FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Real-Time Performance Analysis of Synchronous Distributed Systems

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

DI Alexander Kößler
Matrikelnummer 0325498

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Ulrich Schmid

Diese Dissertation haben begutachtet:

(Univ.Prof. Dr. Ulrich Schmid)

(Prof. Dr. Krishnendu Chatterjee)

Wien, Oktober 2014

(DI Alexander Kößler)

Erklärung zur Verfassung der Arbeit

DI Alexander Kößler
Wasserburgergasse 5/5, 1090 Wien

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

(Ort, Datum) (Alexander Kößler)

Acknowledgments

FIRST OF ALL, I would like to thank my adviser Ulrich Schmid. He provided the conditions for fruitful research in the Embedded Computing Systems Group and his distributed systems lecture opened my mind and caught my interest for this research topic. Discussions with him were always very helpful and his extensive and nevertheless detailed knowledge about literature is amazing and very valuable.

Furthermore, I would like to express my gratitude to the co-authors of the publications that lead to this thesis. Particularly I would like to thank Matthias Függer, Thomas Nowak, and Martin Zeiner, for their support when we wrote the synchronizer papers, as well as Krishnendu Chatterjee and Andreas Pavlogiannis for their support when writing the real-time scheduling papers.

I would like to thank Josef Widder for his support during all the years. Discussions with him have always been very fruitful and his comments to improve this thesis were very valuable.

I would also like to thank Heinz Deinhart for keeping the working infrastructure up and running, as well as, Traude Sommer for her pragmatic way of solving administrative barriers. Furthermore, I want to thank my colleagues from the Embedded Computing Systems Group for their companionship during my stay at the department and their feedback to this thesis.

Last but not least, I would like to thank all my Friends and my Family, especially Johanna, for their continuous support during all the years of my studies.

This work was partially supported by grants P21694 and P20529 as well as by the NFN RiSE S11405 of the Austrian Science Foundation (FWF).

Alexander Kößler

Vienna, October 2014

for Johanna

Abstract

THE TIME IT TAKES for an algorithm to perform its task is a central question in computer science. It has been extensively studied for (centralized) sequential algorithms, while a comprehensive treatment of time complexity in the distributed setting is still lacking. In synchronous round-based distributed algorithms, the number of rounds until the problem is solved represents a performance measure analogous to standard time complexity (Newtonian real-time), but puts under the rug the many intricacies that occur in a round due to faults, message passing, timing uncertainties due to asynchrony etc.

This thesis provides a novel framework for analyzing distributed systems running multiple independent distributed algorithms simultaneously on a shared message passing communication infrastructure. In particular, the previously mentioned timing uncertainties are addressed, and the performance of the executed algorithms are analyzed with respect to Newtonian real-time. To do so, the internals of a round, that is, transmitting and receiving messages, must be taken into account. We study these mechanisms with different mathematical tools.

On the transmitter side, a real-time scheduler responsible for scheduling the messages of multiple algorithms on the shared communication channels. Since suitable fault-tolerance techniques allow to deal with dropped messages in a synchronous distributed algorithm, messages can be modeled as firm deadline jobs with the end of a round as their deadline. Obviously, a good scheduling algorithm maximizes the cumulative utility gained by the successfully scheduled jobs. The quality of a scheduler is usually characterized by its competitive factor, that is, the performance of the scheduler with respect to an optimal clairvoyant scheduler, that knows the future. This thesis lays the foundation for a new approach to automatically perform the competitive analysis of scheduling algorithms with respect to given task sets. This is done by using a reduction to the problem of finding minimum mean-weight-cycles in multi-objective graphs. In addition, albeit being computationally hard, algorithmic game theory also allows to synthesize optimal scheduling algorithms.

On the receiver side, a synchronizer algorithm can be used to maintain a consistent round structure by compensating messages dropped by the scheduler. The performance of a distributed algorithm running atop of such a thus synchronizer directly depends on the performance of the synchronizer. Abstracting dropped (and otherwise lost) messages by a probabilistic link failure model allows to calculate the expected round duration using Markov theory. By analyzing the series of the starting times of the rounds generated by the synchronizer, and by modeling its execution as a Markov chain, this thesis finally develops results regarding the expected round duration.

Kurzfassung

EINE DER ZENTRALEN Fragen der Informatik ist, wie lange ein Algorithmus braucht, um seine Aufgabe zu erfüllen. Während diese Frage in (zentralisierten) sequenziellen Algorithmen bereits ausgiebig untersucht wurde, sind in verteilten Algorithmen noch viele Fragen zur Zeitkomplexität offen. In synchronen rundenbasierenden verteilten Algorithmen stellt die Rundenanzahl bis zur Termination ein zur klassischen Zeitkomplexität analoges Maß dar, jedoch werden dabei viele Feinheiten unter den Teppich gekehrt, die zum Beispiel durch Fehler, das Kommunikationssystem oder Ungewissheiten der zeitlichen Abfolgen durch Asynchronität während einer Runde auftreten.

Die vorliegende Arbeit stellt eine neuartige Analyse verteilter Systeme vor, die mehrere unabhängige verteilte Algorithmen nebenläufig auf einer gemeinsamen Kommunikationsinfrastruktur ausführen. Dabei wird besonders auf die bereits angesprochenen Ungewissheiten der zeitlichen Abfolgen eingegangen und die Leistungsfähigkeit der Algorithmen in Bezug auf das Echtzeitverhalten analysiert. Das Hauptaugenmerk liegt dabei auf den Interna einer Runde, konkret dem Senden und Empfangen von Nachrichten, die mit verschiedenen mathematischen Werkzeugen analysiert werden.

Um den Zugriff der Algorithmen auf die gemeinsam verwendeten Kommunikationskanäle zu steuern, wird ein Echtzeitscheduler verwendet. Geeignete Fehlertoleranz-Mechanismen erlauben es, verloren gegangene Nachrichten in synchronen verteilten Algorithmen zu tolerieren. Daher können Nachrichten als Arbeitspakete aufgefasst werden, die als Bearbeitungsfrist das Ende der Runde haben. Ein guter Echtzeitscheduler versucht, den kumulativen Ertrag zu maximieren, den er durch die fristgerechte Bearbeitung der Pakete erhält. Seine Konkurrenzfähigkeit wird üblicherweise in Relation zu einem optimalen hellseherischen Scheduler gemessen, der in die Zukunft sehen kann. Diese Arbeit legt den Grundstein für eine neue Methode zur automatischen Analyse der Konkurrenzfähigkeit von Schedulingalgorithmen. Dabei wird das Problem bei vorgegebenen Arbeitspakettypen auf das Finden von Kreisen mit minimalem durchschnittlichen Gewicht in einem Graphen reduziert. Weiters, wenngleich mit sehr großem Rechenaufwand verbunden, kann durch algorithmische Spieltheorie ein optimaler Schedulingalgorithmus synthetisiert werden.

Auf der Empfängerseite kommt ein Synchronisationsalgorithmus zum Einsatz, der vom Schduler verworfene gegangenen Nachrichten implizit kompensiert und wieder eine konsistente Rundenstruktur herstellt. Das Zeitverhalten eines darauf aufbauenden verteilten Algorithmus hängt stark von der Leistungsfähigkeit dieses Synchronisationsalgorithmus ab. Eine Abstraktion der verworfenen (oder sonstwie verlorengegangenen) Nachrichten in einem probabilistischen Kommunikations-Fehlermodell ermöglicht die Anwendung von Markoff-Theorie zur Berechnung der zu erwarteten Rundendauer. Die Analyse der Folge der Startzeiten der von dem Synchronisieralgorithmus generierten Runden durch die Modellierung als Markoff-Kette liefert schlussendlich Erkenntnisse über die zu erwarteten Rundendauern.

Contents

List of Figures

Chapter 4: Round Synchronization

Chapter 1

Introduction

TODAY, DISTRIBUTED SYSTEMS are ubiquitous. The most prominent example is the Internet: Our society cannot possibly be imagined anymore without this distributed system containing more than 903 million hosts.[1] But distributed systems not only exist in such large scales. A set of sensor nodes connected by a wireless ad-hoc network executing a distributed algorithm, e.g., for air pollution monitoring, or a set of autonomous robots cooperating, for example, in a robot soccer game, form distributed systems as well. Distributed systems also exist at the hardware level. In [Fü10, FS12] a distributed system is built from multiple self-contained hardware blocks that, by communicating with each other, establish a fault-tolerant clock in a System-on-Chip.

What are Distributed Systems?

There are various definitions of "what forms a distributed system." In this work, we adhere to the criteria given in [Gho06].

Multiple processes: There is no point in calling a single-process-system distributed. Therefore, a distributed system consists of at least two distinct processes executing their programs concurrently. Usually the processes of the distributed system are also spatially distributed by assigning every process a dedicated processing node.

Interprocess communication: Communication is one of the key elements in distributed systems. In message passing systems information between processing nodes, and thus the processes of an algorithm, can only propagate by sending and receiving

[1] According to the 2012 figures of the CIA World Factbook, U.S. Central Intelligence Agency, 2012, https://www.cia.gov/library/publications/the-world-factbook/rankorder/2184rank.html, accessed: 29/04/2014.

messages. Shared-memory systems are providing a different model for interprocess communication: All processes have access to a shared memory address space that is used for information exchange. In this work, we restrict ourselves to message passing systems; a communication graph indicates which processes can directly communicate with each other.

Disjoint local states: A process does not have access to the local state of any other process. As mentioned before, this work focuses on message passing systems, therefore systems in which processes have access to a shared-memory are not taken into account.

Collective goal: Processes must communicate with each other to meet a common goal. A typical goal in distributed computing, for example, is to decide on a common value.

Distributed Systems are Important

Over the last decades distributed systems have gained substantial importance. Besides scalable performance, the most prominent advantage of distributed systems is that they can be made fault-tolerant. The most powerful computing system using a single central process can be rendered unusable if this process fails. Unfortunately, there are faults like single event transients and bit-flips in memory cells caused by ionized particle hits, cross-talk and electromagnetic interference, or failures in the power supply that cannot be controlled or avoided by the system designer that can make the central process fail. Such single points of failure are an intrinsic problem in centralized systems and are clearly problematic in systems aiming for high availability and reliability requirements. To overcome this significant issue and hence to increase the robustness, distributed systems employ space redundancy.

General distributed systems literature, such as [Gho06], describes additional benefits of distributed systems like a geographically distributed environment to provide geographically distributed services, the possibility of advancing the system performance beyond the (physical) performance limitations of a single processing node, and better scalability. However, these issues are less relevant in the context of this thesis.

Distributed Systems are Challenging

In centralized systems, a decision like opening or closing a valve depending on the temperature is straightforward. The centralized process just fetches all the sensor readings it needs to decide if the valve should be opened or closed, and executes the corresponding action.

If a distributed system is required, e.g., because the requirements demand that a single point of failure has to be avoided, opening or closing the valve gets much more involved. Multiple processes have to communicate with each other to build up

a consistent view of the global system state. Additionally, this system state may be time-variant and thus change during the process of data collecting. Hence, the multiple sensor readings cannot be compared directly as replica determinism[2] cannot always be guaranteed and distributed consensus algorithms have to be used. Even worse, the valve has to be built in a fault-tolerant manner to ensure the (process controlling the) valve does not form the single point of failure.

Furthermore, while the access to a resource is straightforward in a centralized system, distributed systems might need to provide additional services such as mutual exclusion, synchronization, or atomic transactions for consistent access by all processes to shared resources.

Everything comes at a price, and so does distributing a system for the sake of fault tolerance. To tolerate f Byzantine-faulty processes (i.e., processes that do not adhere to their algorithms but may behave arbitrarily) in distributed agreement, at least a total number of $3f + 1$ processes are needed, together with additional restrictions on the minimal connectivity of the communication graph [LSP82]. When it comes to tolerating transient failures, additional non-trivial problems like the recovery of processes and their reintegration have to be taken care of.

This small example demonstrates some of the implications that arise when actually making systems fault-tolerant. It also indicates that the algorithms running on the processing nodes in the distributed system get much more involved, thus increasing the efforts required for proving them correct. While computer-aided verification of algorithms is already common practice in fault-free systems, for example, for the verification of device drivers in MS Windows and GNU/Linux as in [BBC+06], model checking of fault-tolerant distributed systems is only at its beginning [JKS+13].

1.1 Synchronous Distributed Systems: Theory vs. Reality

Figure 1.1 shows a computer scientist's "view" of a distributed system running four processes of a synchronous distributed algorithm $\mathcal{A}$ on four processing nodes. The structure of the distributed system is nicely shown by the communication graph representing the dedicated point-to-point links used for communication between the processes. The algorithm's processes are usually modeled as (possibly infinite) state machines, specified in pseudo code and manually or automatically proven correct. A synchronous distributed algorithm assumes a consistent and communication-closed round structure provided by the distributed system (i.e., if a message is sent by a process, it will be delivered to the receiver process in the same round or not at all, as there are no late messages). Unfortunately, a real-world implementation of this distributed system may look completely different.

[2] *Replica determinism* [Kop97] guarantees that replicated objects deterministically produce the same output at all times and thus is a prerequisite for some fault tolerance mechanisms like triple modular redundancy (TMR). In the introduced example, temperature readings of replicated sensor nodes usually are not replica determinant as their output may slightly differ. For example assume the correct sensor readings are 49.7 °C and 50.1 °C while the faulty one is 53.5 °C. TMR is useless in this case.

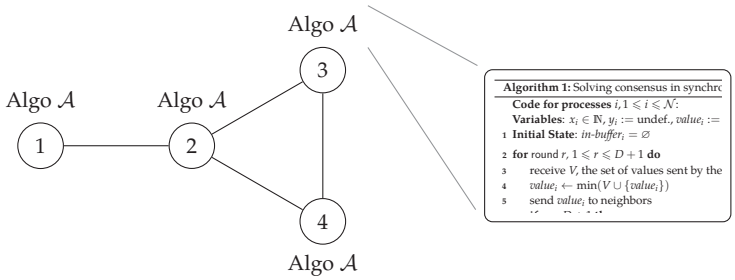


Figure 1.1: How a distributed system is usually analyzed.

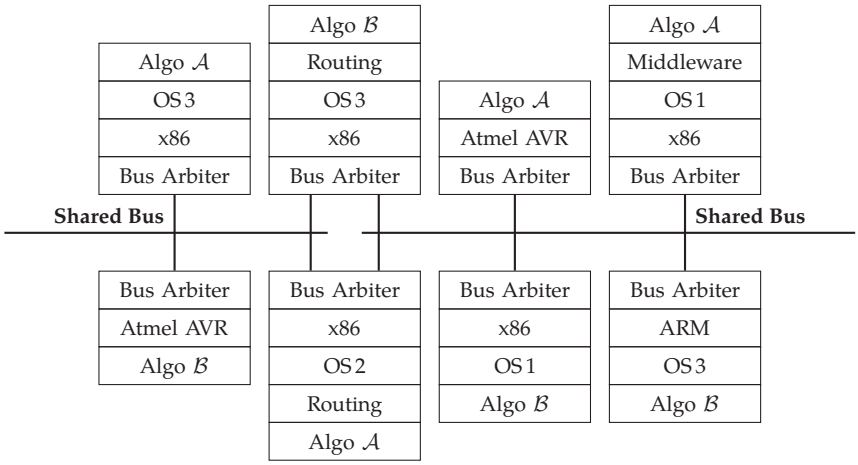


Figure 1.2: How a distributed system might end up being implemented.

As shown in Figure 1.2, the previously mentioned dedicated point-to-point links may be realized by a shared bus system. Therefore, every processing node needs some kind of bus arbitration to avoid collisions. Additionally, the pseudo code is not executed directly; rather, there are various computer architectures and operating systems involved in the implementation. There might also be other distributed algorithms executed in a shared environment, which are not considered in the original analysis and thus may make all timing assumptions about the bus access void.

In the following, some of the most common stumbling blocks that may arise when implementing a theoretically sound distributed system/algorithm in a real-world application are identified.

Unexpected timing uncertainties:

In the analysis of distributed systems, typically, only a single instance of a distributed algorithm is executed exclusively by the computation nodes in the system. This allows to abstract all timing related parameters such as network delay, message queuing, and the execution times of the processes, in the end-to-end delay of the communication (i.e., the duration between sending a message and processing it). An upper-bound on

this end-to-end delay is then used in the implementation as the round duration of the synchronous distributed system.

In reality, things may look a bit different: A shared communication medium, possibly needing bus arbitration, like a CAN-Bus in a car, and communication timeouts in combination with retransmission protocols applying additional load are not uncommon. Such implementations can easily result in violations of the synchrony assumptions used in the design of a synchronous distributed system, and thus in the failure of the algorithm.

Analysis of a distributed algorithm is based on its pseudo code specification. No operating system is mentioned, no middleware, no third-party-libraries adding execution time uncertainties are present. Unfortunately, this is not always true when it comes to the implementation of a distributed algorithm. Heterogeneous hardware, various operating systems combined with architecture dependent implementations or the need for additional middleware to compensate missing system features are adding additional timing uncertainties and thus challenges for the system designer when it comes to the implementation of an algorithm.

Faults and Errors:

When designing a fault-tolerant distributed system, one of the most crucial things is a valid fault hypothesis. A fault hypothesis describes exactly which types of faults a system has to deal with and how many/often faults may occur. Common types of faults are: (i) Faults regarding the execution of processes, such as crash faults, where a process prematurely stops executing the specified algorithm, and Byzantine faults, where a process deviates from the specified algorithm and can behave arbitrarily; (ii) Faults regarding communication, such as omission faults, where a process fails to send/receive a set of messages. An example for a fault hypothesis is: During the execution of a distributed algorithm at most f of its processes may crash. Clearly, if the fault hypothesis is not valid for the real-world application, e.g., because suddenly $f + 1$ processes crash or one process behaves Byzantine faulty, the correctness proofs of the distributed algorithm are useless for its real-world implementation.

Furthermore, there are hardware phenomena, like metastability in digital circuits, that are even stronger than Byzantine faults, as they cannot be captured inside fault-containment regions.[3] This is due to the fact that, even though a binary Byzantine signal can behave arbitrarily, it still has to adhere to the binary signal specification, while a metastable signal is an out-of-spec operation (cf. [FFS09]).

Finally, when it comes to the hardware implementation of a distributed system, additional challenges arise. Constructs regularly used in theory such as unbounded counters, e.g., used as sequence numbers for messages, or implicit mutual exclusion constraints, both present in [FFSS08], need to be treated very carefully to avoid implementation errors.

[3] A fault-containment region is the set of subsystems that can be affected by a single fault (cf. [Kop97]).

Scalability problems:

Manually proving a distributed algorithm correct for a system with four, 40 or 4 000 processes usually makes no difference. The number of processes is abstracted away by an ominous variable $\mathcal{N}$, and the correctness proofs usually hold for arbitrary large values of $\mathcal{N}$. When it comes to realization, missing or insufficient analysis of the algorithm, e.g., w.r.t. the memory usage, can make distributed systems fail. Algorithms working perfectly fine for four or 40 processes break when executed, for example, in a wireless network with 800 nodes, like the one deployed in the University of California, Berkeley,[4] as a poor implementation the routing table suddenly exceeds the 512 bytes of locally available main memory or because piggybacking of messages exceeds the maximal supported message size.

1.2 Questions and Contributions of this Thesis

This thesis mainly addresses the problem of unexpected timing uncertainties. It provides a framework for analyzing the real-time performance of multiple synchronous distributed algorithms running independently and in parallel on a common distributed system. We extend the common perception of computing nodes in a distributed system to allow them to execute processes of multiple distributed algorithms concurrently. By using a multi-core argument, we can motivate that the nodes' computational power is sufficient to run all the algorithms in parallel (i.e., we assign every algorithm its dedicated processor). As usual for distributed algorithms, the problem arises at the communication between the processing nodes where the cumulative bandwidth demand of the distributed algorithms executed on a node may (temporarily) exceed the capacity of the shared communication channels. We propose an approach that splits the problem into two parts.

On the transmitter side, sending various messages from multiple distributed algorithms via an unreliable shared communication channel may exceed the channel's capacity. Hence, not all messages will always be sent and thus received. However, this does not necessarily prohibit the fault-tolerant distributed algorithm to operate correctly system-wide. Our goal is to analyze the performance of message scheduling algorithms at the transmitter. To do so, we model the messages as real-time jobs with firm deadlines, i.e., jobs that do not harm if they miss their deadline (which is the end of the synchronous round), but do not provide any utility to the system in this case. A good scheduling algorithm is one which maximizes the cumulative utility in the worst-case scenario. This is reflected by a high competitive ratio w.r.t. the cumulated utility achieved by a clairvoyant scheduler that knows the future, which is determined by competitive analysis. We utilize a novel approach based on graph games here, which reduces the competitive analysis of a given on-line scheduling algorithm to solving certain problems on multi-objective graphs.

[4] http://webs.cs.berkeley.edu/800demo/, accessed: 12/05/2014.

Classically, the competitive analysis of an on-line algorithm involved considerable effort to find and to analyze worst-case scenarios, which are usually very problem specific. Using our algorithmic approach we can replace human ingenuity by computing power.

The analysis and modeling of the real-time scheduling problem was done in a joint work with Krishnendu Chatterjee, Andreas Pavlogiannis, and Ulrich Schmid. The author's contribution covers the modeling and the automated generation of the on-line algorithms' state spaces. The credits for the contributions of the state space reduction and solving the resulting graph problems are awarded to Andreas Pavlogiannis.

On the other hand, when it comes to receiving messages, we observe message loss as a result of the abandoned jobs by the real-time scheduler, as well as the typically unreliable communication channel. There are two ways to overcome this issue: The first one is to use a synchronous distributed algorithm that can tolerate receive omissions. The second one, which is the one pursued in this thesis, is to build a synchronous round abstraction (without receive omissions) atop the imperfect communication system, using a so-called synchronizer. Introduced by Awerbuch [Awe85], a synchronizer triggers the next round of the algorithm's process on each node only after (i) it is aware that all neighbors have received its associated message of the current round and (ii) that it has received all neighbors' messages of the current round. Clearly, the real-time performance of an atop running synchronous algorithm heavily depends on the performance of this synchronizer.

Abstracting both the dropped messages of the real-time scheduler as well as the lost messages of the unreliable channel via a probabilistic link failure model, parametrized by the results of the real-time scheduling analysis and the channel characteristics, finally allows us to derive non-trivial results on the expected round durations of the synchronous algorithms running atop.

The performance analysis of the synchronizer was done in joint work with Matthias Függer, Thomas Nowak, Ulrich Schmid, and Martin Zeiner.

The core results of this thesis have been published in several papers that appeared in proceedings of international conferences and journals: [CKS13, CPKS14, NFK13, FKN+13].

The two parts considered in this thesis are thus "linked" by the probabilistic assumption on the link failures. In this thesis, we take the simplistic approach of just assuming that the dropping of messages by the message scheduler at the transmitter can be modeled this way at the receiver. An interesting goal of future work is to extend the game-theoretic competitive analysis framework appropriately to automatically provide the matching probability assumptions.

1.3 Road Map of this Thesis:

Two different ways to model distributed computations are introduced in Chapter 2. First, "classic" distributed computing models widely used in literature are explained together with their pros and cons. The notion of *synchrony* is motivated and introduced,

and by examples it is shown how to simulate strong synchrony guarantees, i.e., a round structure, on top of systems providing only weaker synchrony assumptions. Second, the *Real-Time Distributed Computing Model* is introduced, which models distributed systems more accurately by removing some of the strong restrictions of the "classic" models, e.g., zero-time computing steps. We then identify shortcomings of the classic models compared to the real-time model by means of an instance of the one-shot clock synchronization problem.

By giving a brief introduction to real-time scheduling, the prerequisites for the following chapters are established. Finally, we give a high-level overview about the main contribution of this thesis: A new model allowing the analysis of the real-time performance of a distributed system running multiple independent distributed algorithms simultaneously, and suitable analysis methods and tools.

Chapter 3 deals with the message scheduling aspects of running multiple distributed algorithms simultaneously on the same distributed system, which is the underlying part of the new model. Messages are modeled as real-time jobs and have to be scheduled on the finite capacity channels between the processes of the distributed system. Modeling schedulers as labeled transition systems enables a reduction of the competitive analysis problem to decision problems on multi-objective graphs, which provides quantitative measures about their performance. Moreover, a reduction to partial-information graph games allows to synthesize an optimal on-line algorithm, albeit this is computationally hard.

Chapter 4 presents a synchronizer approach that takes action mainly on the reception of messages, which compensates the messages dropped by the scheduler and provides a virtual round structure to the atop-running distributed algorithms. Most of this chapter is devoted to the performance analysis of this synchronizer approach. Using Markov chain modeling and probability theory gives specific and general results about the real-time performance of the resulting synchronous system abstraction.

Finally, Chapter 5 summarizes the contributions of this thesis and explains how they can be applied. It concludes by giving a glimpse of future work.

Chapter 2

Modeling Distributed Systems

When it comes to the modeling of computing systems, abstractions are inevitable. While Maxwell's Equations [Max65] accurately describe the physical effects within electrical circuits, nobody will use those for describing a complete 8-bit adder. Usually, there are already a multitude of abstraction levels between the digital design of the circuitry and the physical structure of the system. Figure 2.1 shows the most common abstractions in computer engineering as described in [AL05]. Even more abstraction is used beyond the computer engineering level: Pseudo code is used to specify algorithms independent from specific programming languages, directed graphs provide abstractions from physical communication channels [CCJ90], etc.

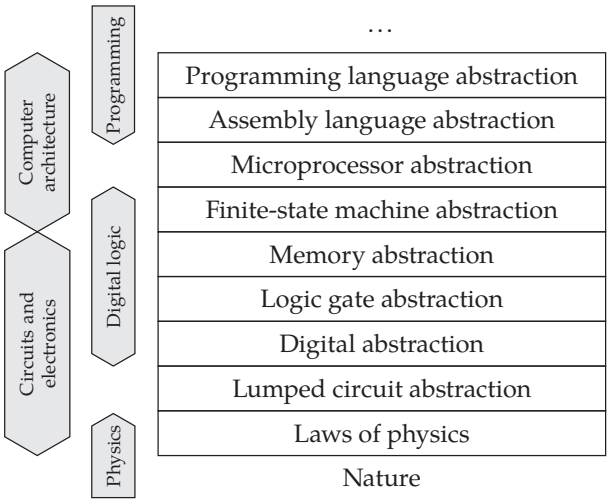


Figure 2.1: Abstraction in Computer Engineering.

In general, multiple abstraction levels allow to capture the essential properties of very complex systems without the need to care about all the underlying entities. In case of a distributed algorithm, a flat model would have to incorporate the algorithm itself, the operating system running the algorithm, the computer architecture of the processor executing the algorithm, the network controller used for communication including the waiting queues in the receiver and transmitter, and the communication network of the system itself. Clearly, without abstraction, (nowadays) it is not possible to pack all those involved parameters into a mathematical model and, even less, to reason about it.

Distributed computing models allow us to abstract from all low-level details and thus to focus on the analysis of and the reasoning about distributed algorithms. This chapter gives an introduction, kept informal for the most part, into the "classical" approaches for modeling distributed systems and their computations. It illustrates how small changes in the model assumption impact worst case bounds and even the solvability of problems. In addition, the chapter motivates the need for a more detailed analysis, including the process and/or message scheduling, by means of the example of clock synchronization in real-time distributed computing systems, and introduces the basics of real-time scheduling along with its challenges. It thereby lays the foundations for the rest of this thesis.

2.1 Classic Distributed Computing Models

In the context of this thesis, a *distributed system* consists of $\mathcal{N}$ spatially distributed processing nodes coupled by a message passing network. A *distributed algorithm* is a collection of (not necessarily identical) processes $1, 2, \ldots, \mathcal{N}$, each of them executed concurrently by a dedicated processing node. *Processes* are modeled as independent state machines, performing state transitions at computing steps. A *state transition* maps the process' current state together with some input to a successor state and some output. While a *state* represents a certain valuation of local variables, the *output* are messages put into the *out-buffer* of a process, denoted as sending, while the *input* consists of messages received from the *in-buffer* of a process. In a *deterministic* distributed algorithm, for every current state and input there exists exactly one state transition that can be performed, while in *randomized* distributed algorithms there may be more than one state transition possible. Which of the state transitions is performed at a computing step is chosen probabilistically, e.g., by flipping a coin. This thesis focuses on deterministic distributed algorithms. A message is *delivered* via the message passing system by removing it from the senders out-buffer and placing it into the receivers in-buffer. Note that this action is usually externally triggered, controlled by an adversary, and not within the scope of a process.

A *communication graph* specifies which processes can directly exchange messages. Therefore, processes are modeled as vertices and a directed edge between two vertices indicates a possible communication flow. The *diameter* of the communication graph is the longest shortest path between any two vertices of the graph, whereas the length of a path is given by the number of the edges it contains. Communication is usually

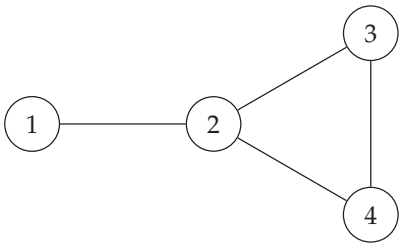


Figure 2.2: Example for the communication graph of a distributed system.

assumed to be reliable, except when the sender or receiver process are faulty (see Section 2.1.3). For now, we will assume that all processes are correct. If communication is symmetric, the directed edges are usually substituted by undirected ones. A fully connected message passing system corresponds to a complete communication graph, i.e, its diameter is 1.

An example of the communication graph of a distributed system with four processes is shown in Figure 2.2. While process 2 is able to send messages directly to all other processes, process 1 can send messages to processes 3 and 4 only by using other processes as relays. The diameter of the graph shown in the example is 2.

The *transmission delay* denotes the time between the sending of a message and the delivery of a message, where delivery denotes placing the message in the receiver's in-buffer and therefore enabling it as an input for the next computing step. Furthermore, the *end-to-end delay* of a message denotes the time between sending the message and actually processing it at the receiver. We will see later in this chapter that there is a fundamental difference between the transmission delay and the end-to-end delay when it comes to the timing analysis of distributed algorithms.

In classic distributed computing models, the state transition of a computing step is assumed to happen instantly and take no time to be completed. Differences arise in the occurrence times of state transitions. In discrete time models, as in [DLS88], state transition happen only at certain points in time, whereas they can occur at arbitrary times in continuous time models, as in [AW04]. Bounds on the number of state transitions some process can perform while another process performs two successive state transitions, as well as bounds on the transmission delay of the messages, determine the *synchrony* of a distributed system.

2.1.1 Synchrony

In this section, the most common synchrony models used in distributed computing, namely synchrony, asynchrony, and partial synchrony, are reviewed. Algorithms solving consensus, one of the most important problems in distributed computing, are used to point out the differences between the models, and to show canonical ways to prove algorithms correct. There exist different versions of the consensus problem in literature, differing for example in the set of the possible input values or whether the processes may decide only on values present in the set of the actual inputs (and in "who must

decide" in the presence of faults). In this section, we adhere to a variant based on the definition of the consensus problem given in [AW04].[1]

The Consensus Problem

The state of every process i in the distributed system has special components x_i, the input of the consensus problem, and y_i, the output of the consensus problem. Initially, x_i holds a value from some well ordered set of possible inputs, e.g., $\mathbb{N}$, $\{0,1\}$, ..., and y_i is undefined. Any assignment of a value to y_i is irreversible and is called *decision*. A solution to the consensus problem has to guarantee the following three properties:

Termination: Eventually, every process i assigns y_i a value.

Agreement: If y_i and y_j are assigned, then $y_i = y_j$, for all processes i and j. Informally, all processes decide on the same value.

Validity: If, for some value v, $x_i = v$ for all processes i, and if y_j is assigned by some process j, then $y_j = v$. Informally, if all processes have the same input, any value decided upon must be that input.

Lock-step synchronous systems

In lock-step synchronous systems, processes perform their state transitions in lockstep. This can for example be achieved by attaching perfectly synchronized hardware clocks to the processes, triggering the state transitions simultaneously on all processes. Alternatively, as discussed in Section 2.2, lock-step rounds can also be built (simulated) atop of approximately synchronized clocks. If the time between computation steps is longer than the upper bound on the end-to-end delay, it is guaranteed that all messages sent by the computing steps can be processed in the next computing steps. Figure 2.3 shows part of a synchronous execution as it could occur in the distributed system shown in Figure 2.2, in a *time/space diagram*. Horizontal arrows depict the advance of time simultaneously at all vertically aligned processes. The aligned marks on the time lines show the synchronized clock ticks at which the state transitions of the processes happen, while the dashed arrows visualize the transmission delay of messages. The tail of the arrows show the process from which and the instant of time when a message is sent. The head of the arrow points to the receiver of the message and to the time instance of the delivery of the message to the receiver's in-buffer.

Due to the alignment of the state transitions and the timing constraints on the transmission delays of the messages, synchronous executions can be abstracted by a round

[1] The definition of the consensus problem in [AW04, Section 5.1.2] is taking faults into account. However, in this part we consider only fault-free executions and therefore the set of non-faulty processes from the original definition is naturally extended to the set of all processes. Also the exact notion of an execution is dismissed, allowing an intuitive description in this informal part of the thesis.

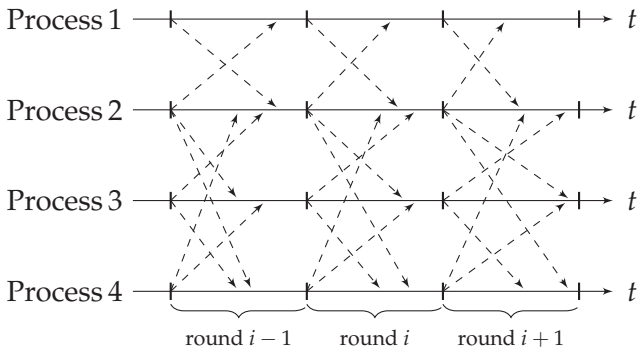


Figure 2.3: Example of a part of a synchronous execution as it could occur in the distributed system shown in Figure 2.2.

Algorithm 1: Solving consensus in synchronous systems of diameter D

Code for processes $i, 1 \leqslant i \leqslant \mathcal{N}$:
Variables: $x_i \in \mathbb{N}$, $y_i :=$ undef., $value_i := x_i$, $in\text{-}buffer_i$
1 **Initial State**: $in\text{-}buffer_i = \varnothing$
2 **for** round r, $1 \leqslant r \leqslant D+1$ **do**
3 receive V, the set of values sent by the neighbors from $in\text{-}buffer_i$
4 $value_i \leftarrow \min(V \cup \{value_i\})$
5 send $value_i$ to neighbors
6 **if** $r = D+1$ **then**
7 $y_i \leftarrow value_i$

structure, allowing distributed algorithms to be specified round-wise. Algorithm 1 states an example for a consensus algorithm for synchronous message passing systems on arbitrary undirected but connected communication graphs.[2] The algorithm terminates after $D+1$ rounds, where D denotes the diameter of the communication graph that has to be provided to the algorithm. Note that the algorithm is anonymous, that is, the index i of a process is only used to name the variables but not directly in the algorithm. To show canonical ways to prove algorithms correct, the proof of the algorithm is sketched in the following.

Theorem 1. *Algorithm 1 solves the consensus problem in synchronous systems with arbitrary but finite and connected undirected communication graphs with diameter D.*

Proof. In the case of a consensus algorithm, three things have to be proven:

Termination: The termination of the algorithm in round $D+1$ follows directly from lines 6 and 7 of the algorithm.

[2] This algorithm only works in a fault-free scenario.

Agreement: Without loss of generality, assume process j is among the processes with the minimal initial value. Per induction it follows that, at the end of round k, all processes in the $k-1$-hop neighborhood[3] of process j have set their variable $value_i$ to this minimum (see lines 3 and 4 of the algorithm). At the end of round $D+1$ this neighborhood spans the whole system, as the communication graph has to be connected.

Validity: Follows directly from the proofs of termination and agreement. ■

Asynchronous systems

If the time between two processing steps and the transmission delay of messages is finite but can not be bounded, the system is called *asynchronous*. This is the weakest of the classic execution models for distributed algorithms and, as a result, an algorithm designed for an asynchronous system can be employed in any system, including a synchronous one. In an asynchronous system, it is assumed that the processes do not have access to local or global clocks, hence the only feasible mechanism for synchronization in such a system is communication between processes. In the following, we restrict ourselves to message-driven algorithms, i.e., computing steps are triggered only by reception of messages. Figure 2.4 shows a time/space diagram of an execution in an asynchronous system.

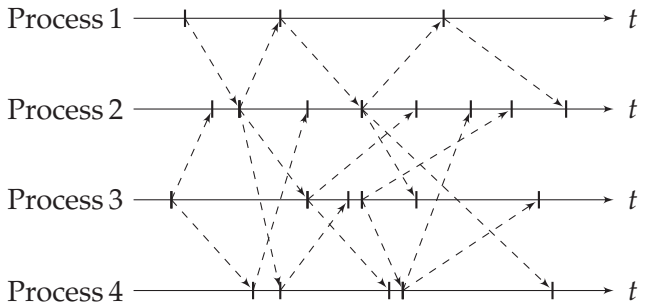


Figure 2.4: Example of an asynchronous execution.

Algorithm 2 is a consensus algorithm for an asynchronous distributed message passing system.[4] Line 2 of the algorithm points out the message-driven nature of the algorithm. Compared to the algorithm solving consensus in the synchronous model, the proposed asynchronous algorithm needs more information about the distributed system. In addition to the exact number of processes involved in the distributed systems it also needs every process to have a unique identifier used as an index for the processes' values. Again, we give a sketch of the proof for the asynchronous algorithm:

[3]The k-hop neighborhood defined by induction: The 0-hop neighborhood of a process is the process itself. The k-hop neighborhood of a process is the $k-1$-hop neighborhood and all processes connected by an edge to it.

[4]Like the synchronous algorithm, this one also requires a fault-free system as [FLP85] shows the impossibility of consensus in asynchronous systems with faults.

Algorithm 2: Asynchronous algorithm solving consensus (in the fault-free case)

Code for processes $i, 1 \leqslant i \leqslant \mathcal{N}$:
Variables: $x_i \in \mathbb{N}$; $y_i :=$ undef.; $terminated_i :=$ false; $Values_i := \{\}$, $in\text{-}buffer_i$

1 **Initial State**: $\langle i, x_i \rangle$ *is initially in* $in\text{-}buffer_i$
2 **on** receiving W **:**
3 $\quad Values_i \leftarrow Values_i \cup W$
4 $\quad$ send $Values_i$ to all neighbors
5 $\quad$ **if** $|Values_i| = \mathcal{N}$ *and* $terminated_i =$ *false* **then**
6 $\quad\quad y_i \leftarrow \min_j v_j$ from $\langle j, v_j \rangle \in Values_i$
7 $\quad\quad terminated_i \leftarrow$ true

Theorem 2. *Algorithm 2 solves the consensus problem in asynchronous systems with $\mathcal{N}$ processes, unique IDs, and arbitrary but connected undirected communication graphs.*

Proof. The same three properties have to be proven:

Termination: The key to the proof of the algorithm is the set $Values_i$, which eventually holds the initial values of all processes. Assume by contradiction that process i does not decide. This can only happen, if the size of $Values_i$ never becomes $\mathcal{N}$. Without loss of generality, assume the initial value of process j is among the missing ones. By induction on the hop-distance, we can show that process i will eventually receive the initial value of process j. At its first step, process j will receive its own initial value as it is initially placed in the in-buffer (cf. line 1) and send it to all neighbors (at hop-distance 1).
Eventually, every process at hop-distance k will receive the initial value of process j and send it to all its neighbors.
By the system assumption, every process will eventually receive its own value (placed in the in-buffer initially) and thus every process will send its initial value to its neighbors. As the communication graph is connected, eventually process i will receive sets that, if merged, contain the initial values of all $\mathcal{N}$ processes. A contradiction to the assumption that process i will not decide.

Agreement: Every terminating process has the same set $Values_i$ and therefore decides on the same value right before terminating.

Validity: Follows directly from the proofs of termination and agreement. ■

Partially synchronous systems

Partial synchrony, introduced in [DLS88], is located between the previously discussed cases of lock-step synchronous and asynchronous systems and tries to provide a more realistic execution model. In their partial synchronous system model [DLS88], Dwork et al. assume a globally consistent notion of a discrete time, i.e., there exists a (virtual) global real-time clock, and on every tick some of the processes perform one step of their

state machine. Note that their notion of a step and message delivery differs slightly from the models introduced earlier. Every step of a process can be either a *send* step, placing a message in the receiver's in-buffer, or a *receive* step, removing some (possibly empty) set of messages of the process's in-buffer and processing (delivering) them. For every tick of the real-time clock, an adversary decides (1) which processes make a step and (2) if the step is a receive step, the set of messages that are delivered. There exist two restrictions on the adversary determining the level of synchrony in this model: *Communication synchrony*, determined by the parameter Δ, denoting a fixed upper bound on the number of real-time clock ticks until a message is delivered; and *Processor synchrony*, determined by the parameter Φ, denoting an upper bound on the relative processor speed (during Φ consecutive real-time clock ticks, every process has to perform at least one step).

Dwork et al. investigate multiple variants of partial synchrony: For example, Δ and/or Φ exist, but are not known[5] and thus cannot be exploited by the algorithm, e.g., as a timeout. Alternatively, Δ and/or Φ are known but do not hold right from the beginning but only after some unknown time, called global stabilization time.

In [DLS88], Dwork et al. provide a generic way to implement synchronized clocks and hence to simulate lock-step rounds on the processes in all those model variants. Given simulated lock-step rounds, Algorithm 1 could be used to solve consensus in the absence of failures; actually, [DLS88] also considers Byzantine failures.

2.1.2 Execution and Communication Primitives

Natural performance measures of distributed algorithms are the (worst case) *message complexity*, i.e., the number of messages that need to be sent in the worst case to solve a problem, and the (worst case) *time complexity*: In synchronous systems, the time complexity can be determined by counting the number of rounds until an algorithm has solved a certain problem. In asynchronous systems, the time complexity is determined by normalizing the largest end-to-end delay of a message to some unit-time, whereas the transmission times of the other messages sent are scaled appropriately. The time complexity of the asynchronous algorithm is then expressed as the termination time for the worst case scenario.

Which problems are solvable in a distributed system and how "good" algorithms can solve problems depends on the assumptions on the allowed executions. System features like unique identifiers, the ability to send different messages to different neighbors, or communication graphs a priori known to the processes can improve the (worst case) complexity of algorithms solving certain problems in a distributed system, or might even be mandatory for solving certain problems. In the following, we will elaborate on some of those features.

[5] In this manifestation of partial synchrony, in every run of the algorithm it holds that there exists a upper bound, while in synchronous systems there exists an upper bound holding in every run of the algorithm.

Unique identifiers

If the communication system provides unique identifiers (IDs), every process can be identified explicitly. For example, the previously illustrated Algorithm 2 uses such unique IDs in the tuples sent to the neighbors to distinguish between the initial values of the processes and therefore enables termination. Without unique IDs, the stated algorithm would not work. Furthermore, unique IDs are often used for (deterministic) symmetry breaking, for example, in the *leader election problem*.

The Leader Election Problem

In the leader election problem, exactly one process has to elect itself as leader, e.g., by irreversibly assigning the value *leader* to a state variable. In some definitions of the problem, it is also required for all other processes to know the id of the leader.

The well-known impossibility result for deterministic anonymous leader election in distributed systems with a ring as communication graph, e.g., presented by [AW04, Theorem 3.2] and [Lyn96, Theorem 3.1], was first proven by Angluin [Ang80].

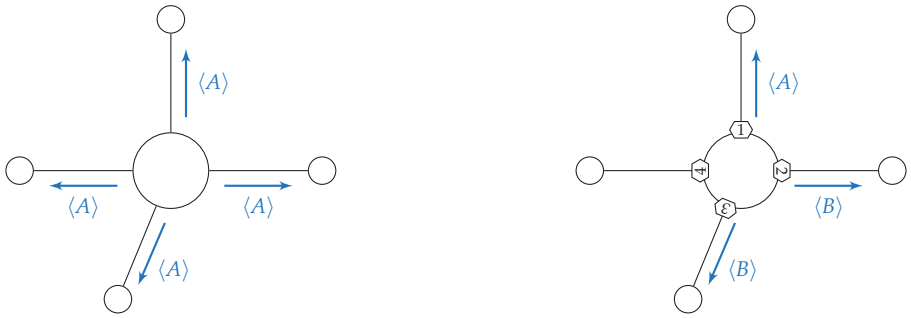
On the other hand, algorithms for *anonymous* systems, i.e., systems lacking unique identifiers, do have interesting applications. The fact that the same algorithm is running on all the processes makes it easier to replace defective processes, as no care has to be taken for altered IDs. Additionally, large sensor networks, consisting of tiny agents, like the ones proposed in [AAD+06], do not even support identifiers. Finally, today more than ever, privacy might need to be protected by forbidding any form of identity [BFK00].

While an algorithm designed to run anonymously can of course be executed in a system with IDs by just ignoring them, the previously mentioned impossibility of leader election in rings without IDs shows that anonymous systems are strictly weaker than systems providing unique IDs.

Sending messages

In distributed systems where the processes are connected by a network with dedicated (or at least virtual) point-to-point links, a process can typically address all its neighbors individually (see Figure 2.5b, where the small numbers represent port numbers used for addressing). This means that a process can send different messages to different neighbors. In case of wireless sensor networks, messages are usually sent in a broadcast fashion to all neighbors at once. This is illustrated in Figure 2.5a.

Again, the provided send primitive of the communication network directly influences the solvability of problems. [Sak99] and [YK96] have shown that in an anonymous distributed system with unknown communication graphs, the existence of a unique leader combined with addressable neighbors is enough to establish unique identifiers



(a) Broadcast paradigm for sending messages. The same message is sent to all neighbors.

(b) Individual messages can be sent do different neighbors. Port numbers can be used for addressing.

Figure 2.5: Two different paradigms for sending messages.

