

INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

CONCURRENT PROGRAMMING: CONCEPTS AND LANGUAGES

Raphael Hagl

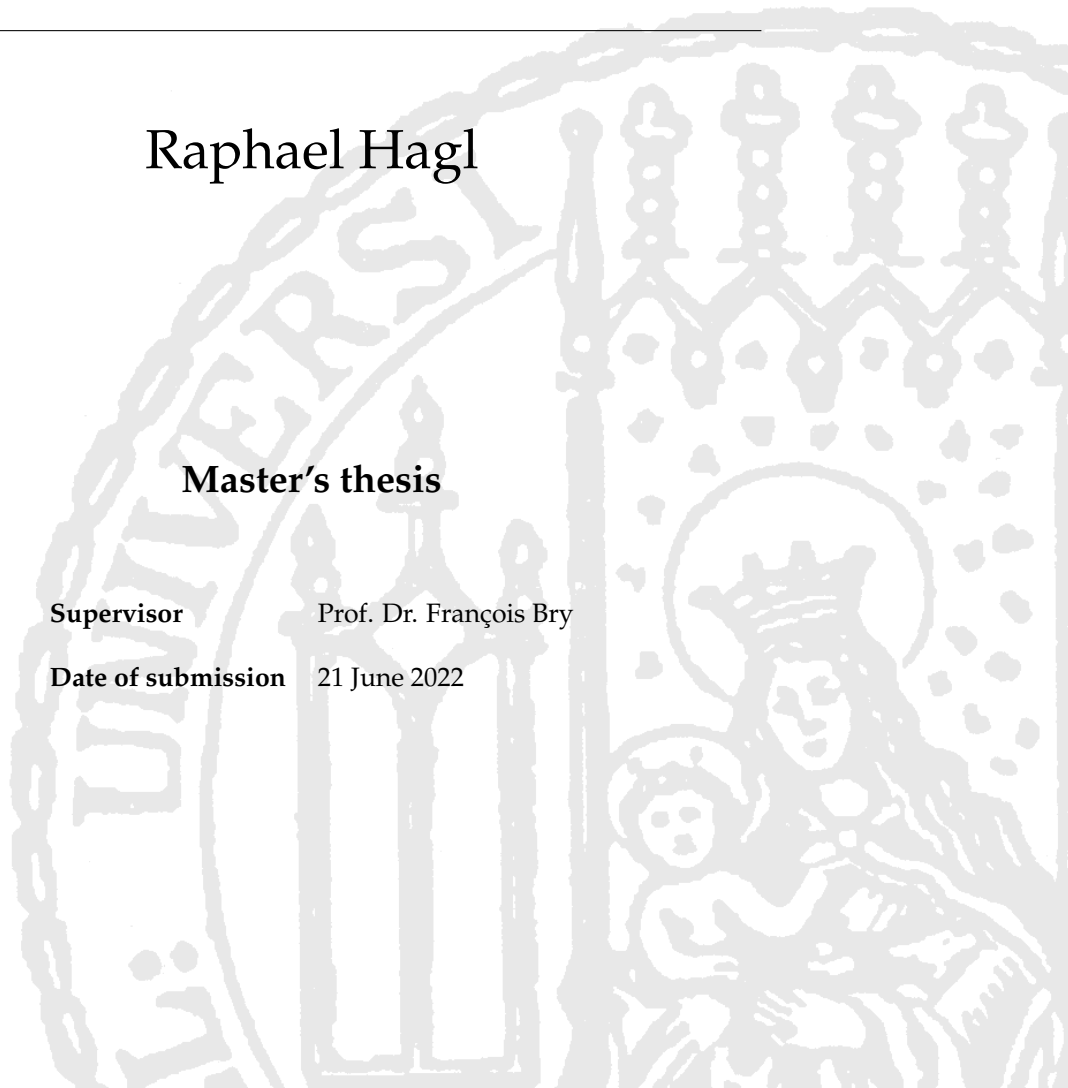
Master's thesis

Supervisor

Prof. Dr. François Bry

Date of submission

21 June 2022



Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Munich, 21 June 2022

Raphael Hagl



Abstract

Developing reliable software is a hard task and concurrent programming is even more so. Concurrency entails a whole new level of complexity. In multithreaded environments the execution paths of separate threads may be interleaved in incalculable ways. This may lead to unexpected and faulty behavior.

Regardless of the system under consideration, concurrency almost certainly plays an integral role in many computing applications and handling it is always challenging. This holds true in particular for large software systems where lots of different code pieces are executed concurrently, likely interwoven with each other in incalculable ways. Moreover, as concurrency becomes inevitable for ever-growing software projects, code bases tend to get cluttered, testability decreases, and they become hard to maintain and even harder to extend.

Furthermore, while concurrent programming techniques like locks eliminate certain failure scenarios, they may introduce other errors, such as deadlocks and livelocks. Concurrent programming done wrong can be hazardous for the performance of a program.

This thesis aims at addressing the need for a comprehensive introduction to concurrent programming. Many concepts like locks, atomic operations and semaphores are discussed in detail. Practical challenges that may arise when designing and implementation concurrent software are also covered.

Afterwards, concurrent programming features of several languages are laid out and demonstrated through several code examples. This work emphasizes the fact that different concepts and techniques of concurrent programming are related to and especially useful in combination with each other.

Acknowledgments

First and foremost, I want to thank Prof. Dr. François Bry who made this thesis possible. He was of great support along the way, and always had an open ear to my problems and ideas. I also want to thank Bagrat Ter-Akopyan for his great help in improving the structure of this thesis and keeping me on track when my focus went astray. Special thanks go to Maximilian Weber for his commitment for proofreading my work despite being completely foreign to the subject.

This work is dedicated to my sons, Matthias and Tobias, and my wife, Diana. I am grateful for their love, support and understanding.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 2 | Concepts of concurrent programming | 3 |
| 2.1 | Foundations | 4 |
| 2.2 | Concurrent programming primitives | 6 |
| 2.3 | Synchronization | 8 |
| 2.4 | Semaphores | 9 |
| 2.5 | Models and patterns in concurrent programming | 9 |
| 2.5.1 | Message passing | 10 |
| 2.5.2 | Main-Subordinate | 11 |
| 2.5.3 | Producer-Consumer | 12 |
| 2.5.4 | Actor model | 13 |
| 3 | Practical challenges of concurrent programming | 15 |
| 3.1 | A Messenger - an introductory example | 15 |
| 3.2 | Potential for errors | 17 |
| 3.2.1 | Safety | 17 |
| 3.2.2 | Liveness | 19 |
| 3.3 | Decision-making | 20 |
| 4 | Languages | 23 |
| 4.1 | Java | 24 |
| 4.1.1 | Thread | 24 |
| 4.1.2 | Foundations | 25 |
| 4.1.3 | Locks | 30 |
| 4.1.4 | Creating and managing tasks | 33 |
| 4.1.5 | Concurrent collections by example | 34 |
| 4.1.6 | Scala | 37 |
| 4.2 | Rust | 38 |
| 4.2.1 | Fearless concurrency | 39 |
| 4.2.2 | Running code in threads | 39 |
| 4.2.3 | Reference counting | 39 |
| 4.2.4 | Locks | 40 |
| 4.2.5 | Ordering of atomic operations | 41 |
| 4.2.6 | Send and Sync | 42 |
| 4.2.7 | Message passing | 42 |
| 4.2.8 | A task scheduler | 43 |

| | | |
|----------|---|-----------|
| 4.3 | JavaScript | 48 |
| 4.3.1 | Event loop | 48 |
| 4.3.2 | Promise | 50 |
| 4.4 | Go | 51 |
| 4.4.1 | Goroutines | 52 |
| 4.4.2 | Channels | 52 |
| 4.4.3 | Channel-based primitives | 54 |
| 4.4.4 | Message routing topologies | 55 |
| 4.5 | A comparison of the languages | 56 |
| 5 | Conclusion | 59 |
| 5.1 | Concurrent by design | 59 |
| 5.2 | Outlook | 60 |
| | Bibliography | 61 |

CHAPTER 1

Introduction

Consider the underground railroad system of a big city. It consists of many underground stations interconnected by rails. A lot of trains are driven separately on various of these railways. At each station passengers may get on and off a train. Due to fixed scheduling plans, passengers know — with some uncertainty caused by unexpected delays — the departure times of the underground trains on a respective line. This enables them to find the apparently best route through the underground railroad system from one station to another. Sometimes — especially during rush hours — it can occur that a specific train is too crowded for more passengers to enter. Thus, they either have to wait for the next train or decide whether to take a different route which might be a better option than waiting.

While being independent of one another in the outer areas, most railways are joined with others when getting closer to the city center. At these spots underground trains are sometimes obliged to wait for a prior train to leave before they can reach the next station. Usually the train driver is made aware of that via specific traffic signs.

Such an underground system is a highly distributed concurrent system. What does this mean? At an abstract level, train stations can be seen as individual nodes with the rails building pathways of communication between them. There are two kinds of acting entities: First, on a large scale, the trains running on the underground system. Their behavior becomes concurrent whenever two or more trains *compete* for the entrance to a same station, that is the joining points of the rails. Second, on a smaller scale, there are the people using the trains to travel. Here concurrency is involved in scenarios where passengers *compete* for a place on a train. In general, concurrency is the overlapping of activities (driving into the next station or taking place on a train) performed by different acting entities (trains or passengers) within the same system.

Motivation

Regardless of the system under consideration, concurrency almost certainly plays an integral role in many computing applications and handling it is always challenging. This holds true in particular for large software systems where lots of different code pieces are executed concurrently, likely interwoven with each other in incalculable ways. The craftsmanship of concurrent programming is still not well understood among many software devel-

opers¹. Moreover, as concurrency becomes inevitable for ever-growing software projects, code bases tend to get cluttered, testability decreases, and they become hard to maintain and even harder to extend.

The reasons for this phenomenon are manifold, but can be traced to a general lack of knowledge transfer regarding this topic. At many universities concurrent programming and its related concepts are only briefly touched upon [LW11]. Other kinds of education, like trainings at companies or online courses, do not suffice to widespread a competence in concurrent programming. Concurrent programming is treated shabbily, little to no attention is paid on this topic. A lot of research regarding concurrent programming has been conducted in the past decades, but most of it has not reached out of academia to real-world enterprise development. There is profound literature on specific concepts of concurrent programming, especially about concurrent collections [MS96, ZS05, HHL⁺07, HSY04, SLS09, SS04]. Yet, educational work seldom deals with the topic as a whole [Tur13], showing how to put these pieces together — specifically in the context of already large and complex software projects.

This state of affairs is understandable because, concurrency entails high intrinsic complexity. Various code segments possibly interact concurrently with each other in uncountable ways. Manifold issues can arise during the interleaving of executions².

Thus, it is no wonder that — more often than not — software developers shun concurrent programming altogether whenever possible.

Contribution

This thesis aims at addressing the need for a comprehensive introduction to concurrent programming. It has been conceived as sort of cook book. While dealing with concurrent programming, questions like the following may arise:

- When is a non-blocking solution preferable?
- When do simple atomic operations suffice?
- How to gracefully handle faults in concurrent executions?
- How to efficiently join different paths of concurrent execution?

To answer these and other questions regarding concurrent programming, the concepts of concern are examined in a non-isolated fashion, clarifying their relations with each other and emphasizing the respective scenarios in which to use either of them.

This work is structured as follows: Following this introduction, Chapter 2 clarifies terminology used throughout the thesis and gives a detailed explanation on important concepts of concurrent programming. Chapter 3 lays out practical challenges for concurrent programming. Chapter 4 discusses concurrent programming techniques in different languages. Carefully constructed and handwritten code examples illustrate the respective topics. Chapter 5 concludes this thesis.

¹I do not intend to tread on someone's toes, but experience has shown just that.

²See Chapter 3

Concepts of concurrent programming

This chapter explains important concepts of concurrent programming and describes how to distinguish it from parallel and asynchronous programming. For better understanding, the chapter also bridges back to the introductory example of the underground system as it depicts a lot of the concepts and building blocks of concurrent systems.

*We think of threads, when we think of parallelism, and we think of threads, when we think of asynchrony.*¹ Concurrent, parallel and asynchronous programming are distinct. Nonetheless, they often get confused with each other. This state of affairs is understandable, because from a practical point of view, these methodologies are usually closely entangled. The following gives a short definition of each and points out in which ways they intercorrelate.

Concurrent programming Depending on the language and/or framework in use, there are different *units of execution*. They may be named processes, operating system as well as so-called *green* threads, tasks, promises, futures, subroutines and others [Erb12]. The preferred term used throughout this thesis is *task*.

Concurrent programming deals with all kinds of task overlapping. As Alan Turon describes it, *concurrent programming is the management of sharing and timing* [Tur13]. This statement synthesizes the primary duties of a developer concerned with concurrent programming: (1) resources have to be shared among different tasks, (2) the sharing has to be timed correctly, that is, scenarios where the overlapped execution of tasks may lead to a faulty behavior of the program must be avoided.

Tasks running concurrently do not necessarily have to be executed in parallel. In fact, before the dawn of multicore processors, every program — and as well each task within a program — on a machine was executed by one and the same core.

Parallel programming Parallel programming utilizes computational resources of the underlying machine to execute several computations simultaneously. Concurrency is not necessary to achieve parallelization. Different tasks might run completely independent of one another. This holds especially for hardware parallelization. For instance, hyperscaling CPU's have several ALU's built-in that enable simultaneous computations directly on the hardware.

¹Venkat Subramaniam, Devovx UK, 2018

However, simultaneous task executions might as well overlap, that is run concurrently. For the soundness of a parallelized program, choosing the proper technique of concurrent programming is vital.

Asynchronous programming Since *async/await* have been introduced to many popular programming languages, such as C#, Rust or JavaScript, asynchronous programming has become prominent in the software developer community. When code is run asynchronously, it will be executed some time in the future. This usually involves querying a database, communicating with an external API, reading from a file or executing another program. If the resulting value is mandatory for further progress in the program, these asynchronous tasks have to be *awaited*.

Asynchrony does not necessarily entail concurrency, but asynchronous tasks might run concurrently - and they might as well be parallelized².

[...] *interesting applications of concurrency involve the deliberate and controlled mutation of shared state* [JGF96]. Concurrent programming is all about properly guarding shared mutable state. Essentially, state represents the specific value configuration of data structures in a program. The fundamental job of imperative programs is the management of its state. In object-oriented programming languages, state is typically expressed by objects and their properties. Due to the manifold issues regarding state changes³, purely functional programming languages prevent state from being mutable, which then in turn entails a hit on practicality [FC12].

Due to its non-deterministic characteristics, concurrent programming is particularly difficult. Concurrency overturns the sequential view on a program, which is the intuitive way of approaching and understanding code.

2.1 Foundations

Usually, the creation of tasks utilizes threads⁴ in one way or another. Therefore, a comprehension of a thread's life cycle is important to fully understand the usage of techniques in concurrent programming. Figure 2.1 illustrates the different states a thread can assume. Once a thread is *ready*, it can be executed. The scheduler grants each thread a time slot on the processor [TB14]. Once the time is over, another thread gets the chance to execute. This so-called *context switch* is quite costly [LDS07]. A running thread might get *blocked*. This is the case when explicitly set to sleep or when a lock cannot be acquired. After the timeout or when the respective lock is released again, the thread once again becomes ready. A thread can also be in *waiting*. This usually occurs when the thread is waiting for another thread to terminate, to *join*. After all the work is done, a thread assumes the state *terminated*.

While some concepts of concurrent programming seem to be completely independent of thread state, others are obviously closely related. Nonetheless, keeping in mind the different states a thread can assume is always helpful when having to decide on which concept to rely on for a certain task. The following sections give a detailed introduction to different techniques used for concurrent programming.

²In *node.js* several asynchronous calls can be executed simultaneously.

³Recall the infamous *The loop is running infinitely because I did not consequently update dependent variables*.

⁴Not every programming language supports threads. This will be discussed in Chapter 4

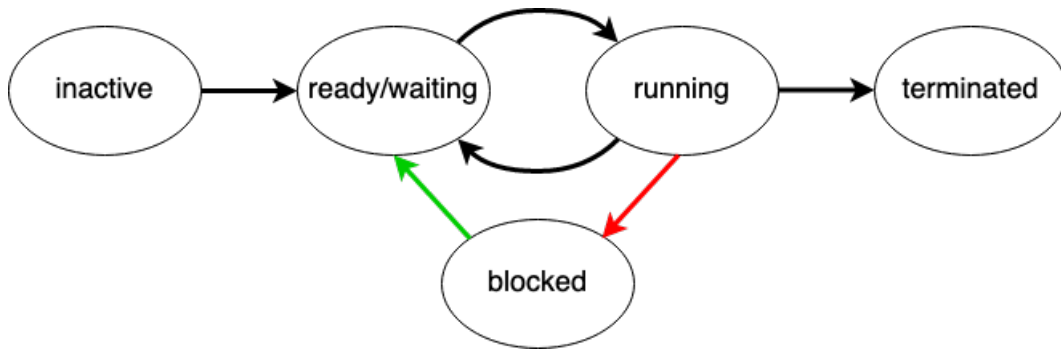


Figure 2.1: States of a Thread

Atomicity

Atomicity originates from database systems and is part of the *ACID* [GMUW09] properties every database transaction should adhere to. In an atomic operation either all or none of the included statements are executed. From the viewpoint of an external observer, everything inside an atomic operation seems to happen at once. It cannot be divided into smaller pieces and is therefore atomic. Care has to be taken as to which operations are actually atomic. Statements like `counter += 1` might seem to be atomic, when in fact they are not. They consist of three separate operations: (1) load the current value of a variable, (2) calculate the new value, (3) write it back.

Mutual exclusion

Recall the introductory example of the underground system. At two points the acting entities, trains and passengers, compete for limited shared resources: (1) stations where several underground lines are joined, (2) places on a train. These are called *critical sections*. Concurrent programming has to ensure that only one task at a time can enter such a critical section. This is *mutual exclusion*. Figure 2.2 shows three concurrent tasks competing for a critical section: *Guard* protects the entrance to the critical section so that only one task at a time can enter it. Once the current task has finished, *Release* signals the guard that another task may now enter.

