FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Concurrent Programming with Actors and Microservices

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Maximilian Irro, BSc BSc
Matrikelnummer 01026859

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam

Wien, 30. September 2018

Maximilian Irro	Franz Puntigam

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Concurrent Programming with Actors and Microservices

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Maximilian Irro, BSc BSc
Registration Number 01026859

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam

Vienna, 30th September, 2018

Maximilian Irro	Franz Puntigam

Erklärung zur Verfassung der Arbeit

Maximilian Irro, BSc BSc
Pfalzstraße 10, 5282 Ranshofen

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. September 2018

Maximilian Irro

Danksagung

An dieser Stelle möchte ich einigen wenigen meinen Dank aussprechen:

- Meiner Familie, die mich in meinem Studium immer unterstützt haben, obwohl ich viel zu selten anrufe.
- Meinem Betreuer, Franz Puntigam, für die guten Ratschläge durch welche diese Arbeit zustande kam.
- Theresa und Alex, die sich die Mühe gemacht haben diese Arbeit Korrektur zu lesen.

Abstract

Common problems require applications to manage multiple concerns simultaneously. A convenient approach is the concept of *concurrent programming*. In this thesis, we investigate two different models for introducing concurrent computational units into software architectures. One approach is the *actor model* that defines theoretically well-known constructs supporting concurrent, parallel and distributed execution in a transparent way. The other approach is an architectural style based on *microservices*, a recent trend that gained academic and industrial popularity. Microservices facilitate many principles of the old Unix philosophy by composing complex functionality through small, independent, highly cohesive and loosely coupled executables. These programs interoperate via lightweight, technology-heterogeneous messaging channels. The deployment modality of microservices conceives concurrent execution through the operating system scheduler. This thesis compares the programming of concurrent computation through actors and microservices with respect to a non-trivial concurrent system scenario. We argue that both approaches share many conceptual similarities and show few but significant differences. Both models have the same expressive capabilities regarding concurrent programming concerns like communication and scalability, but are subject to different trade-offs. We provide implementations of the system scenario based on actor and microservice architectures. Benchmark results of these implementations suggest that actors provide better system efficiency through a smaller codebase. Microservice architectures consume significantly more system resources and suffer especially from purely synchronous communication mechanisms.

Kurzfassung

Applikationen benötigen häufig eine simultane Bearbeitung mehrerer Aufgaben. *Nebenläufige Programmierung* ist hierfür ein verbreitetes Konzept. Diese Arbeit beschäftigt sich mit zwei Modellen zur Definition nebenläufiger Programmeinheiten innerhalb von Softwarearchitekturen. Eines dieser Modelle ist das *Actor Model*. Es definiert theoretisch wohlfundierte Konstrukte, welche transparent eine nebenläufige, parallele und verteilte Ausführung ermöglichen. Bei dem anderen Modell handelt es sich um einen relativ neuen Architekturstil unter Verwendung von *Microservices*, welche sich kürzlich im akademischen und industriellen Umfeld großer Beliebtheit erfreuen. Microservices bauen auf viele Prinzipien der alten Unix-Philosophie, indem sie komplexe Funktionalität durch den Zusammenschluss kleiner, unabhängiger, kohäsiver und lose gekoppelter Programme konzipieren. Diese Programme interagieren über leichtgewichtige, auf Nachrichtenaustausch basierende, technologisch heterogene Kommunikationskanäle. Microservices unterliegen einer implizit nebenläufigen Ausführungsmodalität durch den Prozess-Scheduler des Betriebssystems. Diese Arbeit vergleicht die Programmierung von nebenläufiger Ausführung mittels Actors und Microservices relativ zu einem nichttrivialen Fallbeispiel eines nebenläufigen Systems. Wir argumentieren, dass beide Ansätze viele Gemeinsamkeiten und wenige aber wichtige konzeptionelle Unterschiede besitzen. Beide Modelle haben gleichwertige Möglichkeiten um typische Anliegen der nebenläufigen Programmierung wie Kommunikation und Skalierbarkeit auszudrücken. Jedoch unterliegen die Modelle unterschiedlichen Trade-offs. Wir stellen Implementierungen des Fallbeispiels bereit, welche jeweils auf Actors bzw. Microservices basieren. Die Resultate eines Benchmarkings dieser Implementierungen legen nahe, dass Actors eine bessere Systemeffizienz verbunden mit einer kleineren Codebasis ermöglichen. Microservice-Architekturen hingegen konsumieren erheblich mehr Systemresourcen und leiden vor allem unter den Auswirkungen rein synchroner Kommunikationsmechanismen.

To whom it may concern

Contents

1 Introduction

I think the computer is the world's greatest toy. You can invent wonderful things and actually make them happen.
— Butler Lampson

The physical world is a composition of simultaneous activities. Programmers experience nature as a concurrent environment. As such, the idea of simultaneous actions has not been absent from the intangible world of computer programming. Numerous models to conceive concurrent execution have been proposed over the decades. Now that many-core machines are widespread and distribution is popular in the current trend of *cloud computing*, concurrent programming techniques have become essential. Many of the proposed models are therefore now heavily applied in practice.

In this thesis, we pay attention to two approaches toward concurrent programming. The first approach is the *actor model* [59], a decade-old model dedicated to express concurrency. The second approach is based on the *microservice paradigm* [42], originally an architectural style for software systems that adds concurrent execution implicitly.

1.1 Problem Statement

Dragoni *et al.* [34] point out that there is yet a gap in the literature that emphasizes the connections of the actor model and the microservice model. This work aims to fill this gap, with a focus on the concurrent programming aspects of these two concepts. Specifically, we ask the following research questions:

RQ1 Why do actors and microservices qualify for programming concurrency?

RQ2 How do the actor and the microservice model facilitate concurrent execution?

RQ3 What are the expressive capabilities of actors and microservices regarding concurrent programming concerns?

RQ4 How does the performance of actors and microservices compare in a multi-core environment relative to a concurrent system scenario?

1.2 Methodological Approach

We conduct our research using the following methodological steps:

1. Identify the key characteristics and resulting model semantics of actors and microservices through literature review.
2. Define a non-trivial scenario for a concurrent system.
3. Develop two implementations of this scenario. One implementation is based on actors and the other implementation is based on microservices.
4. Using the knowledge gained from implementing the systems, evaluate the expressive capabilities of both models.
5. Perform an efficiency benchmark of both system implementations.
6. Evaluate the benchmark results, and answer the research questions.

1.3 Structure of the Thesis

This thesis has the following structure: Chapter 2 discusses *concurrency* in general with a focus on the concerns relevant for our subsequent discussion. Chapter 3 introduces the *actor model of computation* and subsequently Chapter 4 the *microservice architecture style*. Chapter 5 concerns programming with actors and microservices, where Section 5.1 describes a scenario system we implement, Section 5.2 the implementation strategies with actors, and Section 5.3 the implementation strategies with microservices. Chapter 6 evaluates both programming models based on the implementations, where Section 6.1 concerns the expressiveness of the models, and Section 6.2 their efficiency. Chapter 7 gives our conclusive view.

2 Concurrent Computation

What matters for simplicity is that there's no interleaving.
— Rich Hickey

Computation is conceived through the execution of instructions. We call activities *sequential*, if their list of atomic statements execute in a sequential manner. Given two or more sequential activities are executing either pseudo-simultaneously (in alternation on a single processing unit), or truly simultaneously (on multiple processing units), they interleave and we therefore call these activities *concurrent*. Interleaving weakens the total ordering from sequential actions to a merely partial ordering. As a result, concurrency is *nondeterministic*. Repeated invocations on the same input can result in different outputs in general [6,18,122]. In this chapter, we cover the foundational concerns of concurrency, the distinction to parallel and distributed computing, correctness properties, and different kinds of programming abstractions.

The overall requisite for every kind of concurrency is the simultaneous presence of multiple active computational units. Depending on the context in which scholars discuss concurrency, they established different terminologies. The programming language level often uses the term *thread* for the concurrent unit. Concurrency theory uses the *process* construct in general [122]. However, the term *process* interferes with other notions of executable units we discuss in due course. In order to be able to refer to different concepts without aliasing among the terminologies, we follow the suggestion of Ben-Ari [18] and denote abstract units of activity as *tasks* throughout the remainder of this thesis. This designation is an homage to the Ada programming language, where *task* refers to an activity associated with a concurrent unit. The term is well-known and offers a neutral way to refer to every kind of concurrently executed computation within a logical unit.

2.1 Foundational Issues

Many different notions of concurrency exist. Regardless of the chosen approach, we always have to pay attention to three basic concerns [12]:

Expression of concurrent execution

Concurrent computation must be *indicated*. The literature proposes various abstractions and subsequently numerous implementations exist in practice. We discuss some of these abstractions in due course. In general, all concurrency abstractions need to provide the possibility to define tasks as well as manage the tasks [14]. Examples are *channels*, *coroutines*, *fork and joins*, *futures* and *threads*. The interfaces for the creation of tasks can be arbitrary, e.g. as primitives directly within a programming language, as libraries and through operating system calls.

Communication

Tasks must be able to interact and cooperate. *Communication* allows tasks to influence each other [12]. The *shared state* communication style rests on commonly accessible memory (e.g. variables, objects, etc.). Several tasks then interact by reading and writing to the same state. In contrast, *message passing* communication forgoes any access to shared data. Instead, it builds on the exchange of messages (fixed, immutable data) sent through communication *links*. The links are additionally required elements. Shared memory is only possible among tasks which gain access to a common memory section. A shared physical machine is the most basic form to get access to shared memory between tasks. A network can also simulate a mutual memory region. Message passing on the other hand is not even concerned by the locality or remoteness of memory. Messages can transit numerous links between sender and recipient. Therefore, messages easily bypass machine boundaries [36]. Message passing can happen in either *synchronous* fashion (messages are sent and the execution delays until the response is available) or *asynchronous* fashion (the execution resumes immediately after sending a message) [12].

Synchronization

Although concurrent execution has merely a partial ordering, communication still requires some ordering constraints. We must perform an action before we can detect its effect. *Synchronization* refers to mechanisms we use to ensure such constraints [12,97]. *Semaphores*, *locks* and *transactional memory* are prominent examples. The literature mostly discusses synchronization techniques regarding the shared state scenario, since the communication requires state modification by the sender before the receiver is allowed to read the information. Also, only one task can modify state at a time to avoid unexpected behavior due to low-level data races. The modification or evaluation of shared state occurs within a *critical section* or *critical region*. Synchronization mechanisms realize *mutual exclusion* where no task can access the shared state while another is within its critical region [18,122].

Message passing on the other hand provides an implicit form of synchronization.

Intrinsically, a message must be sent before it can be received. As a result, the semantics of message passing constrains the order by design [12]. Synchronous passing additionally constraints the sender from resuming its computation until it receives an answer.

There are two kinds of synchronization mechanisms. *Enforced primitives* guarantee no access to the state outside the primitive, thus ensuring order. *Unenforced primitives* grant a certain degree of freedom in their usage and therefore provide no guarantee of mutual exclusion, as do e.g. semaphores [36].

2.2 Concurrency, Parallelism and Distribution

So far, we have discussed concurrency as a generic term that denotes a simultaneous execution of activities. Yet there are more diverse notions that regard the implementation of concurrent execution. A strict distinction in our terminology is therefore in order.

We use *concurrency* to refer to the multiplexing of multiple tasks among one or more processors. We cannot make more specific assumptions in general. On a single *central processing unit* (CPU), the interleaving of computation is merely pseudo-simultaneous via time slicing, since all computation is still sequential [118].

Parallelism refers to truly simultaneous execution on different CPUs. Whether parallel execution is possible depends on the concurrency abstraction and its implementation [14]. On a programming language level, referencing components is usually subject to physical limitations regarding the program's memory. For example, objects can only reference other objects that are inside the memory space of the same program in general [97]. We regard a notion of concurrent objects that is able to surmount this restriction in due course. This limitation on memory space does not prevent us from writing concurrent code in general. But the limitation certainly complicates the writing of parallel code, e.g. when we use shared state communication. Parallel execution requires code execution on different CPU cores *at the same time*, which usually means distinct process memory boundaries. Inter-component communication must happen across memory and process boundaries. If a programming language uses a *virtual machine* (VM) to execute code, we can charge transparent inter-component communication across boundaries to this VM. For example, the *Java Virtual Machine* (JVM) has different approaches to implement threads. One is to map Java threads to system processes for parallelization [54]. The JVM hides the resulting gap in memory sections transparently. Writing explicit parallel code, e.g. with a *Fork/Join* framework, can be painful and requires us to explicitly prepare code segments and data that we can process in parallel [89,122].

Distributed computation is regarded by the literature as its own research discipline separate from parallel computation. However, both concepts build on the same fundamental idea: Truly concurrent execution (as in *at the same time*) of physically distributed tasks [5]. Agha, Frølund and Kim formulated a simple and sound argumentation [6]:

"In a parallel computation some actions overlap in time; by implication these events must be distributed in space."

This argument suggests that every parallel task is also a distributed task in a certain sense. The major distinction is that we expect parallel tasks to be physically closer to each other (same CPU) than distributed tasks (distinct CPUs and machines). Due to this distance, distributed tasks cannot share main memory directly [5,25]. Distribution therefore relies on message passing communication over the network. Of course, we can use the network to create abstractions for shared memory, so-called *spaces*. One example for a space is the *Linda* model [18,122].

Due to the physical separation, a subset of all distributed tasks executes on different locations (host machines) in general. We also refer to these hosts as *nodes* [18]. A single node can have one or more processors on which one or more tasks run concurrently. As a result, we make three fundamental observations about the interrelations of concurrency, parallelism, and distribution:

1. Concurrent systems can be parallel and distributed.
2. Parallel and distributed systems are inherently concurrent.
3. Distributed systems with two or more nodes are parallel.

Baeten [15] gives a general definition that incorporates these interrelations and which reflects our view on concurrency within this thesis:

"Concurrency theory is the theory of interacting, parallel and/or distributed systems."

In subsequent sections, we pay attention to two selected task abstractions. Both conceive concurrent computation in general. Additionally, they are able to provide parallelization and even distribution in a transparent way.

2.3 Correctness Properties

On a fundamental level, the basic purpose of every program is the computation of a result. From a sequential program, we always expect the same results for the same inputs[1]. We can verify the correctness of a program using theoretical methods, although these methods are not widely adopted in practice. For concurrent programs, many standard verification techniques do not hold anymore, due to the intrinsic interleaving of computations [18].

Many different issues regarding concurrent computation are well-known in the literature. Examples are deadlocks, livelocks, starvation, race conditions, mutual exclusion and

[1]All sorts of side effects, like IO, are also forms of input to a program and must be stable as well.

fairness. Due to the high-level view on concurrency in this thesis, we do not immerse into detailed discussions on each of these issues, as we often find it in other literature that concerns concurrency concepts. Here, simply two types of properties are relevant to us. Both types have an effect on the correctness of concurrent programs. We can classify all the issues we gave above in terms of these two property types [25,127,133]:

Safety
: asserts the operations that are allowed (safe) to be performed. As a result, given correct inputs result in correct outputs, while the computation never enters an undesired state. Examples of safety properties are race conditions and mutual exclusion. Informally, safety guarantees that "nothing bad will happen".

Liveness
: asserts the operations that have to be performed, such that a certain state will be reached eventually (progress). In other words, if we provide correct inputs, we have the guarantee for correct outputs in finite time (cf. termination in sequential programs). Examples of liveness properties are fairness, starvation, and reliable communication. Informally, liveness guarantees that "something good will happen".

Safety is *invariant*, such that a property `P` holds in *every* state of *every* execution. In contrast, liveness of `P` demands that the property holds in *some* state of *every* execution. As a result, safety and liveness have a so-called *duality* relationship. The negation of a member of one type is a member of the other [18]. Safety relates to *partial correctness* (the result is correct if the program terminates). Liveness on the other hand relates to *total correctness* (the programs terminates with a correct result) [25].

We have found *deadlocks* (blocking operations which for some reason do not unblock) to have a controversial role. On the one hand, an execution path must not lead into a deadlock (safety), while a deadlock also threatens the progression of a program, thus its liveness. In contrast, so-called *livelocks* (loops never meeting their termination condition) are, as the name suggest, clearly relate to liveness. The operations of a livelock are safe while the program does not progress.

2.4 Programming Abstractions

Most programs are concurrent in some way. An example of *implicit concurrency* is *input/output* (IO). There, we trigger IO devices to perform operations simultaneous to the executing program [25]. Also, compilers and interpreters exploit the concurrency inherent to a language's constructs. On the other hand, *explicit concurrency* must be indicated. We require appropriate programming abstractions. In general, we require concurrency abstractions to be powerful and expressive models, fit harmoniously into the programming language in terms of their interface, and exploit the underlying hardware resources efficiently [126].

2.4.1 Language-Construct Approach

Many different approaches to explicitly express concurrent computation on a programming language level were proposed over the decades and are now in use. A programming language either provides concurrency constructs by design, or we can utilize such constructs through libraries and frameworks [18,122]. Therefore, most concurrent task abstractions are available in most programming languages. The implementation part of this thesis in Chapter 5 is in the context of Java and its virtual machine. A brief discussion of Java's basic approach towards concurrency is in order, since alternative abstractions have to build on it.

Case Study: Concurrency in Java

Java is an object-oriented programming language with a C-inspired syntax for the JVM. The language expresses concurrency via threads and offers basic concepts to manage the access to shared resources. We define concurrent computation through the `Runnable` interface. The default implementation of `Runnable` is available in the `Thread` class [44,46]. The following example illustrates the principle approach:

```
class State {
    public int x = 0;
}

final State s = new State();
final Runnable task = () -> {
    final String name = Thread.currentThread().getName();
    synchronized(s) {
        s.x += 1;
        out.println(name + " " + s.x);
    }
};

new Thread(task).start();
new Thread(task).start();
```

The `synchronized` primitive allows us to express mutual exclusion to a shared resource whenever concurrent modification to this resource is possible. In this example, `s` denotes some state. Two threads have access to `s` through the scope of the `Runnable` lambda. Note that though we declared `s` as `final`, its publicly visible member `x` remains mutable.

The mechanism behind Java's synchronization is *locking* on a common reference among all involved threads, the so-called *monitor* object [44,46]. When we use `synchronized` as a block construct, we must provide this monitor as an argument. In our example, the state variable simply doubles as the monitor in addition to being the shared resource. Alternatively, we could have used the `synchronized` keyword also as a part of the signature of a method in `State` which holds the logic. A `synchronized` method signature is equal to `synchronized(this)` around the whole method's body, where `this` refers to the object `s`. The method's object reference then acts as the monitor, just as in our example.

A more modern alternative towards synchronization is the `Lock` interface. The previous `synchronized` keyword is an enforced synchronization primitive. The locks of `synchronized` are always exclusive to one thread at a time. In contrast, the various implementations of `Lock` do not need to be enforced. `Lock`s can therefore offer fine-grained options to control the locking, e.g. for simultaneous access of multiple readers [44,132]. To provide this degree of freedom, Java neither detects shared state nor requires its synchronization per se. As a result, programmers can easily introduce data races when they simply omit access control. Alternative concurrency abstractions for Java, e.g. through libraries and frameworks, always have to take this into account.

Expressing concurrency on the programming language level has its perils due to the overlapping of language concepts. We have already demonstrated the introduction of mutable state via scopes and visibility. Many concepts have an influence on concurrency considerations. Shared mutability and synchronization especially require utmost care of the programmer for handling access to the data.

2.4.2 Operating System Approach

Operating systems (OS) use the *process* as their computational model. A process describes the instantiation of a program image with associated *resources* (e.g. memory). Processes express dynamic execution. In the most basic case, a single processor alternately executes these computational units. *Scheduling* is the activation and passivation of processes and it is in the responsibility of the operating system. Scheduling results in a quasi-parallel execution of active processes. If we utilize multiple processors, the execution is truly parallel [14,122]. As a result, we can state that processes are inherently concurrent units due to their execution modality.

In contrast, *threads* are tasks *inside* a process. One process can have numerous threads which all share the same memory space [134]. Since threads within the same process have access to the same memory locations, we are free to introduce shared state among them. In the Java case study, we already demonstrated that shared state requires synchronization. In contrast to the JVM, an operating system strictly enforces the memory boundaries between processes. Communication between two processes requires either an explicit arrangement of so-called *shared memory*, or another means of message passing communication which we subsume as *inter-process communication*[2] (IPC) [14,94]. We extend our focus on OS-conceived concurrent tasks which rely on IPC in due course. A consolidating example for future reference is in order.

[2]In concurrency theory, *process* is also the general term for a concurrent unit, as we have mentioned. Therefore, the literature often denotes all communication between concurrent units as *inter-process communication*. To avoid confusion, we use the IPC designation only for communication mechanisms between OS processes.

Case Study: Concurrent Processes in C

Only with C11[3] did the programming language add native support for the expression of concurrency via threads. Prior to this, programmers had to use more operating system depending approaches like the POSIX[4] threads binding `pthreads`. An additional strategy was to compose concurrent computation in an ad hoc way by relying on the operating system's scheduler to assign processes to processors in a concurrent way [14,36].

An operating system allows a process to spawn new processes. The process we call the *parent* uses system calls that the OS provides to spawn a new process we call the *child*. For example, the `exec`-family of Unix calls allows us to instantiate arbitrary processes from executables we reference through a filesystem path. However, the new child replaces the parent process. `exec`-calls alone are therefore not insufficient to compose concurrency [94]. The expedient path is the alternative `fork`-call. It replicates the current process's address space into a separate address space of a new child [14,36]. The following example illustrates the control flow:

```c
void
parentBehavior(int fd[]);

void
childBehavior(int fd[]);

int
main(void)
{
    int fd[2];
    pipe(fd);

    pid_t pid = fork();

    if (pid == 0)
        childBehavior(fd);
    else
        parentBehavior(fd);
}
```

Both processes are based on the same program image. The parent receives the *process identifier* (PID) of the child process as the result of `fork()`. The child does not receive its own PID information. We can therefore use the PID to distinguish the further program flow of both processes. This mechanism effectively supports two separate behaviors. Consecutive forks are possible of course.

[3]The C standard revision of 2011, specifically *ISO/IEC 9899:2011*. It succeeds C99, the 1999 revision.

[4]**P**ortable **O**perating **S**ystem **I**nterface, a collection of standardized programming interfaces for operating systems. The **X** is a remnant from the original draft name *IEEE-IX*.

By using additional Unix system calls, we install a so-called *pipe* as the IPC between the two processes. Pipes are a form of byte stream across the memory boundaries of the respective processes [14]. Since Unix follows an *everything is a file* design principle, two file descriptors symbolize the endpoints of the pipe. This principle makes the interfaces simple and consistent [126]. The first descriptor `fd[0]` is in read mode and the second `fd[1]` in write mode. For example, the `parentBehavior` writes data to `fd[1]` and the `childBehavior` subsequently reads this data from `fd[0]`. Hence, the data crosses memory boundaries. Pipes are a communication link for message passing which avoid the critical region problem.

2.4.3 Network Approach

As we have outlined, distribution is another approach to the conception of concurrent execution within a system. Aside from the lack of shared memory, the distinguishing characteristic between parallel and distributed computing is the geographical distance between tasks. Therefore, the communication between distributed tasks happens via networked message passing mechanisms. Networks introduce a wide range of perils. We can neither assume that the communication links are reliable nor static. Also, messages are more costly in terms of time (latency) and effort (e.g. due to data serialization) [5]. The famous *Fallacies of Distributed Computing* by Deutsch subsume many of the problematic aspects [134]:

Fallacy 1: *the network is reliable.*
Fallacy 2: *latency is zero.*
Fallacy 3: *bandwidth is infinite.*
Fallacy 4: *the network is secure.*
Fallacy 5: *topology doesn't change.*
Fallacy 6: *there is one administrator.*
Fallacy 7: *transport cost is zero.*
Fallacy 8: *the network is homogeneous*[5].

Fallacies 4 and 6 are outside the scope of this thesis. The remaining aspects are relevant to some concepts we have already discussed or will soon discuss. For example, Fallacy 2 affects synchronous communication. Asynchronous messaging does not concern the latencies which delay the travel time of messages. Time constrains synchronous communication and therefore the network-induced latencies also affect synchronization.

All concurrency through distribution is subject to these fallacies in general. Network programming interfaces depend on the concrete network mechanism. A very basic example is the concept of *sockets* that Unix introduced. The standard interface of sockets provides the means to write data to the network. Alternatively, we use a socket to receive data from the network [14,134]. Practically every operating system provides

[5]Fallacy 8 was not part of Deutsch's original proposal, but later added by Gosling. Hence, the literature sometimes refers to merely seven fallacies.

sockets and bindings exist for almost all programming languages. Therefore, sockets are a technology-heterogeneous mechanism, although rather low-level (transport layer). More high-level is for example HTTP (**H**yper**t**ext **T**ransfer **P**rotocol), a generic, originally stateless and text-based communication protocol that provides platform-neutral data transfer on the application-level [30].

3 Actor Model

Some problems are better evaded than solved.

— C.A.R. (Tony) Hoare

In this thesis, we particularly focus on one of the traditional models of concurrent computation. In the 1970s, Hewitt *et al.* [59] formulated the *actor model of computation*. As the name suggests, this model builds upon the concept of *actors* as basic building blocks. In this chapter, we describe the model's properties, its unified abstraction, different implementations of the actor model, and its combination with other models of concurrent computation.

Agha [2] describes actors as self-contained, interactive and independent components that communicate via asynchronous message passing. He also reformulated the actor model into three basic primitives an actor can perform upon receiving a message [4]:

1. Send a finite number of messages to itself and other actors.
2. Create a finite number of new actors.
3. Substitute its current behavior with a *replacement behavior*.

In order to send a message, we must know the unique *address* of an actor. The underlying *actor system* delivers every message. In general, the order of delivery is nondeterministic. We can announce an actor's addresses by sending the address as a message. This method of propagating location information provides the ability of *dynamic reconfiguration* [2].

An actor processes one message at a time. Every message gets buffered in the so-called *mailbox*, if an actor is not able to process an incoming message immediately, because it is already engaged in a message handling operation. Access to the mailbox is race-free, and therefore safe [54].

3.1 Message Passing and Encapsulation

The actor concept defines that the only possible form of communication between actors is the exchange of messages. This restriction implies that there is no directly shared state between actors. An actor *encapsulates* its entire state exclusively. To access an actor's state, we must send a message that requests the state information. Actors process messages in a serialized fashion. This provides the basis for *isolation* [90]. All state modifications an actor does while it processes a single message appear to be *atomic*. New messages do not interrupt an actor that currently processes a message [5], because the mailbox buffers all messages.

It is important to realize that the state information we request from an actor is a mere snapshot of the actor's state at a specific point in time (the point when this actor processed the respective message). When we receive a snapshot answer, we must be aware that this information is already outdated in general [12]. On the other hand, the isolation of state frees actors from the implications of shared state and resource handling, like the bottlenecks that sequential locking introduces [2]. We must either copy, proxy, or make the messages otherwise immutable in order to ensure that the snapshots do not violate the encapsulation semantics and prevent that we accidentally expose a direct access to internal state or resources [87]. This immutability guarantee enables save coordination at a distance [58].