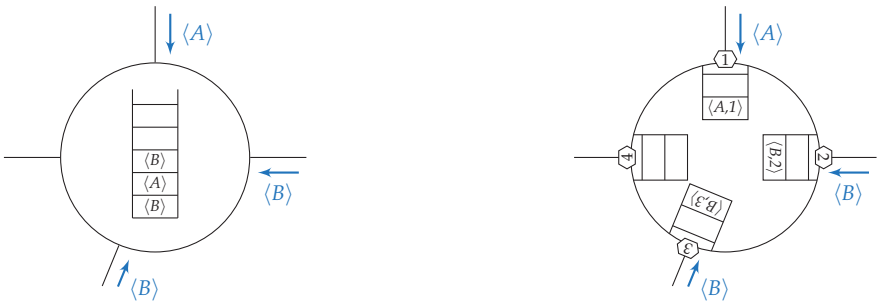
for all processes. Note carefully that port numbers only support *local* neighbor addresses: The same process i may be connected to different ports at process j and process l. In sharp contrast, unique IDs support *global* addressing. However, the authors of [YK99] have shown that, if a broadcast primitive is used for sending messages, a unique leader is not sufficient to generate unique identifiers in the same setting. Furthermore, Dolev et al. have shown in [DDS87] that there is also a difference in the solvability of consensus in systems with various different synchrony assumptions for communication and computation depending on whether processes are able to send a message to at most one single neighbor per computing step, or if they have access to a broadcast primitive and are thus able to send a message to all their neighbors in a single computing step.

Receiving messages

At the beginning of the current section, message delivery was described as "placing the message in the receiver's in-buffer". As shown in Figure 2.6, this in-buffer can either be shared among all neighbors or dedicated, providing explicit access to the messages sent by a certain neighbor. While the second variant (shown in Figure 2.6b) is restricted to distributed systems with dedicated point-to-point links, e.g., in [AW04], the first version (Figure 2.6a) is also used in [DKMP95] to model wireless sensor networks. In their work, [DKMP95] have shown that both paradigms of receiving messages are computationally equivalent in terms of the class of Boolean functions[6] they can compute, if the communication network is a ring.

The problem of leader election in the case of not necessarily unique process IDs with different sending and receiving paradigms was studied in [YK99]. The authors have shown that, by using broadcast as the sending paradigm, the sets of communication graphs in which the problem is solvable with both receiving paradigms are the same. If

[6]Boolean functions describe the class of functions of variables over an arbitrary Boolean algebra which can be constructed from constants and the variables by superpositions of the basic algebra operations (cf. [Rud74]).



(a) Combined in-buffer for all neighbors. Relating a message to a specific neighbor is not possible without unique IDs.

(b) Every neighbor has a distinct in-buffer. A received message can be mapped to a neighbor by the port number.

Figure 2.6: Two different paradigms for receiving messages.

individual messages can be sent to the neighbors, the set of communication graphs in which the problem is solvable with a combined in-buffer is a real subset of the distinct in-buffers case. In [DDS87], Dolev et al. have investigated the difference in the solvability of consensus in systems with different synchrony assumptions for communication and computation. They have shown that allowing atomic receive/send steps, i.e., receiving messages and sending messages can be done in the same computing step, or forcing separate receive and send steps, i.e., in one processing step messages can either be received or sent, but not both only makes a difference for asynchronous computation, i.e., there are no bounds on the relative processor speeds, combined with synchronous communication, i.e, there exists an upper bound on the end-to-end delay of the messages.

Information about the system

The more information an algorithm has available about the distributed system it is executed in, the more efficient it can solve a problem. Take as an example algorithms solving the consensus problem: If every process has a unique ID and is aware of the communication graph, the consensus problem can be solved very efficiently with respect to the message complexity: Every process can calculate an identical spanning tree, without sending messages, just by using the largest ID as root and applying a deterministic spanning tree algorithm on the communication graph. By sending at most two messages over the edges contained in the spanning tree and no messages over the other edges, the algorithm by Fusco and Pelc [FP11] solves consensus by sending at most $\mathcal{O}(\mathcal{N})$ messages. Note that the system model used in [FP11] differs slightly from the previously introduced one, as initially all processes are inactive. The adversary wakes up a subset of the processes and assigns them the input values over which consensus has to be established. Furthermore, all processes have to decide (and terminate) in the same round.

Removing unique IDs increases the lower bound of the message complexity to $\Omega(\mathcal{N} \log \mathcal{N})$ and an algorithm sending at most $\mathcal{O}(\mathcal{N}^{3/2} log^2 \mathcal{N})$ messages is introduced in [FP11]. In the case where only the number of processes is known to the algorithm, the authors prove a tight lower bound of $\Omega(\mathcal{N}^2)$ messages that need to be sent for solving the consensus problem. Interestingly, even the presence of unique IDs is not enough to improve the lower bound if the communication graph is unknown for the algorithm.

For the case of executing a leader election algorithm in a distributed system whose communication graph is a ring, Hirschberg and Sinclair [HS80] give a *uniform*[7] *comparison-based*[8] algorithm sending at most $\mathcal{O}(\mathcal{N} \log \mathcal{N})$ messages, which is later shown in [Bur80] to be asymptotically optimal for the asynchronous case. For the synchronous case, Frederickson and Lynch [FL87] show that processes cannot exploit the synchrony to further reduce the number of messages sent by proving the $\Omega(\mathcal{N} \log \mathcal{N})$ lower bound for comparison-based algorithms together with an asymptotically optimal algorithm. By using the process IDs directly in the algorithm, i.e., removing the restriction of being comparison-based, Frederickson and Lynch [FL87] give an algorithm that has a message complexity in $\mathcal{O}(\mathcal{N})$. However, in their algorithm the number of rounds until termination grows exponentially with the value of the smallest identifier.

If nothing about the system is known, neither consensus nor leader election can be solved, as both depend on the distributed system as a whole, and [Suo13, NS95] showed that such problems cannot be solved locally.

2.1.3 Modeling Faults

If processes of a distributed system do not adhere to their algorithm, they are called *faulty*. This deviation can manifest itself in many ways. The most important ones, e.g., used in [PT86, DLS88, AW04], are explained in the following.

Crash faults: If a process crashes, it stops prematurely performing state transitions. In particular, once crashed, a process does not send any messages. The most restrictive version of crash faults allows only *clean crashes*, where a computing step is either executed completely (sending all messages) or not executed at all (sending no messages at all). A more general version of crash faults enables processes to crash *unclean* during the execution of a computing step, thereby sending a subset of the messages to their neighbors only. This can lead to inconsistent states among the non-faulty receiver processes. However, a message cannot be corrupted by an unclean crash, i.e., all sent messages are in accordance with the algorithm.

Omission faults: If a receive omission happens at a faulty process, some of the messages it receives are dropped and therefore not placed in the in-buffer. This can

[7] A *uniform* algorithm does work unchanged for arbitrarily large networks. Therefore, it does not use the number of processes in the code. See [AW04, Capter 3.4.1].

[8] In a *comparison-based* algorithm, the generated message pattern is not determined by the absolute values of the unique IDs but rather by the order pattern of the identifiers. Informally, a comparison-based algorithm running on order-equivalent rings will "behave the same", with order-equivalent and "behave the same" defined as in [AW04, Capter 3.4.2].

be motivated by a buffer overflow of the receiver's in-buffer, or by a message that got corrupted during transmission and discarded at the receiver due to a failed checksum test. Analogously, a send omission at a faulty process manifests itself by sending only a subset of the messages specified by the state transition of the process, possibly caused by a buffer overflow in the send buffer or by an exhausted capacity constraint on a communication channel. Processes suffering general omission faults may experience a combination of send and receive omissions.

Byzantine faults: A Byzantine faulty process can behave arbitrarily, i.e, it is not restricted at all. In particular, it can send arbitrary messages to its neighbors, alter messages that should be relayed or just drop them. Byzantine faults are the most general ones, and an algorithm designed to cope with a certain number of Byzantine faulty processes will also work if the system only suffers omission or crash faults.

All fault models introduced by now tie faults to certain processes and are static in nature. That is, while the algorithm does not know in advance which of the processes are faulty, it can rely on the fact that the set of faulty processes does not change during the execution.

Santoro and Widmayer [SW89] proposed a synchronous system model, where all processes are correct, but communication between the processes may fail. Additionally, the transmission faults in their model can be dynamic, i.e., the pairs of processes which's communication fail can change in each round. The introduced fault model is more general than the previously mentioned fault models as, for example, the perception of a crash faulty process by its neighbors can be simulated by static faults on all links from (and to) the process whereas the valid situation of one faulty link per process results in a system configuration where all processes have to be declared faulty rendering the system useless for any analysis. Hybrid fault models like [BSW11] combine different types of process and communication faults. There are several attempts to model communication faults in a more general way, e.g., in [CDP96, Sch01, BWCB+07, SW07, CBS09, SWK09].

Typically, it is assumed that an adversary controls when and where faults occur and in which way they manifest themselves. In the case of process faults, for example, the adversary controls which and when processes crash, or, in the case of Byzantine faulty processes, which messages they send. The power of the adversary can be restricted, for example by a limit on the number of faulty processes or links in a system. For example, synchronous consensus is impossible to solve if more than $\frac{N-1}{3}$ processes are Byzantine faulty [LSP82].

Probabilistic faults

A completely different approach is to use a probabilistic fault model, where manifestation of faults is not controlled by an adversary but rather by a random process. For example, in the analysis of wireless networks [KLB04, BV07, PP07] it is often assumed that a message sent via a link is dropped with a certain probability.

Probabilistic fault models have the advantage that they extend solvability of problems beyond the limits of adversarial fault models. However, bounds on measures like time complexity or solvability can be stated at most "almost surely", i.e., with probability 1.

Two problems that are well studied with respect to probabilistic process faults are (1) the broadcast problem, where a message initially only available at one process (called source) has to be delivered to every process in the system, and (2) the gossip problem, where every process has to receive the initial messages of all processes in the system. Those problems are usually studied in discrete time system models where every process is able to send a message to at most one neighbor per time unit, e.g., in the case of the randomized broadcast algorithm *push*, introduced in [DGH$^+$87]. In the push algorithm, at every unit time, every process that has already received the initial message chooses randomly one of its neighbors to forward a copy of the message.

As one example, Elsässer and Sauerwald [ES09] investigated the push algorithm under the presence of random send omission failures where a process may fail in some step with probability $1 - p$ to send the message. They proved that introducing probabilistic failures increases the broadcasting time by at most a factor $6/p$.

In wireless and ad-hoc sensor networks, locally computable variants of distributed algorithms, such as the ones described in [SP04], are preferred as they can be executed in large networks with arbitrary communication graph structures albeit a single process has only local knowledge of the structure. For example, when considering *average consensus* it is sufficient for the processes to reach asymptotic agreement, i.e., in every computing step, a process updates its value with the (weighted) average of its own value and the values received by its neighbors. The average consensus problem, combined with a stochastic link failure model, leads to interesting questions such as the convergence speed of the processes' values [HM05, KM08] or the search for optimized communication graph structures [KM09].

The analysis of distributed algorithms with probabilistic fault models is usually done by stochastic approximation arguments and Markovian analysis [ES09, HM05, KM08, KM09]. This thesis will use a probabilistic link fault model as an abstraction for messages that were discarded by the message scheduler due to an overload scenario on the communication links.

2.2 Achieving a Round Structure

There are many reasons why it is beneficial to achieve synchrony in a distributed system. Due to its round structure, the possible behavior of a synchronous system is more restricted than the possible behavior of an asynchronous system, which allows a multitude of different interleavings of computing steps. This makes designing and understanding algorithms for synchronous systems easier than algorithms for asynchronous systems.

Another benefit of synchrony can be observed by taking a look on the termination time of Algorithm 1 solving consensus in a synchronous distributed system. Every instance of the algorithm will terminate after exactly $D + 1$ rounds. If the algorithm

Algorithm 3: Pseudo code of the Synchronizer α introduced by Awerbuch

Code for processes $i, 1 \leqslant i \leqslant \mathcal{N}$:
Variables: $Rnd_i := 1$; $\text{RcvAck}_i[\cdot][\cdot] :=$ false; $\text{RcvSafe}_i[\cdot][\cdot] :=$ false;
1 **Initial State**: $\mathsf{Pulse}\langle 1\rangle$ *is initially in transit to all neighbors*
2 **When** *receiving* $\mathsf{Pulse}\langle \mathrm{r}\rangle$ *from neighbor* j **do**
3 Send $\mathsf{Ack}\langle r\rangle$ to neighbor j
4 **When** *receiving* $\mathsf{Ack}\langle \mathrm{r}\rangle$ *from neighbor* j **do**
5 $\text{RcvAck}_i[r][j] \leftarrow$ true
6 **if** $\text{RcvAck}_i[Rnd_i][\cdot] =$ *true for all neighbors* **then**
7 Send $\mathsf{Safe}\langle Rnd_i\rangle$ to all neighbors
8 **When** *receiving* $\mathsf{Safe}\langle \mathrm{r}\rangle$ *from neighbor* j **do**
9 $\text{RcvSafe}_i[r][j] \leftarrow$ true
10 **if** $\text{RcvSafe}_i[Rnd_i][\cdot] =$ *true for all neighbors* **then**
11 $Rnd_i \leftarrow Rnd_i + 1$
12 Send $\mathsf{Pulse}\langle Rnd_i\rangle$ to all neighbors

is used repeatedly in a replicated state machine for obtaining consensus on the states of the replicas, the next instance of the algorithm and thus the next state transition of the replicated state machine can be executed back to back after the previous one. In the asynchronous Algorithm 2, a simultaneous termination of the algorithm cannot be guaranteed, which implies the need for buffering transactions.

While a large number of solutions to problems in distributed computing assume lock-step rounds, for example in [Lub86, ST87, Lyn96, Dob03, AW04, CBS09], and thus can be executed directly if such a round structure is present, real-world distributed systems are usually not perfectly synchronous. There are several ways to generate round structures in various different system and failure models, e.g., [Awe85, DLS88, BHPW07, BH09, WS09]. In the following, three examples of how a round structure can be established are given. While in the first two examples the systems use previously introduced synchrony assumptions, in the final example processes have access to approximately synchronized hardware clocks.

Asynchronous systems: In [Awe85], Awerbuch introduces the principle of using a so-called synchronizer to simulate consistent rounds, enabling the execution of any synchronous algorithm, in any fault-free asynchronous system. To this end, processes use explicit handshake messages to inform the sender about the delivery of sent messages. If a process is aware that all its messages sent in the current round have been delivered successfully, and if it has acknowledged the reception of messages affiliated with the current round for all its neighbors, it proceeds to the next round.

Awerbuch presents two different synchronizers, optimized for either time complexity (named *Synchronizer* α) or message complexity (named *Synchronizer* β), and a third combined version (*Synchronizer* γ) optimized for both. The pseudo code of the

α-synchronizer is given in Algorithm 3. The algorithm uses three types of messages: Pulse messages indicate the start of a new round; Ack messages are sent as acknowledgment of Pulse messages; and Safe messages indicate that a process has received Ack messages from all its neighbors. If a process has received Safe messages from all its neighbors it can proceed to the next round.

Awerbuch's α-synchronizer serves as the basis for the retransmission-based synchronizer used in Chapter 4 of this thesis.

Partial synchrony: As described in Section 2.1.1, the partial synchronous system model introduced in [DLS88] makes use of two parameters, namely Δ and Φ for describing bounds on the message delay and the relative speed of processes. That bounds might be either unknown or they might not hold right from the beginning. In [DLS88], Dwork et al. establish a Byzantine fault-tolerant algorithm to implement synchronized distributed clocks, based on the synchronization algorithm published by Lamport in [Lam78] and very similar to the clock synchronization algorithm by Srikanth and Toueg [ST87]. The pseudo code of a variant of this algorithm for fully-connected systems with $\mathcal{N} > 3f$ processes, where f processes may behave arbitrary (Byzantine faulty), is given in Algorithm 4. Initially, every process broadcasts $\mathsf{tick}\langle 1\rangle$. If a process receives $f + 1$ $\mathsf{tick}\langle l\rangle$ messages, at least one of them was sent by a correct process. Thus the process may catch up and send the missing ticks. If a process receives $2f + 1$ $\mathsf{tick}\langle k\rangle$ messages, it starts the next round by broadcasting a $\mathsf{tick}\langle k + 1\rangle$ message. These (approximately) synchronized clocks can be used directly to simulate lock-step rounds, e.g., using the approach below.

> **The Clock Synchronization Problem**
>
> In the (classic) clock synchronization problem, a read-only hardware clock HC_i is attached to every process i. A special state variable can be used to adjust the hardware clock, i.e., apply an offset correction. The goal of the processes in the clock synchronization problem is to calculate this correction value for their hardware clock such that a synchrony condition is satisfied, e.g., minimizing the difference between the value of the adjusted clocks of any pair of processors (called *skew*). Additionally, the adjusted clocks have to satisfy some progress condition, usually requiring them to stay in some linear envelope of real-time (cf. [ST87]).

Using approximately synchronized clocks: If processes have access to a hardware clock (or a comparable mechanism), Srikanth and Toueg provide a solution to generate lock-step rounds in [ST87].

Assume the (adjusted) clocks are synchronized with precision π, i.e., the (real-)time difference when any pair of clocks reach (clock-)time T is at most π. Furthermore, the

Algorithm 4: Fault-tolerant algorithm for generating approximately simultaneous tick messages in a fully-connected network

Code for processes $i, 1 \leqslant i \leqslant \mathcal{N}$:
Variables: $k := 1$;
1 **Initial State**: *send* $\mathsf{tick}\langle 1 \rangle$ *to all [once]*
2 **if** *received* $\mathsf{tick}\langle l \rangle$ *from at least* $f + 1$ *distinct processes with* $l > k$ **then**
3 Send $\mathsf{tick}\langle k+1 \rangle, \ldots, \mathsf{tick}\langle l \rangle$ to all [once]
4 $k \leftarrow l$
5 **if** *received* $\mathsf{tick}\langle k \rangle$ *from at least* $2f + 1$ *distinct processes* **then**
6 Send $\mathsf{tick}\langle k+1 \rangle$ to all [once]
7 $k \leftarrow k + 1$

end-to-end delay of a message is upper-bounded by δ. A lock-step round structure can be generated by starting round k on a process when its clock reaches the value $R \cdot k$, where $R \geqslant \pi + \delta$. Note, that this assumes that the hardware clocks progress with the same rate as real-time. If the hardware clocks suffer clock-drift, i.e., the rate of the clocks are slightly slower or faster than real-time, this scaling factor has to be incorporated to ensure that even the process with the fastest clock does not start the next round while there might be a message in transit. Additionally, drifting clocks have to be resynchronized periodically to maintain the precision bound. Figure 2.7 shows the basic idea of how the rounds are generated.

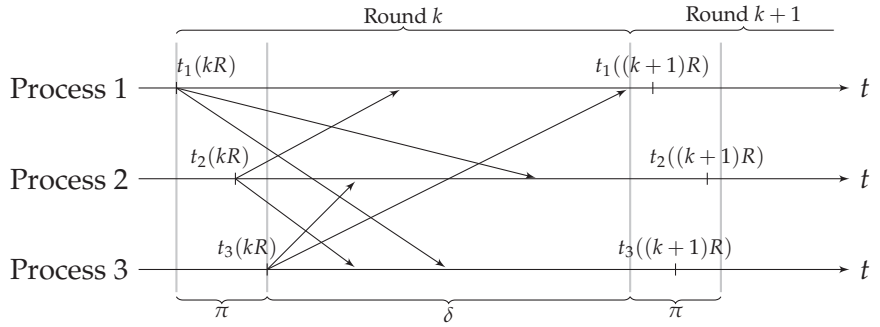
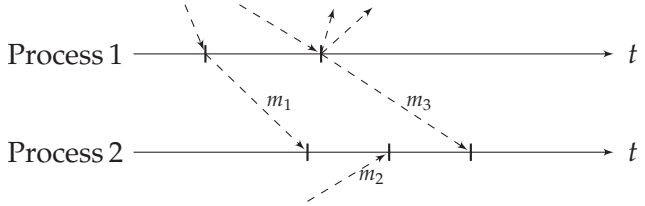


Figure 2.7: Generating rounds from approximately synchronized clocks.

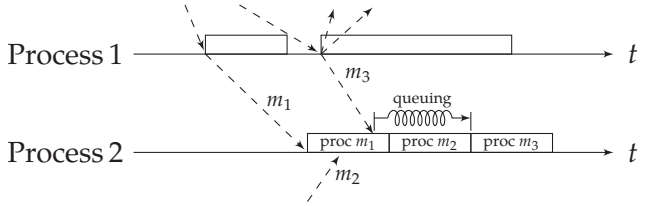
2.3 The Real-Time Distributed Computing Model

The previously introduced computing models are coarse abstractions of distributed systems in the real world. Two simplifications made in all these models are instantaneous state transitions and encapsulation of timing-related parameters, such as network delay, message queuing, scheduling overhead, and sender/receiver processing time in a fixed transmission or end-to-end delay bound of a message.

In [Mos09a, Mos09b, MS14], Moser introduced a new distributed computing model where zero-time computing steps are replaced by non-zero-time atomic jobs. It explicitly incorporates the processing time for the state transitions and the actual transmission delays which allows to compute the actual end-to-end delays, as shown in Figure 2.8b. It is apparent that this refined modeling allows to incorporate both queuing effects and scheduling disciplines in the real-time analysis of the resulting end-to-end delays. For comparison, an equivalent execution in the classic distributed computing model is shown in Figure 2.8a, which highlights the "masking effects" of the end-to-end delay (equal to the transmission delay) in this model.



(a) Execution in classic distributed computing models. The dashed arrows denote transmission delays of messages identical to the end-to-end delays.



(b) Execution in the real-time distributed computing model. The dashed arrows denote the actual transmission delays of messages.

Figure 2.8: Comparison of an execution in classic distributed computing models vs. an execution in the real-time distributed computing model.

The Time Complexity—Precision "Paradox" of Clock-Synchronization

The problem of terminating fault-free clock synchronization is well suited to reveal one of the main issues with classic distributed system models. To this end, a read-only hardware clock is attached to every process. While the clock rates of the individual clocks are exactly the same, the absolute values may differ. Clearly, this is a one-shot problem as, due to the identical clock rates, the clocks stay synchronized after the correction values are computed once and applied.

In [LL84], Lundelius and Lynch prove a lower bound of $\varepsilon_{(c)}(1-1/\mathcal{N})$ on the achievable precision in a distributed system with $\mathcal{N}$ processes and an uncertainty of $\varepsilon_{(c)}$ in the end-to-end delay, i.e., the difference between the upper bound and the lower bound on the end-to-end delay. They also provide an optimal algorithm that achieves this

precision in a fully connected setting and in constant time. However, regarding the time complexity, this algorithm heavily exploits the fact that it is able to process an arbitrary number of messages in zero-time.

Moser and Schmid [MS06] analyzed Lundelius and Lynch's algorithm in the context of the real-time distributed computing model. They observed that using a naive transformation results in a suboptimal precision, as the time-correlated broadcast transmissions of the algorithm introduce additional uncertainty due to the queuing and scheduling decisions at the receiver of the messages. Furthermore, due to the (non-zero) time needed for processing the received messages, the time complexity of the algorithm grows from $\Theta(1)$ in the classic distributed computing model to $\Theta(\mathcal{N})$ in the real-time distributed computing model, which is shown to be optimal as each process needs to receive at least one message from all other processes. Moser and Schmid also present a lower bound of $\varepsilon_{(rt)}(1 - 1/\mathcal{N})$ for the achievable precision for a non-zero step time, where $\varepsilon_{(rt)}$ is the uncertainty of the transmission delay in their real-time model. This does not look too impressive, but while $\varepsilon_{(c)}$ in the classic distributed computing model incorporates the end-to-end delay and hence all timing related parameters of the distributed system, $\varepsilon_{(rt)}$ consists of the uncertainty of the transmission delay in the real-time model only, which does not include any queuing and processing times and is therefore much smaller.

Additionally, Moser and Schmid give a more sophisticated transformation that, by enforcing a message pattern that actively avoids queuing effects on the receiver side, achieves the optimal precision while still maintaining the optimal time complexity of $\Theta(\mathcal{N})$. This is possible because the classic computing model empowers both the algorithm and the adversary. While the algorithm is able to process a large number of messages in no time, the adversary can enforce transmission delays not justified by the actual message patterns. In [MS14], the transformations between classic and real-time distributed system model are extended to also work in the presence of Byzantine faults.

These insights regarding the real-time distributed computing model reveal two facts: First, classic distributed system models mask important real-time characteristics of distributed systems that cannot be neglected. Secondly, scheduling decisions have to be taken into account when analyzing real-time properties of distributed algorithms in a sound way. As mentioned earlier, this is one of the issues this thesis is trying to address, and therefore an introduction into real-time scheduling is essential for the further analysis.

2.4 Basics of Real-Time Scheduling

Scheduling addresses the allocation of a shared resource, e.g., a processor or a communication link, to multiple consumers, e.g., jobs to be processed or messages to be sent. *Real-time scheduling* additionally takes timing constraints into account [SSRB98, But11].

In real-time systems research, a real-time job J_i is defined by at least three parameters: the absolute release time r_i, describing when the job gets ready for execution, its worst

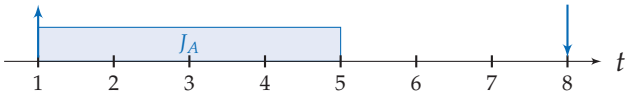


Figure 2.9: Illustration of a real-time job.

case execution time C_i, describing how long the job has to be allocated to the processor before it is completed, and the absolute deadline d_i, describing the latest point in time until the job has to be completed.

In the following we assume a discrete time model with time-slots of unit-time ε, i.e., r_i and d_i are Integer multiples of ε; typically, we will assume $\varepsilon = 1$ for simplicity. Figure 2.9 shows real-time job J_A with $r_A = 1, C_A = 4, d_A = 8$, where the upwards pointing arrow denotes the release of the job, the downwards pointing arrow denotes the deadline of the job, and the colored rectangle shows the allocation of the processor.

If multiple jobs are to be executed on a single processor, the processor has to be assigned to the jobs according to a certain strategy, the *scheduling policy*. The goal is to create such an assignment, a *schedule*, optimized for different application-specific objectives, for example, maximizing the cumulative execution time of the jobs meeting their deadlines, minimizing the number of deadline misses, or minimizing the average response time.[9]

Adding a second job J_B with $r_B = 4, C_B = 2, d_B = 6$ to the previously introduced example reveals some possible scheduling scenarios shown in Figure 2.10. In Figure 2.10a, a preemptive scheduling policy is used, allowing both jobs to complete timely before their deadlines. In preemptive scheduling, at every unit-time, a scheduling decision is made and a currently scheduled job can be preempted in favor of another job, e.g., a newly released job with a more urgent deadline. However, preemptive scheduling is not always possible, for example, when scheduling data packets on a communication link. In such cases, a non-preemptive scheduling policy has to be used, where a job, once its execution has started, can only be aborted but not preempted. Therefore, at unit-time $t = 4$, a scheduling decision has to be made: Either the execution of job J_A is continued, resulting in a violation of job J_B's deadline, shown in Figure 2.10b, or job J_A has to be abandoned in favor of job J_B, shown in Figure 2.10c.

The release times of the jobs are specified by their *arrival pattern*, which can be

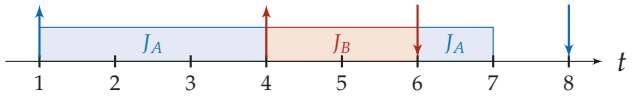
periodic, where jobs are released regularly every k timeslots, where k is called the *period*,

sporadic, where jobs are released irregularly but no two within less than k timeslots, where k is called the *sporadicity interval*, or

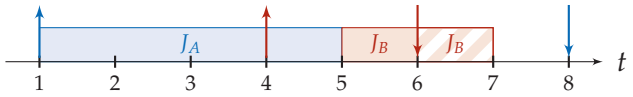
aperiodic, where jobs are released without any timing restrictions.

Usually, jobs are not specified individually, but rather considered as multiple instances of a periodic/sporadic/aperiodic *task* τ_i, specified by at least the worst case

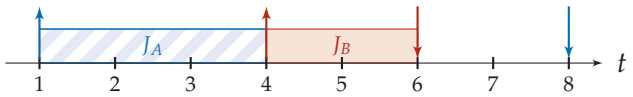
[9]The response time of a job is the time between its release and the finishing time (when the job is executed completely.



(a) Preemptive schedule of jobs J_A and J_B meeting all deadlines.



(b) Deadline violation of job J_B due to non-preemptive scheduling.



(c) Dropping job J_A in favor of job J_B due to non-preemptive scheduling.

Figure 2.10: Different scheduling scenarios for jobs J_A and J_B.

execution time of its corresponding jobs C_i and the relative deadline D_i with respect to the release time of its jobs.

2.4.1 Selection of On-line Scheduling Algorithms

In this section, we will briefly introduce some of the most commonly used single processor scheduling algorithms, ranging from very simple ones like FIFO (First In First Out) to very involved ones like DOVER [KS95].

Notation on Job Sequences

If no scheduler (neither on-line nor off-line) is able to generate a schedule for a certain sequence of jobs in a way that all the jobs' deadlines are met, the system is called *overloaded*, otherwise it is called *underloaded*. A schedule meeting all the jobs' deadlines is called *feasible*.

FIFO: The *First In First Out* scheduler processes released jobs in order of their absolute release times r_i. Thereby always the job with the smallest release time is scheduled, ties broken arbitrarily. Note that this scheduler does not take deadlines into account for its scheduling decision. Therefore, a very urgent job[10] released shortly after a very demanding one will certainly not make its deadline.

SP: The *Static Priority* scheduler processes released jobs in order of a fixed priority assigned to every job. In preemptive scheduling the release of a higher priority

[10] We denote jobs with "short" relative deadlines *urgent*.

job interrupts the execution of a lower priority job. Note that this scheduler also does not take deadlines into account for its scheduling decisions, however, by assigning the priorities w.r.t. the relative deadlines of the jobs, urgent jobs can be favored (see next item).

RMS: *Rate Monotonic Scheduling* is a special case of SP scheduling for preemptive jobs with periodic arrival patterns and relative deadlines equal to the period. In RMS the priority of a job is related to the period of the job, i.e., the shorter the period, the higher the priority. Liu and Leyland introduced a sufficient condition for schedulability in [LL73].

EDF: In *Earliest Deadline First* scheduling, the scheduler processes released jobs in the order of their absolute deadlines, i.e., the job with the most urgent deadline is scheduled. For preemptive uniprocessor scheduling, in [Der74] it has been shown that EDF is optimal with respect to feasibility, i.e., if there exists a feasible schedule for a certain job sequence, using EDF will also meet all the deadlines.

LL: A *Least Laxity* scheduler prefers jobs with smaller slack time, i.e., the maximum time a job can be delayed after its release while still completing within its deadline. At the release time of a job its laxity L_i can be calculated by $L_i = D_i - C_i$.

SRT: The classic *Shortest Remaining Time* scheduler does not take deadlines into account for its scheduling, instead it prefers jobs with a smaller amount of remaining time until completion. One advantage of SRT scheduling is that very short jobs are executed quickly.

While EDF is optimal in underloaded systems, it tends to perform very poorly in overloaded systems: the arrival of a very urgent job may start a domino effect that causes all currently preempted jobs to miss their deadlines. In the presence of overload, it thus makes sense to distinguish between urgency and importance of task instances. More specifically, each task is assigned a utility value V_i that is rewarded to the scheduler if an instance of the task is completely scheduled prior to its deadline. The "importance" of a task can be expressed by its *value density*, which is defined as the utility value of a task divided by its computation time and was first introduced in [BKM+91]. The *importance ratio* of a taskset is the ratio of the largest value density to the smallest value density of its tasks. If the importance ratio is 1, the taskset is called *uniform*.

The goal of an on-line scheduler is to maximize the cumulated utility of a job sequence. The utility values are used in more involved schedulers like TD1 [BKM+92], DSTAR [BKM+91] or DOVER [KS95].

TD1: TD1 is an on-line scheduler specifically designed to guarantee that at least 1/4 of the overload duration is used for jobs that actually finish not later than their deadlines. The simpler version introduced by Baruah et al. in [BKM+92] only works for zero-laxity tasks (i.e., $C_i = D_i$), however they also introduced a more complex version relaxing this constraint. The pseudo code of the simpler version is shown in Algorithm 5 and intuitively works like this:

Algorithm 5: Pseudo code of scheduling algorithm TD1 (simple version)

Variables: $\Delta' := 0$; $\Delta := 0$; $v_{run} = 0$; $J_{run} := \varnothing$

1 **When** J_{next} *is released, where* $k > 0$ *is remaining processing time of* J_{run} **do**
2 $\quad \Delta \leftarrow \max\{\Delta, \Delta' - k + C_{next}\}$
3 $\quad$ **if** $v_{run} < \Delta/4$ **then**
4 $\quad\quad$ abort J_{run}
5 $\quad\quad$ $\Delta' \leftarrow \Delta$
6 $\quad\quad$ schedule J_{next}
7 $\quad\quad$ $J_{run} \leftarrow J_{next}$; $v_{run} \leftarrow C_{next}$
8 **When** J_{run} *completes* **do**
9 $\quad$ $\Delta' \leftarrow 0$; $\Delta \leftarrow 0$; $v_{run} \leftarrow 0$
10 $\quad$ $J_{run} \leftarrow \varnothing$

TD1 keeps track of the time spent for the current and previously discarded jobs (cf. Δ' in the pseudo code). Whenever a new job is released, TD1 checks if scheduling the new job is worth discarding the currently running job. It only discards the current job, if the time for executing the new job in addition to the time spent for discarded jobs is more than four times higher than the current job's execution time (cf. Line 3). If the execution time of the new task is high enough to satisfy this condition, the algorithm discards the currently running job, updates the bookkeeping of the time for the current and the discarded jobs, and schedules the new job. Otherwise the new job is discarded. If a job is successfully finished, the time spent for discarded and currently running jobs is reset to zero.

Similarly as TD1 does with the execution time of discarded jobs, DSTAR and DOVER keep track of the cumulative utility guaranteed by currently preempted jobs. As the pseudo codes for the algorithms are quite extensive they are not explicitly stated in this work. Hence we refer to the corresponding publications (DSTAR in [BKM+91] and DOVER in [KS95]) and provide only an intuitive description of the algorithms:

A currently running job is preempted only in two cases: (1) a new job with a more urgent deadline is released that can be scheduled without making any currently preempted job violating its deadline (i.e., the new job does not generate an overload scenario together with the currently preempted jobs). (2) A previously released but currently not scheduled or preempted job reaches zero-laxity and this job's utility is high enough that it is worth discarding all currently preempted and the running job in favor of it (e.g., for DOVER the utility of this job must be more than $1 + \sqrt{k}$ times the "guaranteed" utility that is rewarded to the scheduler if it would be sticking to the already started jobs). While (1) corresponds to the behavior of EDF in an underloaded system, (2) guarantees a "sufficiently large" cumulative utility.

2.4.2 Time/Utility Functions

Previously we introduced the utility value of a task as the reward given to the scheduler if an instance of the task is completely scheduled prior to its deadline. This timing constraint of a job regarding its finishing time can be generalized to a time/utility function, as described in [JLT85]. To this end, the utility value assigned to a task τ_i is substituted by a utility function $v_i[f_{i,j} - r_{i,j}]$ specifying the utility value rewarded to the corresponding job $J_{i,j}$, parametrized by its release time $r_{i,j}$ and its finishing time $f_{i,j}$.[11] Depending on the type of the function different classes of tasks can be distinguished; four commonly used ones are shown in Figure 2.11 and described in the following.

Non real-time task: If a task does not have time constraints on its finishing time, it always contributes to the system and therefore has a constant utility value, no matter when it completes its execution, visualized in Figure 2.11a.

Soft real-time task: A task with a soft deadline can contribute to the system even though its deadline has already passed, at least providing a reduced utility value as shown in Figure 2.11b.

Firm real-time task: If a task does not provide any utility value after its deadline but a deadline miss does not have catastrophic consequences, it is called a firm deadline task. It corresponds to a utility function that drops to zero when the last piece of work is not finished by the deadline, as shown in Figure 2.11c.

Hard real-time task: If a deadline miss might have catastrophic consequences [Kop97], a task is called a hard real-time task and the value of the corresponding utility function drops to minus infinity when the last piece of work is not finished by the deadline, shown in Figure 2.11d. Thus, one missed deadline makes all previously and future gained utility void.

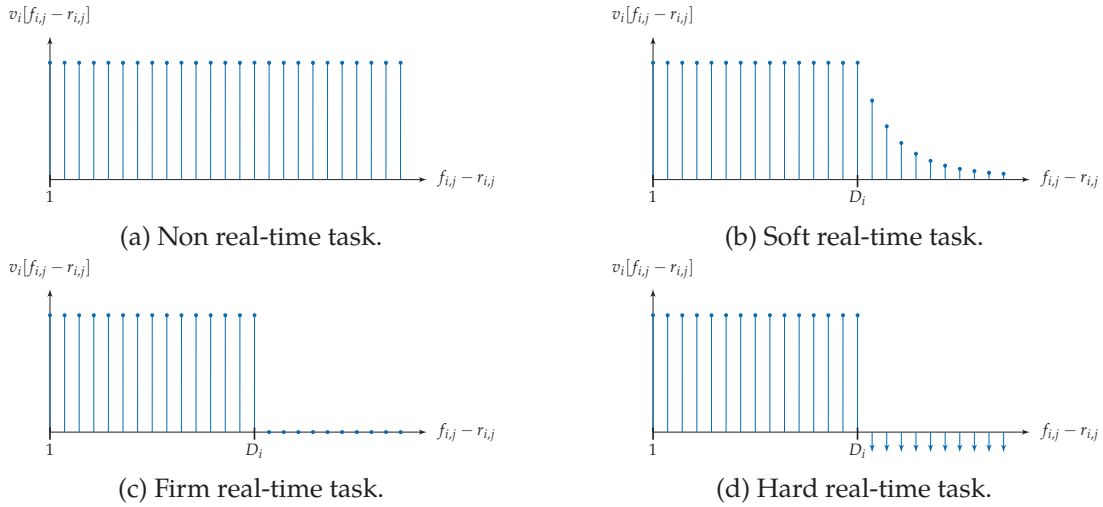
In the remainder of this thesis, we will assume that the jobs of all tasks can be preempted and that they have firm deadlines. To simplify our notation, we say that we assign a utility value V_i to task τ_i if we actually assign the utility function $v_i[f_{i,j} - r_{i,j}] = V_i$ for $f_{i,j} - r_{i,j} \in [1, D_i]$ and 0 else.

2.4.3 Comparing Schedulers

In an environment, where the arrival patterns of the jobs are not predictable, an on-line scheduler is required that makes scheduling decisions only based on present and past job invocations. In particular, it does not know anything about future job releases at the time of the decision. In contrast, an off-line or *clairvoyant* scheduler is aware of all past and future job releases.

As noted before, for preemptive uniprocessor scheduling (without overload), EDF is optimal with respect to feasibility, i.e., if there exists a feasible schedule for a certain

[11] Note that $f_{i,j} \geqslant r_{i,j} + 1$ due to the discrete time model.



(a) Non real-time task.

(b) Soft real-time task.

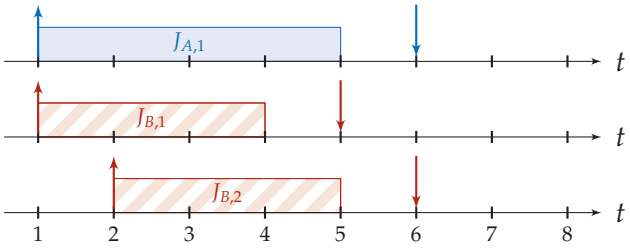
(c) Firm real-time task.

(d) Hard real-time task.

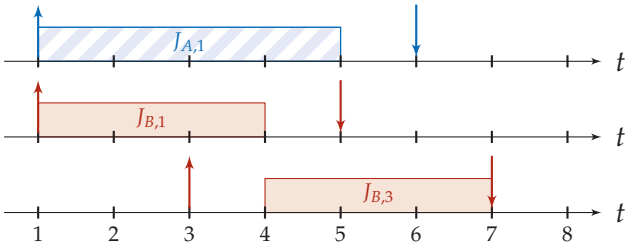
Figure 2.11: Time/utility functions of different kinds of real-time tasks.

scheduling instance, using EDF will meet all the deadlines. If no feasible schedule exists, that is, if the system is overloaded, it can easily be shown that there can be no optimal on-line algorithm for most problems such as maximizing the cumulative utility. Figure 2.12 gives an intuition for this fact using the example of cumulative utility. It shows three different scheduling scenarios, each with one instance of task $\tau_A : \{C_A = 4, D_A = 5, V_A = 4\}$, released at $t = 1$ and denoted $J_{A,1}$, and two instances of task $\tau_B : \{C_B = 3, D_B = 4, V_B = 3\}$, denoted $J_{B,j}$ where j is the release time of the corresponding jobs. While the first job is released at $t = 1$ and thus denoted $J_{B,1}$, the second one is released at $t \geqslant 2$, differently for each scenario. The example demonstrates that without the knowledge of the release time of the second instance of task τ_B already at time $t = 1$, there is no way to maximize the cumulative utility. Assuming that the second instance of τ_B is released at time $t = 2$, the maximum cumulative utility is 4 and is achieved by scheduling $J_{A,1}$. However, if it is released at time $t = 3$, the maximum cumulative utility is 6, achieved by scheduling both instances of task τ_B. If the second job is released at time $t = 4$ or later, the maximum cumulative utility is 7, achieved by scheduling $J_{A,1}$ and $J_{B,j}$ with $j \geqslant 4$. At time $t = 1$, when the first scheduling decision has to be made, all three scenarios are indistinguishable for an on-line scheduler. Thus, no on-line scheduler can guarantee the optimal cumulative utility in all cases.

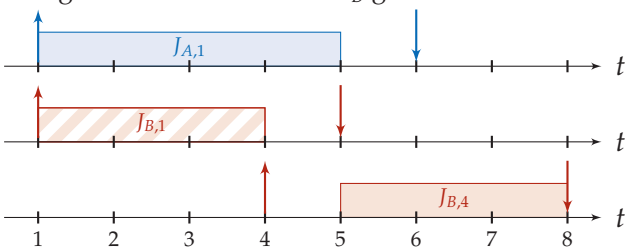
The dominant approach to characterize the performance of a scheduling algorithm for firm deadline tasks, and therefore to compare different scheduling algorithms, is to evaluate its worst case performance with respect to an optimal solution, called *competitive analysis* [BEY98].



(a) Scheduling the instance of task τ_A gains a utility value of 4.



(b) Scheduling both instances of task τ_B gains a cumulative utility of 6.



(c) Scheduling the instance of task τ_A together with the second instance of task τ_B gains a cumulative utility of 7.

Figure 2.12: Three different scheduling scenarios indistinguishable at time $t = 1$. Taskset consists of $\tau_A : \{C_A = 4, D_A = 5, V_A = 4\}, \tau_B : \{C_B = 3, D_B = 4, V_B = 3\}$.

Competitive Ratio vs. Competitive Factor

We introduce the notion of the *competitive ratio* as the worst case utility ratio a on-line scheduler can achieve with respect to an optimal clairvoyant algorithm *on a certain taskset*. To the best of our knowledge, nobody has investigated this problem before. However, Baruah et al. [BKM+91] introduced the more general concept of the *competitive factor* of an on-line scheduler as the worst case utility ratio a on-line scheduler can achieve with respect to an optimal clairvoyant algorithm for any taskset, i.e., without the restriction on a certain taskset. The only restriction in the analysis is a finite duration of overload, as the competitive factor would always be zero otherwise. Clearly, the competitive factor is a lower bound on the competitive ratio.

Baruah et al. have shown the general result that in the case of an overloaded system for a taskset consisting of tasks with firm deadlines and with importance ratio k, no on-line scheduler can guarantee a competitive ratio of more than $1/(1+\sqrt{k})^2$, i.e., there is an upper bound on the competitive factor of $1/4$. Note that this upper bound on the competitive factor is based on constructing a specific job sequence, which takes into account the on-line algorithm's actions and thereby forces it to deliver a sub-optimal cumulative utility. Note that the competitive ratio of some given algorithm is usually higher, as it is very unlikely that the tasks needed to construct the quite exotic worst case job sequence of [BKM+91] are actually available in the given taskset.

EDF, as an on-line scheduling algorithm that, in the case of an underloaded system, has a competitive factor of 1, turns out to have a particularly bad competitive ratio (and therefore also a bad competitive factor) of 0 in the general case.

Several algorithms have been proposed that were optimized for achieving a competitive factor that comes close to or matches the upper bound during overloads.

DSTAR [BKM+91] achieves a competitive factor of $1/5$ during overload and a competitive factor of 1 if the system is not overloaded. However, the algorithm is restricted to uniform tasksets ($k = 1$), thus leaving a gap to the upper bound of $1/4$.

DDSTAR [KS91] achieves a competitive factor of $1/4$ during overload and 1 if the system is not overloaded; as it is also restricted to uniform tasksets, it matches the upper bound with respect to cumulative utility.

TD1 [BKM+92] comes in two versions. One that only works for zero-laxity tasks, and one without this restriction. It guarantees a competitive factor of 1 if the system is not overloaded and a competitive factor of $1/4$ in overloaded systems for uniform tasksets.

DOVER [KS95], as a worst-case optimal algorithm, guarantees not only the upper bound on the competitive factor during overloads for tasksets with arbitrary importance ratios, but also a competitive factor of 1 if the system is not overloaded.

2.5 Putting it all Together

The aim of this thesis is to introduce a new method to build and analyze the real-time performance of a distributed system running processes of multiple independent synchronous distributed algorithms simultaneously on the processing nodes. The high-level system architecture is shown in Figure 2.13. In our system model we assume that every process performs computing steps at every integer multiple of some unit time, i.e., we assume lock-step synchronous processes ($\Phi = 1$). As we can assume that every node in the distributed system has enough processing power to execute the state transitions of all distributed algorithms timely, e.g., by assigning every algorithm's

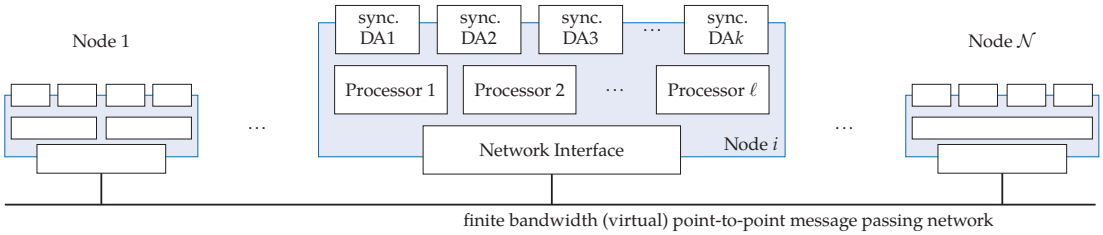


Figure 2.13: Proposed system architecture of the distributed system. Consisting of $\mathcal{N}$ processing nodes, where each node is running one of the processes of k independent synchronous distributed algorithms on its processors.

