DISS. ETH NO. 18249

Foundations of
Aggregation and Synchronization
in Distributed Systems

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

THOMAS LOCHER

MSc ETH CS, ETH Zurich

born March 20, 1980

citizen of
Zurich ZH

accepted on the recommendation of

Prof. Dr. Roger Wattenhofer, examiner
Prof. Dr. Nancy Lynch, co-examiner
Prof. Dr. Christian Scheideler, co-examiner

2009

Abstract

A distributed system consists of several autonomous devices that are capable of performing certain (computational) tasks and that have a means to communicate with each other. A computer network system, such as the Internet, is a prototypical example of a distributed system. While a distributed system has many advantages over a single computational unit, e.g., the combined computational power of all entities of a distributed system typically exceeds the power of any single computational device considerably, the decentralized nature of distributed systems also poses significant challenges.

In this thesis, two fundamental problems of distributed systems are studied. The first part of this thesis focuses on the problem of computing global functions that depend on the state of all devices in the system. Since each device stores only a small part of the state of the entire system, interaction between the devices is required in order compute such functions. If the bandwidth of the communication channels is bounded, it may not be an efficient solution to simply encode the state of each entity and forward this information to a single participant in the system, which could then compute the result of the function locally. Instead, the devices may attempt to *aggregate* the data received from other devices in the system and use this information to compute partial solutions of the global function. Such aggregation techniques may greatly reduce the bandwidth consumption when computing global functions in a distributed manner. The goal is to gain a deeper understanding of the complexity of computing global functions using in-network aggregation.

In the second part of this thesis, we consider the problem that several distributed applications and protocols require that all computational devices maintain a common notion of time, but the devices do not have access to a global timer. If each device possesses its own clock, the different clock rates of these clocks necessitate the use of a *clock synchronization algorithm* whose purpose is to compensate for the clock drifts by exchanging timing information and adjusting the clock values according to the received information. Synchronizing clocks is a challenging task mainly due to the uncontrollable and potentially varying message delays, which render it impossible for the devices to determine how accurate the timing information is that they receive from other devices. The objective is thus to analyze the feasible degree of synchronization, which not only depends on the message delays and the clock drift rates, but also on other parameters such as the frequency of communication.

Zusammenfassung

Ein verteiltes System besteht aus mehreren autonomen Komponenten, die Berechnungen ausführen und untereinander kommunizieren können. Ein Computernetzwerk, wie z.B. das Internet, ist ein prototypisches verteiltes System. Verteilte Systeme bieten viele Vorzüge gegenüber einer einzelnen, zentralen Recheneinheit. So übertrifft beispielsweise die gemeinsame Rechenleistung aller Komponenten eines verteilten Systems typischerweise die Leistung einer einzelnen Recheneinheit beträchtlich. Im Gegenzug bringt die Tatsache, dass alle Komponenten des verteilten Systems dezentral agieren, erhebliche Herausforderungen mit sich.

In dieser Dissertation betrachten wir zwei fundamentale Probleme in verteilten Systemen. Der erste Teil der Arbeit untersucht, wie man globale Funktionen, die vom Zustand des gesamten Systems abhängen, verteilt berechnen kann. Da jede Komponente nur einen kleinen Teil des Gesamtzustands des Systems kennt, ist Interaktion zwischen den einzelnen Komponenten notwendig, um solche Funktionen zu berechnen. Wenn die Bandbreite der Kommunikationskanäle beschränkt ist, dann ist es jedoch häufig ineffizient, den Zustand jeder einzelnen Komponente zu einer zentralen Stelle zu senden, welche anschliessend die Funktion lokal berechnen kann. Ein besserer Ansatz ist es stattdessen, Informationen über die Zustände der Komponenten zu *aggregieren* und mit diesen Aggregaten die Funktion zu berechnen, was den Bandbreitenbedarf markant reduzieren kann. Im ersten Teil dieser Arbeit ist das Ziel ein tieferes Verständnis der Komplexität der Berechnung globaler Funktionen mittels Aggregationstechniken zu erlangen.