Conceptually, we realize the internal state changes within an actor through the third of the basic primitives: behavior replacement. In general, this primitive changes the behavior of the actor entirely. The actor taxonomy also calls this to *become* different operations for all following messages. However, the behavior can also become the same operations, but for a different state [140]. It is important though that behavior replacement does not break the *referential transparency* of the actor's address [2]. Therefore, changing actor internals has no effect on its reachability for other actors. The actor logically stays the same, but behaves differently for the next message. This strict encapsulation of state and decoupling via immutable and asynchronous message passing leads to a strong form of *isolation* between actors [3]. State within an actor is mutable, but isolated from the outside and only available through immutable snapshots.

Additionally, an actor only changes state while it processes a message. Therefore, as De Koster *et al.* [87] illustrate in detail, we can view the processing of a single message as an isolated operation. This is important when we reason about actors, because we always have to reason with respect to the single-threaded semantics that provides the granularity of a turn[6] [20]. We call this the *isolated turn principle*. This principle guarantees the safety of actors, because they are free of low-level data races. However, high-level races (depending on the interleaving of messages within the mailbox) can still occur. The isolated turn principle also guarantees computational progress with each turn [9], and thus liveness.

[6]In this context, *turn* refers to the processing of a single message. The literature defines various terminologies. A good overview of actor model taxonomy and the equivalence of various terms gives [87].

3.2 Unified Abstraction

Until now, we have described the actor model as a general model of computation. The abstraction of actors provides a strict separation of component states, as well as a loose coupling via asynchronous message passing. Actors encapsulate not only data and functionality, but also their own *thread of control*, making actors autonomous [5]. This autonomy enables the concurrent execution of actors, effectively turning the actor abstraction into a model of *inherent concurrent computation* [8].

There are numerous models which are able to provide inherent concurrency, e.g. logic programming or functional programming. The benefit of the actor concept however is its support for the direct expression of state [8]. This state is only mutable within an actor and while the actor processes a message. Each turn is atomic. Omitting to share state and only communicating information via asynchronous message passing greatly improves the safety of actors, as it eliminates a whole class of errors, the *race conditions* [27].

The dynamic data flow of messages is the primary source of nondeterminism. We get no guarantee on the order of messages when various actors send messages to the same actor. Yet the actual order of processing the messages affects the behavior, resulting in nondeterminism. A not enforced message order however eliminates unnecessary synchronization overhead [2,5,85].

The actor model provides a strict concept of isolated and decoupled components. The only link between actors is the delivery of messages, based on their addresses. These addresses are virtual since they do not expose physical location information [20,135]. Addresses therefore do not restrict actors to physical locations. As a result, we do not require the concurrent units to be inside the same process boundaries, nor the same host machine. The addresses bridge the gap in physical distance. *Location transparency* is the general term for separating the virtual and physical location [5,85]. The concurrent components of the actor model inherently support parallel component execution, since they can be transparently assigned to processor cores.

Additionally, location transparent addresses enable us to reference actors even outside the scope of a CPU. We get the foundation for distributed execution on different nodes [85]. We refer to the execution on the same CPU as *intra-node* parallelism, and to the distributed execution as *inter-node* parallelism [117]. Intra-node components are still physically close, and we can assume that their communication channel is reliable. Inter-node components have no guarantee on the safety and reliability of their communication channel. The messages must travel via network links (cf. Fallacy 1: *the network is reliable*). The only valid assumption is that communication is more costly and volatile in any case [5]. A particular characteristic of the actor model is therefore the facilitation of one and the same primitive for task unit communication in concurrent, parallel and distributed execution contexts.

3.3 Actor Systems and Variations

Actors are autonomous computational entities, but not individually deployable on operating systems in general. They require a runtime environment, the so-called *actor system*, to exist within. Actor systems have two general concerns: they provide the linguistic support to utilize the actors (programming interface and model semantics) and they realize efficient actor execution [85].

Depending on the underlying programming model, the actor concept and primitives can pose a challenge for programmers. The model primitives provide a very low-level abstraction to express computation in an (almost) pure communication paradigm. Therefore, actor systems aim for additional, more high-level primitives [2,140], e.g. to express various other communication patterns.

Efficiency can pose a challenge, since the runtime must use the idioms of the underlying system or platform to map concurrency to the actor abstraction. In thread-based environments like Java's virtual machine, we are forced to execute the relatively lightweight actor constructs on top of the relatively heavy JVM threads. One implementation for JVM threads is the direct mapping of a thread to an OS process. In this case, each actor is executed as a system process. Haller & Odersky [54] call this an *impedance mismatch* between message passing (event-based) and thread-based programming. In this concrete example, a runtime can mitigate the negative impact by not assigning one dedicated thread per actor. Instead, the runtime can employ a scheduling strategy similar to operating systems. With scheduling, many actors share the resources that fewer threads provide [85].

Numerous actor system implementations do exist. All diverge in term of features and model semantics realization. We have identified three that merit special attention:

Erlang

A programming language dedicated to actor-based programming is *Erlang* [13,137]. It is well-known for introducing the programming with actors to a broader industrial application. Ericsson first used Erlang in 1986 to build telecommunication infrastructure. In contrast to most programming languages, an Erlang program has the main focus on its so-called *process* constructs (actors), rather than e.g. objects or functions. Erlang is designed to meet challenges like high concurrency, availability, fault-tolerance and live-upgrades [87,137].

Akka

Released in 2009, *Akka* [64] is the most important actor framework for the JVM today. It offers bindings for the Java and Scala programming languages. Akka is highly inspired by Erlang and facilitates many of the same conceptualities in order to meet similar concerns. Examples are scalability, fault tolerance and remoting/distribution. As a library, Akka faces conceptual difficulties which endanger the actor model semantics. Ecosystems dedicated to the actor model typically avoid these dangers, as does Erlang with its virtual machine, the *BEAM*. Section 5.2 concerns Akka and the challenges that the JVM presents as the target platform in more detail.

Orleans

A recent variant of an object-oriented interpretation of actors called *active objects* is *Orleans* [20]. Microsoft Research constructed Orleans in 2011 to meet the requirements of highly distributed deployment setups, currently referred to as *cloud computing*, on the .NET platform. Orleans facilitates what it calls the *virtual actor model*. A virtual actor (called a *grain*) does not exist *physically* as an entity all the time. The actor runtime only (re-)instantiates a grain on demand automatically. In contrast to most other actor variants including Erlang and Akka, this omits the need for lifecycle management. The *virtuality* characteristic turned out to be more suitable in high-scale dynamic workloads of today's cloud computing deployment setups.

To our knowledge, Erlang, Akka and Orleans have the most significance in industrial applications. In the remainder of this thesis, we refer to individual characteristics of all three actor variants to point out relevant differences and noteworthy capabilities.

3.4 Active Objects

As we pointed out, actor systems often aim to provide a more high-level interface than the mere basic primitives to express concurrency. One specific way to realize a higher level is through the concept of *objects* we know from the *object-oriented programming* (OOP) paradigm. Objects encapsulate state and offer operations on this data [2,140]. In the terminology of the Smalltalk programming language, we invoke an operation by sending a so-called *message* to an object. This terminology already points out the conceptual resemblance between actors and objects in general [122].

Yonezawa *et al.* [144] were the first to introduce classic *passive* objects extended by their own *thread of control* in a programming language they call ABCL/1[7]. The state of an object in ABCL/1 is only accessible from within the object itself. We access or modify the state through the invocation of public interface methods of the object. However, the objects are not idle by default and only perform operations when we call their methods. ABCL/1 objects are *active* on their own, since they live in their own thread. Hence comes their name: *active objects* (AO). When an active object's method is invoked, the actual method execution is decoupled and performed concurrently. A *proxy object* realizes the method invocation on the client side (invoker). The proxy merely mirrors the AO's public interface and handles the message dispatch. The actual computation runs inside a server object on a separate thread [88]. Meyer [97] points out one general notion of the concurrency conception of active objects that emphasizes a viewpoint which we pay more attention to in due course:

[7] **A**ctor-**B**ased **C**oncurrent **L**anguage. The */1* indicates that it is merely the first of a whole family of languages. We have found it often omitted in the literature. Consecutive versions do not follow a sequential numbering, e.g. *ABCL/R*, *ABCL/f* or *ABCL/c+*.

> *"Each object is a single, identifiable process-like entity (not unlike a Unix process) with state and behavior."*

AOs aim for a purely object-oriented version of actors. Scholars have argued that actors themselves already represent the very essence of object-orientation [2]. There is a decade-old debated about the fundamental concepts of object-orientation. The author of this thesis came to the conclusion that the only truly undisputed characteristic seems to be the encapsulation of state [86] coupled to a set of operations which share this data [140]. Additionally, actors share object concepts like the ability to be created, having a unique identity (address) and a public interface [129]. As a result, it has often been argued that either actors are convenient for the foundation of active objects, or that AOs are suitable to implement actors [88].

It is worth to point out that due to this similarity of the actor construct with the object essence, scholars do not use the terminologies consistently. We have found that the literature often uses *active object* interchangeably with the *actor* term. In this thesis, we use *actor* to refer to Hewitt's concept that Agha later refined. Subsequently, we use *active object* for the object pattern of Yonezawa to abstract the actor semantics into a classic object API.

The following example illustrates the subtle difference in the behavior of classic versus active objects:

```
class Fnord {
    private int a = 1;
    void add(int b) {
        a += b;
    }
    int get() {
        return a;
    }
}
```

Using the above class `Fnord`, we create the following simple procedure:

```
1   final Fnord f = ...; // obtain reference to an instance
2   f.add(1);
3   print(f.get());
```

In a single-threaded program, once the execution reaches line 3, we can safely assume that the addition finished and the internal field `a` has the new value. The `get()` call subsequently results in `2`.

Now we alter the definition to `class Fnord extends ActiveObject` with some arbitrary base class `ActiveObject` that turns `Fnord` into an active object implementation.

Then, the previous observation does not hold anymore. When the execution reaches line 3, we have no guarantee that the addition was already executed. Line 2 only dispatched the message and returned to leave the activity to its own flow of execution. Line 3 then blocks (because `get()` has a return value) and waits until we receive an answer. But we cannot assume that we receive the value 2 anymore. The active object can receive other messages and process them between our `add(1)` and `get()` messages. The nondeterminism we introduce through the concurrent behavior hinders us to safely reason about our result value.

We see, although active and passive objects offer the same interface, they do not provide the same degree of safety. The author of this thesis believes that this safety mismatch is dangerous for programmers in general. The classic actor variants regarded by the author (Erlang, Akka) do not provide interfaces that we can mistake for non-concurrent entities. Therefore, these actors do not offer us a false sense of safety. However, there is one major benefit of active objects compared to classic actors. The object interface provides *type safety*. Messages to AOs are strongly typed [88]. In contrast, we can send arbitrary types of messages to ordinary actors. Only when an actor processes a message at runtime, it decides whether the behavior is actually able to understand the message type. Actors therefore perform *dynamic type checking* at runtime – even in statically typed programming languages.

On the other hand, active objects provide ordinary object-like interfaces. We send a message to an AO when we call a method of the object with a fixed signature (the proxy respectively). We are only able to call the methods of the object's public interface. A compiler statically ensures the message validity at compile time. Due to the nature of AO interfaces, they only provide message passing in a point-to-point communication style. Broadcasting messages hypothetically requires one method call to address several objects. This behavior is not intended by the object abstraction [144].

The active object method signatures do not only define communication with a certain degree of static type safety. Every signature also influences the behavior of an object's thread of control. Therefore, signatures constrain synchronization [88]. In the original work of Yonezawa *et al.*, they introduce multiple types of message passing for method invocation [87,144]:

Past Type
The message is dispatched and the sending object's thread of control immediately continues. The thread does not wait until the message has been processed by the receiver. This behavior is equal to message passing in the classic actor model.

Now Type
The message is dispatched and the sender waits for a result. Its thread of control blocks until the receiver processed the message and replies with the result. This behavior is equal to a method call (with a return value) on passive objects.

Future Type

The sender's thread of control immediately returns with a reference to the result that will be available at some point in the future. The actual result becomes available once the receiving object has processed the message and replies with a result.

The example above illustrates two of these behaviors for method invocations on active objects. `void add(int)` only dispatches a message and immediately returns (past type). In contrast, `int get(void)` actually waits for a result (now type). Using the future type requires us to include an additional model of concurrency, the *future*.

3.5 Integration of other Concurrency Abstractions

The actor model is a mature, general purpose model for expressing concurrent computation. It has some clear principles which we must uphold in order to ensure the intended semantics. Besides these principles, the model does not make additional assumptions and restrictions. This makes actors flexible and applicable for general purpose computation. Concurrency, or the suitability for it, is basically an inherent side-effect. As a result, we can combine the model with additional, arbitrary approaches to express computation. Even concurrent models are possible, as long as every concept we introduce does not jeopardize the actor semantics.

As a result, mixing actors with other forms of concurrency was always common. The reasons for introducing additional abstractions are manifold. Tasharofi *et al.* [135] empirically found that programmers think that the major inadequacies of actor systems are their lack of efficient support for blocking operations (especially IO). Also, many communication protocols are hard to express in an asynchronous messaging style.

In order to overcome these shortcomings, actor systems interweave with additional concurrency models. We come back and take a closer look at the two deficiencies – efficient IO and communication styles – in Section 5.2. Before, we must know the requirements for two concurrency models to be *composable* without inconsistencies. Swalens *et al.* [133] regard two concurrency models as composable if their integration does not result in new effects regarding their *safety* and *liveness* that have not been there before. The isolated turn principle of the actor model already gives a strong boundary to ensure these properties [87]. Added concurrency concepts must neither weaken these boundaries nor the model properties. Especially, the introduction of low-level data races is very easy for new abstractions and we must therefore carefully avoid any race conditions [135].

3.5.1 Futures

The traditional notion of a procedure call is that the execution flow only continues once the invoked computation has finished. Of course, the procedure we call can dispatch a concurrent execution and return without a result. The flow of execution then resumes before the computation we called executes. However, if the procedure provides a return value, we expect the control flow to *block* until the respective value is available [134].

In many cases however, we do not immediately require the result for the subsequent computation. The control flow is able to continue without accessing the value for some time. Therefore, it is possible to resume the caller's activity, while the called procedure executes concurrently in a separate thread of control. The procedure initially returns a simple placeholder that will contain the actual result value at some point in the future [38,141]. We use such values in a *semi-synchronous* fashion. The value calculation runs *asynchronously* in general. The calling and the called thread once again *synchronize* when we access the placeholder for the actual result. We also call this *touching* or *claiming* the value. Attempting to access a placeholder expands to blocking the current control flow if the result is not yet available [135,141].

The literature does not use a uniform name for the concept of eventually retrievable values. Baker & Hewitt [16] describe the concept of a *future* which delivers the result of an expression eventually. Liskov & Shrira [92] extend this idea and introduce a data type they call *promise* for result values that we single-assign at some point in the future. More seldomly have we found the terms *eventual*, *delay* or *deferred* [114]. *Call-by-future* [16] or *call-by-need* [4] express the kind of evaluation order of these concepts.

Some programming languages, among them Java and Scala, have a special view on eventual values. These languages support both `Future`s as well as `Promise`s. A `Future` represents a read-only container we use as the placeholder for an eventually available value. In contrast, a `Promise` is a single-assignment variable we use to explicitly set a value at *some* point in time, i.e. to complete a `Future`[8]. In other words, a `Future` refers to a `Promise` that we keep eventually [55,114]. Though today we find all designations interchangeably used and they refer to roughly the same idea [133], we confine to the term *future*. Java and Scala use this name and we rely on the specific future semantics of these two languages in due course.

There is a long tradition of combining actors with futures. Agha [4] describes that actors often model call-by-need computation, which is essentially a future. We also trace future combination back to ABCL/1 and its active object notion [144]. We have already discussed three kinds of message passing for AOs. The example then merely demonstrated two (past type and now type). The third, coincidentally called *future type*, is actually the result of combining actor concurrency semantics with the future concurrency abstraction [135]. Orleans uses promises/futures as the only form of method calls for all active objects [20]. For a complete formal definition of future semantics we refer the interested reader to Flanagan & Felleisen [38].

3.5.2 Software Transactional Memory

The asynchronous messaging style of actors becomes a burden when we need some sort of consensus between several actors. The model does not provide an adequate mechanism

[8]Therefore Java calls it `CompletableFuture`, instead of `Promise` as Scala does.

to abstract operations involving several messages [132]. We need an additional high-level model on top of the messages.

The *transaction* is a well-known concept to provide a single-threaded appearance to the access of state or memory that is concurrently accessible. A transaction encapsulates a computation that does not have to be atomic by itself (e.g. code block). The effects of the computation still logically appear to happen within a single instant in time [90]. Therefore, all memory modifications done inside the transaction become atomic from the outside perspective. If a transaction becomes invalid, the transaction roles back all state modifications across the entire code segments involved in the transaction. Write collisions are one reason for transactions to become invalid. Upon a collision, the transaction cannot guarantee the isolation anymore [132]. *Software transactional memory* (STM) refers to transactional semantics realized in software[9]. In the scope of this thesis, STM is the only considered transaction mechanism.

Combining transactions with actors can have a huge impact on performance. Especially write collisions raise the amount of coordination we require. Though all required coordination can happen transparently through an actor system, it always means additional message processing for the involved actors, which potentially turns into a bottleneck [132].

[9]Originally, the concept was proposed for supporting transactions in functional languages, but in hardware. Hence the distinction.

4 Microservice Paradigm

These are my principles. If you don't like them, I have others.
— Groucho Marx

Microservice architecture (MSA) is a recent software architecture style for designing software applications as systems of loosely coupled, independently deployable, single purpose services [42]. In this chapter, we outline the motivation behind microservice architectures, the component properties and the resulting concurrent nature, problems arising from the terminology, and the service programming model.

In contrast to a monolithic application, for a microservice architecture we split the application logic into a suite of small parts. We implement each part as a dedicated program and design it to provide a single task of the business logic. We call these programs, which pose as the application's components, *microservices* (MS). The microservice style is open for every programming language and paradigm. All services communicate with message passing semantics on the OS or network level. Therefore, we can conceive the various services in different programming languages and technologies. As long as every microservice exposes the interface that the architecture requires, the service is a suitable component [34,105].

Unlike the actor model, microservices were not invented as a specific model of computation. Instead, the microservice paradigm emerged from the industrial need to break the scalability barrier that monolithic[10] applications inevitably reach [23]. Only later did the scientific community gain interest in this paradigm. The consequence was an explosion of contributions on this concept in recent years. Yet, academia still has some troubles to settle on a common basic definition of the essential concepts which determine this paradigm [57]. Fowler & Lewis [42] give the seminal review of the microservice concept.

[10]The term originates from the *monolithic kernel* style of operating system architectures. Such is a sole binary running in kernel mode and providing the process- and memory management, file system, etc.

Their work is therefore the original source most scholars refer to. But Fowler & Lewis focus on the common characteristics of microservices from a more practical engineering perspective. We refer to Dragoni *et al.* [34] for further reading as well, since they give an extensive conceptual description. An overview of the publication trends is given in [43].

4.1 Limits of Centralization

Historically, software systems are implemented in a so-called *monolithic* way. All the source code compiles into one single, central artifact that we execute on one machine. This centralization originates from the level of abstraction mainstream programming languages provide to break down the complexity of the programming task. The general term for these abstractions is the *module*, and they allow us to logically separate concerns. Yet, the transformation of modules from program code to machine code leads to a result where all modules merge into one unified construct: the (monolithic) *executable* [122]. By inversion of argument, a monolith is an application of modules that we cannot execute independently [34].

Modules naturally introduce a relatively strong form of coupling. In-memory call communication is a cheap and direct way to address components and we therefore apply it heavily within monoliths and their modules [23]. Every application that is subject to the tight coupling of its modules suffers from certain issues [33,34,121]:

- The components are less reusable.
- Increased interleaving becomes hard to maintain.
- We need to upgrade all modules simultaneously and are limited in the technologies we can use.
- Evolution is generally hindered due to the growing complexity.

Most importantly, the main argument against monoliths is scalability [34]. Each single instantiation of a program executable is intrinsically only able to run on a single machine. Hence, there is a natural upper bound to the application's performance due to the hardware limitations. Many approaches to overcome this limit(s) were proposed over the years. Previously, the *service-oriented architecture* (SOA) approach gained popularity. SOA builds on the idea of combining the capabilities of multiple monoliths – either of the same or different program images – and integrating them through a uniform communication channel like an *enterprise-service bus* [42,121]. This approach allows us to link heterogenous technologies and enables independent deployment, since we do not require in-memory calls between these services anymore. However, SOA still facilitates large monolithic applications which are only able to scale through duplicated instances of the entire application [35].

In order to overcome these limitations, the microservice paradigm aims for a separation of the modules into small service programs. SOAs are called the *first generation* services, while microservices are subsequently the *second generation* of services. Because microservices evolve from SOA, they are also part of the general *service-oriented computing* (SOC) paradigm [34,96]. For an in depth review on how microservices historically emerged from a distributed systems perspective, starting with the client-server paradigm, to mobile agents technology and service-oriented architectures, we refer the interested reader to Salah *et al.* [121].

4.2 Term Ambiguity

The literature uses the *service* term in an overloaded manner. We identify two general meanings attributed to the term. On the one hand, the term refers to a computational task unit, i.e. a process, as part of a service-oriented architecture. This is the predominant intention when authors refer to a *microservice*. On the other hand, a service also describes a specific functionality that we can utilize through an interface. We know this notion from object-orientation, where objects provide services in the form of procedures through their public interface [129].

In order to avoid confusion, some scholars propose a clear distinction. For example, Xu *et al.* [143] use the term *service* exclusively for the functionality part and refer to the component as *agent*. We follow the example of Guidi & Montesi [51] and distinguish between the *service engine*, that is the component we deploy as a process, and the *service behavior* for the functionality that the service engine offers. This terminology suits us, because it highlights the resemblance between an actor's behavior and a microservice's behavior.

4.3 Independence and Interaction

From the separation of modules into dedicated service engines comes a high cohesion within the modules as well as a loose coupling among them. As a result, microservices are highly independent [34,35,49]. The inherent fact that each service engine is a separate application implies that the engines are independently deployable [42]. The decoupling of services also affects their state, since the state becomes conceptually isolated. Therefore, we require that services refrain from sharing any resources related to memory persistence. A database for example introduces a notion of implicit communication among all the services with access to this database. An essential principle of the microservice style is therefore the commitment to provide every service engine with its own exclusive database instance(s) [34,42,107].

Consequently, all communication between microservices happens across the boundaries of the service engine processes. We already know this concept from Section 2.4.2 as interprocess communication. Various forms of IPC channels exist. Example mechanisms are a shared memory section between two processes within the same operating system, or Unix pipes. The microservice idiom specifies the following requirements on service engine IPCs [42]:

1. Communication channels should be open and provide well-defined interfaces, such that heterogenous technologies are able to use them.
2. Communication channels should be lightweight, such that they are cheap mechanisms without much additional functionality besides basic message transportation.
3. Communication channels should only act as message routers, such that they transport immutable messages and do not apply data processing logic on their own.

From the two example IPC mechanisms above, Unix pipes and shared memory, only the pipes qualify for a valid microservice communication mechanism. Data in the form of text strings represents serialized state information of a service. The pipe transports this data in an immutable way between the endpoints of the pipe. Raymond describes this in his Unix *rule of composition* as [116]:

> *"Text streams are to Unix tools as messages are to objects in an object-oriented setting. The simplicity of the text-stream interface enforces the encapsulation of the tools."*

This satisfies the channel requirements given above. Therefore, pipes are a valid communication mechanism. However, pipes do not offer a specific structuring of the byte stream. Programmers have to specify an application-level protocol on top of the pipe mechanism [14]. Shared memory, on the other hand, faces several conceptional problems regarding a microservice communication mechanism. Services would send messages by modifying memory both services have access to. Yet the memory is not necessarily exclusive to both services. A third party that also has access to the memory can intercept a message by getting a lock to the shared state before the intended recipient. This third party is subsequently able to modify the message. Shared memory does not guarantee the delivery of the original message itself. We require that the synchronization mechanism enforces the semantically correct order of state access. This risk to correct message delivery is another argument why microservices do not use shared state communication.

4.4 Concurrent and Distributed Building Blocks

As independently deployable applications, each microservice is by design a dedicated process. These are inherently concurrent on the operating system level and also facilitate parallelization on multiple cores transparently. Every communication mechanism must be able to send data across the distinct address spaces which strictly isolate the states. Recall the case study of Section 2.4.2 on system process programming in C. Fork/pipe-based applications utilize Unix pipes to cross memory boundaries. Like microservices, their components execute concurrently through the process scheduling of the OS. It is therefore worth to debate whether these systems qualify as microservice architectures.

The fork pattern spawns processes in a tree-like fashion. All components rely on a shared ancestor. Every child can replace itself by an arbitrary other program image using `exec`.

However, the setup of the communication routes relies on an instantiation of the pipe *before* the fork. Only then have both the parent and the child access to the pipe. This fact limits the possibility to take down or add new components independently. Hence, pipes are rather restrictive and present a certain degree of coupling [14]. It is difficult to replace the system's components independently.

To overcome this restriction, more modern Unix variants introduce the concept of *named pipes*. This kind of IPC allows processes to hook into a common pipe without a shared ancestor [14]. However, every communication route we fix at compile time – generally called *static channel naming* – limits the ability for changing topologies [12]. In any case, the pipe mechanism definitely does not provide communication outside the boundary of the common operating system. Pipes therefore limit scalability since they restrict the service architecture onto a single host machine – at least for the subset of services that facilitates this mechanism. Network-based IPC mechanisms overcome this restriction inherently and we subsequently prefer them for MSAs in general. Network IPCs provide higher degrees of freedom regarding deployment and heterogenous technology integration at the price of more costly data transfers (cf. Fallacy 7: *transport cost is zero*) [30]. As a result, we always assume that microservices are concurrent components which support parallel execution. However, whether they qualify as distributed components within their respective architecture is subject to the communication mechanism(s).

4.5 Size, Scope and Granularity

Bonér [23] criticizes the term *micro* since it encourages us to debate the actual *size* of a service. Every size definition requires a metric for comparison. Only with a metric can we debate up to which limit we call an application a *micro*service. Therefore, developers focus on metrics like *lines of code* up to more obscure ones, e.g. the reported *two pizza team* size, where a service is written and maintained by a team that we can feed with two pizzas[11] [42]. These discussions are irrelevant.

Instead, Bonér argues, a notion of size should refer to the *scope of responsibility*. A guideline towards this is once again the Unix philosophy [116] we already referred to. The philosophy suggests that programs should have a well-defined and limited scope. We realize more complex functionality by composing multiple of these simple programs. This concept is also found in object-oriented design, where we know it as the *single responsibility principle* of objects [49]. In the microservice context however, authors do not refer to the single responsibility principle a lot. Instead, they tend to phrase it *bounded context* [33,34,96]. Authors argue that services should be an aggregation of functionality around a single business capability. This divergence of granularity is one of the major evolutionary changes from SOA to microservices [49,105].