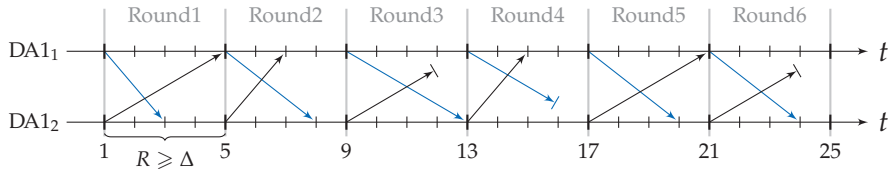
process a dedicated processor, we can focus our analysis on the communication between the processes.

As for the communication, we assume that nodes are connected with each other by a finite bandwidth message passing system shared between all the distributed algorithms' processes running on a node. As shown in Figure 2.13, every node has a network interface taking care of the access to the message passing system, which provides either dedicated or virtual point-to-point links to all other nodes.

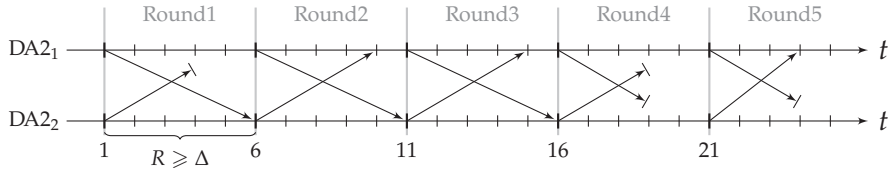
If a message sent from an algorithm process on node p to an algorithm process running on node q takes not longer than Δ time units, communication closed lock-step rounds can be built by starting a new round simultaneously every $R \geqslant \Delta$ time units on the processes. As an example, Figures 2.14a–2.14c illustrate the communication of three distributed algorithms. For simplicity, only the messages between two processing nodes are shown. Note that, as also shown in Figure 2.14, two different synchronous distributed algorithms executed in the system may have a different round durations R: While DA1 starts a new round every four time units, DA2 and DA3 start new rounds every five respectively every six time units. The algorithms may send messages of different size and given the finite bandwidth of the communication channel, different message sizes may result in different transmission durations of the messages. Furthermore, the executed algorithms do not need to be synchronized with respect to their rounds, not even when their round duration is the same.

Determining Δ for a distributed system is usually a non-trivial task. Hence, typically, one fixes R and hopes that $R \geqslant \Delta$. If the executed distributed algorithm is fault-tolerant this approach is often enough for the algorithm to work. Therefore, it may happen that in some rounds the choice of R was not large enough, as shown in Round 4 in Figure 2.14a.

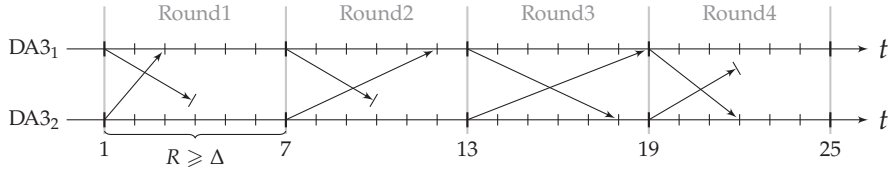
In our system model we use a more systematic approach: We explicitly model the messages of the algorithms as firm deadline real-time jobs released at the beginning of a round and with a deadline corresponding to the end of the round. A real-time scheduling algorithm is managing the access to the communication medium. In an overload scenario, where the channel capacity is not always sufficient for all messages that have to be sent, the scheduler will drop messages. Figure 2.15 shows the three algorithms executed on processing node 1 from the previously introduced example and



(a) Round structure of distributed algorithm DA1 using $R = 4$.



(b) Round structure of distributed algorithm DA2 using $R = 5$.



(c) Round structure of distributed algorithm DA3 using $R = 6$.

Figure 2.14: Communication between two processes of three synchronous distributed algorithms running on the distributed system.

their generated messages. Thereby it is assumed that algorithm DA1 and DA3 generate messages that need two time units to be transmitted while algorithm DA3 generates larger messages that need three time units. In this example, processing node 1 uses a non-preemptive earliest deadline first algorithm as its message scheduling strategy where ties are broken by the index of the algorithms. The resulting message scheduling on the channel is also shown in Figure 2.15. The messages sent by DA1 are highlighted in blue and directly correspond to the blue arrows in Figure 2.14a.

Additionally, the unreliable communication channels introduce a probability that a message gets lost during transmission (e.g., it gets corrupted and therefore does not make it through the receiver's parity check).

The send omissions from the real-time scheduler, as well as the link failures can either be handled directly by a fault-tolerant distributed algorithm or, as in the model proposed in this thesis, handled by a synchronizer algorithm, providing a virtual round structure without message loss to the distributed algorithm. To this end, we abstract dropped messages together with the link failures by a probabilistic link failure model. Sticking to the previous example, the analysis of the hyper-period generated by the three algorithms results in a probability of 13/15 that a message sent by DA1's process is successfully scheduled.

Analyzing the performance of the synchronizer, parametrized by a suitable probability of successful message transmission, thus allows to state the expected round

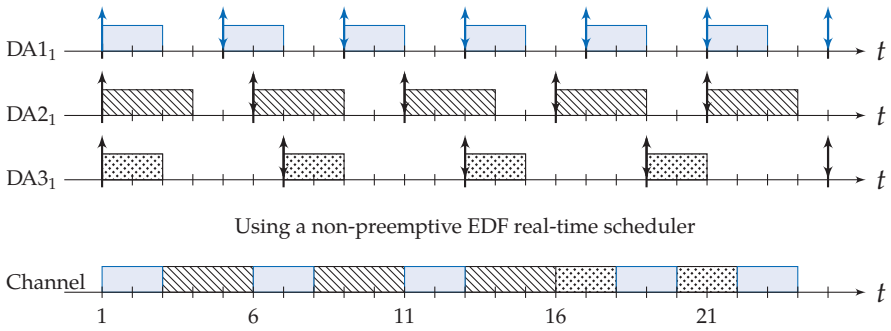


Figure 2.15: Messages generated by the three distributed algorithms on processing node 1 and how they get scheduled on the shared channel to another processing node.

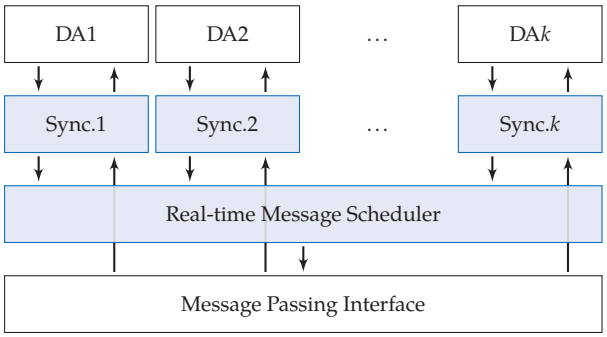


Figure 2.16: Layered structure of one node in the proposed system model, running processes of k independent distributed algorithms. Contributions of this thesis are colored.

duration of the synchronous distributed algorithm running atop. To achieve this goal, we suggest a layered approach described in the following.

The general structure of every processing node can be divided into four layers shown in Figure 2.16. While the message passing interface and the distributed algorithms are only of minor interest, the real-time message scheduler and the synchronizer (both highlighted in the figure) form the major contributions of this thesis. The four layers are described more in detail in the following.

Message passing interface

As mentioned before, we assume a fully connected synchronous message passing network. Therefore we either need dedicated or virtual point-to-point links between all processes and thus nodes. The requirements of anonymity heavily depend on the distributed algorithms finally running on the uppermost level. For the results in this thesis it is sufficient if received messages can be traced back to their incoming port. As for the sending it is sufficient if the message passing interface provides a send-to-all primitive. The presence of unique IDs to identify a process is not needed. Furthermore,

we assume that the links between the nodes are full-duplex, each direction has a finite bandwidth shared by all distributed algorithms running on a node, and that they are unreliable. Therefore, two origins of message loss can be identified: (i) Messages dropped by the scheduler due to the finite bandwidth constraint of the link, and (ii) messages lost due to the unreliable communication. For tractability, we assume that both effects can be jointly abstracted via a probabilistic link failure model, where a message is received only with a certain (non-zero) probability.

Real-time message scheduler

Finite bandwidth on the channels of the message passing system implies the need for some scheduling whenever the demand exceeds the available bandwidth. As round-based distributed systems assume communication closedness, i.e., all messages are delivered before the next round starts, a message does not provide any value if it is delivered late. Thus, we use a real-time scheduling algorithm and model messages as jobs where the release times of the jobs correspond to the time of the processing step at which the messages are sent, the deadline corresponds to the time of the processing step starting a new round, and the worst case execution time is proportional to the amount of bandwidth needed to transmit the message.

As a message does not provide any value to the system after the next round has started, but due to the synchronizer at the receiver does not harm, a firm deadline job is a well suited model for it. If, because of an overload scenario, a message cannot be delivered timely it is dismissed by the scheduler, corresponding to an omission fault in classic distributed systems.

In this work we provide a game theoretic framework, in particular using multi-objective graphs and graph games, to automatically analyze the competitiveness of on-line scheduling algorithms with respect to a given taskset. Using this framework we can compare different scheduling algorithms w.r.t. their worst case average utility on a particular taskset. Furthermore, we can also restrict the adversary by applying constraints on release patterns of tasks (such as periodic tasks, maintaining a sporadicity interval, or restriction on the maximum workload released in a certain time), as well as constraints on the long run behavior of the system (such as limiting the released workload on average). We also give a generic, but computationally very expensive method to synthesize the optimal on-line scheduler for a specific taskset.

Chapter 3 is devoted to these real-time scheduling aspects.

Round synchronization

A retransmission-based synchronizer is used to regain a consistent round structure on top of the omission and link failure affected communication.

To reduce complexity, the message losses of the previous layers (the message drops by the real-time scheduler as well as the probabilistic link failures) are combined and abstracted by a probabilistic link failure model. Therefore, we assume a (non-zero) probability that a message is successfully received. The analysis of the synchronizer

focuses on the processing steps of the synchronizer where it actually processes received messages and sends new messages. As the real-time performance of the atop running synchronous distributed algorithm heavily depends on the performance of this synchronizer, an accurate prediction of the durations of the simulated rounds is crucial for drawing meaningful conclusions about the performance of the higher-level algorithm.

To this end we use probability analysis as well as Markov chain modeling to analyze the expected round duration for the synchronizer. In general, these calculations are computationally very expensive, therefore we introduce additional model restrictions to get rid of the inherent complexity of the problem without weakening the applicability too much, e.g., by assuming (m,k)-firm deadline scheduling for the real-time scheduler to bound the number of retransmissions to a finite number.

The performance of the synchronizer is studied in Chapter 4.

Distributed algorithms

Processes of different synchronous algorithms can run simultaneously on the distributed system. As the round synchronization layer below provides a round structure, the strongest canonic synchrony assumption, all distributed algorithms can be run as long as their needs of execution and communication primitives are matched by the message passing system, or can at least be simulated atop.

The modeling approach introduced in this thesis provides expected values for the round durations of the algorithms with respect to real-time. The synchronizer analysis, carried out with a suitable probability of successful message transmission extracted from the real-time scheduling and link failure analysis, combined with a classic performance analysis of the algorithms (i.e., round complexity for a synchronous algorithm or maximum number of processing steps for an asynchronous algorithm) thus allows to state the expected real-time performance of the synchronous distributed algorithm.

Communication between the processes of a distributed algorithm

Figure 2.17 provides deeper insights into how the communication between the processes of a distributed algorithm actually works. It shows part of a distributed system with three nodes running three distributed algorithms simultaneously. In this example we retain the timing properties of the algorithms as in the previous Figures 2.14 and 2.15. Algorithm DA1 performs a processing step every four time units where it sends messages that need two time units for successful transmission. Messages of DA1 are solidly filled in blue and events related to the distributed algorithm DA1, the main point of interest in the figure, are marked in blue. Algorithms DA2 and DA3 perform their processing steps every five respectively six time units. DA2 sends messages that need three time units for transmission (filled with diagonal stripes in the image) while DA2 sends messages that need two time units for transmission (filled with a dotted pattern).

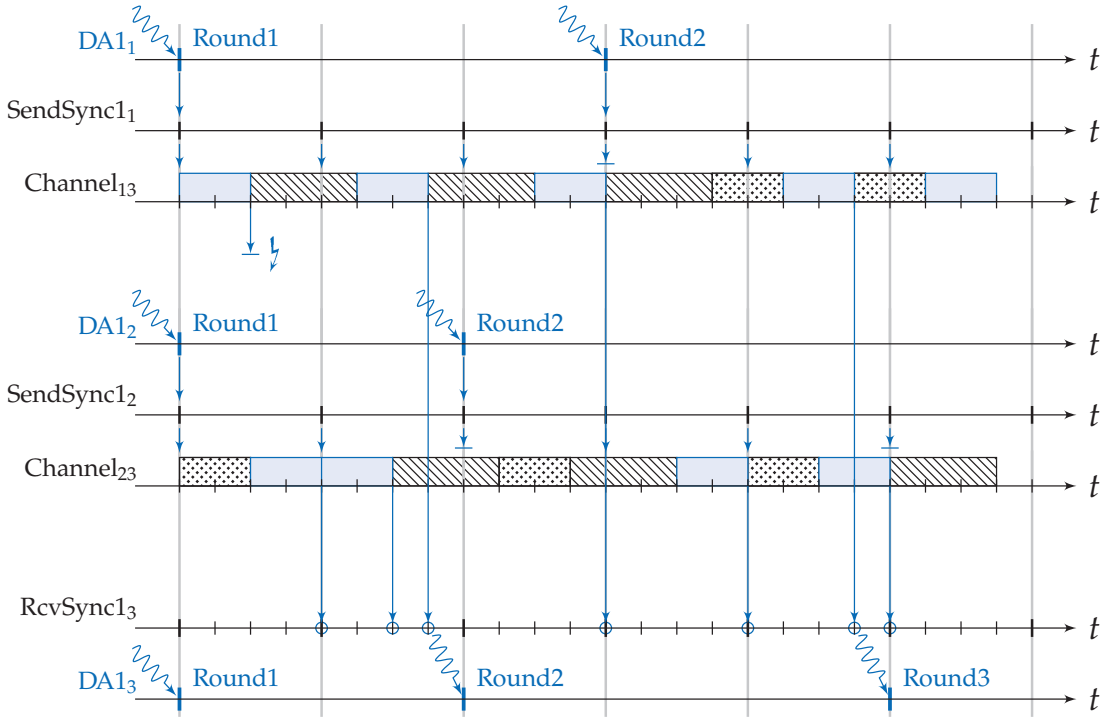


Figure 2.17: Detailed communication schema for sending messages within an execution of distributed algorithm DA1. Events and messages related to DA1 are colored.

The upper part of the time/space diagram is devoted to the transmitter part of node 1 containing the synchronous distributed algorithm DA1, the transmitter part of the synchronizer, as well as the communication channel between node 1 and 3. The middle part of the diagram shows the transmitter part of node 2, and the lower part of the diagram shows the receiver part of node 3 containing the receiver part of the synchronizer and the distributed algorithm DA1. The snake-arrows in the diagram denote the start of a new round of DA1 on the respective node. At such a round switch the process of the distributed algorithm performs its state transition, generating messages that need to be sent. Therefore, the process of DA1 triggers the transmitter part of the retransmission-based synchronizer (labeled SendSync1 in the figure) and provides the algorithm messages for the new round to it. As noted before, the synchronizer performs its computation steps every four time units, marked by the gray vertical lines. In every step, it tries to send the message of DA1 via the message passing system to the corresponding receiver. In the figure, only the (virtual) channels to node 3 are shown, labeled Channel$_{x3}$. As shown in Figure 2.15, the bandwidth of the channel is not sufficient to transmit the messages of all algorithms timely.

The transmitter part of the synchronizer sends the messages generated by the algorithm in every step, i.e., it periodically releases jobs corresponding to the messages at every step, each with a relative deadline equal to the period duration of the steps.In case of node 1, the channel allocation results from using the same non-preemptive

earliest deadline first scheduler as in Figure 2.15 while node 2 uses a different scheduling strategy.

In the first step of the illustrated execution of the system, the message sent by DA1 on node 1 is successfully scheduled on the channel but gets corrupted during transmission and does not make it to its receiver (illustrated by the $\lightning$ symbol). The message is retransmitted in the next step of the synchronizer, again scheduled on the channel, and this time successfully received by the process running on node 3 timely before the next processing step. The delivery of the message triggers an round switch at process $DA1_3$ (illustrated by the snake arrow), as the round 1 message from $DA1_2$ was already received earlier (both attempts were successful).

Note that the round 2 message of the process running on node 1 also needs two transmission attempts as in the first attempt the message sent by DA2 is more urgent. Therefore the DA1 message is discarded in favor of the DA2 message.

Chapter 3

Real-Time Scheduling

In this chapter, we study the well-known problem of scheduling a sequence of dynamically arriving real-time task instances with firm deadlines on a single processor. The problem is analogous to the originally introduced problem of message scheduling on a communication channel; in literature, however, task scheduling is more common than message scheduling and thus we go with this slightly shifted focus.

As pointed out in the previous chapter, in firm deadline scheduling, a task instance (a *job*) that is completed by its deadline contributes a positive utility value; a job that does not meet its deadline does not harm, but does not add any utility. The goal of the scheduling algorithm is to maximize the cumulated utility. Firm deadline tasks arise in various application domains, e.g., machine scheduling, multimedia and video streaming, QoS management in switches and data networks, and other systems that may suffer from overload [KS95].

Competitive analysis [BEY98] has been the primary tool for studying the performance of such scheduling algorithms [BKM+92]. In general, it allows to compare the performance of an *on-line* algorithm $\mathcal{A}$, which processes a sequence of inputs without knowing the future, with what can be achieved by an optimal *off-line* algorithm $\mathcal{C}$ that does know the future (a *clairvoyant* algorithm): The *competitive factor* gives the worst-case performance ratio of $\mathcal{A}$ vs. $\mathcal{C}$ over all possible scenarios.

Since the taskset arising in a particular application is usually known, this thesis focuses on the competitive analysis problem for *given* tasksets: Rather than from *all* possible tasksets as in [BKM+92], the job sequences used for computing the *competitive ratio* are chosen from a taskset given as an input. There are two relevant problems for the automated competitive analysis for a given taskset: (1) The *synthesis* question asks to find an algorithm with optimal competitive ratio; and (2) the *analysis* question asks to compute the competitive ratio of a given on-line algorithm. In this chapter we will address both questions.

More specifically, on the analysis side, we provide a flexible, automated analysis framework that also supports additional constraints on the adversary, such as sporadicity constraints and longrun-average load. We show that the analysis problem (with additional constraints) can be reduced to a multi-objective graph problem, which can be solved in polynomial time. We also present several optimizations and an experimental evaluation of our algorithms that demonstrates the feasibility of our approach, which effectively allows to replace human ingenuity (required for finding worst-case scenarios) by computing power: Using our framework, the application designer can analyze different scheduling algorithms for the specific tasksets arising in her/his particular application, and compare their competitive ratio in order to select the best one.

On the synthesis side, we provide a reduction from labeled transition systems, our preferred way to model scheduling algorithms to partial-observation graph games. We furthermore show that deciding whether there is an on-line algorithm with competitive ratio above some threshold is NP-complete.

Detailed Road Map of this Chapter:

This chapter is based on the published articles [CKS13] and [CPKS14] in a joint work with Krishnendu Chatterjee, Andreas Pavlogiannis, and Ulrich Schmid. While the author's contribution covers the modeling and the automated generation of the on-line algorithms' state spaces, the credits for the contributions of the state space reduction and solving the resulting graph problems are awarded to Andreas Pavlogiannis.

In Section 3.1, we will formally define our scheduling problem along the relevant additional constraints that can optionally restrict the power of the adversary. In Section 3.2, we introduce labeled transition systems as a formal model for specifying on-line and off-line scheduling algorithms. In Section 3.3, we present the formal framework to specify the constraints on the adversary, and argue how it allows to model a wide variety of constraints. We also give an overview of all the steps involved in our approach in Section 3.4.

In Section 3.5, we present the *multi-objective graphs* used by our solution algorithm. Multiple objectives are required to represent the competitive analysis problem with various constraints. In Section 3.6, we describe a theoretical reduction of the competitive analysis problem to solving a multi-objective graph problem, where the graph is obtained as a product of the on-line algorithm, an off-line algorithm, and the constraints specified as automata. Our algorithmic solution is polynomial in the size of the graph; however, the product graph can be large for representative tasksets. In Section 3.7, we present both general and implementation-specific optimizations, which considerably reduce the size of the resulting graphs. In Section 3.8, we provide competitive ratio analysis results obtained by our method. More specifically, we present a comparative study of the performance of several existing firm deadline real-time scheduling algorithms. Our results show that, for different tasksets (even with no constraints), different algorithms achieve the highest competitive ratio (i.e., there is no universal optimal algorithm). Moreover, even for a fixed taskset and varying constraints on the

adversary, different algorithms achieve the highest competitive ratio. This highlights the importance of our framework for selecting optimal algorithms for specific applications. Furthermore, we construct a series of the worst-case task sets for the scheduler TD1 using the recurrence given in [BKM+92] and show the convergence of the competitive ratio to 1/4.

In Section 3.9, we introduce graph games as a formal model to describe the introduced real-time scheduling problems and their objectives. In Section 3.10, we give a reduction from 3-SAT to partial-information games and thus show that the synthesis problem is NP-complete. In Section 3.11, we show that the general partial-information game model with ratio objectives is suitable to construct an on-line scheduler with optimal competitive ratio. Finally, Section 3.12 provides additionally bibliographic information about this chapter.

3.1 Formal Problem Definition

We consider a finite set of tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_N\}$, to be executed on a single processor. We assume a discrete notion of real-time $t = k\varepsilon$, $k \geqslant 1$, where $\varepsilon > 0$ is both the unit time and the smallest unit of preemption (called a *slot*). Since both task releases and scheduling activities occur at slot boundaries only, all timing values are specified as positive integers. Every task τ_i releases countably many task instances (called *jobs*) $J_{i,j} := (\tau_i, j) \in \mathcal{T} \times \mathbb{N}^+$ (where $\mathbb{N}^+$ is the set of positive integers) over time (i.e., $J_{i,j}$ denotes that a job of task i is released at time j). All jobs, of all tasks, are independent of each other and can be preempted and resumed during execution without any overhead. Every task τ_i, for $1 \leqslant i \leqslant N$, is characterized by a 3-tuple $\tau_i = (C_i, D_i, V_i)$ consisting of its non-zero *worst-case execution time* $C_i \in \mathbb{N}^+$ (slots), its non-zero *relative deadline* $D_i \in \mathbb{N}^+$ (slots) and its non-zero *utility value* $V_i \in \mathbb{N}^+$ (rational utility values $V_1, \ldots, V_N$ can be mapped to integers by proper scaling). We denote with $D_{\max} = \max_{1 \leqslant i \leqslant N} D_i$ the maximum relative deadline in $\mathcal{T}$. Every job $J_{i,j}$ needs the processor for C_i (not necessarily consecutive) slots exclusively to execute to completion. All tasks have firm deadlines: only a job $J_{i,j}$ that completes within D_i slots, as measured from its release time, provides utility V_i to the system. A job that misses its deadline does not harm but provides zero utility. The goal of a real-time scheduling algorithm in this model is to maximize the *cumulated utility*, which is the sum of V_i times the number of jobs $J_{i,j}$ that can be completed by their deadlines, in a sequence of job releases generated by the *adversary*.

Notation on Sequences

Let X be a finite set. For an infinite sequence $x = (x^\ell)_{\ell \geqslant 1} = (x^1, x^2, \ldots)$ of elements in X, we denote by x^ℓ the element in the ℓ-th position of x, and denote by $x(\ell) = (x^1, x^2, \ldots, x^\ell)$ the finite prefix of x up to position ℓ. We denote by X^∞ the set of all infinite sequences of elements from X. Given a function $f : X \to \mathbb{Z}$ (where $\mathbb{Z}$ is the set of integers) and a sequence $x \in X^\infty$, we denote with $f(x,k) = \sum_{\ell=1}^{k} f(x^\ell)$ the sum of the images of the first k sequence elements under f.

Job Sequences

When generating a job sequence, the adversary releases at most one new job from every task in every slot. Formally, the adversary generates an infinite *job sequence* $\sigma = (\sigma^\ell)_{\ell \geqslant 1} \in \Sigma^\infty$, where $\Sigma = 2^{\mathcal{T}}$. If a task τ_i belongs to σ^ℓ, for $\ell \in \mathbb{N}^+$, then a (single) new job $J_{i,j}$ of task i is released at the beginning of slot ℓ: $j = \ell$ denotes the *release time* of $J_{i,j}$, which is the earliest time $J_{i,j}$ can be executed, and $d_{i,j} = j + D_i$ denotes its absolute *deadline*.

Admissible Job Sequences

We present a flexible framework where the set of admissible job sequences that the adversary can generate may be restricted. The set $\mathcal{J}$ of *admissible* job sequences from Σ^∞ can be obtained by imposing one or more of the following (optional) admissibility restrictions:

($\mathcal{S}$) Safety constraints, which are restrictions that have to hold in every finite prefix of an admissible job sequence; e.g., they can be used to enforce job release constraints such as periodicity or sporadicity, and to impose temporal workload restrictions.

($\mathcal{L}$) Liveness restrictions, which assert infinite repetition of certain job releases in a job sequence; e.g., they can be used to force the adversary to release a certain task infinitely often.

($\mathcal{W}$) Limit-average constraints, which restrict the long run average behavior of a job sequence; e.g., they can be used to enforce that the average load in the job sequences does not exceed a threshold.

Schedule

Given an admissible job sequence $\sigma \in \mathcal{J}$, the *schedule* $\pi = (\pi^\ell)_{\ell \geqslant 1} \in \Pi^\infty$, where $\Pi = ((\mathcal{T} \times \{0, \ldots, D_{\max} - 1\}) \cup \varnothing)$, computed by a real-time scheduling algorithm for σ, is a function that assigns at most one job for execution to every slot $\ell \geqslant 1$: π^ℓ is

either $\varnothing$ (i.e., no job is executed) or else (τ_i, j) (i.e., the job $J_{i,\ell-j}$ of task τ_i released j slots ago is executed). The latter must satisfy the following constraints:

1. $\tau_i \in \sigma^{\ell-j}$ (the job has been released),
2. $j < D_i$ (the job's deadline has not passed),
3. $|\{k : k > 0 \text{ and } \pi^{\ell-k} = (\tau_i, j') \text{ and } k + j' = j\}| < C_i$ (the job released in slot $\ell - j$ has not been completed).

Note that our definition of schedules uses relative indexing in the scheduling algorithms: At time point ℓ, the algorithm for schedule π^ℓ uses index j to refer to slot $\ell - j$. Recall that $\pi(k)$ denotes the prefix of length $k \geqslant 1$ of π. We define $\gamma_i(\pi, k)$ to be the number of jobs of task τ_i that are completed by their deadlines in $\pi(k)$. The cumulated utility $V(\pi, k)$ (also called utility for brevity) achieved in $\pi(k)$ is defined as $V(\pi, k) = \sum_{i=1}^{N} \gamma_i(\pi, k) \cdot V_i$.

Competitive Ratio

We are interested in evaluating the performance of deterministic *on-line* scheduling algorithms $\mathcal{A}$, which, at time ℓ, do not know any of the σ^k for $k > \ell$ when running on $\sigma \in \mathcal{J}$. In order to assess the performance of $\mathcal{A}$, we will compare the cumulated utility achieved in the schedule $\pi_\mathcal{A}$ to the cumulated utility achieved in the schedule $\pi_\mathcal{C}$ provided by an optimal *off-line* scheduling algorithm, called a *clairvoyant* algorithm $\mathcal{C}$, working on the same job sequence. Formally, given a taskset $\mathcal{T}$, let $\mathcal{J} \subseteq \Sigma^\infty$ be the set of all admissible job sequences of $\mathcal{T}$ that satisfy given (optional) safety, liveness, and limit-average constraints. For every $\sigma \in \mathcal{J}$, we denote with $\pi^\sigma_\mathcal{A}$ (resp. $\pi^\sigma_\mathcal{C}$) the schedule produced by $\mathcal{A}$ (resp. $\mathcal{C}$) under σ. The *competitive ratio* of the on-line algorithm $\mathcal{A}$ for the taskset $\mathcal{T}$ under the admissible job sequence set $\mathcal{J}$ is defined as

$$\mathcal{CR}_\mathcal{J}(\mathcal{A}) = \inf_{\sigma \in \mathcal{J}} \liminf_{k \to \infty} \frac{1 + V(\pi^\sigma_\mathcal{A}, k)}{1 + V(\pi^\sigma_\mathcal{C}, k)} \tag{3.1}$$

that is, the worst-case ratio of the cumulated utility of the on-line algorithm versus the clairvoyant algorithm, under all admissible job sequences. Note that adding 1 in numerator and denominator simply avoids division by zero issues.

> *Remark* 1. Since, according to the definition of the competitive ratio $\mathcal{CR}_\mathcal{J}$ in Equation (3.1), we focus on worst-case analysis, we do not consider randomized algorithms (such as Locke's best-effort policy [Loc86]). Generally, for worst-case analysis, randomization can be handled by additional choices for the adversary. For the same reason, we do not consider scheduling algorithms that can use the unbounded history of job releases to predict the future (e.g., to capture correlations).

3.2 Labeled Transistion Systems as Models for Algorithms

We will consider both on-line and off-line scheduling algorithms that are formally modeled as *labeled transition systems (LTSs)*: Every deterministic finite-state on-line scheduling algorithm can be represented as a deterministic LTS, such that every input job sequence generates a unique run that determines the corresponding schedule. On the other hand, an off-line algorithm can be represented as a non-deterministic LTS, which uses the non-determinism to guess the appropriate job to schedule.

Labeled Transition Systems (LTSs)

Formally, a *labeled transition system* (LTS) is a tuple $L = (S, s_1, \Sigma, \Pi, \Delta)$, where S is a finite set of states, $s_1 \in S$ is the initial state, Σ is a finite set of input actions, Π is a finite set of output actions, and $\Delta \subseteq S \times \Sigma \times S \times \Pi$ is the transition relation. Intuitively, $(s, x, s', y) \in \Delta$ if, given the current state s and input x, the LTS outputs y and makes a transition to state s'. If the LTS is deterministic, then there is always a unique output and next state, i.e., Δ is a function $\Delta : S \times \Sigma \to S \times \Pi$. Given an input sequence $\sigma \in \Sigma^\infty$, a *run* of L on σ is a sequence $\rho = (p_\ell, \sigma_\ell, q_\ell, \pi_\ell)_{\ell \geqslant 1} \in \Delta^\infty$ such that $p_1 = s_1$ and for all $\ell \geqslant 2$, we have $p_\ell = q_{\ell-1}$. For a deterministic LTS, for each input sequence, there is a unique run.

3.2.1 Deterministic LTS for an On-line Algorithm

For our analysis, on-line scheduling algorithms are represented as deterministic LTSs. Recall the definition of the sets $\Sigma = 2^{\mathcal{T}}$, and $\Pi = ((\mathcal{T} \times \{0, \ldots, D_{\max} - 1\}) \cup \varnothing)$. Every deterministic on-line algorithm $\mathcal{A}$ that uses finite state space (for all job sequences) can be represented as a deterministic LTS $L_{\mathcal{A}} = (S_{\mathcal{A}}, s_{\mathcal{A}}, \Sigma, \Pi, \Delta_{\mathcal{A}})$, where the states $S_{\mathcal{A}}$ correspond to the state space of $\mathcal{A}$, and $\Delta_{\mathcal{A}}$ correspond to the execution of $\mathcal{A}$ for one slot. Note that, due to relative indexing, for every current slot ℓ, the schedule π^ℓ of $\mathcal{A}$ contains elements from the set Π, and $(\tau_i, j) \in \pi^\ell$ uniquely determines the job $J_{i,\ell-j}$. Finally, we associate with $L_{\mathcal{A}}$ a reward function $r_{\mathcal{A}} : \Delta_{\mathcal{A}} \to \mathbb{N}$ such that $r_{\mathcal{A}}(\delta) = V_i$ if the transition δ completes a job of task τ_i, and $r_{\mathcal{A}}(\delta) = 0$ otherwise. Given the unique run $\rho^\sigma = (\delta^\ell)_{\ell \geqslant 1}$ of $L_{\mathcal{A}}$ for the job sequence σ, where δ^ℓ denotes the transition taken at the beginning of slot ℓ, the cumulated utility in the prefix of the first k transitions in ρ^σ is $V(\rho^\sigma, k) = \sum_{\ell=1}^{k} r_{\mathcal{A}}(\delta^\ell)$.

Most scheduling algorithms (such as EDF, FIFO, DOVER, TD1) can be represented as a deterministic LTS. An illustration for EDF is given in the following example.

Example 1. *Consider the taskset $\mathcal{T} = \{\tau_1, \tau_2\}$, with $D_1 = 3$, $D_2 = 2$ and $C_1 = C_2 = 2$. Figure 3.1 represents the EDF (Earliest Deadline First) scheduling policy as a deterministic LTS for $\mathcal{T}$. Each state is represented by a matrix M, such that $M[i, j]$, $1 \leqslant i \leqslant N$, $1 \leqslant j \leqslant D_{\max} - 1$,*

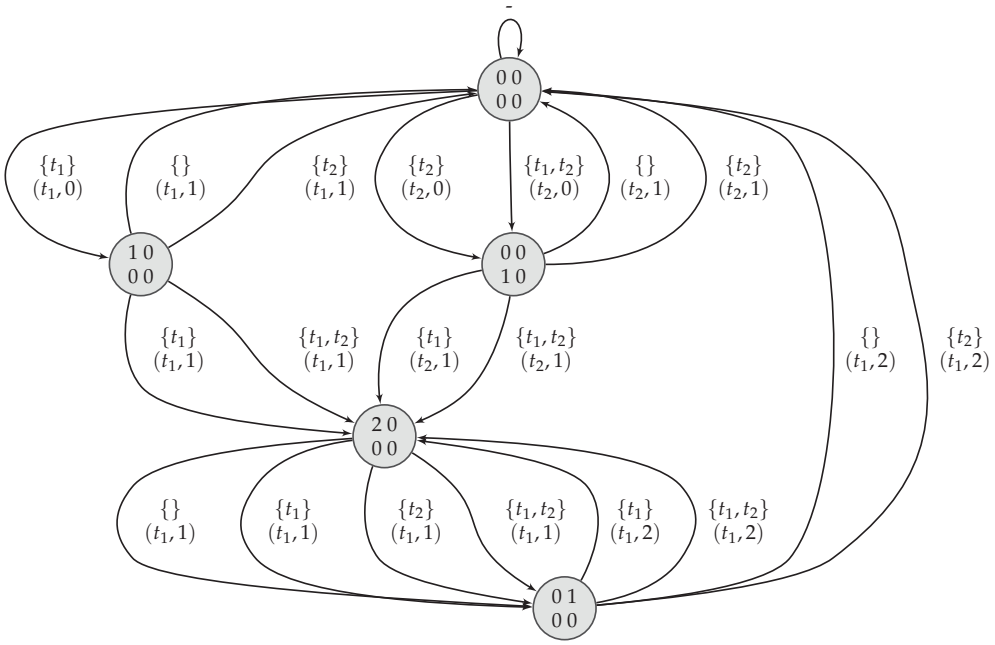


Figure 3.1: EDF for $\mathcal{T} = \{\tau_1, \tau_2\}$ with $D_1 = 3$, $D_2 = 2$ and $C_1 = C_2 = 2$, represented as a deterministic LTS.

denotes the remaining execution time of the job of task τ_i released j slots ago. Every transition is labeled with a set $T \in \Sigma$ of released tasks as well as with $(\tau_i, j) \in \Pi$, which denotes the unique job $J_{i,\ell-j}$ to be scheduled in the current slot ℓ. Released jobs with no chance of being scheduled are not included in the state space.

3.2.2 The Non-deterministic LTS

The clairvoyant algorithm $\mathcal{C}$ is formally a non-deterministic LTS $L_{\mathcal{C}} = (S_{\mathcal{C}}, s_{\mathcal{C}}, \Sigma, \Pi, \Delta_{\mathcal{C}})$ where each state in $S_{\mathcal{C}}$ is a $N \times (D_{\max} - 1)$ matrix M, such that for each time slot ℓ, the entry $M[i, j]$, $1 \leqslant i \leqslant N$, $1 \leqslant j \leqslant D_{\max} - 1$, denotes the remaining execution time of the job $J_{i,\ell-j}$ (i.e., the job of task i released j slots ago). For matrices M, M', subset $T \in \Sigma$ of newly released tasks, and scheduled job $P = (\tau_i, j) \in \Pi$, we have $(M, T, M', P) \in \Delta_{\mathcal{C}}$ iff $M[i, j] > 0$ and M' is obtained from M by

1. inserting all $\tau_i \in T$ into M,
2. decrementing the value at position $M[i, j]$, and
3. shifting the contents of M by one column to the right.

That is, M' corresponds to M after inserting all released tasks in the current state, executing a pending task for one unit of time, and reducing the relative deadlines

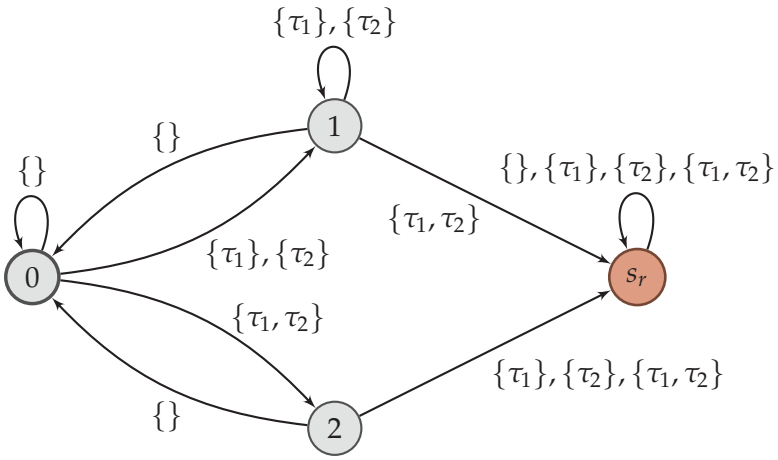


Figure 3.2: Example of a safety LTS $L_{\mathcal{S}}$ that restricts the adversary to release at most 2 units of workload in the last 2 rounds.

of all tasks currently in the system. The initial state $s_{\mathcal{C}}$ is represented by the zero $N \times (D_{\max} - 1)$ matrix, and $S_{\mathcal{C}}$ is the smallest $\Delta_{\mathcal{C}}$-closed set of states that contains $s_{\mathcal{C}}$ (i.e., if $M \in S_{\mathcal{C}}$ and $(M, T, M', P) \in \Delta_{\mathcal{C}}$ for some T, M' and P, we have $M' \in S_{\mathcal{C}}$). Finally, we associate with $L_{\mathcal{C}}$ a reward function $r_{\mathcal{C}} : \Delta_{\mathcal{C}} \to \mathbb{N}$ such that $r_{\mathcal{C}}(\delta) = V_i$ if the transition δ completes a task τ_i, and $r_{\mathcal{C}}(\delta) = 0$ otherwise.

3.3 Admissible Job Sequences

Our framework allows to restrict the adversary to generate admissible job sequences $\mathcal{J} \subseteq \Sigma^\infty$, which can be specified via different constraints. Since a constraint on job sequences can be interpreted as a language (which is a subset of infinite words Σ^∞ here), we will use automata as acceptors of such languages. Since an automaton is a deterministic LTS with no output, all our constraints will be described as LTSs with an empty set of output actions. We allow the following types of contraints:

($\mathcal{S}$) Safety constraints are defined by a deterministic LTS $L_{\mathcal{S}} = (S_{\mathcal{S}}, s_{\mathcal{S}}, \Sigma, \varnothing, \Delta_{\mathcal{S}})$, with a distinguished *absorbing* reject state $s_r \in S_{\mathcal{S}}$. An absorbing state is a state that has outgoing transitions only to itself. Every job sequence σ defines a unique run $\rho^\sigma_{\mathcal{S}}$ in $L_{\mathcal{S}}$, such that either no transition to s_r appears in $\rho^\sigma_{\mathcal{S}}$, or every such transition is followed solely by self-transitions to s_r. A job sequence σ is *admissible* to $L_{\mathcal{S}}$, if $\rho^\sigma_{\mathcal{S}}$ does not contain a transition to s_r. To obtain a safety LTS that does not restrict $\mathcal{J}$ at all, we simply use a trivial deterministic $L_{\mathcal{S}}$ with no transition to s_r.

Safety constraints restrict the adversary to release job sequences, where every finite prefix satisfies some property (as they lead to the absorbing reject state s_r of $L_{\mathcal{S}}$ otherwise). Some well-known examples of safety constraints are (i) periodicity and/or sporadicity constraints, where there are fixed and/or a minumum time

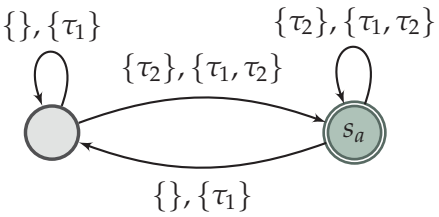


Figure 3.3: Example of a liveness LTS $L_{\mathcal{L}}$ that forces τ_2 to be released infinitely often.

between the release of any two consecutive jobs of a given task, and (ii) absolute workload constraints [Gol91, Cru91], where the total workload released in the last k slots, for some fixed k, is not allowed to exceed a threshold λ. For example, in case of absolute workload constraints, $L_{\mathcal{S}}$ simply encodes the workload in the last k slots in its state, and makes a transition to s_r whenever the workload exceeds λ. Figure 3.2 shows an example of a constraint LTS for the taskset $\mathcal{T} = \{\tau_1, \tau_2\}$ with $C_1 = C_2 = 1$ that restricts the adversary to release at most 2 units of workload in the last 2 rounds.

($\mathcal{L}$) Liveness constraints are modeled as a deterministic LTS $L_{\mathcal{L}} = (S_{\mathcal{L}}, s_{\mathcal{L}}, \Sigma, \varnothing, \Delta_{\mathcal{L}})$ with a distinguished *accept* state $s_a \in S_{\mathcal{L}}$. A job sequence σ is *admissible* to the liveness LTS $L_{\mathcal{L}}$ if $\rho^{\sigma}_{\mathcal{L}}$ contains infinitely many transitions to s_a. For the case where there are no liveness constraint in $\mathcal{J}$, we use a LTS $L_{\mathcal{L}}$ consisting of state s_a only.

Liveness constraints force the adversary to release job sequences that satisfy some property infinitely often. For example, they could be used to guarantee that the release of some particular task τ_i does not eventually stall; the constraint is specified by a two-state LTS $L_{\mathcal{L}}$ that visits s_a whenever the current job set includes τ_i. A liveness constraint can also be used to prohibit infinitely long periods of overload [BKM+92]. Figure 3.3 shows an example of a constraint LTS for the taskset $\mathcal{T} = \{\tau_1, \tau_2\}$ that forces the adversary to release τ_2 infinitely often.

($\mathcal{W}$) Limit-average constraints are defined by a deterministic weighted LTS $L_{\mathcal{W}} = (S_{\mathcal{W}}, s_{\mathcal{W}}, \Sigma, \varnothing, \Delta_{\mathcal{W}})$ equipped with a weight function $w : \Delta_{\mathcal{W}} \to \mathbb{Z}^d$ that assigns a vector of weights to every transition $\delta_{\mathcal{W}} \in \Delta_{\mathcal{W}}$. Given a threshold vector $\vec{\lambda} \in \mathbb{Q}^d$, where $\mathbb{Q}$ denotes the set of all rational numbers, a job sequence σ and the corresponding run $\rho^{\sigma}_{\mathcal{W}} = (\delta^{\ell}_{\mathcal{W}})_{\ell \geqslant 1}$ of $L_{\mathcal{W}}$, the job sequence is *admissible* to $L_{\mathcal{W}}$ if $\liminf_{k\to\infty} \frac{1}{k} \cdot w(\rho^{\sigma}_{\mathcal{W}}, k) \leqslant \vec{\lambda}$ with $w(\rho^{\sigma}_{\mathcal{W}}, k) = \sum_{i=1}^{k} w(\delta^{\ell}_{\mathcal{W}})$.

Consider a relaxed notion of workload constraints, where the adversary is restricted to generate job sequences whose *average* workload does not exceed a threshold λ. Since this constraint still allows "busy" intervals where the workload temporarily exceeds λ, it cannot be expressed as a safety constraint. To support such interesting average constraints of admissible job sequences, where the adversary is more relaxed than under absolute constraints, our framework explicitly supports limit-average constraints. Therefore, it is possible to express the average workload assumptions commonly used in the analysis of aperiodic

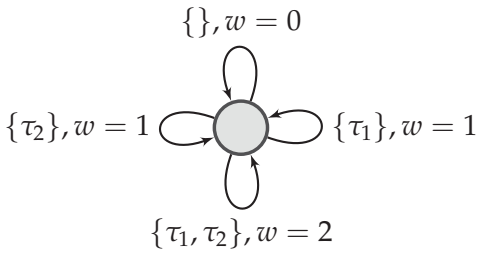


Figure 3.4: Example of a limit-average LTS $L_{\mathcal{W}}$ that tracks the average workload of jobs released by the adversary.

task scheduling in soft-real-time systems [AB98, HCL90]. Other interesting cases of limit-average constraints include restricting the average sporadicity, and, in particular, average energy: ensuring that the limit-average of the energy consumption is below a certain threshold is an important concern in modern real-time systems [AMMMA04]. Figure 3.4 shows an example of a constraint LTS for the taskset $\mathcal{T} = \{\tau_1, \tau_2\}$ with $C_1 = C_2 = 1$ that can be used to restrict the average workload the adversary is allowed to release in the long run.