Der zweite Teil dieser Dissertation befasst sich mit dem Problem, dass viele verteilte Anwendungen und Protokolle eine gemeinsame Vorstellung aller Komponenten des Systems von der momentanen Zeit voraussetzen. Aufgrund der dezentralen Natur verteilter Systeme ist nicht immer möglich, oder erwünscht, dass hierfür eine zentrale (exakte) Uhr herangezogen wird. Wenn jede Komponente eine eigene Uhr besitzt, dann bedarf es eines *Uhrensynchronisationsalgorithmus*, um das Auseinanderdriften der Uhrenwerte aufgrund unterschiedlicher Uhrenraten auszugleichen. Diese Kompensation erfolgt durch das Austauschen von Information über die lokal gemessene Zeit und das allfällige Korrigieren der eigenen Uhrzeit gemäss den empfangenen Werten. Uhrensynchronisation ist ein schwieriges Problem, da die variablen Nachrichtenverzögerungen ein präzises Bestimmen der genauen Uhrzeit anderer Komponenten verunmöglichen. Wir beantworten die Frage, was der bestmögliche Grad an Synchronisation ist, der in einem verteilten System erreicht werden kann. Dies hängt nicht nur von den Nachrichtenverzögerungen ab, sondern auch von weiteren Parametern, wie z.B. der Häufigkeit, mit der kommuniziert wird.

Acknowledgements

First and foremost, I would like to express my gratitude to my advisor Roger Wattenhofer for giving me the opportunity to work in his Distributed Computing Group. I sincerely appreciate that you encouraged me incessantly to study problems that I considered interesting myself and come up with my own ideas, yet you always helped me in thought-provoking discussions when I needed inspiration.

I am further deeply indebted to my co-examiners Nancy Lynch and Christian Scheideler who took time out of their busy schedules to carefully read this thesis and provide valuable feedback, which greatly helped me to improve the presentation of this thesis. Moreover, I particularly have to thank Fabian Kuhn and Christoph Lenzen who not only tremendously helped me to get rid of hopefully all the (more or less severe) errors that crept into my thesis by proof-reading it with meticulous care, but also co-authored many of the publications that contain the results presented in this thesis. I further have to thank Simon Heimlicher for helping me to enrich this thesis with visually appealing graphics. Of course, I have to extend my thanks to my most prolific collaborator Stefan Schmid. It was always a great pleasure to steal bits with you and to enjoy the fruits of our labor by watching a sometimes pretentious but always entertaining movie together whenever one of our papers got accepted for publication. I am also very grateful that I could share my time in this research group with Yvonne Anne Pignolet, the best office mate one could wish for. Furthermore, I have to thank all the other former and current DCG members, Keno Albrecht, Nicolas Burri, Thomas Moscibroda, Regina O'Dell, Pascal von Rickenbach, and Aaron Zollinger for establishing an outstanding working atmosphere in our group, and Raphael Eidenbenz, Roland Flury, Olga Goussevskaia, Michael Kuhn, Remo Meier, Johannes Schneider, Beni Sigg, and Philipp Sommer, for upholding the DCG spirit.

If it was not for the continued support of my parents, Walter Locher and Doris Maag, I could not be in the position that I am today. I will forever be grateful for your encouragement and unconditional love. I also want to thank Martin Locher for being a great brother and an even better friend. Last but certainly not least, I would like to thank my wife, Wan Lin, for always believing in me, for always standing by me, and for always being there for me.

Contents

Chapter 1

Introduction

1.1 Distributed Computing

OVER the course of the last few decades, we witnessed a remarkable paradigm shift in computing: While in the past computational tasks were fed into monolithic high-capacity mainframe computers, nowadays workloads are often distributed among a large group of less powerful computational devices. In order to transfer jobs between these machines, they must be interconnected, forming a *computer network system.* The advantages of using such a computer network instead of a single mainframe are manifold. First, mainframes are highly expensive, often even more so than many less efficient computers together. Second, even if the machines of a computer network are less powerful and the jobs must be transferred between them, which costs time and thus lowers the throughput of the system, a computer network can nevertheless outperform any single mainframe if a sufficiently large number of computers are able to share the workload. Third, a computer network is more fault-tolerant as the failure of a single machine does not bring the entire system to a halt. Once the mainframe ceases to operate, no more tasks can be processed until the mainframe is either repaired or replaced. Finally, a computer network is typically extensible and thus more scalable. If the capacity of the network is no longer sufficient to handle all tasks within a given time-frame, it is usually fairly simple to integrate new machines into the network at a low cost in order to increase the capacity. However, if the mainframe does no longer offer the needed computational power, it must be replaced by a more powerful (and more expensive) machine. Thus, it comes as no surprise that computer networks managed to supersede mainframe computers and are now the preferred solution for various purposes. In the following, we briefly highlight the potential of computer networks by discussing a few examples of applications that can only be tackled through the joint force of numerous interconnected computational devices.