However, a too-fine granularity becomes an issue. In a distributed setting, granularity is always a balance between the number of messages that we send versus the perfor-

[11]Assuming an arbitrary pizza size, this either suggests very small teams, or really big pizzas.

mance implications we expect by every message. We must consider latencies of network channels, processing time of addressed services, and delay penalties that result from service unavailability when we design the granularity of microservices [123]. In general, we expect Fallacies 1-3, 5 and 7 to contribute to bounded context considerations.

4.6 Service-oriented Programming

Until now, we have regarded microservices in the light of a software architectural style. Within this context, the building blocks are all truly isolated components we instantiate from independently executable artifacts. We merely link the artifacts through rather loosely coupled message passing communication channels. Scholars argue that due to these characteristics the perspective in the literature (both academic as well as industrial) has a focus on the *deployment*, which is the operation of a service engine [49,143]. After all, the deployment context of MSA is the origin of the concurrent nature, that is independent processes of the OS or network. In this section however, we favor an argumentation towards a linguistic viewpoint on microservices as a programming model instead.

Programming paradigms build upon respective conceptual constructs, e.g. objects in OOP or functions in *functional programming*. In more recent time, a new paradigm called *service-oriented programming* (SOP) [51,104] emerged. It builds upon the service-oriented computing approach that SOA and subsequently microservices emphasize and introduces it into the design of programming languages. Services become first-class entities of the language and are the smallest programmable constructs. Instead of a single executable artifact, service-oriented programs compile into multiple executables, one for each of the service constructs in the source code. Initially, the conception of SOP aimed for an evolutionary step. The idea was to combine the object-oriented notion with the SOA paradigm to program distributed systems. When the microservices principles finally distilled, it transpired that the compilation of SOP languages essentially produces microservice architectures. Instead of the total technology freedom of the mainstream MS paradigm, SOP languages are a separate and more restricted approach towards the microservice style [49,104]. Various prototype languages facilitate the service-oriented programming model. Two SOP languages merit attention:

CAOPLE

One attempt of a service-oriented programming language is *CAOPLE* [143]. It calls its basic programming constructs *agents* (microservices). Agents provide a very strict notion of state encapsulation, autonomy, and well-defined communication interfaces. Unlike traditional microservices, agents do not execute directly on the host's OS, but run on a dedicated virtual machine called *CAVM-2* instead. CAOPLE's VM focuses on providing a lightweight dynamic deployment of services compared to container technology. Additionally, the VM is optimized for running large quantities of services in a lightweight manner, as well as abstracting the network distribution of agents across host machines.

Jolie

Currently the most advanced and scientifically best described service-oriented language is *Jolie* [49,51,100,103,104]. We therefore use Jolie as the primary linguistic reference throughout the remainder of this thesis. A Jolie program defines services which describe two basic aspects. First, the *behavior* expresses all functionality that the service offers. The behavior makes this functionality available on the so-called *communication ports*. Second, the *deployment* describes how a service is used, i.e. the communication technology, addresses of the exposed functionality, and data protocols. These two main parts of every Jolie program, behavior and deployment, indicate the strong *separation of concerns* between what we designate service behavior and service engine. Even the root level syntax expresses this separation in the program structure:

$$Program ::= D \ \mathsf{main} \ \{ \ B \ \}$$

The language's syntactic rules prevent us from introducing the deployment expression (D) into a behavioral expression (B) and vice versa. The only connection between behavior and deployment are the communication ports. The behavior abstractly uses the communication ports, and the deployment concretely defines the ports. Hence, behavior and deployment are complemental. As a scientific prototype language, Jolie incorporates many interesting concepts. There is only an implicit notion of concurrency from the service execution as OS processes, and the `concurrent` primitive as one option for the so-called *execution modality*. In this case, upon receiving a message on a communication port, a Jolie service spawns a dedicated process such that the behavior executes in a separate local memory space (cf. the fork approach in the C processes case study of Section 2.4.2). The primitive hereby allows concurrent message processing. Among other concepts, Jolie supports complex message routing through *correlation sets*, and facilitates transparent delegation through the so-called *aggregation* primitive, which extends a service's interface with the interfaces of other services.

Microservices share many similarities with objects in general [34]. Therefore, service-oriented languages tend to have many analogies to object-oriented languages, as the computational units in both paradigms provide functionality via a public interface [129,143]. However, this marks also the most important difference to objects. In general, objects facilitate information hiding and encapsulation in a *shared memory* setup. Microservices on the other hand solely rely on *message passing* [34]. Besides this difference, advanced object-oriented concepts can also be part of a language's service constructs. For example, Jolie has static type checking for service interfaces [100], and CAOPLE even supports polymorphism via an inheritance mechanism.

5 Implementation

Every good work of software starts by scratching a developer's personal itch.
— Eric S. Raymond

In this chapter, we cover the practical aspects of programming with actors and microservices. Section 5.1 describes a scenario for a concurrent system. Section 5.2 covers the strategies we apply to implement this scenario using the actor model. Subsequently, Section 5.3 describes the implementation of the scenario using the microservice model.

5.1 Concurrent System Scenario

In this section, we outline a domain-specific search engine we call *Echo*[12]. This search engine is our non-trivial scenario of a concurrent system that serves us as the reference for evaluating the programming of concurrent systems with actors and microservices.

Search engines are a composition of rather loosely coupled and independent subsystems [30]. Users interact with a search engine by submitting search requests in the form of so-called *queries*. The search engine then presents respective results to the user. This functionality however is merely the so-called retrieval phase performed by the *retrieval subsystem*. As the name indicates, this phase retrieves information. By implication, the information must have been collected and stored beforehand. A second so-called *indexing subsystem* is responsible for gathering the information and storing it in a form that is optimized for searching, the so-called *reverse index* [91,113]. The reverse index maps a *document-term* relationship – where documents are arbitrary text collections – into a *term-document* structure [95].

[12]We chose the name "Echo" for its wonderful characteristics of providing a short package name and the analogy to *recalling spoken words*.

Several factors contribute to the fact that search engine architectures are suitable for concurrent programming research. First, both subsystems are mostly independent. They merely make use of a common information index, where the indexing subsystem is exclusively adding information and the retrieval subsystem is exclusively reading the information. Hence, the subsystems are independent and can run concurrently. Second, since many kinds of search engines regard very large amounts of data, their construction was always led by the effort to leverage concurrency in order to improve their scalability. Especially the parallel and distributed computing research merits attention to search engines, for example to explore cluster architectures [95,113]. Additionally, our specific domain we outline below is also very suitable for concurrent processing.

The design of search engine architectures is generally led by two basic requirements [95]:

Effectiveness
The quality of search results is the *effectiveness* of a search engine. Effectiveness is the sole concern of the scientific discipline called *information retrieval* (IR). *Precision* and *recall* are the two metrics that IR defines in order to assess the effectiveness.

Efficiency
Factors like the response time and the throughput determine the *efficiency* of a search engine. These factors are highly affected by the concurrent processing capabilities of the system.

The optimization of effectiveness is not within the scope of this thesis. We merely apply a basic scoring method of the utilized information retrieval library. Our sole goal is to increase the efficiency of the system by leveraging concurrent programming techniques.

5.1.1 Domain Description

We build our domain-specific search engine for the *podcast* domain. On the one hand, the term refers to content, that is an episodic series of digital media. The media is usually audio, more seldomly video. On the other hand, *podcast* can also refer to the distribution mechanism. The distribution builds upon XML (**Ex**tensible **M**arkup **L**anguage) web feeds. RSS 2.0 [142] (**R**ich **S**ite **S**ummary) and Atom [108] are the established syndication formats. Since RSS 2.0 has always been the more dominant format, we simply refer to *RSS feeds* from here on. Both formats gained popularity in the 2000s as an effective, decentralized mechanism to publish the updates to a website's content. Podcasts build upon the same principle. Yet they utilize an otherwise optional field for items of an RSS feed, the `<enclosure>` tag. This tag provides an URL (**U**niform **R**esource **L**ocator) to the media file. Subscribers of the feed download the file behind the URL and watch/listen to the media, usually through a specialized software application. The `<enclosure>` is therefore the main content of each item in a podcast RSS feed. Additionally, there are other fields within the feed. Some of these fields contain human readable information about the linked media file, so-called *metadata* [111]. Appendix A gives an example RSS feed structure with dummy metadata.

Our search engine is designed to regularly analyze RSS feeds of podcasts. The metadata allows us to add information for every media file to the search index. Although we do not analyze the media itself, we can still provide search functionality based on the metadata information. The domain is very suitable for concurrent processing, since the RSS feeds are decentralized. Every podcast content creator is publishing a separate feed. There is no interrelation between feeds. We can process each feed separately and therefore concurrently.

5.1.2 System Components

At the core, our basic architecture and the components are inspired by the work of Brin & Page [24] on large scale web search engine "anatomy", as well as more modern interpretations of associated design principles given in [30] and [113].

The two high-level subsystems we have given above, indexing and retrieval, are internally composed of several smaller components. We specify that each of these components has to be a concurrent task unit of the programming model, i.e. an actor or a microservice. The respective units are:

CatalogStore (C)
holds a catalog of all metadata information we gather about podcasts, their feeds and episodes. The Store persists this information in a relational database.

IndexStore (I)
holds the data structure we use for searching (reverse index). Registered information entities are called *documents*. Each document relates to one podcast or episode. The IndexStore documents are merely the part of the metadata we need to match search queries to matching results.

Web Crawler (W)
acquires the information that the search engine stores by downloading data from URLs. These URLs relate to feed files.

Parser (P)
transforms the XML data into internal representation formats. This extracted data is what we consider when running search queries and subsequently display in the Web application.

Searcher (S)
performs the search operations. This component applies some basic query pre-processing and delegates the retrieval of relevant documents to the IndexStore with its inverted index. The Searcher communicates the results from the IndexStore back to the Gateway.

Gateway (G)

provides the link between the Web UI and the system-internal capabilities. The Gateway exposes a REST interface to the outside and uses respective mechanisms to interact with other internal system components. The REST interface allows us to request cataloged information and perform searches.

Updater (U)

determines which feeds to re-download next in order to register new episodes, and update existing metadata.

The CatalogStore and the IndexStore are stateful, all other task units are stateless. The complete search engine architecture is the composition of all these components according to the interaction model shown in Figure 1.

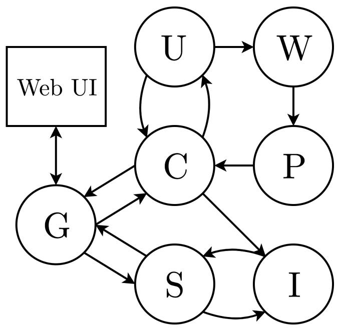

Figure 1. Complete interaction model of the task units in the Echo search engine

When we give some basic dataflow examples in due course, we use the following shorthand notation for arbitrary components `X` and `Y`, where `X` and `Y` get substituted by the component abbreviations (`C`, `I`, `W`, `P`, `S`, `G`, `U`). `X` $\rightarrow$ `Y` expresses `X` sending a message to `Y` (push). `X` $\leftarrow$ `Y` denotes `X` fetching a message from `Y` (pull). `X` $\rightleftarrows$ `Y` is short for `X` sending a request message to `Y` with a direct response (synchronous remote procedure call, RPC).

The system shown in Figure 1 merely forms the concurrent indexing and retrieval system. It is therefore a backend application only. In order to actually use the search engine, we provide the backend with a web-based user interface (UI). This *Web* application is based on the Angular [93] framework. The actor and microservice implementations of the backend have to provide a REST interface within the Gateway component to allow interaction from the outside. The Web UI serves us as the proof of concept for the desired functionality of the engine's backend implementations.

Since our scientific focus is on the concurrent programming aspect and not the information retrieval aspect, we want to implement the domain-specific logic only once. Therefore, we provide each backend with a common *Core* library written in Java. The Core offers most domain-specific functionality, so that each backend codebase can focus on the concurrent execution and interaction. For example, the actual searching is done through a specialized data structure, the reverse index. We use Lucene [39] to create this structure. Lucene offers a Java interface that is interoperable with most JVM-based programming languages. RSS/Atom feed parsing is done using ROME [47], enriched by an extension we wrote to support additional *Simple Chapter* [83] metadata information.

5.1.3 Processing Pipelines

In this section, we give a brief outline of the data processing pipelines which make up the indexing and the retrieval subsystem. The processing pipelines are the result of the composition of the architecture components.

Note that Figure 1 shows an interaction between the Gateway and the CatalogStore. The pipelines below do not mention this interaction. The Web UI can display the entire metadata of an item. Therefore we must retrieve the complete metadata from the CatalogStore. For search requests, the retrieval subsystem merely produces the reduced metadata that is stored in the search index. We nevertheless show the $\texttt{G} \rightleftarrows \texttt{C}$ call for completeness.

Indexing Pipeline

We process feeds either when they are new to us (initial indexing), or to check for new episodes (update). Hence, there are two cases when the indexing pipeline gets triggered. Either we add a new feed, or the Updater determines that a feed requires a check for new episodes. In order to determine which feeds require an update, the Updater regularly inquires the database of the CatalogStore. The Updater passes the update candidates to the Web Crawler. The Crawler retrieves the XML data of the feed via HTTP. Then the Crawler passes the raw feed data to the Parser. The Parser extracts the podcast and episodes metadata from the XML into domain objects. The Parser forwards all metadata objects to the CatalogStore. The database of the Store persists the complete metadata. The Catalog also sends the search-relevant part[13] of the metadata to the IndexStore, which adds the data to the Lucene reverse index data structure. The overall flow is: $\texttt{U} \to \texttt{C} \to \texttt{U} \to \texttt{W} \to \texttt{P} \to \texttt{C} \to \texttt{I}$

Retrieval Pipeline

The essential purpose of the engine is search. The Web UI offers an interface similar to well-known search providers on the world wide web. The Gateway registers search requests from the UI on the REST interface and forwards the request to a Searcher ($\texttt{G} \to \texttt{S}$). This Searcher is doing some basic query processing and then forwards the resulting query to an IndexStore ($\texttt{S} \to \texttt{I}$). The IndexStore propagates the search results back via the Searcher ($\texttt{I} \to \texttt{S}$) and the Gateway ($\texttt{S} \to \texttt{G}$) to the Web UI. We require this flow to complete in a timely manner, thus synchronous. The complete flow is: $\texttt{G} \rightleftarrows \texttt{S} \rightleftarrows \texttt{I}$.

[13]Some parts of the metadata, like the byte size or MIME type of the `<enclosure>` file, is important to determine new entries. Therefore, the CatalogStore persist this data. This metadata is however hardly relevant for search queries, therefore we do not include it in the search index.

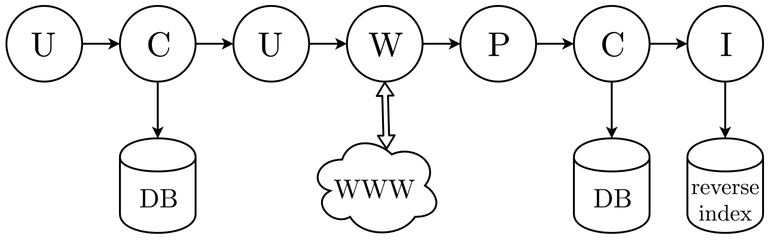

Figure 2. The indexing pipeline: The Updater (`U`) uses the CatalogStore's (`C`) metadata to determine feeds that require updating (`U` $\rightarrow$ `C` $\rightarrow$ `U`). The Web Crawler (`W`) loads the XML from the web, the Parser (`P`) transforms the feed data to domain objects. The CatalogStore persists the data and forwards selected metadata to the IndexStore (`I`)

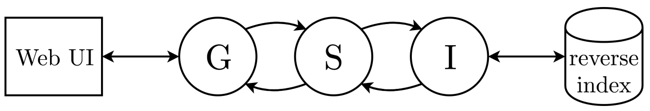

Figure 3. The retrieval pipeline: The Gateway (`G`) registers requests, forwards each query to the Searcher (`S`), who retrieves data from the IndexStore (`I`). The respective results travel back from `I` via `S` to `G`

5.2 Actor-based Implementation

This section covers the strategies we apply when we program with the actor model. We implement the backend of the concurrent system which we outlined in Section 5.1. All concepts we discuss are with respect to the specific actor variant we use. The focus is on the linguistic support provided by the actor library. Efficiency considerations are part of Chapter 6.

It is important to realize that although there is *the* conceptual actor model, there are numerous system implementations available through various forms of interfaces, either integrated into the programming language or as a library [87]. These systems are all based on the theoretical model, but can choose to compromise some of the semantic properties in order to increase their efficiency [85]. Such considerations are relevant when we evaluate the linguistic support.

We build our Echo implementation with an actor variant called *Akka* [64]. We have mentioned Akka already in Section 3.3 alongside Erlang and Orleans. The Akka library is available for the JVM through bindings for Java and Scala, but was later ported to other ecosystems such as .NET and JavaScript runtimes (through Scala.JS). The .NET variant (called *Akka.NET* [115]) is to our knowledge not able to interweave with the original JVM version at the moment. Because our solution is solely based on the JVM, all following discussions refer to the capabilities of Akka's original variant.

Akka theoretically builds on Agha's [4] vision of the actor model and harnesses its potential for distributed problem solving [87]. An archetype has been Erlang. Akka is designed as a toolkit collection consisting of several libraries. We can use these libraries in arbitrary combination based on our actual need of them. The actor runtime system is a lightweight execution environment based on work stealing thread-pools with local task queues which schedule the actor execution [52].

As of Scala version 2.10, Akka replaces the default actor implementation that Scala originally offered [52]. We therefore refer to the former as *Scala actors* in contrast to *Akka actors*. Among the reasons were the better performance, transparent actor addresses, expressing resilience as well as fault tolerance [135]. In fact, [117] found that Akka actors have up to 10 times higher message throughput and a network latency under 1ms, in contrast to the 0.2 seconds of Scala actors.

5.2.1 Striving for Isolation

While actors encapsulate state conceptually, in practice their *full isolation* must be ensured to avoid accidentally sharing state. This is essential to guarantee the safety properties, that is prevent data races and state modifications [52,84]. Akka offers interfaces for Scala and Java. Both languages support object-orientation in an imperative programming style – even though Scala is primarily a functional programming language. Since Akka is not directly integrated into either of these two languages[14], it cannot ensure isolation by itself. This restriction is true for most library-based actor systems running on execution environments that support shared-memory multithreading like the JVM [87]. Therefore it is especially interesting how Akka handles the isolated turn principle, because, as was outlined in Section 3.1, internal state of an actor must be mutable exclusively from within the actor itself to preserve the model semantics.

Issue of Data Hiding

Akka's actor runtime provides a transparent interface for component communication which exist either within the same local scope (same JVM) or remote scope (distinct JVMs). In the first case, we must take different notions of state into account. Kniesel [86] defines *weak state* as the state given through an object's instance variables. *Strong state* is the combination of *local state* (the object's instance variables) and *transient state* (the state of objects referenced by instance variables).

Actor semantics implies the need for a strict conception of encapsulation where the strong state is exclusive to the actor. We must not expose mutable local state outside the actor's scope, nor import mutable transient state into the scope of the actor. Violation of this requirement leads to the overlapping (sharing) of mutable state, which is in contrast to the message passing semantics of the model.

Visibility is a property of the variables and methods of an object which are part of the interface of the object [86]. Visibility is a concern for encapsulation and subsequently

[14]In Scala, Akka is the built-in actor *library*, but not a language feature.

shared mutability [44,132]. Java for example offers multiple granularities for visibility of class fields. The following code snippet illustrates the resulting problem:

```
public class Foo extends UntypedActor {
    public String bar;
    public static Props props() {
        return Props.create(Foo.class, () -> new Foo());
    }
    @Override
    public void onReceive(Object msg) { /* handle msg */ }
}
```

We extend `UntypedActor`, the base class for classic actors which do not provide type-safety for messages. In contrast, Akka's `TypedActor` is the base class for active objects, which do provide type-safety for messages [132]. We give the field `bar` in class `Foo` *external visibility* by declaring it `public`. The field is therefore part of every object of type `Foo` and influences the object's encapsulation [86]. From visibility follows accessibility, such that `bar` is also accessible from outside the scope of `Foo`. Since `bar` is not `final`, we can also modify `bar` from outside the object's scope. External modifications violate the requirement on exclusive state mutability of the actor semantics.

All Java-based actor implementations therefore face the problem that custom-written actor classes can easily break the required model semantics. In order to cope with this problem, object-oriented implementations can offer us APIs where we do not issue interactions with an actor instance directly through the instance's method interfaces, but instead via proxy constructs like [52]:

```
final ActorRef foo = system.actorOf(Foo.props());
```

We do not directly create an instance of `Foo` using the `new` keyword as it is custom in Java. Instead, we use the Akka system's factory method `actorOf` that hides the actual instantiation. The `create` method of `Props` takes a Java 8 lambda as an actor object factory. The lambda and the `create` call are commonly wrapped in a `props` method of the actor class. Java lambdas are basically functional closures[15] and only allow us to access effectively `final` fields inside the lambda's scope. This restriction prevents us from exposing mutable state to the constructor of an actor class. Of course, this is only true for the `final` fields themselves, but not their members (cf. Java case study in Section 2.4.1).

The actor system only exposes a proxy object of type `ActorRef` to the user. An `ActorRef` instance does not have the external interface of the actor class it represents. The `ActorRef` merely offers a variety of methods for sending messages to its actor. Messages sent through these methods are delivered by the actor system and then consumed by the actor through the `onReceive` method [132].

[15]Not to be confused with *Clojure*, a dialect of the programming language Lisp for the JVM.

The use of `ActorRef`s has the benefit that no direct contact with an actor instance object is possible. This lack of contact prevents both visibility and accessibility to any actor object fields or method calls. Additionally, `ActorRef` proxies enable *location transparency* [65].

References and Immutability

Preventing visibility of actor object fields and methods is not sufficient for guaranteeing the required strong state encapsulation on the JVM. The method signature of `onReceive` indicates that messages are received with type `Object`. Though Java has pass-by-value method parameters, variables with a non-primitive type (all besides `byte`, `int`, `char`, etc.) are actually reference variables storing the address to their objects. A passed-by-value parameter is therefore a copy of the object-address [46]. By implication, each message sent between actors contains a copy of the reference to the object representing the message[16]. In general, we must expect that the reference we send and the reference we receive point to *one and the same* object. Akka only serializes messages in case both counterparts are not within the same JVM [132]. In case both actors are on the same local JVM, a given message object is therefore in the scope of both the sending and the receiving actor. This message introduces shared state between these two actors, which is in contrast to the strong state encapsulation requirement.

However, messages are meant to represent snapshot information of a state at a given point in time. Therefore, shared state is not a problem if it refers to immutable snapshots, such that there is no memory with read-write or write-write access by two distinct actors [87]. Then the facts cannot be modified by either of the holders. The encapsulation requirement explicitly refers to *mutable* strong state, as immutability avoids what Akka calls the *shared mutable state trap* [66].

One option for Scala is to use `case class` constructs, which are immutable by default except for the transient state of the constructor parameters [65]. Java offers less syntactic support for expressing immutability. The property is neither formally defined in the *Java Language Specification* nor the *Java Memory Model* [44,46]. However, the basic requirement is to have `final` fields only[17]. This means that the transient state through internally referenced objects must be `final` too. In the author's experience, libraries which facilitate source-level annotation processing[18] provide useful tools for generating immutable value objects. These libraries use annotated `interface` declarations to generate consistent implementations offering builders and factory methods for instantiation [44].

All the restrictions we discussed so far still cannot prevent all obstacles Java and Scala offer to break the actor model semantics. Nothing can hinder an actor from sending a

[16]This causes the illusion that Java has pass-by-reference parameters. It does not.

[17]From a technical point of view, a class can have non-`final` fields and still instantiate immutable objects. `String` is a prominent example. However, deeper insight into the Java Memory Model is required. Goetz gives an outline of the principal approach [44, p.47].

[18]Such as `https://immutables.github.io` for example.

message to another actor containing the `this` reference of its object. `this` within an Akka actor is the standard self-reference object pointer and therefore not equal to *self* from the theoretical actor model. Akka provides us with the `self()` method that returns an `ActorRef` inside the actor for when we need to communicate the actor's location. But having access to the `this` reference of another actor breaks location-transparent access to the respective actor. Additionally, Java access modifiers are on class level instead of object level. If the recipient is of the same dynamic type as the `this` reference sender, then the recipient (after the corresponding typecast) has access to all `private` fields of the corresponding actor object. Though this visibility feature completely bypasses the encapsulation principle, it is intended behavior of the Java language design.

We see, a library-based actor variant like Akka cannot enforce strict actor semantics by itself, if the programming language offers ways to break the semantics to the programmers. Only a programming language itself can enforce a strict notion of the actor semantics, as does for example Erlang. However, in general it is sufficient if programmers comply to coding conventions specific to the language to avoid shared state by accident [135]. Conventions come with the burden of ensuring immutable value objects or manually deep-copying messages. Other actor frameworks like Orleans always provide deep-copied messages automatically, which comes with a performance penalty [20].

It is worth pointing out that though the actor model is Scala's standard concurrency variant, the language was not like Erlang designed to enforce strict actor semantics. Instead it accepts the perils that come with a library-based implementation. The arguments for a library are [54,135]:

- A library does not require special support by the compiler, JVM or extra syntax.
- A library can be easily extended, adapted, and even replaced. This has already happened, when the standard Scala actors have been replaced under the hood by Akka actors.
- A library can break the actor semantics *intentionally*, e.g. to introduce an additional concurrency abstraction, as the next section demonstrates.

Main findings

- Programming language features can jeopardize actor state encapsulation.
- Library-based actor systems cannot ensure isolation by themselves.
- Guaranteeing message immutability is the obligation of the programmer.

5.2.2 Utilizing other Concurrency Constructs

Section 3.5 motivated why we can combine the actor model with other abstractions of concurrency, as long as the actor semantics is not jeopardized. Akka offered support for several additional concurrency models. With version 2.3 however, Akka dropped the combination of actors and software transactional memory into so-called *transactors*. In principle, transactors have been useful for coordinating computations which span over the scope of multiple actors and require consensus between all of them [132]. However, transactional memory usage has never been able to abstract distribution transparently in Akka, since STM requires shared memory which is difficult across JVMs [133]. Transactor support was removed eventually.

Besides STM, much more prominently used is the future concept. Futures allow us to define concurrent computation *inside* an actor [67]. However, futures are not without perils of their own, as the following example illustrates:

```
var a = 0
override def receive = {
  case _ =>
    implicit val ec: ExecutionContext = context.dispatcher
    Future { a += 1 }
    a -= 1
    print(a)
}
```

First of all, Akka requires a so-called `ExecutionContext` in the actor's scope to run the future [55]. The example uses the actor's `Dispatcher`, which represents the thread-pool on which the actor runtime executes the actor. We can also specify a separate thread-pool instead [67]. Most importantly however, we can misuse futures to introduce nondeterminism into the scope of an actor. The example defines a mutable state variable. Upon receiving an arbitrary message, we dispatch a `Future` with the task to modify the state variable `a`. Concurrently, the actor continues to process the message and attempts to also modify the very same state variable. Due to the nondeterministic nature of the underlying thread-pool, multiple orders of execution are possible, and therefore also multiple results for the output statement. This is possible because Java as well as Scala do not provide any kind of guarantee regarding the safety of data inside the scope of a `Future` that exceeds the regular notion of safety of the respective language [141].