Remark 2. While in general constraints are encoded as independent automata, it is often possible to encode certain constraints directly in the non-deterministic LTS of the clairvoyant scheduler instead. In particular, this is true when restricting the limit-average workload, generating finite intervals of overload, and releasing a particular job infinitely often.

3.4 Overall Approach for Computing $\mathcal{CR}$

Our goal is to determine the worst-case competitive ratio $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$ for a given on-line algorithm $\mathcal{A}$. The inputs to the problem are the given taskset $\mathcal{T}$, an on-line algorithm $\mathcal{A}$ specified as a deterministic LTS $L_{\mathcal{A}}$, and the safety, liveness, and limit-average constraints specified as deterministic LTSs $L_{\mathcal{S}}, L_{\mathcal{L}}$ and $L_{\mathcal{W}}$, respectively, which constrain the admissible job sequences $\mathcal{J}$. Our approach uses a reduction to a multi-objective graph problem, which consists of the following steps:

1. Construct a non-deterministic LTS $L_{\mathcal{C}}$ corresponding to the clairvoyant off-line algorithm $\mathcal{C}$. Note that since $L_{\mathcal{C}}$ is non-deterministic, for every admissible job sequence σ, there are many possible runs in $L_{\mathcal{C}}$, of course also including the runs with maximum cumulative utility.

2. Take the synchronous product LTS $L_{\mathcal{A}} \times L_{\mathcal{C}} \times L_{\mathcal{S}} \times L_{\mathcal{L}} \times L_{\mathcal{W}}$. By doing so, a path in the product graph corresponds to *identically* labeled paths in LTSs, and

thus ensures that they agree on the same job sequence σ. This product can be represented by a multi-objective graph (as introduced in Section 3.5).

3. Determine $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$ by reducing the computation of the ratio given in Equation (3.1) to solving a multi-objective problem on the product graph.

4. Finally, we employ several optimizations in order to reduce the size of product graph (see Sections 3.6 and 3.7).

3.5 Graphs with Multiple Objectives

In this section, we define various objectives on graphs and outline the algorithms to solve them. We later show how the competitive analysis of on-line schedulers reduces to the solution of this section.

Notation on Multi-Graphs

A *multi-graph* $G = (V, E)$, hereinafter called simply a *graph*, consists of a finite set V of *n nodes*, and a finite set of *m directed multiple edges* $E \subset V \times V \times \mathbb{N}^+$. For brevity, we will refer to an edge (u, v, i) as (u, v), when i is not relevant. We consider graphs in which for all $u \in V$, we have $(u, v) \in E$ for some $v \in V$, i.e., every node has at least one outgoing edge. An *infinite path* ρ of G is an infinite sequence of edges $e^1, e^2, \ldots$ such that for all $i \geqslant 1$ with $e^i = (u^i, v^i)$, we have $v^i = u^{i+1}$. Every such path ρ induces a sequence of nodes $(u^i)_{i \geqslant 1}$, which we will also call a path, when the distinction is clear from the context, and ρ^i refers to u^i instead of e^i. Finally, we denote with ρ^∞ the set of all paths of G.

3.5.1 Objectives

Given a graph G, an objective Φ is a subset of ρ^∞ that defines the desired set of paths. We will consider safety, liveness, mean-payoff (limit-average), and ratio objectives, and their conjunction for multiple objectives.

Safety and liveness objectives: We consider safety and liveness objectives, both defined with respect to some subset of nodes $X, Y \subseteq V$. Given $X \subseteq V$, the *safety* objective defined as $\mathsf{Safe}(X) = \{\rho \in \rho^\infty : \forall i \geqslant 1, \rho^i \notin X\}$, represents the set of all paths that never visit the set X. The *liveness* objective defined as $\mathsf{Live}(Y) = \{\rho \in \rho^\infty : \forall j \exists i > j \text{ s.t. } \rho^i \in Y\}$ represents the set of all paths that visit Y infinitely often.

Mean-payoff and ratio objectives: We consider the mean-payoff and ratio objectives, defined with respect to a weight function and a threshold. A *weight function*

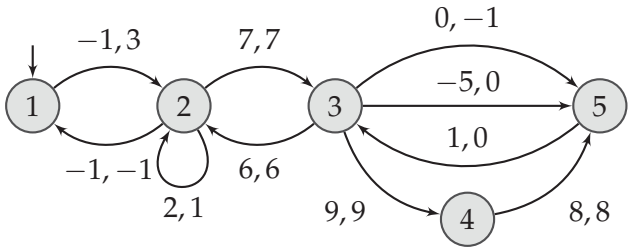


Figure 3.5: An example of a multi-graph G.

$w : E \to \mathbb{Z}^d$ assigns to each edge of G a vector of d integers. A weight function naturally extends to paths, with $w(\rho, k) = \sum_{i=1}^{k} w(\rho^i)$. The *mean-payoff* of a path ρ is defined as:

$$\mathsf{MP}(w, \rho) = \liminf_{k \to \infty} \frac{1}{k} \cdot w(\rho, k);$$

i.e., it is the long-run average of the weights of the path. Given a weight function w and a threshold vector $\vec{v} \in \mathbb{Q}^d$, the corresponding objective is given as:

$$\mathsf{MP}(w, \vec{v}) = \{\rho \in \rho^\infty : \mathsf{MP}(w, \rho) \leqslant \vec{v}\};$$

that is, the set of all paths such that the mean-payoff (or limit-average) of their weights is at most $\vec{v}$ (where we consider pointwise comparision for vectors). For weight functions $w_1, w_2 : E \to \mathbb{N}^d$, the *ratio* of a path ρ is defined as:

$$\mathsf{Ratio}(w_1, w_2, \rho) = \liminf_{k \to \infty} \frac{\vec{\mathbf{1}} + w_1(\rho, k)}{\vec{\mathbf{1}} + w_2(\rho, k)},$$

which denotes the limit infimum of the coordinate-wise ratio of the sum of weights of the two functions; $\vec{\mathbf{1}}$ denotes the d-dimensional all-1 vector. Given weight functions w_1, w_2 and a threshold vector $\vec{v} \in \mathbb{Q}^d$, the ratio objective is given as:

$$\mathsf{Ratio}(w_1, w_2, \vec{v}) = \{\rho \in \rho^\infty : \mathsf{Ratio}(w_1, w_2, \rho) \leqslant \vec{v}\}$$

that is, the set of all paths such that the ratio of cumulative rewards w.r.t w_1 and w_2 is at most $\vec{v}$.

Example 2. *Consider the multi-graph shown in Figure 3.5 with a weight function of dimension $d = 2$. Note that there are two edges from node 3 to node 5 (represented as edges $(3, 5, 1)$ and $(3, 5, 2)$). In the graph we have a weight function with dimension 2. Note that the two edges from node 3 to node 5 have incomparable weight vectors.*

3.5.2 Decision Problem

The decision problem we consider is as follows: Given the graph G, an initial node $s \in V$, and an objective Φ (which can be a conjunction of several objectives), determine if there exists a path ρ that starts from s and belongs to Φ, i.e., $\rho \in \Phi$. For simplicity of presentation, we assume that every $u \in V$ is reachable from s (unreachable nodes can be discarded by preprocessing G in $\mathcal{O}(m)$ time). We first present algorithms for each of safety, liveness, mean-payoff, and ratio obejctives separately, and then for their conjunction.

Algorithms for safety objectives

The algorithm for the objective $\mathsf{Safe}(X)$ is straightforward. We first remove the set X of nodes, and iteratively remove nodes without outgoing edges. In the end, we obtain a graph $G = (V_X, E_X)$ such that $X \cap V_X = \varnothing$, and every node in V_X has an edge to a node in V_X. Thus, in the resulting graph, the objective $\mathsf{Safe}(X)$ is satisfied, and the algorithm answers yes iff $s \in V_X$. The algorithm requires $\mathcal{O}(m)$ time.

Algorithms for liveness objectives

To solve for the objective $\mathsf{Live}(Y)$, initially perform an SCC (maximal strongly connected component) decomposition of G. We call an SCC V_{SCC} *live*, if (i) either $|V_{\mathrm{SCC}}| > 1$, or $V_{\mathrm{SCC}} = \{u\}$ and $(u, u) \in E$; and (ii) $V_{\mathrm{SCC}} \cap Y \neq \varnothing$. Then $\mathsf{Live}(Y)$ is satisfied in G iff there exists a live SCC V_{SCC} that is reachable from s (since every node in a live SCC can be visited infinitely often). Using for example the algorithm of [Tar72] for the SCC decomposition also requires $\mathcal{O}(m)$ time.

Algorithms for mean-payoff objectives

We distinguish between the case when the weight function has a single dimension ($d = 1$) versus the case when the weight function has multiple dimensions ($d > 1$).

Single dimension: In the case of a single-dimensional weight function, a single weight is assigned to every edge, and the decision problem of the mean-payoff objective reduces to determining the mean weight of a minimum-weight simple cycle in G, as the latter also determines the mean-weight by infinite repetition. Using the algorithms of [Kar78, Mad02], this process requires $\mathcal{O}(n \cdot m)$ time. When the objective is satisfied, the process also returns a simple cycle C, as a witness to the objective. From C, a path $\rho \in \mathsf{MP}(w, \vec{v})$ is constructed by infinite repetitions of C.

Multiple dimensions: When $d > 1$, the mean-payoff objective reduces to determining the feasibility of a linear program (LP). For $u \in V$, let $\mathsf{IN}(u)$ be the set of incoming, and $\mathsf{OUT}(u)$ the set of outgoing edges of u. As shown in [VCD+12], G satisfies

$\mathsf{MP}(w, \vec{v})$ iff the following set of constraints on $\vec{x} = (x_e)_{e \in E_{\text{SCC}}}$ with $x_e \in \mathbb{Q}$ is satisfied simultaneously on some SCC V_{SCC} of G with induced edges $E_{\text{SCC}} \subseteq E$.

$$\begin{aligned} & x_e \geqslant 0 && e \in E_{\text{SCC}} \\ & \sum_{e \in \mathsf{IN}(u)} x_e = \sum_{e \in \mathsf{OUT}(u)} x_e && u \in V_{\text{SCC}} \\ & \sum_{e \in E_{\text{SCC}}} x_e \cdot w(e) \leqslant \vec{v} \\ & \sum_{e \in E_{\text{SCC}}} x_e \geqslant 1 \end{aligned} \tag{3.2}$$

The quantities x_e are intuitively interpreted as "flows". The first constraint specifies that the flow of each edge is non-negative. The second constraint is a flow-conservation constraint. The third constraint specifies that the objective is satisfied if we consider the relative contribution of the weight of each edge, according to the flow of the edge. The last constraint asks that the preceding constraints are satisfied by a non-trivial (positive) flow. Hence, when $d > 1$, the decision problem reduces to solving a LP, and the time complexity is polynomial [Kha79].

Witness construction: The witness path construction from a feasible solution consists of two steps:

1. Construction of a multi-cycle from the feasible solution; and
2. Construction of an infinite witness path from the multi-cycle.

We describe the two steps in detail. Formally, a *multi-cycle* is a finite set of cycles with multiplicity $\mathcal{MC} = \{(C_1, m_1), (C_2, m_2), \ldots, (C_k, m_k)\}$, such that every C_i is a simple cycle and m_i is its multiplicity. The construction of a multi-cycle from a feasible solution $\vec{x}$ is as follows. Let $\mathcal{E} = \{e : x_e > 0\}$. By scaling each edge flow x_e by a common factor z, we construct the set $\mathcal{X} = \{(e, z \cdot x_e) : e \in \mathcal{E}\}$, with $\mathcal{X} \subset E_{\text{SCC}} \times \mathbb{N}^+$. Then, we start with $\mathcal{MC} = \varnothing$ and apply iteratively the following procedure until $\mathcal{X} = \varnothing$:

(i) find a pair $(e_i, m_i) = \arg\min_{(e_j, m_j) \in \mathcal{X}} m_j$,

(ii) form a cycle C_i that contains e_i and only edges that appear in $\mathcal{X}$ (because of Equation (3.2), this is always possible),

(iii) add the pair (C_i, m_i) in the multi-cycle $\mathcal{MC}$,

(iv) subtract m_i from all elements (e_j, m_j) of $\mathcal{X}$ such that the edge e_j appears in C_i,

(v) remove from $\mathcal{X}$ all $(e_j, 0)$ pairs, and repeat.

Since V_{SCC} is an SCC, there is a path $C_i \rightsquigarrow C_j$ for all C_i, C_j in $\mathcal{MC}$. Given the multi-cycle $\mathcal{MC}$, the infinite path that achieves the weight at most $\vec{v}$ is not periodic, but generated by Algorithm 6.

Algorithm 6: Multi-objective witness

Input: A graph $G = (V, E)$, and a multi-cycle $\mathcal{MC} = \{(C_1, m_1), (C_2, m_2), \ldots, (C_k, m_k)\}$
Output: An infinite path $\rho \in \mathsf{MP}(w, \vec{v})$

$\ell \leftarrow 1$
while *True* **do**
    Repeat C_1 for $\ell \cdot m_1$ times
    $C_1 \rightsquigarrow C_2$
    Repeat C_2 for $\ell \cdot m_2$ times
    ...
    Repeat C_k for $\ell \cdot m_k$ times
    $C_k \rightsquigarrow C_1$
    $\ell \leftarrow \ell + 1$
end

Algorithms for ratio objectives

We now consider ratio objectives, and present a reduction to mean-payoff objectives. Consider the weight functions w_1, w_2 and the threshold vector $\vec{v} = \frac{\vec{p}}{\vec{q}}$ as the component-wise division of vectors $\vec{p}, \vec{q} \in \mathbb{N}^d$. We define a new weight function $w : E \to \mathbb{Z}^d$ such that for all $e \in E$, we have $w(e) = \vec{q} \cdot w_1(e) - \vec{p} \cdot w_2(e)$ (where $\cdot$ denotes component-wise multiplication). It is easy to verify that $\mathsf{Ratio}(w_1, w_2, \vec{v}) = \mathsf{MP}(w, \vec{0})$, and thus we solve the ratio objective by solving the new mean-payoff objective, as described above.

Algorithms for conjunctions of objectives

Finally, we consider the conjunction of a safety, a liveness, and a mean-payoff objective (note that we have already described a reduction of ratio objectives to mean-payoff objectives). More specifically, given a weight function w, a threshold vector $\vec{v} \in \mathbb{Q}$, and sets $X, Y \subseteq V$, we consider the decision problem for the objective $\Phi = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{MP}(w, \vec{v})$. The procedure is as follows:

1. Initially compute G_X from G as in the case of a single safety objective.
2. Then, perform an SCC decomposition on G_X.
3. For every live SCC V_{SCC} that is reachable from s, solve for the mean-payoff objective in V_{SCC}. Return yes, if $\mathsf{MP}(w, \vec{v})$ is satisfied in any such V_{SCC}.

If the answer to the decision problem is yes, then the witness consists of a live SCC V_{SCC}, along with a multi-cycle (resp. a cycle for $d = 1$). The witness infinite path is constructed as in Algorithm 6, with the only difference that at end of each while loop a live node from Y in the SCC V_{SCC} is additionally visited. The time required for the conjunction of objectives is dominated by the time required to solve for the mean-payoff objective. Figure 3.5 provides a relevant example.

Example 3. *Consider the graph in Figure 3.5. Starting from node* 1*, the mean-payoff-objective* $\mathsf{MP}(w, \vec{0})$ *is satisfied by the multi-cycle* $\mathcal{MC} = \{(C_1, 1), (C_2, 2)\}$*, with* $C_1 = ((1,2),(2,1))$ *and* $C_2 = ((3,5),(5,3))$*. A solution to the corresponding LP is* $x_{(1,2)} = x_{(2,1)} = \frac{1}{3}$ *and* $x_{(3,5)} = x_{(5,3)} = \frac{2}{3}$*, and* $x_e = 0$ *for all other* $e \in E$*. Procedure 6 then generates a witness path for the objective. The objective is also satisfied in conjuction with* $\mathsf{Safe}(\{4\})$ *or* $\mathsf{Live}(\{4\})$*. In the latter case, a witness path additionally traverses the edges* $(3,4)$ *and* $(4,5)$ *before transitioning from* C_1 *to* C_2*.*

Theorem 3 summarizes the results of this section.

Theorem 3. *Let* $G = (V, E)$ *be a graph,* $s \in V$*,* $X, Y \subseteq V$*,* $w : E \to \mathbb{Z}^d$*,* w_1*,* w_2 $E \to \mathbb{N}^d$ *weight functions, and* $\vec{v} \in \mathbb{Q}^d$*. Let* $\Phi_1 = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{MP}(w, \vec{v})$ *and* $\Phi_2 = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{Ratio}(w_1, w_2, \vec{v})$*. The decision problem of whether G satisfies the objective* Φ_1 *(resp.* Φ_2*) from s requires*

1. $\mathcal{O}(n \cdot m)$ *time, if* $d = 1$*.*
2. *Polynomial time, if* $d > 1$*.*

If the objective Φ_1 *(resp.* Φ_2*) is satisfied in G from s, then a finite witness (an SCC and a cycle for single dimension, and an SCC and a multi-cycle for multiple dimensions) exists and can be constructed in polynomial time.*

3.6 Reduction to Multi-Objective Graphs

We present a formal reduction of the computation of the competititve ratio of an on-line scheduling algorithm with constraints on job sequences to the multi-objective graph problem. The input consists of the taskset, a deterministic LTS for the on-line algorithm, and optional deterministic LTSs for the constraints. We first describe the process of computing the competitive ratio $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$ where $\mathcal{J}$ is a set of job sequences only subject to safety and liveness constraints. We later show how to handle limit-average constraints.

3.6.1 Reduction for Safety and Liveness Constraints

Given the deterministic and non-deterministic LTS $L_{\mathcal{A}}$ and $L_{\mathcal{C}}$ with reward functions $r_{\mathcal{A}}$ and $r_{\mathcal{C}}$, respectively, and optionally safety and liveness LTS $L_{\mathcal{S}}$ and $L_{\mathcal{L}}$, let $L = L_{\mathcal{A}} \times L_{\mathcal{C}} \times L_{\mathcal{S}} \times L_{\mathcal{L}}$ be their synchronous product (refer to the end of this section for the formal definition of the synchronous product). Hence, L is a non-deterministic LTS $(S, s_1, \Sigma, \Pi, \Delta)$, and every job sequence σ yields a set of runs R in L, such that each $\rho \in R$ captures the joint behavior of $\mathcal{A}$ and $\mathcal{C}$ under σ. Note that for each such ρ the behavior of $\mathcal{A}$ is unchanged, but the behavior of $\mathcal{C}$ generally varies, due to non-determinism. Let $G = (V, E)$ be the multi-graph induced by L, that is, $V = S$ and

$(M, M', j) \in E$ for all $1 \leqslant j \leqslant i$ iff there are i transitions $(M, T, M', P) \in \Delta$. Let $w_{\mathcal{A}}$ and $w_{\mathcal{C}}$ be the weight functions that assign to each edge of G the reward that the respective algorithm obtains from the corresponding transition in L. Let X be the set of states in G whose $L_{\mathcal{S}}$ component is s_r, and Y the set of states in G whose $L_{\mathcal{L}}$ component is s_a. It follows that for all $\nu \in \mathbb{Q}$, we have that $\mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant \nu$ iff the objective $\Phi_\nu = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{Ratio}(w_{\mathcal{A}}, w_{\mathcal{C}}, \nu)$ is satisfied in G from the state s_1. As the dimension in the ratio objective is one, Case 1 of Theorem 3 applies, and we obtain the following:

Lemma 1. *Given the product graph $G = (V, E)$ of n nodes and m edges, a rational $\nu \in \mathbb{Q}$, and a set of job sequences $\mathcal{J}$ admissible to safety and liveness LTSs, determining whether $\mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant \nu$ requires $\mathcal{O}(n \cdot m)$ time.*

Since $0 \leqslant \mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant 1$, the problem of determining the competitive ratio reduces to finding $v = \sup\{\nu \in \mathbb{Q} : \ \Phi_\nu \text{ is satisfied in } G\}$. Because this value corresponds to the ratio of the corresponding rewards obtained in a simple cycle in G, it follows that v is the maximum of a finite set, and can be determined exactly by an *adaptive binary search* (the algorithm used for the adaptive binary search is explained at the end of this section).

Synchronous product of LTSs

The *synchronous product* of two LTSs $L_1 = (S_1, s_1, \Sigma, \Pi, \Delta_1)$ and $L_2 = (S_2, s_2, \Sigma, \Pi, \Delta_2)$ is an LTS $L = (S, s, \Sigma, \Pi', \Delta)$ such that:

1. $S \subseteq S_1 \times S_2$,
2. $s = (s_1, s_2)$,
3. $\Pi' = \Pi \times \Pi$, and
4. $\Delta \subseteq S \times \Sigma \times S \times \Pi'$ such that $((q_1, q_2), T, (q_1', q_2'), (P_1, P_2)) \in \Delta$ iff $(q_1, T, q_1', P_1) \in \Delta_1$ and $(q_2, T, q_2', P_2) \in \Delta_2$.

The set of states S is the smallest Δ-closed subset of $S_1 \times S_2$ that contains s (i.e., $s \in S$, and for each $q \in S$, if there exist $q' \in S_1 \times S_2$, $T \in \Sigma$ and $P \in \Pi'$ such that $(q, T, q', P) \in \Delta$, then $q' \in S$). That is, the synchronous product of L_1 with L_2 captures the joint behavior of L_1 and L_2 in every input sequence $\sigma \in \Sigma^\infty$ (L_1 and L_2 synchronize on input actions). Note that if both L_1 and L_2 are deterministic, so is there synchronous product. The synchronous product of $k > 2$ LTSs $L_1, \ldots, L_k$ is defined iteratively as the synchronous product of L_1 with the synchronous product of $L_2, \ldots, L_k$.

Adaptive Binary Search

Algorithm AdaptiveBinarySearch (Algorithm 7) implements an *adaptive* binary search for the competitive ratio in the interval $[0, 1]$. The algorithm maintains an interval $[\ell, r]$ such

that $\ell \leqslant \mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant r$ at all times, and exploits the nature of the problem for refining the interval as follows: First, if the current objective $\nu \in [\ell, r]$ (typically, $\nu = (\ell + r)/2$) is satisfied in G, i.e., Lemma 1 answers "yes", and provides the current minimum cycle C as a witness, the value r is updated to the ratio ν' of the on-line and off-line rewards in C, which is typically less than ν. This allows to reduce the current interval for the next iteration from $[\ell, r]$ to $[\ell, \nu']$, with $\nu' \leqslant \nu$, rather than $[\ell, \nu]$ (as a simple binary search would do). Second, since $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$ corresponds to the ratio of rewards on a simple cycle in G, if the current objective $\nu \in [\ell, r]$ is not satisfied in G, the algorithm assumes that $\mathcal{CR}_{\mathcal{J}}(\mathcal{A}) = r$ (i.e, the competitive ratio equals the right endpoint of the current interval), and tries $\nu = r$ in the next iteration. Hence, as opposed to a naive binary search, the adaptive version has the advantages of (i) returning the exact value of $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$ (rather than an approximation), and (ii) being faster.

Algorithm 7: AdaptiveBinarySearch

Input: Graph $G = (V, E)$ and weight functions $w_{\mathcal{A}}$, $w_{\mathcal{C}}$
Output: $\min_{C \in G} \frac{w_{\mathcal{A}}(C)}{w_{\mathcal{C}}(C)}$

$\ell \leftarrow 0, r \leftarrow 1, \nu \leftarrow \frac{(\ell+r)}{2}$
while *True* **do**
  Solve G for obj. Φ_ν and find min simple cycle C
  $\nu_1 \leftarrow w_{\mathcal{A}}(C), \nu_2 \leftarrow w_{\mathcal{C}}(C)$
  if $\nu = \frac{\nu_1}{\nu_2}$ **then**
    return ν
  else
    if $\nu > \frac{\nu_1}{\nu_2}$ **then**
      $r \leftarrow \frac{\nu_1}{\nu_2}, \nu \leftarrow \frac{(\ell+r)}{2}$
    else
      $\ell \leftarrow \nu, r \leftarrow \min\left(\frac{\nu_1}{\nu_2}, r\right), \nu \leftarrow r$
    end
  end
end

3.6.2 Reduction for Limit-Average Constraints

Finally, we turn our attention to limit-average constraints and the LTS $L_{\mathcal{W}}$. We follow a similar approach as above, but this time including $L_{\mathcal{W}}$ in the synchronous product, i.e., $L = L_{\mathcal{A}} \times L_{\mathcal{C}} \times L_{\mathcal{S}} \times L_{\mathcal{L}} \times L_{\mathcal{W}}$. Let $w_{\mathcal{A}}$ and $w_{\mathcal{C}}$ be weight functions that assign to each edge $e \in E$ in the corresponding multi-graph a vector of $d + 1$ weights as follows. In the first dimension, $w_{\mathcal{A}}$ and $w_{\mathcal{C}}$ are defined as before, assigning to each edge of G the corresponding rewards of $\mathcal{A}$ and $\mathcal{C}$. In the remaining d dimensions, $w_{\mathcal{C}}$ is always 1, whereas $w_{\mathcal{A}}$ equals the value of the weight function w of $L_{\mathcal{W}}$ on the corresponding transition. Let $\vec{\lambda}$ be the threshold vector of $L_{\mathcal{W}}$. It follows that for all $\nu \in \mathbb{Q}$, we have that $\mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant \nu$ iff the objective $\Phi_\nu = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{Ratio}(w_{\mathcal{A}}, w_{\mathcal{C}}, (\nu, \vec{\lambda}))$ is satisfied

in G from the state s that corresponds to the initial state of each LTS, where $(\nu, \vec{\lambda})$ is a $d+1$-dimension vector, with ν in the first dimension, followed by the d-dimension vector $\vec{\lambda}$. As the dimension in the ratio objective is greater than one, Case 2 of Theorem 3 applies, and we obtain the following:

Lemma 2. *Given the product graph $G = (V, E)$ of n nodes and m edges, a rational $\nu \in \mathbb{Q}$, and a set of job sequences $\mathcal{J}$ admissible to safety, liveness, and limit average LTSs, determining whether $\mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant \nu$ requires polynomial time.*

Again, since $0 \leqslant \mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant 1$, the competitive ratio is determined by an adaptive binary search similar to Algorithm 7. However, this time $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$ is not guaranteed to be realized by a simple cycle (the witness path in G is not necessarily periodic, see Algorithm 6), and is only approximated within some desired error threshold $\varepsilon > 0$.

3.7 Optimized Reduction

In Section 3.6, we have established a formal reduction from determining the competitive ratio of an on-line scheduling algorithm in a constrained adversarial environment to solving multiple objectives on graphs. In the current section, we present several optimizations in this reduction that significantly reduce the size of the generated LTSs.

3.7.1 Clairvoyant LTS

Recall the clairvoyant LTS $L_{\mathcal{C}}$ with reward function $r_{\mathcal{C}}$ from Section 3.2 that non-deterministically models a scheduler. Now we encode the off-line algorithm as a non-deterministic LTS $L'_{\mathcal{C}} = (S'_{\mathcal{C}}, s'_{\mathcal{C}}, \Sigma, \varnothing, \Delta'_{\mathcal{C}})$ with reward function $r'_{\mathcal{C}}$ that lacks the property of being a scheduler, as information about released and scheduled jobs is lost. However, it preserves the property that, given a job sequence σ, there exists a run $\rho^{\sigma}_{\mathcal{C}}$ in $L_{\mathcal{C}}$ iff there exists a run $\widehat{\rho}^{\sigma}_{\mathcal{C}}$ in $L'_{\mathcal{C}}$ with $V(\rho^{\sigma}_{\mathcal{A}}, k) = V(\widehat{\rho}^{\sigma}_{\mathcal{A}}, k)$ for all $k \in \mathbb{N}^{+}$. That is, there is a bisimulation between $L_{\mathcal{C}}$ and $L'_{\mathcal{C}}$ that preserves rewards.

Intuitively, the clairvoyant algorithm need not partially schedule a task, i.e., it will either discard it immediately, or schedule it to completion. Hence, in every release of a set of tasks T, $L'_{\mathcal{C}}$ non-deterministically chooses a subset $T' \subseteq T$ to be scheduled, as well as allocates the future slots for their execution. Once these slots are allocated, $L'_{\mathcal{C}}$ is not allowed to preempt those in favor of a subsequent job.

The state space $S'_{\mathcal{C}}$ of $L'_{\mathcal{C}}$ consists of binary strings of length $D_{\max}$. For a binary string $B \in S'_{\mathcal{C}}$, we have $B[i] = 1$ iff the i-th slot in the future is allocated to some released job, and $s'_{\mathcal{C}} = \vec{\mathbf{0}}$. Informally, the transition relation $\Delta'_{\mathcal{C}}$ is such that, given a current subset $T \subseteq \Sigma$ of released jobs, there exists a transition δ from B to B' only if B' can be obtained from B by non-deterministically choosing a subset $T' \subseteq T$, and for each task $\tau_i \in T'$ allocating non-deterministically C_i free slots in B. Finally, set $r'_{\mathcal{C}} = \sum_{\tau_i \in T'} V_i$.

By definition, $|S'_{\mathcal{C}}| \leqslant 2^{D_{\max}}$. In laxity-restricted tasksets, we can obtain an even tighter bound. Let $L_{\max} = \max_{\tau_i \in \mathcal{T}} (D_i - C_i)$ be the maximum laxity in $\mathcal{T}$, and I :

$S'_{\mathcal{C}} \to \{\perp, 1, \ldots, D_{\max} - 1\}^{L_{\max}+1}$ a function such that $I(B) = (i_1, \ldots, i_{L_{\max}+1})$ are the indexes of the first $L_{\max} + 1$ zeros in B. That is, $i_j = k$ iff $B[k]$ is the j-th zero location in B, and $i_j = \perp$ if there are less than j free slots in B.

Claim 1. *The function I is bijective.*

Proof. Fix a tuple $(i_1, \ldots, i_{L_{\max}+1})$ with $i_j \in \{\perp, 1, \ldots, D_{\max} - 1\}$, and let $B \in S'_{\mathcal{C}}$ be any state such that $I(B) = (i_1, \ldots, i_{L_{\max}+1})$. We consider two cases.

1. If $i_{L_{\max}+1} = \perp$, there are less than $L_{\max} + 1$ empty slots in B, all uniquely determined by $(i_1, \ldots, i_k)$, for some $k \leqslant L_{\max}$.

2. If $i_{L_{\max}+1} \neq \perp$, then all $i_j \neq \perp$, and thus any job to the right of $i_{L_{\max}+1}$ would have been stalled for more than $L_{\max}$ positions. Hence, all slots to the right of $i_{L_{\max}+1}$ are free in B, and B is also unique.

Hence, $I(B)$ always uniquely determines B, as desired. ■

For $x, k \in \mathbb{N}^+$, denote with $\mathsf{Perm}(x, k) = x \cdot (x-1) \ldots (x-k+1)$ the number of k-permutations on a set of size x.

Lemma 3. *Let $\mathcal{T}$ be a taskset with maximum deadline $D_{\max}$, and $L_{\max} = \max_{\tau_i \in \mathcal{T}} (D_i - C_i)$ be the maximum laxity. Then, $|S'_{\mathcal{C}}| \leqslant \min(2^{D_{\max}}, \mathsf{Perm}(D_{\max}, L_{\max} + 1))$.*

Hence, for zero and small laxity environments [BKM+92], as e.g. arising in wormhole switching in NoCs [LJ07], $S'_{\mathcal{C}}$ has polynomial size in $D_{\max}$ (also see Section 3.8.4 for zero-laxity tasksets with large $D_{\max}$).

3.7.2 Clairvoyant LTS Generation

We now turn our attention on efficiently generating the clairvoyant LTS $L'_{\mathcal{C}}$ as described in the previous paragraph. There is non-determinism in two steps: both in choosing the subset $T' \subseteq T$ of the currently released tasks for execution, and in allocating slots for executing all tasks in T'. Given a current state B and T, this non-determinism leads to several identical transitions δ to a state B'. We have developed a recursive algorithm called ClairvoyantSuccessor (Algorithm 8) that generates each such transition δ exactly once.

The intuition behind ClairvoyantSuccessor is as follows. It has been shown that the earliest deadline first (EDF) policy is optimal in scheduling job sequences where every released task can be completed [Der74]. By construction, given a job sequence σ_1, $L'_{\mathcal{C}}$ non-deterministically chooses a job sequence σ_2, such that for all ℓ, we have $\sigma_2^\ell \subseteq \sigma_1^\ell$, and all jobs in σ_2 are scheduled to completion by $L'_{\mathcal{C}}$. Therefore, it suffices to consider a transition relation $\Delta'_{\mathcal{C}}$ that allows at least all possible choices that admit a feasible EDF schedule on every possible σ_2, for any generated job sequence σ_1.

Algorithm 8: ClairvoyantSuccessor

Input: A set $T \subseteq \mathcal{T}$, state B, index $1 \leqslant k \leqslant D_{\max}$
Output: A set $\mathcal{B}$ of successor states of B

```
1  if T = ∅ then return {B};
2  τ ← arg min_{τ_i ∈ T} D_i, C ← execution time of τ
3  T' ← T \ {τ}
   // Case 1: τ is not scheduled
4  B ← ClairvoyantSuccessor(T', B, k)
   // Case 2: τ is scheduled
5  F ← set of free slots in B greater than k
6  foreach F ⊆ F with |F| = C do
7      B' ← Allocate F in B
8      k' ← rightmost slot in F
9      B' ← ClairvoyantSuccessor(T', B', k')
       // Keep only non-redundant states
10     foreach B'' ∈ B' do
11         if B''[1] = 1 and knapsack(B'', T) then
12             B ← B ∪ {B''}
13         end
14     end
15 end
16 return B
```

In more detail, ClairvoyantSuccessor is called with a current state B, a subset of released tasks T and an index k, and returns the set $\mathcal{B}$ of all possible successors of B that schedule a subset $T' \subseteq T$, and every job of T' is executed later than k slots in the future. This is done by extracting from T the task τ with the earliest deadline, and proceeding as follows: The set $\mathcal{B}$ is obtained by constructing a state B' that considers all the possible ways to schedule τ to the right of k (including the possibility of not scheduling τ at all), and recursively finding all the ways to schedule $T \setminus \{\tau\}$ in B', to the right of the rightmost slot allocated for task τ.

Finally, we exploit the following two observations to further reduce the state space of $L'_{\mathcal{C}}$. First, we note that as long as there is some load in the state of $L'_{\mathcal{C}}$ (i.e., at least one bit of B is one), the clairvoyant algorithm gains no benefit by not executing any job in the current slot. Hence, besides the zero state $\vec{\mathbf{0}}$, every state B must have $B[1] = 1$. In most cases, this restriction reduces the state space by at least 50%. Second, it follows from our claims on the off-line EDF policy of the clairvoyant scheduler that for every two scheduled jobs J and J', it will never have to preempt J for J' and vice versa. A consequence of this is that, for every state B and every continuous segment of zeros in B that is surrounded by ones (called a *gap*), the gap must be able to be completely filled with some jobs that start and end inside the gap. This reduces to solving a knapsack problem [Kar72] where the size of the knapsack is the length of the gap, and the set of items is the whole taskset $\mathcal{T}$ (with multiplicities). We note that the problem has to be

Taskset	A1				A2		A3			A4			A5			A6		
	τ_1	τ_2	τ_3	τ_4	τ_1	τ_2	τ_1	τ_2	τ_3	τ_1	τ_2	τ_3	τ_1	τ_2	τ_3	τ_1	τ_2	τ_3
C_i	1	4	1	3	2	2	2	1	1	1	2	1	2	6	1	1	2	1
D_i	2	6	3	4	3	2	2	5	5	2	3	6	2	6	1	5	2	1
V_i	3	2	3	3	5	1	1	2	2	3	2	1	1	10	2	5	4	1

Table 3.1: The tasksets used to generate Figure 3.6.

solved on identical inputs a large number of times, and techniques such as *memoization* are employed to avoid multiple evaluations of the same input.

These two improvements were found to reduce the state space by a factor up to 90% in all examined cases (see Section 3.8 and Table 3.4), and despite the non-determinism, in all reported cases the generation of L_C was done in less than a second.

3.7.3 On-line State Space Reduction

Typically, most on-line scheduling algorithms do "lazy dropping" of the jobs, where a job is dropped only when its deadline passes. To keep the state space of the LTS small, it is crucial to only store those jobs that have the possibility of being scheduled, at least partially, under some sequence of future task releases. We do so by first creating the LTS naively, and then iterating through its states. For each state s and job $J_{i,j}$ in s with relative deadline D_i, we perform a *depth-limited search* originating in s for D_i steps, looking for a state s' reached by a transition that schedules $J_{i,j}$. If no such state is found, we merge state s to s'', where s'' is identical to s without job $J_{i,j}$.

3.8 Experimental Results

We have implemented our approach for automated competitive ratio analysis, and applied it to a range of case studies: four well-known scheduling policies, namely, EDF (Earliest Deadline First), SRT (Shortest Remaining Time), SP (Static Priorities), and FIFO (First-in First-out), as well as some more elaborate algorithms that provide non-trivial performance guarantees, in particular, DSTAR [BKM+91], and DOVER [KS95], are analyzed under a variety of tasksets. For the scheduling algorithm TD1 [BKM+92], we constructed a series of task sets according to the recurrence given in [BKM+92] that lead to its competitive ratio of 1/4.

Our implementation is done in Python and C, and uses the lp_solve [BEN] package for linear programming solutions. All experiments are run on a standard 2010 computer with a 3.2GHz CPU and 4GB of RAM running Linux.

3.8.1 Varying Tasksets Without Constraints

The algorithm DOVER was proved in [KS95] to have optimal competitive factor, i.e., optimal competitive ratio under the worst-case taskset. However, our experiments

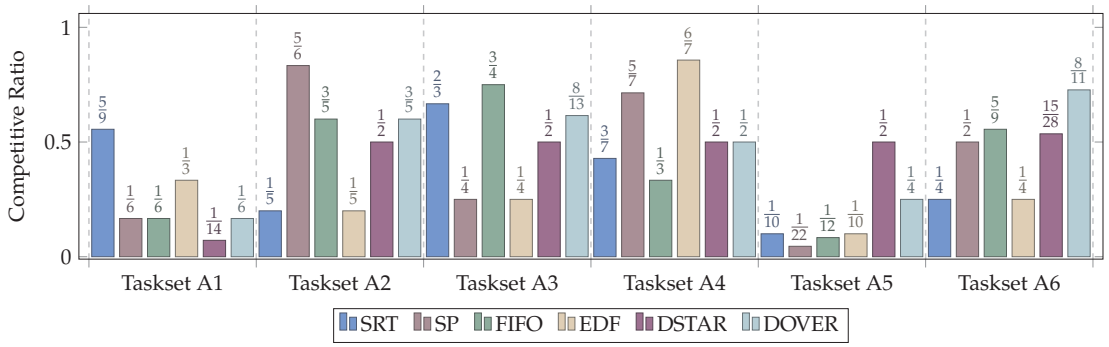


Figure 3.6: The competitive ratio of the examined algorithms in various tasksets under no constraints. Every examined algorithm is optimal in some taskset, among all others.

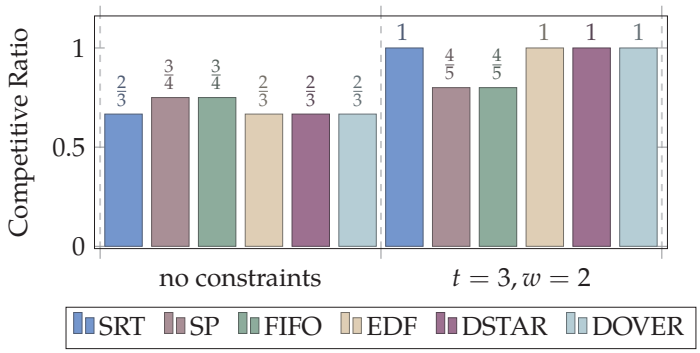


Figure 3.7: Restricting the absolute workload generated by the adversary typically increases the competitive ratio, and can vary the optimal scheduler. On the left, the performance of each scheduler is evaluated without restrictions: FIFO, SP behave best. When restricting the adversary to at most 2 units of workload in the last 3 rounds, FIFO and SP become suboptimal, and are outperformed by other schedulers.

reveal that this performance guarantee is not universal, in the sense that DOVER is outperformed by other schedulers for specific tasksets. This observation applies to all on-line algorithms examined: As shown in Figure 3.6, even without constraints on the adversary, for every scheduling algorithm, there are tasksets in which it achieves the highest competitive ratio among all others. Note that this high variability of the optimal on-line algorithm across tasksets makes our automated analysis framework an interesting tool for the application designer. Table 3.1 lists the tasksets A1-A6 used for Figure 3.6. The tasks are ordered by their static priorities, which determine the SP scheduler, as well as the way ties are broken by other schedulers. Along with each tasket its importance ratio k is shown.

	1.5	1	0.8	0.6	0.4	0.3	0.1	0.78	0.05
FIFO	✓	✓	✓	✓	✓				✓
SP	✓						✓		✓
SRT	✓				✓	✓	✓	✓	✓

Table 3.2: Columns show the mean workload restriction. The check-marks indicate that the corresponding scheduler is optimal for that mean workload restriction, among the six schedulers we examined. We see that the optimal scheduler can vary as the restrictions are tighter, and in a non-monotonic way. EDF, DSTAR and DOVER were not optimal in any case and hence not mentioned.

	τ_1	τ_2	τ_3
C_i	1	1	1
D_i	1	2	1
V_i	3	3	1

	τ_1	τ_2	τ_3
C_i	2	5	5
D_i	7	5	6
V_i	3	2	1

Table 3.3: Taskset of Figure 3.7 (left) and Table 3.2 (right).

3.8.2 Fixed Taskset with Varying Constraints

We also consider fixed tasksets under various constraints (such as sporadicity or workload restrictions) for admissible job sequences. Figure 3.7 shows our experimental results for workload safety constraints, which again reveal that, depending on workload constraints, we can have different optimal schedulers. Finally, we consider limit-average constraints and observe that varying these constraints can also vary the optimal scheduler for a fixed taskset: As Table 3.2 shows, the optimal scheduler can vary highly and non-monotonically with stronger limit-average workload restrictions. The tasksets for both experiments are shown in Table 3.3

3.8.3 Running Times

Table 3.4 summarizes some key parameters of our various tasksets, and gives some statistical data on the observed running times in our respective experiments. Even though the considered tasksets are small, the very short running times of our prototype implementation reveal the principal feasibility of our approach. We believe that further application-specific optimizations, augmented by abstraction and symmetry reduction techniques, will allow to scale to larger applications.

3.8.4 Competitive Ratio of TD1

We have also considered the performance of the on-line scheduler TD1 in zero laxity tasksets with uniform value-density (i.e., for each task τ_i, we have $C_i = D_i = V_i$). Following [BKM+92], we have constructed a series of tasksets parametrized by some

Taskset	N	$D_{\max}$	Size (nodes)		Time (s)	
			Clairv.	Product	Mean	Max
B01	2	7	19	823	0.04	0.05
B02	2	8	26	1997	0.39	0.58
B03	2	9	34	4918	10.02	15.21
B04	3	7	19	1064	0.14	0.40
B05	3	8	26	1653	0.66	2.05
B06	3	9	34	7705	51.04	136.62
B07	4	7	19	1711	2.13	6.34
B08	4	8	26	3707	13.88	34.12
B09	4	9	44	10 040	131.83	311.94
B10	5	7	19	2195	5.73	16.42
B11	5	8	32	9105	142.55	364.92
B12	5	9	44	16 817	558.04	1342.59

Table 3.4: Scalability of our approach for tasksets of various sizes N and $D_{\max}$. For each taskset, the size of the state space of the clairvoyant scheduler is shown, along with the mean size of the product LTS, and the mean and maximum time to solve one instance of the corresponding ratio objective.

Taskset	η	Taskset	**Comp. Ratio**
C1	2	$\{1,1\}$	**1**
C2	3	$\{1,2,3\}$	**1/2**
C3	3.1	$\{1,3,7,13,19\}$	**7/25**
C4	3.2	$\{1,3,7,13,20,23\}$	**1/4**
C5	3.3	$\{1,3,7,14,24,33\}$	**1/4**
C6	3.4	$\{1,3,7,14,24,34\}$	**1/4**

Table 3.5: Competitive ratio of TD1.

positive real $\eta < 4$, which guarantee that the competitive ratio of every on-line scheduler is upper bounded by $\frac{1}{\eta}$. Given η, each taskset consists of tasks τ_i such that C_i is given by the following recurrence, as long as $C_{i+1} > C_i$.

$$(i)\ C_0 = 1 \qquad (ii)\ C_{i+1} = \eta \cdot C_i - \sum_{j=0}^{i} C_j$$

In [BKM+92], TD1 was shown to have competitive factor $\frac{1}{4}$, and hence a competitive ratio that approaches $\frac{1}{4}$ from above, as $\eta \to 4$ in the above series of tasksets. Table 3.5 shows the competitive ratio of TD1 in this series of tasksets. Each taskset is represented as a set $\{C_i\}$, where each C_i is given by the above recurrence, rounded up to the first integer. We indeed see that the competitive ratio drops until it stabilizes to $\frac{1}{4}$.

Finally, note that the zero-laxity restriction allows us to process tasksets where $D_{\max}$ is much higher than what we report in Table 3.4. The results of Table 3.5 were produced in less than a minute in total.

3.9 Modeling as a Graph Game

Obviously, the real-time scheduling problem can be viewed as an instance of a game between the on-line algorithm (Player 1) and an adversary (Player 2) that determines the task sequence. In the last sections of this chapter we will show how the powerful "framework" of *graph games* [Mar75, Sha53] can be utilized for competitive analysis of real-time scheduling algorithms.

Notation on Graph Games

A *partial-observation game* (or simply a *game*) is a tuple $\mathcal{G} = \langle S^G, \Sigma_1^G, \Sigma_2^G, \delta^G, \mathcal{O}_S, \mathcal{O}_\Sigma \rangle$ with the following components:

State space: The set S^G is a finite set of states.

Actions: Σ_i^G $(i = 1, 2)$ is a finite set of actions for Player i.