Interleaving and reordering

Every thread and thus every task running within it is granted some execution time on a processor (core). Therefore, it is possible that the execution of a running task is stopped amidst carrying out a function body, right after the last atomic operation that fits into the time slot. Another task might then continue its work at a similar spot within the code, resulting in an interleaved execution of the respective tasks.

Listing 2.1 shows a fraction of a *Runnable* implementation in Java⁵. Consider tasks A and B both running this code. A might just have been suspended while performing the *add* method in line five. As this is a compound operation, the call to it may not have returned yet.

The statement in line six does not build upon the result of the prior one. Line five and six are therefore technically independent and might be reordered by the compiler to increase performance. It is possible that *generatedValuesCount* has already been incremented,

⁵Java language features are discussed in Section 4.1

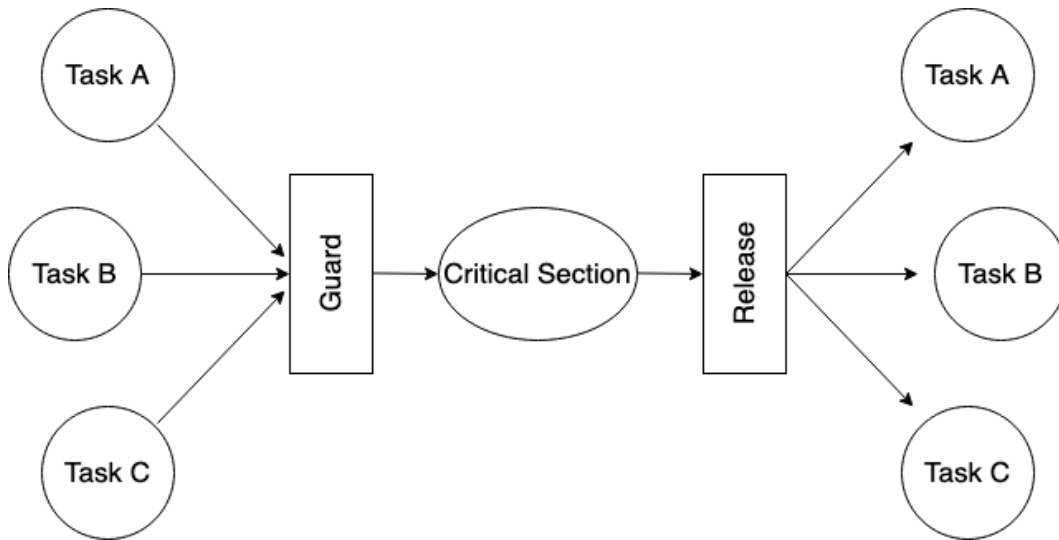


Figure 2.2: Critical section

even before *add* was called. Thus, relying on the order of statement execution to ensure soundness of a program in a concurrent environment is not advisable.

Listing 2.1: Reordering of independent statements

```

1  public void run() {
2      ...
3      var newValue = ...;
4      ...
5      values.add(newValue);
6      generatedValuesCount++;
7      ...
8  }

```

2.2 Concurrent programming primitives

There are many higher-level data structures and functionalities dealing with concurrency issues. Several of them are based upon the same set of — usually hardware supported — programming primitives. These operations are formally defined as functions over shared memory registers. They enable more complex atomic actions beyond mere *read* and *write* operations.

Test-and-set-bit

Probably the simplest routine among concurrent programming primitives is the *test-and-set-bit*. The value of the underlying register can either be 0 or 1. Algorithm 1 illustrates the technique. When *test-and-set* is called, the value of the shared register is set to 1. Its previous value is returned which can then be tested by the caller.

Testing the resulting value can serve as guard for a critical section. When *test-and-set* returns 1, the section is *locked*. Calling code can now actively wait for it to return 0 again⁶. It spins around the register's value and is therefore called *spin lock*. This is shown

⁶This is often referred to as busy-waiting

Algorithm 1 Test-and-set-bit

```

function TEST-AND-SET(r: Register)
    temp  $\leftarrow$  r
    r  $\leftarrow$  1
    return temp
end function

function RESET(r: Register)
    r  $\leftarrow$  0
end function

```

in Listing 2.2. Resetting the register releases the lock.

Listing 2.2: Spin lock

```

...
while (test-and-set(r) == 1) { /* wait */}
/* enter critical section */
...
/* release */
reset(r)
...

```

Compare-and-swap

In theory, the simplistic *test-and-set-Bit* alone suffices to guarantee mutual exclusion. However, there is a more versatile primitive to achieve this, *compare-and-swap*. It also operates on a shared register that can assume an arbitrary finite number of values of an arbitrary data type. The function takes two more arguments, the currently expected value (also referred to as the *witness*) and a new value the register should be set to. The semantics is straightforward: Compare, whether the register holds the expected value. If so, set the register to the new value. Otherwise, the value of the register remains unchanged. Afterwards, return the old value. The depicted implementation is shown in Algorithm 2.

Like test-and-set-bit, compare-and-swap can be used to establish a spin lock. In contrast, it gives rise to use and test against various values. This enables a sort of state machine, where specific concurrent operations can be handled differently. For example, differentiating between *read* and *write* operations becomes fairly easy⁷.

Algorithm 2 Compare-and-swap

```

function COMPARE-AND-SWAP(r: Register, expected: Value, new: Value)
    temp  $\leftarrow$  r
    if r == expected then
        r  $\leftarrow$  new
    end if
    return temp
end function

```

⁷An example implementation

Read-Modify-Write

While *compare-and-swap* provides a flexible way of atomically updating a single value, it does not suffice in situations where compound operations are needed to calculate the new value for a shared register. The primitive *read-modify-write* shown in Algorithm 3 serves this purpose. Instead of a concrete value, it expects a function as argument that uses the current value of the underlying register to calculate its new value. The given function is expected to have no side effects.

Algorithm 3 Read-Modify-Write

```
function READ-MODIFY-WRITE(r: Register, f: Function)
    temp ← r
    r ← f(temp)
    return temp
end function
```

There are other current programming primitives, like *swap* which unconditionally exchanges the old with the new value and returns the former, or *fetch-and-add* working on numeric values adding the given number to the present value. Regardless of the primitive in use, the foundation always is the same, atomicity.

2.3 Synchronization

Synchronization is used in asynchronous environments. To synchronize asynchronously running tasks is to structure and sequence their behavior at certain points. It reduces non-determinism to a bearable degree.

Originally, synchronization stems from multi-threading. In the realm of threads, it serves two main purposes. On the one hand, is joining the execution paths of several co-dependent tasks. This is usually the case when some tasks have to wait for others to make progress or to terminate [BD80]. On the other hand, synchronization is also used to guard critical sections. The former section has shown how to achieve mutual exclusion by using a spin lock. Yet, there is another sort of locks, *synchronization locks*, also referred to as *mutex*. The usage is illustrated in Listing 2.3. They handle locking per thread. Whenever a task running within one thread successfully acquires the lock to a guarded section, every other thread attempting to acquire it, blocks. For that matter these locks and data structures utilizing them are called *blocking*. Notably, many synchronization locks are *reentrant*. When a thread already owns a lock, it can acquire it once again⁸.

Listing 2.3: Synchronization lock

```
DECLARE lock
...
lock.acquire() {
/* enter critical section */
...
} /* automatic release */
...
```

This is the traditional view on synchronization. In a modern, asynchronous system synchronization is primarily involved when a program's progress depends on the result of an asynchronous task. The resulting value has to be *awaited*.

⁸<https://docs.oracle.com/javase/tutorial/essential/concurrency/locksyc.html>

2.4 Semaphores

Perhaps the most fundamental technique used in concurrent programming is the *Semaphore* [Dij68]. The term originates from mechanical signaling systems used on railways to notify train drivers about whether they can go on or have to stop the train temporarily and wait until signaled. Additionally, some semaphores can indicate that a train's speed should be throttled.

In concurrent programming a semaphore is a data type associated with a counter (a non-negative integer) and provides two operations: (1) *wait* decrements the counter, (2) *signal* increments it. The counter represents the number of permits a semaphore can grant at a time. The operation *wait* often expects a positive integer as argument telling how many available permits have to be waited for. A semaphore data type is illustrated in Algorithm 4.

Algorithm 4 Semaphore

```

Semaphore
  counter
  function WAIT(permits: Integer)
    wait until permits available
    counter := counter − permits
  end function
  function SIGNAL
    counter := counter + 1
  end function

```

The use cases for semaphores are manifold. Mutual exclusion can be achieved by allowing only 1 permit at a time. Calling *wait* acquires the *lock*, *signal* releases it. They also enable fine-grained coordination of different tasks. When several tasks are required to progress up to a certain point, one can simply wait for all of them to call *signal* on a shared semaphore.

While semaphores provide an elegant mechanism to deal with different kinds of concurrency-related challenges, they are seldom used in modern projects [PTF⁺15]. This might be due to a lack of knowledge about the possibilities semaphores come with, but also because there are higher-level programming models and abstractions enabling developers to achieve similar results — often in a simpler fashion. The most common and useful models and patterns are discussed in the following section.

2.5 Models and patterns in concurrent programming

Over the last decades, computer scientists have always come up with principles, patterns and models to simplify programming in general, to reduce code complexity and to ease the implementation and understandability of control flow, data structure creation and adaptation. The *Factory* design pattern [GHJV95], the *Inversion of control* principle [Mar96, Rob22], or the *Model-View-Control* scheme [Fow02, HSD10] have made the lives of developers easier.

The discipline of concurrent programming makes no exception to this. All the models discussed in this section follow a common conception: Reduce code segments managing concurrency as much as possible and encapsulate them into a small set of well-developed data structures. This leads to cleaner, better understandable and less error-prone programs that are even more capable of taking advantage of the powers concurrent programming entails.

2.5.1 Message passing

*The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.*⁹

The standard way of establishing task intercommunication is by sharing mutable data structures between them. While this can be safely achieved in a concurrent environment through locking techniques, there is an alternative, *message passing* [Buo21]. A message is an abstract data container of arbitrary content, ranging from simple numbers and strings to complex objects. Tasks communicate by exchanging messages. They might send the result of a computation to another task further processing the data, notify that specific progress has been made, or request data. This is remarkably useful for concurrent programming, because the implementation of each individual task does not need to worry about mutual exclusion on shared mutable state. This only holds true if the messages sent are immutable. Message passing scales also very well to distributed systems, as primary communication mechanism stays the same [Sin97, SS84]. By operating through an abstract message passing routine, sending messages to tasks located on the same node appears to be indifferent from sending them to remote tasks.

Data transfer between tasks can either be handled in two different fashions: (1) synchronously [Hoa78], where the sender waits until the message was received, (2) asynchronously [MK06], where the sender simply continues after the send process has been triggered. Synchronous message passing is only sensible in situations where the sender has to ensure that the message was received right away. This may include a direct answer from the receiver.

Usually, asynchronous message passing is preferable, because it decouples actions of sending and receiving tasks and does not obstruct the sender from further progress. In an asynchronous scenario, messages are enqueued and the receiver processes them sequentially in the order they arrived. When a task receives messages from several senders, it is unknown which message reaches the task first. Thus, developers should not depend on any order [LBL⁺16]. Task communication via message passing is illustrated in Figure 2.3. Note that a task that is the receiver to other sending tasks can likewise send messages.

Channels

Data transfer in message passing is realized via *channels*. A channel is associated with two functions, *send* and *receive*. Depending on the language or library at hand, channels either connect one sender and one receiver (*one-to-one*) or several senders to one receiver (*many-to-one*). Languages like Ada [86512] do even support selective receive on multiple channels. Each possible channel is guarded by a boolean expression. They are tested sequentially in the given order and the first channel which's guard evaluates to true is chosen. The principal of selective receive is depicted in Listing 2.4.

Listing 2.4: Selective receive

```
select
  when guard_i do v <- chan_i
  when guard_j do v <- chan_j
end
```

Message passing and shared state

There is still broad consensus that message passing and sharing state are opposing models [KL94]. But this is a false dichotomy. It has already been proven that these models are

⁹<https://wiki.c2.com/?AlanKayOnMessaging>

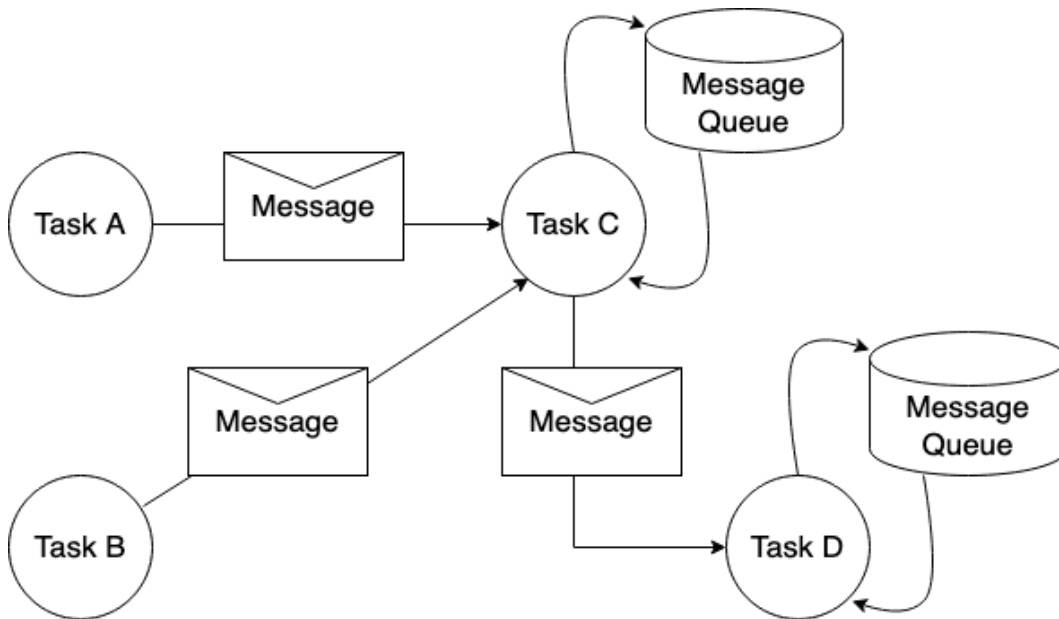


Figure 2.3: Message passing

equivalent [ABND95]. Thus, they can be expressed by one another [DGFG03, Tur13]. Research gives evidence that combining message passing and shared state can improve the performance of a concurrent software system [MNM14]. Moreover, by utilizing the best of both worlds, this hybrid approach raises fault tolerance, simplifies handling of asynchrony, reduces the need for synchronization points, enhances scalability of the software, and alleviates concurrency issues [Nie07, ABDC⁺18]. Due to recent developments in the hardware sector, such as *Remote Direct Access Memory* [KKA14], it is especially of interest for multi-processor intercommunication architectures built upon distributed shared memory clusters [FXL03, CWG09, KJA⁺02].

2.5.2 Main-Subordinate

Main-Subordinate was formerly known as *master-slave*. The latter expression has, for obvious reasons, fallen out of fashion with historically conscious programmers¹⁰. Depending on the literature at hand, it might be termed differently, e.g. *Supervisor-Worker* [MK06]. The principle idea of the main-subordinate pattern is simple: There is one main unit telling an arbitrarily large amount of subordinates what to do next [Bus95]. Once a subordinate unit has finished the work, it reports the results of the performed computations to the main unit. Figure 2.4 illustrates these relations. A command sent by the main unit can contain data of various types.

A software system utilizing the main-subordinate pattern benefits from increased modularization. It can be used to partition heavy workload into smaller chunks. These are then distributed by the main unit to the respective subordinates. This approach enables developers to run subtasks of *divide-and-conquer* algorithms [Dix22] concurrently and in parallel. There are several frameworks for conveniently implementing this methodology [Lea00, NS08]. They automatically assign subtasks — also referred to as jobs — to different threads, ensuring (close to) optimal usage of the systems computational resources. Thus, developers can focus on an algorithms' logic.

¹⁰<https://linux.slashdot.org/story/20/07/11/2037250/the-linux-team-approves-new-neutral-terminology>

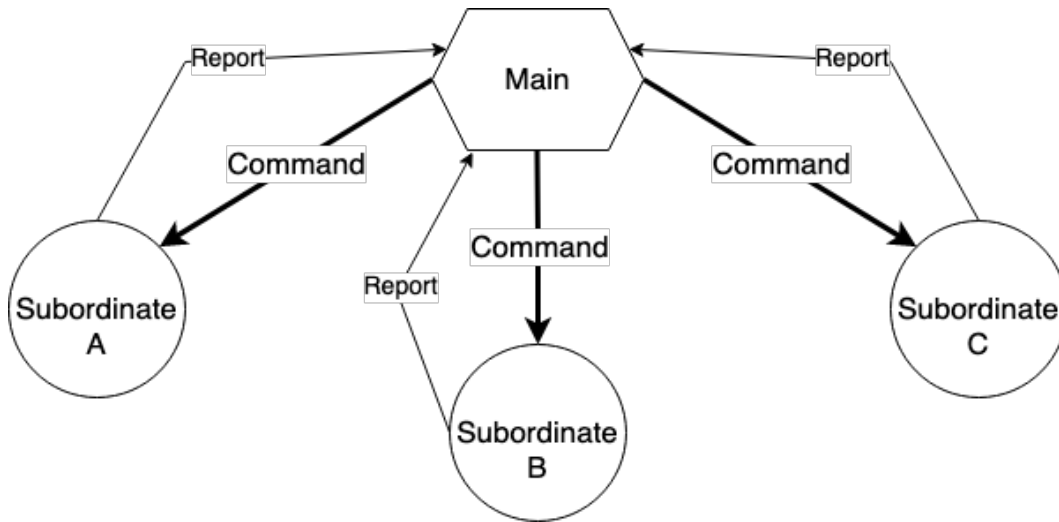


Figure 2.4: Main-Subordinate

Especially in cases where the execution times of tasks might differ markedly, making use of main-subordinate can come in handy. It makes dynamic configuration of a task, such as setting a timeout or assigning it to a specific thread pool in, fairly easy. It is also useful in situations where several solutions to a problem exist, all optimized for certain cases at the expense of others [BPR22]. Main-subordinate enables orchestrating the various solutions such that each can benefit from intermediate results of the others.

Some systems, like databases and multimedia applications, use it to replicate data of parts of or of the system as whole. Each replication represents a subordinate [Str14, SG97]. Main-subordinate is by design well-suited for distributed software systems. Technologies like REST and GraphQL make communication between main and subordinates a breeze. When a subordinate node fails, it can most likely be reloaded by the main, but in case of failure on the main node, failover protocols like leader election have to be employed [Str14].