The largest and most prominent computer network is clearly the Internet. The millions of machines connected to the Internet together exhibit a vast and unprecedented computational power, and there is a growing number of projects striving to harness this power in order to solve computationally intense tasks. A widely popular project is SETI@home (Search for Extra-Terrestrial Intelligence)[1] whose objective is to detect extra-terrestrial radio signals using observational data from a radio telescope. Millions of volunteers have installed the SETI@home software on their computers since the project started in 1999, and hundreds of thousands of computers permanently use their idle time to analyze chunks of data that they received from the SETI@home servers in search of signals that cannot be ascribed to noise. In 2008, the computational power of these machines combined exceeded 500 TFLOPS ($5 \cdot 10^{14}$ floating point operations per second).[2] The *Berkeley Open Infrastructure for Network Computing* (BOINC)[3] software, which has originally been developed for the SETI@home project, is now used for many other purposes in a variety of fields. For example, the goal of the Rosetta@home[4] project is to predict protein structures and design new proteins to fight a wide range of diseases. Another topic of major concern is climate change. The climateprediction.net[5] project is dedicated to investigating and reducing uncertainties in climate modeling.

Every single computer in the Internet can be regarded as a tiny part of a large system, i.e., the Internet is a system whose constituent parts are distributed all over the world. We call a system consisting of numerous computational devices that are distributed, but have a means to communicate, a *distributed system*. While a computer network such as the Internet can be considered the prototypical distributed system, there are many distributed systems that distinguish themselves from the Internet in various ways. For example, other systems may have different means to exchange information: In contrast to the Internet where most communication is wired, the sensor nodes of a wireless sensor network use their radio transceivers to communicate over wireless channels. Furthermore, a distributed system is not necessarily a network of computers. Note that a computer itself is a distributed system as, e.g., the workload must be distributed efficiently among the single cores of a multi-core processor in order to achieve maximum performance. Finally, it has to be pointed out that distributed systems do not only occur in the world of computer science. For example, the human brain can also be considered a distributed system in which neurons convey information to other cells.

All distributed systems have in common that they are able to cope with

[1] See http://setiathome.berkeley.edu/.

[2] See http://boincstats.com/.

[3] See http://boinc.berkeley.edu/.

[4] See http://boinc.bakerlab.org/rosetta/.

[5] See http://climateprediction.net/.

tasks that exceed the capacity of any single part of the system by using the communication channels to share the workload. The concept of leveraging the resources of the participants of a distributed system in order to solve a computational problem is called *distributed computing*. Finding the most efficient way to solve a problem distributively is the primary objective in distributed computing. This is a challenging endeavor since the knowledge of each participant of a distributed system is always restricted to its own local state and the information it received from other participants, i.e., each participant only has a *partial view* of the entire system. The fact that the state of the system is distributed itself is one of the key characteristics of a distributed system.

If there is a centralized authority in the system that is, to some extent, aware of the state of all other participants or even has control over the entire system (such as the SETI@home servers that centrally manage and distribute chunks of data to all other computers), then the complexity of handling the distributed system is minimized. However, similar to the approach with a single mainframe computer, in this scenario we again have a single point of failure, which ought to be avoided. In a system where all entities are of equal importance, the loss of a single entity can easily be absorbed as its tasks can be redistributed among the other entities. We call a distributed system in which all participants are equivalent a *decentralized system* or a *decentralized network*. Due to their fault-tolerance and resilience it is worthwhile to study such networks and, given that all participants are of equal value and thus the state of such systems is completely distributed, also most demanding to analyze problems in decentralized networks.

Distributing computing has undoubtedly become a significant branch of computer science. There is a plethora of challenging problems in distributed computing, some of which merely occur in specific distributed systems while others can be encountered in many distributed systems. In this thesis, we are concerned with two fundamental problems of decentralized systems, which are briefly introduced in the following section.

1.2 Aggregation and Synchronization

The focus of this thesis lies on two elementary problems in distributed computing. The first part of this thesis deals with computing aggregate functions in a distributed manner, and the second part is concerned with the problem of synchronizing clocks in distributed systems.