Transition function: The transition function $\delta^G : S^G \times \Sigma_1^G \times \Sigma_2^G \to S^G$ given the current state $s \in S^G$, an action $\alpha_1 \in \Sigma_1^G$ for Player 1, and an action $\alpha_2 \in \Sigma_2^G$ for Player 2, gives the next (or successor) state $s' = \delta^G(s, \alpha_1, \alpha_2)$. A shorter form to depict the previous transition is to write the tuple $(s, \alpha_2, \alpha_1, s')$.

Observations: The set $\mathcal{O}_S \subseteq 2^{S^G}$ is a finite set of observations for Player 1 that partitions the state space S^G. The partition uniquely defines a function $\mathsf{obs}_S : S^G \to \mathcal{O}_S$ that maps each state to its observation such that $s \in \mathsf{obs}_S(s)$ for all $s \in S^G$. In other words, the observation partitions the state space according to equivalence classes. Similarly, $\mathcal{O}_\Sigma$ is a finite set of observations for Player 1 that partitions the action set Σ_2^G, and analogously defines the function obs_Σ. Intuitively, Player 1 will have partial observation, and can only obtain the current observation of the state (not the precise state but only the equivalence class the state belongs to) and current observation of the action of Player 2 (but not the precise action of Player 2) to make her choice of action.

3.9.1 Plays

In a game, in each turn, first Player 2 chooses an action, then Player 1 chooses an action, and given the current state and the joint actions, we obtain the next state following the transition function δ^G.

A *play* in $\mathcal{G}$ is an infinite sequence of states and actions $\mathscr{P} = s^1, \alpha_2^1, \alpha_1^1, s^2, \alpha_2^2, \alpha_1^2, s^3, \alpha_2^3, \alpha_1^3, s^4 \ldots$ such that, for all $j \geqslant 1$, we have $\delta^G(s^j, \alpha_1^j, \alpha_2^j) = s^{j+1}$. The *prefix up to* s^n of the play $\mathscr{P}$ is denoted by $\mathscr{P}(n)$ and corresponds to the starting state of the n-th turn. The set of plays in $\mathcal{G}$ is denoted by $\mathscr{P}^\infty$, and the set of corresponding finite prefixes is denoted by $\mathsf{Prefs}(\mathscr{P}^\infty)$.

3.9.2 Strategies

A *strategy* for a player is a recipe that specifies how to extend finite prefixes of plays. We will consider *memoryless* strategies for Player 1 (where its next action depends only on the current state, but not on the entire history) and general history-dependent strategies for Player 2. A strategy for Player 1 is a function $\pi^G : \mathcal{O}_S \times \mathcal{O}_\Sigma \to \Sigma_1^G$ that given the current observation of the state and the current observation on the action of Player 2, selects the next action. A strategy for Player 2 is a function $\sigma^G : \mathsf{Prefs}(\mathscr{P}^\infty) \to \Sigma_2^G$ that given the current prefix of the play chooses an action. Observe that the strategies for Player 1 are both observation-based and memoryless; i.e., depend only on the current observations (rather than the whole history), whereas the strategies for Player 2 depend on the history. A memoryless strategy for Player 2 only depends on the last state of a prefix. We denote by $\Pi_{\mathcal{G}}^M$, $\Sigma_{\mathcal{G}}$, $\Sigma_{\mathcal{G}}^M$ the set of all observation-based memoryless Player 1 strategies, the set of all Player 2 strategies, and the set of all memoryless Player 2 strategies, respectively. In sequel when we write strategy for Player 1 we consider only observation-based memoryless strategies. Given a strategy π^G and a strategy σ^G for Player 1 and Player 2, and an initial state s^1, we obtain a unique play $\mathscr{P}(s^1, \pi^G, \sigma^G) = s^1, \alpha_2^1, \alpha_1^1, s^2, \alpha_2^2, \alpha_1^2, s^3, \ldots$ such that, for all $n \geqslant 1$, we have $\sigma^G(\mathscr{P}(n)) = \alpha_2^n$ and $\pi^G(\mathsf{obs}_S(s^n), \mathsf{obs}_\Sigma(\alpha_2^n)) = \alpha_1^n$.

3.9.3 Objectives

Very similar to the objectives introduced in the multi-graph representation that was discussed in Section 3.5.1 we are defining *mean-payoff* (or long-run average or limit-average) objectives, as well as *ratio* objectives on graph games. Safety and liveness objective in graph games can be modeled easily by introducing additional winning states for Player 1 and by Büchi states.

For mean-payoff objectives we will consider games with a reward function $r : S^G \times \Sigma_1^G \times \Sigma_2^G \times S^G \to \mathbb{Z}$ that maps every transition to an integer valued reward. The reward function naturally extends to plays, where $r(\mathscr{P}, k) = \sum_{i=1}^k r(s^i, \alpha_1^i, \alpha_2^i, s'^i)$, for $k \geqslant 1$ denote the sum of the rewards for the prefix $\mathscr{P}(k+1)$, i.e., the sum of the rewards for the first k turns. The mean-payoff of a play $\mathscr{P}$ is defined as:

$$\mathsf{MP}(r, \mathscr{P}) = \liminf_{k \to \infty} \frac{1}{k} \cdot r(\mathscr{P}, k).$$

In case of ratio objectives, we will consider games with two reward functions $r_1 : S^G \times \Sigma_1^G \times \Sigma_2^G \times S^G \to \mathbb{N}$ and $r_2 : S^G \times \Sigma_1^G \times \Sigma_2^G \times S^G \to \mathbb{N}$ that map every transition

to a non-negative valued reward. Using the same extension of reward functions to plays as before, the ratio of a play $\mathscr{P}$ is defined as:

$$\mathsf{Ratio}(r_1, r_2, \mathscr{P}) = \liminf_{k\to\infty} \frac{1 + r_1(\mathscr{P}, k)}{1 + r_2(\mathscr{P}, k)}.$$

3.9.4 Decision Problems

Analogous to Section 3.5.2, we define the relevant decision problems on games as: given a game $\mathscr{G}$, a starting state s^1, reward functions r, r_1, r_2, and a rational threshold $\nu \in \mathbb{Q}$, whether it holds that

$$\sup_{\pi^G \in \Pi^M_{\mathscr{G}}} \inf_{\sigma^G \in \Sigma_{\mathscr{G}}} \mathsf{MP}(r, \mathscr{P}(s^1, \pi^G, \sigma^G)) \geqslant \nu;$$

and

$$\sup_{\pi^G \in \Pi^M_{\mathscr{G}}} \inf_{\sigma^G \in \Sigma_{\mathscr{G}}} \mathsf{Ratio}(r_1, r_2, \mathscr{P}(s^1, \pi^G, \sigma^G)) \geqslant \nu.$$

Remark 3. Note, that the decision problems of the graph game problem are defined over the $\sup_{\pi^G \in \Pi^M_{\mathscr{G}}}$, taking all possible memoryless strategies into account. This corresponds to all possible on-line scheduling strategies, whereas the previously discussed multi-graph problem explicitly used one deterministic strategy for the on-line scheduler.

3.9.5 Perfect-information Games

Games of complete-observation (or perfect-information games) are a special case of partial-observation games where $\mathcal{O}_S = \{\{s\} \mid s \in S^G\}$ and $\mathcal{O}_\Sigma = \{\{\alpha_2\} \mid \alpha_2 \in \Sigma_2^G\}$, i.e., every individual state and action is fully visible to Player 1, and thus she has perfect information. For perfect-information games, for the sake of simplicity, we will omit the corresponding observation sets from the description of the game.

3.10 Complexity Results

In this section, we establish the complexity of the decision problems we consider for partial-observation mean-payoff and ratio objectives. In particular, we will show that for partial-observation games with memoryless strategies for Player 1 all the decision problems are NP-complete.

Transformation

We start with a simple transformation that will allow us to simplify the technical results we prove. In the definition of games, we defined them in such a way that in every state every action was available for the players for simplicity. For technical development, we will consider games where, at certain states, some actions are not allowed for a player. The transformation of such games to games where all actions are allowed is as follows: we add two absorbing dummy states (with only a self-loop), one for Player 1 and the other for Player 2, and assign rewards in a way such that the objectives are violated for the player. For example, for mean-payoff objectives with threshold $\nu > 0$, we assign reward 0 for the only out-going (self-loop) transition of the Player 1 dummy state, and a reward strictly greater than ν for the self-loop of the Player 2 dummy state; and in case of ratio-objectives we assign the reward pairs similarly. Given a state s, if Player 1 plays an action that is not allowed at s, we go to the dummy Player 1 state; and if Player 2 plays an action that is not allowed we go to the Player 2 dummy state. Thus we have a simple linear time transformation. Hence, for technical convenience, we can assume in the sequel that different states have different sets of available actions for the players. We first start with the hardness result.

Lemma 4. *The decision problems for partial-observation games with mean-payoff objectives and ratio objectives, i.e, whether* $\sup_{\pi^G \in \Pi^M_{\mathscr{G}}} \inf_{\sigma^G \in \Sigma_{\mathscr{G}}} \mathsf{MP}(r, \mathscr{P}(s^1, \pi^G, \sigma^G)) \geqslant \nu$ *(respectively* $\sup_{\pi^G \in \Pi^M_{\mathscr{G}}} \inf_{\sigma^G \in \Sigma_{\mathscr{G}}} \mathsf{Ratio}(r_1, r_2, \mathscr{P}(s^1, \pi^G, \sigma^G)) \geqslant \nu$*), are* NP*-hard in the strong sense.*

Proof. We present a reduction from the 3-SAT problem, which is NP-hard in the strong sense [Pap93]. Let Ψ be a 3-SAT formula over n variables $x_1, x_2, \ldots, x_n$ in conjunctive normal form, with m clauses $\mathfrak{c}_1, \mathfrak{c}_2, \ldots, \mathfrak{c}_m$ consisting of a disjunction of 3 literals (a variable x_k or its negation $\overline{x}_k$) each. We will construct a game graph $\mathscr{G}_\Psi$ as follows:

State space: $S^G = \{s_{\mathsf{init}}\} \cup \{s_{i,j} \mid 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant 3\} \cup \{\mathsf{dead}\}$; i.e., there is an initial state s_{init}, a dead state dead, and there is a state $s_{i,j}$ for every clause $\mathfrak{c}_i$ and a literal j of i.

Actions: The set of actions applicable for Player 1 is $\{\mathsf{true}, \mathsf{false}, \bot\}$ and for Player 2 is $\{1, 2, \ldots, m\} \cup \{\bot\}$.

Transitions: In the initial state s_{init}, Player 1 has only one action $\bot$ available, and Player 2 has actions $\{1, 2, \ldots, m\}$ available, and given action $1 \leqslant i \leqslant m$, the next state is $s_{i,1}$. In all other states Player 2 has only one action $\bot$ available. In states $s_{i,j}$ Player 1 has two actions available, namely, true and false. The transitions are as follows:

- If the action of Player 1 is true in $s_{i,j}$, then (i) if the j-th literal in $\mathfrak{c}_i$ is x_k, then we have a transition back to the initial state; and (ii) if the j-th literal in $\mathfrak{c}_i$

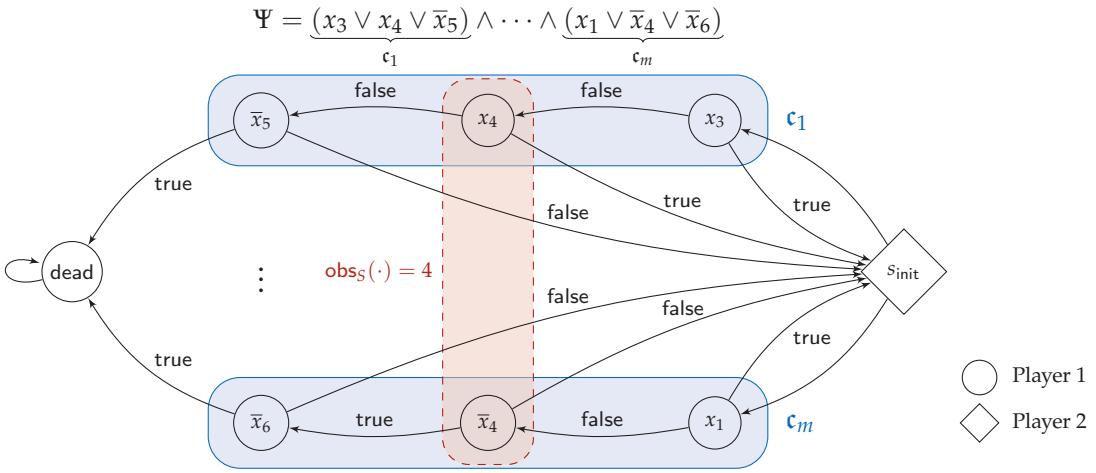


Figure 3.8: Illustration of construction of the game from a 3-SAT formula.

is $\overline{x}_k$ (negation of x_k), then we have a transition to $s_{i,j+1}$ if $j \in \{1,2\}$, and if $j = 3$, we have a transition to dead.

- If the action of Player 1 is false in $s_{i,j}$, then (i) if the j-th literal in $\mathfrak{c}_i$ is $\overline{x}_k$ (negation of x_k), then we have a transition back to the initial state; and (ii) if the j-th literal in $\mathfrak{c}_i$ is x_k, then we have a transition to $s_{i,j+1}$ if $j \in \{1,2\}$, and if $j = 3$, we have a transition to dead.

In state dead both players have only one available action $\perp$, and dead is a state with only a self-loop (transition only to itself).

Observations: First, Player 1 does not observe the actions of Player 2 (i.e., Player 1 does not know which action is played by Player 2). The observation mapping for the state space for Player 1 is as follows: the set of observations is $\{0, 1, \ldots, n\}$ and we have $\mathsf{obs}_S(s_{\mathsf{init}}) = \mathsf{obs}_S(\mathsf{dead}) = 0$ and $\mathsf{obs}_S(s_{i,j}) = k$ if the j-th variable of $\mathfrak{c}_i$ is either x_k or its negation $\overline{x}_k$, i.e., the observation for Player 1 corresponds to the variables.

A pictorial description is shown in Fig 3.8. The intuition for the above construction is as follows: Player 2 chooses a clause from the initial state s_{init}, and an observation-based memoryless strategy for Player 1 corresponds to a non-conflicting assignment to the variables. Note that Player-1 strategies are observation-based memoryless; hence, for every observation (i.e., a variable), it chooses a unique action (i.e., an assignment) and thus non-conflicting assignments are ensured. We consider $\mathcal{G}_\Psi$ with reward functions r, r_1, r_2 as follows: r_2 assigns reward 1 to all transitions; r and r_1 assigns reward 1 to all

transitions other than the self-loop at state dead, which is assigned reward 0. We ask the decision questions with $\nu = 1$. Observe that the answer to the decision problems for both mean-payoff and ratio objectives is "Yes" iff the state dead can be avoided by Player 1 (because if dead is reached, then the game stays in dead forever, violating both the mean-payoff as well as the ratio objective). We now present the two directions of the proof.

Satisfiable implies dead **is not reached:** We now show that if Ψ is satisfiable, then Player 1 has an observation-based memoryless strategy π^{G*} to ensure that dead is never reached. Consider a satisfying assignment A for Ψ, then the strategy π^{G*} for Player 1 is as follows: given an observation k, if A assigns true to variable x_k, then the strategy π^{G*} chooses action true for observation k, otherwise it chooses action false. Since the assignment A satisfies all clauses, it follows that for every $1 \leqslant i \leqslant m$, there exists $s_{i,j}$ such that the strategy π^{G*} for Player 1 ensures that the transition to s_{init} is chosen. Hence the state dead is never reached, and both the mean-payoff and ratio objectives are satisfied.

If dead **is not reached, then Ψ is satisfiable:** Consider an observation-based memoryless strategy π^{G*} for Player 1 that ensures that dead is never reached. From the strategy π^{G*} we obtain an assignment A as follows: if for observation k, the strategy π^{G*} chooses true, then the assignment A chooses true for variable x_k, otherwise it chooses false. Since π^{G*} ensures that dead is not reached, it means for every $1 \leqslant i \leqslant m$, that there exists $s_{i,j}$ such that the transition to s_{init} is choosen (which ensures that $\mathfrak{c}_i$ is satisfied by A). Thus since π^{G*} ensures dead is not reached, the assignment A is a satisfying assignment for Ψ.

Thus, it follows that the answers to the decision problems are "Yes" iff Ψ is satisfiable, and this establishes the NP-hardness result. ■

> *Remark* 4. Note that our reduction used only weight values 0 and 1. This implies that NP-hardness holds also for the case where weight values are bounded by a constant. This is also true for objectives with multiple dimensions as introduced in Section 3.5.2.

The NP upper bounds

We now present the NP upper bounds for the problems. In both cases, the polynomial witness for the decision problem is a memoryless strategy (i.e., if the answer to the

decision problem is "Yes", then there is a witness memoryless strategy π^G for Player 1, and the NP algorithm just guesses the witness strategy π^G). Once the memoryless strategy is guessed and fixed, we need to show that we have a polynomial time verification procedure. The polynomial time verification procedures are as follows:

Mean-payoff objectives: Once the memoryless strategy for Player 1 is fixed, the game problem reduces to a problem when there is only Player 2. Therefore, the problem reduces to the path problem in directed graphs analyzed and shown to be solvable in polynomial time in the Section 3.5.2.

Ratio objectives: Again once the memoryless strategy for Player 1 is fixed, the game problem reduces to a decision problem on directed graphs. The same reduction from ratio objectives to mean-payoff objectives introduced in Section 3.5.2 can be applied, giving an polynomial time verification algorithm for ratio objectives.

We summarize the result in the following theorem.

Theorem 4. *The decision problems for partial-observation games with mean-payoff objectives and ratio objectives, i.e., whether* $\sup_{\pi^G \in \Pi^M_{\mathscr{G}}} \inf_{\sigma^G \in \Sigma_{\mathscr{G}}} \mathsf{MP}(r, \mathscr{P}(s^1, \pi^G, \sigma^G)) \geqslant \nu$ *respectively* $\sup_{\pi^G \in \Pi^M_{\mathscr{G}}} \inf_{\sigma^G \in \Sigma_{\mathscr{G}}} \mathsf{Ratio}(r_1, r_2, \mathscr{P}(s^1, \pi^G, \sigma^G)) \geqslant \nu$, *are* NP*-complete.*

3.11 The Synthesis Problem

Devising an algorithm-specific LTS, like described in Section 3.2.1, is already sufficient for computing its competitive ratio. For a general competitive analysis, a *non-deterministic* LTS $L_{\mathscr{G}} = (S_{\mathscr{G}}, s_{\mathscr{G}}, \Sigma, \Pi, \Delta_{\mathscr{G}})$ with an associated reward function $r_{\mathscr{G}}$ that can simulate all possible on-line algorithms in a memoryless way is required. Such an LTS is already introduced in Section 3.2.2 for the clairvoyant algorithm. Unfortunately, this time we cannot apply the Reductions introduced in Section 3.7 and thus not benefit of the 90% state space reduction. Obviously, this non-deterministic finite-state LTS $L_{\mathscr{G}}$ can simulate any possible real-time scheduling algorithm with a memoryless strategy, i.e., one that does not need to refer to the history of actions/transitions, as all required history information is encoded in the state.

We can simply interpret such a non-deterministic transition system as a graph game $\langle \mathcal{S}, \Sigma_1, \Sigma_2, \delta \rangle$, where Σ_1 (the actions of Player 1) correspond to the output actions Π in the LTS and Σ_2 (the actions of Player 2) correspond to the input actions Σ. That is, Player 2 (the adversary) chooses the released tasks while Player 1 chooses the actual transitions in δ. Thus we have a perfect-information game, and every memoryless strategy correspond to a scheduling algorithm and vice-versa (i.e., every scheduling algorithm is a memoryless strategy of the game).

For solving the synthesis problem, we construct a partial-observation game $\mathscr{G}_{\mathcal{CR}}$ as follows: $\mathscr{G}_{\mathcal{CR}} = \langle S^G = S \times S, \Sigma_1^G = \Sigma_1, \Sigma_2^G = \Sigma_2 \times \Sigma_1, \delta^G, \mathcal{O}_S, \mathcal{O}_\Sigma \rangle$. Intuitively, we construct a product game with two components, where Player 1 only observes the first component and makes the choice of the transitions there; and Player 2 is in charge of

choosing the input and also the transitions of the second component. However, due to partial observation, Player 1 does not observe the choice of transitions in the second component. We describe the transition and the observation mapping to capture this:

(i) the transition function $\delta^G : S^G \times \Sigma_1^G \times \Sigma_2^G \to S^G$ is as follows:

$$\delta^G((s_1, s_2), \alpha_1, (\alpha_2, \alpha_1')) = (\delta(s_1, \alpha_1, \alpha_2), \delta(s_2, \alpha_2, \alpha_1'));$$

(ii) the observation for states for Player 1 maps every state to the first component, i.e., $\mathsf{obs}_S((s_1, s_2)) = s_1$ and the observation for actions for Player 1 maps every state to the first component as well (i.e., the input from Player 2), i.e., $\mathsf{obs}_\Sigma((\alpha_2, \alpha_1')) = \alpha_2$.

The two reward functions needed for solving the ratio objective in the game are as follows: the reward function r_1 gives reward according to $r_{\mathscr{G}}$ and the transitions of the first component and the reward function r_2 assigns reward according to $r_{\mathscr{G}}$ and the transitions of the second component. Note that the construction ensures that we compare the utility of an on-line algorithm (transitions of the first component chosen by Player 1) and an off-line algorithm (chosen by Player 2 using the second component) that operate on the *same* input sequence. It follows that there is an on-line algorithm with competitive-ratio at least ν iff $\sup_{\pi^G \in \Pi^M_{\mathscr{G}}} \inf_{\sigma^G \in \Sigma_{\mathscr{G}}} \mathsf{Ratio}(r_1, r_2)(\mathscr{P}(s^1, \pi^G, \sigma^G)) \geqslant \nu$, where s^1 is the start state derived from the LTSs. By Theorem 4, the decision problem is in NP in the size of the LTS. Since the strategy of Player 1 can directly be translated to a scheduling algorithm the solution of the synthesis problem directly follows from finding $\nu^* = \sup\{\nu \in \mathbb{Q} : \text{the answer to the decision problem is yes}\}$ and the corresponding witness.

Theorem 5. *For the class of scheduling problems defined in Section 3.1, the decision problem, if there is an on-line scheduler with a competitive ratio at least a rational number ν is in* NP *in the size of the LTS constructed from the scheduling problem.*

3.12 Bibliographic Remarks

Algorithmic game theory [NRTV07] has been applied to classic scheduling problems since decades, primarily in economics and operations research, see e.g. [Kou11] for just one example of some more recent work. It has also been applied for real-time scheduling of *hard* real-time tasks in the past: Besides Altisen et al. [AGS02], who used games for synthesizing controllers dedicated to meeting all deadlines, Bonifaci and Marchetti-Spaccamela [BMS12] employed graph games for automatic feasibility analysis of sporadic real-time tasks in multiprocessor systems: Given a set of sporadic tasks (where consecutive releases of jobs of the same task are separated at least by some sporadicity interval), the algorithms provided in [BMS12] allow to decide, in polynomial time, whether some given scheduling algorithm will meet *all* deadlines. A partial-information game variant of their approach also allows to synthesize an optimal scheduling algorithm for a given task set (albeit not in polynomial time).

Whereas these approaches do not generalize to competitive analysis of tasks with firm deadlines, we showed in [CKS13] that graph games with mean-payoff resp. ratio objectives provide a very powerful and flexible framework for this purpose as well: In case of a given on-line algorithm, like algorithm TD1 of [BKM+92], this results in a perfect-information game that can be solved in polynomial time (in the size of the game graph). In order to automatically determine optimal scheduling algorithms, which we proved to be NP-complete, one has to resort to a partial-information game. We also argued that any safety objective (prohibiting reachability of certain bad states), any Büchi objective (ensuring infinite reachability of certain target states), and multiple mean-payoff objectives (securing desired limit-average behaviors) can be added without unduly increasing the complexity of the decision problems involved. In [CPKS14] we extended our framework for the automated competitive analysis of on-line scheduling algorithms to handle optional safety, liveness, and limit-average constraints along with the reduction to multi-objective graphs.

Regarding firm deadline task scheduling in general, starting out from [BKM+92], classic real-time systems research has studied the competitive factor of both simple and extended real-time scheduling algorithms. The competitive analysis of simple algorithms (see Section 3.8 for the references) has been extended in various ways later on: Energy consumption [AMMMA04, DLA10] (including dynamic voltage scaling), imprecise computation tasks (having both a mandatory and an optional part and associated utilities) [BH98], lower bounds on slack time [BH97], and fairness [Pal04]. Note that dealing with these extensions involved considerable ingenuity and efforts w.r.t. identifying and analyzing appropriate worst case scenarios, which do not necessarily carry over even to minor variants of the problem. Maximizing cumulated utility while satisfying multiple resource constraints is also the purpose of the Q-RAM (QoS-based Resource Allocation Model) [RLLS97] approach.

Chapter 4

Round Synchronization

ANALYZING THE time complexity of an algorithm is at the core of computer science. Classically this is carried out by counting the number of steps executed by a Turing machine. In distributed computing, computations are typically viewed as being completed in zero time, focusing on communication delays only. This view is useful for algorithms that communicate heavily, with local operations of negligible duration between two communications.

Recall that this thesis is focusing on the implementation of an important subset of distributed algorithms where communication and computation are highly structured, namely synchronous, or *round-based algorithms* [Awe85, BK02, CBS09, RS94] where each process performs its computations in consecutive rounds. Thereby a single *round* consists of

(1) the processes exchanging data with each other and

(2) each process executing local computations.

In Section 2.1.1, we introduced the *round-complexity* of a distributed algorithm as the number of rounds it takes to complete a task. We consider repeated instances of a problem, i.e., a problem is repeatedly solved during an infinite execution. Such problems arise when the distributed system under consideration provides a continuous service to the top-level application, e.g., repeatedly solves distributed consensus [LSP82] in the context of state-machine replication [Sch90]. A natural performance measure for these systems is the average number of problem instances solved per round during an execution. In case a single problem instance has round-complexity of a constant number $R \geqslant 1$ of rounds, we readily obtain a rate of $1/R$.

In this thesis we are interested in the time complexity in terms of Newtonian real-time. We therefore can scale the round-complexity with the duration (bounds) of a round, yielding a real-time rate of $1/RT$, if T is the duration of a single round. Note

that the attainable accuracy of the calculated real-time rate thus heavily relies on the ability to obtain a good measurement of T. In case the data exchange within a single round comprises each process broadcasting messages and receiving messages from all other processes, T can be related to message latency and local computation upper and lower bounds, typically yielding precise bounds for the round duration T. However, in the distributed systems in this thesis T cannot be easily related to message delays as they are facing the problem of message loss, and it might happen that processes have to resend messages several times before they are correctly received, and the next round can be started.

Detailed Road Map of this Chapter:

This chapter is based on the published articles [NFK13] and [FKN+13] in a joint work with Matthias Függer, Thomas Nowak, Ulrich Schmid, and Martin Zeiner. It is devoted to the analysis of an retransmission-based synchronizer, i.e., a technique to cope with message loss and to simulate a perfect round structure.

In Section 4.1, we specify the synchronizer algorithm and formally prove it correct. In Section 4.2, we define probabilistic message loss and fair-lossy executions, our preferred model for execution. Furthermore we will introduce a constraint on the maximum number of retransmissions, which reduces the computational effort from exponentially to polynomial w.r.t. the system size. In Section 4.3, we construct a Markov chain suitable to calculate the expected round duration of the synchronizer and we provide a generic way to solve it. First results are given in Section 4.4 together with a bound on the rate of convergence of the Markov chain towards the expected round duration. In Section 4.5, the constraint on the finite number of retransmissions is relaxed by describing the problem in a dual way. As a drawback, the new model increases the inherent complexity of the problem from polynomial to exponential. We therefore introduce conditional forgetting as a way to reduce the complexity. In Section 4.6, we derive explicit formulas for two cases of the conditional forgetting that can be solved very efficiently. In Section 4.7, we specify the Markov chain in this new model and derive some general results for the derivative in $p = 1$ and for the order of growth for $p \to 0$. Furthermore, we derive computationally feasible lower bounds for the computationally exhaustive cases. In Section 4.8, we discuss the results of the analysis and compare them to Monte-Carlo simulations. Finally, Section 4.9 provides additionally bibliographic information about this chapter.

4.1 The Retransmission Scheme

In this section, we formally present the object of study: a general technique to cope with message loss in distributed systems by retransmissions. Instead of handling message loss directly in the algorithm, it is often more convenient for the algorithm's designer to separate concerns into (1) simulating perfect rounds, i.e., rounds *without* message loss, on top of a system with message loss, and (2) to run a simpler algorithm on top

of the simulated perfect rounds. Simulations that provide stronger communication directives on top of a system satisfying weaker communication directives are commonly used in distributed computing [DLS88, CBS09]. In this section we present one such simulation—a retransmission scheme—and prove it correct. Note that the proposed retransmission scheme is a modified version of the α synchronizer [Awe85]. However, it does not use the acknowledgment message.

4.1.1 Computational Model

We assume a distributed system comprising a fully connected communication network between *processes* taken from the set $\mathcal{P} = \{1, 2, \ldots, \mathcal{N}\}$. Each process i has a *local state* s_i; a *global state* of the distributed system is a collection of local states $(s_i)_{i \in \mathcal{P}}$. Processes communicate by message passing.

Formally, an *algorithm* A for the distributed system comprises the following parts:

(A1) For every process i, a *set of possible local states* $\mathcal{S}_i$, a *set of possible initial local states* $\mathcal{S}_i^0$, and the *set of possible messages* $\mathcal{M}$, not containing $\perp$. We assume without loss of generality that the sets $\mathcal{S}_i$ are pairwise disjoint.

(A2) A pair of functions $(\text{Send}_i, \text{Next}_i)$ for every process i: The *send function* Send_i for every process i, is from $\mathcal{S}_i$ to $2^{\mathcal{M}}$, and maps a local state to a nonempty finite set of messages to send. The *next state function* Next_i for every process i, is from $\mathcal{S}_i \times 2^{\mathcal{M} \times \mathcal{P}}$ to $\mathcal{S}_i$, and maps a local state and a set $R \subseteq \mathcal{M} \times \mathcal{P}$ of received messages, labeled with their respective sender, to the next local state.

Computation at processes is assumed to occur in sequences of steps locally happening at the processes. In a step, a process atomically

(E1) receives a set of messages,

(E2) computes its next local state, and

(E3) sends (broadcasts) a nonempty finite set of messages to all other processes.

Note that our definition of a step differs from classic definitions with respect to (E3), potentially allowing an algorithm to broadcast a set of messages instead of a single message per step. While in distributed systems without transmission failures, algorithms for both kinds of definitions can be easily reduced to each other by joining all messages to be sent in a step into a single message, this is not the case for distributed systems that have to cope with transmission failures, like those we consider in this work. There, the extension allows for finer grained modeling of benign transmission failures, i.e., failures where contents of messages are not changed: Instead of the single message, sent in a step, either being received in some other step or not, an arbitrary subset of messages sent in a step can be received in some other step.

Formally we define: An *event* is a tuple (i, R), where i is a process and R is the set of messages, tagged with their respective senders (i.e., $R \subseteq \mathcal{M} \times \mathcal{P}$) that are

received by process i in the event. An *execution* E of an algorithm A is a sequence of events and local states such that for every process i, the projection $E(i)$ to process i's events and states is an alternating sequence of local states and events $E(i) = s_i(1), e_i(2), s_i(2), \ldots, e_i(k), s_i(k), \ldots$, such that

(Ex1) every $s_i(1)$ is an initial (local) state of process i and

(Ex2) for every $k > 1$ with $e_i(k) = (i, R)$, it is $s_i(k) = \text{Next}_i\big(s_i(k-1), R\big)$.

In execution E, event e is *before* event e' if e appears before e' in sequence E. We say that process i *receives* message m from j in *step* k if $(m, j) \in R$ where $e_i(k) = (i, R)$. We further say that process i *sends* (broadcasts) message m in *step* k, if $m \in \text{Send}_i\big(s_i(k)\big)$.

It remains to specify the relation between message sends and receives that has to hold during an execution. We do this by means of communication axioms which denote a condition on the distributed system's communication behavior: The system can either satisfy an axiom or not. The following are communication axioms used in the sequel:

NoGen For all processes i and j, if j receives message m from i, then i broadcasted m before.

FairLoss For all processes i and j, if i broadcasts the same message m in infinitely many steps, then j receives m from i in infinitely many steps.

Further desirable axioms are that of *communication closedness* CommClosed [CBS09], *perfect communication* PerfComm, and perfect communication for self loops, i.e., PerfComm*. They are defined by:

CommClosed For all processes i and j, if j receives message m from i in step $k > 1$, then i broadcasted m in step $k - 1$.

PerfComm For all processes i and j, if i broadcasts message m in step $k - 1$, $k > 1$, then j receives m from i in step k.

PerfComm* For all processes i, if i broadcasts message m in step $k - 1$, $k > 1$, then i itself receives m from i in step k.

Call an execution *admissible* if it satisfies NoGen, which is reasonable to assume for benign communication, and for each process i, $E(i)$ is infinite. A *fair-lossy execution* of an algorithm A is an admissible execution that satisfies axiom FairLoss. A *perfect round execution* is an admissible execution that satisfies axioms CommClosed and PerfComm.

4.1.2 Simulating Perfect Round Executions

Our goal is to determine the round duration of a retransmission scheme that simulates a perfect round execution on top of a fair-lossy execution. We thus proceed by introducing a notion of simulation. Let B be an algorithm (designed for perfect round executions). We define what it means for an algorithm A (designed for fair-lossy executions) to

simulate algorithm B. The idea is that algorithm A's local state includes B's local state in a special variable *Bstate*. Further, in each event, algorithm A is allowed to trigger a local event of algorithm B. It does this by setting a local variable *trigger* to *true*, and handing over a set of received messages to its local instance of B. Algorithm B then makes a step and updates *Bstate*.

Formally we define: Let $\mathcal{S}_i^{(B)}$ and $\mathcal{M}^{(B)}$ denote the sets of local states and the set of messages of B, respectively. We demand of algorithm A that its local states contain the variables *Bstate*, *trigger*, and *Bevent*. Variable *Bstate*'s type at process i is $\mathcal{S}_i^{(B)}$, variable *trigger* is Boolean, and variable *Bevent*'s type is $\Sigma^{(B)}$, where $\Sigma^{(B)}$ is the set of events of algorithm B. Given an execution E of algorithm A, we define the *B-projection* $E \restriction B$ *of* E in the following way:

(P1) Let F denote the subsequence of E that arises when (a) deleting all events, and (b) all states in which $trigger = false$.

(P2) We define $E \restriction B$ to be the sequence arising from F when replacing each processor's first state, $s_i(1)$, by $s_i(1)[Bstate]$, and every but each processor's first state, $s_i(r)$, by the two elements $s_i(r)[Bevent]$, $s_i(r)[Bstate]$ where $s[X]$ denotes the value of variable X in state s.

Definition 1. *We say that algorithm A* simulates *B in perfect rounds on top of fair-lossy executions if, (S1) $trigger = true$ in every initial state of A, (S2) for every initial state $s_i^{(B)}(1)$ of B, there exists an initial state $s_i(1)$ of A such that $s_i(1)[Bstate] = s_i^{(B)}(1)$, and (S3) for every fair-lossy execution E of A, execution $E \restriction B$ is a perfect round execution of B.*

4.1.3 The Algorithm

We are now ready to formally state a retransmission-based algorithm that simulates perfect round executions on top of fair-lossy ones, and prove it correct.

For every algorithm B, consider algorithm $A = A(B)$ presented in Algorithm 9. The idea of the simulation is simple: Each process steadily broadcasts (B1) its current (simulated) round number *Rnd* together with algorithm B's messages for the current round (*Rnd*) and, (B2) the previous round number $Rnd - 1$ together with algorithm B's messages for the previous round ($Rnd - 1$). A process waits in round *Rnd* until it has received all processes' round *Rnd* messages. When it does, it starts (simulated) round $Rnd + 1$.

The intuition for a process sending both its current and its previous round messages is the following: At some point during the execution, the value of any two processes' *Rnd* variables may differ by one, because of transmission failures. That is, while some process i already started simulated round K, and therefore waits for messages with round number K, another process j may still be in simulated round $K - 1$, waiting for messages with round number $K - 1$. Clearly, process i therefore must still send round $K - 1$ messages to j, until j, too, starts round K. Messages with round number less than $K - 1$, however, need not be sent by process i: It can be shown that at any point during

Algorithm 9: Process i's code in simulation algorithm $A(B)$

Code for processes $i, 1 \leqslant i \leqslant \mathcal{N}$:
Variables: $BState \leftarrow s_i^{(B)}(1)$; $trigger \leftarrow true$; $Bevent \leftarrow \perp$;
Variables: $BState_{old} \leftarrow \perp$; $\forall j\ \forall r$: $Rcv[j,r] \leftarrow \perp$; $Rnd \leftarrow 1$;

1 **Next State Function( )** *when receiving set of messages* R
2 **for** *received message* $(r,m) \in R$ *from process* j **do**
3 $Rcv[j,r] \leftarrow m$
4 $trigger \leftarrow false$
5 **if** *for all* j *in* Π: $Rcv[j,Rnd] \neq \perp$ **then**
6 $Bstate_{old} \leftarrow Bstate$
7 $trigger \leftarrow true$
8 $R' \leftarrow \{(Rcv[j,Rnd], j) \mid j \in \Pi\}$
9 $Bevent \leftarrow (i, R')$
10 $Bstate \leftarrow \text{Next}_i^{(B)}\left(Bstate, R'\right)$
11 $Rnd \leftarrow Rnd + 1$
12 **Send Function( )**
13 broadcast $\left(Rnd - 1, \text{Send}_i^{(B)}(Bstate_{old})\right)$; broadcast $\left(Rnd, \text{Send}_i^{(B)}(Bstate)\right)$

the execution, the values of any two processes' *Rnd* variables differ by at most one (cf. proof of Proposition 1).

Proposition 1. *In every fair-lossy execution E of $A(B)$ holds: If there exists a process $i \in \mathcal{P}$ such that $s_i(k)[Rnd] \leqslant K$ for all k, then $s_j(k)[Rnd] \leqslant K+1$ for all k and all $j \in \mathcal{P}$.*

Proof. By code line 13, i never sends a message of the form (r,m) with $r > K$. By NoGen, no process receives a message of the form (r,m) with $r > K$ from process i. Hence, by lines 2–3, all processes always have $Rcv[i,r] = \perp$ for all $r > K$, and, by lines 5 and 11, do not set *Rnd* to a higher value than $K+1$. ■

Proposition 2. *In every fair-lossy execution E of $A(B)$ holds: If for all $i \in \mathcal{P}$ there exists a k such that $s_i(k)[Rnd] = K$, then for all $i \in \mathcal{P}$ there exists a k' such that $s_i(k')[Rnd] = K+1$.*

Proof. Suppose, by means of contradiction, that there exists some process i such that $s_i(k')[Rnd] \leqslant K$ for all k'. Then by Proposition 1, $s_j(k')[Rnd] \leqslant K+1$ for all k' and all $j \in \mathcal{P}$. Hence by code line 13 and the facts that every process $j \in \mathcal{P}$ has $Rnd = K$ in one of its steps and takes infinitely many steps, it follows that every process sends a message of the form (r,m) infinitely often where $r \in \{K, K+1\}$. By FairLoss, all of these messages are received at least once. Then, by code line 13 and 2–3, process i has $Rcv[j,K] \neq \perp$ for all processes $j \in \mathcal{P}$ during some step of the execution. But then, by code line 11, also $Rnd = K+1$. Contradiction. ■

Proposition 3. *In every fair-lossy execution E of $A(B)$, for every process $i \in \mathcal{P}$, the sequence $s_i(k)[Rnd]$ is unbounded as $k \to \infty$.*

Proof. This is an immediate consequence of Proposition 2. ■

From Propositions 1–3 we immediately obtain the correctness of the retransmission scheme:

Theorem 6. *For every algorithm B, algorithm $A(B)$ simulates B in perfect rounds on top of fair-lossy executions.*

Proof. It remains to show that (S3a) $E \restriction B$ is an execution of B and (S3b) $E \restriction B$ is perfect whenever E is fair-lossy. Property (S3a) follows from code lines 4, 7, and 8–10. Property (S3b) follows from code line 5 and Proposition 3. ■

4.2 Round Durations under Probabilistic Message Loss

We have presented a simple algorithm to simulate perfect rounds on top of fair-lossy executions. In the rest of this chapter, we analyze the performance of this solution.

In a fair-lossy execution E of algorithm $A(B)$, we define the *start of simulated round r* at process i, denoted by $T_i(r)$, to be the number of the step in $E(i)$ in which the state change from $Rnd = r-1$ to $Rnd = r$ was triggered; formally, $T_i(r) = k$ if $E(i) = s_i(1), e_i(2), s_i(2), \ldots$ and k is the smallest index such that $s_i(k)[Rnd] = r$. $L(r)$ is the number of the step where the last process starts its simulated round r, i.e., $L(r) = \max_i T_i(r)$. The *duration of (simulated) round r* at process i is $T_i(r+1) - T_i(r)$, that is, we measure the round duration in the number of local process steps.

Define the *effective transmission delay* $\delta_{j,i}(r)$ to be the number of tries until process j's simulated round r message is successfully received by i. Formally, for any two processes i and j, let $\delta_{j,i}(r) - 1$ be the smallest number $\ell \geqslant 0$ such that

(D1) process j sends a message m in its $(T_j(r) + \ell)$-th step and

(D2) process i receives m from j in its $(T_j(r) + \ell + 1)$-th step.

We thus obtain the following proposition relating the starts of simulated rounds:

Proposition 4. *Let E be a fair-lossy execution of $A(B)$. For each process i: $T_i(1) = 1$, and for each $r \geqslant 1$:*

$$T_i(r+1) = \max_{1 \leqslant j \leqslant N} \big(T_j(r) + \delta_{j,i}(r)\big) \tag{4.1}$$

Figure 4.1 depicts part of a fair-lossy execution of algorithm $A(B)$. The arrows in the figure indicate the time until the first successful reception of a message sent in round r: The tail of the arrow is located at time $T_i(r)$, when a process i starts round r and thus broadcasts round-r messages for the first time. The head of the arrow marks the smallest

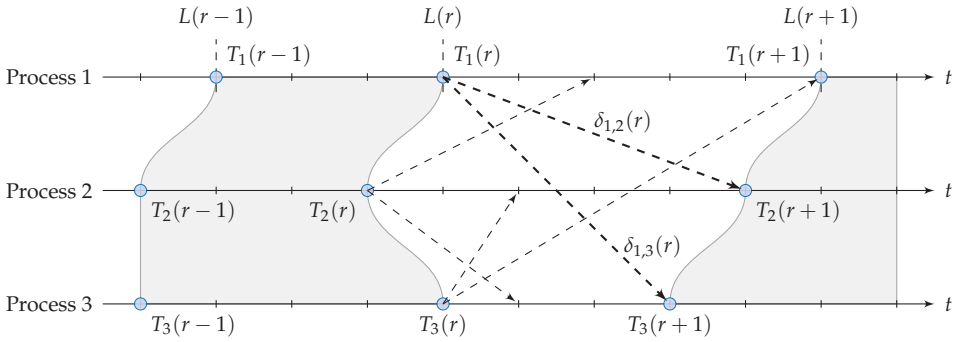


Figure 4.1: Fair-lossy execution of $A(B)$.

time after $T_i(r)$ at which a process j receives a message from i. Messages from processes to themselves are not explicitly shown in the figure. For example, processes 1 and 3 start round r at time $T_{\{1,3\}}(r)$ sending round-r messages for the first time. While process 2 receives the message from 3 in the next step, it needs an overall amount of 4 time steps and consecutive retransmissions to receive a message from process 1.

To allow for a quantitative assessment of the durations of the simulated rounds, besides the trivial bounds of $(0, \infty)$, we extend the modeling of the environment with a probability space: We introduce probability spaces ProbLoss and ProbLoss*, for which we exemplarily calculate the expected average simulated round duration.

For all processes i and j, if process i sends message m in its $(k-1)$-th step, $k > 1$, then process j receives m from i in its k-th step with probability p, where $0 < p \leqslant 1$, is called the *probability of successful transmission*.[1]

Formally, let ProbLoss(p) be the probability distribution on the set of fair-lossy executions defined by: The random variables $\delta_{j,i}(r)$ are pairwise independent, and for any two processes i, j, the probability that $\delta_{j,i}(r) = z$ is $(1-p)^{z-1} \cdot p$. For computational purposes we also introduce the probability distribution ProbLoss(p, M), where $M \in \mathbb{N}^+ \cup \{\infty\}$, which is obtained from ProbLoss(p) by modifying the distribution of the $\delta_{j,i}(r)$: In contrast to ProbLoss(p) we bound the number of tries per simulated round message until it is successfully received by M. Call M the *maximum number of tries per round*.[2] Variable $\delta_{j,i}(r)$ can take values in the set $\{z \in \mathbb{N}^+ \mid 1 \leqslant z \leqslant M\}$. For any two processes i, j, and for integers z with $1 \leqslant z < M$, the probability that $\delta_{j,i}(r) = z$ is $(1-p)^{z-1} \cdot p$. In the remaining cases, i.e., with probability $(1-p)^{M-1}$, $\delta_{j,i}(r) = M$. If $M = \infty$, this case vanishes. In particular, ProbLoss(p, ∞) = ProbLoss(p). The analysis for the case $M = \infty$ is postponed by now but is done in the Sections 4.5–4.7 by using a dual description of the problem.

[1] In systems in which the probability of successful transmission is bounded from below by some $p > 0$, axiom FairLoss holds with probability 1.

[2] In the setting of this thesis this is motivated by using (m, k)-firm deadline scheduling for the messages of an algorithm, i.e., from k consecutive message releases at least m have to meet their deadline, with $m = 1$ and $k = M$. The relaxation of this restriction to $M = \infty$ is analyzed later in this chapter starting with Section 4.5.

In order to describe systems satisfying the realistic assumption PerfComm*, we define ProbLoss$^*(p)$ and ProbLoss$^*(p, M)$ in the same way as ProbLoss(p) and ProbLoss(p, M), except that always $\delta_{i,i}(r) = 1$ for all r and processes i.