2.5.3 Producer-Consumer

Another way of compartmentalizing software is the *Producer-Consumer* pattern. A producer executes arbitrary calculations — this may likely include calls to remote APIs — to produce data. A consumer is handed over the data to do other processing on in. Producers and consumers can be related to each other in any way possible: (1) one-to-one, (2) one-to-many, (3) many-to-one, and (4) many-to-many.

Usually, queues are used to pass data from producers to consumers [SLS09]. In an implementation following this convention, access to the queue, that is, placing data in and retrieving it from the queue, is the only place where synchronization is involved. Great effort has been put in the design and implementation of concurrency-ready queues [MS96]. They guarantee thread-safe, efficient data handling.

Alternatively, a producer-consumer system can be implemented via messages. This might increase flexibility of the software. But these two approaches are hardly distinct. As depicted before, message passing also involves queues most of the time. In fact, a distributed producer-consumer system is likely using remote message queues to pass data from producers to consumers. Figure 2.5 illustrates the described scheme.

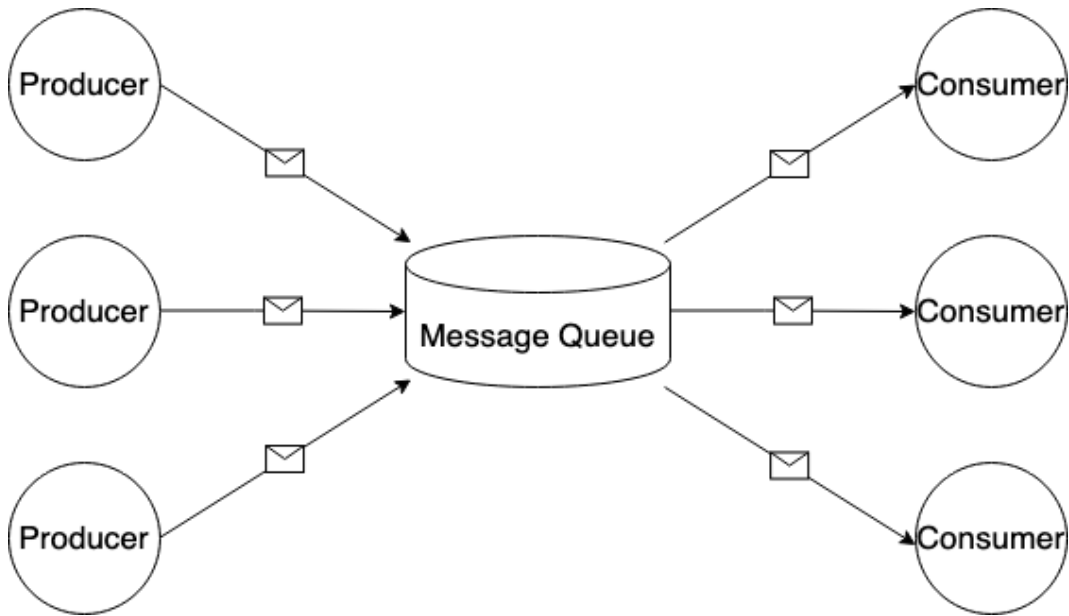


Figure 2.5: Producer-Consumer

Publisher-Subscriber

The *Publisher-Subscriber* pattern is a special case of producer-consumer. Under arbitrary circumstances, a publisher propagates data to its subscribers. By subscribing to a publisher, a subscriber consents the way of passing data to it. This process is also referred to as *registration*. By default, a subscriber registers one of its procedures which is being called upon publish. Usually, one producer manages subscriptions of several subscribers. At the same time, a subscriber might register to different publishers.

Subscriptions can also be cancelled. This is an integral part of the pattern, as it allows subscribers to only receive published data conditionally, being in a certain state of progress. Dynamically subscribing and unsubscribing extends flexibility of the whole system.

Event-based systems An event is the notification that *something* has happened. Usually but not exclusively, they are triggered due to user interactions. Events can simply notify of certain state changes, but they may also carry arbitrary data. Event-based systems can elegantly be implemented via the publisher-subscriber pattern. For instance, this is how JavaScript's native events are handled¹¹. To trigger an event, the publisher refers to its respective subscriptions.

2.5.4 Actor model

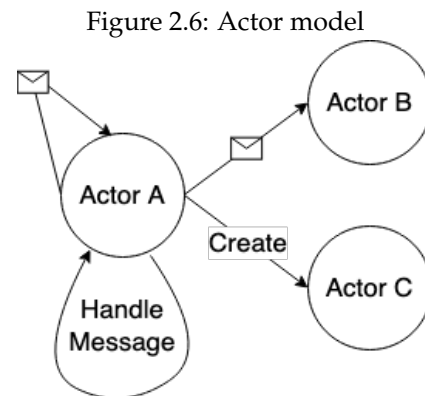
The *Actor model* is a mathematical theory of computation that treats "Actors" as the universal primitives of concurrent digital computation [Hew77]. The previously described patterns were all designed with Von Neumann computer architectures [OAE07] and with mostly object-oriented programming [GHJV95] in mind. The *Actor model*, however, is a completely independent computational model of its own. Actors are the fundamental units of computation. As Carl Hewitt puts it, an actor embodies 3 essential elements [HBS73]:

¹¹See Section 4.3

- The capability of processing
- A storage to keep the internal state
- The capability of communication

*One ant is no ant.*¹² The same holds for actors. One actor alone is hardly capable of achieving anything. They come in systems.¹³ Within these systems an actor can do one of the following: (1) create other actors, (2) send messages to actors it is connected with, (3) designate how to handle the next message received [Hew10]. These actions could be executed concurrently. A sample Actor model is illustrated in Figure 2.6. As portrayed, an actor may send messages to itself.

In terms of actors, being connected to each other means that an actor *knows* the address of the other one, and vice versa. This could be a concrete memory address, but also a network address or any other way allowing to refer to an actor. The address should not be confused with the identity of an actor. In fact, one address could refer to several actors at the same time. Furthermore, many addresses could likely belong to one and the same actor.



Message passing in the Actor model

The theory of the Actor model does not specify how messages are exchanged between actors. Message passing is simply taken as fundamental part of the Actor model [Hew10]. Technical aspects like message queues and channels are completely left aside. Nonetheless, the behavior of message passing in the Actor model is well-defined. Messages are sent asynchronously, thus, actors do have to wait until a message is received. Delivery of messages happens on best effort basis, meaning a message is received *at most once*. It may take arbitrarily long to send a message to an actor. One cannot make any assumptions on the order of message delivery. It is possible to send two messages A and B sequentially, with B received before A.

Modularity and concurrency

Actors do solely interact via message passing and messages are processed sequentially. These properties make the Actor model appealing for concurrent programming, as well as distributed systems. Moreover, due to its inherent modularity the Actor model is well-suited for composing large, flexible structures [Agh86]. For instance, *Producer-Consumer* or *Main-Subordinate* can easily be simulated by the Actor model.

[...] I'll say that in a nutshell, the actor approach is about the future of computing [Akm90].

¹²Edward O. Wilson, https://de.wikipedia.org/wiki/Edward_O._Wilson, accessed 16. May 2022

¹³Hewitt, Meijer and Szyperski: The actor model, https://www.youtube.com/watch?v=7erJ1DV_Tlo, accessed 03. March 2022

Practical challenges of concurrent programming

The previous chapter explained principles, routines and methodologies that can be employed in concurrent programming. Throughout the descriptions, it is stated at several points how challenging concurrent programming can get. This chapter covers challenges developers have to face when implementing software systems involving concurrency. It is thereby divided into three main parts. First, an example of such a system is portrayed showcasing several scenarios of concurrent programming. It is followed by a section discussing correctness and concrete errors which may arise during execution of concurrently running tasks. The third section is dedicated to decision-making, which in this context means deciding between different techniques depending on the needs of the problem at hand.

To simplify matters, this chapter often uses the term *resource*. A resource denotes shared state like the instance of a data structure, a file on the system, a task control structure, such as a lock or semaphore, or even a network port.

3.1 A Messenger - an introductory example

A lot of applications people use on a daily basis incorporate concurrency. It is pretty likely to find messenger programs among these applications. Consider such a messenger like the one portrayed in Figure 3.1. One can write messages in groups as well as to a single person. Besides regular text, messages can also contain markup and whole files. It is possible to reply to a message from a group chat in a separate thread. Therefore, another chat view is opened to the side. Typically, views for *Options* or *Tools* are provided via modal windows.

An implementation of the depicted application involves concurrent programming at several places. In fact, without it, the user experience would be anything but great. The following paragraphs shed light on that.

Responsive user interface Graphical user interfaces, usually simply referred to as UI, always imply concurrency. To understand why that is the case, image the whole application to be executed in a single thread. Whenever user interactions trigger longer running actions, e.g. logging in, sending a message, loading a file, or choosing some customizations, the UI would seem to be frozen. Clicking a button or a menu item would appear to have no effect. Thus, to establish reasonable interaction of the user with the application, such

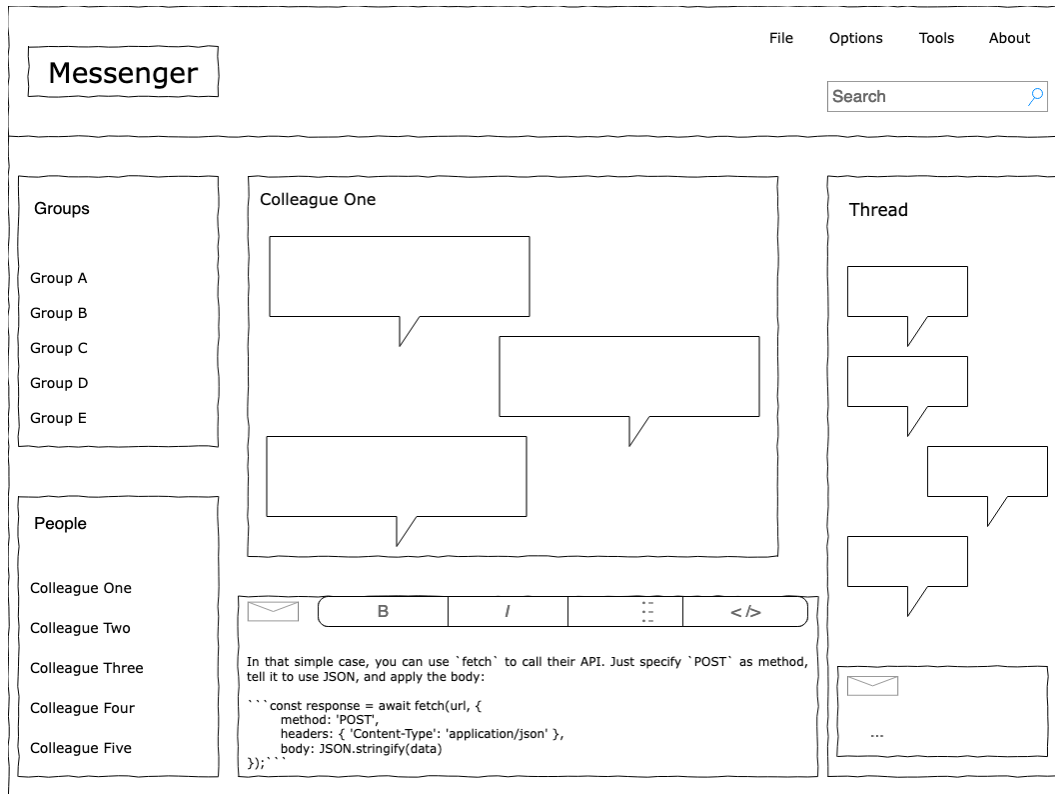


Figure 3.1: Messenger application

operations have to run in background tasks. Users should not notice anything about the heavy lifting that is going on behind the scenes. The only thing they should see are the effects resulting from the completion of those tasks, like a changed color theme or the next message appearing in the chat history.

For these reasons, UI frameworks usually run the graphical user interface and all its related updates in a separate thread. This keeps the UI *responsive*.

Message input The primary intention of a messenger application is the exchange of messages. Especially those applications to be used in companies want to enable the user to draw from ample opportunities. Therefore, messages can (and will) contain arbitrarily formatted text, code snippets, audio and video material and files like images. They are backed by complex data structures which are, for obvious reasons, subject to change. Different tasks may concurrently interact with and alter these data structures. For instance, consider a file being loaded. Modern UIs will likely add a placeholder to indicate the file is still being processed. While this proceeds, the user can also continue to write. Furthermore, the user might switch back and forth between several unfinished messages.

Receiving messages The interconnectedness to other people using the same messenger adds another layer of complexity which entails even more concurrency. One might receive multiple messages in several groups from different peers at the same time. The data structures backing the respective chat histories have to manage concurrent access, while the UI also has to be kept up to date. Mutual exclusivity for writing operations must be granted

and separate tasks working in the same context have to be made aware of changes. Otherwise, messages might get lost or the appearance on the UI might be inconsistent.

Persistence On top of everything mentioned beforehand comes data storage. All the information the application constitutes must be persisted. Data only relevant to the particular instance of the application are stored locally. Everything else, including chat histories, information about users and logged material, is usually kept on a remote server. Server applications and databases are themselves subject to concurrency as they interact with hundreds of thousands local of systems.

3.2 Potential for errors

Synchronization errors [...] are among the most difficult programming errors to detect, reproduce, and eliminate [BLR02]. Developing software is a difficult task. Even more so is the development of correct concurrent software. Correctness in terms of concurrency goes beyond the proper implementation of semantics. It is divided into two topics, *safety* and *liveness*. The following sections give an overview of them and explain related error potentials.

3.2.1 Safety

A *safety property* is one which states that something will not happen [Lam77]. In concurrent programming, all safety properties refer to the interleaved executions of concurrently running tasks. Regardless of the scheduling and suspension of tasks and the order in which they enter certain (critical) sections, all performed actions have to take effect with their respective invariants upheld.

The safety property used most often for concurrent programs is *linearizability* [HW90]. A data structure is linearizable if each invocation of its operations appears to take effect instantaneously at some point. Thus, it is possible to construct a sequential history of operation invocations which is equivalent to the concurrent history and results in the same state the data structure currently assumes.

Several possibilities to violate the safety properties of a concurrent program are discussed subsequently.

Data races

Recall the characteristic of atomic operations: Their execution is not interruptible and therefore everything happening inside them appears to take effect at once. Atomicity is an indispensable property of programming primitives like *compare-and-swap* discussed in Section 2.2. Yet, most operations are not atomic. Thus, a task can be suspended amidst their execution and several tasks might also execute them simultaneously.

Consider the algorithm illustrated in Algorithm 5. It uses an external function, *CreateTasks*, to spawn as many new tasks as is defined in *numTasks*. Each task runs the *Increment* function which loops *numCycles* times. In each iteration the variable *counter* is incremented. The expected result, namely, the final value of *counter* is given by the equation $4 \times 1000 = 4000$.

Note that *counter* is shared among the tasks. Since the statement *counter* += 1 is not atomic, two tasks, A and B, might now load the same value for it, say 20. Loading is done locally, thus each task holds a copy of the respective value. A and B both increment the fetched value and they both write back 21. This is called a *data race*. Effectively, one increment step is lost.

Using the atomic operation *fetch-and-add* would suffice to get rid of data races in this simple case. More complex scenarios require additional synchronization techniques. In general, compound operations executed on shared state in a concurrent environment have to be guarded properly.

Algorithm 5 Data races in non-atomic operations

DECLARE FUNCTION *CreateTask*

INITIALIZE

const numTasks := 4

const numCycles := 1000

counter := 0

function INCREMENT

for *i*=0,..., *numCycles* **do**

counter + = 1

end for

end function

for *i*=0, ..., *numTasks* **do**

CreateTask(*Increment*)

end for

Deadlocks

Separate concurrent operations might acquire the same resources in different order. This can lead to *deadlocks*. Listing 3.1 and Listing 3.2 depict such a situation: Task A tries to acquire *lockA* first, and subsequently also tries to acquire *lockB*. Task B does the same conversely. Both hold a lock the other one needs to proceed. Thus, no further progress can be made by either of the tasks.

Listing 3.1: Deadlock - Task A

```
// acquire lockA
lockA.acquire()
...
// acquire lockB
lockB.acquire()
...
// release the locks
lockB.release()
lockA.release()
```

Listing 3.2: Deadlock - Task B

```
// acquire lockB
lockB.acquire()
...
// acquire lockA
lockA.acquire()
...
// release the locks
lockA.release()
lockB.release()
```

Deadlock detection Boyapati, Lee and Rinard [BLR02] give a formal definition of deadlocks: A deadlock occurs when there is a cycle of the form: $\forall i \in \{0..n-1\}$, Thread_{*i*} holds Lock_{*i*} and Thread_{*i*} is waiting for Lock_{*i+1 mod n*}.

A simple code examination would reveal the obvious deadlock potential in the example above. More complex cases involving cycles of resource acquisition, however, make the analysis much harder. To detect deadlocks, several algorithms have been developed [Elm86]. Essentially, concurrent resource acquisition is represented by a directed graph and deadlock detection algorithms search for cycles within these graphs.

Deadlock avoidance Deadlock detection algorithms are employed in a running system to recover from deadlocks. This, however, enforces that either, operations are abortable, or that the system can release a resource held by a task, suspend it, and later on reassign ownership of the resource to the task before its execution continues.

Deadlock recovery entails overhead, may not be feasible to achieve and is often not even necessary. Recall one of the primary intentions of the concurrent programming models discussed in Section 2.5, centralizing and encapsulating concurrent interactions. This significantly reduces deadlock potential. Another way good software design can aid to eliminate deadlock occurrences is by conceptualizing concurrent operations as closed and as compact as possible. Avoid holding several resources at the same time, if practicable.

Visibility

Threads can cache values of shared variables locally. Thus, updates made by other threads (and tasks running within them, respectively) may not be visible. For this concern, programming languages usually provide mechanisms to ensure a consistent view on shared data. In Java, for instance, a variable may be declared *volatile* [GBB⁺06]. Values of these variables are not being cached thread-locally.

This is often referred to as weak synchronization. While it ensures visibility of updates among threads, it doesn't prevent data races.