The isolated turn principle demands a guarantee that nothing interferes with the internal state of an actor except the actor itself, at the very least while processing a message. Yet futures have the potential to violate this constraint, thus breaking the actor semantics. Once again, Akka can neither check nor prevent this kind of concurrent modification. The programming languages visibility concepts simply allow us to pass mutable state into the scope of the futures. Then the safety notion permits the mutation of this state. Again, it is up to the programmer to ensure that only *immutable* state is introduced into the scope of a future [66,132].

There are also less expected issues related to futures. The Crawler retrieves the XML feeds via HTTP. In principle, HTTP is a synchronous communication protocol, such that there is always a response to every request[19]. Most APIs are therefore blocking as they

[19]The most basic form is merely a status code, e.g. the famous 404.

abstract over remote procedure call semantics. However, some APIs allow us to handle requests asynchronously by providing a future result. The author expected to improve the throughput of the Crawler when it retrieves feeds via an asynchronous handling of HTTP requests. The basis for this assumption was that HTTP connections to remote servers pose as potential bottlenecks (unknown server response time, network latency), thus reducing the liveness of the actor. However, this approach dispatches great many `Future`s simultaneously. Feed endpoints have a wide variation in response times and an asynchronous API allows us to start requests before the previous request has finished. All these futures stress the thread-pool of the Crawlers. We have experienced temporary starvations due to a lack of available threads in the Crawler's thread-pool. The author observed this effect with both asynchronous client APIs of the *Akka HTTP* [68] module and the *Apache HTTP Components* [40] library. Akka HTTP also provides a *flow*-based variant, where the concept of *backpressure* known from *stream*-based programming should limit throughput accordingly. However, the author experienced that the underlying *super connection pool flow* also introduces a limit to the amount of concurrent requests to a single host [68]. Feed publishers nowadays often choose to distribute their feeds via dedicated providers. As a result, a great amount of feeds are centralized on a small amount of hosts, rendering Akka's flow variant inapplicable.

Although simple RPC-styled retrieval did limit throughput, we have experienced this limitation as an advantage. The limitation puts a uniform and more predictable stress on the thread-pool, avoids problems like actor starvation and maintains their overall liveness. There are however still other cases where futures can come into play. The following section continues discussing future usage in the light of communication.

Main findings

- We can combine actors with additional compatible concurrency models.
- As with isolation, programming language features can jeopardize actor semantics when applying alternative concurrency constructs.
- Combining futures with actors can have a negative impact on performance.

5.2.3 Communication Abstractions

The actor model is solely built on the concept of *asynchronous* message passing. Akka provides a method called `tell`, with an additional alias `!` for Scala, on `ActorRef`. We use the method to send a message object to an actor. However, many real-life scenarios expect synchronous communication. Echo faces this problem whenever a user requests information, i.e. a search request through the Web application (`G` $\rightleftarrows$ `S` $\rightleftarrows$ `I`) or metadata retrieval from the CatalogStore (`G` $\rightleftarrows$ `D`). Fortunately, we can model synchronous communication with an asynchronous information flow [9].

Future-based Messaging

Akka provides a primitive to introduce a synchronous information flow. In addition to the asynchronous `tell [!]` command, `ActorRef` also offers the `ask` method, with alias `?` in Scala [65]. We use `ask` to model request/reply-style communication [54]. An `ask`-call resembles a `tell`-call in that it dispatches the method's argument as a message to the actor behind the reference. However, `ask` offers a result value which is the expected result of a synchronous call wrapped in a `Future`. The caller of `ask` is free to either proceed its computation, or go directly into blocking until the `Future` is resolved. This semantics resembles the future type message passing of active objects.

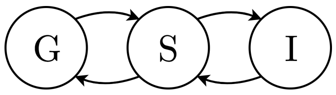

Figure 4. Example flow of a future-based synchronous call in the retrieval phase

A search request is the prime example of a synchronous call. Figure 4 shows how a request travels from the Gateway to the Searcher and finally to the IndexStore. The results travel back from `I` via `S` to `G`. On the client-side, the results are wrapped in a `Future` until they become available. We can resolve a `Future` inside an actor (e.g. a Searcher) in a waiting fashion, which causes the actor to block. The actor cannot process other requests until it receives the answer from the IndexStore. However, we need to avoid blocking if we expect the Searcher to process messages in reasonable time. We want to improve throughput. To prevent unnecessary blocking, Scala provides monadic methods for the `Future` trait that we use to define subsequent computation once the results are available. We can also utilize these methods when we dispatch several synchronous messages inside actors. Scala even offers specialized syntax through the so-called *for-comprehension* [55]:

```scala
val f1: Future[Int] = actor1 ? msg1
val f2: Future[Int] = actor2 ? msg2
val f3: Future[Int] = actor3 ? msg3

val r = for {
   r1 <- f1
   r2 <- f2
   r3 <- f3
} yield (r1 + r2 + r3)
```

It is important however that we dispatch the messages prior to the `for`-block's scope. Otherwise, it enforces sequential composition, if the `ask`-calls are inlined into the block scope [67]. This is because for-comprehension unfolds to monadic combinator usage of `flatMap` and `map`, which are sequential by nature [55]. The example above becomes:

```scala
val r = f1.flatMap(r1 => f2.flatMap(r2 => f3.map(r3 => r1 + r2 + r3)))
```

It is clear to see that if the ask-calls are inlined into the for-block, then the second message only gets dispatched once the first Future is resolved. Yet if used correctly, we can harness futures to preserve the single-threaded semantics of actors and still leverage parallel computation inside an actor.

However, this approach has two downsides. First, it is a load on resources, since every Future also stresses the actor's thread pool. Second, there is always the risk of accidentally passing the actor's internal mutable state into the Future's scope, thus introducing race conditions [132]. ask per se is therefore not ideal, but using futures with actors still has a long tradition [135].

Delegation-based Messaging

One of the basic actor primitives allows an actor to spawn new actors. We leverage this ability to model synchronous request handling. Hereby, we relocate the result handling to a dedicated child actor, individually spawned for each request. We create a child in Akka by:

```
val handler = context.actorOf(ReponseHandler.props())
index.tell(msg, handler)
```

Using context.actorOf instead of system.actorOf makes the response handler a direct descendent of the current actor. Providing the obtained ActorRef as a second argument to tell[20] sets the response handler as the official sender of the message. This way, we set the handler as the recipient to the response of the message. This dynamically created actor poses as a temporary component in the architecture:

Response Handlers (H)
exist with the sole purpose of posing as the original sender of a simple tell message dispatch and eventually receiving an answer in a purely asynchronous fashion. Upon message reception, a response handler passes on the result and deconstructs.

In the retrieval subsystem, the actual information flow turns from the concept definition $\mathtt{G} \rightleftarrows \mathtt{S} \rightleftarrows \mathtt{I}$ into the concrete realization $\mathtt{G} \to \mathtt{S} \to \mathtt{I} \to \mathtt{H} \to \mathtt{G}$ (Figure 5). Altering the reply destination is a form of the *delegation* concept known from object-orientation [144]. The overall approach is sometimes referred to as *cameo pattern* and mostly used for brief and simple interactions between actors [11]. The delegation pattern provides an asynchronous composition style to handle synchronous communication requirements. The approach is also more implementation independent, if the actor variant does not offer a handy concept like futures.

Time in general constrains synchronous communication. It is important that we process the messages for a cameo delegation swiftly. The actor model's mailbox construct

[20]Note that the ! alias of tell does not allow more than one parameter.

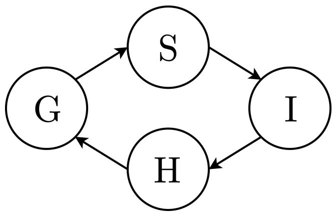

Figure 5. Example flow of a delegation-based synchronous call in the retrieval phase

however buffers all incoming messages to an actor in a strict FIFO (**F**irst **I**n **F**irst **O**ut) order. Large mailboxes with many messages queued up prevent timely message processing. A common property of actor systems is the ability to influence the order of message reception [87]. Akka provides the concept of a `PriorityMailbox`, which is utilizing the pattern matching syntax of Scala to assign priority levels to messages based on their type. IndexStores and CatalogStores facilitate priority mailboxes to process all messages of synchronous flows first, regardless of the current mailbox size.

Modelling Timeouts

Synchronous information flow requires a mechanism to implement timeouts in order to prevent starvation. Akka supports a special timing mechanism. Actors, e.g. newly spawned delegation-slaves, register to receive a timeout message after a given time period. The actor then simply needs to provide an appropriate message behavior to handle the timeout message. When the actor receives the expected message of the synchronous flow before the timeout message, the actor cancels the dispatch of the timeout message. However, if the actor receives the timeout message prior to the expected response message, then the timeout occurred and the actor performs a timeout reaction [11,118].

It is interesting how timed messages are introduced into the actor system. Timers require some sort of concurrent thread that constantly checks the current time and performs registered trigger actions. Dedicated thread-based concurrency is somewhat opposed to the actor model and message passing in general, where each action happens as a reaction to a received message, decoupled from a notion of time. Therefore, offering solutions for timer mechanisms is a concern of many actor systems, e.g. also Erlang [13] and Orleans [20]. To avoid interference of outside threads with actor states, Akka provides a special `Scheduler` instance that is unique for each runtime system. Actors register a timed message sending operation with this `Scheduler`:

```scala
val messenger: Cancellable = context.system.scheduler
    .scheduleOnce(5.seconds) {
        self ! TimeoutMessage
    }
```

The provided `Cancellable` reference allows us to prevent the trigger from firing by calling `messenger.cancel`. This scheduling mechanism introduces a notion of time into the actor semantics that feels natural to the actor model [69]. In combination with the cameo pattern, we can implement synchronous information flow semantics including time constraints by using purely asynchronous message passing operations.

Type-restricted Messages and Compatibility

One of the basic actor primitives allows actors to send messages to other actors. However, in general this does not define restrictions on the types of messages that are sent. Messages are untyped in the theoretical actor model. The behavior of an actor decides at runtime whether it is able to handle the message.

The active object concept aims to provide a higher-level abstraction and provides static guarantees for message types. Active objects leverage the method signatures of the standard object model for static type checking. However, the active object model also uses the method's signature to define the method's dispatch semantics. The work of Waldo *et al.* [138] describes in detail when and why this abstraction becomes problematic, especially in a context with transparent distribution. In short, the abstraction provided by the object model does not incorporate some effects of distribution well. Examples for such effects are latency, memory access (via pointers), partial failure and concurrency effects in general. The transparent abstraction of local and remote execution is therefore problematic with active objects. The object model must explicitly distinguish local and remote interaction to be robust and reliable. The classic actor abstraction on the other hand is resilient without explicitly distinguishing between local and remote message recipients. Current developments in Akka therefore address the challenge of offering some level of type safety for messaging without using Akka's `TypedActor`s for active objects. The APIs are summarized under the name *Akka Typed* [70].

One part of Akka's isolation strategy is to never expose a reference to an actual actor instance directly. Instead, all communication happens via the messaging interfaces of the `ActorRef` proxies (`tell`, `ask`). These interfaces take arbitrary types as messages. Akka Typed introduces a generic type parameter to the address, i.e. `ActorRef[U]`. The range of accepted messages is then limited to `U`-typed objects. The signatures of the messaging methods changes from `tell(msg: Any)` to `tell(msg: U)`.

It is worth pointing out that in Scala the actor messages have `Any` as the most general type. In Java however, `Object` is the most general message type. This discrepancy in the typization is somewhat counterintuitive, since Akka offers compatible bindings for both languages. The type systems of Java and Scala deviate however. Java distinguishes between reference types with `Object` as $\top$ in the type hierarchy and primitive types (`int`, `char`, etc.) which are not subtypes of `Object`. Hence, we cannot use primitive types as messages. In Scala on the other hand, all types have `Any` as their unified $\top$. The direct descendent `AnyVal` is the supertype of all value types (`Int`, `Char`, etc.), while `AnyRef` corresponds to Java's `Object` and is therefore also supertype to all non-value types. If we send an `AnyVal` from a Scala to a Java actor, the corresponding primitive type's wrapper class (`Integer`, `Character`, etc.) is received and vice versa.

In any case, type-restricted actor addresses are not without drawbacks. The third basic model primitive states that actors can change their behavior. When we restrict accepted messages by an `ActorRef[U]` that is hiding behavior changes transparently, we must also constrain a new behavior to process messages of type `U` exclusively. Otherwise, the address of the actor represented by the `ActorRef[U]` breaks the semantics and becomes invalid [70]. To ensure address and behavior compatibility at compile time, we must define type-restricted actors through a behavior function that is also restricted by a type parameter:

```
val behavior: Behavior[U] =
    Actor.immutable[U] { (_, msg) =>
        // process msg
        Actor.same
    }
```

In this example, the typization `msg:U` is guaranteed. The `Actor.immutable[U]` factory prevents the constructed behavior from holding and passing over mutable state [70]. Every behavior has to specify the replacement behavior for the next message explicitly. In this example, `Actor.same` declares that the replacement behavior stays the same.

Though Akka Typed provides static type safe messaging, it still has its limits. For example, Akka Typed is not able to statically ensure that a behavior is in a certain state. The association of an actor's address and behavior is a fundamentally dynamic property of the actor model [70].

We have found that compile-time message safety restrictions, apart from active objects, are a rare capability among actor systems. However, the Akka Typed library is in the current Akka version 2.5 still under active research. It is marked as “may change” and to be considered experimental.

Main findings

- Abstractions for synchronous-styled actor communication require additional concurrency models (futures) or additional actors (delegation).
- An explicit sense of time is contrary to the actor concept. The runtime must handle time and communicate it in terms of messages.
- The actor model intrinsically types all received messages dynamically. Type-restricting the actor addresses constrains the range of acceptable messages.

5.2.4 Supervision and Monitoring

In Section 3.3 we have pointed out that actor systems aim for higher-level constructs than the low-level basic model primitives. One example for higher-level abstractions are the different messaging styles Akka facilitates. Another important reason for providing more expressive constructs is the encapsulation of faults [2]. Being aware of the possibility of faults and handling them adequately is key, especially in a distributed context [30]. Orleans for example directly reports exceptions back to the message sender [20].

The key to handling faults in Akka is its concept of *supervision.* No actor exists for itself, but is always a subordinate to its supervising actor. We call this dependency relationship a *supervision hierarchy.* The actor system provides a default supervisor at the top-level, which has the eventual supervision of all other actors [71].

The hierarchical relationship is fairly simple: a supervisor delegates work to its subordinate children, but has to *monitor* each child in return. Monitoring is the concept of getting notified of a subordinate's failures, and in turn reacting to these failures. A fault can be of arbitrary nature, i.e. an unhandled exception or invalid state. An actor automatically suspends itself and all its subordinates upon the occurrence of a failure. The runtime then notifies the superior, which has to provide a response to the failure [11,118].

The signaling of an occurred failure is not communicated via a "normal" actor message, but on a side channel [71]. A so-called *supervison strategy* handles the failure notifications. This strategy can take one of four possible actions:

Action 1: Resuming the suspended child, when the fault can be safely ignored.
Action 2: Restarting the suspended child, if its internal state is invalid.
Action 3: Stopping the child completely by not continuing its execution.
Action 4: Escalating the failure, hence the supervisor fails itself.

A supervisor's failure notification has the sole form of an exception. The actor system does not offer state information that puts this exception into context. The reason is that state should only belong to and be processed by one actor exclusively. Let us assume that a failure results in the propagation of state from the child to the supervisor. Then a part of the child's implementation logic leaks into its supervisor. The supervisor requires the knowledge to interpret the state in order to evaluate it in a meaningful way. However, Akka isolates the failure in the child. In case the child gets resumed (Action 1), the cleanup falls to the child itself. The runtime does not re-introduce the message on which the fault occurred into the mailbox, to avoid fault-reoccurrence. Actions 2-4 discard the child's state in any case [71,118].

5.2.5 Information Routing and Delivery Reliability

We must frequently send a message not to one actor specifically, but distribute the message among a set of equivalent instances. Akka provides an appropriate concept called *routers*:

```
val router: Router = {
    val routees = Seq(parser1, parser2, parser3)
    Router(RoundRobinRoutingLogic(), routees)
}
router.route(xmlData, sender())
```

When a Web Crawler needs to pass on the XML feed data to the Parser, the Crawler does not directly select the recipient. The selection of recipients follows a strategy that depends on the used `RoutingLogic`. The `RoundRobinRoutingLogic` of the example redirects the message to the next routee in a cyclic order. Many alternative routing strategies are available, e.g. `RandomRoutingLogic` for a random recipient, and `BroadcastRoutingLogic` to distribute the message to all routees [11].

Conceptually, we can do the routing directly inside the sending actor by replacing the `tell(m,s)` call with `router.route(m,s)`. Alternatively, we can also employ an intermediate actor. The decision is based on the burden of managing the routees.

We have to manage the set of routees and keep it up to date, since actors can fail and we must not send any message to a terminated actor. If the routees are children of the routing actor, then the router gets informed through its supervision obligation in the case of a subordinate's demise. A supervision-managed set of routees is called a *pool* [118].

Alternatively, routees are part of a so-called *group* when the routing actor does not supervise the routees [118]. Akka provides the so-called *actor selection* mechanism to send a message to an address matching a certain pattern. When no supervision relationship exists, message delivery reliability becomes a special concern. Actor selection does not guarantee that a recipient for a selection pattern exists. Therefore, a message becomes a so-called *dead letter* when the runtime cannot deliver the message.

Delivery reliability is also a general concern of sending messages to actors, besides the context of routing. The theoretical actor model guarantees that all messages are always delivered [7]. Conceptually, this insurance is important since it implies that no actor can permanently starve [85]. In practice however, we cannot safely assume perfect delivery reliability to hold. Because we consider actors in a potentially distributed context, message delivery can be subject to a network link. Recalling Fallacy 1: *The network is reliable* warns us that we cannot trust the network to transport the data in general. As a result, neither can we assume actor message delivery. Akka therefore provides a weaker insurance regarding message delivery reliability than the theoretical actor model. Particularly, all messages are merely delivered *at-most-once*. Additionally, when several actors send messages to the same recipient, there is no guarantee of a general order of the messages in the mailbox. The runtime merely enqueues the messages of each particular sender into the receivers mailbox in the same order as the sender dispatched the messages (FIFO order) [72].

In contrast, other actor systems like Orleans provide *at-least-once* delivery. They resend messages that were not acknowledged within a certain timeframe [20]. No message is ever lost but it can emerge duplicate instead. As a consequence, the application logic of actors with at-least-once delivery must cope with the fact that the actor can receive one and the same message several times.

Main findings

- Actors exist within a hierarchical supervision structure.
- Failure is communicated along side the hierarchy while state does not leave the boundary of an actor.
- Supervision is also useful for complex message routing logics.

5.2.6 Persistence and IO

We must employ some kind of data persistence mechanism, e.g. a database, in order to persist the internal states of actors. Echo's CatalogStore is a prime example. Due to the single-threaded semantics of actors, only a single interaction with the database is possible at the same time. This is inefficient, since database access is input/output and therefore a performance limiting factor in general [132]. Theoretically, the single-threaded semantics of actors makes database transactions obsolete, but the author found that common APIs demand an active transaction in any case, e.g. with providers of the **J**ava **P**ersistence **API** (JPA).

We can overcome the single-threaded limitation when we utilize a non-blocking API for database connections. Such APIs provide a `Future` reference to an eventual result at the cost of additional stress to the actor's thread-pool. With the monadic methods of `Future` we define further computation on the result once it becomes available. The price is the immanent risk of accidental data races when we pass mutable state into the `Future`'s scope.

An example for a situation where we cannot apply a non-blocking API is the experiment for the benchmark we describe later on in Section 6.2.3. We need an alternative strategy. The delegation principle we discussed for synchronous message handling is also applicable for database interaction. We can handle several database interactions concurrently by having as many actors communicate with the database at the same time.

Each CatalogStore actor has a database. Conceptually, we use the database to persist the state of a single actor. The state is not exclusive anymore, if several actors manage the same database. Yet we need several actors for concurrent interaction. Hence, we have to intentionally weaken the encapsulation principle. Although the database conceptually belongs to the CatalogStore, the store actor delegates all database interactions to it's children (Figure 6). A `RoundRobinRoutingLogic` distributes the database interaction operations between the child actors. Now, the CatalogStore architecture component consists of several task units of the actor model. All these actors conceptually share a single persistent state. We must not break the isolation of each actor however. Therefore,

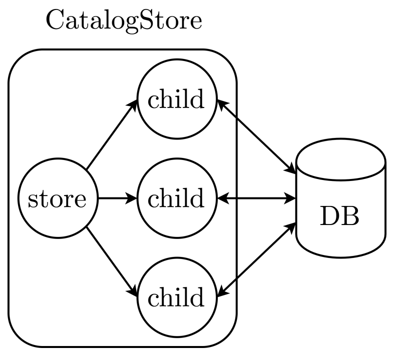

Figure 6. Example of a stateful actor sharing its persistent state with several child actors: The store delegates all database interactions to the child actors

these children neither share the same database connection interface object, nor any other mutable data. We merely loosen up the restriction on encapsulating the persistent state inside one single actor. A narrow group of actors is managing the persistent state instead. All these actors must utilize the database system's transaction mechanism. We use a dedicated dispatcher for all actors involved in the logical encapsulation of the persistent state. The underlying thread-pool uses a fixed number of threads. This provides a predictable impact on performance, in contrast to Akka's default dynamically sized thread-pools. Dynamic pools add threads when demand is high, i.e. due to many blocking operations, and can therefore consume a lot of system resources [11]. The idea of actor-managed databases is not novel. Shah & Salles give an entire manifesto on what they call *actor database systems* [125].

Main findings

- Efficient handling of persistence and IO in general uses the same strategies as for synchronous actor communication.
- Concurrent database interaction via delegation forces us to intentionally weaken the conceptual encapsulation of actor state.

5.3 Microservice-based Implementation

This section covers the strategies we apply when we program with the microservice model. Again, we implement the backend of the concurrent system which we outlined in Section 5.1. All concepts we discuss are with respect to a specific technology stack. The focus is on the linguistic support provided by the technology stack. Efficiency considerations are part of Chapter 6.

5.3.1 Service Technology Stack

Service-oriented programming languages seem to be a good choice for a microservice architecture. Though some languages are theoretically matured, from a practical point the available SOP languages are as of yet still at an early prototypical stage. Jolie for example still misses an ecosystem and tool support we are looking for in comparison to Akka. Therefore, we refrained from using Jolie to implement Echo. Instead, we compose microservices using a more traditional technology stack. We use Java as the programming language for all services and the *Spring* [78] framework, most notably its *Spring Boot* [79] module, for application fundamentals. Additionally, we apply many libraries of the *Spring Cloud* [80] collection, which have proven very effective for microservice development in industrial applications [26]. For the webserver, we configure Spring to use *Undertow* [29].

Spring is based around the concept of *inversion of control* (IoC). Spring's IoC variant applies *dependency injection*. We do not instantiate objects directly, but instead define for certain classes what kinds of dependencies they need (i.e. we declare fields but do not initialize them directly). These kinds of classes are called *beans*. A so-called IoC *container* instantiates these beans and *injects* their dependencies through constructor arguments, factory methods or setter methods. An IoC container's primary responsibility is therefore the management of beans[21]. We call this *lifecycle management*. The actual execution logic of a bean, i.e. the scheduling on a thread-pool, is also left to the IoC container [81,139]. Therefore, Spring's IoC container is the internal concurrency managing system of each of our microservices. Conceptually, IoC management has a certain resemblance to the execution of actors by an actor runtime [53].

Spring Boot is primarily an application skeleton based on the Spring framework. The skeleton is a fully executable application without domain-specific functionality. We use Spring beans to add custom configuration and the functionality we want the application to fulfill [79]. In other words, Spring Boot provides us with a general foundation for the microservice's engine, and we as programmers merely add the specific service behavior through custom beans.

Spring Cloud is a set of libraries which focus on features like data access, messaging, streams, discovery, load balancing, circuit breaker, and many more [80]. The term *cloud* indicates that the libraries are intended for cloud computing scenarios. This makes them useful to us since the industry uses microservice architectures for cloud deployment scenarios [23,35].

As we did with the actor implementation, in subsequent sections we pay attention to the linguistic support provided by the framework regarding the expression of service requirements.

[21]Sometimes Spring beans are therefore referred to as *managed objects*.

Main findings

- We can leverage inversion of control frameworks to reduce the effort that comes with writing many separate applications for MSAs.
- Programming with IoC transfers the management of concurrent execution to the IoC container.

5.3.2 Internal Service Concurrency

The microservice model paradigm does not dictate restrictions on the internal service structure. A service is free to apply concurrency internally, and to utilize every concurrency mechanism it sees fit (including actors). We build the Echo services on top the Spring framework and utilize the concurrent programming structure that Spring's IoC container provides. Spring-based microservices receive requests as method calls to beans. We discuss the concrete communication mechanisms in Section 5.3.4 below. The respective bean classes have a so-called *stereotype annotation*[22] decoration (e.g. `@Component`, `@Controller`, etc.). The IoC container turns each of these method calls into a so-called *task* by wrapping the call into a `Runnable`. Every task gets appended to a task queue. We have already demonstrated how to wrap method calls into a `Runnable` in the Java concurrency case study of Chapter 2. Spring's `TaskExecutor` constantly processes the task queue. An `Executor` is Java's variant of a thread-pool. An allocated thread of the thread-pool eventually executes each task [81].

Since microservices are concurrent internally, we have to pay attention to their shared internal resources. All shared resources must be either immutable as the messages in Akka, or we must synchronize the access to the resource. Spring uses software transactions for synchronization.

Spring provides linguistic support in a *declarative programming* style for many strategies and mechanisms. This is particularly interesting since the declarative style is not intrinsic to Java's imperative programming concept. However, the language allows us to introduce declarative programming through *annotations*, so that the IoC container then applies appropriate behavior. In contrast to annotation processors, where the compiler reads annotations to influence the compilation process (e.g. to generate class implementations), Spring uses *reflection* at runtime. The result is a form of *aspect-oriented programming*. We leverage the declarative style for synchronization by defining software transactions on method calls using the `@Transactional` annotation. Spring also extends the STM to database transactions transparently [81,139].