We will see in Sections 4.4 and 4.8, that the error we make when calculating the expected duration of the simulated rounds in ProbLoss(p, M) with finite M instead of ProbLoss(p) is small, even for small values of M. It is further shown in these sections that for $M \geqslant 4$, ProbLoss(p, M) is a good approximation of ProbLoss$^*(p, M)$.

Since for each process i and $r \geqslant 1$, it holds that $T_i(r) \leqslant L(r) \leqslant T_i(r+1)$, we obtain the equivalence:

Proposition 5. *If $T_i(r)/r$ converges, then $\lim_{r\to\infty} T_i(r)/r = \lim_{r\to\infty} L(r)/r$.* ■

We can thus reduce the study of the processes' average round durations to the study of the sequence $L(r)/r$ as $r \to \infty$.

4.3 Calculating the Expected Round Duration

The expected round duration of the retransmission algorithm, in the case of fair-lossy executions distributed according to ProbLoss(p, M) or ProbLoss$^*(p, M)$, is determined by introducing an appropriate Markov chain, and analyzing its steady state. To this end, we define a Markov chain $\Lambda(r)$, for an arbitrary round $r \geqslant 1$, that

(1) captures enough of the dynamics of round construction to determine the round durations and

(2) is simple enough to allow efficient computation of each of the process i's *expected round duration* λ_i, defined by $\lambda_i = \mathbb{E} \lim_{r\to\infty} T_i(r)/r$.

Because of Proposition 5, for any two processes i, j it holds that $\lambda_i = \lambda_j = \lambda$, where $\lambda = \mathbb{E} \lim_{r\to\infty} L(r)/r$.

The section is structured as follows: Section 4.3.1 provides the definition of the Markov chain $\Lambda(r)$. Section 4.3.2 develops a method to compute the expected round duration using $\Lambda(r)$. Section 4.4 shows the use of $\Lambda(r)$ by giving several examples. Section 4.4 presents lower bounds of the convergence speed of the round durations. A certain familiarity with basic notions of probability theory is assumed; however, no advanced knowledge is necessary for the comprehension of this section.

4.3.1 Round Durations as a Markov Chain

Markov Chain Facts

A *Markov chain* is a stochastic process, i.e., a sequence $\big(X(r)\big)_{r\geqslant 0}$ of random variables, such that the value of $X(r)$ does not depend on the value of the full history $\big(X(0), X(1), \ldots, X(r-1)\big)$, but only on the value of $X(r-1)$; more formally, $X(r)$'s conditional probability distribution for fixed values of $\big(X(0), \ldots, X(r-1)\big)$ is the same as for the sole fixed value $X(r-1)$.
Given the set $\mathcal{X}$ of possible values for $X(r)$ (its *state space*) and a distribution for $X(0)$, the Markov chain $\big(X(r)\big)$ is fully determined once we fix a *transition probability distribution* P, i.e., a collection $(P_X)_{X\in\mathcal{X}}$ of probability distributions on $\mathcal{X}$. We denote the transition probability from state $Y \in \mathcal{X}$ to state $X \in \mathcal{X}$ by $P_{X,Y}$.
Let $X(r)$ be a Markov chain with state space $\mathcal{X}$. We say that $X(r)$ is *aperiodic* if, for every $X \in \mathcal{X}$, the integers in the set $\{r\colon \mathbb{P}\big(X(r) = X \mid X(0) = X\big) > 0\}$ are relatively prime. We say that $X(r)$ is *irreducible* if for all $X, Y \in \mathcal{X}$, there exists an r such that $\mathbb{P}\big(X(r) = Y \mid X(0) = X\big) > 0$. We say that $X(r)$ is *Harris recurrent* if, for every $X \in \mathcal{X}$, $\mathbb{P}\big(X(r) = X \text{ for infinitely many } r\big) = 1$.

A Markov chain that, by definition, fully captures the dynamics of the round durations is $T(r)$, where $T(r)$ is defined to be the collection of local round finishing times $T_i(r)$ from Equation (4.1). However, directly using Markov chain $T(r)$ for the calculation of λ is impossible since $T_i(r)$, for each process i, grows without bound in r, and thereby its state space is infinite. For this reason we introduce Markov chain $\Lambda(r)$ which optimizes $T(r)$ in two ways and which we use to compute λ: One can achieve a finite state space by considering differences of $T(r)$, instead of $T(r)$; for a process executing algorithm $A(B)$ decides to increment its variable *Rnd* in step k based only on the round numbers it receives in step k and the value of its variable *Rnd* in step $k-1$. Thus the probability that $T(r) = X$ given that $T(r-1) = Y$ is equal to the probability that $T(r) = X - c$ given that $T(r-1) = Y - c$, if $c \in \mathbb{N}$. Choosing $c = L(r-1)$, and observing that $T_i(r) - L(r-1)$ is upper bounded by M, yields a finite state space for finite M, which enables us to calculate the expected round duration.

Also, we do not record the local round finishing times (resp. the difference of local round finishing times) for every of the $\mathcal{N}$ processes, but only record the *number* of processes that are associated a given value. This is feasible because the system is invariant under permutation of processes: The probability that $T(r) = X$ given that $T(r-1) = Y$ is equal to the probability that $T(r) = X'$ given that $T(r-1) = Y'$, where $X'_i = X_{\phi(i)}$ and $Y'_i = Y_{\phi(i)}$ for an arbitrary permutation ϕ of $\mathcal{P}$. This optimization further reduces the size of the state space from $M^{\mathcal{N}}$ to $\binom{\mathcal{N}+M-1}{M-1}$, which is polynomial in $\mathcal{N}$; in practical situations, it suffices to use modest values of M as will be shown in Section 4.8. We show in Theorem 7 that the information recorded in the states of Markov chain $\Lambda(r)$ suffices to determine the expected round duration λ.

We are now ready to formally define $\Lambda(r)$. Its state space $\mathcal{L}$ is defined to be the set of M-tuples $(\sigma_1, \ldots, \sigma_M)$ of nonnegative integers such that $\sum_{z=1}^{M} \sigma_z = \mathcal{N}$. The M-tuples in $\mathcal{L}$ are related to $T(r)$ as follows: Let $\#X$ be the cardinality of the set X, and set

$$\sigma_z(r) = \#\big\{i \mid T_i(r) - L(r-1) = z\big\} \tag{4.2}$$

for $r \geqslant 1$, where we set $L(0) = 0$ to make the case $r = 1$ in Equation (4.2) well-defined. Note that $T_i(r) - L(r-1)$ is always greater than 0, because $\delta_{j,i}(r)$ in Equation (4.1) is greater than 0. Finally, set

$$\Lambda(r) = \big(\sigma_1(r), \ldots, \sigma_M(r)\big) \ . \tag{4.3}$$

The intuition for $\Lambda(r)$ is as follows: For each z, $\sigma_z(r)$ captures the number of processes that start simulated round r, z steps after the last process started the last simulated round, namely $r-1$. For example, in case of the execution depicted in Figure 4.1, $\sigma_1(r) = 0$, $\sigma_2(r) = 1$ and $\sigma_3(r) = 2$. Since algorithm $A(B)$ always waits for the last simulated round message received, and the maximum number of tries until the message is correctly received is bounded by M, we obtain that $\sigma_z(r) = 0$ for $z < 1$ and $z > M$. Knowing $\sigma_z(r)$, for each z with $1 \leqslant z \leqslant M$, thus provides sufficient information

(1) on the processes' states in order to calculate the probability of the next state $\Lambda(r+1) = (\sigma_1, \ldots, \sigma_M)$, and

(2) to determine $L(r+1) - L(r)$ and by this the simulated round duration for the last process.

We first obtain:

Proposition 6. *$\Lambda(r)$ is a Markov chain.*

Proof. On the set of collections (x_i) of numbers indexed by $\mathcal{P} = \{1, 2, \ldots, \mathcal{N}\}$, we introduce equivalence relation $\sim$ by defining $(x_i) \sim (y_i)$ if and only if there exists a bijection $\phi : \mathcal{P} \to \mathcal{P}$ such that $x_i = y_{\phi(i)}$ for every $i \in \mathcal{P}$. We have $(x_i) \sim (y_i)$ if and only if the multisets $\{x_i \mid i \in \mathcal{P}\}$ and $\{y_i \mid i \in \mathcal{P}\}$ are equal. Denote by $\big[(x_i)\big]$ the equivalence class of collection (x_i). Every state $\Lambda \in \mathcal{L}$ naturally corresponds to such an equivalence class.

Let $r > 0$ and $\Lambda_1, \Lambda_2, \ldots, \Lambda_{r-1} \in \mathcal{L}$. We need to show that the conditional distribution for $\Lambda(r)$, given $\Lambda(1), \ldots, \Lambda(r-1) \equiv \Lambda_1, \ldots, \Lambda_{r-1}$, is the same as the conditional distribution for $\Lambda(r)$, given only $\Lambda(r-1) \equiv \Lambda_{r-1}$. By Equations (4.3) and (4.2), it suffices to show that the conditional distributions for $\mathcal{A}(r) = \big[\big(A_i(r)\big)\big]$ where $A_i(r) = T_i(r) - L(r-1)$, are equal.

We claim that the distribution of $\mathcal{A}(r)$ only depends on $\mathcal{B}(r) = \big[\big(B_i(r)\big)\big]$ where $B_i(r) = T_i(r-1) - L(r-1)$. From Equation (4.1) it follows that $A_i(r) = \max_j\big(B_j(r) +$

$\delta_{j,i}(r-1)\big)$. Let $\widetilde{B}(r) \in \mathcal{B}(r)$, i.e., $\widetilde{B}_i(r) = B_{\phi(i)}(r)$ for a bijection $\phi : \mathcal{P} \to \mathcal{P}$ and define $\widetilde{A}_i(r) = \max_j\big(\widetilde{B}_j(r) + \delta_{j,i}(r-1)\big)$. We show that there exists a bijection $\psi : \mathcal{P} \to \mathcal{P}$ such that the distributions for $A_i(r)$ and $\widetilde{A}_{\psi(i)}(r)$ are equal. It suffices to set $\psi = \phi^{-1}$. Then, $\widetilde{A}_{\psi(i)}(r) = \max_j\big(B_{\phi(j)}(r) + \delta_{j,\psi(i)}(r-1)\big) = \max_j\big(B_j(r) + \delta_{\psi(j),\psi(i)}(r-1)\big)$. Since $(j,i) \mapsto \big(\psi(j),\psi(i)\big)$ is a permutation of $\mathcal{P}^2$, and $\delta_{\psi(j),\psi(i)}(r-1)$ and $\delta_{j,i}(r-1)$ are identically distributed for all $(j,i) \in \mathcal{P}^2$, the claim follows.

Equivalence class $\mathcal{B}(r)$, in turn, is completely determined by Λ_{r-1} because of the identity $B_i(r) = A_i(r-1) - \max_j A_j(r-1)$. This concludes the proof. ■

In fact, Proposition 6 holds for a wider class of delay distributions $\delta_{j,i}(r)$, namely those invariant under permutation of processes. Likewise, many results in the remainder of this section are applicable to a wider class of delay distributions: For example, we might drop the independence assumption on the $\delta_{j,i}(r)$ for fixed r and assume strong correlation between the delays, i.e., for each process j and each round r, $\delta_{j,i}(r) = \delta_{j,i'}(r)$ for any two processes i, i'.[3]

Let $X(r)$ be a Markov chain with countable state space $\mathcal{X}$ and transition probabilities P. A probability distribution π on $\mathcal{X}$ is a *stationary distribution* for $X(r)$ if $\pi(X) = \sum_{Y\in\mathcal{X}} \pi(Y) \cdot P_{X,Y}$ for all $X \in \mathcal{X}$. Intuitively, $\pi(X)$ is the asymptotic relative amount of time in which Markov chain $X(r)$ is in state X.

Definition 2. *Call a Markov chain* good *if it is aperiodic, irreducible, Harris recurrent, and has a unique stationary distribution.*

Proposition 7. *$\Lambda(r)$ is a good Markov chain.*

Proof. $\Lambda(r)$ is aperiodic because every state can be reached from every other in two and in three steps with nonzero probability: The transition probability from every state to state $(\mathcal{N}, 0, \ldots, 0)$ is nonzero, for this transition occurs if all messages arrive on their first try. Also, the transition probability from state $(\mathcal{N}, 0, \ldots, 0)$ to every other state is nonzero.

Harris recurrence follows from the fact that every state can be reached in two steps with nonzero probability, together with the fact that the state space is finite.

Existence and uniqueness of the stationary distribution follows from recurrence [MT93, Theorem 10.0.1]. ■

Denote by π the unique stationary distribution of $\Lambda(r)$, which exists because of Proposition 7. Define the function $\sigma : \mathcal{L} \to \mathbb{R}$ by setting $\sigma(\Lambda) = \max\{z \mid \sigma_z \neq 0\}$ where $\Lambda = (\sigma_1, \ldots, \sigma_M) \in \mathcal{L}$. By abuse of notation, we write $\sigma(r)$ instead of $\sigma\big(\Lambda(r)\big)$. From the next proposition it follows that $\sigma(r) = L(r) - L(r-1)$, i.e., knowing $\sigma(1)$ to $\sigma(r)$ suffices to determine $L(r)$. For example, $\sigma(r+1) = 5$ in the execution in Figure 4.1.

[3] This is the case of "negligible transmission delays" considered by Rajsbaum and Sidi [RS94].

Proposition 8. $L(r) = \sum_{k=1}^{r} \sigma(k)$

Proof. The proof is by induction on r. The case $r = 1$ is trivial. We are done if we show $L(r) = L(r-1) + \sigma(r)$ for all $r > 1$. By definition, we have $L(r-1) + \sigma(r) = L(r-1) + \max_i \big(T_i(r) - L(r-1)\big)$. Noting the rule $A + \max_i B_i = \max_i(A + B_i)$ concludes the proof. ■

Proposition 9. *Let $X(r)$ be good Markov chain with state space $\mathcal{X}$ and stationary distribution π. Further, let $g : \mathcal{X} \to \mathbb{R}$ be a function such that $\sum_{X\in\mathcal{X}} |g(X)| \cdot \pi(X) < \infty$. Then,*

$$\lim_{r\to\infty} \frac{1}{r} \sum_{k=1}^{r} g\big(X(k)\big) = \sum_{X\in\mathcal{X}} g(X) \cdot \pi(X)$$

with probability 1 *for every initial distribution.*

Proof. [MT93, Theorem 17.0.1(i)] ■

Proposition 10. *Let $X(r)$ be a good Markov chain with finite state space $\mathcal{X}$ and stationary distribution π. Then there exists some ρ, $0 < \rho < 1$, such that for all $X \in \mathcal{X}$:*

$$\mathbb{P}\big(X(r) = X\big) = \pi(X) + O\left(\rho^r\right)$$

as $r \to \infty$.

Proof. [MT93, Theorem 13.0.1(i)], [MT93, Theorem 16.0.2(iii)] ■

The following theorem is key for calculating the expected simulated round duration λ. We will use the theorem for the computation of λ starting in Section 4.3.2. The theorem states that the simulated round duration averages $L(r)/r$ up to some round r converge to a finite λ almost surely as r goes to infinity. This holds even for $M = \infty$, that is, if no bound is assumed on the number of tries until successful reception of a message. The theorem further relates λ to the steady state of $\Lambda(r)$. Let $\mathcal{L}_z \subseteq \mathcal{L}$ denote the set of states Λ such that $\sigma(\Lambda) = z$. Then:

Theorem 7. *$L(r)/r$ converges to λ with probability* 1*. Furthermore, $\lambda = \sum_{z=1}^{M} z \cdot \pi(\mathcal{L}_z) < \infty$.*

Proof. We use Proposition 9 and prove that its hypothesis holds by showing $\sum_{z\geqslant 1} z \cdot \pi(\mathcal{L}_z) \leqslant 2^{\mathcal{N}^2} p^{-2}$.

As a first step, we show $\pi(\mathcal{L}_z) \leqslant 2^{\mathcal{N}^2}(1-p)^{z-1}$. Because $\mathbb{P}\big(\sigma(r) = z\big)$ converges to $\pi(\mathcal{L}_z)$ as $r \to \infty$ (Proposition 10), it suffices to prove this inequality for $\mathbb{P}\big(\sigma(r) = z\big)$. The event $\sigma(r) = z$ implies the event $\exists i, j : \delta_{i,j}(r) \geqslant z$, i.e., the complement of the event

$\forall i,j : \delta_{i,j}(r) \leqslant z-1$. The events $\delta_{i,j}(r) \leqslant z-1$ each have probability $1-(1-p)^{z-1}$. Hence

$$\mathbb{P}\big(\sigma(r)=z\big) \leqslant 1-\Big(1-(1-p)^{z-1}\Big)^{\mathcal{N}^2} \tag{4.4}$$

for all $r \geqslant 1$.

We now manipulate the right-hand side of Equation (4.4) with operations that preserve the inequality. We invoke the binomial theorem and the triangle inequality, arriving at $\sum_{k=0}^{\mathcal{N}^2}\binom{\mathcal{N}^2}{k}(1-p)^{k(z-1)}$. Finally, we substitute $k(z-1)$ by $z-1$ and use the identity $\sum_k \binom{n}{k} = 2^n$ to prove the claimed inequality $\pi(\mathcal{L}_z) \leqslant 2^{\mathcal{N}^2}(1-p)^{z-1}$.

Using the derivative of the geometric sum formula, we calculate $\sum_{z=0}^{\infty} z(1-p)^{z-1} = 1/p^2$. This concludes the proof. ■

4.3.2 Using $\Lambda(r)$ to Compute λ

We now state a method that, given parameters $M \neq \infty, \mathcal{N}$, and p, computes the expected simulated round duration λ (see Theorem 7). In its core is a standard procedure to compute the stationary distribution of a Markov chain, in form of a matrix inversion. In order to utilize this standard procedure, we need to explicitly state the transition probability distributions $P_{X,Y}$, from each state Y to each state X, which we regard as a matrix P. We will do this using two different assumptions on the communication system:

(i) for the simpler case $\mathsf{ProbLoss}(p, M)$ of a system with probabilistic loop-back links, i.e., where we do not assume that $\mathsf{PerfComm}^*$ holds, and

(ii) for a system $\mathsf{ProbLoss}^*(p, M)$ with the (more realistic) assumption of $\mathsf{PerfComm}^*$.

A first observation, that is valid for both systems, yields that matrix P bears some symmetry, and thus some of the matrix' entries can be reduced to others. In fact we first consider the transition probability from *normalized* Λ states only, that is, $\Lambda = (\sigma_1, \ldots, \sigma_M)$ with $\sigma_M \neq 0$.

In a second step we observe that a non-normalized state Λ can be transformed to a normalized state $\Lambda' = \mathrm{Norm}(\Lambda)$ without changing its outgoing transition probabilities, i.e., for any state X in $\mathcal{L}$, it holds that $P_{X,\Lambda} = P_{X,\Lambda'}$: Thereby Norm is the function $\mathcal{L} \to \mathcal{L}$ defined by:

$$\mathrm{Norm}(\sigma_1, \ldots, \sigma_M) = \begin{cases} (\sigma_1, \ldots, \sigma_M) & \text{if } \sigma_M \neq 0 \\ \mathrm{Norm}(0, \sigma_1, \ldots, \sigma_{M-1}) & \text{otherwise} \end{cases}$$

For example, assuming that $M = 5$, and considering the execution in Figure 4.1, it holds that $\Lambda(r) = (0,1,2,0,0)$. Normalization, that is, right alignment of the last processes, yields $\mathrm{Norm}(\Lambda(r)) = (0,0,0,1,2)$.

Probabilistic loop-back links ProbLoss

For any $\Lambda = (\sigma_1, \ldots, \sigma_M)$ in $\mathcal{L}$ with $\sigma_M \neq 0$, and any $1 \leqslant z \leqslant M$, let $P(\leqslant z \mid \Lambda)$ be the conditional probability that a specific process i is in the set $\{i \mid T_i(r+1) - L(r) \leqslant z\}$, given that $\Lambda(r) = \Lambda$, i.e.,

$$P(\leqslant z \mid \Lambda) = \mathbb{P}(T_i(r+1) - L(r) \leqslant z \mid \Lambda(r) = \Lambda) \ . \tag{4.5}$$

Since the right-hand side is independent of i and r, $P(\leqslant z \mid \Lambda)$ is well-defined. We easily observe that $T_i(r+1) - L(r) \leqslant z$, given that $\Lambda(r) = \Lambda$, if and only if all the following M conditions are fulfilled: For each u, $1 \leqslant u \leqslant M$: for *all* processes j for which $T_j(r) - L(r-1) = u$ (this holds for $\sigma_u(r)$ many) it holds that $\delta_{j,i}(r) \leqslant z + M - u$. Therefore we obtain:

$$P(\leqslant z \mid \Lambda(r)) = \prod_{1 \leqslant u \leqslant M} \mathbb{P}(\delta \leqslant z + M - u)^{\sigma_u(r)} \ , \tag{4.6}$$

for all z, $1 \leqslant z \leqslant M$. Let $P(z \mid \Lambda)$ be the conditional probability that a specific process is in the set $\{i \mid T_i(r+1) - L(r) = z\}$, given that $\Lambda(r) = \Lambda$, i.e.,

$$P(z \mid \Lambda(r)) = \mathbb{P}(T_i(r+1) - L(r) = z \mid \Lambda(r) = \Lambda) \ . \tag{4.7}$$

From Equations (4.5) and (4.7), we immediately obtain:

$$\begin{aligned} &P(1 \mid \Lambda) = P(\leqslant 1 \mid \Lambda) \text{ and,} \\ &P(z \mid \Lambda) = P(\leqslant z \mid \Lambda) - P(\leqslant z-1 \mid \Lambda) \ , \end{aligned} \tag{4.8}$$

for all z, $1 < z \leqslant M$. We may finally state the transition matrix P: for each $X, Y \in \mathcal{L}$, the probability that the system makes a transition from state $Y = \Lambda(r) = (\sigma_1, \ldots, \sigma_M)$ to state $X = \Lambda(r+1) = (\sigma'_1, \ldots, \sigma'_M)$ is given by the probability that of the $\mathcal{N}$ processes, there are σ'_1 processes in the set $\{i \mid T_i(r+1) - L(r) = 1\}$, of the $\mathcal{N} - \sigma'_1$ remaining processes, there are σ'_2 processes in the set $\{i \mid T_i(r+1) - L(r) = 2\}$, etc. Finally, the remaining $\sigma'_M = \mathcal{N} - \sum_{z=1}^{M-1} \sigma'_z$ processes are in the set $\{i \mid T_i(r+1) - L(r) = M\}$. This yields,

$$P_{X,Y} = \binom{\mathcal{N}}{\sigma'_1, \sigma'_2, \ldots, \sigma'_M} \prod_{1 \leqslant z \leqslant M} P(z \mid \mathrm{Norm}(Y))^{\sigma'_z} \ , \tag{4.9}$$

where for any finite sequence $a_1, \ldots, a_m$ with $m \geqslant 1$ and elements from $\mathbb{N}$, the multinomial coefficient $\binom{\sum_{\ell=1}^{m} a_i}{a_1, a_2, \ldots, a_m}$ is equal to $\prod_{1 \leqslant \ell \leqslant m} \binom{\sum_{k=1}^{\ell} a_k}{a_\ell}$, i.e., the number of possibilities to distribute $\sum_{\ell=1}^{m} a_i$ processes into m bins of sizes $a_1, \ldots, a_m$.

Deterministic loop-back links ProbLoss*

Note that for a system where PerfComm* holds, in Equation (4.6), one has the account for the fact that a process i definitely receives its own message after 1 step. In order to specify a transition probability analogous to Equation (4.6), it is thus necessary to know to which of the $\sigma_k(r)$ in $\Lambda(r)$, process i did count for, that is, for which k, $T_i(r) - L(r-1) = k$ holds. We then replace $\sigma_k(r)$ by $\sigma_k(r) - 1$, and keep $\sigma_u(r)$ for $u \neq k$. Formally, let $P(\leqslant z \mid \Lambda, k)$, with $1 \leqslant k \leqslant M$, be the conditional probability that process i is in the set $\{j \mid T_j(r+1) - L(r) \leqslant z\}$, given that $\Lambda(r) = \Lambda$, as well as $T_i(r) - L(r-1) = k$. Then:

$$P(\leqslant z \mid \Lambda(r), k) = \prod_{1 \leqslant u \leqslant M} P(\delta \leqslant z + M - u)^{\sigma_u(r) - \mathbf{1}_{\{k\}}(u)}$$

where $\mathbf{1}_{\{k\}}(u)$ is the indicator function, having value 1 for $u = k$ and 0 otherwise. Equation (4.8) can be generalized in a straightforward manner to obtain expressions for $P(z \mid \Lambda, k)$, i.e., for the conditional probability that process i is in the set $\{i \mid T_i(r+1) - L(r) = z\}$, given that $\Lambda(r) = \Lambda$, as well as $T_i(r) - L(r-1) = k$.

When stating a formula for $P_{X,Y}$ analogous to Equation (4.9), one has to account for the dependency of $P(z \mid \Lambda, k)$ on k. For that purpose let $P_{X,Y}(Q)$, where Q is an $M \times M$ matrix with elements from $\mathbb{N}$, be the transition probability from state $Y = \Lambda(r)$ with $\mathrm{Norm}(Y) = (\sigma_1, \ldots, \sigma_M)$ to state $X = \Lambda(r+1) = (\sigma'_1, \ldots, \sigma'_M)$, provided that $Q_{z,k}$ is the number of processes which are in both $\{i \mid T_i(r+1) - L(r) = z\}$ and $\{i \mid T_i(r) - L(r-1) = k\}$. By definition, $P_{X,Y}(Q)$ is nonzero only if $\sum_{z=1}^{M} Q_{z,k} = \sigma_k$, for $1 \leqslant k \leqslant M$, and $\sum_{k=1}^{M} Q_{z,k} = \sigma'_z$, for $1 \leqslant z \leqslant M$. We readily obtain,

$$P_{X,Y}(Q) = \prod_{1 \leqslant k \leqslant M} \left(\binom{\sigma_k}{Q_{1,k}, Q_{2,k}, \ldots, Q_{M,k}} \prod_{1 \leqslant z \leqslant M} P(z \mid \mathrm{Norm}(Y), k)^{Q_{z,k}} \right) . \quad (4.10)$$

To calculate $P_{X,Y}$ one has to account for all possible choices of Q, each of which occurs with probability $P_{X,Y}(Q)$. With $\mathcal{Q}$ being the set of $M \times M$ matrices with elements from $\mathbb{N}$ for which $\sum_{z=1}^{M} \sum_{k=1}^{M} Q_{z,k} = \mathcal{N}$, we finally obtain

$$P_{X,Y} = \sum_{Q \in \mathcal{Q}} P_{X,Y}(Q) . \quad (4.11)$$

While the calculation of the transition probabilities $P_{X,Y}$ depends on the specific communication assumptions made, the method to obtain λ from the expressions for $P_{X,Y}$ is independent from all these assumptions. It is presented in the following. Let $\Lambda_1, \Lambda_2, \ldots, \Lambda_n$ be any enumeration of states in $\mathcal{L}$. We write $P_{i,j} = P_{\Lambda_i, \Lambda_j}$ and $\pi_i = \pi(\Lambda_i)$ to view P as an $n \times n$ matrix and π as a row vector. By definition, the unique stationary distribution π satisfies

(1) $\pi = \pi \cdot P$,

(2) $\sum_i \pi_i = 1$, and

(3) $\pi_i \geqslant 0$.

It is an elementary linear algebraic fact that these properties suffice to characterize π by the following formula:

$$\pi = e \cdot \left(P^{(n\to 1)} - I^{(n\to 0)}\right)^{-1} \tag{4.12}$$

where $e = (0, \ldots, 0, 1)$, $P^{(n\to 1)}$ is matrix P with its entries in the n-th column set to 1, and $I^{(n\to 0)}$ is the identity matrix with its entries in the n-th column set to 0.

After calculating π, we can use Theorem 7 to finally determine the expected simulated round duration λ. The time complexity of this approach is determined by (T1) building transition matrix P, and (T2) the matrix inversion of P. For both probability spaces (i) $\mathsf{ProbLoss}(p, M)$ and (ii) $\mathsf{ProbLoss}^*(p, M)$, matrix P is of the same size $n \times n$, where $n = \binom{\mathcal{N}+M-1}{M-1}$ is the number of states in the Markov chain $\Lambda(r)$. Thus the time complexity of (T2) is within $\mathcal{O}(n^3)$, which is polynomial in $\mathcal{N}$. With respect to (T1) a naïve implementation of the procedure presented in (ii) has time complexity at least $\#\mathcal{Q} = \binom{\mathcal{N}+M^2-1}{M^2-1}$, which outweighs (T2), in contrast to the method presented in (i).

In Sections 4.4 and 4.8 we show that already small values of M yield good approximations of λ, that quickly converge with growing M. This leads to a tractable time complexity of the proposed method.

4.4 Results for Finite Retransmission Bounds

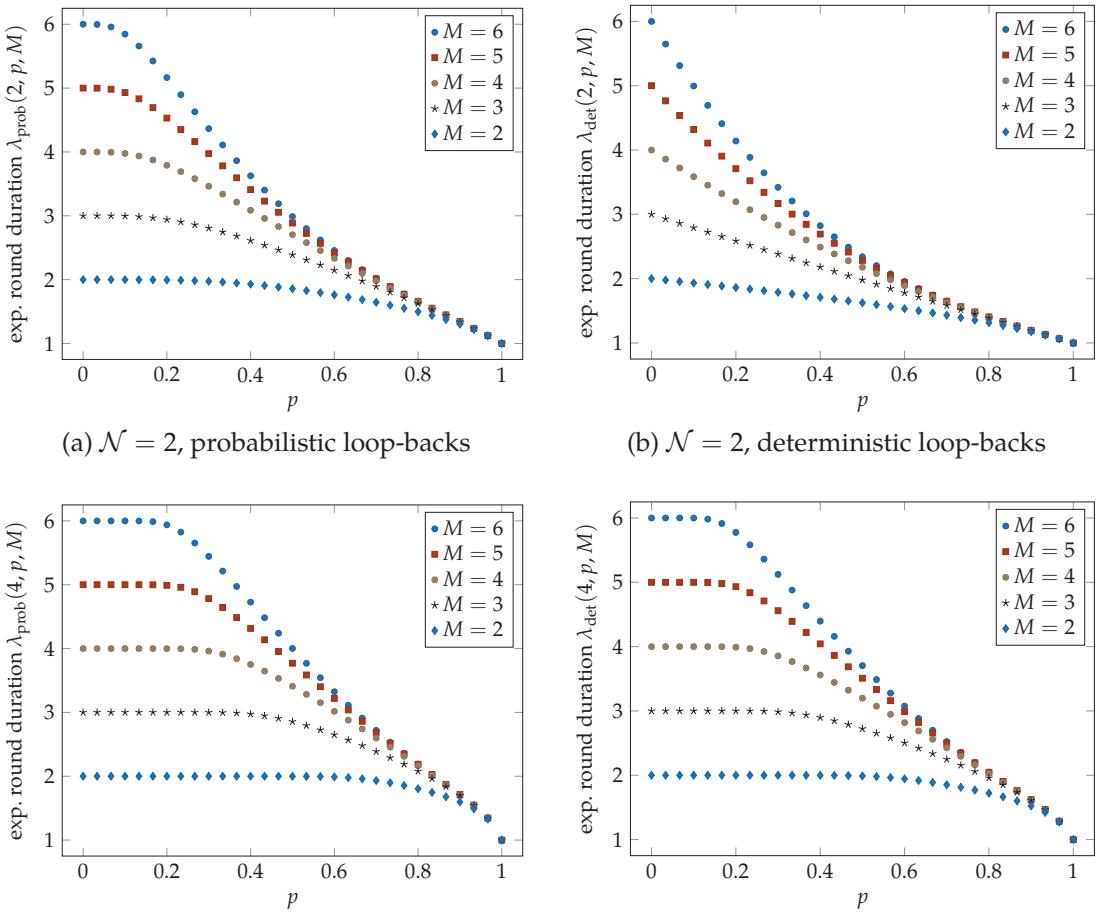
The presented method allows to obtain analytic expressions for λ for fixed M and $\mathcal{N}$ in terms of probability p. Denote by $\lambda_{\text{prob}}(\mathcal{N}, p, M)$ respectively $\lambda_{\text{det}}(\mathcal{N}, p, M)$ the value of λ for probability space $\mathsf{ProbLoss}(p, M)$ respectively $\mathsf{ProbLoss}^*(p, M)$ with $\mathcal{N}$ processes. Figure 4.2 contains $\lambda_{\text{det}}(\mathcal{N}, p, M)$ for $M = 2$ and $\mathcal{N}$ equal to 2 and 3. For larger M and $\mathcal{N}$, the expressions already become significantly longer.

$$\lambda_{\text{det}}(2,p,2) = \frac{6-6p+p^2}{3-2p}$$

$$\lambda_{\text{det}}(3,p,2) = \frac{2-8p+18p^2-16p^3+12p^4+24p^5-64p^6+22p^7+30p^8-22p^9+3p^{10}}{1-4p+9p^2-8p^3+6p^4+12p^5-27p^6+6p^7+12p^8-6p^9}$$

Figure 4.2: Expressions for $\lambda_{\text{det}}(\mathcal{N}, p, M)$ with $M = 2$ and $\mathcal{N} = \{2, 3\}$.

Clearly for all p, M and $\mathcal{N}$, $\lambda_{\text{det}}(\mathcal{N}, p, M)$ is less or equal to $\lambda_{\text{prob}}(\mathcal{N}, p, M)$, since $\mathsf{ProbLoss}$ differs from $\mathsf{ProbLoss}^*$ only by restricting $\delta_{i,i}(r)$ to attain the minimum value of 1 for each process i in each simulated round r. So if one is interested in nontrivial upper bounds of deterministic loop-back systems, probabilistic loop-back systems are a good choice. Figures 4.3a–4.3d even suggest that $\lambda_{\text{prob}}(\mathcal{N}, p, M)$ is a good approximation for $\lambda_{\text{det}}(\mathcal{N}, p, M)$ for $\mathcal{N} \geqslant 4$: Figures 4.3a and 4.3b show solutions of $\lambda_{\text{prob}}(2, p, M)$ and $\lambda_{\text{det}}(2, p, M)$ while Figures 4.3c and 4.3d show solutions for $\lambda_{\text{prob}}(4, p, M)$ and $\lambda_{\text{det}}(4, p, M)$ respectively.



(a) $\mathcal{N} = 2$, probabilistic loop-backs

(b) $\mathcal{N} = 2$, deterministic loop-backs

(c) $\mathcal{N} = 4$, probabilistic loop-backs

(d) $\mathcal{N} = 4$, deterministic loop-backs

Figure 4.3: $\lambda_{\text{prob}}(\mathcal{N}, p, M)$ and $\lambda_{\text{det}}(\mathcal{N}, p, M)$ versus p for $\mathcal{N} = \{2, 4\}$ and $2 \leqslant M \leqslant 6$.

We further observe that for high values of the probability of successful communication p, systems with different M have approximately the same slope. The derivative of λ_{det} for $M = \infty$ and $p \to 1$ is analyzed in Section 4.7.2. Since real distributed systems typically have a high p value, we may approximate $\lambda_{\text{det}}(\mathcal{N}, p, M)$ as well as $\lambda_{\text{prob}}(\mathcal{N}, p, M)$ for higher M values with that of significantly lower M values. The effect is further investigated in Section 4.8 by means of Monte Carlo simulation.

Rate of Convergence

We know from Theorem 7 that $L(r)/r$ converges to λ. The purpose of this section is to establish results on the rate of this convergence. As a particular result, we will see that also $\sigma(r)$ converges to λ. Our main result of this section will be a lower bound on the probability for the event $|L(r)/r - \lambda| < A$ (Theorem 8). We assume $M < \infty$ in this section.

The first proposition shows exponential convergence of $\sigma(r)$'s expected value to λ. It is the consequence of a standard result in Markov theory.

Proposition 11. *There exists some* ρ, $0 < \rho < 1$, *such that* $\mathbb{E}\,\sigma(r) = \lambda + \mathcal{O}(\rho^r)$ *as* $r \to \infty$.

Proof. By definition of the expected value, $\mathbb{E}\,\sigma(r) = \sum_{z=1}^{M} z \cdot \mathbb{P}\big(\Lambda(r) \in \mathcal{L}_z\big)$. By Proposition 10, it is $\mathbb{P}\big(\Lambda(r) \in \mathcal{L}_z\big) = \pi(\mathcal{L}_z) + \mathcal{O}(\rho^r)$ for some ρ, $0 < \rho < 1$. Combining the two equations yields the claimed formula by Theorem 7. ■

Having established the rate of convergence of $\sigma(r)$, we may conclude something about the rate of convergence of $L(r)/r$, i.e., its averages. However, we do not arrive at exponential convergence of $L(r)/r$ towards λ, but only $\mathcal{O}(r^{-1})$. This can be seen as a consequence of the tendency of averages to even out drastic changes. The mathematical reason for it is that the sum $\sum_{k=1}^{r} \rho^k$ does not tend to zero as $r \to \infty$.

Proposition 12. $\mathbb{E}\,L(r)/r = \lambda + \mathcal{O}(1/r)$ *as* $r \to \infty$.

Proof. By Proposition 8, we have $\mathbb{E}\,L(r)/r = 1/r \sum_{k=1}^{r} \mathbb{E}\,\sigma(k)$. Now, using Proposition 11 and noting that $\sum_{k=1}^{r} \rho^k = \mathcal{O}(1)$ as $r \to \infty$ concludes the proof. ■

Next, we investigate the *variance* of $\sigma(r)$.

Proposition 13. *There exists some* ρ, $0 < \rho < 1$, *such that* $\mathrm{Var}\big(\sigma(r)\big) = \beta - \lambda^2 + \mathcal{O}(\rho^r)$ *as* $r \to \infty$, *where* $\beta = \sum_{z=1}^{M} z^2 \cdot \pi(\mathcal{L}_z)$.

Proof. The proposition follows by the same means as Proposition 11 after using the formula $\mathrm{Var}(X) = \mathbb{E}\,X^2 - (\mathbb{E}X)^2$. ■

The next proposition provides two insights:

(1) As r tends to infinity, the variance of $L(r)/r$ tends to zero; in contrast, the variance of $\sigma(r)$ tends to $\beta - \lambda^2$ (Proposition 13). This is a common phenomenon when considering averages of random variables (cf. Law of Large Numbers).

(2) We show a rate of convergence of $\mathcal{O}(1/r)$ for the variance of $L(r)/r$. This is an improvement over standard Markov theoretic results, which are able to show that the variance is $\mathcal{O}(\log\log r/r)$ [MT93, Theorem 17.0.1(iv)-LIL].

Proposition 14. $\mathrm{Var}\big(L(r)/r\big) = \mathcal{O}(1/r)$ *as* $r \to \infty$.

Proof. We subdivide the proof into a sequence of claims, which we prove separately.

Claim 2. $\mathbb{E}\,\sigma(k) \cdot \sigma(\ell) = \lambda^2 + \mathcal{O}\Big(\rho^{\min(k,\ell-k)}\Big)$ *uniformly for all* $k < \ell$.

By definition of the expected value, $\mathbb{E}\,\sigma(k)\cdot\sigma(\ell)$ is equal to

$$\sum_{z=1}^{M}\sum_{u=1}^{M} z\cdot u\cdot \mathbb{P}\big(\Lambda(k)\in\mathcal{L}_z \wedge \Lambda(\ell)\in\mathcal{L}_u\big)\ . \tag{4.13}$$

But $\mathbb{P}\big(\Lambda(k)\in\mathcal{L}_z \wedge \Lambda(\ell)\in\mathcal{L}_u\big)$ is equal to

$$\sum_{\Lambda\in\mathcal{L}_z} \mathbb{P}\big(\Lambda(k)=\Lambda\big)\cdot\mathbb{P}\big(\Lambda(\ell)\in\mathcal{L}_u \mid \Lambda(k)=\Lambda\big)\ . \tag{4.14}$$

Proposition 10 states that there exists a ρ, $0<\rho<1$ such that $\mathbb{P}\big(\Lambda(k)=\Lambda\big) = \pi(\Lambda)+\mathcal{O}(\rho^k)$ and $\mathbb{P}\big(\Lambda(\ell)\in\mathcal{L}_u \mid \Lambda(k)=\Lambda\big)=\pi(\mathcal{L}_u)+\mathcal{O}(\rho^{\ell-k})$.

Substituting this last equality into (4.14), together with $\pi(\mathcal{L}_z)=\sum_{\Lambda\in\mathcal{L}_z}\pi(\Lambda)$ and Theorem 7, yields that (4.13) is equal to $\lambda^2+\mathcal{O}\left(\rho^{\min(k,\ell-k)}\right)$. We have thus proved Claim 2.

Claim 3. $\mathrm{Cov}\big(\sigma(k),\sigma(\ell)\big)=\mathcal{O}\Big(\rho^{\min(k,\ell-k)}\Big)$ *uniformly for all* $k<\ell$.

This claim follows from the formula $\mathrm{Cov}(X,Y)=\mathbb{E}(X\cdot Y)-\mathbb{E}X\cdot\mathbb{E}Y$, together with Claim 2 and Proposition 11.

Claim 4. $\displaystyle\sum_{1\leqslant k<\ell\leqslant r}\rho^{\min(k,\ell-k)}=\mathcal{O}(r)$

Define $a(k,\ell)=\rho^{\min(k,\ell-k)}$. Denote by $A(r)$ the set of pairs (k,ℓ) such that $1\leqslant k<\ell\leqslant r$. Further define $B(r)$ to be the set of pairs (k,ℓ) in $A(r)$ that satisfy $2k<\ell$ and $C(r)$ to be the set of pairs (k,ℓ) in $A(r)$ that satisfy $2k\geqslant\ell$. It is $A(r)=B(r)\cup C(r)$. For $(k,\ell)\in B(r)$, we have $a(k,\ell)=\rho^k$ and for $(k,\ell)\in C(r)$, we have $a(k,\ell)=\rho^{\ell-k}$.

Hence,

$$\sum_{(k,\ell)\in B(r)} a(k,\ell)\leqslant\sum_{\ell=1}^{r}\sum_{k=1}^{r}\rho^k\ . \tag{4.15}$$

We calculate $\sum_{k=1}^{r}\rho^k=(\rho-\rho^{r+2})/(1-\rho)=\mathcal{O}(1)$, which implies that the right-hand side of (4.15) is $\mathcal{O}(r)$.

Similarly,

$$\sum_{(k,\ell)\in C(r)} a(k,\ell)\leqslant\sum_{k=1}^{r}\sum_{\ell=k+1}^{2k}\rho^{\ell-k}=\sum_{k=1}^{r}\sum_{\ell=1}^{k}\rho^{\ell}\leqslant\sum_{k=1}^{r}\sum_{\ell=1}^{r}\rho^{\ell} \tag{4.16}$$

is also $\mathcal{O}(r)$. This proves Claim 4.

Claim 5. $\mathrm{Var}\big(L(r)/r\big)=\mathcal{O}(1/r)$

We use the formulas $\mathrm{Var}\big(\sum_i X_i\big) = \sum_i \mathrm{Var}(X_i) + 2\sum_{i<j}\mathrm{Cov}(X_i, X_j)$ and $\mathrm{Var}(aX) = a^2 \cdot \mathrm{Var}(X)$, which, together with Proposition 13 and Claims 3 and 4, implies Claim 5. This concludes the proof. ■

We can utilize the acquired knowledge about expected value and variance of $L(r)/r$ to explicitly state an asymptotic lower bound on the probability that $L(r)/r$ has distance at most α to the expected value λ. This is a standard procedure and uses Chebyshev's inequality, which can be stated as

$$\mathbb{P}\big(|X - \mathbb{E}X| \geqslant A\big) \leqslant (\mathrm{Var}\, X)^2 / A^2 \ . \tag{4.17}$$

In our case, however, we do not have *one* random variable, but countably many. Thus, we do not limit ourselves to considering a single constant A, but we allow a sequence α_r instead of A. The case of a constant is a particular case.

Theorem 8. *If $M < \infty$ and $\alpha_r \cdot r \to \infty$ as $r \to \infty$, then*

$$\mathbb{P}\left(|L(r)/r - \lambda| \geqslant \alpha_r\right) = \mathcal{O}\big(1/r^2\alpha_r^2\big)$$

as $r \to \infty$.

Proof. Let $\mathbb{E}\, L(r)/r = \lambda + g_r$. Then, by Proposition 12, we have $g_r = \mathcal{O}(1/r)$. The condition $|L(r)/r - \lambda| \geqslant \alpha_r$ is equivalent to $|L(r)/r - \lambda| - |g_r| \geqslant \alpha_r - |g_r|$, which, by the triangle inequality, implies $|L(r)/r - (\lambda + g_r)| \geqslant \alpha_r - |g_r|$.

Hence, $\mathbb{P}\big(\,|L(r)/r - \lambda| \geqslant \alpha_r\big)$ is less or equal to $\mathbb{P}\big(\,|L(r)/r - (\lambda + g_r)| \geqslant \alpha_r - |g_r|\big)$, which, by Chebyshev's inequality (4.17), yields

$$\mathbb{P}\left(|L(r)/r - \lambda| \geqslant \alpha_r\right) \leqslant \frac{\mathrm{Var}(L(r)/r)^2}{(\alpha_r - |g_r|)^2} \ ,$$

which is $\mathcal{O}\big(1/r^2\alpha_r^2\big)$. Here we used Proposition 14 and the fact that $\alpha_r - |g_r| = \Omega(\alpha_r)$, which follows from $g_r = \mathcal{O}(1/r)$ and $\alpha_r \cdot r \to \infty$. ■

Corollary 1. *For all $A > 0$, the probability that $|L(r)/r - \lambda| \geqslant A$ is $\mathcal{O}\big(r^{-2}\big)$.* ■

4.5 Removing the Maximum Retransmission Bound

In the previous sections of this chapter we used the time series $\lim_{r\to\infty} T_i(r)/r$, introduced in Section 4.2, to calculate the expected round duration. As one consequence we had to limit the number of retransmissions until successful reception of a message for being able to compute the steady state of the Markov chain. To overcome this limitation

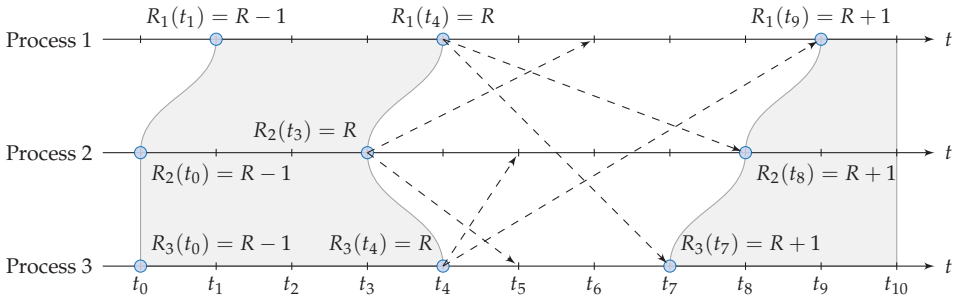


Figure 4.4: Fair-lossy execution of $A(B)$ in the dual space (cf. Figure 4.1).

we can switch to a dual description using $\lim_{t\to\infty} t/R_i(t)$, where $R_i(t)$ denotes the *local round number* of process i at time t, to calculate the expected round duration instead.