3.2.2 Liveness

*You get safety alone by doing nothing at all*¹. Safety alone doesn't suffice for a program to be correct. Liveness properties stipulate, that something good happens [AS85]. This means that a task eventually performs its specified operations. Violation of liveness properties can lead to one of the following problems.

Starvation A task is said to starve if it does not succeed to acquire a specific resource at all. Although quite unlikely, with an unlucky scheduling, a *spin lock* might loop forever. This is the case when *compare-and-swap* returns unsuccessful each time.

Livelocks A livelock occurs when several tasks try to get out of each other's way. They repeatedly backoff from their current work in order for the other tasks to proceed. Listing 3.3 and Listing 3.4 illustrate a naive deadlock recovery strategy. Both tasks, A and B, try to acquire the second lock. When this attempt fails, they both release the respective other lock and restart from the top. Thus, they get stuck in an endless loop as the break condition is never reached.

Livelocks are similar to deadlocks and, indeed, one could argue that deadlocks do also belong to liveness instead of safety. Yet, Alpern And Schneider [AS85] rank deadlock-freedom among safety properties. This makes sense, because tasks in a deadlock do not progress at all. Nothing beyond the attempt of acquiring the resource is happening. Tasks stuck in a livelock on the hand still proceed in some sense. After successfully acquiring *lockA*, Task A could still execute arbitrarily many other statements before attempting to acquire *lockB*.

¹Michel Scott, Nonblocking data structures, Summer school on Practice and Theory of Distributed Computing (SPTDC), July 2019, St. Petersburg

Listing 3.3: Livelock - Task A

```

done = false
while (!done) {
    // acquire lockA
    lockA.acquire()
    ...
    // try to acquire lockB
    success = lockB
        .tryAcquire()
    if (!success) {
        lockA.release()
        continue
    }
    ...
    done = true
}

```

Listing 3.4: Livelock - Task B

```

done = false
while (!done) {
    // acquire lockB
    lockB.acquire()
    ...
    // try to acquire lockA
    success = lockA
        .tryAcquire()
    if (!success) {
        lockB.release()
        continue
    }
    ...
    done = true
}

```

Levels of liveness

There are three levels of liveness in a software system, defining to which degree a program or specific algorithm adheres to liveness properties:

- Starvation-free: Every operation is guaranteed to complete in a bounded number of steps [Her96].
- Livelock-free: An operation is guaranteed to complete in a bounded number of steps [Her96].
- Obstruction-free: If an operation gets to run all by itself, it is guaranteed to complete in a bounded number of steps [HLM03].

Most of the time, livelock-free or even obstruction-free approaches suffice for regular applications. Especially the latter can sometimes be implemented amazingly simple and elegant [Sco13]. Starvation-free data structures introduce overhead due to their rigid guarantees regarding execution time and latency. Yet, in some cases, software has to grant starvation-freedom. Consider a commercial real-time trading system. Asset courses, like for stocks or foreign exchange, might change in millisecond intervals. Professional traders rely on almost instantaneous updates within their trading software.

3.3 Decision-making

Choosing the right operations and data structures for the task at hand is a vital part of software development. The best choice for one situation may be insufficient for another. It can have a great impact on code complexity and performance. Concurrent programming makes no exception to this.

Guaranteeing safety and liveness properties is only one side of the coin. In order to develop a scalable concurrent program, it is also important to understand which synchronization technique to prefer in which scenario.

Blocking and Non-blocking data structures Concurrent data structures incorporating synchronization can roughly be divided into two sorts: The first are blocking data structures [Sco13]. They block a thread when an operation cannot be executed at the time of invocation. This is the case when an attempt to acquire a lock fails or when other conditions are not met, for instance, when the maximum internal capacity is reached.

The second are non-blocking data structures [SS04]. As the name implies, these do not block threads. They use atomic primitives to guard critical sections and notify the caller when an operation cannot be carried out. Non-blocking solutions are preferable in systems that schedule many tasks within threads. Blocking the whole thread would either prevent every task associated to it to be suspended or would require to reassign them to other threads². Context switching should also be considered. The higher the expected costs are, the better it becomes to use non-blocking data structures.

Blocking data structures on the other hand are well-suited for specific *Producer-Consumer* implementations. Consider a scenario with many producers and only one consumer. A queue is used to transfer data. To avoid running short on memory the queue has a maximum capacity. It is therefore called a *bounded* queue. Once the limit is reached, blocking the producers can be advantageous.

Instead of going solely one way or the other when choosing a concurrent data structure, there is also an alternative. Some implementations use a hybrid approach: They provide both blocking and non-blocking operations. The exemplary code fragment shown in Listing 3.3 makes use of this approach. The call of *lockA.acquire()* is a blocking operation. Invoking *lockB.tryAcquire()* on the other hand is non-blocking. Data structures employing the hybrid approach can take advantage of the benefits of both techniques. It also increases reusability.

Number of threads An intricate part of the implementation of concurrent software is the decision on how many threads to create. This depends on two factors: (1) the number of processor cores, (2) the blocking factor. The latter describes the ratio between computational operations on the CPU and I/O bounded operations. For computational intensive tasks only a few threads should be used, whereas for I/O intensive tasks many more threads may be created.

As a rule of thumb, the number of threads to be used can be estimated as follows: $numThreads \leq numCores \div (1 - blockingFactor)$ with $0 \leq blockingFactor < 1$

Message passing and shared state - revisited As already discussed in Section 2.5.1, the conviction that message passing and shared state are opposing concepts is a false dichotomy. Software systems can benefit from utilizing both. Frequent read and write operations on simple data structures are likely to perform better on shared state. Versatile synchronization structures, such as *read-write locks*, can even increase this advantage. Furthermore, it might also be simpler to implement.

Message passing on the other hand improves the code structure through enhanced modularization. With proper abstractions, the difference between communication within the same node and among several nodes in a distributed environment can be neglected. This methodology is often used in sophisticated actor model implementations.

²See Section 4.4

CHAPTER 4

Languages

In order to demonstrate concurrent programming, I have chosen four languages due to different characteristics and specialities in handling concurrency. First up is Java as it is one of the most widely used programming languages nowadays. Its standard library provides highly sophisticated concurrent data structures. The second one is Rust, a systems programming language. Besides being fascinating of its own, it is especially interesting for concurrent programming due to its ownership model.

JavaScript was chosen as third language to stress the fact that single-threaded languages also involve concurrency. Furthermore, JavaScript's *Promise* type enables simple handling of asynchronous programming. Go is the last language discussed. It has the *Actor model* built-in and is therefore a great choice when implementing a highly concurrent system.

Blueprints like the *Consumer-Producer* pattern or the *Actor model* build a solid foundation for concurrent programming. Still, the language is use poses specific implications on the way software is written. Likewise, does the runtime — and, if given, specifications of hardware resources. Thus, these models should always be taken with a grain of salt. For instance, while the *Actor model* in theory simply assumes message passing as given, a concrete implementation has to use some sort of intermediary data structure.

Furthermore, concurrent programming is not solely about *Safety* and *Liveness*. It also incorporates the creation and lifecycle management of tasks. Many programming languages approach this similarly. The section on Java covers foundations of multi-threading in detail. The other sections focus on aspects the respective language adds to concurrent programming.

For each of the selected languages, in regard to their respective characteristics, one or several substitutes can be found. Programming languages like C#, Julia or Elixir (and many others) do also provide great concurrent programming features. However, since discussing several languages with similar approaches to concurrent programming would bloat this chapter without adding much value, these alternative languages will not be featured.

While all data structures and operations are discussed thoroughly, and the corresponding examples are explained in detail, a decent understanding of the respective language is advantageous.

4.1 Java

By the time of this writing, Java ranks among the most popular programming languages¹. It is an object-oriented, general purpose programming language. In the last decade, advanced programming techniques like lambda expressions² or method references found their way into the language, and it progressed rapidly overall.

Within the package *java.util.concurrent*, Java provides highly sophisticated concurrent data structures. Many of them employ higher-order functions. To support these, Java introduces the notion of functional interfaces. A functional interface declares a single method. This way, calling code can simply supply a lambda expression to fulfill the implementation of such an interface.

Listing 4.1 depicts the declaration and usage of the functional interface *Callable*. It declares a single method, *call*, which returns an instance of the given data type. Instead of providing a concrete or anonymous implementation for it, a lambda expression denoted by an arrow is used. Note that functional interfaces are usually annotated with *@FunctionalInterface*.

Listing 4.1: Callable

```
@FunctionalInterface
public interface Callable<V> {
    V call();
}
...
int execute(Callable<Integer> callable) { ... }
var executionResult = execute(() -> 1);
...
```

The following sections discuss concurrent programming in Java thoroughly. At the end a brief introduction to Scala's concurrency features is given, as Scala is a functional programming language executed on the JVM.

4.1.1 Thread

The foundation of concurrent programming in Java builds the class *Thread*. Each instance of a thread is directly coupled with an operating system thread. The class *Thread* provides various methods to retrieve information about it and to handle its lifecycle. The state a thread currently assumes, for instance, can be obtained by calling *getState*. While a thread can also have a name, identification should usually be done via the *getId* method which returns a thread's unique identifier.

Runnable At the very core of code execution within a thread lies the interface *Runnable*. It declares a single method, *void run()*, and is therefore a functional interface. The class *Thread* implements *Runnable*. Yet, without an external implementation, the thread object will execute nothing. Hence, on thread instantiation, a *Runnable* can be supplied as argument. Listing 4.2 illustrates this. Upon creation, a thread assumes the state *NEW* (corresponding to ready). To begin executing a thread, one simply calls *start* on it. The thread then assumes the State *Runnable* (corresponding to running).

Listing 4.2: Instantiation of a thread with a Runnable

```
...
var thread = new Thread(() -> { ... });
```

¹<https://www.tiobe.com/tiobe-index/>

²The lambda calculus is, indeed, not new at all.

```
thread.start();
...
```

Thread interruption Already in the early days of the Java programming language³, the method *stop* of *Thread* has already been deprecated. And for good reasons: Invoking *stop* on a thread will cause the thread to terminate immediately, aborting the execution of whatever operation it performed. This may lead to an inconsistent state of the respective data structures the thread was operating on⁴.

This entails unbearable error potential. Thus, a cooperative approach has to be used in order to properly interrupt threads. Consider two threads, A and B. Thread A wishes to shut down Thread B. Instead of abnormally aborting its execution, A politely tells B to finish. Once B notices the intended interruption, it may clean up its state and gracefully return.

In Java, there are two standard ways of achieving this. On the one hand, it can be achieved via a thread's lifecycle. By calling *interrupt* on the thread, a simple flag within the thread's state is set. The thread can obtain whether it should interrupt its current execution through the method *isInterrupted*. A thread noticing an intended interrupt may throw an *InterruptedException*. This approach is primarily suited for library code.

On the other hand, interrupting threads within client code might be much simpler to handle. Global (at least in a thread's context) instances of configuration variables can be checked occasionally to see whether a thread should terminate its execution. Likewise, arbitrarily many other state changes may be indicated.

Groups and security Threads can be collected in groups. This is done through the class *ThreadGroup*. A *ThreadGroup* can contain threads and other instances of *ThreadGroup*. This way, they build tree structures. Grouping threads eases handling and lifecycle management of the contained threads. For instance, the method *interrupt* can be invoked on the whole group.

A *ThreadGroup* constrains access to the threads it contains: Only threads within the same group may interact with each other. A thread can obtain whether it is allowed to modify another thread by calling the method *checkAccess*. Limiting the access to threads can increase the security of a system. Certain critical operations may be restricted to threads in a specific *ThreadGroup* and thus inaccessible to threads outside the group.

4.1.2 Foundations

Synchronization

Every object in Java has an intrinsic lock. It is acquired through the keyword *synchronized*. A failed attempt to acquire this lock blocks the current thread. All threads blocked by the lock are kept in a queue and one after another is unblocked whenever the lock is released again. The order in which threads will be unblocked, however, is non-deterministic and can therefore not be relied upon. The acquisition of an object's intrinsic lock is depicted in Listing 4.3.

Listing 4.3: Intrinsic object lock

```
...
Object lock = new Object();
...
```

³The *stop*-method is deprecated since Version 1.2

⁴<https://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html>

```
synchronized (lock) {
    ...
}
```

Associated with the intrinsic lock of an object are the methods *wait*, *notify*, and *notifyAll*. They can be used to further limit access to shared state. If certain conditions are not met, *wait* can be invoked to set the current thread into the *WAITING* state. Calling *notify* wakes up a single thread, *notifyAll* wakes up all threads waiting on the same object.

Intrinsic locks can be used to grant mutual exclusion on critical sections. Combined with conditional waiting, concurrent data structures can be implemented that do not only block when a lock cannot be acquired, but also conditionally.

Consider a *Producer-Consumer* scenario. Producers and consumers need an intermediate to transfer data. To avoid running short on memory, a buffer of limited size can be used. Due to the limited capacity, threads attempting to put an element into a full buffer have to wait until space is available again. Likewise, threads attempting to take an element from an empty buffer have to wait as well. An implementation of such a buffer is given in Listing 4.4.

Listing 4.4: BoundedBuffer

```
public class BoundedBuffer<T> {
    private final T[] buffer;
    private int head, tail, count;

    public BoundedBuffer(int capacity) {
        this.buffer = (T[]) new Object[capacity];
    }

    public synchronized void put(T element) throws InterruptedException {
        while (this.count == this.buffer.length)
            wait();
        this.buffer[tail] = element;
        this.tail++;
        this.tail %= this.buffer.length;
        this.count++;
        notifyAll();
    }

    public synchronized T take() throws InterruptedException {
        while (this.count == 0)
            wait();
        T element = this.buffer[head];
        this.buffer[head] = null;
        this.head++;
        this.head %= this.buffer.length;
        this.count--;
        notifyAll();
        return element;
    }
}
```

The class *BoundedBuffer*⁵ is based on a single array. The variables *head* and *tail* are used as ring counters, pointing to the first and last index respectively where elements can be found within the array. *BoundedBuffer* provides the thread-safe methods *put* and *take* to interact with the underlying array. Instead of an additional object, the intrinsic lock of the

⁵This implementation is inspired by *Java Concurrency in Practice*, Bryan Goetz, Chapter 14. [GBB⁺06]

instance itself is used to guard the methods. Declaring a method *synchronized* is syntactic sugar for wrapping the whole method body into a *synchronized* block.

Both methods employ the strategy of conditional waiting. While the buffer is full or empty, respectively, the method *wait* is invoked. The loop is necessary because threads may wake up spuriously, that is, for no comprehensible reason. Thus, re-evaluating the condition on which the thread was set to wait is inevitable. Usually, however, a thread wakes up due to being notified. A thread waiting in the *put* method will react upon the call to *notifyAll* within the *take* method and vice versa.

Thread execution lifecycle

Some operations in the Java standard library, such as *wait*, rigidly examine whether the current thread was interrupted. In that case, they throw an *InterruptedException* and reset the interrupted flag of the respective thread. In the context of the class *BoundedBuffer* from Listing 4.4, handling an interrupt does not make sense. Thus, the exception is propagated upwards. Calling code should take further action when a thread interruption occurs. It is bad practice to swallow the exception. Logging it does also not provide any benefit to the running system. At least, client code can set the interrupted flag again. This enables code further up the call stack to cancel the respective task gracefully.

Listing 4.5 depicts a scenario in which a consumer thread uses an instance of *BoundedBuffer* to continuously retrieve new elements. It catches an occurring *InterruptedException*, polls the current thread and resets its interrupted flag. Furthermore, the consumer recurrently checks whether to continue its work through the *running* flag⁶. The main thread occasionally examines the interrupted state of the consumer thread. The combination of thread state with custom control instructions enables fine-grained control over the execution lifecycle of threads.

Listing 4.5: Reacting on interruption

```
...
volatile boolean running = true;
var boundedBuffer = new BoundedBuffer(capacity);
...
var consumerThread = new Thread(() -> {
    while (running) {
        try {
            var nextElement = boundedBuffer.take();
            databaseService.write(nextElement);
        } catch (InterruptedException ie) {
            Thread.currentThread().interrupt();
        }
    }
});
consumerThread.start();
...
while (true) {
    ...
    if (consumerThread.isInterrupted())
        running = false;
    ...
}
```

⁶Note that *running* is declared volatile. This ensures visibility among threads.

Concurrent control flow using semaphores

The semaphore, as mentioned in Section 2.4, is one of the most fundamental concurrency control structures. With the class *Semaphore* Java provides a profound implementation. Semaphores take a similar route towards synchronization as do objects with their intrinsic lock paired with conditional waits.

However, Semaphores span another area of concurrent programming, concurrent control flow. Consider an algorithm that finds all occurrences of a specific element in an array. The array is expected to be large. Thus, the algorithm uses multithreading to parallelize the computation. The array is split into several chunks of about equal size. Each part is then traversed by a different thread. A sample implementation is given in Listing 4.6.

Listing 4.6: The method `searchOccurrences`

```
public static <T> int searchOccurrences(
    T[] items, T searchTerm, int numThreads) throws InterruptedException
{
    var numOccurrences = new AtomicInteger(0);
    var readySem = new Semaphore(0);

    var workerPool = new ArrayList<Runnable>();
    final int numItems = items.length / numThreads;
    var lastExclusiveEnd = 0;
    for (int i = 1; i < numThreads; i++) {
        final int start = lastExclusiveEnd;
        lastExclusiveEnd += numItems;
        workerPool.add(createSearchWorker(
            items, searchTerm,
            start, lastExclusiveEnd,
            numOccurrences, readySem));
    }
    // Create the last worker with the remaining items
    workerPool.add(createSearchWorker(
        items, searchTerm,
        lastExclusiveEnd, items.length,
        numOccurrences, readySem));

    for (var worker : workerPool) {
        new Thread(worker).start();
    }

    readySem.acquire(numThreads);
    return numOccurrences.get();
}
```

The static method `searchOccurrences` expects the array, *items*, and the element, *searchTerm*, to be searched for. The third parameter, *numThreads*, indicates how many threads to use. A list of instances fulfilling *Runnable* – further referred to as *workers* — is built through the helper method `createSearchWorker`, discussed below. Besides the array and the search element, it expects the start and end index of the respective array part. Indices are simply computed by using the last exclusive end index as the next start index. Adding up the number of items gives the next exclusive end index. It further expects instances of an *AtomicInteger* and a *Semaphore*. In the main thread, the semaphore is used to wait on the completion of each worker by invoking `acquire(numThreads)`. This method call might throw an *InterruptedException*.