While Spring enqueues each request into a concurrent task queue automatically, we as programmers also want to leverage this technique to achieve a higher degree of concurrency inside a microservice. The `@Async` annotation allows us to declare methods that

[22]Stereotypes in Spring describe certain patterns for beans. Hence, we expect special behaviors of stereotype-annotated beans.

we want to dispatch asynchronously. When we call an `@Async`-annotated method, Spring wraps the call into a `Runnable` and enqueues the resulting task into the task queue of the `TaskExecutor`. Since the method executes at an unknown point in time for the caller, we cannot expect a result value directly. An `@Async` method is therefore `void` in general. Alternatively, we return the expected result wrapped in a `Future` [81]. We have discussed this idea for active objects and their future type methods already. The future enables us to return an intermediate result and provides an interface to check whether the actual result is yet available. The author has experienced that Spring's default `AsyncTaskExecutor` does neither handle nor log the occurrence of exceptions. We have found this factor troubling for development. Therefore, we use a custom asynchronous task executor implementation capable of handling exceptions to fix this flaw.

Although Spring's declarative programming style for concurrency is very powerful, we have also experienced some limitations. The `@Async` and `@Transactional` annotations only show effect on `public` methods. Additionally, self-invocation of an `@Async` method does not spawn a new asynchronous task, but instead executes within the same task in a synchronous fashion. Combining `@Async` and `@Transactional` is also possible. However, the asynchronously dispatched method does not run within the same transaction as the dispatching method, but in a fresh transaction instead [81].

Main findings

- The IoC container builds on Java's standard threads, but exposes a completely different programming interface (implicit concurrency, `@Async`, STM).
- A declarative programming style can simplify the handling of concurrency issues in an imperative programming language.

5.3.3 Isolation and Persistence

The incarnation of a microservice is by definition a system level process. The foundation of the isolation of services is the strict memory boundary of every process that the operating system enforces. By convention, services refrain from implementing shared memory sections among them. We therefore never require synchronization among the components. The principle must also extend to the persistence of state.

We can persist information with many different strategies. Database systems are one well established approach. If we share a database between several microservices, every service gets access to all other component's persistent state. Shared databases are a simple way to skip isolation mechanisms and bypass the service interfaces, thus breaking the encapsulation principle. Therefore, one convention of the microservice principles is that each service owns its databases exclusively [34]. Sharing a persistence mechanism conceptually relates to sharing state, which introduces an implicit form of shared state communication [30]. As a consequence, we must provide each component with its very own database instance if we require persistence. We deploy every CatalogStore instance with a dedicated database instance, and every IndexStore has a separate Lucene reverse index data structure.

Persistence is a form of IO and a potentially performance limiting factor. Hence, database management systems usually support concurrent access to the database. As with actors, concurrent connections increase the throughput of the component. The *Spring Data* module offers us a good interface as well as a transparent abstraction to interact with the database in a concurrent way [139].

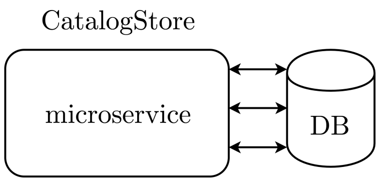

Figure 7. Example of a stateful microservice maintaining several concurrent connections to an exclusive database

Since Spring executes every request concurrently inside a `Runnable` task on a thread-pool, concurrent database interactions are implicitly available. Every thread of the pool can interact with the database at the same time. The transactional memory of Spring extends to database transactions. Programmers do not have to pay heed to or apply additional strategies to leverage efficient persistency through database concurrency. Hence, we expect a microservice to have several database connections in place at the same time (Figure 7).

Main findings

- State is exclusive to the entire microservice.
- Isolating state must extend to the isolated persistence of state.
- Concurrent access to persistent state is transparently possible.

5.3.4 Communication Mechanisms

Communication in microservice architectures happens via inter-process communication mechanisms. Various kinds of interfaces are possible. While communication for actors happens in a uniform way, microservices in general face more challenges. The freedom in the design of services does not dictate a specific communication interface. The only restriction regarding the interaction is that we need to omit shared memory between services. Solely relying on message passing mechanism makes services cohesive and loosely coupled.

We have found scholars to give REST (**Re**presentational **S**tate **T**ransfer) as the prime (and often sole) example of valid communication channels throughout the literature. However, the author experienced that REST is practical only in certain situations. Since

REST builds upon synchronous HTTP, it is a good solution for synchronous requirements. Echo facilitates REST for search requests through the Web application (`G` $\rightleftarrows$ `S` $\rightleftarrows$ `I`), as well as metadata retrieval from the CatalogStore (`G` $\rightleftarrows$ `D`). As even Fowler & Lewis [42] in their seminal work on microservices point out, other mechanisms are applicable as well, as long as they are lightweight and do not apply logic of their own. The indexing subsystem is more predestined for an asynchronous workflow. Therefore, we desire a message queue-like mechanism. JMS (**J**ava **M**essage **S**ervice) [30] is a prominent example among JVM technologies. However, JMS is also limited *to* the JVM, which contradicts the open and well-defined interface principle of microservices.

We require a technology-heterogeneous message queue standard. AMQP (**A**dvanced **M**essage **Q**ueuing **P**rotocol) [1] is an open specification for asynchronous messaging. While JMS only defined the interfaces, AMQP also defines the message format. Therefore, different implementations exist which we can interchange freely. Echo builds upon *RabbitMQ* [82], a messaging system that proved to integrate well into MSAs, according to the literature [33]. A message queue conceptually introduces a new concurrent component into the architecture:

Message Queue (Q)
: is a distributed point-to-point communication channel. The queue offers message delivery from a sender to a qualified receiver (possibly unknown to the sender), decoupled in time and space (asynchronous) [30].

The queue becomes an intermediate for all asynchronous messages. Senders push messages to the queue, and receivers subsequently pull those messages from the queue. For example, we do not send a message directly from a Web Crawler to a Parser (`W` $\rightarrow$ `P`), where the active component is only `W`. Instead, the Crawler pushes a message the queue (`W` $\rightarrow$ `Q`). At some later point and idle Parser pulls this message from the queue (`P` $\leftarrow$ `Q`). The message travels asynchronously. The actively communicating components are `W` and `P`. The queue merely performs internal routing logic and reacts to requests from others. The queue also decouples the sender `W` from the actual receiver `P`, i.e. `W` does not know which concrete `P` receives the message.

In general, a service can have several different interfaces, based on heterogenous technologies. As a result, this allows the service to provide the *same* functionality on *different* interfaces. Since all interfaces of Echo's microservices produce and consume messages in JSON (**J**ava**S**cript **O**bject **N**otation) format, there are no data type incompatibility problems. Echo's microservices provide a REST interface for every message a service consumes from the AMQP queue. We can therefore also send a message to a service directly via HTTP. The additional option to invoke service functionality turned out especially useful for testing and debugging purposes when we implemented Echo. This suggests that it is valuable to maintain different interfaces for development, production, and maintenance.

Programming Abstractions

We have already seen that Spring provides a declarative programming style through annotations. The IoC container applies a behavior to a bean based on an annotation using reflection. The benefit of reflection is that we can still apply deployment configuration without the need to recompile, which is especially useful for communication configuration. The downside is additional runtime overhead and the lack of static compatibility checking. For example, a Searcher queries an IndexStore using a synchronous REST call (`S` $\rightleftarrows$ `I`). We use the Spring binding for *Feign* [110], a library dedicated to annotation-based decorations for Java interfaces. Clients consume RESTful endpoints using a dynamic interface implementation. We express the example in the Searcher through:

```
@FeignClient(name = "index")
public interface IndexClient {
    @GetMapping("/query")
    List<Result> query(@RequestParam("q") String q);
}
```

The stereotype annotation `@FeignClient` is for REST clients. Feign automatically generates an implementation class of the given `IndexClient` interface. Spring's IoC then instantiates a bean of this implementation class. Calling the `query` of this bean dispatches the REST call in a blocking fashion. The method only returns once the result from the IndexStore is available. Mapping the HTTP body content (in JSON format) of the response to domain objects happens transparently, provided that we configured a JSON serializer for the IoC container. As a result, we can use every domain object for the method result type. The IndexStore receives the request by declaring an appropriate REST-endpoint using a similar annotation driven implementation approach. `@RestController` is the stereotype annotation for beans that receive REST calls:

```
@RestController
public class IndexResource {
    @GetMapping("/query")
    public List<Result> query(@RequestParam("q") String q) {
        // query reverse index for phrase q
    }
}
```

This approach models a remote procedure call between the two components. The call `query("TU Wien")` of the `IndexClient` in the Searcher results in a call of `query(String)` method with argument `"TU Wien"` of the IndexStore's `IndexResource`. The service's programmer invokes the method on the client-side. Then, the receiver's inversion of control container registers the request on the transparently exposed REST interface and calls the method on the server-side. We can express message queue interaction in a similar fashion through respective AMQP stereotype annotations. The resulting beans interact then via RabbitMQ. While the above REST example declares a synchronous API, the AMQP annotations declare asynchronous APIs. An interface method call on the client-side returns before the server-side receives and processes the message.

Service Discovery

Message queues decouple the sender from the receiver. Therefore, the sender neither requires nor knows the address of the actual receiver. For direct communication like REST however, we require the actual address of a recipient. Yet in certain deployment scenarios this information is not statically available. We apply the concept of *service discovery* known from SOA [30] to bridge this lack of static information. The so-called *registry* is a dedicated service component that provides binding information about other services. We merely predefine the connection to the registry statically and are obligated to ensure the availability of the connection at runtime. Microservices then register with the registry service, in order to be discoverable by others [105]. This dedicated service adds a new concurrent component into the architecture:

Discovery Registry (D)
is a centralized service and provides address information for dynamic connections. Other services register with the registry service under a name and their address. Clients lookup the current address of registered services for a given name.

The `name` argument of the `@FeignClient` annotation in the REST example we gave before relates to the name we use to register the IndexStore unit. The advantage of Feign is that it automatically integrates with discovery mechanisms. Examples for service registry technologies are *Consul* [56], a standalone registry service solution, and *Eureka* [75], a module of Spring Cloud to add registry capabilities to custom applications. Echo supports Consul, but uses a dedicated service based on Eureka by default. The author of this thesis experienced Consul as very resource demanding in comparison.

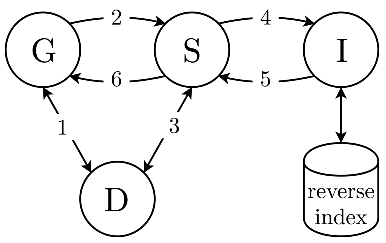

Figure 8. Example of service discovery usage in the retrieval phase: The consecutive lookups delay the overall synchronous communication

Discovery mechanisms impact the response times of services. It can become necessary to lookup an address before a service is able to make a request. The service must then make additional RPCs for registry lookups, in the worst case for each of the involved services. Figure 8 shows the order of interactions in the worst case for search requests in our scenario. When all location information is outdated, then `G` must first lookup `I` with `G` ⇄ `D` (1) before it can do `G` → `S` (2). Subsequently, `S` must lookup `I` with `S` ⇄ `D` (3) before it can send `S` → `I` (4). This dampens the liveness of the request flow for our search results (`G` ⇄ `S` ⇄ `I`). We also need to ensure that the information in the registry is correct and up to date. *Health checks* are a common feature of discovery services to determine if their registrees are actually up and alive [33].

In contrast, message queues have the major benefit that a sender dispatches a message to the queue without needing the address of the receiver. This circumstance provides a lower degree of coupling as well as a notion of location transparency between sender and receiver. Therefore, when we use message queue channels, we do not need service discovery technology if we statically know the queue address. Otherwise, the queue needs to register with the discovery mechanism. The clients then simply retrieve the queue's address dynamically.

Load Balancing

The idea of distributing work (*load*) between several instances of the same task unit is called *load balancing* (LB). The goal is to optimally utilize the resources of all instances and prevent that a single unit is overloaded. Load balancing maximizes throughput and minimizes response time of the overall system [18]. There are two directions towards load balancing. A central supervising entity that distributes the work between receiving services is balancing load *server-side*. Spring Cloud offers *Zuul* [76] to create balancing server. In contrast, a service that distributes the work itself is doing *client-side* balancing [30].

Echo's microservices use the Spring Cloud module *Ribbon* [77] for client-side load balancing. The main reason for Eureka over Consul as the Discovery service is that Ribbon integrates with Eureka. A Ribbon client does not dispatch a message directly to an address. Instead, Ribbon uses a static name to lookup the current address of a server from the discovery service. Ribbon can then balance individual requests directly on the client-side across server instances, as it cooperates with Eureka to maintain a set of valid instances of a static name [26]. This name is in fact the `name` argument for the `@FeignClient` annotation of the declarative REST interface, since Ribbon integrates transparently into Feign.

Main findings

- Different communication styles require different communication channels.
- Microservices are free to serve the same functionality on different communication interfaces and technologies at the same time.
- Microservices always serialize and exchange data in a technology neutral format (e.g. JSON or XML). This prevents data type compatibility issues.
- Location transparency is not intrinsically available in MSAs. Network communication coupled with discovery technology adds this feature.

6 Evaluation

Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction.
— Jeannette M. Wing

In Chapters 3 and 4 we introduced the concepts of actors and microservices. Chapter 5 described the strategies of each model to implement concurrent systems like the Echo scenario. We discussed each model separately and focused on the individual concepts of each model. In this chapter, we compare and evaluate both models relative to each other. As we have demonstrated, both actors and microservices qualify for expressing concurrent computation. Their mechanisms and abstractions also support parallel computation on multicore processor as well as distributed execution on multiple hosts. Several authors [5,84,85] suggest that programming models regarding parallel and distributed contexts should be evaluated based on two objectives: *expressiveness* for programmers and *efficiency* of execution.

Section 6.1 compares the key properties that are the foundation of the concurrent execution of both models and the resulting capabilities. The model capabilities allow us to evaluate the expressiveness of each programming model. Section 6.2 provides a benchmark for the actor- and microservice-based implementations of the Echo scenario. The results of this benchmark evaluate the efficiency of the programming models.

6.1 Expressiveness and Capabilities

In this section, we evaluate and compare the key properties of the actor and microservice programming models, as well as the capabilities we can express with these models. Programming language theory knows a concept called *expressiveness* or *expressive power*. This concept also becomes more and more relevant in the light of concurrency theory. There, the *relative expressive power* is used to compare two formal concurrency models [37,45]. Evaluations on a strictly formal level require the rules and boundaries of formal frameworks. Therefore, often programming languages are analyzed. Concurrency theory focuses on *process languages*. These languages are founded on the formal frameworks of so-called *process calculi* [112], which we briefly discuss in due course. Here, we do not concern ourselves with formal proofs of behavioral equivalence however. Instead, we revert to informal discussions about the observational equivalence of concepts we can express in the actor and microservice model, as has been often done in the literature before [37]. All strategies that we express in both models are essentially encodings of ideas [45]. Ultimately, we are interested if actors and microservices have the expressive power to convey the same ideas. The two models are equally capable to express a concept if the solution of one model produces an equally powerful functionality (informally) as the solution of the other model.

6.1.1 Encapsulation and Isolation

Actor and microservice semantics rely on the strict separation of component states. We must ensure that state is conceptually well encapsulated within a component, and practically isolated from the outside. Encapsulation is a concept we know from object-oriented programming. As Snyder [129] points out, OOP usually offers mechanisms to restrict the access to an object's state. A client gets access to an object's state by issuing requests for the *services* an object offers. These services – not to be confused with the concept of a *microservice* – are what Meyer [97] calls *well-defined interfaces* in the form of *routines*. Meyer considers these services a necessity for encapsulation.

Both actors and microservices offer well-defined interfaces in their own way. For an actor, the interface is the sum of the messages the actor understands through its behavior. For microservices, the interface is based on the sum of the facilitated communication channels, e.g. the REST interfaces the service exposes and the messages it consumes from a message queue. Only through the interfaces can we access or modify the service's state.

Table 1 at the end of this section summarizes the encapsulation-related matters as they are facilitated by Akka actors and Spring-based microservices.

Shared and Mutable State

One fundamental characteristic of both actors and microservices is their notion of shared state. Summarizing Chapter 3, the actor model encapsulates state exclusively within an actor. Therefore, the state is only accessible to and modifiable by the actor itself. Additionally, actors provide single-threaded semantics while processing messages. An actor processes only one message at a time. The isolated turn principle eliminates the need for synchronization, since message are free of low-level data races, and a turn has exclusive access to the current state. The actor's state is fully isolated.

On the other hand, we cannot apply the same reasoning for microservices in general. The paradigm states nothing about how state has to be handled internally. Depending on the programming paradigm we use to implement the service, state is not necessarily exclusive internally. In OOP for example, several objects can access the same memory location. Furthermore, concurrent access to the state is also possible, e.g. as a reaction to several invocations of the service's interface within a short time span. Microservices do not ensure single-threaded semantics. In general, we therefore assume that we must synchronize the access to the service's internal state. Additionally, the microservice paradigm dictates that we must avoid shared memory *between* services, as well as all kinds of shared resources in general. Since every service runs within a dedicated system process, avoiding shared memory implies no direct intersection between service process boundaries. Typical communication channels satisfying the requirements given in Section 4.3 also prevent reference sharing to joined mutable data. All viable channel technologies provide some sort of message passing designed to transfer information between the different memory spaces of distinct processes [90].

Actors generally face more challenges when it comes to truly ensuring state separation. The components can exist within the same process boundaries and access the same memory locations. Depending on the programming paradigm we use to implement the actors, exposing shared state to others can be rather easy. At the same time, exposed state is not necessarily apparent to the programmer. Think of the example we discussed for Akka. We can use an arbitrary object as a message. If we do not construct an object in an immutable fashion, the message transfer shares the object's state. Concurrent access and modification to this shared state causes unpredictable runtime issues. Especially the imperative programming style leans on mutable state. Actors in imperative languages therefore require extra care to preserve the model semantics. The functional paradigm tends to avoid these problems inherently. Functional languages model the behavior as a function and only this function is modifiable exclusively by the actor. If the actor model is integrated into the programming language directly, the language is able to enforce restrictions preventing shared mutability problems by design. Erlang is the prime example [137]. However, library-based actor implementations cannot ensure full isolation by themselves [87].

Persistence and IO

The encapsulation and isolation principle of actors and microservices has one more important implication. Recall that state is exclusive to a single component. Hence, each component must also take exclusive care if we require durable state across the component's lifetime. In our scenario, every CatalogStore must have its exclusive database and every IndexStore its exclusive reverse index data structure. However, neither an actor runtime nor a microservice technique can enforce this persistence restriction. The obligation of correct configuration lies solely with the programmer.

We also desire concurrent interaction to persistence mechanisms to increase throughput. Microservices easily leverage the concurrent interaction capabilities of a database management system via to the service's internal concurrent structure. The thread-pool strategy of Spring as well as the software transaction mechanism transparently extend to database interactions.

Actors face more problems, since they are not concurrent internally. Like with all actions in the actor model, database interactions must execute in a turn-based fashion and are therefore sequential. We have outlined that we can apply the same strategies as for synchronous communication to improve the persistence-efficiency of actors. Either we employ additional concurrency constructs (futures) or we delegate interactions to child actors. Futures always have the potential to violate the isolated turn principle. Delegation forces us to share the database between several actors, violating the exclusive ownership of (shared) state.

Cohesion, Coupling and Independence

Message passing interfaces and strong encapsulation make actors and microservices *cohesive*. Bonér [22] also defines truly isolated components combined through message passing communication as *decoupled* in two dimensions. On the one hand, the components are decoupled in *time*, which is the requisite for concurrent execution. On the other hand, the components are decoupled in *space*, and therefore we can execute them remotely and even move them between locations.

Regarding time, actors facilitate asynchrony intrinsically through the model design. Nevertheless, actor systems tend to offer synchronous primitives on top of the asynchronous style. Programmers receive better abstractions, at the cost of increased coupling. Microservices are free to choose the IPC style, as long as the IPC mechanism is based on message passing rather than shared memory, which further reduces the coupling.

Regarding space, actors are conceptually fully isolated. In practice, ensuring true isolation is difficult, especially for library-based actor implementations. We have discussed the conceptional problems for Akka in detail in Section 5.2.1. In the end, the programmer has to guarantee the isolation by complying to programming conventions. Infringing these conventions introduces or exposes shared mutable state, which is a violation of the actor model. Also, shared mutability increases the coupling to the sharing component, both in time and in space.

Microservices have an inherent advantage regardless of the chosen programming model. The only true paradigm requirement is the avoidance of shared memory sections with other processes. Either the operating system enforces the memory boundaries, or the hardware separation resulting from distribution guarantees spacial decoupling. Besides, their distinct codebases further decouple the microservices. Only a shared library increases their coupling. For example, we facilitate a custom Core library in Echo. Core increases the service's coupling relative to the library. We have implemented all domain-specific functionality, as well as *data transfer objects* (DTO) within the Core. The DTOs are the domain objects we use as messages. Therefore, all microservices are coupled among each other by the DTO classes. Actors suffer from the coupling problem unequally more. The actor runtime intrinsically binds every actor. Different interfaces to an actor system can exist. Akka demonstrates interface diversity for Java and Scala. Yet, the actors cannot escape the coupling to Akka's codebase. Additionally, Echo's actors are also coupled by the DTOs in Core, just like the microservices.

These two notions, high cohesion and low coupling, allow us to reason about the *independence* of task units. Independence is one of the primary concepts of the microservice paradigm. The literature describes independence as a direct consequence of high cohesion and low coupling [33–35,124]. We deem the strong process-based form of independence superior to the notion provided by the actor construct. Actors are passive tasks which react to messages. An actor can only perform actions when the runtime executes the actor. Microservices can show active behavior on their own.

Characteristic	**Akka**	**Spring MSA**
Shared state	Not enforced, obligation of the programmer	Ensured through processes memory boundaries
State mutation	Single-threaded semantics, free of synchronization, threatened by language features	Concurrent access inside the service is possible, but requires synchronization
Persistence/IO	Single-threaded semantics dampens throughput, improved with futures or delegation	Concurrent interaction with outside freely possible
Cohesion	High cohesion through strong encapsulation and message passing	
Coupling	Coupled to common codebase and the runtime	Low coupling through independent codebases
Independence	Passive units, subject to runtime	Active units, subject to OS

Table 1. Comparison of encapsulation-related matters in Akka and a Spring-based MSA

6.1.2 Communication and Message Routing

According to the general concerns of concurrent computation, tasks without mutable shared memory require other kinds of communication links which facilitate message passing instead. Since both the actor- and the microservice model strictly omit shared state, they too require what we call *communication channels* for messaging. These channels transport information from a source to a destination. A *sender* writes data to the channel, and subsequently the *receiver* reads the data from the channel. Independent of the concrete channel technology, message passing is a form of implicit synchronization of the information, since the event of reading a message can intrinsically only occur *after* the message was sent. In contrast, shared state explicitly requires a defined order of accessing the information [12].

We have identified various forms of information flows offered by channel concepts throughout the literature [5,14,30,51,104,118,130,134]. Generally, the flows can be distinguished alongside two dimensions: number of recipients and response coupling. Authors declare varying taxonomies for the resulting combinations. Table 2 provides an overview of the terminology we use in the remainder of this work. Subsequently, Table 3 summarizes the communication capabilities of Akka actors and Spring-based microservices regarding these communication styles at the end of this section.

	One-to-One	**One-to-Many**
Synchronous	Request/response	—
Asynchronous	Notification Request/async. response	Publish/subscribe Publish/async. responses

Table 2. Communication Styles

Asynchronous one-to-one messaging is inherent to the actor model. The message-sending primitive realizes notification style communication. Responses are asynchronous too. Akka provides the `sender()` method within actors. The method produces the `ActorRef` of the originator of the current turn's message. Microservices achieve asynchrony via message queues. Responding depends on the channel protocol. AMQP does not transmit the sender's location. If we must send a response, we need to include the location information into request messages individually.

We can model synchronous one-to-one messaging on top of the actor messaging primitive. Every synchronous communication can be expressed using asynchronous constructs in general, and vice versa [9]. Akka provides linguistic support for request/response messaging through the `ask [?]` method of `ActorRef`. Microservices utilize network mechanism dedicated to the synchronous interaction pattern. In the service context, REST is the most prominent example.

One-to-many communication is neither inherent as a primitive to actors nor microservices, and therefore requires additional effort. Conceptually, we model the communication style by sending a message to each intended recipient in a notification fashion (message broker). Akka has message broker capability in their message router constructs through the `BroadcastRoutingLogic`. Echo implements the routing logic inside a separate actor. In Echo's MSA, the RabbitMQ service does not suffice for one-to-many message distribution. AMQP only supports what the JMS terminology calls *queue* semantics (one-to-one), but not the JMS *topics* (one-to-many). We must employ another messaging technology. *Kafka* [41] is a widely adopted publish/subscribe streaming system with very lightweight message constructs.

One important realization is the difference in messaging interfaces. While Akka provides few but homogenous interfaces (`tell [!]`, `ask [?]`, and `RoutingLogic`s) across all language bindings, the microservice model does not enforce any kind of interface. REST, AMQP and Kafka all have different interfaces. Since all Echo services are based on Spring, at least we express the interaction with each communication mechanism in the same way within each service. But the microservice paradigm is also open for arbitrary technology stacks, and every stack provides its own interface for each mechanism.

Communication Style	Akka	Spring MSA
Request/response	`ask` method of `ActorRef`, delegation pattern	Remote procedure call with REST
Notification	`tell` method of `ActorRef`	Message queue service like RabbitMQ
Request/async response	Request and response with `tell` method of `ActorRef`	Request and response through message queue
Publish/subscribe	Router with `BroadcastRoutingLogic`	Message broker service like Kafka
Publish/async responses	`BroadcastRoutingLogic` for request, responses with `tell` method of `ActorRef`	Request through message broker, responses through message queue

Table 3. Comparison of communication styles and their implementation constructs as we express them in Akka and a Spring-based MSA

6.1.3 Conception of Concurrent Execution

Actors and microservices are both concurrently executed components within their system architectures. However, their execution-modalities are fundamentally different. Hence, both constructs have different *notions* of concurrency. Each notion is a direct result of the underlying concepts.

Continuations, Threads and Processes

Scala's original actors provide two different execution semantics [54]. One of the semantics schedules the actors on threads. These actors are executed in an inversion of control manner [53], similarly to the strategy pursued by the Spring framework. The other of the two semantics is purely event-based and therefore without IoC. Thread-based actors are invoked by their current thread to execute a turn. Upon completion, the actor returns to the calling thread. On the other hand, event-based actors do not have (or need) a dedicated thread and therefore cannot return to one. Instead, they facilitate a much cheaper concept called *continuation-passing style* known from functional programming. A function refrains from returning a computed result and calls a subsequent function instead, the so-called *continuation closure*. Akka extends this approach and defines a single closure for all messages until we replace the behavior. This approach is more effective [52]. We already gave an example of a continuation with `Actor.same` for Akka's type-restricted behaviors. `Dispatcher`s influence the thread assignment strategy for concurrently running the behavior closures [52]. Although it uses threads, we still consider the continuation closure approach as event-based, since a thread can be seen as merely a trajectory in continuation space [126]. Nevertheless, threads are conceptually similar to processes, except that several threads exist within a single process [14]. There, the actor thread-pools live in one or few processes and many actors share the same memory boundaries of their respective process.