The same execution of the simulated algorithm $A(B)$ already shown in Figure 4.1 is depicted in Figure 4.4 using the new model annotations. For the analysis we have to change the system model slightly as unfortunately we cannot stick to the same compact state space as before. Moving to the dual space introduces the need to explicitly store the last received message of every channel within the state of the Markov chain. This makes solving the Markov chain computationally very exhaustive and therefore we will investigate in some complexity reductions measures. By discarding a part of the local state of the processes from time to time, we call it "forgetting", we obtain systems with degraded performance but computationally feasible analysis.

4.5.1 System Model in the Dual Space

In the updated system model, the synchronizer algorithm uses two local variables, specified for every process i at time t: The *local round number* $R_i(t)$ and the *knowledge vector* $\big(K_{i,1}(t), K_{i,2}(t), \ldots, K_{i,N}(t)\big)$. Processes continuously broadcast their local round number. The knowledge vector contains information on other processes' local round numbers, accumulated via received messages. A process increments its local round number, and thereby starts the next round, after it has gained knowledge that all other processes have already started the current round. The round increment rule again assures a precision of 1, i.e., $|R_i(t) - R_j(t)| \leqslant 1$ for all t. Dually to $L(r) = \max_i T_i(r)$, the step number where the last process starts to simulate round r, we define $R_G(t) = \min_i R_i(t)$, the smallest round any process is simulating at time t, and call it the *global round number* at time t.

After updating its local round number, a process may *forget*, i.e., lose its knowledge about other processes' local round numbers. We are considering four different conditions COND, describing the times when process i forgets:

I. Never, i.e., COND := *false*.[4]

II. At every *local round switch*, i.e., COND := $\big[R_i(t) = R_i(t-1) + 1\big]$.

[4]This case corresponds to the original problem introduced earlier by the probability space $\mathsf{ProbLoss}^*(p)$.

III. At every *global round switch*, i.e., COND := $\big[R_G(t) = R_G(t-1)+1\big]$.

IV. Always, i.e., COND := *true*.

Formally, we write $\mathcal{M}_{i,j}(t) = 0$ if process j's message to process i sent at time t was lost, and $\mathcal{M}_{i,j}(t) = 1$ if it arrives (at time $t+1$). Process i's computation in its step at time t consists of the following:

1. *Update knowledge according to received messages:*
   $K_{i,j}(t) \leftarrow R_j(t-1)$ if $\mathcal{M}_{i,j}(t-1) = 1$, and $K_{i,j}(t) \leftarrow K_{i,j}(t-1)$ otherwise.
2. *Increment round number if possible:* $R_i(t) \leftarrow R_i(t-1)+1$ if $K_{i,j}(t) \geqslant R_i(t-1)$ for all j, and $R_i(t) \leftarrow R_i(t-1)$ otherwise.
3. *Conditional forget:* $K_{i,j}(t) \leftarrow 0$ if COND is true.

Initially, $K_{i,j}(1) = 0$, and no messages are received at time 1. In particular, $R_i(1) = 1$. In the remainder of this chapter, when we refer to $K_{i,j}(t)$, we mean its value after step 3.

We assume that the $\mathcal{M}_{i,j}(t)$ are pairwise independent random variables with

$$\mathbb{P}\big(\mathcal{M}_{i,j}(t) = 1\big) = p \text{ if } i \neq j \text{ and } \mathbb{P}\big(\mathcal{M}_{i,i}(t) = 1\big) = 1 \ . \tag{4.18}$$

Figure 4.4 shows part of an execution for condition I on forgetting, the original version of the problem introduced in Section 4.2 (cf. Figure 4.1 for the figure in the original model). Times are labeled t_0 to t_{10} and directly corresponds to the processes computing steps. Processes 1 and 3 start their local round R at time t_4 while process 2 has already started its local round R at time t_3. Again, the arrows in the figure indicate the time until the first successful reception of a message sent in round R.

Note that we are only analyzing the case of deterministic loop-back links in the dual space. We introduced probabilistic loop-back earlier for the purpose to reduce the analysis complexity. In the current state space deterministic loop-backs can be modeled without increasing the complexity and therefore we stick to the more realistic case.

4.5.2 Performance Measure

For a system with $\mathcal{N}$ processes and probability p of successful transmission, we define the *expected round duration* of process i by $\lambda_i(\mathcal{N}, p) = \mathbb{E}\lim_{t\to\infty} t/R_i(t)$. Since our synchronization algorithm guarantees precision 1, it directly follows that $\lambda_i(\mathcal{N}, p) = \lambda_j(\mathcal{N}, p)$ for any two processes i and j. We will henceforth refer to this common value as $\lambda(\mathcal{N}, p)$, or simply λ if the choice of parameters $\mathcal{N}$ and p is clear from the context. To distinguish the four proposed conditions on forgetting, I to IV, we will write λ^{I}, λ^{II}, λ^{III}, and λ^{IV}, respectively.

Note that the condition in case III cannot be detected locally and thus does not allow for a distributed implementation. We rather use λ^{III} as a bound (cf. Equation (4.19)). For case IV, where processes always forget, and for case III, where processes forget on global round switches, λ can be calculated efficiently with explicit formulas, which

we give in Section 4.6 in Theorems 9 and 10. For the remaining cases, I and II, we compute $\lambda(\mathcal{N}, p)$ by means of a steady state analysis of a finite Markov chain with time complexity exponential in $\mathcal{N}$. We show how to do this in Section 4.7.1. The Markov chain model is also useful to study the behavior of λ, for all four conditions on forgetting, when $p \to 1$ and $p \to 0$. We do this in Sections 4.7.2 and 4.7.3, respectively. We derive explicit lower bounds on λ^{I} and λ^{II} in Section 4.7.4.

Proposition 15. *For all four conditions on forgetting,* $\lambda = \mathbb{E} \lim_{t\to\infty} t/R_i(t) = \mathbb{E} \lim_{r\to\infty} T_i(r)/r$.

Proof. From the equality $T_i(r) = \inf\{t \mid R_i(t) = r\}$ we obtain $R_i(T_i(r)) = r$. It follows that $(T_i(r)/r)_{r\geqslant 1} = (T_i(r)/R_i(T_i(r)))_{r\geqslant 1}$. Since the latter is a subsequence of $(t/R_i(t))_{t\geqslant 1}$, both converge to the same value, which is equal to λ by definition. ■

By comparing $T_i(r)$ for every fixed choice of the sequence $\mathcal{M}_{i,j}(t)$ one can show that

$$\lambda^{\mathrm{I}} \leqslant \lambda^{\mathrm{II}} \leqslant \lambda^{\mathrm{III}} \leqslant \lambda^{\mathrm{IV}} \ . \tag{4.19}$$

4.6 Explicit Formulas for λ^{III} and λ^{IV}

In this section, by elementary probability theory and calculations, we derive explicit formulas for λ^{III} and λ^{IV} in Theorems 9 and 10, respectively. Both use a formula for the expected maximum of geometrically distributed random variables (Proposition 16). For that purpose define for pairwise independent with parameter p geometrically distributed random variables $G_i(p)$

$$\xi(M, p) = \mathbb{E} \max_{1\leqslant i\leqslant M} G_i(p) \ .$$

We will make use of the following well-known proposition [KP93, SR90].

Proposition 16. $\xi(M, p) = \sum_{i=1}^{M} \binom{M}{i} (-1)^i \frac{1}{(1-p)^i - 1}$

Consider case III, i.e., processes forget on global round switches. Initially, all processes i are in round $R_i(1) = 1$, and their knowledge is $K_{i,j}(1) = 0$. Observe that processes switch to round 2 as messages are received. At the time t at which the last process switches to round 2, it holds that (i) all processes i have $R_i(t) = 2$, (ii) all processes have knowledge $K_{i,j}(t) \geqslant 1$ for all j before forgetting, and (iii) all processes forget, since a global round switch occurred, ultimately resulting in $K_{i,j}(t) = 0$. The only difference between the initial state and the state at time t is the constant round number offset $R_i(t) = R_i(1) + 1$. By repeated application of the above arguments we obtain that the system is reset to the initial state modulo a constant offset in round numbers R_i, each time a global round switch occurs. This allows to determine the expected average round duration by analyzing the expected time until the first round switch.

We will now state explicit formulas for the expected round duration in cases III and IV. We will use these formulas in particular in Section 4.7.3 when studying the behavior of λ for $p \to 0$.

Theorem 9. $\lambda^{\mathrm{III}}(\mathcal{N}, p) = \xi(\mathcal{N}(\mathcal{N}-1), p) = \sum_{i=1}^{\mathcal{N}(\mathcal{N}-1)} \binom{\mathcal{N}(\mathcal{N}-1)}{i} \frac{(-1)^i}{(1-p)^i - 1}$

Proof. Recall that the events that process i receives a message from process j at time t are pairwise independent for all i, j and times t. Thus the smallest time t, at which i receives a message from j is geometrically distributed with parameter p. Noting that the first global round switch occurs at time $L(2) = \max_i(T_i(2))$, we obtain

$$\lambda(\mathcal{N}, p) = \mathbb{E} \lim_{r\to\infty} L(r)/r = \mathbb{E}L(2) = \mathbb{E} \max_{1 \leqslant i \leqslant \mathcal{N}(\mathcal{N}-1)} G_i(p)$$

where the G_i are geometrically distributed with parameter p. The theorem now follows from Proposition 16. ■

Theorem 10. $\lambda^{\mathrm{IV}}(\mathcal{N}, p) = \xi(\mathcal{N}, p^{\mathcal{N}-1}) = \sum_{i=1}^{\mathcal{N}} \binom{\mathcal{N}}{i} (-1)^i \frac{1}{(1-p^{\mathcal{N}-1})^i - 1}$

Proof. Observe that the first global round switch occurs at the minimum time t by which each of the processes has received messages from all processes simultaneously; and that $R_i(t) = 2$ as well as $K_{i,j}(t) = 0$ holds at this time. Again the state at time t is identical to the initial state with all round numbers incremented by 1. Repeated application of the above arguments allows to calculate the expected round duration by $\lambda(\mathcal{N}, p) = \mathbb{E}L(2)$. The first time i receives a message from all processes simultaneously is geometrically distributed with parameter $p^{\mathcal{N}-1}$. Since we have $\mathcal{N}$ processes, we take the maximum over $\mathcal{N}$ such geometrically distributed random variables. The theorem now follows from Proposition 16. ■

4.7 Markovian Analysis

Determining λ^{I} and λ^{II}, the expected round duration in the cases that processes never forget or forget at local round switches, is more involved. In the following, we will calculate λ by modeling the system as a finite Markov chain and analyzing its steady state distribution. Additionally, we derive the asymptotic behaviors for $p \to 1$ and for $p \to 0$ from the Markov chain model. As the computation of the chain's steady state distribution is computationally very expensive, we will give analytical lower bounds in Section 4.7.4.

Let $A(t)$ be the sequence of matrices with $A_{i,i}(t) = R_i(t)$ and $A_{i,j}(t) = K_{i,j}(t)$ for $i \neq j$. It is easy to see that $A(t)$ is a Markov chain, i.e., the distribution of $A(t+1)$

depends only on $A(t)$. Since both $R_i(t)$ and $K_{i,j}(t)$ are unbounded, the state space of Markov chain $A(t)$ is infinite. We therefore introduce the sequence of *normalized* states $a(t)$, defined by $A(t) - \min_k A_{k,k}(t)$ cropping negative entries to -1, i.e.,

$$a_{i,j}(t) = \max\left\{A_{i,j}(t) - \min_k A_{k,k}(t)\ ,\ -1\right\}\ . \tag{4.20}$$

Normalized states belong to the finite set $\{-1,0,1\}^{\mathcal{N}\times\mathcal{N}}$.

The sequence of normalized states $a(t)$ is a Markov chain: The probability that $A(t+1) = Y$, given that $A(t) = X$, is equal to the probability that $A(t+1) = Y + c$, given that $A(t) = X + c$. We may thus restrict ourselves without loss of generality to considering the system being in state $X - \min_i(X_{i,i})$ at time t. Further, by the algorithm and the fact that the precision is 1, cropping the entries of $X - \min_i(X_{i,i})$ at -1 does not lead to different transition probabilities: the probability that $A(t+1) = Y$ given that $A(t) = X - \min_i(X_{i,i})$ is equal to the probability that $A(t+1) = Y$ given that $A(t)$ is $X - \min_i(X_{i,i})$ cropped at -1. It follows that $a(t)$ is a finite Markov chain, for the algorithm with any of the four conditions on forgetting.

We will repeatedly need to distinguish whether there is a global round switch at time t or not. Let $\hat{a}(t)$ be the Markov chain obtained from $a(t)$ by adding to each state a an additional flag Step such that $\text{Step}(\hat{a}(t)) = 1$ if there is a global round switch at time t, and 0 otherwise.

4.7.1 Using $\hat{a}(t)$ to Calculate λ

Recall the definition of a *good* Markov chain, i.e., one that is aperiodic, irreducible, Harris recurrent, and has a unique steady state distribution (Definition 2). It is not difficult to see that $\hat{a}(t)$ is good for all four conditions on forgetting. A standard method, given the chain's transition matrix, to compute the steady state distribution is by matrix inversion, as already shown in Equation (4.12)

We next utilize Proposition 9 to obtain that $\lim_{t\to\infty} R_i(t)/t$ converges to a constant with probability 1. We call a processes i a *1-process* in state $\hat{a}$ if $\hat{a}_{i,i} = 1$. Likewise, we call i a *0-process* in $\hat{a}$ if $\hat{a}_{i,i} = 0$. Denote by $\#_{-1}(\hat{a})$ the number of -1 entries in rows of matrix $\hat{a}$ that correspond to 0-processes in $\hat{a}$.

Proposition 17. *For all conditions of forgetting, $R_i(t)/t \to 1/\lambda$ with probability 1 as $t \to \infty$. Furthermore, $\lambda = 1/\left(\sum_{\hat{a}} p^{\#_{-1}(\hat{a})} \cdot \pi(\hat{a})\right)$.*

Proof. It holds that $R_G(t) = \sum_{k=1}^{t} \text{Step}\left(\hat{a}(k)\right)$. By Proposition 9, with probability 1 it holds that:

$$\lim_{t\to\infty} R_i(t)/t = \lim_{t\to\infty} R_G(t)/t = \lim_{t\to\infty} \frac{1}{t}\sum_{k=1}^{t} \text{Step}(\hat{a}(k)) = \sum_{\hat{a}} \text{Step}(\hat{a}) \cdot \pi(\hat{a})\ .$$

Since $\hat{a}(t)$ is a finite Markov chain, the last sum is finite. It follows that $R_i(t)/t$ converges to a constant, say c, with probability 1. Thus $t/R_i(t)$ converges to $1/c$ with probability 1. By definition of λ, it follows that $\lambda = 1/c$. This shows the first part of the proposition.

The second part of the proposition is proved by the following calculation:

$$\begin{aligned}
1/\lambda &= \mathbb{E} \lim_{t\to\infty} R_i(t)/t = \mathbb{E} \lim_{t\to\infty} R_G(t)/t = \mathbb{E} \lim_{t\to\infty} \frac{1}{t} \sum_{k=1}^{t} \text{Step}\left(\hat{a}(k)\right) \\
&= \sum_{\hat{a}} \lim_{t\to\infty} \frac{1}{t} \sum_{k=1}^{t} \mathbb{P}\big(\hat{a}(k-1) = \hat{a}\big) \cdot \mathbb{E}\big(\text{Step}(\hat{a}(k)) \mid \hat{a}(k-1) = \hat{a}\big) \\
&= \sum_{\hat{a}} p^{\#_{-1}(\hat{a})} \lim_{t\to\infty} \frac{1}{t} \sum_{k=1}^{t} \mathbb{P}\big(\hat{a}(k-1) = \hat{a}\big) = \sum_{\hat{a}} p^{\#_{-1}(\hat{a})} \cdot \pi(\hat{a}) \ . \qquad \blacksquare
\end{aligned}$$

4.7.2 Behavior of λ for $p \to 1$

The next theorem provides means to approximate the expected round duration for all conditions on forgetting when messages are successfully received with high probability. Since this is typically the case for real-world systems, it allows to characterize their expected round duration very efficiently.

Theorem 11. *For all four conditions on forgetting,* $\frac{d}{dp}\lambda(\mathcal{N}, p)\big|_{p=1} = -\mathcal{N}(\mathcal{N}-1)$.

Proof. Let $p \in (0,1)$. Let $\pi_{\mathcal{N},p}(\hat{a})$ be the steady state probability of state $\hat{a}$ of Markov chain $\hat{a}(t)$. From Proposition 17, $1/\lambda(\mathcal{N}, p) = \sum_{\hat{a}} p^{\#_{-1}(\hat{a})} \cdot \pi_{\mathcal{N},p}(\hat{a})$. Then

$$\frac{d}{dp} 1/\lambda(\mathcal{N}, p) = \sum_{\hat{a}} \#_{-1}(\hat{a}) \cdot p^{\#_{-1}(\hat{a})-1} \cdot \pi_{\mathcal{N},p}(\hat{a}) + \sum_{\hat{a}} p^{\#_{-1}(\hat{a})} \cdot \frac{d}{dp} \pi_{\mathcal{N},p}(\hat{a}) \ .$$

Evaluation of the derivative at $p = 1$ leads to

$$\frac{d}{dp} 1/\lambda(\mathcal{N}, p)\Big|_{p=1} = \sum_{\hat{a}} \#_{-1}(\hat{a}) \cdot \pi_{\mathcal{N},1}(\hat{a}) + \sum_{\hat{a}} \frac{d}{dp} \pi_{\mathcal{N},p}(\hat{a})\Big|_{p=1} \ .$$

Observe that as p goes to 1, $\pi_{\mathcal{N},p}(\hat{a})$ goes to 0 for all states $\hat{a}$, except for $\hat{a}_0$, the state with 0 in the diagonal, -1 everywhere else, and $\text{Step}(\hat{a}) = 1$. It is $\#_{-1}(\hat{a}_0) = \mathcal{N}(\mathcal{N}-1)$. Moreover, as p goes to 1, $\pi_{\mathcal{N},p}(\hat{a}_0)$ approaches 1. Hence,

$$= \mathcal{N}(\mathcal{N}-1) + \frac{d}{dp} \left(\sum_{\hat{a}} \pi_{\mathcal{N},p}(\hat{a}) \right) \Bigg|_{p=1} = \mathcal{N}(\mathcal{N}-1) + 0 \ ,$$

as the sum of the steady state probabilities over all states a equals 1. The theorem follows from $\frac{d}{dp}\lambda(\mathcal{N}, p)\big|_{p=1} = -\frac{d}{dp} 1/\lambda(\mathcal{N}, p)\big|_{p=1} \cdot \lambda^2(\mathcal{N}, 1)$ and $\lambda(\mathcal{N}, 1) = 1$. $\blacksquare$

4.7.3 Behavior of λ for $p \to 0$

In systems with unreliable communication, in which Theorem 11 is not valuable, the following theorem on the asymptotic behavior of the expected round duration for all our conditions on forgetting, is useful. It turns out that λ^{I}, λ^{II}, and λ^{III} have the same order of growth for $p \to 0$, namely p^{-1}, while λ^{IV} has a higher order of growth.

Theorem 12. *For $p \to 0$, $\lambda^{\mathrm{I}}(\mathcal{N}, p)$, $\lambda^{\mathrm{II}}(\mathcal{N}, p)$ and $\lambda^{\mathrm{III}}(\mathcal{N}, p)$ are in $\Theta(p^{-1})$, and $\lambda^{\mathrm{IV}}(\mathcal{N}, p)$ is in $\Theta(p^{-(\mathcal{N}-1)})$.*

Proof. We first show the statement for λ^{III}. It is $(1-p)^i - 1 = \sum_{j=1}^{i} \binom{i}{j}(-p)^j = \Omega(p)$ for $p \to 0$. Hence by Theorem 9, $\lambda^{\mathrm{III}}(\mathcal{N}, p) = \mathcal{O}(p^{-1})$ for $p \to 0$.

For all conditions on forgetting, all transition probabilities of the Markov chain $\hat{a}(t)$ are polynomials in p. Hence by Equation (4.12), all steady state probabilities $\pi(\hat{a})$ are rational functions in p. Proposition 17 then in particular implies that $\lambda^{\mathrm{I}}(\mathcal{N}, p)$ is also rational in p. Clearly, $\lambda^{\mathrm{I}}(\mathcal{N}, p) \to \infty$ as $p \to 0$. Hence $\lambda^{\mathrm{I}}(\mathcal{N}, p)$ has a pole at $p = 0$ of order at least 1. This implies $\lambda^{\mathrm{I}}(\mathcal{N}, p) = \Omega(p^{-1})$. From the inequalities $\lambda^{\mathrm{I}} \leqslant \lambda^{\mathrm{II}} \leqslant \lambda^{\mathrm{III}}$, the first part of the theorem follows.

To show the asymptotic behavior of $\lambda^{\mathrm{IV}}(\mathcal{N}, p)$, observe that by $(1-p)^i - 1 = -p\sum_{j=1}^{i} \binom{i}{j}(-p)^{j-1} \sim -p \cdot i$ for $p \to 0$ and by Proposition 16, we have

$$p \cdot \xi(M, p) \sim \sum_{i=1}^{M} \binom{M}{i} (-1)^{i+1} \frac{1}{i} \ .$$

As shown in the textbook by Graham et al. [GKP89, (6.72) and (6.73)] this sum equals H_M, denoting the M-th harmonic number. This concludes the proof. ■

4.7.4 Lower Bounds on λ^{I} and λ^{II}

Determining the expected round duration for cases I and II by means of the Markov chain $a(t)$ is computationally intensive, even for small system sizes $\mathcal{N}$. We can, however, compute efficient lower and upper bounds on $\lambda(\mathcal{N}, p)$: For both, case I and II, $\lambda^{\mathrm{III}}(\mathcal{N}, p)$ is an upper bound. We will next derive computationally feasible lower bounds for $\lambda^{\mathrm{I}}(\mathcal{N}, p)$ and $\lambda^{\mathrm{II}}(\mathcal{N}, p)$.

From Propositions 15 and 9 follows, by considering the conditional expectation of $L(r)$:

$$\lambda = \frac{1}{\sum_{\hat{a}} \mathrm{Step}(\hat{a}) \cdot \pi(\hat{a})} \sum_{\hat{a}} \mathrm{Step}(\hat{a}) \cdot \pi(\hat{a}) \cdot \mathbb{E}(L(2) \mid \hat{a}(1) = \hat{a}) \ ,$$

where $\mathbb{E}(L(2) \mid \hat{a}(1) = \hat{a})$ is the expected time until the first global round switch, given that the system initially is in state $\hat{a}$. It holds that $\mathbb{E}(L(2) \mid \hat{a}(1) = \hat{a}) = \xi(\#_{-1}(\hat{a}), p)$.

Let $[n]$ denote the set of states $\hat{a}$ with $\#_{-1}(\hat{a}) = n$ and $\mathrm{Step}(\hat{a}) = 1$, and denote by $\bigcup[n]$ the union of all $[n]$ for $0 \leqslant n \leqslant \mathcal{N}(\mathcal{N}-1)$. Further let $\hat{\pi}(n) = \sum_{\hat{a} \in [n]} \pi(\hat{a}) / (\sum_{\hat{a}} \mathrm{Step}(\hat{a}) \cdot$

$\pi(\hat{a})$). It follows that $\hat{\pi}(n) = 0$ for $n < 2\mathcal{N} - 2$ in case II and $\hat{\pi}(n) = 0$ for $n < \mathcal{N} - 1$ in case I.

The basic idea of the bounds on λ is to bound $\hat{\pi}(n)$. Let $\mathbb{P}(\hat{a} \rightsquigarrow [n])$ be the probability that, given the system is in state $\hat{a}$ at some time t, for the minimum time $t' > t$ at which a global round switch occurs, $\hat{a}(t') \in [n]$. We obtain for $\hat{\pi}(n)$:

$$\begin{aligned}
\hat{\pi}(n) &= \sum_{\hat{a}} \mathrm{Step}(\hat{a}) \cdot \hat{\pi}(\hat{a}) \cdot \mathbb{P}(\hat{a} \rightsquigarrow [n]) = \sum_{\hat{a} \in \bigcup [n]} \hat{\pi}(\hat{a}) \cdot \mathbb{P}(\hat{a} \rightsquigarrow [n]) \\
&= \sum_{\hat{a} \in [n]} \hat{\pi}(\hat{a}) \cdot \mathbb{P}(\hat{a} \rightsquigarrow [n]) + \sum_{\hat{a} \in \bigcup [n] \setminus [n]} \hat{\pi}(\hat{a}) \cdot \mathbb{P}(\hat{a} \rightsquigarrow [n]) \\
&\geqslant \hat{\pi}(n) \min_{\hat{a} \in [n]} \mathbb{P}(\hat{a} \rightsquigarrow [n]) + (1 - \hat{\pi}(n)) \min_{\hat{a} \in \bigcup [n] \setminus [n]} \mathbb{P}(\hat{a} \rightsquigarrow [n]) \\
&\geqslant \hat{\pi}(n) c_n + (1 - \hat{\pi}(n)) d_n
\end{aligned}$$

for c_n, d_n suitably chosen. One can derive valid choices for both parameters for cases I and II by excessive case inspection of transition probabilities for all state equivalence classes $[k]$, $k \geqslant 0$ which we will do in Sections 4.7.5 and 4.7.6.

Partitioning the above sum into a one term from states in $[n]$ to states in $[n]$, and one remaining term, allows us to finally state inequality

$$\hat{\pi}(n) \geqslant \frac{d_n}{1 + d_n - c_n} =: \pi_n \ . \tag{4.21}$$

The resulting lower bounds on $\hat{\pi}(n)$, denoted by π_n^{I} and π_n^{II} for cases I and II respectively, finally yield lower bounds on λ. Since ξ is nondecreasing in its first argument, we can bound $\lambda(\mathcal{N}, p)$ by

$$\left(1 - \sum_{n=\mathcal{N}}^{\mathcal{N}(\mathcal{N}-1)} \pi_n^{\mathrm{I}}\right) \xi(\mathcal{N} - 1, p) + \sum_{n=\mathcal{N}}^{\mathcal{N}(\mathcal{N}-1)} \pi_n^{\mathrm{I}} \xi(n, p) \leqslant \lambda^{\mathrm{I}}(\mathcal{N}, p) \tag{4.22}$$

in case I. For case II we obtain

$$\left(1 - \sum_{n=2\mathcal{N}-1}^{\mathcal{N}(\mathcal{N}-1)} \pi_n^{\mathrm{II}}\right) \xi(2\mathcal{N} - 2, p) + \sum_{n=2\mathcal{N}-1}^{\mathcal{N}(\mathcal{N}-1)} \pi_n^{\mathrm{II}} \xi(n, p) \leqslant \lambda^{\mathrm{II}}(\mathcal{N}, p) \ . \tag{4.23}$$

4.7.5 Lower Bound on Parameters for λ^{II}

We next show how to derive bounds on parameters c_n and d_n, in the following denoted by d_n^{II} and c_n^{II}. From these we obtain bounds on $\pi_{\mathcal{N}(\mathcal{N}-1)}$ from (4.21).

We start our analysis with determining $\pi_{\mathcal{N}(\mathcal{N}-1)}$. Since $\mathbb{P}(\hat{a} \rightsquigarrow [\mathcal{N}(\mathcal{N}-1)])$ is greater than the probability that $\hat{a}(t+1) \in [\mathcal{N}(\mathcal{N}-1)]$, given that $\hat{a}(t) = \hat{a}$, for arbitrary

t, we have $\mathbb{P}(\hat{a} \rightsquigarrow [\mathcal{N}(\mathcal{N}-1)]) \geqslant p^{\#_{-1}(\hat{a})}$. Thus we may choose $c^{\mathrm{II}}_{\mathcal{N}(\mathcal{N}-1)} = p^{\mathcal{N}(\mathcal{N}-1)}$, $d^{\mathrm{II}}_{\mathcal{N}(\mathcal{N}-1)} = p^{\mathcal{N}(\mathcal{N}-1)-1}$ and obtain

$$\pi_{\mathcal{N}(\mathcal{N}-1)} = \frac{p^{\mathcal{N}(\mathcal{N}-1)-1}}{1 + p^{\mathcal{N}(\mathcal{N}-1)-1}(1-p)} \ .$$

Next we turn to the analysis of $\pi_{\mathcal{N}(\mathcal{N}-1)-1}$. Since it is not possible to make a direct transition from a state $\hat{a} \in \bigcup[n]$ to a state in $[\mathcal{N}(\mathcal{N}-1)-1]$, we consider bounds on the probability that the system is in a state within $[\mathcal{N}(\mathcal{N}-1)-1]$ at time $t+2$, given that $\hat{a}(t) = \hat{a}$. Fix in $\hat{a}$ one column j whose all non-diagonal entries equal -1. Clearly such a column must exist, since $\mathrm{Step}(\hat{a}) = 1$. Given that $\hat{a}(t) = \hat{a}$, assume that at time $t+1$, all messages from processes $i \neq j$ to all processes i' with $K_{i',i}(t) = -1$, and one message from process j to some fixed $j' \neq j$, are received. That is, $\mathcal{N}(\mathcal{N}-2) + 2 - \#_0(\hat{a})$ messages are received. Moreover, at time $t+1$, k (up to $\mathcal{N}-3$) of the remaining $\mathcal{N}-2$ message sent by j are received. By construction, $k+2$ of the processes are 1-processes at time $t+1$. For $\hat{a}(t+1) \in [\mathcal{N}(\mathcal{N}-1)-1]$ to hold, it is sufficient that: For all 0-processes i with $\hat{a}_{i,j}(t+1) = -1$, process i must receive a message from j at time $t+2$; exactly one of the messages from a 1-process to a 1-process is received. Since at time $t+1$ there are $(k+2)(k+1)$ messages from 1-processes to 1-processes, we obtain: For all $\hat{a} \in \bigcup[n]$,

$$\begin{aligned}
&\mathbb{P}(\hat{a} \rightsquigarrow [\mathcal{N}(\mathcal{N}-1)-1]) \geqslant \\
&\quad \geqslant \sum_{k=0}^{\mathcal{N}-3} \binom{\mathcal{N}-2}{k} p^{\mathcal{N}(\mathcal{N}-2)+2-\#_0(\hat{a})+k} (1-p)^{\mathcal{N}-2-k} \cdot \\
&\qquad \cdot p^{\mathcal{N}-2-k} \cdot p \cdot (1-p)^{(k+2)(k+1)-1} \cdot ((k+2)(k+1)) \\
&\quad = p^{\mathcal{N}(\mathcal{N}-1)-\#_0(\hat{a})+1} \cdot \\
&\qquad \cdot \sum_{k=0}^{\mathcal{N}-3} \binom{\mathcal{N}-2}{k} ((k+2)(k+1))(1-p)^{\mathcal{N}+k^2+2k-1} \\
&\quad =: \beta(\#_0(\hat{a})) \ .
\end{aligned}$$

So we choose $c^{\mathrm{II}}_{\mathcal{N}(\mathcal{N}-1)-1} = \beta(1)$ and $d^{\mathrm{II}}_{\mathcal{N}(\mathcal{N}-1)-1} = \beta(0)$.

Finally we turn to the analysis of π_n for $n = 2(\mathcal{N}-1) + x$, where $0 \leqslant x \leqslant (\mathcal{N}-2)(\mathcal{N}-1) - 2$. Again we bound $\mathbb{P}(\hat{a} \rightsquigarrow [2(\mathcal{N}-1)+x])$, for $\hat{a} \in \bigcup[n]$, by analyzing the probability that $\hat{a}(t+2) \in [2(\mathcal{N}-1)+x]$, given that $\hat{a}(t) = \hat{a}$. Fix a row j of $\hat{a}$ with T non-diagonal entries equal to 0. Given that $\hat{a}(t) = \hat{a}$, assume that at time $t+1$, all messages to processes $i \neq j$ from all processes i' with $K_{i,i'}(t) \neq 0$ are received. That is, $(\mathcal{N}-1)(\mathcal{N}-1) - \#_0(\hat{a}) + T$ messages are received. Moreover, at time $t+1$, k (up to $\mathcal{N}-T-2$) of the remaining $\mathcal{N}-T-1$ messages to j are received. Hence, all processes different from j are 1-processes at time $t+1$. At time $t+2$ all remaining messages to process j are received. From the $(\mathcal{N}-2)(\mathcal{N}-1)$ messages sent by 1-processes to

1-processes exactly x are not allowed to be received for $\hat{a}(t+2) \in [2(\mathcal{N}-1)+x]$ to hold. Thus, for fixed row j and $\hat{a} \in \bigcup[n]$,

$$\begin{aligned}
&\mathbb{P}(\hat{a} \rightsquigarrow [2(\mathcal{N}-1)+x] \mid \text{row } j) \geqslant \\
&\quad \geqslant \sum_{k=0}^{\mathcal{N}-2-T} \binom{\mathcal{N}-T-1}{k} p^{k+(\mathcal{N}-1)^2-\#_0(\hat{a})+T} \\
&\qquad \cdot (1-p)^{\mathcal{N}-1-k-T} p^{\mathcal{N}-1-k-T} p^{(\mathcal{N}-2)(\mathcal{N}-1)-x} (1-p)^x \\
&\qquad \cdot \binom{(\mathcal{N}-1)(\mathcal{N}-2)}{x} \\
&= \binom{(\mathcal{N}-1)(\mathcal{N}-2)}{x} (1-p)^x p^{\mathcal{N}(\mathcal{N}-1)-\#_0(\hat{a})+(\mathcal{N}-2)(\mathcal{N}-1)-x} \\
&\qquad \cdot \Big((2-p)^{\mathcal{N}-1-T} - 1\Big) \\
&=: \gamma(\#_0(\hat{a}), T, x) \ .
\end{aligned}$$

Note that γ is nonincreasing in its second and third argument. Every state $\hat{a}$ has at least one row with $T = 0$ non-diagonal entries equal to 0. All other rows must have $T \leqslant \mathcal{N}-2$ non-diagonal entries equal to 0, since a row must have at least one entry equal to -1. Thus, we have

$$\begin{aligned}
&\mathbb{P}(\hat{a} \rightsquigarrow [2(\mathcal{N}-1)+x]) \geqslant \\
&\gamma(\#_0(\hat{a}), 0, x) + (\mathcal{N}-1) \cdot \gamma(\#_0(\hat{a}), \mathcal{N}-2, x) =: \tilde{\gamma}(\#_0(\hat{a}), x) \ .
\end{aligned}$$

We thus choose $c^{\mathrm{II}}_{2(\mathcal{N}-1)+x} = \tilde{\gamma}((\mathcal{N}-1)(\mathcal{N}-2)-x, x)$ and $d^{\mathrm{II}}_{2(\mathcal{N}-1)+x} = \tilde{\gamma}(0, x)$.

The lower bound on λ^{II} follows from Equations (4.21) and (4.23).

4.7.6 Lower Bound on Parameters for λ^{I}

In this section we derive bounds on parameters c_n and d_n for cases I, denoted by c^{I}_n and d^{I}_n respectively. From these bounds we obtain analytic lower bounds for π_n and thus on λ^{I}, as detailed in Section 4.7.4.

For $\pi_{\mathcal{N}(\mathcal{N}-1)}$ we may choose $d^{\mathrm{I}}_{\mathcal{N}(\mathcal{N}-1)} = d^{\mathrm{II}}_{\mathcal{N}(\mathcal{N}-1)}$ and $c^{\mathrm{I}}_{\mathcal{N}(\mathcal{N}-1)} = c^{\mathrm{II}}_{\mathcal{N}(\mathcal{N}-1)}$, by the same arguments as in Section 4.7.5.

To determine $\pi_{\mathcal{N}(\mathcal{N}-1)-1}$, we use the same construction as in Section 4.7.5, with the modification that at time $t+2$, exactly one of the $(k+2)(\mathcal{N}-1)$ messages sent by 1-processes is received. We thus obtain for all $\hat{a} \in \bigcup[n]$,

$$\begin{aligned}
&\mathbb{P}(\hat{a} \rightsquigarrow [\mathcal{N}(\mathcal{N}-1)-1]) \geqslant \\
&\quad \geqslant \sum_{k=0}^{\mathcal{N}-3} \binom{\mathcal{N}-2}{k} p^{\mathcal{N}(\mathcal{N}-2)+2-\#_0(\hat{a})+k}(1-p)^{\mathcal{N}-2-k}. \\
&\qquad \cdot p^{\mathcal{N}-2-k} p (1-p)^{(k+2)(\mathcal{N}-1)-1}(k+2)(\mathcal{N}-1) \\
&\quad = p^{\mathcal{N}(\mathcal{N}-1)-\#_0(\hat{a})+1} \sum_{k=0}^{\mathcal{N}-3} \binom{\mathcal{N}-2}{k}(k+2)(\mathcal{N}-1) \\
&\qquad \cdot (1-p)^{\mathcal{N}-2-k+(k+2)(\mathcal{N}-1)-1} \\
&\quad =: \beta'(\#_0(\hat{a})) \ .
\end{aligned}$$

So we choose $c^{\mathrm{I}}_{\mathcal{N}(\mathcal{N}-1)-1} = \beta'(1)$ and $d^{\mathrm{I}}_{\mathcal{N}(\mathcal{N}-1)-1} = \beta'(0)$.

Next consider π_n with $n = (\mathcal{N}-1)+x$, and $0 \leqslant x \leqslant (\mathcal{N}-2)(\mathcal{N}-1)+1$. Choose an arbitrary $\hat{a} \in [n]$. It holds that $\hat{a}$ has at least $\mathcal{N}-1$ non-diagonal entries equal to 0. Now fix a row j with $T \leqslant \mathcal{N}-2$ non-diagonal entries equal to 0. We use the same construction as in case II, but note that at time $t+2$, the messages received by j lead to 0-entries. Thus,

$$\begin{aligned}
&\mathbb{P}(\hat{a} \rightsquigarrow (\mathcal{N}-1)+x \mid \text{row } j) \geqslant \\
&\quad \geqslant \sum_{k=0}^{\mathcal{N}-2-T} \binom{\mathcal{N}-T-1}{k} p^{k+(\mathcal{N}-1)^2-\#_0(\hat{a})+T}. \\
&\qquad \cdot (1-p)^{\mathcal{N}-1-k-T} p^{\mathcal{N}-1-k-T}. \\
&\qquad \cdot p^{\mathcal{N}(\mathcal{N}-1)-(\mathcal{N}-1)-x-(\mathcal{N}-1-k-T)}. \\
&\qquad \cdot (1-p)^{(\mathcal{N}-1)^2-(\mathcal{N}(\mathcal{N}-1)-(\mathcal{N}-1)-x)}. \\
&\qquad \cdot \binom{(\mathcal{N}-1)^2-(\mathcal{N}-1-k-T)}{x} \\
&\quad \geqslant p^{2\mathcal{N}^2-4\mathcal{N}+T-x+2-\#_0(\hat{a})}(1-p)^x \binom{(\mathcal{N}-1)(\mathcal{N}-2)}{x} \\
&\qquad \cdot \sum_{k=0}^{\mathcal{N}-2-T} \binom{\mathcal{N}-T-1}{k} p^k (1-p)^{\mathcal{N}-1-k-T} \\
&\quad = p^{2\mathcal{N}^2-4\mathcal{N}+T-x+2-\#_0(\hat{a})}(1-p)^x \binom{(\mathcal{N}-1)(\mathcal{N}-2)}{x}. \\
&\qquad \cdot (1-p^{\mathcal{N}-1-T}) \\
&\quad =: \gamma(\#_0(\hat{a}), T, x) \ .
\end{aligned}$$

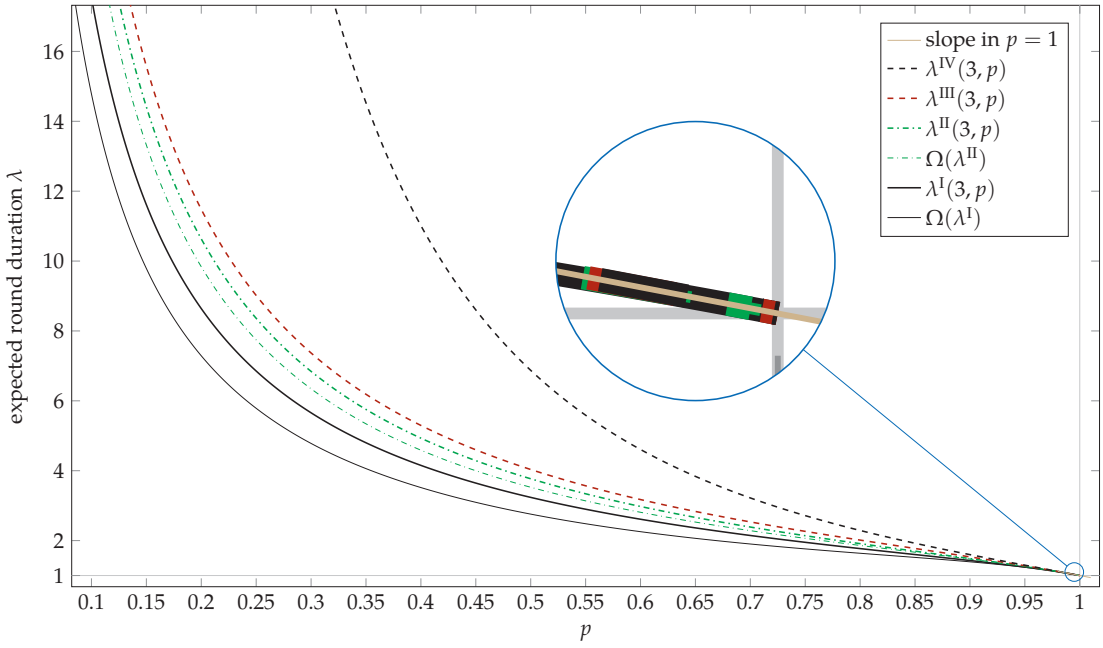


Figure 4.5: Expected round durations for $\mathcal{N} = 3$ and lower bounds for cases I and II.

Note that γ is nonincreasing in its second argument. Every state $\hat{a} \in [n]$ has at least one row with $\lceil n/\mathcal{N} \rceil$ non-diagonal entries equal to -1. Such a row must have $T \leqslant \mathcal{N} - 1 - \lceil n/\mathcal{N} \rceil$ non-diagonal entries equal to 0. Thus, we have

$$\mathbb{P}(\hat{a} \rightsquigarrow [(\mathcal{N} - 1) + x]) \geqslant \gamma(\#_0(\hat{a}), \mathcal{N} - 1 - \lceil n/\mathcal{N} \rceil, x) \ .$$

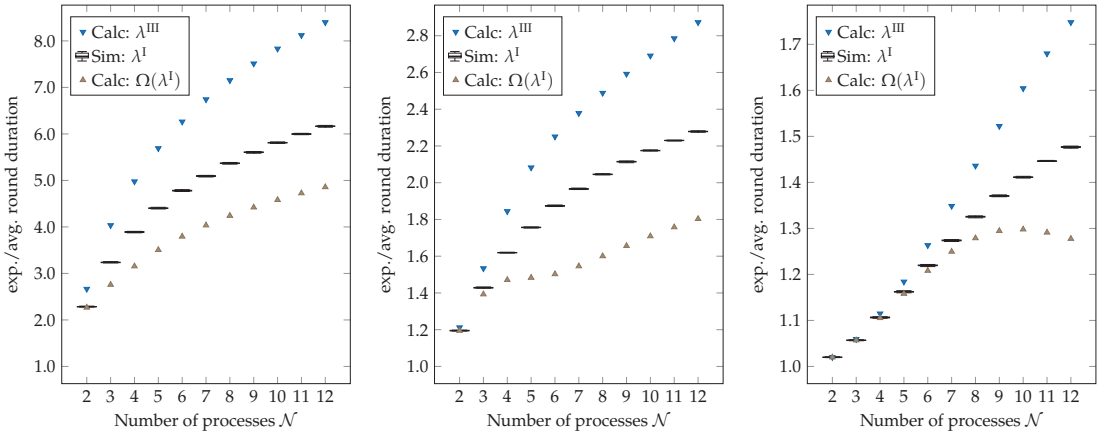
We may thus choose $c_n^{\mathrm{I}} = \gamma((\mathcal{N} - 1)^2 - x, \mathcal{N} - 1 - \lceil n/\mathcal{N} \rceil, x)$ and $d_n^{\mathrm{I}} = \gamma(0, \mathcal{N} - 1 - \lceil n/\mathcal{N} \rceil, x)$.

The lower bound on λ^{I} follows from the Equations (4.22) and letting $\pi_n = 0$ for $\mathcal{N}(\mathcal{N} - 1) - \mathcal{N} + 2 \leqslant n \leqslant \mathcal{N}(\mathcal{N} - 1) - 2$.

4.8 Discussion of Results

In this section we present the results obtained by calculating the expected round duration λ for the four conditions on forgetting that we consider. Additionally, we used Monte-Carlo simulations to estimate λ.

Figure 4.5 shows, with varying probability p, the exact value of the expected round duration for conditions on forgetting I–IV in a system with $\mathcal{N} = 3$ processes. As stated in Section 4.7.3, the figure shows the gap between the cases I, II, and III, having an asymptotic growth in $\Theta(1/p)$ when p approaches 0, and the case IV, which has an asymptotic growth in $\Theta(1/p^{\mathcal{N}-1})$. Furthermore, as depicted in Section 4.7.2, all the



(a) $p = 0.5$ (b) $p = 0.9$ (c) $p = 0.99$

Figure 4.6: Monte-Carlo simulation results for case I compared against the calculated lower bound and the calculated expected round duration of case III serving as an upper bound.