AtomicInteger is one of several data structures in Java which provide atomic operations. The instance *numOccurrences* is used by the workers to collect the number of occurrences

found. Due to its atomicity, no further mechanism is needed to grant mutual exclusion.

The method *createSearchWorker* returns an instance of *Runnable* via a lambda expression. The range from the start to the end index is traversed. For every occurrence of the search element the atomic operation *getAndIncrement* is invoked. As the name implies, it returns the current value and increments the internal value. The result is of no particular value⁷ here.

The last step is to release the semaphore. Note that the code is enclosed in a *try-finally* clause. This ensures release of the semaphore in case of an exception being thrown. Otherwise, a thread might not release the semaphore and the program would not terminate. It is advisable to use this approach whenever working with data structures that are held exclusively by a thread and need to be released in some way.

Listing 4.7: The method *createSearchWorker*

```
private static <T> Runnable createSearchWorker(
    T[] items, T searchTerm,
    int start, int end,
    AtomicInteger numOccurrences, Semaphore readySem
) {
    return () -> {
        try {
            IntStream.range(start, end)
                .forEach(index -> {
                    if (items[index].equals(searchTerm)) {
                        numOccurrences.getAndIncrement();
                    }
                });
        } finally {
            readySem.release();
        }
    };
}
```

Exception handling

Being simple and straightforward, the implementation of *createSearchWorker* is correct. Yet, this only holds when the start and end indices are correct, that is, they are inside the array's bounds. If this is not the case, calling *items[index]* will eventually throw an *IndexOutOfBoundsException*. Note that the respective thread will terminate abruptly.

To handle uncaught exceptions thrown by separate threads, developers can specify an *UncaughtExceptionHandler*. This interface declares a single method, *uncaughtException*, which expects an instance of *Thread* and one of *Throwable*. *Throwable* is the base class for all errors and exceptions in Java. An *UncaughtExceptionHandler* can be registered per thread via the method *setUncaughtExceptionHandler*. Listing 4.8 illustrates this. Through *setDefaultUncaughtExceptionHandler* a default exception handler can be specified which is used when no other handler has been defined for the respective thread.

Listing 4.8: Specifying an *UncaughtExceptionHandler*

```
for (var worker : workerPool) {
    var t = new Thread(worker);
    t.setUncaughtExceptionHandler((thread, throwable) -> {
        if (throwable instanceof IndexOutOfBoundsException)
            System.err.println("Thread " + thread.getId());
    });
}
```

⁷Pun intended.

```

        + "was created with wrong indices.");
    });
    t.start();
}

```

4.1.3 Locks

Several lock implementations can be found in the Java standard library. They are all blocking data structures. The Basis for these implementations is the interface *Lock*. The declared methods are depicted in Listing 4.9. Implementations of *Lock* provide a more versatile way of synchronization than *synchronized* does. Locks can be chained in arbitrary order which allows fine-grained control over shared resources. With *tryLock* and its overload, non-blocking operations are provided. Using *lockInterruptibly* will release the lock when the thread holding it is interrupted. This makes the implementation of cancellable tasks easier.

Listing 4.9: The Lock interface

```

void lock();
void lockInterruptibly();
boolean tryLock();
boolean tryLock(long timeout, TimeUnit timeUnit);
void unlock();

```

The class *ReentrantLock* is perhaps the most commonly used implementation. As the name implies, the lock allows the thread currently holding it to reacquire it.

Read-write lock

A *Readers-Writer* scenario involves shared state which is read by arbitrarily many readers, but only mutated by a single writer. For instance this approach is beneficial when a global configuration data structure is read regularly by several threads, but only occasionally written to.

Read-write locks are optimized accordingly. Multiple readers can acquire the lock at the same time, but only if no writer is present. The writer on the other can acquire the lock when not a single reader holds it.

StampedLock provides a sophisticated implementation. A writer acquires the lock via the method *writeLock*. Analogously, readers acquire it through *readLock*. Optimistically locking for a read is possible with *tryOptimisticRead*.

Each of the acquire operations returns a number of type *long*, called *stamp*, which represents the state of the acquisition. To verify the validity on an optimistic read, the method *validate* is called on the stamp. The lock is released by supplying the stamp to the method *unlock*.

Listing 4.10: Optimistic read

```

var stampedLock = new StampedLock();
...
var stamp = stampedLock.tryOptimisticRead();
var running = configuration.applicationRunning();
if (!stampedLock.validate(stamp)) {
    stamp = stampedLock.readLock();
    try {
        running = configuration.applicationRunning();
        ...
    } finally {

```



```

        stampedLock.unlock(stamp);
    }
}
...

```

In a program scheduling many tasks on one and the same thread, one might prefer a non-blocking lock. This way, only a single task is blocked instead of the thread which increases throughput of the system as a whole. Furthermore, providing a data structure that encapsulates access to the shared resource in a readers-writer manner can rapidly simplify client code.

Listing 4.11: NonblockingReadWriteLock

```

public class NonblockingReadWriteLock<TValue>{
    static final int WRITE_STATE = -1;
    static final int EMPTY_STATE = 0;
    private final AtomicInteger lockState;
    private final Supplier<TValue> readOperation;
    private final Consumer<TValue> writeOperation;

    public NonblockingReadWriteLock(
        Supplier<TValue> readOperation,
        Consumer<TValue> writeOperation
    ) {
        this.lockState = new AtomicInteger();
        this.readOperation = readOperation;
        this.writeOperation = writeOperation;
    }

    public TValue read() {
        try {
            var done = false;
            while (!done) {
                var previousValue = this.lockState.getAcquire();
                if (previousValue == WRITE_STATE)
                    continue;
                done = this.lockState.getAndUpdate((currentValue) -> {
                    if (currentValue == WRITE_STATE) return WRITE_STATE;
                    return currentValue + 1;
                }) != WRITE_STATE;
            }
            var value = this.readOperation.get();
            return value;
        } finally {
            this.lockState.decrementAndGet();
        }
    }

    public void write(TValue value) {
        try {
            var done = false;
            while (!done) {
                if (this.lockState.getAcquire() != EMPTY_STATE)
                    continue;
                done = this.lockState.compareAndSet(EMPTY_STATE, WRITE_STATE);
            }
            this.writeOperation.accept(value);
        } finally {
            this.lockState.set(EMPTY_STATE);
        }
    }
}

```

```

    }
}

```

Listing 4.11 gives an implementation. *NonblockingReadWriteLock* is based on a single *AtomicInteger*, the *lockState*. The semantic of the state is straightforward. As long as *lockState* assumes 0, the lock is not held. -1 indicates exclusive access to a writer. Any positive value stands for the number of readers currently holding the lock.

A spin lock based on *compare-and-set* and *read-modify-write*, here *getAndUpdate*, is used to grant mutual exclusion. To reduce contention on *lockState*, the current value is polled, and is checked upon whether the lock even can be acquired. Note that the method *getAcquire* of *AtomicInteger* is used. This ensures that subsequent calls are not reordered prior to this statement. As mentioned in Section 2.1, the order of statements in code is subject to change due to compiler optimizations.

NonblockingReadWriteLock encapsulates read and write operations. Reading a value is defined by the instance of *Supplier* given. *Supplier* declares a single method, *T get()*; which returns a value of the respective type. The interface *Consumer* declares a single method, *void accept(T t)*. The given instance represents write operations on the underlying data structure. Confined in such a way, resources can be shared easily and safely across the application.

Try-with-resources on locks

Like the implementation in Listing 4.7 already illustrated, locks like *ReentrantLock* have a downside. A thread holding the lock is also obliged to release it.

Similarly, file handles and database connections have to be closed. With the *try-with-resources* statement, Java simplifies this open-close lifecycle. The *try* operator expects in instance of *AutoCloseable*. This interface declares a single method, *void close()*. When the *try* block returns, the *close* is automatically invoked.

While searching for whether it is possible to handle instances of *Lock* the same way, I stumbled across a surprisingly simple solution: A custom implementation extends *ReentrantLock* and implements *AutoCloseable*. Thus, it can be subject to *try-with-resources*.

The implementation⁸ is given in Listing 4.12. By calling the method *open*, the lock is acquired. The automatically invoked method *close* releases the lock.

Listing 4.12: CloseableReentrantLock

```

public class CloseableReentrantLock
    extends ReentrantLock implements AutoCloseable {
    public CloseableReentrantLock open() {
        this.lock();
        return this;
    }
    @Override
    public void close() {
        this.unlock();
    }
}
...
var c = new CloseableReentrantLock();
try (var closeableLock = c.open()) {
    ...
}

```

⁸Found on *stackoverflow*, <https://stackoverflow.com/a/11000458>, accessed 30. May 2022.

4.1.4 Creating and managing tasks

Java provides several facilities for the creation and management of asynchronous tasks. This section gives a compact introduction. Concrete use cases are shown in the subsequent section.

Future

The foundation for asynchronous computations in Java is the interface *Future*, depicted in Listing 4.13. Through the methods *cancel* and *isCancelled*, instances fulfilling *Future* represent cancellable tasks. How cancellation is handled differs between implementations.

The method *isDone* provides a soft mechanism to check whether the computation has already finished. Using *get* to retrieve the result, on the other hand, lets the current thread wait until the computation completes. Additionally, the retrieval can be given a timeout. Should the result not be available once the specified time span has elapsed, a *TimeoutException* is thrown.

Listing 4.13: The Future interface

```
interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    V get();
    V get(long timeout, TimeUnit unit);
    boolean isCancelled();
    boolean isDone();
}
```

ExecutorService

Hand in hand with *Future* goes *ExecutorService*. This interface pairs creation and execution of asynchronous tasks with termination management. In order to run a task, the method *submit* is provided which takes an instance of *Callable* and returns a *Future*. An overload of *submit* expects an instance of *Runnable* instead. While instances of *Runnable* do not provide a value upon completion, the associated *Future* is still useful for completion checks and cancellation.

An executor service is of particular interest when many tasks are to be executed. The method *invokeAll* takes a collection of *Callable* instances and runs them all. With *isTerminated* one can check whether all tasks have completed. To wait for the completion of tasks, *awaitTermination* is used. This method takes a timeout as parameter and will return *false* when not all tasks have completed before the specified time span has elapsed.

Executor services implementing *ScheduledExecutorService* make it possible to delay task execution or to schedule tasks periodically.

In general, executor services can be divided into three different groups: (1) single-threaded executors, (2) multithreaded executors with a fixed number of threads, (3) multithreaded executors with an unbounded number of threads.

To create instances of *ExecutorService*, one should refer to the class *Executors*. It provides static convenience functions to instantiate different kinds of executor services. An executor service using a fixed number of threads can be created via *newFixedThreadPool* or *newScheduledExecutor*, respectively. Using *newSingleThreadExecutor* creates an executor service operating on a single thread (*newSingleThreadScheduledExecutor* instantiates the scheduling pendant).

Calling *newCachedThreadPool* creates an executor service using arbitrarily many threads. Each task is scheduled to a separate thread, run immediately. A cache of already created threads enables thread reuse. Whenever a cached thread is idle for 60 seconds, it is

terminated and removed from the cache. This scheduling approach is viable for scenarios where many short-running tasks have to be run as soon as possible.

CompletableFuture

The class *CompletableFuture* provides a versatile implementation for *Future*. Its capabilities lie far beyond those of other implementations. *CompletableFuture*, as the name implies, can be completed. To do so, the method *complete* is called. It takes a concrete value and completes the task.

Instances can be chained, combined and composed in various ways. Furthermore, *CompletableFuture* provides several static functions to conveniently create instances. The usage is depicted in Listing 4.14. A predefined number of tasks is created using the static function *supplyAsync*. It takes an instance fulfilling *Supplier*. The helper, *aggregate*, is already completed, its value is available instantaneously. Through *thenCombine*, the computation results of each worker are collected and aggregated one by one. The method expects another *CompletableFuture* and a function to process both results. The overall result is retrieved through *join*.

Listing 4.14: CompletableFuture

```
List<CompletableFuture<Integer>> workers =
    IntStream.range(0, numTasks)
        .mapToObj(i -> CompletableFuture.supplyAsync(() -> i))
        .collect(Collectors.toList());

var aggregate = CompletableFuture.completedFuture(0);
for (var worker : workers) {
    aggregate = aggregate
        .thenCombine(worker, (left, right) -> left + right);
}
var result = aggregate.join();
```

Note that the usage of *thenCombine* resembles the *reduce* operation in a typical *Collection Pipeline*⁹ of the form *filter-map-reduce*. And in fact, *CompletableFuture* supports arbitrary pipelines which make asynchronous programming a breeze.

4.1.5 Concurrent collections by example

The Java standard library provides numerous concurrent collections, blocking and non-blocking alike. They all have their application. Yet, it is out of the scope of this work to discuss each of them.

The following examples showcase two particularly interesting data structures, *ConcurrentHashMap* and *ConcurrentLinkedQueue*.

Counting occurrences

Recall the algorithm, *searchOccurrences*, from Listing 4.6. The number of occurrences for a given element was requested. The requirements have changed. Now the number of occurrences for each unique element have to be retrieved. Thus, a mapping from elements to their count is needed. In a concurrent environment, *ConcurrentHashMap* is a great choice. Due to fine-grained locking it provides highly concurrent read and write access.

Changes to the existing algorithm are quite simple: instead of atomically incrementing a shared number, the count of each element is collected and updated in a map. The

⁹See <https://martinfowler.com/articles/collection-pipeline/>, accessed 24. May 2022.

functionality of a worker is given in Listing 4.15. Simply checking whether an element is already present in the map and acting accordingly is not possible in this scenario as it might result in race conditions. To illustrate this, consider two tasks, A and B, both finding out simultaneously that the current item is not yet present in the map. Now both would put the element as key and 1 as value into the map which results in a lost update.

Listing 4.15: The method `createSearchWorker`

```
private static <T> Runnable createSearchWorker(T[] items,
    int start, int end, Map<T, Integer> elementsToOccurrences) {
    return () -> {
        IntStream.range(start, end)
            .forEach(index -> {
                var item = items[index];
                var previous = elementsToOccurrences.putIfAbsent(item, 1);
                if (previous != null)
                    elementsToOccurrences
                        .computeIfPresent(item, (key, value) -> value + 1);
            });
    };
}
```

To avoid race conditions, the task first attempts to put a completely new key-value pair. The method `putIfAbsent` returns `null` if the specified key was not already associated with a value. Otherwise, the current value is returned. When the put attempt fails, the key is already present in the map and the current value associated with it must be incremented. Again, retrieving the current value, calculating the new value and putting it into the map is not applicable due to race conditions. In this situation, the method `computeIfPresent` comes to the rescue. It allows to aggregate the current value to create a new mapping.

The updated implementation of `searchOccurrences` is given in Listing 4.16. The method takes an additional parameter, `awaitTime`, denoting the time to wait for termination. Note that `Semaphore` is not used anymore. Instead, the facilities of executor services are utilized. `AtomicInteger` is replaced by `ConcurrentHashMap`.

The workers are created in the same fashion as before. Only now, an executor service of a fixed size is used to run the workers and await their termination.

Listing 4.16: The method `searchOccurrences`

```
public static <T> Map<T, Integer> searchOccurrences(
    T[] items, int numTasks, long awaitTime
) throws InterruptedException {
    var elementsToOccurrences =
        new ConcurrentHashMap<T, Integer>();

    var workerPool = new ArrayList<Runnable>();
    final int numItems = items.length / numTasks;
    var lastExclusiveEnd = 0;
    for (int i = 1; i < numTasks; i++) {
        final int start = lastExclusiveEnd;
        lastExclusiveEnd += numItems;
        workerPool.add(createSearchWorker(
            items,
            start, lastExclusiveEnd,
            elementsToOccurrences));
    }

    workerPool.add(createSearchWorker(
```

```

        items ,
        lastExclusiveEnd , items.length ,
        elementsToOccurrences));

    var executorService = Executors.newFixedThreadPool(numTasks);
    for (var worker : workerPool)
        executorService.submit(worker);
    executorService.awaitTermination(awaitTime, TimeUnit.SECONDS);
    executorService.shutdownNow();

    return elementsToOccurrences;
}

```

Message passing

While all sorts of collections have their application, queues play an integral role in concurrent programming. The different implementations of *ExecutorService* all use a queue to manage threads internally.

Likewise, they are incorporated as intermediary data structure for *Producer-Consumer* scenarios, as a bag containing tasks in a *Supervisor-Worker* implementation, or for channels.

While some other languages support message passing via channels, Java uses them solely within the package *java.nio* (new I/O). Implementations are package-private and thus cannot be applied to client code.

However, message passing is a versatile approach to concurrent programming. Thus, a data structure providing asynchronous message passing facilities is highly appreciable.

Listing 4.17: TaskCommunicationChannel

```

public class TaskCommunicationChannel<T> {
    private final Queue<T> messageQueue;
    private final Queue<CompletableFuture<T>> registeredReceives;

    public TaskCommunicationChannel() {
        this.messageQueue = new ConcurrentLinkedQueue<>();
        this.registeredReceives = new ConcurrentLinkedQueue<>();
    }

    public CompletableFuture<T> receive() {
        var nextMessage = this.messageQueue.poll();
        if (nextMessage == null) {
            var newRegisteredReceive = new CompletableFuture<T>();
            this.registeredReceives.add(newRegisteredReceive);
            return newRegisteredReceive;
        }
        return CompletableFuture.completedFuture(nextMessage);
    }

    public void send(T message) {
        var registeredReceive = this.registeredReceives.poll();
        // A prior receive could not be served.
        if (registeredReceive != null) {
            registeredReceive.complete(message);
            return;
        }
        this.messageQueue.add(message);
    }
}

```

An implementation for an asynchronous channel is given in Listing 4.17. The idea is straightforward: An asynchronous data structure is non-blocking per definition. Thus, waiting operations are prohibited. This gives rise to the notion of a *pending receive*. A receive operation on the channel is pending as long as no message was sent to the channel to reply to it.

The implementation is based upon two queues, one storing incoming messages, the other one pending receives. Both are instances of *ConcurrentLinkedQueue*. This is a fantastic implementation of a non-blocking queue algorithm described by Scott and Michael [MS96].