Although they execute on top of threads, the isolated turn principle essentially defines actors as single-threaded entities. The combination with asynchronous message passing allows the runtime to concurrently execute these logical single-threaded components. In general, an actor has no notion of concurrency at all. However, we have demonstrated how to introduce additional concurrency constructs into the scope of actors, as long as these additional constructs do not break the actor model semantics. Section 5.2 demonstrated futures with their pitfalls as one option. In combination with the continuation abstraction, Akka actors are powerful and still extremely light-weighted constructs.

On the other hand, microservices are concurrent distributed processes. The model paradigm tolerates a more widespread notion of concurrency than actors do. Internally, services do not have a counterpart to the isolated turn principle. Though a service also receives messages via a public API and reacts to them, the MSA style permits design flexibility regarding internal task unit concurrency. For example, a widespread strategy in the domain of distributed systems is to utilize several threads to perform blocking operations [134]. A microservice applies this strategy to perform blocking operations without blocking the service's entire process, and react to numerous messages simultaneously. Spring's IoC container provides this strategy automatically for requests through its thread-pool.

The drawback of this degree of freedom is the set of issues internal concurrency introduces. In general, accessing state is not a safe operation anymore, if the respective state is read- and writable across threads. We must use synchronization then, e.g. in the form we have demonstrated in the Java case study. When we program with microservices, we are therefore not free from the many hassles of low-level concurrency per se. We have seen this in Echo's MSA variant, where we use transactional memory for synchronization.

However, a service's scope of responsibility limits the service's size. Consequently, the size also limits the internal concurrency considerations of a service relative to the overall system. As a result, linguistic approaches to SOC tend to avoid internal concurrency considerations completely by applying an idea resembling the C processes case study of Section 2.4.2. For example, Jolie offers the `concurrent` primitive as one option for the execution modality of services. The primitive spawns dedicated processes to concurrently execute the service behavior in response to messages [103]. This idea has close resemblance to the cameo delegation pattern of actors. Of course, the design freedom of the MSA style also allows actor-based concurrency internally. The benefits of synchronization-free programming can be harvested by microservices too.

Distribution and Location Transparency

Communication via message passing has one fundamental property: no mutable memory is shared between the communicating components. Conceptually, message passing does not require the components within the same memory space. As a result, it does not matter whether the components run on the same core, different processors or even different host machines [36]. In short, message passing enables distribution.

The actor model builds upon message passing to share state information between actors. Additionally, actors are well isolated from each other. Based on these properties, Agha [4] recast the initial notion of actors in the light of distributed computation. All actor addresses, and therefore also Akka's `ActorRef`s, make the location of the underlying actor transparent. A runtime system that is part of a cluster handles the actual message delivery on the same local node and on remote nodes. Addresses provide a uniform interface free of location considerations.

Microservices intrinsically fall into the domain of distributed systems as well, if the services leverage a network-based communication mechanism. There is no guarantee of a unified channel interface for all services and mechanisms. We have given Unix pipes as an illustrative mechanism. Pipes facilitate the file descriptor interface that Unix promotes. This interface is limited to local node interaction, that is the memory boundaries of a single host OS. Channels operating on the network level are distributed communication mechanisms. Network channels facilitate message passing and provide a uniform interface for remote as well as local communication [130]. Sockets are the most basic example. The socket interface is homogenous regarding whether a socket is on the same local or a remote node. However, sockets still require the *concrete* address [14]. In order to have the recipients' location transparent, the microservice paradigm requires additional effort in the form of discovery mechanisms. We have demonstrated how Echo integrates discovery based on Feign and Eureka directly into the client-side communication interface using Spring's declarative programming style.

Fairness and Resource Consumption

Actors and microservices represent concurrent building blocks of equal status in their respective architectures. It is important to reason about their chances to make even progress, since we have a uniform view on concurrent execution [25]. Our view does not distinguish quasi-simultaneous execution on a single processor, truly parallel execution on multiple CPUs, and distributed execution among several host machines. The property of uniform progress is called *fairness* and is closely related to the liveness of concurrent programs and systems, i.e. to avoid starvation [4,5].

Actors are entities inside an actor runtime. Their scheduling is the responsibility of this runtime system, viz. the system's execution strategy. As passive components, actors have no proactive sense and therefore merely react to events (received messages). The actor system delivers the message and assures that the receiving actor processes the message eventually. Therefore, the runtime must schedule every actor regularly to prevent starvation [84,85]. The major benefit is that actor systems can greatly reduce processing resource consumption. Given an actor's mailbox is empty, the runtime does not have to invoke the actor, since there is no work to do [3].

On the other hand, scheduling is a nonconcern for microservices. Every MSA is a composition of concurrent distributed processes. The MSA implicitly delegates the scheduling to the host operating system(s) – namely their *scheduling policies* [14]. We cannot make specific assumptions on execution rates in general, but require these rates to be positive. This *finite progress assumption* [12] is the foundation of liveness for every microservice. However, there is one drawback to the lack of scheduling concern in MSAs. The architecture cannot influence the resource consumption based on a service's actual demand. An operating system always allocates resources towards every process. Therefore, every task unit of an MSA also stresses its hosts processing power on a regular basis, at least to a small amount. This drain on resources exists even when the services neither have pending requests nor perform active behaviors of their own. In these cases, every process activation is simply a waste of energy and host resources [134].

Table 4 gives an overview of how actors and microservices meet fundamental issues of concurrent programming.

Issue	Akka	Spring MSA
Expression	Actor object, concurrent execution by runtime	Service program execution, concurrent scheduling by OS
Communication	Message passing primitives (e.g. `tell`, `ask`), uniform interface across all actors	Message passing IPC mechanisms, no shared memory, no uniform interface across all services
Synchronization	Implicit among actors due to message passing, single-threaded semantics internally	Implicit among services due to message passing, potentially required internally
Progress	Guaranteed by runtime	Expected from operating system

Table 4. Comparison of Akka actors and Spring-based microservices meeting fundamental issues of concurrent execution

6.1.4 Scalability and Modularity

Their ability to *scale* is arguably one of the most relevant reasons given in the literature to utilize actor- and microservice-based architectures [28,33,35,54,105,121,121,135]. Based on the definition given by Bondi [21], *scalability* is an attribute that influences the performance of networks, systems and processes in general. From the empirical knowledge of industrial Erlang applications it has been suggested that scalability is more important than raw system performance [52]. Many different aspects influence scalability. From a concurrent point of view, every influence hindering parallelism has a negative impact. Examples are synchronization (cf. Java case study in Section 2.4.1) and (temporal) deadlocks. The strong isolation and message passing principles of actors and microservices (isolated turn principle, avoidance of shared memory) reduce coordination and contention cost. Therefore, message passing and isolation have a positive influence on scalability capabilities by limiting safety and liveness issues [22].

Forms of Scalability

Many different forms and classification approaches for scalability exist. Two merit attention here. *Load scalability* refers to a steady performance if the demand or work increases. *Structural scalability* refers to the ability of the topology to change the amount of components, in this case concurrent task units [21].

Two notions of scaling a system are relevant to us. *Vertical scalability*, or simply *scaling up*, refers to an increase of resource utilization, especially multiple cores and memory on a single host. The influencing factors are asynchronous messaging, refrain from blocking, and synchronization. Actors conceptually support these requirements well, as long as the units refrain from blocking inside their turns. The microservice approach meets the requirements also quite well, when services back asynchronous communication mechanisms and refrain from RPCs. In general though, the internal concurrency capability introduces the risk of facilitating potential hindrances of scaling up. Besides, the scheduling efforts of actor runtimes and operating systems aim for an optimal utilization of available resources [14]. *Horizontal scalability*, also *scaling out*, *distance scalability* or *geographical scalability*, refers to the utilization of additional hardware resources (hosts). Distribution capability is the prerequisite. The uniform abstraction of concurrent and distributed execution of actors as well as the process nature of microservices combined with network IPCs provide the foundation to scale out [5,21,33,35,72,134].

One approach to achieve scaling out is the concept of load balancing we have already discussed. Akka actors build upon the same conceptual ideas using the router constructs. They allow us to distribute work in various strategies, e.g. round-robin or broadcast. Round-robin is one example of a load distribution strategy. Server- and client-side load balancing is merely the distinction of a `Router` within the sending actor itself, or inside an intermediate routing actor. Microservices either use dedicated balancing services for server-side balancing, or client-side mechanisms like Ribbon. In one way or another, load balancing is a valuable concept for both actors and microservices, e.g. to avoid overflowing mailboxes and to enable timely responses to requests. Some task units are naturally well suited for load balancing capabilities [26,105], e.g. Echo's API Gateway. The major difference lies in the trade-off that comes with load balancing in each model. As we have mentioned, a `Router` is a supervision-managed set of routees and therefore brings all obligations of actor supervision. Microservices are more loosely coupled and do not know the concept of supervision in general. Hence, load balancing for microservices does not come with additional obligations, except the operation of the additional server-side balancing service.

A conceptual disadvantage of load balancers is that the balancers do not know about the progress of recipients by default. The work gets distributed regardless of the current capacity of the receivers. Load balancers can take work load metrics into account of course, as does for example the `SmallestMailboxRoutingLogic` of Akka. This routing logic aggregates the mailbox capacities of all routees first. With this information, the logic then forwards the message to the actor with the smallest mailbox. This capacity aggregation *before* the message dispatch increases the overall processing time [11,118].

Another kind of load balancers are message queue channels. The receiving components actively pull messages from the queue when they have computation capacity. Therefore, the tasks distribute load among themselves based on their demand [30,33].

Dynamic Reconfiguration

Dynamic reconfiguration relates to a change in a system's topology at runtime. We can add, remove or relocate task units divergent from the static initialization configuration [7]. Actors support dynamic reconfiguration inherently through the primitive that allows actors to spawn new actors. We already demonstrated this ad hoc instantiation with the dynamically created response handlers that the Searcher spawns for the delegation-based synchronous communication. The loose coupling of microservices also provides opportunities for changing topologies. Discovery registries and asynchronous messages through intermediate message queues provide the foundation. Reconfiguration greatly effects scalability, because it increases the optimal utilization of hardware resources [84].

A general prerequisite for dynamic reconfiguration is location transparency. Actors as well as microservices have sophisticated solutions. Two additional properties, *mobility* and *elasticity*, become possible as a result. Mobility refers to the relocation of components between nodes [5,84,134]. Elasticity is a form of scalability summarizing the ability of a system to scale the number of components dynamically depending on the current demand. Hence, elasticity improves the load scalability while minimizing the resource consumption. We can scale actors and microservices in a non-uniform way due to their component properties. Each individual component type allows us to incarnate many instances without the requirement to duplicate the residual component types as well (in contrast to classic monolithic applications) [33,35].

Echo does not provide mobility nor elasticity. However, the general approach is conceptually identical for actors and microservices, therefore we present a short outline. In principle, we are able to create and terminate stateless task units easily on demand (elasticity). Since these units have no state that requires relocation, this reconfiguration also doubles as mobility. Akka does not provide strong mobility that requires the migration of state. Other runtimes like Orleans do provide state migration [20]. Stateful Stores are a bigger concern. In general, when we (re-)created a stateful unit, it is not safe to assume that its persistent state (database, reverse index) is up to date. Therefore, we must update the state with regard to all outstanding modifications. *Event sourcing* is a convenient concept which persist all modifying commands to Stores in a so-called *event log* [19,107]. A new or reactivated Store requests the history of modifications prior to the unit's existence. Message brokers have the potential to double as event logs. Kafka offers optional persistence support for messages. Akka's *Persistence* module is also convenient to introduce persistence of routed messages. In both cases, it is within the obligation of the Store units to persist a counter or reference to the last received event separately. This reference is needed to determine the required partial history.

In general, we deem the concerns related to dynamic reconfiguration more easily met with actors. Spawning new units is a core concept of the basic model primitives. Microservices themselves have no general notion of other services beyond interaction. We need an additional layer to manage changing topologies. *Cloud management frameworks* are a category of tools that seem convenient.

Table 5 summarizes the capabilities for each scalability variant provided through Akka and Spring.

Scalability Form	**Akka**	**Spring MSA**
Vertical scalability	Very efficient resource utilization if no blocking inside actor	Limited by the synchronization of the internal service concurrency (STM)
Horizontal scalability	Akka cluster	Requires network IPC
Load scalability	Server-side load balancing with routing logic	Client-side load balancing with Ribbon, message queues
Structural scalability	Location transparency inherent in actor addresses	Requires discovery mechanism for location transparency
Dynamic reconfiguration	Inherent through basic model primitive	Requires service registry for integration
Mobility	Weak mobility (no relocation of persistent state of Stores)	
Elasticity	Requires resource control of cluster (see [101]), not supported by Echo	Requires appropriate cloud management framework, not supported by Echo

Table 5. Capability matrix of scalability variants and their support by Akka actors and Spring-based microservices

Extensibility and Technology Diversity

Another result of the reconfiguration capability is that actor and microservice architectures are also open for extension. In contrast to dynamic reconfiguration, where we add or remove instances of existing task units at runtime, we use *extensibility* to refer to the introduction of either new versions of existing components (update) or new components entirely (addition). Extensibility in general benefits from high cohesion and low coupling of the components. There are two different kinds of extensibility [4,14]:

- static, where we adapt the architecture's code, recompile and then redeploy
- dynamic, where we add a new component to the architecture at runtime

The independent deployment capability of each single service engine allows us to simply add new components at runtime. The reconfiguration does not impact existing services. New services simply consume the existing services. We then gradually update old services to let them integrate with new components [105].

Actors face more challenges. Every actor requires an actor system to exist within. Erlang and its runtime system were tailored to support actors. As a result, the *BEAM* virtual machine supports code loading and replacement for live upgrades [107,137]. The JVM does not support similar features. Therefore, Akka requires the restart of the system to introduce new kinds of actors. The same program structure defines all actors (monolith). Subsequently, the actors compile into a single monolithic executable, which limits Akka actors to static extensibility. In a clustered setup however, we do not need to introduce a new component into all cluster nodes at once. Therefore, the cluster is able to retain its uptime, while we upgrade and redeploy the individual cluster nodes. Besides differences in the VM optimization, Erlang's upgrade strategy is conceptually similar. The BEAM does not support the replacement of single actors, but merely of entire code modules [107].

Another interesting concern is the technology limitations regarding the conception of new components. An actor is bound to its runtime, which is free to provide interface bindings for numerous programming languages, e.g. as Akka does for Java and Scala. We are able to use the Java binding from other JVM languages as well, as [132] demonstrates for Groovy and JRuby. However, we cannot overcome the JVM as the target platform. Interoperability with Akka.NET would broaden our possibilities, but to our knowledge interoperability is not available. The strong memory boundaries and open communication interfaces of microservices provide a whole different level of flexibility. We can conceive a service using every programming language and technology we desire, as long as the tools are able to interact with the open communication channels.

Echo's services facilitate Java and the Spring framework for all microservice components. However, we could have also written every service with a different technology stack. Even now, we can replace existing services by new versions using different technologies.

Characteristic	**Akka**	**Spring MSA**
Extensibility	Static (phased restarts)	Dynamic
Technological Diversity	Compatible to the Akka interface (JVM technology), no compatible runtimes available today	Open for arbitrary technology stacks due to open interfaces and communication channels

Table 6. Comparison of modularity capabilities of Akka actors and Spring-based microservices

6.1.5 Integrating Actors and Microservices

So far, our evaluation has shown that we can express the same capabilities with both actors and microservices.

Both constructs isolate state, communicate through a wide range of interaction styles, and their execution modality enables parallelization and distribution in a transparent way. The major benefit of all these capabilities is a high degree of modularity as well as great flexibility for scaling. Each model brings its own set of trade-offs, and we as programmers must accept one of these sets in order to leverage the capabilities.

As a final capability evaluation, we want to reason whether all these properties enable actors and microservices to integrate as equal concurrent task units. We have already seen that the combination of concurrency models is a common and important practice. Actors often incorporate futures, and microservices use and mix arbitrary concurrency constructs internally. We will entertain this integration thought first with some theoretical considerations and then discuss the practical approach regarding Akka and Spring.

As we have pointed out, there are two kinds of correctness properties: safety properties and liveness properties. The combination of two concurrency models must not weaken the safety and liveness properties. There are also two approaches to assess the correctness of a computation: *testing* and *verification*. From a theoretical point of view, we can violate every safety property by a finite execution. Hence, we can test for safety at runtime. On the other hand, liveness properties cannot be violated by some finite execution in general. Even if an arbitrary finite execution causes a violation, there is some continuation of the same execution for which the property still holds eventually [128]. Thus, we cannot test for liveness at runtime. This theoretical limitation raises the need for other concepts to either guarantee or at least inspect the liveness of tasks. We require strict theoretical frameworks to apply formal verification techniques [127]. For actors, the theoretical actor model provides this framework. However, we have seen that rather pragmatic rules define the microservice model in comparison. We yet lack a formal framework.

Actor Model and Process Calculi

Among the theories of formulating concurrent computation we find a family called *process calculi* or *process algebras*. These calculi define formal models composed of so-called *processes*[23] which communicate within the laws and conditions laid out by their theory. Baeten [15] defines *process* as any kind of *behavior* of a *discrete event system*, such that it is observable through discrete actions. These actions include interactions with other discrete event systems. The other systems then react to these interactions. Therefore, Baeten terms all interacting systems as *reactive systems*, which are the base for parallel and distributed computing. As a result, one approach towards concurrency theory is the path of process algebras.

Some process calculi gained considerable prominence. Examples are Milner's *Calculus of Communicating Systems* (CCS) [98] that was the initial work in this domain. Hoare's *Communicating Sequential Processes* (CSP) [60] was the first to introduce message passing instead of global variables for process communication. For our considerations, the *π-calculus* [99], also by Milner, merits special attention. The π-calculus has a notion of process networks, including mobility and dynamic reconfiguration [15,102].

[23]These are the processes of concurrency theory we have mentioned before. We must not confuse them with operating system processes.

Much work was done on the field of process algebras, since they allow us to express a theoretical basis for arbitrary domains and requirements. In fact, more practical approaches to express interacting processes frequently have their foundation in a process calculus. Some calculi also suit well for microservices. The Jolie language for example is based on a calculus dedicated to service-orientated computing called *SOCK* (**S**ervice **O**riented **C**omputing **K**ernel) [50]. In turn, SOCK is inspired by CCS and the π-calculus. This relation of process algebra and microservices is especially interesting, since the actor model and process calculi share a long history. Hewitt as well as Milner published their initial works on the actor model and CCS in 1973. Since then, these two approaches mutually influenced and inspired the scientific development of each other. Fitting examples are the *Asynchronous Sequential Processes* (ASP), which bear close resemblance to active objects, but with a more coarse granularity [87].

Scholars have long tried to formulate a theoretical link between actors and various calculi, with mixed and mostly limited success. To our knowledge, to most promising approaches in the literature so far merely succeeded at describing interoperability between actors and some calculi that show a strong similarity. For example, Montanari & Talcott [102] demonstrate the cooperation of actors and agents of the π-calculus.

As Agha *et al.* [9] discuss in an extensive work, a true equivalence theory among a formal *actor calculus* and some process calculus requires the formulation of a *simulation relation* among the primitives. This was done among different process calculi but is yet unfound regarding actors. Agha *et al.* state as the foundational challenges:

> *"Three points of contrast between the basic actor model and process calculi are: the choice of communication model, the choice of communicable values, and the issue of fairness."*

Note that we have discussed these issues as concerns in varying degrees throughout this thesis. Eventually, Agha *et al.* argue that instead of trying to find an analogy between an actor- and a π-calculus, it is expedient to engage in the definition of high-level semantics for programming languages. Then, we are able to reason about program equivalence of actors- and π-programs.

Combining Akka Actors and Spring Microservices

Since we did not implement our Echo microservices in a SOP language like Jolie, we did not provide them the formal foundation of a process calculus. We would not be able to reason about the correctness properties after integration anyway, because there is no known theoretical link between the actor model and a SOC calculus yet. Nevertheless, we have seen that scholars are still motivated to describe interoperability between actors and calculus-based processes. Therefore, we think about the approach to integrate our Akka actors and Spring-based microservices. From a theoretical point of view, two concurrent units require a shared communication channel in order to interact [9].

The microservice principles allow the services a rather high degree of freedom regarding their communication. REST-based state transfer via HTTP is a popular choice. We have demonstrated that AMQP is also a viable channel. Actors, on the other hand, are more restricted, since all actors must receive the *message* constructs in a uniform way. By implication, this restriction means that we can still use different communication channels as long as the channels comply to a homogenous message receiving abstraction.

Akka HTTP [68] is part of the Akka library collection. It provides a full client- and server-side HTTP stack on top of the basic actor construct. This way, actors receive HTTP requests like ordinary messages, just as if the messages were sent by other actors:

```
override def receive = {
  case SomeMessage(_)                          => // handle message
  case HttpResponse(status, headers, entity, _) => // process response
}
```

A unique address identifies every actor. This address is specific to a concrete actor system and not conform to a URI (**U**niform **R**esource **I**dentifier) as is required by HTTP in general. We must utilize an integration layer between a HTTP-addressable unit and the actual actor [68].

Though this style of receiving messages is in principle conform to the actor abstraction, there are certain limits. By implication, HTTP endpoints also do not comply to actor addresses in general. The standard *send* mechanism through `tell` and `ask` therefore do not qualify to dispatch HTTP requests. We require an alternative interface. The example below uses `pipeTo` to have the integration layer reroute the response as an `HttpResponse` message to the sending actor (delegation):

```
Http().singleRequest(HttpRequest(URI(s"http://example.com")
  .withQuery(s"name", s"value"))
  .pipeTo(self)
```

We see, not all the basic actor primitives suit for usage outside the actor construct. However, well introduced abstractions like Akka HTTP allow actors to provide their *actor behavior* to the outside in a microservice-like *service behavior* fashion. Communication between actors and microservices is possible in principle. Hence, we can construct systems which distribute tasks between actors and microservices.

The general integration idea is not novel. We have mentioned the theoretical foundation from [102] for incorporating actors and agents of the π-calculus. They also use so-called *actor-π coordinators* to translate between communication channels. These coordinators are similar to the integration layer of Akka HTTP. However, interoperability is merely a concern of static configuration, because processes have only a static notion of interconnection topology in general. This static notion violates the dynamic reconfiguration inherent to actors [7,10]. In order to overcome this limitation, microservice architectures utilize bridge technologies like service discovery. Hence, we require actors to integrate into this discovery mechanism, effectively rendering each actor into a mere microservice. This unified approach is not quite common yet, but technologies start to emerge which build on this idea. An example is *Lagom* [73], a reactive microservice system framework built on top of Akka.

Main findings

- Process calculi provide the theoretical frameworks for formal microservice specifications and service-oriented computing languages.
- The theoretical link between actors and process calculi is yet an open scientific question.
- Actors and microservices can integrate without a formal basis. However, the correctness properties of the models are then not guaranteed.

6.1.6 Software Artifact Analysis

Sections 5.2 and 5.3 covered the solution strategies when we program with actors and microservices. The previous sections of this chapter compared the resulting capabilities. Now we give attention to the effects that the programming models have on the resulting software artifacts when we express these capabilities. We concern ourselves with three metrics: the *lines of code* (LoC) it takes to express the respective functionality, the *size* of the resulting artifacts, and the *startup time* of each artifact.

All Akka actors are compiled into a single monolithic application. Therefore, the respective Akka metrics refer to the resulting monolith. The Spring-based microservices are independent programs, so we give the metrics for each service engine separately. The LoC give us an indication of the programming effort is takes to implement a given functionality. We count the LoC using the CLOC [31] tool. Since Spring Boot relies heavily on configuration to adapt it's default behavior, we count the content of configuration files as source code. We measure size in terms of the bytes of each JAR (**J**ava **Ar**chive) file. There are several ways to package a JAR. Two kinds are relevant to us. We define a *skinny JAR* (sJAR) as the archive that contains merely the bytecode and direct resources (e.g. property files) of a program's source code. Subsequently, we use *fat JAR* (fJAR) for an archive that contains the data of the skinny JAR version, together with its direct dependencies (e.g. other libraries in their skinny JAR version), and the deployment information that a standard Java runtime environment requires to execute the application. Fat JARs are therefore executable artifacts. The startup time is the time from process incarnation until the application is fully operational.

Table 7 gives the metrics for our Echo artifact implementations. As a reminder, the Core artifact is the library that implements the domain-specific logic. Hence, Core's skinny JAR is part of all fat JARs of the microservice engines as well as the monolithic Akka backend. The Web application is the frontend client that connects to both Echo backend implementations. We give the LoC metric of the Web application merely as a reference relative to the other artifacts.

Artifact	LoC	sJAR (KB)	fJAR (KB)	Startup (sec)
Akka backend	4487	1004.3	76 775.1	5.5
CatalogStore (MS)	1838	56.1	89 225.8	14.6
IndexStore (MS)	724	23.8	83 518.2	8.8
Searcher (MS)	656	22.2	81 754.4	8.1
Web Crawler (MS)	716	23.5	83 517.9	9.2
Parser (MS)	703	24.2	83 519.1	8.6
Registry (MS)	334	9.9	90 699.7	9.4
Gateway (MS)	889	30.5	83 655.1	9.7
Updater (MS)	693	23.9	83 518.3	8.7
Core (library)	5203	323.1	—	—
Web (frontend)	3144	—	—	—

Table 7. Lines of code, bytecode sizes, and startup times of software artifacts

The Akka backend engine has of course by far the most lines of code. A monolith implements the whole system within a single artifact, while each microservice merely implements a part of the overall system. Note that the Core library implements the DTO classes we use to send asynchronous messages via AMQP. The microservices do not implement these DTOs themselves within their codebases. The resulting overall LoC sum of all microservices is 6553. This makes the microservice codebase about 46 % larger than the actor codebase. We see, although Spring provides very expressive declarative programming APIs, the Akka interface is still less verbose. However, the more compact syntax of Scala compared to Java also contributes to the difference. Even more interestingly, each microservice engine takes considerably more time to startup than the entire Akka backend engine. The inversion of control model that powers the declarative programming style of Spring adds considerable runtime overhead.

The sum of skinny JAR sizes of all microservice codebases is 241.1 KB. This size is considerably less compared to the 1004.3 KB of the skinny Akka backend. We have analyzed the content of the respective skinny archive files and made the following observations that affect the difference in bytecode size:

- The actor codebase is written in Scala, and we make heavy use of `case class` constructs. Scala's `case class`es are very compact class definitions that are mostly written in a single line of code. Though these class definitions are compact, they still compile to dedicated `.class` files. Instantiated objects are immutable, hence they suit very well for the objects we exchange as messages in actor programming. Messages in the actor model are not only for transferring data, but also to transmit commands. Thus, we use very fine-grained message types, and subsequently a lot of `case class` definitions. Few lines of code for these definitions let the compiler produce lots of separate `.class` files. In contrast, all the DTO classes for the messages between microservices are part of the shared Core library. The bytecode

sizes of these classes consequently do not participate to the sJAR size of each microservice's artifact.
- Scala is primarily a functional programming language. Hence, we make frequent use of so-called *anonymous functions*. Since Scala targets the JVM, the compiler evaluates these anonymous functions as instance creation expressions of the `Function` class. Subsequently, every anonymous function compiles into a dedicated `.class` file too [109].
- Spring provides a lot of utility functionality transparently, for example DTO class to JSON marshalling. Akka HTTP requires us to use the *Spray* [74] library to define custom JSON serializer classes manually. These serializers compile to relatively significant bytecode sizes (e.g. merely 31 LoC produce about 200 KB[24]).