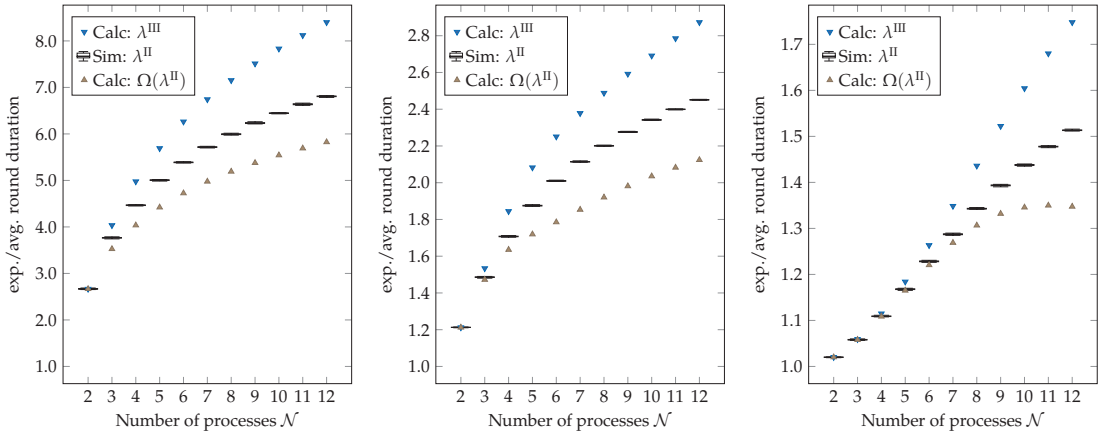
plots have the same slope in the point $p = 1$ resulting in a good approximation for the hard to calculate cases I and II in a system with reliable communication.

In settings with unreliable communication, for which the approximation result on the derivative of λ at $p = 1$ is not valuable, cases I and II can be approximated by their analytical lower bounds (Section 4.7.4), and bounded from above by the λ for case III (Theorem 9). A comparison between the lower bounds and the actual systems is illustrated in Figure 4.5.

Simulations

As the calculations of the exact values for the expected round duration using the Markov chain model are computationally very expensive, we used Monte-Carlo simulations to compare them with our calculations. To this end, we simulated systems with $2 \leqslant \mathcal{N} \leqslant 12$ processes for 100 000 steps and averaged over 30 runs. The simulations were done using three different values for p. Figures 4.6 and 4.7 show the obtained average round durations with the calculated lower bound and with case III as upper bound. The average round durations for case I (where processes never forget) — corresponding to $\lambda_{\mathrm{det}}(p, \mathcal{N})$ — is shown in Figures 4.6a to 4.6c and the case II (where processes forget after a local round switch) is shown in Figures 4.7a to 4.7c. Figures 4.8a to 4.8c depict the calculated expected round duration for case IV, i.e., the synchronizer variant that forgets at each time step. Note that it is significantly higher than all the other variants when message loss is considerable and thus not usable as a bound.

The method presented in Section 4.3.2 allows to calculate $\lambda_{\mathrm{prob}}(\mathcal{N}, p, M)$ as well as $\lambda_{\mathrm{det}}(\mathcal{N}, p, M)$ if $M < \infty$ with reasonable effort. Therefore, the question arises whether the solutions for finite M yield good approximations for $M = \infty$. In this section, we



(a) $p = 0.5$ (b) $p = 0.9$ (c) $p = 0.99$

Figure 4.7: Monte-Carlo simulation results for case II compared against the calculated lower bound and the calculated expected round duration of case III serving as an upper bound.

study the behavior of the random process $T(r)/r$ for increasing r, for different M, with Monte Carlo simulations carried out in Matlab.

In Figure 4.9 we considered the behavior of deterministic loop-back systems with $\mathcal{N} = 5$ processes, for different parameters M and p. The results of the simulation are plotted in Figures 4.9a–4.9c. Each of them includes:

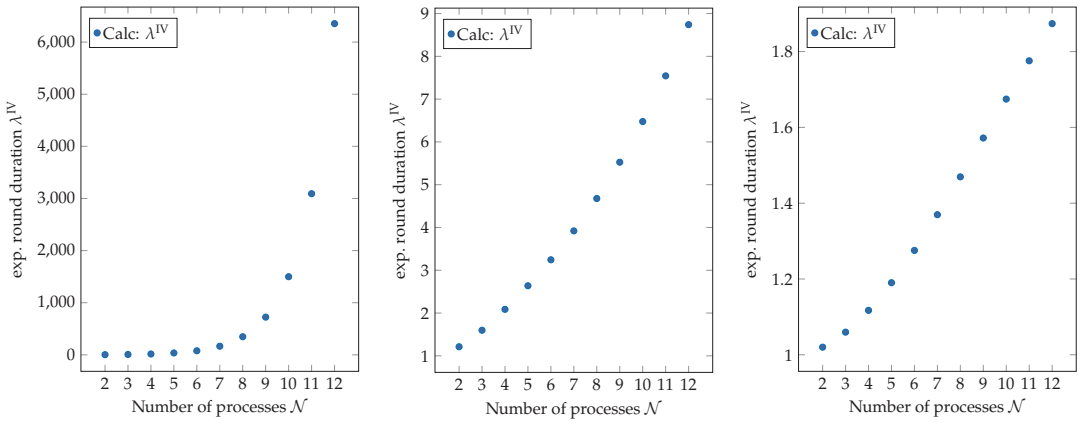
(1) The expected round duration λ_{det}, computed by the method presented in Section 4.3.2 for a deterministic loop-back system with $M = 4$, drawn as a constant function.

(2) The simulation results of sequence $T_1(r)/r$, that is process 1's average round duration, normalized to the calculated λ_{det}, for rounds $1 \leqslant r \leqslant 150$, for two systems: one with parameter $M = 4$, the other with parameter $M = \infty$, both averaged over 1000 runs.

Considering λ_{prob} instead of λ_{det} resulted in similar graphs.

In all three cases, it can be observed that the simulated sequence with parameter $M = 4$ rapidly approximates the theoretically predicted rate for $M = 4$. From the figures we further conclude that calculation of the expected simulated round duration λ for a system with finite, and even small, M already yields good approximations of the expected rate of a system with $M = \infty$ for $p \geqslant 0.75$, while for practically relevant $p \geqslant 0.99$ one cannot distinguish the finite from the infinite case.

In Figure 4.10 we compared the calculated values $\lambda_{\mathrm{prob}}(\mathcal{N}, p, M)$ and $\lambda_{\mathrm{det}}(\mathcal{N}, p, M)$ for $p = 0.5, 0.75, 0.99$, $\mathcal{N} \leqslant 9$, and $M \leqslant 4$ to simulated values of $T_1(1000)/1000$ obtained from 100 Monte Carlo simulations of a deterministic loop-back system with $M = \infty$. The results of the simulation are depicted as box-plots. Note that for $p = 0.75$ the discrepancy between the analytic results for $\lambda_{\mathrm{det}}(\mathcal{N}, p, 4)$ and the simulation results for



(a) $p = 0.5$ (b) $p = 0.9$ (c) $p = 0.99$

Figure 4.8: Calculated expected round duration of case IV.

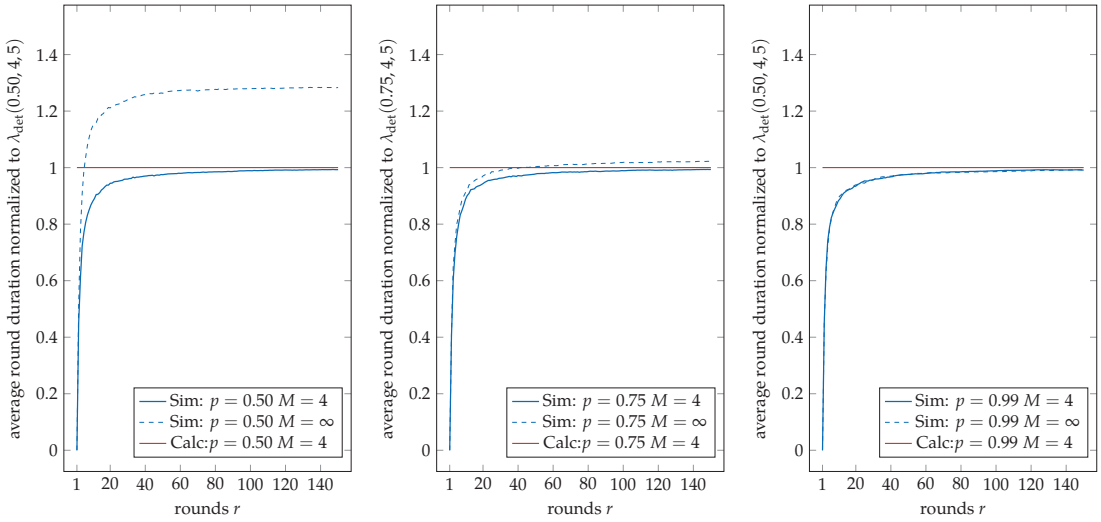
$M = \infty$ is already small, and for $p = 0.99$ the analytic results for all choices of M are in-between the lower quartile and the upper quartile of the simulation results.

Furthermore, the convergence of the average round times from the deterministic loop-back system and the probabilistic loop-back system can be observed, the higher $\mathcal{N}$ gets. Intuitively, this can be explained by the decreasing percentage of the deterministic loop-back links. Since the overall number of links are in $\mathcal{O}(\mathcal{N}^2)$, but the number of deterministic loop-back links is only $\mathcal{O}(\mathcal{N})$. Again, the upper-bound characteristics of probabilistic loop-back systems versus deterministic loop-back systems can be seen.

4.9 Bibliographic Remarks

The notion of simulating a stronger system on top of a weaker one is common in the field of distributed computing [AW04, Part II]. For instance, Neiger and Toueg [NT90] provide an automatic translation technique that turns a synchronous algorithm B that tolerates benign failures into an algorithm $A(B)$ that tolerate more severe failures. Dwork, Lynch, and Stockmeyer [DLS88] use the simulation of a round structure on top of a partially synchronous system, and Charron-Bost and Schiper [CBS09] systematically study simulations of stronger communication axioms in the context of round-based models.

In contrast to randomized algorithms, like Ben-Or's consensus algorithm [BO83], the notion of a probabilistic *environment*, as we use it, is less common in distributed computing: One of the few exceptions is Bakr and Keidar [BK02] who provide practical performance results on distributed algorithms running on the Internet. On the theoretical side, Bracha and Toueg [BT85] consider the Consensus Problem in an environment, for which they assume a nonzero lower bound on the probability that a message m sent from process i to j in round r is correctly received, and that the correct reception of m is independent from the correct reception of a message from i to some process $j' \neq j$

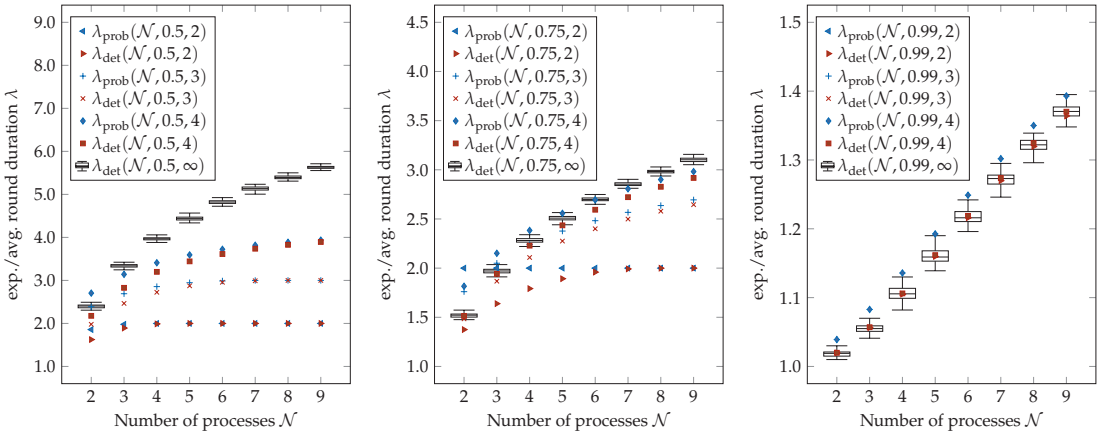


(a) $p = 0.5$ (b) $p = 0.75$ (c) $p = 0.99$

Figure 4.9: Simulated $T_1(r)/r$ versus r for $\mathcal{N} = 5$ and $M = \{4, \infty\}$ in deterministic loop-back systems with $p = \{0.5, 0.75, 0.99\}$, normalized to $\lambda_{\text{det}}(5, p, 4)$.

in the same round r. While we, too, assume independence of correct receptions, we additionally assume a constant probability $p > 0$ of correct transmission, allowing us to derive exact values for the expected round durations of the presented retransmission scheme, which was shown to provide perfect rounds on top of fair-lossy executions. The presented retransmission scheme is based on the α-synchronizer introduced by Awerbuch [Awe85] together with correctness proofs for asynchronous (non-faulty) communication networks of arbitrary structure. However, since Awerbuch did not assume a probability distribution on the message receptions, only trivial bounds on the performance could be stated. Rajsbaum and Sidi [RS94] extended Awerbuch's analysis by assuming message delays to be negligible, and a process i's processing time to be distributed. They consider (1) the general case as well as (2) exponential distribution, and derive performance bounds for (1) and exact values for (2). In terms of our model their assumption translates to assuming maximum positive correlation between message delays: For each (sender) process j and round r, $\delta_{j,i}(r) = \delta_{j,i'}(r)$ for any two (receiver) processes i, i'. They then generalize their approach to the case where $\delta_{j,i}(r)$ comprises a dependent (the processing time) and an independent part (the message delay), and show how to adapt the performance bounds for this case. However, only bounds and no exact performance values are derived for this case. Rajsbaum [Raj94] presented bounds for the case of identical exponential distribution of transmission delays and processing times. Bertsekas and Tsitsiklis [BT89] state bounds for the case of constant processing times and independently, exponentially distributed message delays. However, again, no exact performance values were derived.

Our model comprises negligible processing times and transmission faults, which result in a discrete distribution of the effective transmission delays $\delta_{j,i}(r)$. Interestingly,



(a) $p = 0.5$ (b) $p = 0.75$ (c) $p = 0.99$

Figure 4.10: λ_{prob}, λ_{det} for $M \leqslant 4$ and simulations (deterministic loop-backs, $M = \infty$) versus $\mathcal{N}$ for $p = \{0.5, 0.75, 0.99\}$.

with one sole exception [RdVH$^+$90] which considers the case of a 2-processor system only, we did not find any published results on exact values of the expected round durations in this case. The nontriviality of this problem is indicated by the fact that finding the expected round duration is equivalent to finding the exact value of the *Lyapunov exponent* of a nontrivial stochastic max-plus system [Hei06], which is known to be a hard problem (e.g., [BH00]). In particular, our results can be translated into novel results on stochastic max-plus systems.

The notion of knowledge we introduced differs from that of Fagin et al. [FMHV95], who studied the evolution of knowledge in distributed systems with powerful agents; in particular, their agents do not forget. While Jayaram and Varghese [JV96] use crashing processes and the forgetting during reboot in a destructive way, we use forgetting in a constructive manner.

Chapter 5

Conclusion and Future Work

ANALYZING THE time complexity of an algorithm is at the core of computer science. Algorithms are usually modeled as state machines; when executed in a centralized system, counting the numbers of state transitions until an instance of a problem is solved is a suitable measurement for its time complexity. Unfortunately, centralized systems have an intrinsic problem: they cannot be designed without a single point of failure. This is a significant issue whenever the robustness of a system is of concern. A popular way to overcome this deficiency is to employ space redundancy, by distributing the algorithm among multiple independent processes. In distributed systems using message passing, this processes are distributed among a set of processing nodes connected via a message passing network used for communication.

Synchronous round-based algorithms facilitate a natural time complexity measure by the number of rounds needed to solve a problem. However, when it comes to the actual implementation of such an algorithm in a real-world application, timing uncertainties originating in simplifying assumptions in the analysis like dedicated point-to-point links, no bandwidth limit on the channels, or the assumption of a homogeneous execution environment arise. Dealing with these timing uncertainties, especially when evaluating the performance of an algorithm in the context of Newtonian real-time, was the focus of this thesis.

Furthermore, by distinguishing between a distributed computing node and the distributed algorithm's process, we were able to analyze distributed systems running the processes of multiple independent distributed algorithms concurrently on a shared distributed system. Applying a multi-core argument, i.e., assigning every process of a distributed algorithm its own dedicated processor on a computing node, allowed us to focus our analysis on the communication between the processes which, in this thesis, was assumed to take place via unreliable finite bandwidth channels. For the analysis, we split the problem into two parts.

The transmitter side

On the transmitter side, it might happen that the cumulative bandwidth demand of the executed distributed algorithms on a computing node (temporarily) exceeds the capacity of a communication channel. Therefore, the first part of the thesis covered the performance analysis of message scheduling algorithms at the transmitter side. Messages were modeled as real-time jobs (released at the beginning of the synchronous rounds) with firm deadlines (set to the end of the synchronous rounds). If a firm deadline real-time job does not meet its deadline it does not harm (i.e., it has no severe consequences), but it does not provide any utility to the system, making it a natural match for messages in a synchronous system. A good scheduling algorithm tries to maximize the cumulative utility gained by successfully scheduling released jobs. The performance of an scheduling algorithm is usually evaluated by performing a competitive analysis, i.e., determining the ratio of the cumulative utility of the algorithm with respect to an optimal clairvoyant scheduler that knows the future.

We presented a flexible framework for automated competitive analysis of on-line firm deadline scheduling algorithms for a given taskset. We introduced labeled transition systems as a suitable way to model scheduling algorithms, while additional (optional) restrictions on the adversary-generated job sequences can be specified as safety, liveness, and limit average automata. We gave a reduction of the competitive analysis problem, incorporating all the specified restrictions, to a multi-objective graph problem that can be solved automatically. The algorithmic approach for solving the emerging multi-objective graph problems presented in our solution does not involve human ingenuity at all. This constitutes a significant improvement over previous approaches, which involved considerable effort to analyze worst case scenarios that do not necessarily carry over even to minor variants of the problem.

Whereas computing the competitive ratio of a given on-line algorithm can be done in polynomial time, the utility of our approach for the automated construction of an optimal on-line algorithm and its competitive analysis is currently severely impaired by the exponential time complexity.

Another limitation of our approach, though not relevant in practice, follows from the required finiteness of the state space of the LTS (without which all problems of interest are undecidable). In particular, we cannot deal with an unbounded number of tasks. Consequently, we could not prove $1/4$ to be the competitive factor (i.e., the worst-case utility ratio an on-line scheduler can achieve with respect to an optimal clairvoyant algorithm over all possible tasksets), as this effectively requires countably many tasks [BKM$^+$92]. What we could show, however, is the competitive ratio (i.e., the worst-case utility ratio an on-line scheduler can achieve with respect to an optimal clairvoyant algorithm with respect to a fixed taskset) rapidly approaches $1/4$, by applying our approach to a series of tasksets determined by the recurrence specified in [BKM$^+$92].

Our experimental results demonstrate that our framework allows to solve small-sized problem instances efficiently. Moreover, they highlight the importance of our automated approach, as there is neither a "universally" optimal algorithm for all tasksets

(even in the absence of additional constraints) nor an optimal algorithm for different constraints for the same taskset.

The receiver side

On the receiver side, we are facing message loss not only as a result of the abandoned jobs by the real-time message scheduler at the transmitter, but also by the unreliable communication channel. This issue can either be dealt with by restricting the applicability to executing only omission-tolerant distributed algorithms (therefore, shifting the real-time analysis problem to the algorithm designer), or by introducing a so-called synchronizer that simulates a synchronous round abstraction on top of the imperfect communication. Clearly, in the latter case the real-time performance of a synchronous algorithm running on top of this abstraction layer heavily depends on the performance of the synchronizer. We studied the more general approach of using a synchronizer in this thesis, thus the second part is devoted to the performance analysis of synchronizer variants based on the α-synchronizer introduced by Awerbuch [Awe85].

We used a simplifying approach by assuming that both sources of message loss, the lost messages of the unreliable channel as well as the messages dropped by the real-time scheduler, can be modeled together by a probabilistic link failure model. This abstraction allowed us to calculate the expected round duration using Markov theory.

We analyzed the time-series of the round starting times by modeling them as a Markov chain and calculating its stationary distribution. By initially limiting the number of consecutive transmission faults on a channel by a finite number (i.e., limiting the number of retransmissions of a message that are needed until it is received correctly), we were able to come up with a state space description of the Markov chain whose steady-state analysis is computationally feasible. The assumption can be motivated by using a scheduler with an imposed restriction on the maximum number of drops of consecutive job releases (such as in (m,k)-firm scheduling) and reliable communication links. A comparison to simulations without this limitation suggest that distributed systems restricted to a small number of retransmissions already yield good approximations (with respect to the expected round duration) of the distributed system in which the number of retransmissions until a message is correctly received is not bounded.

Subsequently, we removed the restriction on the number of consecutive transmission faults on a channel and again used a Markov chain to describe this dual system.

As hinted before, the modeling of the state space, and in particular the analysis of the Markov chain's steady-state, get computationally very expensive. We focused on non-trivial lower bounds on the expected round duration and discovered that discarding part of the local memory state of processes as a promising method to reduce the computational complexity of the system, albeit degrading the performance of the synchronizer and thus also of the distributed algorithm running atop.

By parametrizing the synchronizer analysis with the probability of successful message transmission over a channel we are able to state non-trivial formulas for the expected round duration of the distributed algorithms running on top of our framework.

Future work

The presented work sets up the basic framework for a novel way of analyzing distributed systems running more than one synchronous distributed algorithm simultaneously. By now, we assume that every distributed algorithm has its dedicated processor where it is executed. This restriction allows us to neglect the task scheduling of the distributed algorithms and to focus on the message scheduling only. The first interesting research question for future work arises in limiting the number of processors to also enforce the need to incorporate task scheduling in the real-time analysis of the distributed system.

Currently, the applicability of the introduced automated competitive analysis of real-time scheduling is limited to tasksets with rather short tasks, as the graph instances grow considerably with increased relative deadlines as well as laxities of the tasksets. This is an inherent property of the problem and originates in the increasing number of possible interleaving patterns between the released jobs. Furthermore, when considering the labeled transition systems used to model the off-line algorithm, the number of transitions per state grows exponentially with the number of tasks in the tasksets. One idea to cope with this complexity issues in the future is to identify dispensable states as early as possible and remove them. By using an optimized state space representation, the applicability can be increased even further. Combined with abstraction, similarly as it is used in formal verification, and by using symmetry reduction techniques, we expect our approach to also scale to larger instances.

Furthermore, we want to investigate the nature of the worst-case scenarios as our analysis highlights that there is no "universally" optimal algorithm. We expect that a deeper understanding of the worst-case scenarios leads to a more efficient way to synthesize optimal algorithms or at least results in suitable heuristics.

Further extensions of our real-time analysis approach could also incorporate more involved time/utility functions and constraints not only on the adversary but also on the algorithms, e.g., energy restrictions.

One particularly promising possibility for further research is to explore the "linking" assumption that ties together the two parts of our work: Can the dropped messages indeed be modeled as probabilistic link failures, and if so, could our competitive analysis be automated to also provide the loss parameter p? If not, which generalization would provide a better "linking"?

For the round synchronization algorithm, we plan to extend the analysis to a Byzantine fault-tolerant variant (like the one presented in [WS09]) in the future, increasing the applicability of our abstraction model.

Finally, we want to remove the restriction of (virtual) fully-connected communication graphs and investigate different structures of the communication graph, e.g., such as mesh networks.

Bibliography

[AAD+06] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.

[AB98] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, RTSS '98, pages 4–13, December 1998.

[AGS02] Karine Altisen, Gregor Gößler, and Joseph Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Real-Time Systems*, 23(1-2):55–84, 2002.

[AL05] Anant Agarwal and Jeffrey Lang. *Foundations of Analog and Digital Electronic Circuits*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2005.

[AMMMA04] Hakan Aydin, Rami Melhem, Daniel Mossé, and Pedro Mejía-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 53(5):584–600, May 2004.

[Ang80] Dana Angluin. Local and global properties in networks of processors (extended abstract). In *Proceedings of the 12th annual ACM Symposium on Theory of Computing*, STOC '80, pages 82–93, New York, NY, USA, 1980. ACM.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley Series on Parallel and Distributed Computing. Wiley, 2004.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[BBC+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 73–85, New York, NY, USA, 2006. ACM.

[BEN] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. *lpsolve : Open source (Mixed-Integer) Linear Programming system*. Version 5.0.0.0, May 2004.

[BEY98] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[BFK00] Oliver Berthold, Hannes Federrath, and Marit Köhntopp. Project "anonymity and unobservability in the internet". In *Proceedings of the 10th Conference on Computers, Freedom and Privacy: Challenging the assumptions*, CFP '00, pages 57–65, New York, NY, USA, 2000. ACM.

[BH97] Sanjoy K. Baruah and Jayant R. Haritsa. Scheduling for overload in real-time systems. *IEEE Transactions on Computers*, 46:1034–1039, September 1997.

[BH98] Sanjoy K. Baruah and Mary Ellen Hickey. Competitive on-line scheduling of imprecise computations. *IEEE Transactions on Computers*, 47(9):1027–1032, September 1998.

[BH00] François Baccelli and Dohy Hong. Analytic expansions of max-plus lyapunov exponents. *The Annals of Applied Probability*, 10(3):779–827, 08 2000.

[BH09] Martin Biely and Martin Hutle. Consensus when all processes may be byzantine for some time. In *Stabilization, Safety, and Security of Distributed Systems*, volume 5873 of *Lecture Notes in Computer Science*, pages 120–132. Springer Berlin Heidelberg, 2009.

[BHPW07] Martin Biely, Martin Hutle, Lucia D. Penso, and Josef Widder. Relating stabilizing timing assumptions to stabilizing failure detectors regarding solvability and efficiency. In *Stabilization, Safety, and Security of Distributed Systems*, volume 4838 of *Lecture Notes in Computer Science*, pages 4–20. Springer Berlin Heidelberg, 2007.

[BK02] Omar Bakr and Idit Keidar. Evaluating the running time of a communication round over the internet. In *Proceedings of the 21st annual Symposium on Principles of Distributed Computing*, PODC '02, pages 243–252, New York, NY, USA, 2002. ACM.

[BKM+91] Sanjoy Baruah, Gilad Koren, Bhubaneswar Mishra, Arvind Raghunathan, Lou Rosier, and Dennis Shasha. On-line scheduling in the presence of overload. In *Proceedings of the 32nd annual Symposium on Foundations of Computer Science*, FOCS '91, pages 100–110, Oct 1991.

[BKM+92] Sanjoy Baruah, Gilad Koren, Decao Mao, Bhubaneswar Mishra, Arvind Raghunathan, Lou Rosier, Dennis Shasha, and Fuxing Wang. On the competitiveness of on-line real-time task scheduling. *Real-Time Systems*, 4(2):125–144, 1992.

[BMS12] Vincenzo Bonifaci and Alberto Marchetti-Spaccamela. Feasibility analysis of sporadic real-time multiprocessor task systems. *Algorithmica*, 2012.

[BO83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the 2nd annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 27–30, New York, NY, USA, 1983. ACM.

[BSW11] Martin Biely, Ulrich Schmid, and Bettina Weiss. Synchronous consensus under hybrid process and link failures. *Theoretical Computer Science*, 412(40):5602 – 5630, 2011.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[BT89] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Bur80] James E. Burns. A formal model for message passing systems. Technical Report 91, Computer Science Department, Indiana University, Bloomington, September 1980.

[But11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd edition, 2011.

[BV07] Vartika Bhandari and Nitin H. Vaidya. Reliable broadcast in wireless networks with probabilistic failures. In *Proceedings of the 26th IEEE International Conference on Computer Communications*, INFOCOM '07, pages 715–723, May 2007.

[BWCB+07] Martin Biely, Josef Widder, Bernadette Charron-Bost, Antoine Gaillard, Martin Hutle, and André Schiper. Tolerating corrupted communication. In *Proceedings of the 26th annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 244–253, New York, NY, USA, 2007. ACM.

[CBS09] Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[CCJ90] Brent N. Clark, Charles J. Colbourn, and David S. Johnson. Unit disk graphs. *Discrete Mathematics*, 86(1–3):165–177, 1990.

[CDP96] Bogdan S. Chlebus, Krzysztof Diks, and Andrzej Pelc. Broadcasting in synchronous networks with dynamic faults. *Networks (New York, NY)*, 27(4):309–318, 1996. eng.

[CKS13] Krishnendu Chatterjee, Alexander Kößler, and Ulrich Schmid. Automated analysis of real-time scheduling using graph games. In *Proceedings of the 16th ACM international conference on Hybrid Systems: Computation and Control*, HSCC '13, pages 163–172, New York, NY, USA, 2013. ACM.

[CPKS14] Krishnendu Chatterjee, Andreas Pavlogiannis, Alexander Kößler, and Ulrich Schmid. A framework for automated competitive analysis of online scheduling of firm-deadline tasks. RTSS '14, 2014. To be published.

[Cru91] Rene L. Cruz. A calculus for network delay. I. network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, Jan 1991.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[Der74] Michael L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974.

[DGH+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[DKMP95] Krzysztof Diks, Evangelos Kranakis, Adam Malinowski, and Andrzej Pelc. Anonymous wireless rings. *Theoretical Computer Science*, 145(1-2):95 – 109, 1995.

[DLA10] Vinay Devadas, Fei Li, and Hakan Aydin. Competitive analysis of online real-time scheduling algorithms under hard energy constraint. *Real-Time Syst.*, 46(1):88–120, September 2010.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[Dob03] Stefan Dobrev. Communication-efficient broadcasting in complete networks with dynamic faults. *Theory of Computing Systems*, 36(6):695–709, 2003.

[ES09] Robert Elsässer and Thomas Sauerwald. On the runtime and robustness of randomized broadcasting. *Theoretical Computer Science*, 410(36):3414–3427, 2009.

[FFS09] Gottfried Fuchs, Matthias Függer, and Andreas Steininger. On the threat of metastability in an asynchronous fault-tolerant clock generation scheme. In *Proceedings of the 15th IEEE International Symposium on Asynchronous Circuits and Systems*, ASYNC '09, pages 127–136, Chapel Hill, N. Carolina, USA, May 2009.

[FFSS08] Gottfried Fuchs, Matthias Függer, Ulrich Schmid, and Andreas Steininger. Mapping a fault-tolerant distributed algorithm to systems on chip. In *Proceedings of the 11th Euromicro Conference on Digital System Design Architectures, Methods and Tools*, DSD '08, pages 242–249, Parma, Italy, September 2008.

[FKN+13] Matthias Függer, Alexander Kößler, Thomas Nowak, Ulrich Schmid, and Martin Zeiner. The effect of forgetting on the performance of a synchronizer. In *Algorithms for Sensor Systems*, Lecture Notes in Computer Science, pages 185–200. Springer Berlin Heidelberg, 2013.

[FL87] Greg N. Frederickson and Nancy A. Lynch. Electing a leader in a synchronous ring. *Journal of the ACM*, 34(1):98–115, January 1987.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FMHV95] Ronald Fagin, Yoram Moses, Joseph Y. Halpern, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[FP11] Emanuele G. Fusco and Andrzej Pelc. Communication complexity of consensus in anonymous message passing systems. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, volume 7109 of *Lecture Notes in Computer Science*, pages 191–206. Springer-Verlag, Berlin, Heidelberg, 2011.

[FS12] Matthias Függer and Ulrich Schmid. Reconciling fault-tolerant distributed computing and systems-on-chip. *Distributed Computing*, 24(6):323–355, 2012.

[Fü10] Matthias Függer. *Analysis of On-Chip Fault-Tolerant Distributed Algorithms*. PhD thesis, Technische Universität Wien, 2010.

[Gho06] Sukumar Ghosh. *Distributed Systems: An Algorithmic Approach*. Chapman & Hall/CRC Computer & Information Science Series. Taylor & Francis, 2006.

[GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete mathematics: A foundation for computer science*. Addison-Wesley, 1989.

[Gol91] S. Jamaloddin Golestani. A framing strategy for congestion management. *IEEE Journal on Selected Areas in Communications*, 9(7):1064–1077, Sepember 1991.

[HCL90] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. On being optimistic about real-time constraints. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 331–343, New York, NY, USA, 1990. ACM.

[Hei06] Bernd F. Heidergott. *Max-Plus Linear Stochastic Systems and Perturbation Analysis (The International Series on Discrete Event Dynamic Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[HM05] Yuko Hatano and Mehran Mesbahi. Agreement over random networks. *IEEE Transactions on Automatic Control*, 50(11):1867–1872, Nov 2005.

[HS80] Daniel S. Hirschberg and James B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 23(11):627–628, November 1980.

[JKS+13] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *Formal Methods in Computer-Aided Design*, FMCAD '13, pages 201–209. IEEE, Oct 2013.

[JLT85] E. Douglas Jensen, C. Douglass Locke, and Hideyuki Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, pages 112–122. IEEE Computer Society, 1985.

[JV96] Mahesh Jayaram and George Varghese. Crash failures can drive protocols to arbitrary states. In *Proceedings of the 15th annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 247–256, New York, NY, USA, 1996. ACM.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*. Springer US, 1972.

[Kar78] Richard M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 1978.

[Kha79] Leonid G. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR*, 244, 1979.

[KLB04] Santosh Kumar, Ten H. Lai, and József Balogh. On k-coverage in a mostly sleeping sensor network. In *Proceedings of the 10th annual International Conference on Mobile Computing and Networking*, MobiCom '04, pages 144–158, New York, NY, USA, 2004. ACM.

[KM08] Soummya Kar and José M.F. Moura. Sensor networks with random links: Topology design for distributed consensus. *Signal Processing, IEEE Transactions on*, 56(7):3315–3326, July 2008.

[KM09] Soummya Kar and José M.F. Moura. Distributed consensus algorithms in sensor networks with imperfect communication: Link failures and channel noise. *Signal Processing, IEEE Transactions on*, 57(1):355–369, Jan 2009.

[Kop97] Hermann Kopetz. *Real-time Systems: Design Principles for Distributed Embedded Applications*. Kluwer international series in engineering and computer science. Kluwer Academic Publishers, 1997.

[Kou11] Elias Koutsoupias. Scheduling without payments. In *Proceedings of the 4th international conference on Algorithmic game theory*, SAGT'11, pages 143–153, Berlin, Heidelberg, 2011. Springer-Verlag.

[KP93] Peter Kirschenhofer and Helmut Prodinger. A result in order statistics related to probabilistic counting. *Computing*, 51(1):15–27, 1993.

[KS91] Gilad Koren and Dennis Shasha. An optimal scheduling algorithm with a competitive factor for real-time systems. Technical report, New York University, New York, NY, USA, 1991.

[KS95] Gilad Koren and Dennis Shasha. D^{over}: An optimal on-line scheduling algorithm for overloaded uniprocessor real-time systems. *SIAM Journal on Computing*, 24(2):318–339, April 1995.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LJ07] Zhonghai Lu and Axel Jantsch. Admitting and ejecting flits in wormhole-switched networks on chip. *IET Computers & Digital Techniques*, 1, 2007.

[LL73] Chang L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[LL84] Jennifer Lundelius and Nancy A. Lynch. An upperand lower bound for clock synchronization. *Information and Control*, 62:190–204, 1984.

[Loc86] Carey Douglass Locke. *Best-effort Decision-making for Real-time Scheduling*. PhD thesis, CMU, Pittsburgh, PA, USA, 1986.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[Lub86] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal of Computing*, 15(4):1036–1055, November 1986.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 1996.

[Mad02] Omid Madani. Polynomial value iteration algorithms for deterministic MDPs. In *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence*, UAI '02, 2002.

[Mar75] Donald A. Martin. Borel determinacy. *Annals of Mathematics*, 102(2):pp. 363–371, 1975.

[Max65] James Clerk Maxwell. A dynamical theory of the electromagnetic field. *Philosophical Transactions of the Royal Society of London*, 155:459–512, 1865.

[Mos09a] Heinrich Moser. *A model for distributed computing in real-time systems*. PhD thesis, Technische Universität Wien, 2009.

[Mos09b] Heinrich Moser. Towards a real-time distributed computing model. *Theoretical Computer Science*, 410(6–7):629 – 659, 2009. Principles of Distributed Systems.

[MS06] Heinrich Moser and Ulrich Schmid. Optimal clock synchronization revisited: Upper and lower bounds in real-time systems. In *Principles of Distributed Systems*, volume 4305 of *Lecture Notes in Computer Science*, pages 94–109. Springer Berlin Heidelberg, 2006.

[MS14] Heinrich Moser and Ulrich Schmid. Reconciling fault-tolerant distributed algorithms and real-time computing. *Distributed Computing*, 27(3):203–230, 2014.

[MT93] Sean P. Meyn and Richard L. Tweedie. *Markov Chains and Stochastic Stability*. Springer, Heidelberg, 1993.

[NFK13] Thomas Nowak, Matthias Függer, and Alexander Kößler. On the performance of a retransmission-based synchronizer. *Theoretical Computer Science*, 509:25–39, 2013.

[NRTV07] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, New York, NY, USA, 2007.

[NS95] Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal of Computing*, 24(6):1259–1277, December 1995.

[NT90] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, September 1990.

[Pal04] Michael A. Palis. Competitive algorithms for fine-grain real-time scheduling. In *Proceedings of the 25th IEEE Real-Time Systems Symposium*, RTSS '04, pages 129–138. IEEE Computer Society, 2004.

[Pap93] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1993.

[PP07] Andrzej Pelc and David Peleg. Feasibility and complexity of broadcasting with random transmission failures. *Theoretical Computer Science*, 370(1–3):279 – 292, 2007.

[PT86] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, SE-12(3):477–482, 1986.

[Raj94] Sergio Rajsbaum. Upper and lower bounds for stochastic marked graphs. *Information Processing Letters*, 49(6):291 – 295, 1994.

[RdVH+90] J.A.C. Resing, R.E. de Vries, G. Hooghiemstra, M.S. Keane, and G.J. Olsder. Asymptotic behavior of random discrete event systems. *Stochastic Processes and their Applications*, 36(2):195 – 216, 1990.

[RLLS97] Ragunathan Rajkumar, Chen Lee, John Lehoczky, and Dan Siewiorek. A resource allocation model for qos management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS '97, pages 298 –307, 1997.

[RS94] Sergio Rajsbaum and Moshe Sidi. On the performance of synchronized programs in distributed networks with random processing times and transmission delays. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):939–950, 1994.

[Rud74] Sergiu Rudeanu. *Boolean functions and equations*. North-Holland, 1974.

[Sak99] Naoshi Sakamoto. Comparison of initial conditions for distributed algorithms on anonymous networks. In *Proceedings of the 18th annual ACM Symposium on Principles of Distributed Computing*, PODC '99, pages 173–179, 1999.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[Sch01] Ulrich Schmid. How to model link failures: A perception-based fault model. In *International Conference on Dependable Systems and Networks*, DSN '01, pages 57–66, July 2001.

[Sha53] Lloyd S. Shapley. Stochastic games. *Proceedings of the National Academy of Sciences*, 39(10):1095–1100, 1953.

[SP04] Dzulkifli S. Scherber and Haralabos C. Papadopoulos. Locally constructed algorithms for distributed computations in ad-hoc networks. In *Third International Symposium on Information Processing in Sensor Networks*, IPSN '04, pages 11–19, April 2004.

[SR90] Wojciech Szpankowski and Vernon Rego. Yet another application of a binomial recurrence order statistics. *Computing*, 43(4):401–410, 1990.

[SSRB98] John A. Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C. Buttazzo. *Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms*. Real-time systems series. Springer US, 1998.

[ST87] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[Suo13] Jukka Suomela. Survey of local algorithms. *ACM Computing Surveys*, 45(2):24:1–24:40, March 2013.

[SW89] Nicola Santoro and Peter Widmayer. Time is not a healer. In *Proceedings of the 6th annual Symposium on Theoretical Aspects of Computer Science*, STACS '89, pages 304–313, London, UK, 1989.

[SW07] Nicola Santoro and Peter Widmayer. Agreement in synchronous networks with ubiquitous faults. *Theoretical Computer Science*, 384(2–3):232 – 249, 2007.

[SWK09] Ulrich Schmid, Bettina Weiss, and Idit Keidar. Impossibility results and lower bounds for consensus under link failures. *SIAM Journal on Computing*, 38(5):1912–1951, 2009.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[VCD+12] Yaron Velner, Krishnendu Chatterjee, Laurent Doyen, Thomas A. Henzinger, Alexander Rabinovich, and Jean-François Raskin. The complexity of multi-mean-payoff and multi-energy games. *Computing Research Repository (CoRR)*, abs/1209.3234, 2012.

[WS09] Josef Widder and Ulrich Schmid. The theta-model: achieving synchrony without clocks. *Distributed Computing*, 22(1):29–47, 2009.

[YK96] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks: Part I–Characterizing the solvable cases. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):69–89, 1996.

[YK99] Masafumi Yamashita and Tsunehiko Kameda. Leader election problem on networks in which processor identity numbers are not distinct. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):878–887, 1999.

Curriculum Vitae

Alexander Kössler

Personal Information

Date of Birth	February 10, 1983
Place of Birth	Waidhofen an der Ybbs
Address	Wasserburgergasse 5/5, 1090 Wien
Email	koe@itcorner.net

Education

since 04/2009	**Doctoral studies**, *Vienna University of Technology*, Austria.
21/04/2009	Finished master studies with *highest distinction*.
10/2007 – 06/2008	Participation at "TUtheTOP—Das High Potential Programm der TU Wien". *High-potential program.*
03/2007 – 04/2009	**Master studies: Computer Science**, *Vienna University of Technology*, Austria, Degree: Dipl.-Ing.
22/02/2007	Finished bachelor studies with *highest distinction*.
10/2003 – 02/2007	**Bachelor studies: Computer Science**, *Vienna University of Technology*, Austria, Degree: BSc.
09/2002 – 04/2003	Military service.
09/1997 – 06/2002	**Technical High School**, *Höhere Technische Bundeslehranstalt Steyr*, elektronics and computer engineering. Final exam passed with *highest distinction*.

Working Experience

since 05/2013	**Research assistant (Projektassistent)**, *Vienna University of Technology, Embedded Computing Systems Group (Austria)*, funded by FWF.
05/2009 – 04/2013	**Assistant professor (Universitätsassistent)**, *Vienna University of Technology, Embedded Computing Systems Group (Austria)*, university-funded position that includes teaching and research, partly funded by FWF.
05/2008 – 01/2009	**Trainer**, *Berufsförderungsinstitut (BFI) Wien (Austria)*, teaching subjects: mathematics, hardware, and electrical engineering.
02/2008 – 08/2008	**Teaching assistant (Studienassistent)**, *Vienna University of Technology, Department of Computer Engineering (Austria)*, university-funded position.
02/2007 – 08/2007	**Teaching assistant (Studienassistent)**, *Vienna University of Technology, Department of Computer Engineering (Austria)*, university-funded position.
10/2005 – 02/2007	**Scholarship**, *Vienna University of Technology, Department of Computer Engineering (Austria)*, granted for project participation.

Publications

2014	K. Chatterjee, A. Pavlogiannis, A. Kößler, and U. Schmid. A framework for automated competitive analysis of on-line scheduling of firm-deadline tasks. Submitted to the *IEEE Real-Time Systems Symposium (RTSS'14)*.
2013	K. Chatterjee, A. Kößler, and U. Schmid. Automated analysis of real-time scheduling using graph games. In: *Proceedings of the 16th ACM International Conference on Hybrid Systems: Computation and Control (HSCC'13)*, ACM, (2013), pp. 163–172.
	M. Függer, A. Kößler, T. Nowak, U. Schmid, and M. Zeiner. The effect of forgetting on the performance of a synchronizer. In: *Algorithms for Sensor Systems (ALGOSENSORS'13)*, LNCS, (2013), pp. 185–200.
	T. Nowak, M. Függer, and A. Kößler. On the performance of a retransmission-based synchronizer. In: *Theoretical Computer Science*, vol. 509, (2013), pp. 25–39.

2012 M. Függer, A. Kößler, T. Nowak, and M. Zeiner. Brief Announcement: The degrading effect of forgetting on a synchronizer. In: *Stabilization, Safety, and Security of Distributed Systems (SSS'12)*, LNCS, 7596 (2012), pp. 90–91.

2011 T. Nowak, M. Függer, and A. Kößler. On the performance of a retransmission-based synchronizer. In: *Proceedings of the 18th International Colloquium on Structural Information and Communication Complexity, (SIROCCO'11)*, LNCS, vol. 6796, Springer, Heidelberg (2011), pp. 234–245.

2010 A. Kößler, H. Moser, and U. Schmid. Real-time analysis of round-based distributed algorithms. In: *Proceedings of the 1st International Real-Time Scheduling Open Problems Seminar (RTSOPS'10)*, in conjunction with *22nd Euromicro Conference on Real-Time Systems (ECRTS'10)*, (2010).

2009 A. Kößler. A platform for teaching and researching distributed real-time systems. Master Thesis. Technische Universität Wien.

2007 A. Kößler, T. Mair, M. Hofer, and W. Elmenreich. A platform for teaching and research on distributed real-time systems. Poster at the *5th International Conference on Industrial Informatics (INDIN'07)*, Vienna, Austria. *Winner of the best poster award.*

2006 G. Klingler, A. Kößler, and W. Elmenreich. The smart car – a distributed controlled autonomous robot. In: *Proceedings of the Junior Scientist Conference 2006*, (2006), pp. 33–34.

A. Kößler and W. Elmenreich. Automated solution evaluation during a practical examination. In: *Proceedings of the Junior Scientist Conference 2006*, (2006), pp. 35–36.

G. Klingler and A. Kößler. Das 'Smart Car' – ein verteilt kontrollierter, autonomer Roboter. Invited Talk: *more@Informatics*, Vienna, Austria.

2005 W. Elmenreich, G. Klingler, A. Kößler and S. Krywult. Time-triggered smart transducer networks. Poster at the *Siemens PSE Technology Day*, Vienna, Austria.