Instead of a concrete value, the method *receive* returns a *CompletableFuture*. As a wrapper for asynchronous computations, it represents the message and the pending receive at the same time. If polling the next message from the message queue succeeds, a completed instance containing the polled message is returned. Otherwise, an empty *CompletableFuture* is first registered, then returned.

The method *send* on the other hand, first checks whether a prior receive is still pending. If so, it is served. Otherwise, the message queue is appended. Any thread can help out other threads in completing pending receives — a common feat of non-blocking data structures.

4.1.6 Scala

Scala is a programming language combining object-oriented and functional programming paradigms. Scala is designed to seamlessly integrate with Java functionality. Java libraries can be called from Scala code. Furthermore, Scala uses the same class path and runs on the JVM.

Values in Scala are immutable by default. As opposed to purely functional programming languages, it is possible to explicitly declare variables and object properties as mutable. Nonetheless, Scala encourages developers to rely on immutable objects primarily. For instance, immutable collections are simply available just like everything contained in the Java package *java.lang*. Mutable collections in the other hand have to explicitly be imported from the *scala.collection.mutable* package.

Immutable state is thread-safe by design which is why Scala is useful for concurrent programming.

Future in Scala

Similar to Java, Scala provides *Future* for light weighted concurrent tasks. Despite the name, due to the fact that futures in Scala are composable in various ways, they do resemble Java's *CompletableFuture*. Listing 4.18 illustrates this. Two futures of type integer, *task1* and *task2* are created. Via the *for ... yield* statement, they are combined into a single instance. Finally, the resulting value is used in the callback supplied to *onComplete*. Note that errors are also handled this way.

Listing 4.18: Futures in Scala

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Failure, Success}

val task1: Future[Int] = Future {
  // Execute computation
}
val task2: Future[Int] = Future { ... }
val result =
  for
```

```

    res1 <- task1
    res2 <- task2
  yield
    (res1 + res2)

result.onComplete {
  case Success(value) => handleResult(value)
  case Failure(err) => handleError(err)
}

```

Futures are executed in an *ExecutionContext*. With *ExecutionContext.global* Scala provides a default implementation sufficient for most use cases. In special scenarios, a Java *ExecutorService* can be transformed into an *ExecutionContext* using the function *fromExecutor*. Listing 4.19 illustrates how to run a future in a *FixedThreadPool*.

Listing 4.19: Transforming an *ExecutorService*

```

import java.util.concurrent.Executors;

val executionContext = ExecutionContext.fromExecutor(
  Executors.newFixedThreadPool(limit: Int))
val task: Future[Int] = Future { ... }(executionContext)

```

4.2 Rust

Rust is a systems programming language. It combines low-level facilities with high-level *zero cost* abstractions. Zero cost means they add no runtime overhead. For several years in a row, it has been the *most loved* programming language among developers¹⁰. And for good reason: The rigid memory safety properties it grants render most memory related issues, such as *use-after-free*, void. Many data types in the standard library enable pipeline actions which makes composition and error handling a breeze¹¹.

Unlike Java, the Rust standard library is rather sparse regarding concurrent data structures. There simply are no concurrent collections, executor services or light-weight task implementation like *CompletableFuture*. The upcoming sections describe why concurrent programming in Rust is a charm, nonetheless.

A word on asynchronous programming Rust does incorporate *async/await* to extend possibilities for asynchronous programming. However, it can be difficult to bring synchronous and asynchronous code together [Ros17]. A synchronous function, for instance, cannot directly call an asynchronous one. To work reasonable with the language features, external libraries have to be included¹².

Support for asynchronous programming in Rust is rapidly evolving. Thus, the current state of development cannot be perceived as stable and is therefore not included in this thesis.

Yet, asynchronous programming as an element of software design does not depend on specific language or runtime features. The fundamental idea is always the same: The value of an asynchronous computation will eventually be available at some point in the future. This is why many languages, including Java and Rust, call the handle for that value *Future*. The closing example in Section 4.2.8 showcases how asynchrony can be achieved through synchronous mechanisms.

¹⁰<https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted-language-love-dread>

¹¹For a thorough introduction to Rust, please refer to <https://doc.rust-lang.org/book/>

¹²<https://rust-lang.github.io/async-book/01.getting-started/03.state-of.async-rust.html>

4.2.1 Fearless concurrency

From the very beginning, one of Rust's major goals was to prevent concurrency problems. While seemingly unrelated, memory safety guarantees combined with a highly expressive type system inhibit concurrency related issues. Sharing data between threads is constrained. Thus, potentially erroneous operations on shared state cannot be performed at all.

Rust's compiler will report violations of thread-safety. This is what the Rust community calls *fearless concurrency*.

Ownership and borrowing

A data item in Rust has a single owner. This might be the instance of a *struct*, a function, or simply a code block. The owner decides how to use the data. Usage, however, is constrained: At a point, there can be arbitrarily many immutable references, but exclusively only one mutable reference. Passing references to other functions or data structures is called *borrowing*. Ownership can also be transferred. Data is then said to be consumed. Once an owner goes out of scope, the contained data can no longer be used and the associated memory is freed.

The notion of shared state is directly incorporated into the language. The fact that only a single piece of code can mutate data, prevents data races altogether.

4.2.2 Running code in threads

A new thread is created by the function *spawn*. It takes a closure as argument which is run in the thread. It returns a *JoinHandle*. As the name implies, it can be used to join the thread, that is, to wait on its execution to finish.

Threads in Rust can return a value. This is depicted in Listing 4.20. The specified closure calculates and returns the sum from 0 to 10 (exclusively). The resulting value can be retrieved by calling *join* on the join handle. An instance of type *Result* is returned, which either contains the computed value or an error. The latter is only the case when the execution of the respective thread led to an unrecoverable error. In Rust terms: it *panicked*. In this simple scenario, the thread will always return normally. Thus, it is safe to retrieve the value via *unwrap*.

Listing 4.20: Spawning a thread

```
use std::thread::spawn;
...
let n = 10;
let handle: JoinHandle<i32> = spawn(move || {
    return (0..n).sum();
});
let result = handle.join().unwrap();
...
```

Due to its simplicity, it is tempting to *spawn* threads all over the place. In case of many short-lived operations that are expected to return early, it might be fine to do so. Nonetheless, each thread created is associated exclusively to an operating system thread.

4.2.3 Reference counting

Smart pointers in Rust simplify the usage of references and data. The smart pointer *Rc* — reference counted — employs reference counting to allow for multiple quasi-owners. To share the data it points to, a clone is created. *Rc* keeps track of all of its clones via an

internal counter. Memory is only freed when the last clone goes out of scope. In order to handle this, Rust provides the *Drop* trait. Implementing data structures alter the way in which their data is *dropped*.

The usage of *Rc* is depicted in Listing 4.21. By calling the associated function *clone*, a new clone is created which is then consumed by the construction of an instance of a custom data type. Note that *Rc* allows only immutable access to its contained data.

Listing 4.21: The smart pointer *Rc*

```
use std::rc::Rc;
...
let rc = Rc::new(1);
let rc_clone = Rc::clone(&rc);
let custom_struct = CustomStruct::new(rc_clone);
...
```

Instances *Rc* cannot be shared across different threads¹³. To do so, *Arc* is needed. *Atomically reference counted*, that is. Atomicity of reference counting makes it possible to safely share *Arc* among threads.

Listing 4.22 illustrates a typical scenario. A clone of an *Arc* instance, *a_c*, is handed over to another thread. Notice the *move* operator preceding the closure. Its effects are as follows: Ownership of the clone is transferred to the code running within the created thread, it is *moved*. Thus, the reference *a_c* is invalidated for its originating context and can no longer be used.

Listing 4.22: Move *Arc* to new thread

```
use std::sync::Arc;
...
let a = Arc::new(1);
let a_c = Arc::clone(&a);
let t = spawn(move || {
    let val = a_c.as_ref();
    ...
});
...
```

4.2.4 Locks

While *Arc* enables data sharing among threads, it does not provide mutable access to the contained data. Another layer has to aid in an additional control structure providing mutual exclusion of involved threads. The Rust standard library provides two implementations of locks: *Mutex*, which is shorthand for mutual exclusion, and *RwLock*, an implementation of *read-write lock*. Both are blocking data structures.

At first, this might seem rather meager — especially when coming from a language like Java which provides a vast amount of varying implementations. But, indeed, the combination of *Arc* with either of them suffices to fulfill many synchronization scenarios.

The beauty of locks in Rust is that they contain data to which mutually exclusive access shall be granted. Thus, upon lock acquisition, a handle to the internal data is returned. This handle lets the owner access the data according to the requested access rights.

Common usage of *Mutex* is illustrated in Listing 4.23. A new instance, guarding a single integer, is put inside an instance of *Arc*. The clone is moved to the other thread where the lock can be acquired. Calling *lock* blocks the current thread until the mutex is acquired and then returns a *MutexGuard*, the handle. Once this handle is dropped, the lock is released.

¹³See Section 4.2.6

Thus, no additional unlocking is needed. Notice the similarity between this and Java's *try-with-resources* shown in Listing 4.12.

Listing 4.23: Combination of Arc and Mutex

```
use std::sync::{Arc, Mutex}
...
let mutex = Arc::new(Mutex::new(1));
let mutex_c = Arc::clone(&mutex);
let handle = spawn(move || {
    *mutex_c.lock().unwrap() += 1;
});
...
```

RwLock works quite similar. Instead of *lock*, it provides the methods *read* and *write* for reading and writing operations. *RwLock* follows standard semantics of read-write locks. The handle returned by *read* only allows immutable access.

Poisoned locks The methods described above do actually return an instance of *Result* which contains, upon success, the concrete handle. An error is only given when a thread panicked amidst holding a lock. The lock is then said to be *poisoned*. The resulting error contains a handle to the inner data of the lock which raises the opportunity to potentially restoring a correct state.

In addition to the blocking versions of lock acquisition, non-blocking operations are also provided. *Mutex* has the function *tryLock* which only succeeds when the lock is free. Analogously, *RwLock* provides *try_read* and *try_write*. A simple use case is depicted in Listing 4.24.

Listing 4.24: *RwLock::tryRead*

```
use std::sync::RwLock;
...
let rw_lock = RwLock::new(1);
...
if let Ok(val) = rw_lock.try_read() {
    handle_state(*val)
}
...
```

4.2.5 Ordering of atomic operations

In the module *std::sync::atomic* Rust provides many data structures supporting atomic operations, such as *AtomicBool* and *AtomicUsize*. Unlike their counterparts in Java, each atomic operation has to be given one or more orderings. An ordering, defined by the enum *Ordering*, specifies the relationship between separate calls. The five members are as follows: (1) *Relaxed* poses no ordering constraints, (2) *Release* has the effect that every prior call will be executed before this call, (3) *Acquire* ensures that every subsequent call is executed after this call, (4) *AcqRel* acts for loading operations like *Acquire* and for storing one's like *Release*, (5) *SeqCst*¹⁴ is a combination of all other orderings.

Just like in *ReadWriteLock* shown in Listing 4.11, *Acquire* should be used to load data at the beginning of a critical section. Analogously, *Release* goes hand in hand with (re)storing at the end. *SeqCst* poses the strongest constraints on reordering. If every atomic operation were to use it, there would be a single history of calls that every thread would agree on.

¹⁴Sequentially Consistent

4.2.6 Send and Sync

Marker traits form a special class of traits in Rust. They do not declare any methods. Types implementing marker traits ensure the compiler to encompass specific properties.

The marker traits *Send* and *Sync* enforce implementing types to adhere to concurrency properties in two ways: *Send* requires the implementing type to be safely transferrable between threads. The vast majority of Rust's types, including many members of the standard library, are *Send*. Yet, some types cannot allow to be sent to another thread. For instance, if a clone of a *Rc* was transferred from thread A to thread B, both A and B could concurrently alter the reference counter. This is solved by *Arc* due to the atomicity of its internal reference counter.

Sync, on the other hand, requires that the implementing type can safely be referenced by several threads. Again, *Rc* is not *Sync*. An immutable reference suffices to clone a *Rc*. Thus, the reference counter can likewise be mutated concurrently.

Types that consist solely of other types that are *Send* or *Sync*, are also considered to be *Send* or *Sync*, respectively. Custom types are most seldom required to implement them on their own. However, constraining generic types with either marker trait is almost certainly required when approaching concurrent programming in Rust on a broader scale.

4.2.7 Message passing

Rust encourages developers to employ message passing for concurrent programming. Via the function `std::sync::mpsc::channel`¹⁵, a channel can be created. The channel is divided into a *Sender* and a *Receiver*. Multiple senders can exist, but only a single receiver is allowed.

Listing 4.25 illustrates common usage of the channel. The sending end is moved into another thread, wherein it sends a message by calling *send*. The originating thread calls *recv* on the receiving end. The thread will block until a message is available. The method *recv* returns a *Result* which will only contain an error when all corresponding senders have hung up, that is, when they were dropped.

Listing 4.25: Channel

```
use std::sync::mpsc::channel;
use std::thread;
...
let (sender, receiver) = channel();
let handle = thread::spawn(move || {
    sender.send(1).expect("Broken channel.");
});
if let Ok(message) = receiver.recv() {
    ...
}
```

Receiver provides several methods for receiving messages. With *try_recv*, an optimistic non-blocking approach can be employed. Additionally, to the reason *recv* might return an error, *try_recv* will also fail when no message is available.

The function *sync_sender* creates a *SyncSender* which blocks when the channel's internal buffer is full. In addition to the *send* method, it also provides a non-blocking alternative, *try_send*.

Regardless of whether the channel is synchronous or not, the respective *send* methods fail when there is no receiver to send messages to. Furthermore, *try_send* will also return an error where *send* would block.

¹⁵*mpsc* stands for multi-producer, single-consumer.

4.2.8 A task scheduler

Threads in Rust can be used as tasks possibly returning a value. However, spawning many threads executing computationally intensive work will slow down the application due to overuse of the systems resources and is therefore highly discouraged. Furthermore, managing a thread's lifecycle, such as joining it, can be quite tedious. Thus, a data structure similar to Java's *ExecutorService* is appreciable. By combining the techniques discussed in the previous sections, it becomes fairly easy to implement custom concurrent data structures. Throughout this section, the implementation of a simple task scheduler is laid out.

The specification of abilities and properties of the scheduler are as follows: (1) the scheduler is easy to use, including the creation of tasks, (2) tasks may return a value, (3) the current state of a task can be polled, (4) arbitrary many tasks may be scheduled, (5) the scheduler should internally manage a configurable number of threads, called worker threads, (6) the scheduler enables proper shutdown.

Listing 4.26: Job

```
pub type Job<T> = Box<dyn Send + FnMut() -> T>;
```

Rust has great support for higher-order functions. This gives rise to a simple mechanism for specifying the work a task has to execute. In Listing 4.26, the generic type alias *Job* is defined. The type has to be a function that supports mutable operations and returns a value of the given generic type. But it must also be *Send*, as otherwise it cannot be submitted to a worker thread. The smart pointer *Box* ensures that the function resides on the heap. Therefore, its size is known at compile time which simplifies matters.

The result of a job will be available at some point in the future. To retrieve the result and to check a task's current state, a handle on the task is handed over to client code. Listing 4.27 shows the generic structure *Task* that contains both state and result. For obvious reasons, an instance of *Task* is shared by multiple threads. Thus, both fields are wrapped in an *Arc*. The result is represented by an optional value, as it is only available upon completion. While *result* is guarded by a *Mutex*, *schedule_state* uses *RwLock* because the state is expected to be read often.

The optional result value alone does not suffice to fully represent the state of execution. Therefore, *ScheduleState* defines three states a task can assume, *Pending*, which is the default value, *Running*, and *Finished*.

Listing 4.27: Task

```
#[derive(Clone, Copy, PartialEq, Debug)]
pub enum ScheduleState {
    Pending,
    Running,
    Finished
}

impl Default for ScheduleState {
    fn default() -> Self { ScheduleState::Pending }
}

pub struct Task<T> {
    schedule_state: Arc<RwLock<ScheduleState>>,
    result: Arc<Mutex<Option<T>>>
}
```

Admittedly, retrieving the result inside an *Arc* and guarded by a *Mutex* is tricky. The

current thread must be the exclusive owner of a given *Task* instance. Otherwise, unwrapping *Arc* will fail. The method *try_get*, shown in Listing 4.28, consumes the instance, returns either the result or itself as error type.

This is a common pattern in Rust. The associated function *Arc::try_unwrap* does exactly the same. It consumes the *Arc*, returning the respective instance on failure, that is, when the reference counter is greater than 1.

Calling *Mutex::into_inner* or the *read* operation on *RwLock* will only ever result in an error when a thread holding the respective lock panics. However, both locks are encapsulated and will solely be acquired through the methods of *Task*. Thus, they will never be poisoned. Once a task assumes the state *Finished*, the result is guaranteed to be available.

Retrieving the current state of a task is done via the method *state*.

Listing 4.28: Retrieving result and state of a task

```
pub fn try_get(self) -> Result<T, Self> {
    if ScheduleState::Finished != *self.schedule_state.read().unwrap() {
        return Err(self);
    }
    match Arc::try_unwrap(self.result) {
        Ok(mutex_val) => {
            Ok(Mutex::into_inner(mutex_val)
                // The lock is never poisoned
                .unwrap()
                // Since the state is FINISHED, a value is present
                .unwrap())
        },
        Err(arc) => Err(Task { result: arc, ..self })
    }
}

pub fn state(&self) -> ScheduleState {
    *self.schedule_state.read().unwrap()
}
```

Note the resemblance to Java's *CompletableFuture*. Once a job has finished, the result will be set. Hence, the task is completed. The code for setting state and result is given in Listing 4.29. While the methods *set_state* and *set_result* both use an *if let* clause matching the positive case, since the lock is never poisoned, acquiring it will always succeed (the current thread might, however, block until the lock is released by another thread).

Calling the associated function *Task::new* creates an instance with default values. *ScheduleState* implements the trait *Default* to support this operation. The method *clone* utilizes the reference counting of *Arc* to construct a new instance with the exact same content. Note that none of these functions is part of the public interface.

Listing 4.29: Lifecycle methods of Task

```
fn new() -> Self {
    Task {
        schedule_state: Default::default(),
        result: Default::default()
    }
}

fn set_state(&self, state: ScheduleState) {
    if let Ok(mut current_state) = self.schedule_state.write() {
        *current_state = state
    }
}
```