The fat JAR sizes show the impact of the declarative programming style of Spring. The price for the relatively low LoC of each service (and subsequently the reduced effort to write many distinct programs) is that the compiler adds lots of dependencies into the fJAR executables. These dependencies come into play at runtime to realize the declared functionality. As a result, every microservice engine is larger (in bytecode size) and upon incarnation also considerably slower than the entire actor engine.

Main findings

- Programming actors demands less program code than microservices. Even compact programming styles cannot compensate the overhead in LoC from several codebases.
- Spring produces huge executable artifacts. Their bytecode size and the overhead at startup is the price for the reduced programming effort.
- Microservice architectures are drastically larger in terms of executable component sizes compared to a monolithic actor architecture.

6.2 Efficiency and Benchmark

We have compared the expressiveness and conceptual capabilities of actors and microservices, and demonstrated that both are able to meet similar concerns. Now we are interested in the efficiency we leverage from each programming model. The Echo system implementations provide us with the foundation for a benchmark of the two models.

6.2.1 Performance Metrics

A benchmark requires measurable and comparable metrics. As we have already mentioned, information retrieval traditionally uses precision and recall metrics to evaluate search engines. However, precision and recall assess the effectiveness of the retrieval

[24]A JAR file is essentially a compressed archive file. The direct size of a `.class` file does not contribute at a ratio of 1:1 to the sJAR size.

techniques. IR effectiveness is not within the scope of this thesis. We require metrics which reflect efficiency. These metrics must be applicable to actors, microservices, and our scenario.

Savina is a benchmark suit specifically designed to evaluate actor libraries [62]. Profiling studies used Savina to gather detailed metrics for Akka [119,120]. However, the benchmark suit as well as the profiling tools are actor model and Akka specific. Hence, the metrics provided by the profiling are tailored to the actor model.

Recent works point out that there is still a lack of microservice benchmarks [145]. Due to the technological diversity, there is also no general microservice profiling tool available. Hence, the literature does not establish widely agreed upon metrics yet. We must revert to model unspecific metrics. Besides a lack of common metrics established between actors and microservices, there is also no general simulation approach for MSAs as of yet [48]. Additionally, we have to design a custom experiment too.

Lillis *et al.* [91] and Pedzai & Suleman [113] each use techniques we discussed in this work to improve the performance of search engines. Examples are synchronous/asynchronous and one-to-one/one-to-many communication styles, message brokers, and lifecycle management (cf. Akka runtime, Spring IoC). Among other things, they evaluate the performance by measuring the time it takes a system to index a given number of documents. The retrieval subsystem's performance is the time it takes to process a given number of queries. We take this experiment design and use it to assess the efficiency of our actor and microservice implementations.

6.2.2 Simulation Workloads

Echo has two essential subsystems: the indexing subsystem and the retrieval subsystem. Since search requests to the retrieval subsystem do not affect the indexing subsystem (and vice versa), we can evaluate both subsystems separately [91]. Each subsystem requires a different kind of input data. For benchmarking a subsystem, we need to simulate a workload scenario with appropriate input data. Although we are not interested in evaluating the effectiveness, we can still look to information retrieval for workload data. IR uses standardized dataset collections for evaluations. These dataset collections usually consist of three parts [95]:

Part 1: Set of documents
Part 2: Set of queries
Part 3: Relevance assessment between queries and documents

We are interested in Part 1 as the input for the indexing subsystem and in Part 2 as the input for the retrieval subsystem. Effectiveness evaluation uses Part 3, hence we do not require this data. To our knowledge, there is only one available dataset collection provided by Spina *et al.* [131] for the podcast domain. This collection contains audio files, manual and automatic audio transcripts, queries, and relevance assessments. Since the collection misses RSS feed data, the input documents are inadequate for the Echo implementations.

Therefore, we must design our own dataset collection. In general, we face two problems: selecting documents (RSS feeds) and determining suitable queries. The execution time of operations, e.g. parsing a feed, affects the performance. XML parsing time depends on the document size. The literature therefore usually assesses execution time with respect to the input size. Real world RSS feeds have arbitrary data and we have no control over the input size per feed. Since we do not mind the actual information within either the feeds, the queries, nor the quality of search results, we can simply create custom feeds and queries using placeholder text. Appendix A shows the feed structure we use for evaluation. We have analyzed 500 arbitrary feeds from the Fyyd Podcast Directory [17] and found that the average feed has 70 episodes. To make the simulation workloads more realistic, the test feeds also have 70 elements.

6.2.3 Experiment Setup

We measure all evaluation results on a multicore platform (Intel Kaby Lake Core i5 3.1 GHz with 2 cores, 64 MB of eDRAM, 4 MB shared level 3 cache, 16 GB of 2133 MHz LPDDR3 SDRAM, Java HotSpot 64-bit server VM build 25.172-b11, macOS 10.13.6, 1 TB SDD flash storage, APFS file system). The actor implementation uses Akka version 2.5.11 and the microservices build on Spring Boot version 1.5.10.RELEASE. All code is compiled with Java compiler version 1.8.0 update 172 and Scala compiler version 2.12.6. The components with persistent states are deployed with an H2 [106] database engine version 1.4.196 (in-memory persistence mode) and Lucene [39] version 7.2 respectively.

All measurements we report are for cold starts of each system. We restart all involved JVMs for each simulation. The indexing experiments start on an empty catalog/index. The retrieval experiments start on a filled index. Crawlers load the benchmark feeds from the local file system, and are therefore not subject to network induced latencies.

To eliminate a threat to the validity of the benchmark (discussed in Section 6.2.6 below), each programming model's CatalogStore implementation uses the Spring Data JPA library for database interaction. Usually, Spring Data JPA expects a Spring IoC container to handle concurrent connections and transaction management. To handle concurrent database interaction directly via actors, the Akka implementation uses Spring Data JPA without an IoC container. The respective CatalogStore therefore has to manage transactions manually.

We assign each architecture component with a fixed number of threads to ensure that both implementation variants have the same resources for concurrent execution available to them. The Akka components use a `Dispatcher` backed by a `ThreadPoolExecutor` (in contrast to the default `ForkJoinExecutor` with a dynamic pool size). The Spring IoC containers of the microservices are configured to use a `ThreadPoolTaskExecutor`, where the `corePoolSize` is equal to the `maxPoolSize`. We use 16 threads per component.

This way, the benchmark results indicate which programming model better utilizes the available thread resources.

6.2.4 Benchmark Results

To reduce the effects of outliers, we have conducted each experiment 3 times. Each data point in the following results is the mean of the three measured values. Note that all experiments are for static system configurations. Therefore, the results do not reflect any forms of structural scalability (mobility, elasticity). Also, due to the single multicore host, we cannot draw conclusion to the horizontal scalability behavior of the architectures.

Experiment 1: Indexing Subsystem

Figure 9 shows the benchmark results of the indexing subsystems for the overall time it takes the implementations to process the workload with respect to the input size. The Akka implementation shows better performance for all input sizes.

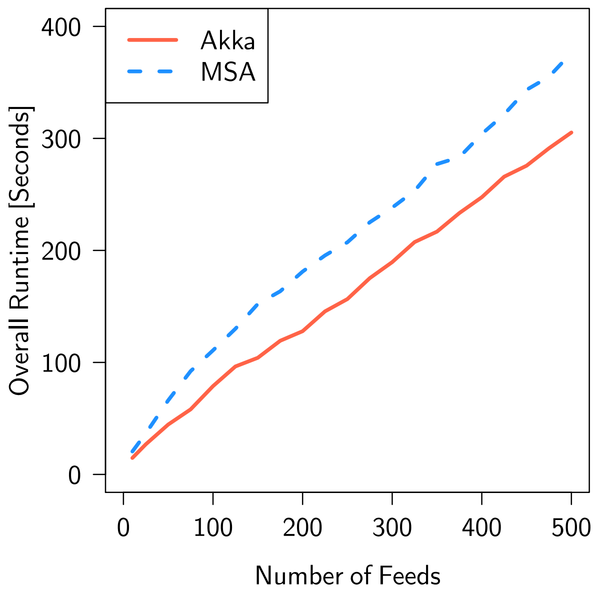

Figure 9. Benchmark results for the overall processing time of the indexing subsystem

The figure also describes the load scalability behavior of the systems. With an increasing load for the indexing subsystem, both implementations scale uniformly, which is the desired behavior. This uniform behavior is the result of good vertical scalability, since the architectures are able to uniformly leverage the resources of the single multicore host used for the benchmark.

In Section 6.1.3 we discussed the issue of fairness and the implications on resource consumption. Figure 10 shows the mean memory resource consumption of the Echo implementations in the indexing phase. We measured the memory usage (heap memory + non-heap memory) using the `java.lang.management.MemoryMXBean` that every JVM provides. The results illustrate how every microservice consumes separate system resources, even those who do not perform any work in the indexing phase (Gateway, Searcher). The Akka backend that implements the entire Echo system consumes only slightly more memory resources as a single Spring-based microservice. The CatalogStore MS in an exception. The author suspects that the reason for the CatalogStore service's memory demand is that the IoC container of this service has to extend the STM to the database. The entire MSA has a considerably higher memory requirement than the entire actor-based system. The figure also illustrates the impact of the JVM as a microservice platform. The JVM is a relatively heavy-weight VM. We must deploy every process incarnation of a Java-based microservice in its own separate VM, which poses a considerable impact on the system resources. The operating system always allocates resources towards every process on a regular basis. In contrast, Akka actors, besides being more lightweight constructs in general, only get scheduled and thus only consume resources when they have messages in their mailbox.

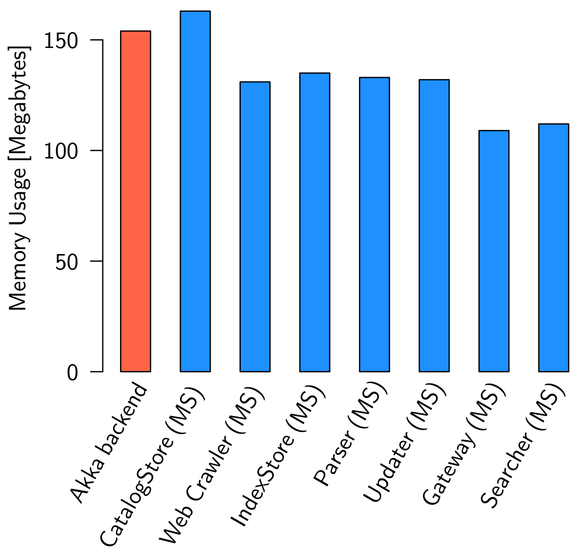

Figure 10. Memory consumption of the executable artifact VMs in the indexing phase

Experiment 2: Retrieval Subsystem

Figure 11 shows the results of the experiment to assess the retrieval subsystem's performance. It clearly indicates that the request/asynchronous response style of Akka actors is superior to the synchronous REST-based communication of the microservices. Since the Akka implementation processes heavy loads of requests considerably faster, this subsystem is more available and thus has better liveness. The figure also describes the load scalability behavior of the retrieval subsystems. Both implementations scale uniformly as desired. Again, we trace this result back to the fact that the implementations leverage the available system resources of the single multicore host well (vertical scalability). However, the microservice variant shows considerably lower overall efficiency. Section 6.1.4 gave the synchronous nature of RPC as a major factor limiting the scalability of an MSA. The results of our benchmark support this claim. The request/asynchronous response style of Akka is clearly more efficient than REST-based communication.

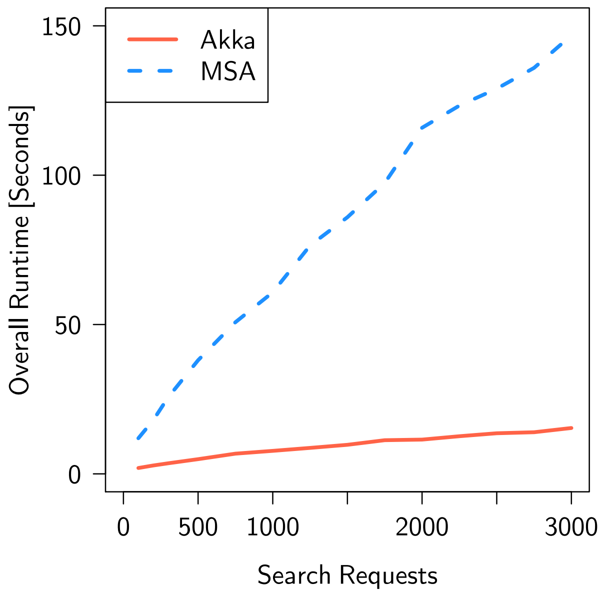

Figure 11. Benchmark results of the overall processing time for the retrieval subsystem

In Section 5.2.3 we have actually discussed two variants to model request/asynchronous response communication with Akka. One variant is based on futures, the other on custom child actors and delegation. The author expected that the delegation approach would show better performance, since a future always stresses the thread-pool, while the actor runtime must only schedule a delegation child once the response is available in the mailbox. The Akka result in Figure 11 therefore shows the performance when the retrieval subsystem uses delegation. To evaluate the future- and delegation-based modelling of (semi-)synchronous communication, we have conducted the retrieval experiment also with futures. Figure 12 illustrates the future results in contrast to the delegation results.

We see, neither of the two request/async. response styles show a considerable performance advantage. Recall that the future and delegation strategies are also applicable for database interaction and IO. The results of Figure 12 suggest that each strategy is also equally efficient for database interaction. Therefore, we assume that the actor

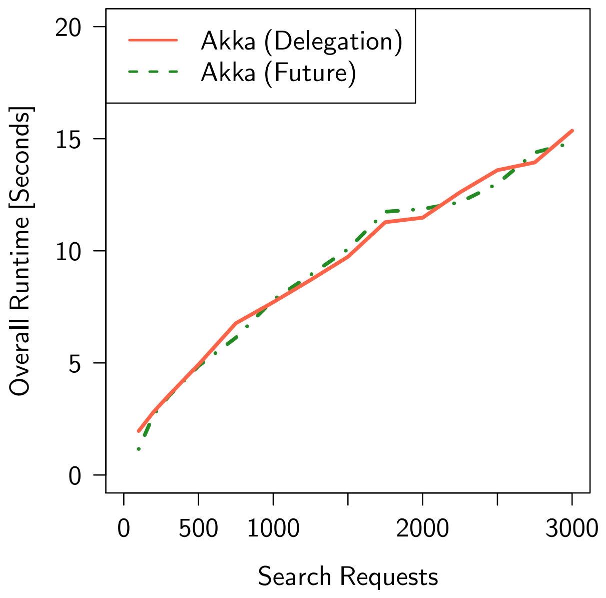

Figure 12. Comparison of the benchmark results for the retrieval subsystem using either delegation or futures for request/response communication in the Akka-based implementation

implementation does not suffer from a negative impact because it has to use the Spring Data JPA library for database access in these benchmark experiments.

6.2.5 Relevance of the Benchmark

To the authors knowledge, there exists no benchmark comparing Akka actors and Spring-based microservices yet. We were only able to find one project on GitHub[25] which benchmarks popular microservices frameworks. The benchmark results include Spring Boot with the Undertow webserver and Akka HTTP. The experiment setup is rather simple. The frameworks merely serve "*Hello World!*" on a REST interface as the workload, which does not resemble a real-world workload scenario.

The literature reports especially a lack of different interaction modes in microservice architecture benchmarks [145]. Most available benchmarks merely focus on one interaction mode, while this literature also reports that MSA-related problems originate from asynchronous and mixed communication setups. Echo's subsystems engage this circumstance, since we have modeled the indexing subsystem in an asynchronous fashion, and the retrieval subsystem in a synchronous fashion. Our experiences do not reflect problems with the asynchronous style. In contrast, we have found that the synchronous style is considerably less performant than the asynchronous style.

[25]https://github.com/networknt/microservices-framework-benchmark

Interestingly, Bonér, the creator of Akka, advocates in his recent works [22,23] for what he calls *reactive microservices*. Essentially, a reactive microservice is a service that orients itself on the actor principles, especially asynchronous messaging and the lack of global consistency (cf. eventual consistency). Bonér argues that reactive microservices are more performant than tightly-coupled synchronous services facilitating global consistency (e.g. via a transactions mechanism). However, he provides no benchmark results to back his claim. The subsystems of our microservice implementation reflect a reactive microservice style (indexing pipeline) and a more "traditional" synchronous style (retrieval pipeline). The Akka implementation provides the reference to a purely reactive (asynchronous) system. Our benchmark results support Bonér's argumentation that the reactive style is more performant.

6.2.6 Threats to Validity

Like all experiments, we are also subject to some factors threatening the validity of our results.

External Threats to Validity

The external threats concern how much our results are generalizable. The domain-specific actions of our scenario have an influence on the performance. Examples are the HTTP retrieval of RSS feeds, XML parsing and database IO. These actions dampen e.g. the throughput. The utilized underlying technologies (HTTP library, XML parser, database system) influence the performance of these actions. This threatens the comparability of the benchmark metrics across systems, if the implementations apply different technologies. We mitigated this threat by founding all task units on the JVM. We provide all components with the same Core library, which implements the domain-specific functionality. The CatalogStores access the same kind of database system, and both implementations use the Spring Data JPA library for database interaction. Platform- and domain-specific effects are therefore uniform across the system implementations. Nevertheless, due to the domain influence, no general statement about the performance of Akka or Spring-based services is possible. Other metrics are not even measurable in the scenario. Examples are the creation time and maximum process support as suggested in [136]. The static configuration of our scenario does not intend elasticity, i.e. a dynamic creation of task units. Hence these metrics require an experiment outside the bounds of our scenario.

Additionally, we conducted the experiments merely on a single multicore machine. The experiments therefore only incorporate the effects of vertical scalability we leverage from general concurrency and parallelism on one single machine. Also, the multicore machine has a very small number of cores. The test setup does not consider the horizontal scalability effects of distribution-induced concurrency.

To eliminate the threat of selection bias, we did not use real world RSS feeds for simulation. Instead, we used test data with uniform size and feed structure.

Internal Threats to Validity

The internal threats to the validity of this benchmark concern the accuracy of our data collection tool. Since we did not find a tool that is sufficient to collect the data for Akka actors and Spring-based microservices, we developed a custom benchmark framework. We implemented this toolkit to the best of our ability. But we have to assume that the toolkit's efficiency is not state of the art. This threat is mitigated since both systems are subject to the same potential inefficiency, which does not distort the relative comparison of metrics.

An additional threat results from the fixed amount of threads we provide for each component in the benchmark. Actor-based components can fully leverage all these threads, e.g. for child actors or futures. We have discussed in Section 5.3.2 that the microservices use a different `TaskExecutor` for asynchronous tasks than the standard thread-pool of Spring's IoC container. To ensure the overall thread limit for each component, we have to split up the threads between the two thread-pools. There is the threat that a service's implementation does not distribute the computational load equally between the two thread-pools. We did not measure the thread-partitioning required for ideal thread utilization for each microservice. Instead, we distributed the available threads evenly between both pools for all services. This threat is only mitigated for services which do not apply asynchrony internally. Then there is only the standard `TaskExecutor` with the full amount of threads in place.

7 Conclusion

Most papers in computer science describe how their author learned what someone else already knew.
— Peter Landin

In this chapter, we draw our conclusive view. We give answers to the research questions, summarize the contributions, and finally motivate some future work.

7.1 Research Questions Revisited

In the beginning of this thesis we asked a set of research questions. It is now time to revisit and answer these questions.

RQ1: Why do actors and microservices qualify for programming concurrency?

Actors and microservices encapsulate their state exclusively and all their communication solely facilitates message passing semantics. These properties make the task units highly cohesive and provide a temporal and spacial decoupling. The resulting independence is the foundation that enables an actor runtime or an operating system to execute the task units in a concurrent fashion implicitly.

RQ2: How do the actor and the microservice model facilitate concurrent execution?

The execution modality of actors and microservices already introduces concurrency among the task units. Both models can also utilize additional sources of concurrency. The actor model has a long tradition of using futures, e.g. for (semi-)synchronous communication abstractions. In general, actors can be combined with every other concurrency model, as long as the combination does not introduce new safety or liveness issues.

Microservices are free to employ internal concurrency using every model available to their technology stack. Unlike actors, microservices are therefore not free of internal synchronization in general. Our scenario implementation leverages the implicit scheduling of requests on thread-pools through an inversion of control container. Software transactional memory controls the required synchronization.

RQ3: What are the expressive capabilities of actors and microservices regarding concurrent programming concerns?

We have shown that both models apply different implementation strategies, but in general achieve the same capabilities for concerns like parallelization/distribution, communication, isolation, location transparency, persistence/IO, and scalability. The two implementation strategies come with different trade-offs for each model.

In general, we expect larger executable artifacts and more resource consumption from the microservice approach, since the model produces independent programs. The author initially expected that the microservice style would also show a significantly higher programming effort, e.g. in lines of code. The suspected reason was the need to maintain a dedicated codebase for each service. Indeed, our overall microservice codebase is about 46 % larger compared to the actor codebase, although the declarative programming style we applied for the microservices already reduced our programming effort significantly. The price for this reduction is the resulting size of the executable artifacts.

The actor model achieves all capabilities in a single codebase. This single codebase implies that at least a portion of the actors (in a distributed cluster deployment) exist within the same process and memory boundaries. Much of the programmer's effort is the liability to guarantee that no mutable state is accessible through reference sharing among any two actors in order to ensure the model semantics.

RQ4: How does the performance of actors and microservices compare in a multi-core environment relative to a concurrent system scenario?

The benchmark results show that, with respect to our non-trivial scenario, the performance of an actor-based system is generally better than the performance of a microservice system. Since we have ensured that the domain-induced impact on the performance is uniform between both system implementations, the difference in the performance results from the efficiency we leverage from the underlying concurrent programming model. We have also shown that microservices which facilitate an asynchronous communication style merely have a slight overhead compared to actors. However, the benchmark also exposed that strictly synchronous communication (request/response) among services is clearly inferior to the request/asynchronous response style that actors facilitate. It is therefore surprising to the author that the scientific literature emphasizes REST as the primary microservice IPC mechanism.

7.2 Contributions

In this thesis, we compared the programming of concurrent computation with the actor and microservice model. We have explored the interrelations of the two models and filled a gap in the literature. In order to answer our research questions, we designed Echo, a non-trivial scenario for a concurrent domain-specific search engine prototype. We also provide an actor implementation based on Akka, as well as a microservice implementation based on the Spring framework of this system scenario. We evaluated the capabilities we are able to express with each model regarding concurrent programming concerns like parallelization/distribution, communication, persistence/IO, location transparency, isolation/independence, and scalability. Finally, we reported the results of an efficiency benchmark of the system implementations.

Further materials related to this work are available at:

https://max.irro.at/pub/dipl/

7.3 Future Work

The benchmark in this thesis is limited to a single multicore host. We did not benchmark the effects of distribution and horizontal scalability due to our limited access to hardware resources. Also, although we have motivated the integration of Akka actors and Spring-based microservices in Section 6.1.5, we did not investigate the effects on efficiency within a mixed architecture integrating actors and microservices as equal concurrent task units.

From our literature readings and practical experience, it is our believe that one major limitation of both actors and microservices is the lack of checkable compatibility (design by contract). Some work on static analysis for actors has been done in the literature. The work by D'Osualdo *et al.* [32] for example defines a checkable model for Erlang-styled concurrency. This model can be also expressed as processes of a suitable calculus. Recall that service-oriented programming languages incorporate the microservice model into the language level, and sometimes build upon a process calculus. We deem it worth to fathom into contract verification techniques (especially behavioral types [61,112]) for the actor model and SOC calculi, for example towards a more theoretical foundation for integrating actors and microservices.

From our experiences in this work, it is the author's general believe that the microservice styles suffers from the lack of a theoretical foundation. We therefore think that service-oriented programming languages are a highly promising evolutionary step worth of further investigation.

Appendix A Feed Structure Example

An RSS 2.0 [142] feed is an XML document. The feed's top level element is the `<rss>` tag. A feed becomes an Atom [108] feed if the `<rss>` element additionally has the XML namespace `xmlns:atom="http://www.w3.org/2005/Atom"`. The `<channel>` encloses the metadata entries of the feed (`<title>`, `<link>`, `<description>`, etc.) as well as a set of `<item>` elements. Each item has it's own set of metadata elements (`<title>`, `<pubDate>`, `<description>`, `<guid>`, etc.). The most important element of each item of a podcast feed is the `<enclosure>`, which provides the URL to the media file. Some feeds of the domain also apply additional XML namespaces to provide a wider range of metadata. Two prominent examples of such namespaces we also support in Echo's feed parsers are:

- `xmlns:itunes="http://www.itunes.com/dtds/podcast-1.0.dtd"` for metadata used in the iTunes Podcast Directory [63].
- `xmlns:psc="http://podlove.org/simple-chapters/"` for chaptermark metadata information in the Simple Chapters [83] format.

Below is a complete example of an RSS 2.0 feed. It provides some few metadata elements. The feed merely has a single `<item>` element. In general, a feed has several items. All metadata in this example is text of the so-called *Lorem ipsum*[26], a well-known placeholder text snippet. The text is not intended to transport any meaning. Instead, the design and publishing industries use the text to demonstrate a text structure or visual form. Hence, the Lorem ipsum is also suitable to illustrate the RSS feed structure, without providing actual meaningful content.

