconsistent queue state with unreachable tail . . . 63
26 Temporary inconsistent queue state . . . 64
27 Enqueue operation in a cached stack . . . 65
28 Dequeue operation in a cached stack . . . 65
29 Queue benchmark using 4 threads . . . 66
30 Queue benchmark using 8 threads . . . 66
31 Actor creation performance for 2^{19} actors . . . 85
32 Mailbox performance in N:1 communication scenario . . . 86
33 Performance in mixed use case . . . 89

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw.§24(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 16.01.2012 Dominik Charousset