```

    }
    fn set_result(&self, res: T) {
        if let Ok(mut current_result) = self.result.lock() {
            *current_result = Some(res);
        }
    }
}

fn clone(&self) -> Self {
    Task {
        schedule_state: Arc::clone(&self.schedule_state),
        result: Arc::clone(&self.result)
    }
}

```

Job and *Task* build the foundation of interaction between the scheduler and client code. Jobs and tasks have a one-to-one relationship. The structure *TaskHandle* shown in Listing 4.30 represents this characteristic. It owns exactly one task and one job. This intermediary data structure is necessary, because *FnMut* cannot be cloned. Thus, *TaskHandle* is the exclusive owner of the job.

Task, on the other hand, has to be available on separate threads. Composing the data structures this way is therefore inevitable.

Listing 4.30: TaskHandle

```

struct TaskHandle<T> {
    task: Task<T>,
    job: Job<T>
}

```

Functions provided by *TaskHandle* manage the lifecycle of a *Task*. Usage is straightforward: The associated function *new* expects a *Job* and creates a new instance of *Task*. Listing 4.31 depicts the implementation.

As the name implies, the method *execute* runs the actual work. Before and afterwards, the task's state is set respectively. Once the job is finished, the returned value is set as result of the task.

A separate handle to the internal *Task* instance can be retrieved through the method *task*. It is later on used to conveniently create the instance handed over to client code.

Listing 4.31: Methods of TaskHandle

```

fn new(job: Job<T>) -> Self {
    TaskHandle {
        task: Task::new(),
        job
    }
}

fn execute(&mut self) {
    self.task.set_state(ScheduleState::Running);
    let result = (self.job)();
    self.task.set_result(result);
    self.task.set_state(ScheduleState::Finished)
}

fn task(&self) -> Task<T> {
    self.task.clone()
}

```

The implemented data structures so far pave the way for simple task handling. What is now left is the actual scheduler. Scheduling a new task is to submit it to a worker thread.

Thus, the scheduler needs a mechanism to communicate with its worker threads. Message passing via Rust's channels comes in handy.

Listing 4.32 shows the structure *Scheduler*. For each thread, it keeps a pair consisting of its join handle and the sending part of a channel. The latter transfers instances of *TaskHandle* to the respective thread. Via the boolean flag *running*, the scheduler indicates its threads to terminate. Thread-safety is granted due to atomicity. The number *current_idx* is later on used to determine the thread to submit the next task to.

Listing 4.32: Scheduler

```
pub struct Scheduler<T> {
    handles_senders: Vec<(JoinHandle<()>, Sender<TaskHandle<T>>>>,
    running: Arc<AtomicBool>,
    current_idx: usize
}
```

The amount of threads to use internally should be configurable. Therefore, the instantiation of a *Scheduler* expects a number. The code is shown in Listing 4.33. A list of *JoinHandle Sender*-pairs is built. Only a single instance of *AtomicBool* is used throughout the different threads. These are created by calling the associated function *create_threads*. The first argument is the receiving end of the respective channel. The second argument is the *running* flag.

Due to the fact that instances of *TaskHandle* are sent to separate threads, the generic type is constrained to be *Send* and *'static*. The latter indicates that the type itself stays valid for the application's lifetime. This is necessary, because a thread might outlive its originating context.

Upon creation, each thread first blocks which is done via *thread::park*. It is said to be parked. This saves CPU system resources. Once *unparked*, a thread continuously checks whether to keep running. The non-blocking operation *try_recv* on the channel is preferable in this scenario, as calling *recv* would block the thread until another message is available. Every message sent, indeed, simply represents the next task to execute. The encapsulation within *TaskHandle* makes this a breeze.

Listing 4.33: Creating a new Scheduler

```
pub fn new(num_threads: usize) -> Self {
    let mut handles_senders = Vec::with_capacity(num_threads);
    let running = Arc::new(AtomicBool::new(true));
    for _ in 0..num_threads {
        let (sender, receiver) = channel::new();
        let handle = Self::create_thread(receiver, Arc::clone(&running));
        handles_senders.push((handle, sender));
    }
    Scheduler {
        handles_senders,
        running,
        current_idx: 0
    }
}

fn create_thread(receiver: Receiver<TaskHandle<T>>,
    running: Arc<AtomicBool>)
-> JoinHandle<()> {
    spawn(move || {
        thread::park();
        while running.load(Ordering::Acquire) {
            if let Ok(mut next_task) = receiver.try_recv() {
```



```

        next_task.execute()
    } else { thread::park() }
}
})
}

```

The worker threads are set up and ready. In order to schedule a task, one simply calls the method *schedule* on the scheduler. The implementation is given in Listing 4.34. The scheduler does not employ a specialized algorithm for scheduling. Instead, *current_idx* is used as ring counter to sequentially retrieve the next thread to submit the task to.

A new *TaskHandle* is created supplying the given job. After creating the *Task* clone handed to client code, the task handle is sent to the respective thread.

Listing 4.34: Scheduling a task

```

pub fn schedule(&mut self, job: Job<T>) -> Task<T> {
    let (handle, sender) = self.handles_senders
        .get(self.current_idx).unwrap();
    handle.thread().unpark();
    self.current_idx += 1;
    self.current_idx %= self.handles_senders.len();
    let task_handle = TaskHandle::new(job);
    let task = task_handle.task();
    sender.send(task_handle).expect("Broken channel");
    task
}

```

The last requirement still outstanding is proper shutdown. Due to the fact that each thread cycles around *running*, setting it to false suffices to halt their execution. The method *invoke_termination* shown in Listing 4.35 does exactly that. Afterwards, it *unparks* each thread to guarantee their termination. A currently running task will be executed nonetheless.

Shutting down the scheduler is finally done by calling *join* on it. This will join the execution path of every contained thread, collecting possible errors along the way in a list. If any error occurred, this list is returned. Otherwise, the result is an empty *Ok*, simply indicating success.

Listing 4.35: Shutdown the scheduler

```

pub fn invoke_termination(&self) {
    self.running.store(false, Ordering::Release);
    for (handle, _) in self.handles_senders.iter() {
        handle.thread().unpark();
    }
}

pub fn join(self) -> Result<(), Vec<Box<dyn Any + Send>>> {
    let mut results = Vec::new();
    for (handle, _) in self.handles_senders {
        if let Err(error) = handle.join() {
            results.push(error);
        }
    }
    if results.is_empty() {
        Ok(())
    } else { Err(results) }
}

```

The scheduler implementation discussed above combines the advantages of different concurrent programming techniques. Atomic operations, message passing and mutual exclusion on shared state cooperate with each other which eventually leads to simple, scalable concurrent code.

4.3 JavaScript

JavaScript is a dynamically typed, single-threaded programming language. As opposed to Rust, it hardly poses any constraints on sharing and mutating states. JavaScript is asynchronous by design which makes it interesting for concurrent programming.

Nowadays, developing JavaScript applications without the *node.js*¹⁶ runtime is unthinkable. Thus, this section approaches the discussed topics as if *node.js* were a native component of JavaScript. Node.js is strongly optimized for I/O operations which is why it is gaining popularity for server-side applications. While the main program runs on a single thread, external operations like calling operating system routines can be parallelized.

4.3.1 Event loop

Functions in JavaScript are first-class citizens. In fact, while classes are supported, functions are the foundation of JavaScript code. After all, JavaScript is primarily designed as a scripting language for dynamic web applications. Short scripts are used to manipulate the HTML DOM, create animations, handle user invoked events and fetch data from external sources.

Higher-order functions are prominently used throughout development leading to highly asynchronous software. Listing 4.36 shows the usage of the function *setTimeout*. It expects two arguments, a function, referred to as *callback*, and a timeout specified in milliseconds. The callback is executed after the given timespan has elapsed. Calling *setTimeout* returns a handle to the timer. Through *clearTimeout* the execution of the specified callback can be canceled.

Listing 4.36: The timer function *setTimeout*

```
const timer = setTimeout(() => {
  ...
}, 100)
...
if (shutdown)
  clearTimeout(timer)
```

JavaScript takes a radically different approach to concurrency than multithreaded languages do. A single thread, called *event loop*, runs all the regular JavaScript code. Each statement creates a new frame on the *Call Stack*. In each iteration, the event loop executes every frame on the call stack one after another. This is depicted in Figure 4.1.

Calling long-running, asynchronous functions, such as reading and writing files or interacting with the network would usually block the current thread. In a single-threaded environment consistently updating the user interface, blocking would have a devastating impact on the usability. Thus, JavaScript handles these operations differently: The execution takes place externally, in a separate thread pool. Each operation expects a callback which is triggered upon completion.

¹⁶<https://nodejs.org/>

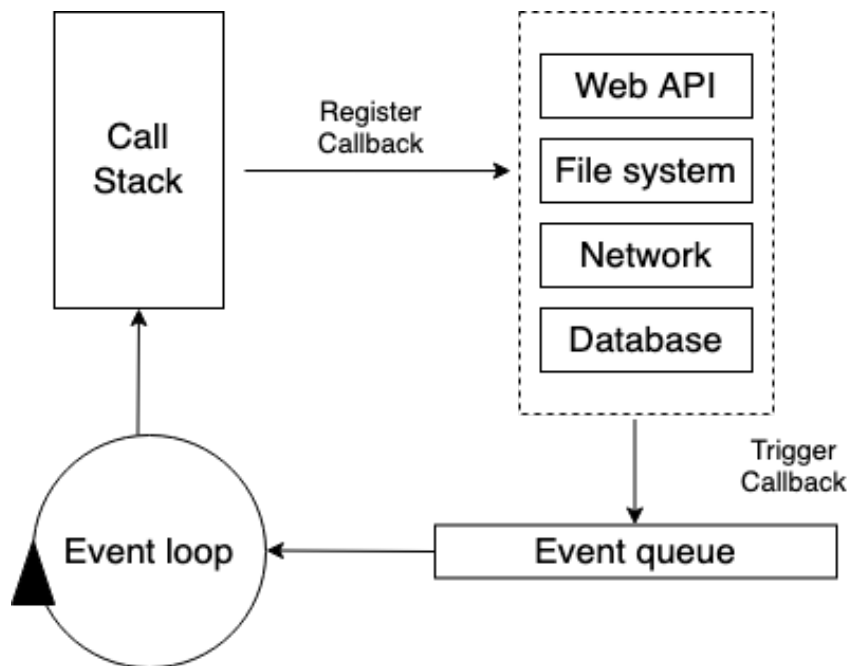


Figure 4.1: JavaScript's Event loop

These callbacks can be seen as asynchronous tasks. They are not executed immediately, but are stored in a queue. The event loop polls the queue for new functions to execute once the call stack is empty. This approach enables progress throughout the whole application.

Listing 4.37: Register an event listener

```
...
Element.addEventListener('click', () => { alert() })
...
```

The name event loop stems presumably from JavaScript's native event system. These events are triggered by user interactions with the UI. To react on an event, a listener has to be registered via the function *addEventListener*, shown in Listing 4.37. It expects the type of event (encoded as string) and a callback to be run when the event was triggered. The respective callback is, again, enqueued and executed once polled by the event loop.

As a consequence of the asynchronous nature the event loop entails, the order of execution is non-deterministic. Recall the function *setTimeout*. The given callback should apparently be run once the timeout is over. This is, however, not the case. Instead, it is sent to the event queue. Therefore, the given timespan is only the minimum delay.

Error handling in asynchronous functions Especially when external resources are involved, things can go abroad. Consider trying to read a file which does not exist, or the current user is not allowed to access. This attempt will obviously result in an error.

To handle occurring errors, the supplied callback usually expects —besides the result of a successful execution — a nullable object representing the error. Listing 4.38 illustrates a scenario where the client code first examines whether any error occurred. If so, a function is called to handle it properly. Otherwise, the result is processed as intended.

Listing 4.38: Reading a file

```
import fs from 'fs'; // import file system utils from node

fs.readFile('./text.txt', (err, data) => {
  if (err) {
    handleError(err)
  } else {
    processFileContent(data)
  }
})
```

4.3.2 Promise

With the built-in type *Promise*, JavaScript provides a versatile mechanism to handle asynchronous functions. As the name implies, it promises that the resulting value of a computation will be available at some point in the future. To construct a new promise, a function is supplied which takes two arguments: *resolve* and *reject*. A successful *Promise* is said to resolve, a failing one to reject. Listing 4.39 illustrates the creation of a new instance.

Listing 4.39: Constructing a Promise

```
const dbConnection = new Promise((resolve, reject) => {
  ...
  if (failure)
    reject('Failed to connect')
  else
    resolve(connection)
})
```

Working with *Promise* is convenient. Several asynchronous function calls can be chained together, using the previous result. In Listing 4.40¹⁷ the response of an HTTP request is handled this way. The method *then* expects the result of the preceding asynchronous operation, constructing yet another instance of *Promise*. Retrieving the actual data from the request, that is, calling *response.json* is also asynchronous and hence processed via the second *then*.

Similar to synchronous code, errors are handled through the *catch* method.

Listing 4.40: Promise chain

```
import fetch from 'node-fetch'; // node-fetch is a separate module

fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then(response => {
    if (!response.ok) {
      throw new Error('Request failed.')
    }
    return response.json()
  })
  .then(result => handleResult(result))
  .catch(err => handleError(err));
```

Combining the callback approach to asynchronous functions with promises may lead to overly complex code and unexpected behavior. It is, however, quite simple to transform a traditional function into a *Promise*. Listing 4.41 shows how this is done for *setTimeout*.

¹⁷<https://jsonplaceholder.typicode.com/> provides a REST API to retrieve mock data. This is useful for rapid prototyping.

Instead of the function to be run after the timeout, *resolve* is supplied as callback. Through chaining, the respective function is still executed once the timespan has elapsed.

Listing 4.41: Promisification

```
new Promise(resolve => {
  setTimeout(resolve, 100)
}).then(deferredFunction())
```

Async/await Chaining promises is a fine way of handling asynchronous function calls, as each succeeding invocation of *then* runs once the previous result is available. Yet, sometimes it might be necessary to wait for a *Promise* to resolve. This is achieved via the keyword *await*. It suspends execution until the asynchronous operation has finished.

As shown in Listing 4.42, this approach resembles a synchronous programming style. Error are handled by a regular *try-catch* block. However, using *await* is only permitted in functions declared as *async*.

Listing 4.42: Async/await

```
async function getData(url) {
  const response = await fetch(url)
  if (!response.ok)
    throw new Error('Request failed.')
  return await response.json()
}
...
try {
  const data = await getData(url)
  handleResult(data)
} catch (err) {
  handleError(err)
}
```

Parallel execution JavaScript provides several functions to run different asynchronous functions in parallel. One of which is *Promise.all*. It expects an iterable object of promises, such as an array, and parallelizes their execution. A single promise is returned which resolves once all the supplied promises have resolved, resulting in an array containing the respective result values. Listing 4.43 illustrates the usage.

The function *Promise.any* works analogously, but it resolves with the value of the promise that resolves first.

Listing 4.43: Promise.all

```
const results = await Promise.all([
  getData(url1),
  getData(url2),
  getData(url3)
])
results.map(res => handleResult(res))
```

4.4 Go

Go is an imperative, general purpose programming language. Due to its low-level facilities, it is suited for system programming. Object-oriented programming is partially supported through interfaces.

First and foremost, however, Go is renowned for its scalable concurrency model. Green threads, light weighted tasks, build the foundation of concurrent programming in Go. These so called *goroutines* are scheduled onto a thread pool by the runtime system. Go's scheduler is highly sophisticated. For instance, when a thread blocks due to a longer lasting external operation, goroutines associated to it might be re-scheduled to other threads¹⁸.

In the Go developer community, there is a typical proverb: *Don't communicate by sharing memory, share memory by communicating*. Goroutines do primarily communicate via message passing which is supported by built-in channels.

Conventional synchronization While message passing and constructs built on top of it suffices for synchronization purposes in most cases, sometimes conventional synchronization mechanisms are still necessary. For instance, mutual exclusion on a cache accessed concurrently must be upheld nonetheless. Go provides several concurrent data structures in the *sync* package. Atomic operations on integer types are given in the *sync/atomic* package.

Usage of data structures, such as *Mutex* or *WaitGroup* (similar to semaphores) is straightforward and does not significantly differ from other languages. Hence, they are not discussed in this work.

4.4.1 Goroutines

As mentioned above, goroutines are light weighted tasks. Built into the language, creating them is simple. As illustrated in Listing 4.44 a goroutine is a regular function scheduled via the keyword *go*. It is also possible to run an anonymous closure as goroutine. The closure is run right after its declaration.

Many concurrent operations in Go are blocking. However, it is the goroutines that block, not the threads they are scheduled on. This increases throughput of the software as a whole.

Listing 4.44: Channel as a semaphore

```
func task() {
    // long-running work
}
func main() {
    go task()
    go func() {
        // Do work in a closure
    }()
    ...
}
```

Goroutines combined with channels encompass all the capabilities of the actor model: Goroutines can create other goroutines, send messages, handle received messages accordingly and store their local state.

4.4.2 Channels

Like goroutines, channels are directly built into the language, represented by the type *chan*. A channel in Go is statically typed, meaning only messages of the specified type can be sent and received.

¹⁸Dmitry Vyukow held an excellent talk on the Go scheduler at the Hydra conference in 2019 (<https://www.youtube.com/watch?v=-K11rY57K7k>, accessed 27. February 2022).

Listing 4.25 depicts a simple Go program using a channel to retrieve the result from a separate task. A channel for integer values is created through the call `make(chan int)`. The function `task`, scheduled as goroutine, expects an integer channel. The input parameter is defined as `chan←` which indicates a channel only used for sending. It is said to be *send-only*. Analogously, a channel annotated with `leftarrowchan` is *receive-only*. The result of the computation is received from the channel via `←c`.

Listing 4.45: Sending message via channels

```
import "fmt"
func task(c chan← int) {
    ...
    c <- result
}

func main() {
    c := make(chan int)
    go task(c)
    var result = <-c
    fmt.Println("Result", result)
}
```

Channels in Go are synchronous by default. Thus, send and receive operations will block until the counterpart is executed. Nonetheless, it is possible to create a bounded channel which uses a buffer to store incoming messages. Sending is therefore asynchronous as long as the buffer's capacity is not reached. Listing 4.46 shows how to create an asynchronous channel. In addition to the channel type, the function `make` also takes an optional argument of type integer specifying the size of the buffer.

Listing 4.46: Creating an asynchronous channel