[26]`https://lipsum.com`

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<rss>
  <channel>
    <title>Lorem ipsum</title>
    <link>https://lorem.ispum.fm</link>
    <language>en-us</language>
    <description>
      Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed
      diam nonumy eirmod tempor invidunt ut labore et dolore magna
      aliquyam erat, sed diam voluptua
    </description>
    <item>
      <title>Lorem ipsum</title>
      <enclosure url="https://lorem.ispum.fm/episode-1.mp3"
                 length="17793193" type="audio/mpeg"/>
      <guid isPermaLink="false">https://lorem.ispum.fm/li001</guid>
      <pubDate>Fri, 13 Jul 2018 16:35:13 +0000</pubDate>
      <description>
        Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
        do eiusmod tempor incididunt ut labore et dolore magna aliqua.
        Ut enim ad minim veniam, quis nostrud exercitation ullamco
        laboris nisi ut aliquip ex ea commodo consequat. Duis aute
        irure dolor in reprehenderit in voluptate velit esse cillum
        dolore eu fugiat nulla pariatur. Excepteur sint occaecat
        cupidatat non proident, sunt in culpa qui officia deserunt
        mollit anim id est laborum.
      </description>
    </item>
  </channel>
</rss>
```

List of Acronyms and Abbreviations

AMQP	Advanced Message Queuing Protocol
AO	Active Object
API	Application Programming Interface
CPU	Central Processing Unit
DTO	Data Transfer Object
FIFO	First In First Out
fJAR	fat JAR
HTTP	Hypertext Transfer Protocol
IPC	Inter-Process Communication
IO	Input/Output
IoC	Inversion of Control
IR	Information Retrieval
JAR	Java Archive
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LB	Load Balancing
MS	Microservice
MSA	Microservice Architecture
OOP	Object-oriented Programming
OS	Operating System
PID	Process Identifier
REST	Representational State Transfer
RPC	Remote Procedure Call
RSS	Rich Site Summary
sJAR	skinny JAR
SOA	Service-oriented Architecture
SOC	Service-oriented Computing
SOP	Service-oriented Programming
STM	Software Transactional Memory
URL	Uniform Resource Locator
VM	Virtual Machine
XML	Extensible Markup Language

List of Figures

List of Tables

Bibliography

[1] Organization for the of Advancement Structured Information Standards (OASIS). Advanced Message Queuing Protocol (AMQP) Version 1.0. 2012. Accessed: 2018-05-22 `http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf`

[2] Gul Agha. Concurrent Object-Oriented Programming. 1990. *Commun. ACM* 33, 9, 125141. DOI:10.1145/83880.84528

[3] Gul Agha. Actors Programming for the Mobile Cloud. 2014. In *IEEE 13th International Symposium on Parallel and Distributed Computing, ISPDC 2014, Marseille, France, June 24-27, 2014*, 39. DOI:10.1109/ISPDC.2014.31

[4] Gul A. Agha. *ACTORS - A Model of Concurrent Computation in Distributed Systems*. 1985. MIT Press. ISBN 978-0-262-01092-4

[5] Gul A. Agha and WooYoung Kim. Actors: A Unifying Model for Parallel and Distributed Computing. 1999. *Journal of Systems Architecture* 45, 15, 12631277. DOI:10.1016/S1383-7621(98)00067-8

[6] Gul Agha, Svend Frølund, WooYoung Kim, Rajendra Panwar, Anna Patterson, and Daniel C. Sturman. Abstraction and modularity mechanisms for concurrent computing. 1993. *IEEE P&DT* 1, 2, 314. DOI:10.1109/88.218170

[7] Gul Agha and Carl Hewitt. Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism. 1985. In *Foundations of Software Technology and Theoretical Computer Science, Fifth Conference, New Delhi, India, December 16-18, 1985, Proceedings*, 1941. DOI:10.1007/3-540-16042-6_2

[8] Gul Agha, Christopher R. Houck, and Rajendra Panwar. Distributed Execution of Actor Programs. 1991. In *Languages and Compilers for Parallel Computing, Fourth International Workshop, Santa Clara, California, USA, August 7-9, 1991, Proceedings*, 117. DOI:10.1007/BFb0038654

[9] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A Foundation for Actor Computation. 1997. *J. Funct. Program.* 7, 1, 172. DOI:10.1017/S095679689700261X

[10] Varol Akman. Review of Actors: A Model of Concurrent Computation in Distributed Systems. 1990. *AI Magazine* 11, 4, 9295. http://www.aaai.org/ojs/index.php/aimagazine/article/view/861

[11] Jamie Allen. *Effective Akka: Patterns and Best Practices*. 2013. OReilly Media. ISBN 978-1449360078

[12] Gregory R. Andrews and Fred B. Schneider. Concepts and Notations for Concurrent Programming. 1983. *ACM Comput. Surv.* 15, 1, 343. DOI:10.1145/356901.356903

[13] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG*. 1993. Prentice Hall. ISBN 978-0-13-285792-5

[14] Kean Bacon and Tim Harris. *Operating Systems: Concurrent and Distributed Software Design*. 2003. Pearson Education. ISBN 978-0321117892

[15] Jos C. M. Baeten. A Brief History of Process Algebra. 2005. *Theor. Comput. Sci.* 335, 2-3, 131146. DOI:10.1016/j.tcs.2004.07.036

[16] Henry G. Baker and Carl Hewitt. The incremental garbage collection of processes. 1977. *SIGART Newsletter* 64, 5559. DOI:10.1145/872736.806932

[17] Christian Bednarek. Fyyd Podcast Directory. Accessed: 2018-09-07 https://fyyd.de

[18] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming*. 1990. Prentice Hall. ISBN 978-0-13-711821-2

[19] Manuel Bernhardt. *Reactive Web Applications*. 2016. Manning Publications Co. ISBN 978-1633430099

[20] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. *Orleans: Distributed Virtual Actors for Programmability and Scalability*. 2014. Accessed: 2018-02-17 https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/

[21] Andre B. Bondi. Characteristics of Scalability and Their Impact on Performance. 2000. In *Workshop on Software and Performance*, 195203. DOI:10.1145/350391.350432

[22] Jonas Boner. *Reactive Microservices Architecture: Design Principles for Distributed Systems*. 2016. OReilly Media. ISBN 978-1-491-95779-0

[23] Jonas Boner. *Reactive Microsystems: The Evolution of Microservices at Scale*. 2017. OReilly Media. ISBN 978-1-491-99433-7

[24] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. 1998. *Computer Networks* 30, 1-7, 107117. DOI:10.1016/S0169-7552(98)00110-X

[25] David W Bustard. *Concepts of Concurrent Programming.* 1990. Carnegie Mellon University, Software Engineering Institute. Accessed: 2018-04-03 `http://repository.cmu.edu/sei/110/`

[26] John Carnell. *Spring Microservices in Action.* 2017. Manning Publications Company. ISBN 978-1617293986

[27] Dominik Charousset, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch. Native Actors A Scalable Software Platform for Distributed, Heterogeneous Environments. 2013. In *Proceedings of the 2013 Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE!SPLASH 2013, Indianapolis, IN, USA, October 27-28, 2013*, 8796. DOI:10.1145/2541329.2541336

[28] Natalia Chechina, Kenneth MacKenzie, Simon J. Thompson, Phil Trinder, Olivier Boudeville, Viktoria Fordós, Csaba Hoch, Amir Ghaffari, and Mario Moro Hernandez. Evaluating Scalable Distributed Erlang for Scalability and Reliability. 2017. *IEEE Trans. Parallel Distrib. Syst.* 28, 8, 22442257. DOI:10.1109/TPDS.2017.2654246

[29] Jboss Community. Undertow Webserver. Accessed: 2018-08-29 `http://undertow.io`

[30] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design* (5th ed.). 2011. Addison-Wesley Publishing Company. ISBN 978-0132143011

[31] Al Danial. CLOC. Accessed: 2018-08-13 `https://github.com/AlDanial/cloc`

[32] Emanuele DOsualdo, Jonathan Kochems, and C.-H. Luke Ong. Automatic Verification of Erlang-Style Concurrency. 2013. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, 454476. DOI:10.1007/978-3-642-38856-9_24

[33] Nicola Dragoni, Schahram Dustdar, Stephan Thordal Larsen, and Manuel Mazzara. Microservices: Migration of a Mission Critical System. 2017. *CoRR* abs/1704.04173,. arXiv:1704.04173

[34] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. 2017. In *Present and Ulterior Software Engineering.* 195216. DOI:10.1007/978-3-319-67425-4_12

[35] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina. Microservices: How To Make Your Application Scale. 2017. *CoRR* abs/1702.07149,. arXiv:1702.07149

[36] Michael B Feldman. *Language and System Support for Concurrent Programming.* 1990. Carnegie Mellon University, Software Engineering Institute. Accessed: 2018-03-18 `http://repository.cmu.edu/sei/194/`

[37] Matthias Felleisen. On the Expressive Power of Programming Languages. 1991. *Sci. Comput. Program.* 17, 1-3, 3575. DOI:10.1016/0167-6423(91)90036-W

[38] Cormac Flanagan and Matthias Felleisen. The Semantics of Future and an Application. 1999. *J. Funct. Program.* 9, 1, 131. `http://journals.cambridge.org/action/displayAbstract?aid=44231`

[39] Apache Software Foundation. Lucene. Accessed: 2018-06-02 `https://lucene.apache.org`

[40] Apache Software Foundation. HttpComponents. Accessed: 2018-06-02 `https://hc.apache.org`

[41] Apache Software Foundation. Kafka. Accessed: 2018-06-02 `https://kafka.apache.org`

[42] Martin Fowler and James Lewis. Microservices: a definition of this new architectural term. 2014. Accessed: 2017-09-22 `http://martinfowler.com/articles/microservices.html`

[43] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. 2017. In *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*, 2130. DOI:10.1109/ICSA.2017.24

[44] Brian Goetz, Tim Peierls, Joshua J. Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice.* 2006. Addison-Wesley. ISBN 978-0-321-34960-6

[45] Daniele Gorla and Uwe Nestmann. Full abstraction for expressiveness: history, myths and facts. 2016. *Mathematical Structures in Computer Science* 26, 4, 639654. DOI:10.1017/S0960129514000279

[46] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification – Java SE 8 Edition.* 2015. Accessed: 2018-06-10 `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`

[47] Patrick Gotthard. ROME. Accessed: 2018-02-06 `https://rometools.github.io/rome/`

[48] Marco Gribaudo, Mauro Iacono, and Daniele Manini. Performance Evaluation Of Massively Distributed Microservices Based Applications. 2017. In *European Conference on Modelling and Simulation, ECMS 2017, Budapest, Hungary, May 23-26, 2017, Proceedings.*, 598604. DOI:10.7148/2017-0598

[49] Claudio Guidi, Ivan Lanese, Manuel Mazzara, and Fabrizio Montesi. Microservices: A Language-Based Approach. 2017. In *Present and Ulterior Software Engineering.* 217225. DOI:10.1007/978-3-319-67425-4_13

[50] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. SOCK: A Calculus for Service Oriented Computing. 2006. In *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, 327338. DOI:10.1007/11948148_27

[51] Claudio Guidi and Fabrizio Montesi. Reasoning About a Service-oriented Programming Paradigm. 2009. In *Proceedings Fourth European Young Researchers Workshop on Service Oriented Computing, YR-SOC 2009, Pisa, Italy, 17-19th June 2009.*, 6781. DOI:10.4204/EPTCS.2.6

[52] Philipp Haller. On the Integration of the Actor Model in Mainstream Technologies: The Scala Perspective. 2012. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! 2012, October 21-22, 2012, Tucson, Arizona, USA*, 16. DOI:10.1145/2414639.2414641

[53] Philipp Haller and Martin Odersky. Event-Based Programming Without Inversion of Control. 2006. In *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006, Proceedings*, 422. DOI:10.1007/11860990_2

[54] Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. 2009. *Theor. Comput. Sci.* 410, 2-3, 202220. DOI:10.1016/j.tcs.2008.09.019

[55] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Futures and Promises. Scala Documentation. Accessed: 201-06-07 `https://docs.scala-lang.org/overviews/core/futures.html`

[56] HashiCorp. Consul. Accessed: 2018-05-16 `https://www.consul.io`

[57] Sara Hassan and Rami Bahsoon. Microservices and Their Design Trade-Offs: A Self-Adaptive Roadmap. 2016. In *IEEE International Conference on Services Computing, SCC 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, 813818. DOI:10.1109/SCC.2016.113

[58] Pat Helland. Immutability Changes Everything. 2015. *ACM Queue* 13, 9, 40. DOI:10.1145/2857274.2884038

[59] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. 1973. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, 235245. http://ijcai.org/Proceedings/73/Papers/027B.pdf

[60] C. A. R. Hoare. Communicating Sequential Processes. 1978. *Commun. ACM* 21, 8, 666677. DOI:10.1145/359576.359585

[61] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. 2016. *ACM Comput. Surv.* 49, 1, 3:13:36. DOI:10.1145/2873052

[62] Shams Mahmood Imam and Vivek Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. 2014. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014*, 6780. DOI:10.1145/2687357.2687368

[63] Apple Inc. iTunes Podcast Directory. Accessed: 2018-09-07 https://itunes.apple.com/us/genre/podcasts/id26

[64] Lightbend Inc. Akka. Accessed: 2018-01-03 https://akka.io

[65] Lightbend Inc. Actors (Scala Variant). Akka Version 2.5 Documentation. Accessed: 2018-04-22 https://doc.akka.io/docs/akka/2.5/scala/actors.html

[66] Lightbend Inc. Akka and the Java Memory Model. Akka Version 2.5 Documentation. Accessed: 2018-04-14 https://doc.akka.io/docs/akka/2.5/general/jmm.html

[67] Lightbend Inc. Futures. Akka Version 2.5 Documentation. Accessed: 2018-04-12 https://doc.akka.io/docs/akka/2.5/futures.html

[68] Lightbend Inc. Akka HTTP. Akka Version 2.5 Documentation. Accessed: 2018-05-24 https://doc.akka.io/docs/akka-http/current/index.html

[69] Lightbend Inc. Scheduler. Akka Version 2.5 Documentation. Accessed: 2018-04-22 https://doc.akka.io/docs/akka/2.5/scala/scheduler.html

[70] Lightbend Inc. Akka Typed. Akka Version 2.5 Documentation. Accessed: 2018-04-26 https://doc.akka.io/docs/akka/2.5.5/scala/typed.html

[71] Lightbend Inc. Supervision and Monitoring. Akka Version 2.5 Documentation. Accessed: 2018-04-26 https://doc.akka.io/docs/akka/2.5/scala/general/supervision.html

[72] Lightbend Inc. Message Delivery Reliability. Akka Version 2.5 Documentation. Accessed: 2018-04-10 https://doc.akka.io/docs/akka/2.5/general/message-delivery-reliability.html

[73] Lightbend Inc. Lagom. Accessed: 2018-05-28 https://www.lagomframework.com

[74] Lightbend Inc. Spray JSON. Accessed: 2018-02-05 https://github.com/spray/spray-json

[75] Netflix Inc. Eureka. Accessed: 2018-08-10 https://github.com/Netflix/eureka

[76] Netflix Inc. Zuul. Accessed: 2018-08-10 https://github.com/Netflix/zuul

[77] Netflix Inc. Ribbon. Accessed: 2018-08-10 https://github.com/Netflix/ribbon

[78] Pivotal Software Inc. Spring. Accessed: 2018-03-14 https://spring.io

[79] Pivotal Software Inc. Spring Boot Version 1.5.10.RELEASE Documentation. Accessed: 2018-05-03 https://docs.spring.io/spring-boot/docs/1.5.10.RELEASE/reference/htmlsingle/

[80] Pivotal Software Inc. Spring Cloud Version Finchley.M5 Documentation. Accessed: 2018-05-03 http://cloud.spring.io/spring-cloud-static/Finchley.M5/single/spring-cloud.html

[81] Pivotal Software Inc. Spring Framework Version 4.3.14.RELEASE Documentation. Accessed: 2018-05-03 https://docs.spring.io/spring/docs/4.3.14.RELEASE/spring-framework-reference/htmlsingle/

[82] Pivotal Software Inc. RabbitMQ. Accessed: 2018-03-14 https://www.rabbitmq.com

[83] Podlove Initiative. Simple Chapters. Accessed: 2018-08-17 https://podlove.org/simple-chapters/

[84] Lianghuan Kang and Donggang Cao. An Extension to Computing Elements in Erlang for Actor Based Concurrent Programming. 2012. In *15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, ISORC Workshops 2012, Shenzhen, China, April 11, 2012*, 99105. DOI:10.1109/ISORCW.2012.28

[85] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. 2009. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ 2009, Calgary, Alberta, Canada, August 27-28, 2009*, 1120. DOI:10.1145/1596655.1596658

[86] Günter Kniesel. *Encapsulation = Visibility + Accessibility.* 1996. Universitat Bonn, Institut fur Informatik III. Accessed: 2018-04-14 http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.536

[87] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 years of actors: a taxonomy of actor models and their key properties. 2016. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands, October 30, 2016*, 3140. DOI:10.1145/3001886.3001890

[88] R. Greg Lavender and Douglas C. Schmidt. Active Object – An Object Behavioral Pattern for Concurrent Programming. 1995. Accessed: 2018-05-15 `https://www.dre.vanderbilt.edu/~schmidt/PDF/Active-Objects.pdf`

[89] Doug Lea. A Java Fork/Join Framework. 2000. In *Proceedings of the ACM 2000 Java Grande Conference, San Francisco, CA, USA, June 3-5, 2000*, 3643. DOI:10.1145/337449.337465

[90] Mohsen Lesani, Martin Odersky, and Rachid Guerraoui. *Concurrent Programming Paradigms: A Comparison in Scala.* `https://infoscience.epfl.ch/record/136824/files/Formatted%20Report.pdf`

[91] David Lillis, Rem W. Collier, Mauro Dragone, and Gregory M. P. OHare. An Agent-Based Approach to Component Management. 2009. In *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 1*, 529536. DOI:10.1145/1558013.1558086

[92] Barbara Liskov and Liuba Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. 1988. In *Proceedings of the ACM SIGPLAN88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, 260267. DOI:10.1145/53990.54016

[93] Google LLC. Angular. Accessed: 2018-08-21 `https://angular.io`

[94] Robert Love. *Linux System Programming: System and Library Calls Every Programmer Needs to Know.* 2007. OReilly. ISBN 978-0-596-00958-8

[95] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* 2008. Cambridge University Press. ISBN 978-0-521-86571-5

[96] Manuel Mazzara, Ruslan Mustafin, Larisa Safina, and Ivan Lanese. Towards Microservices and Beyond: An incoming Paradigm Shift in Distributed Computing. 2016. *CoRR* abs/1610.01778,. arXiv:1610.01778

[97] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition.* 1997. Prentice-Hall. ISBN 978-0136291558

[98] Robin Milner. *A Calculus of Communicating Systems.* 1980. Springer. DOI:10.1007/3-540-10235-3

[99] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I+II. 1992. *Inf. Comput.* 100, 1, 177. DOI:10.1016/0890-5401(92)90008-4

[100] Bogdan Mingela, Nikolay Troshkov, Manuel Mazzara, Larisa Safina, and Alexander Tchitchigin. Towards Static Type-checking for Jolie. 2017. *CoRR* abs/1702.07146,. arXiv:1702.07146

[101] Ahmed Abdel Moamen, Dezhong Wang, and Nadeem Jamali. Supporting Resource Control for Actor Systems in Akka. 2017. In *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, 26422645. DOI:10.1109/ICDCS.2017.291

[102] Ugo Montanari and Carolyn L. Talcott. Can Actors and pi-Agents Live Together? 1997. *Electr. Notes Theor. Comput. Sci.* 10, 189196. DOI:10.1016/S1571-0661(05)80697-8

[103] Fabrizio Montesi. Process-aware web programming with Jolie. 2016. *Sci. Comput. Program.* 130, 6996. DOI:10.1016/j.scico.2016.05.002

[104] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-Oriented Programming with Jolie. 2014. In *Web Services Foundations.* 81107. DOI:10.1007/978-1-4614-7518-7_4

[105] Fabrizio Montesi and Janine Weber. Circuit Breakers, Discovery, and API Gateways in Microservices. 2016. *CoRR* abs/1609.05830,. arXiv:1609.05830

[106] Thomas Müller. H2 Database Engine. Accessed: 2018-09-05 `http://www.h2database.com/html/main.html`

[107] Sam Newman. *Building Microservices: Designing Fine-Grained Systems, 1st Edition.* 2015. OReilly. ISBN 978-1491950357

[108] Mark Nottingham and Robert Sayre. RFC 4287: The Atom Syndication Format. 2005. Accessed: 2018-08-26 `https://tools.ietf.org/html/rfc4287`

[109] Martin Odersky. *The Scala Language Specification – Version 2.12.* 2016. Accessed: 2018-07-16 `https://scala-lang.org/files/archive/spec/2.12/`

[110] OpenFeign. Feign. Accessed: 2018-08-10 `https://github.com/OpenFeign/feign`

[111] Laurie J. Patterson. The Technology Underlying Podcasts. 2006. *IEEE Computer* 39, 10, 103105. DOI:10.1109/MC.2006.361

[112] Jorge A. Pérez. The Challenge of Typed Expressiveness in Concurrency. 2016. In *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, 239247. DOI:10.1007/978-3-319-39570-8_16

[113] Calvin Pedzai, Ndapandula Nakashole, and Hussein Suleman. *An Approach to Better System Resource Utilization for Search Engine Clusters*. 2006. Department of Computer Science, University of Cape Town. Accessed: 2018-07-21 http://pubs.cs.uct.ac.za/archive/00000363/01/techical.pdf

[114] Kisalaya Prasad, Avanti Patil, and Heather Miller. Futures and Promises. In *Programming Models for Distributed Computing*. Accessed: 2018-04-22 http://dist-prog-book.com/chapter/2/futures.html

[115] Akka.NET Project. Akka.NET. Accessed: 2018-09-30 https://getakka.net

[116] Eric S Raymond. *The Art of UNIX Programming*. 2003. Addison-Wesley Professional. ISBN 978-0131429017

[117] Stephan Rehfeld, Henrik Tramberend, and Marc Erich Latoschik. An actor-based distribution model for Realtime Interactive Systems. 2013. In *6th Workshop on Software Engineering and Architectures for Realtime Interactive Systems, SEARIS 2013, Orlando, FL, USA, March 17, 2013*, 916. DOI:10.1109/SEARIS.2013.6798103

[118] Raymond Roestenburg, Rob Bakker, and Rob Williams. *Akka in Action*. 2015. Manning Publications Co. ISBN 978-1617291012

[119] Andrea Rosà, Lydia Y. Chen, and Walter Binder. Profiling actor utilization and communication in Akka. 2016. In *Proceedings of the 15th International Workshop on Erlang, Nara, Japan, September 18-22, 2016*, 2432. DOI:10.1145/2975969.2975972

[120] Andrea Rosà, Lydia Y. Chen, and Walter Binder. AkkaProf: A Profiler for Akka Actors in Parallel and Distributed Applications. 2016. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, 139147. DOI:10.1007/978-3-319-47958-3_8

[121] Tasneem Salah, M. Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, and Yousof Al-Hammadi. The Evolution of Distributed Systems Towards Microservices Architecture. 2016. In *11th International Conference for Internet Technology and Secured Transactions, ICITST 2016, Barcelona, Spain, December 5-7, 2016*, 318325. DOI:10.1109/ICITST.2016.7856721

[122] Michael L. Scott. *Programming Language Pragmatics (2. ed.)*. 2006. Morgan Kaufmann. ISBN 978-0-12-633951-2

[123] Dharmendra Shadija, Mo Rezai, and Richard Hill. Microservices: Granularity vs. Performance. 2017. In *Companion Proceedings of the 10th International Conference on Utility and Cloud Computing, UCC 2017, Austin, TX, USA, December 5-8, 2017*, 215220. DOI:10.1145/3147234.3148093

[124] Dharmendra Shadija, Mo Rezai, and Richard Hill. Towards an Understanding of Microservices. 2017. In *23rd International Conference on Automation and Computing, ICAC 2017, Huddersfield, United Kingdom, September 7-8, 2017*, 16. DOI:10.23919/IConAC.2017.8082018

[125] Vivek Shah and Marcos Vaz Salles. Actor Database Systems: A Manifesto. 2017. *CoRR* abs/1707.06507,. arXiv:1707.06507

[126] Olin Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. 1997. In *In Proceedings of the Second ACM SIGPLAN Workshop on Continuations*, 21. Accessed: 2018-07-26 http://www.ccs.neu.edu/home/shivers/papers/cps-threads.ps

[127] Munindar P. Singh and Amit K. Chopra. Correctness Properties for Multiagent Systems. 2009. In *Declarative Agent Languages and Technologies VII, 7th International Workshop, DALT 2009, Budapest, Hungary, May 11, 2009. Revised Selected and Invited Papers*, 192207. DOI:10.1007/978-3-642-11355-0_12

[128] Paolo A. G. Sivilotti and Charles P. Giles. The Specification of Distributed Objects: Liveness and Locality. 1999. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, 11. DOI:10.1145/781995.782006

[129] Alan Snyder. The Essence of Objects: Concepts and Terms. 1993. *IEEE Software* 10, 1, 3142. DOI:10.1109/52.207219

[130] Giandomenico Spezzano, Domenico Talia, and Marco Vanneschi. A Concurrent Programming Support for Distributed Systems. 1990. *Computing Systems* 3, 3, 423447. http://www.usenix.org/publications/compsystems/1990/sum_spezzano.pdf

[131] Damiano Spina, Johanne R. Trippas, Lawrence Cavedon, and Mark Sanderson. Extracting Audio Summaries to Support Effective Spoken Document Search. 2017. *Journal of the Association for Information Science and Technology* 68, 9, 21012115. DOI:10.1002/asi.23831

[132] Venkat Subramaniam. *Programming Concurrency on the JVM*. 2011. Pragmatic Bookshelf. ISBN 978-1-93435-676-0

[133] Janwillem Swalens, Stefan Marr, Joeri De Koster, and Tom Van Cutsem. Towards Composable Concurrency Abstractions. 2014. In *Proceedings 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2014, Grenoble, France, 12 April 2014.*, 5460. DOI:10.4204/EPTCS.155.8

[134] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms, 2nd Edition*. 2007. Pearson Education. ISBN 978-0-13-239227-3

[135] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why Do Scala Developers Mix the Actor Model with other Concurrency Models? 2013. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, 302326. DOI:10.1007/978-3-642-39038-8_13

[136] Ivan Valkov, Natalia Chechina, and Phil Trinder. Comparing Languages for Engineering Server Software: Erlang, Go, and Scala with Akka. 2018. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, 218225. DOI:10.1145/3167132.3167144

[137] Steve Vinoski. Concurrency with Erlang. 2007. *IEEE Internet Computing* 11, 5, 9093. DOI:10.1109/MIC.2007.104

[138] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A Note on Distributed Computing. 1996. In *Mobile Object Systems - Towards the Programmable Internet, Second International Workshop, MOS96, Linz, Austria, July 8-9, 1996, Selected Presentations and Invited Papers*, 4964. DOI:10.1007/3-540-62852-5_6

[139] Craig Walls and Ryan Breidenbach. *Spring in Action*. 2007. Manning Publications Co. ISBN 9781933988139

[140] Peter Wegner. Concepts and paradigms of object-oriented programming. 1990. *OOPS Messenger* 1, 1, 787. DOI:10.1145/382192.383004

[141] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Safe futures for Java. 2005. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, 439453. DOI:10.1145/1094811.1094845

[142] Dave Winer, Brent Simmons, and Jon Udell. RSS 2.0 Specification. 2003. Accessed: 2018-08-26 `http://cyber.harvard.edu/rss/rss.html`

[143] ChengZhi Xu, Hong Zhu, Ian Bayley, David E. Lightfoot, Mark Green, and Peter Marshall. CAOPLE: A Programming Language for Microservices SaaS. 2016. In *2016 IEEE Symposium on Service-Oriented System Engineering, SOSE 2016, Oxford, United Kingdom, March 29 - April 2, 2016*, 3443. DOI:10.1109/SOSE.2016.46

[144] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming ABCL/1. 1986. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA86), Portland, Oregon, Proceedings.*, 258268. DOI:10.1145/28697.28722

[145] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. Benchmarking microservice systems for software engineering research. 2018. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 323324. DOI:10.1145/3183440.3194991