```
c := make(chan int, size)
```

Closing a channel When a channel has served its purpose, that is, all intended messages have been sent, it is reasonable to shut it down. To do so, the function `close` is called on the respective channel. This, however, should only be done by the sender, because sending on a closed channel will result in an error. Receiving a message actually returns a tuple consisting of the concrete value and a boolean flag telling whether the channel is still alive. For convenience, the second part can be omitted. Closing a channel will send a specific tuple containing the *null* value of the respective type and *false*.

Listing 4.47: Iterating receive

```
func task(c chan← int) {
    for i := 0; i < 10; i++ {
        ...
        c <- partialResult
    }
    close(c)
}

func main() {
    c := make(chan int)
    go task(c)
    var result = 0
    for number := range c {
        result += number
    }
}
```

```

    }
    ...
}

```

Shutting down a channel is often paired with an iteration over received messages. The *for ... range* expression can be used on channels, as illustrated in Listing 4.47. A goroutine sends partial results during each iteration. Once it is done, it closes the channel. The *range* over the channel in *main* will continue until it receives the close message. This ensures the receiver that no more messages will be sent.

Selecting channels With *select* Go provides a mechanism to choose a channel to receive from or to send to. It works quite similar to a *switch* statement. The first channel holding a message is selected. Should none of the channels be ready, *select* blocks until one of the channels becomes available. Sample code is given in Listing 4.48. To avoid blocking altogether, a *default* case can be specified.

Listing 4.48: The select statement

```

chanA := make(chan int)
chanB := make(chan int)
...
select {
case fromA := <-chanA:
    ...
case fromB := <-chanB:
    ...
}

```

4.4.3 Channel-based primitives

Channels are a versatile tool for concurrent programming. Using several channels in combination allows for fine-grained control flow. Consider, for instance, a scenario wherein goroutine A can only progress further once goroutine B is ready. B notifies A, indicating to continue its work. Listing 4.49 sketches the code for this. Typically, simple notifications are represented by a blank struct, because they are zero-sized which means their values do not consume memory.

Listing 4.49: Notification via channels

```

type T = struct{}

func A(ready <-chan T) {
    ...
    <- ready
    // Continue work
}
func B(ready chan T) {
    // Perform preparations
    ready <- struct{}{}
}

```

The channel *ready* is used to transfer the notification from goroutine B to goroutine A which will block until it is notified. It is a standard means to use synchronous channels to discontinue goroutines at a certain point.

Note the resemblance to the concept of a semaphore. Go's common library, indeed, provides several programming primitives based on channels. One of which is the function

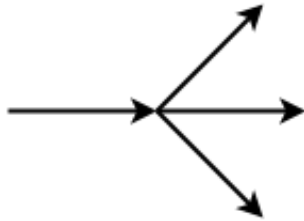


Figure 4.2: Fan-out

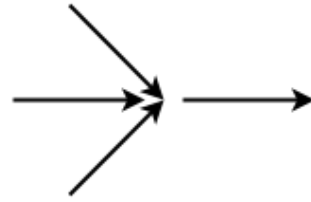


Figure 4.3: Funnel

After from the *time* package. It expects an argument of type *time.Duration* and returns a channel. Once the specified timespan has elapsed, an internally scheduled goroutine sends on this channel.

After can be used to receive from a channel with timeout. Listing 4.50 illustrates this. The function *receive* takes a receive-only channel and the respective timeout duration. It returns a tuple containing the data and a boolean flag indicating whether the receive was successful before the timeout. The *select* statement is used to race between receiving from *inputChannel* and *After*.

Listing 4.50: Receive with timeout

```
import "time"

func receive(
    inputChannel <-chan int,
    timeout time.Duration,
) (int, bool) {
    select {
    case input := <-inputChannel:
        return input, true
    case <- time.After(timeout):
        return 0, false
    }
}

func main() {
    c := make(chan int)
    result, ok := receive(c, 2*time.Second)
    ...
}
```

4.4.4 Message routing topologies

Channels can be seen as streams of data. Some goroutines are therefore used to route the data along. Two common topologies are *Fan-out* and *Funnel*, illustrated in Figure 4.2 and Figure 4.3.

Fan-out is used to distribute data from one channel to multiple other channels. This approach is useful for load balancing when data is created frequently but processing it takes longer. *Funnel*, on the other hand, converges the data flow from several channels into a single one. Consider a scenario where multiple channels are used to notify about the occurrences of an event which are all handled at one place. Here, *Funnel* comes in handy. The code for both functions is given in Listing 4.51.

Listing 4.51: Fan-out and funnel

```

func fanout(input <-chan int, outputA, outputB chan<- int) {
    for data := range input {
        select {
            case outputA <- data:
            case outputB <- data:
        }
    }
}

func funnel(inputA <-chan int, inputB <-chan int, output chan<- int) {
    for {
        select {
            case data := <-inputA:
                output <- data
            case data := <-inputB:
                output <- data
        }
    }
}

```

4.5 A comparison of the languages

The different programming languages discussed in this chapter all employ their own notion of concurrent programming. Some approaches are quite alike, while others differ strongly. This section compares the concurrent programming features of these languages.

Multithreading In Java and Rust, threads are created explicitly by providing a function executed in the respective thread. The *Thread* class in Java is, however, more versatile than its counterpart in Rust and allows for extended control over the thread. Furthermore, threads in Java can be separated into different groups which provides an additional layer of security.

JavaScript is single-threaded. Nonetheless, external function calls, such as network operations, are executed outside the *event loop* in a thread pool. Also, the execution of several promises can be parallelized.

Go is implicitly multithreaded. While developers do not create threads on their own, goroutines are scheduled onto a pool of threads by Go's runtime system.

Tasks With *Future* and *CompletableFuture* Java provides an object-oriented approach to light weighted tasks. Custom data structures providing similar functionalities are easily implemented in Rust.

In JavaScript tasks are created via *Promise* and callbacks. Goroutines are Go's built-in task facility.

Language support for concurrent programming The keyword *synchronized* is the standard way of ensuring mutual exclusion in Java. Declaring a variable as *volatile* makes changes to it visible among separate threads. Rust's ownership model and the marker traits *Send* and *Sync* provide a high level of thread safety.

Channels for message passing are directly built into Go. They are the primary mechanism to interact between and to synchronize goroutines.

Advanced standard library support Java's standard library is high in concurrent data structures. Among them are several lock implementations suited for different scenarios, blocking and non-blocking collections of all kinds and base implementation on top of which custom synchronization facilities can be developed. Atomic operations on primitive data types, arrays and references are also provided.

Rust only includes a few concurrent data structures into the standard library. This is due its concept of *fearless concurrency* and the fact that Rust binaries are intended to be as small as possible.

Likewise, Go provides a rather limited set of concurrent data structures and operations. However, channels already cover many synchronization needs.

This thesis has aimed to explain concurrent programming in a simple, understandable manner. A broad variety of concepts used in concurrent programming was discussed, ranging from thread states, atomic operations, interleaved execution and synchronization mechanisms to advanced techniques and models like message passing, the Producer-Consumer pattern or the actor model. A profound understanding of these concepts is vital for the development of concurrent software.

Further, this work has pointed out potentials for errors when dealing with concurrency. Solving one problem, mutual exclusion on shared state for instance, can lead to other problems like deadlocks and performance issues.

Concrete approaches to concurrent programming in several programming languages were laid out with an emphasis to show the application of several concepts in combination. All these languages have their strengths and weaknesses and none is superior to the others, but are respectively suited for specific needs.

This thesis has shown that the combination of atomic operations, shared state and message passing can alleviate the complexity concurrency entails.

5.1 Concurrent by design

Concurrency is not simply a runtime property of a program, it is about design. Indeed, the principal mechanisms of dealing with concurrency should be manifested into the architecture of a software system. It is important to reason about synchronization, safety and liveness and when to use a certain technique or data structure.

In a properly designed concurrent software architecture, concurrent programming facilities should be viewed upon as low-level features implemented into the core of the application. This encompasses synchronization, visibility of state updates among threads, as well as the prevention of deadlocks and livelocks. As shown at several places in this work, abstraction and encapsulation can work wonders here.

Likewise, the creation and lifecycle management of tasks is equally important. Proven design patterns, such as the *Factory*, *Builder* or *Fluent Interface* are well suited to create tasks in specific contexts. Higher-order functions in general are very useful for concurrent programming. But they are especially beneficial when used for pipelines on tasks to compose

them and handle their results, as well as potential errors. A great example for this approach is the Java class *CompletableFuture*.

A primary aspect of high quality software architecture is the testability of each component a program is built of. Using non-responding programming facilities like Java's *Runnable* makes testing difficult. It is therefore preferable to define tasks as simple functions returning a value. This way the results of concurrent operations can be tested quite simple in isolation.

5.2 Outlook

Much has been said about concurrent programming in this work and much is left to be said. There are many other programming languages employing their own notion of concurrent programming. Elixir, for instance, has the actor model integrated even deeper than Go. Julia brings its very own task model with it. Pure functional languages like Haskell entail other benefits for dealing with concurrency due to immutability of state.

Many third-party libraries do also provide extended support for concurrent programming. The library *Akka* supplies an implementation of the actor model for Java. With supervision [Lev84], *Akka* goes even beyond the actor model. A supervisor in this sense keeps track of subordinate actors and employs recovery strategies when they throw exceptions.

Applications like Hadoop employ the *MapReduce* pattern on a large scale [MJ15, DG08]. This enables processing huge amounts of data on server clusters in a fairly simple pipeline of operations.

Bibliography

- [86512] ISO/IEC 8652:2012. International organization for standardization: Ada. Technical report, International Organization for Standardization, 2012.
- [ABDC⁺18] Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, page 51–60, New York, NY, USA, 2018. Association for Computing Machinery.
- [ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.
- [Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
- [Akm90] Varol Akman. Book review: Actors: A model of concurrent computation in distributed systems. *AI Magazine*, 10(4), 1990.
- [AS85] Bowen Alpern and Fred Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 10 1985.
- [BD80] Randal Bryant and Jack Dennis. Concurrent programming. In *Operating Systems Engineering*, pages 426–451, 1980.
- [BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 37(11):211–230, 2002.
- [BPR22] Federico Bergenti, E Panegai, and G Rossi. A master-slave architecture to integrate sets and finite domains in java. *Presented at CILC'06 – Convegno Italiano di Logica Computazionale, Bari*, 2022.
- [Buo21] Enrico Buonanno. *Functional Programming in C#*. Manning Publications, 2 edition, 2021.
- [Bus95] Frank Buschmann. The master-slave pattern. In *Pattern-oriented Software Architecture*, pages 133–142, 1995.
- [CWG09] Martin J Chorley, David W Walker, and Martyn F Guest. Hybrid message-passing and shared-memory programming in a molecular dynamics application on multicore clusters. *COMPUTING APPLICATIONS The International Journal of High Performance Computing Applications*, 23:196–211, 2009.

- [DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
- [DGFG03] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Shared memory vs message passing. Technical report, I&C - School of Computer and Communication Sciences, 2003.
- [Dij68] Edsger W. Dijkstra. Cooperating sequential processes. *The Origin of Concurrent Programming*, pages 65–138, 1968.
- [Dix22] Andrew Dixon. *Divide and Conquer*, pages 37–40. Auerbach Publications, 2022.
- [Elm86] Ahmed K. Elmagarmid. A survey of distributed deadlock detection algorithms. *SIGMOD Rec.*, 15(3):37–45, sep 1986.
- [Erb12] Benjamin Erb. Concurrent programming for scalable web architectures. Diploma thesis, Institute of Distributed Systems, Ulm University, 2012.
- [FC12] Scott Frame and John Coffey. A comparison of functional and imperative programming techniques for mathematical software development. *Journal of Systemics, Cybernetics and Informatics*, 2:1–4, 2012.
- [Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman, Amsterdam, Boston, MA, USA, 2002.
- [FXL03] Vincent W Freeh, Jin Xu, and David K Lowenthal. Hybrid messaging passing in shared-memory clusters. Technical report, Department of Computer Science, University of Georgia, 2003.
- [GBB⁺06] Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, and Tim Peierls. *Java Concurrency in Practice*. Addison-Wesley Longman, Amsterdam, 2006.
- [GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Longman, Amsterdam, Reading, MA, 1995.
- [GMUW09] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book*. Pearson Education, 2 edition, 2009.
- [HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. *IJCAI*, pages 235–245, 1973.
- [Her96] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13, 03 1996.
- [Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.
- [Hew10] Carl Hewitt. Actor model of computation: Scalable robust information systems, 2010.
- [HHL⁺07] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. *Parallel Processing Letters*, 17:411–424, 2007.

- [HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. *Proceedings - International Conference on Distributed Computing Systems*, pages 522–529, 06 2003.
- [Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.
- [HSD10] Andreas Holzinger, K.H. Struggl, and Matjaž Debevc. Applying model-view-controller (mvc) in design and development of information systems: An example of smart assistive script breakdown in an e-business application. *ICE-B 2010 - ICETE The International Joint Conference on e-Business and Telecommunications*, pages 1 – 6, 2010.
- [HSY04] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, page 206–215, New York, NY, USA, 2004. Association for Computing Machinery.
- [HW90] Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12:463–, 07 1990.
- [JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 295–308, 1996.
- [KJA⁺02] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiawicz, and Beng-Hong Lim. Integrating message-passing and shared-memory: Early experience. *ACM SIGPLAN Notices*, 28:54–63, 2002.
- [KKA14] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, 2014.
- [KL94] A. C. Klaiber and H. M. Levy. A comparison of message passing and shared memory architectures for data parallel programs. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, page 94–105, Washington, DC, USA, 1994. IEEE Computer Society Press.
- [Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.
- [LBL⁺16] Yuheng Long, Mehdi Bagherzadeh, Eric Lin, Ganesha Upadhyaya, and Hriday Rajan. On ordering problems in message passing software. In *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, page 54–65, New York, NY, USA, 2016. Association for Computing Machinery.
- [LDS07] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, pages 2–es, New York, NY, USA, 2007. Association for Computing Machinery.
- [Lea00] Doug Lea. A java fork/join framework. In Dennis Gannon and Piyush Mehrotra, editors, *Proceedings of the ACM 2000 Java Grande Conference*, San Francisco, CA, USA, June 3-5, 2000, pages 36–43. ACM, 2000.

- [Lev84] Henry Levy. *Early Capability Architectures*, pages 41–64. University of Washington Computer Science and Engineering, 12 1984.
- [LW11] Christopher J Lieb and Craig Wills. Concurrent programming in education: Time for a change. Technical report, Polytechnic Institute, Worcester, 2011.
- [Mar96] Robert C. Martin. The dependency inversion principle. *C++ Report*, 8:61–66, 1996.
- [MJ15] Seema Maitrey and C. K. Jha. Mapreduce: Simplified data analysis of big data. *Procedia Computer Science*, 57:563–571, 2015.
- [MK06] Jeff Magee and Jeff Kramer. *Concurrency: State Model and Java Programs*. John Wiley & Sons, New York, 2 edition, 2006.
- [MNM14] Benjamin Morandi, Sebastian Nanz, and Bertrand Meyer. Safe and efficient data sharing for message-passing concurrency. In *COORDINATION*, 2014.
- [MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In James E. Burns and Yoram Moses, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 267–275. ACM, 1996.
- [Nie07] Piotr Nienaltowski. Practical framework for contract-based concurrent object-oriented programming. Doctoral thesis, ETH Zurich, 2007.
- [NS08] Stefan Näher and Daniel Schmitt. A framework for multi-core implementations of divide and conquer algorithms and its application to the convex hull problem. *Proceedings of the 20th Annual Canadian Conference on Computational Geometry, CCCG 2008*, 2008.
- [OAE07] Felix Ogban, Iwara Arikpo, and Idongesit Eteng. Von neumann architecture and modern computers. *Global Journal of Mathematical Sciences* Vol. 6, No. 2, 2007, Page 97: ISSN 1596-6208 Indexed and abstracted on AJOL (UK): <http://www.ajol.info>, Vol. 6, No. 2, 2007, Page 97, 2007.
- [PTF⁺15] Gustavo Pinto, Wesley Torres, Benito Fernandes, Fernando Castor, and Roberto S.M. Barros. A large-scale study on the usage of java’s concurrent programming constructs. *Journal of Systems and Software*, 106:59–81, 2015.
- [Rob22] Martin Robillard. *Introduction to Software Design with Java*, chapter 8, pages 195–242. Springer, 2022.
- [Ros17] Kevin Rosendahl. Green threads in rust. Master’s thesis, Stanford University, Computer Science Department, 2017.
- [Sco13] Michael Scott. Shared-memory synchronization. *Synthesis Lectures on Computer Architecture*, 8:1–221, 2013.
- [SG97] Sahiba Sheikh and R. Ganesan. Replication of multimedia data using master-slave architecture. *COMPSAC ’97. Proceedings., The Twenty-First Annual International*, pages 66 – 70, 1997.
- [Sin97] Pradeep K. Sinha. *Distributed Operating Systems: Concepts and Design*. Wiley-IEEE Press, 4 edition, 1997.

- [SLS09] William N. Scherer, Doug Lea, and Michael L. Scott. Scalable synchronous queues. *Communications of the ACM*, 52:100–108, 2009.
- [SS84] Richard D. Schlichting and Fred B. Schneider. Using message passing for distributed programming: Proof rules and disciplines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6:402–431, 1984.
- [SS04] William Scherer and Michael Scott. Nonblocking concurrent data structures with condition synchronization. In *DISC*, volume 3274, pages 174–187, 2004.
- [Str14] Robbie Strickland. *Cassandra High Availability*. Packt Publishing, Open Source, 2014.
- [TB14] Andrew S. Tanenbaum and Herbert Bos. Modern operating systems. *Education*, 2:1137, 2014.
- [Tur13] Aaron Turon. Understanding and expressing scalable concurrency. Doctoral thesis, Faculty of the College of Computer and Information Science, Northeastern University, Boston, Massachusetts, 2013.
- [ZS05] Marco Zennaro and Raja Sengupta. Distributing synchronous programs using bounded queues. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, page 325–334, New York, NY, USA, 2005. Association for Computing Machinery.