1.2.1 Distributed Aggregation

As mentioned before, there is an enormous number of distributed applications for a wide range of purposes. The applications discussed so far have in common that a centralized entity directly communicates with all other participants and single-handedly generates and distributes all computational tasks. In other words, the centralized authority basically knows the entire state of the system. In general, computational tasks or new data may occur anywhere in a distributed system. For example, in a banking system a server may handle a monetary transaction causing the adjustment of some values in one or more databases. These changes do not entail any updates to other uninvolved databases, e.g., databases of the same bank in other cities or countries. In such a distributed system, each database stores only a (small) part of all information stored in the network and thus it merely has a partial view of the *global state* of the system at any point in time. In order to get information on the global state of the system, several databases must be queried. Another example is a wireless sensor network that, e.g., monitors the outdoor conditions in some area. Each sensor node stores its own measurements and acquiring for example the average over all measurements requires the nodes to gather the measured data.

Any function whose input is a set or a multiset, i.e., a set that may have duplicate entries, of values is called an *aggregate function*. For example, the sum or the maximum/minimum of all values in a given set are simple aggregate functions. The problem of distributed aggregation is defined as follows. Given a multiset $\mathcal{S}$ of data, which is arbitrarily distributed among the participants of a distributed system, and a specific aggregate function f, the goal is to compute $f(\mathcal{S})$. Apart from the fact that the data is distributed, which forces the participants to exchange information in order to compute the aggregate function, another factor that merits attention is that each participant may only be able to communicate directly with a subset of all other participants, incurring the need for some form of routing infrastructure.[6] The problem of routing is discussed in Chapter 3.

Given a routing infrastructure, a simple solution for any aggregate function f is to forward all data to a single participant or to a specific data sink, which will compute the result locally once it has received all data in the network. The crucial drawback of this solution is that it may be highly inefficient. For example, if a participant holds a fraction φ_1 of all data items and computes the average μ_1 of these data items, while another participant holds the remaining fraction $\varphi_2 = 1 - \varphi_1$ of all data items and the average value of its partial set is μ_2, then exchanging φ_i and μ_i suffices to compute the average μ of all data items, since $\mu = \varphi_1\mu_1 + \varphi_2\mu_2$. This solution clearly outperforms the simple solution where all data items are exchanged. Thus, the goal is to derive the most efficient distributed algorithms for a variety of aggregate functions independent of both the distribution of data and the given network structure.

[6] For example, since the transmission range of a sensor node in a wireless sensor network is limited, it is unlikely that all nodes can exchange information directly.

1.2.2 Clock Synchronization

There is a wide range of tasks in distributed systems requiring its participants to maintain a common notion of time. For example, if each participant in a monitoring system adds a timestamp to recorded events, it is desirable for the sake of consistency that any two events that occur at the same time but are registered by two distinct participants obtain roughly the same timestamp. Moreover, the participants may regularly communicate at specific times, which is only possible if there is consensus on the current time. Thus, in order to maintain consistency and enable time-based coordination, it is imperative that the local clocks of all participants deviate as little as possible.

Given that all clocks have a certain *drift*, it is necessary to use a *clock synchronization algorithm*, otherwise the clocks will continually drift apart. The goal of a clock synchronization algorithm is thus to counterbalance the clock drifts by adjusting the local clock values appropriately to ensure that all clocks in the distributed system remain synchronized as closely as possible. The main difficulty arises from the fact that the exchanged messages may be delayed arbitrarily. If a participant p receives a message containing the clock value of another participant, p cannot determine whether the obtained clock value is quite accurate, i.e., the message arrived quickly, or if the clock value is “outdated” due to a large message delay. This uncertainty has severe ramifications: It is generally impossible to determine exactly how much the clock values deviate from each other and therefore also to synchronize the clocks perfectly. Hence, the objective is to study the limitations to the degree of synchronization that can be achieved, and to derive a clock synchronization algorithm that guarantees the best possible accuracy while being light-weight in the sense that the message overhead is small.

The formal model of a distributed system that is used throughout this thesis is introduced in the following section.

1.3 Computational Model

In order to ensure that all techniques and results discussed in this thesis are broadly applicable, the model must capture the most basic commonalities of distributed systems. All distributed systems have in common that they consist of independent devices, each of which is capable of communicating with some other devices in the system. We model a distributed system as a graph $G = (V, E)$ consisting of a set V of nodes (or vertices) and a set E of edges, which are unordered pairs of nodes. Each node $v \in V$ represents a computational device and an edge $\{v, w\} \in E$, where $v, w \in V$, implies that the computational devices v and w share a communication channel, i.e., they can communicate directly. The cardinality of V is denoted by $|V| = n$. In the following, since our model is based on graph theory, we will predominantly use the terms "node" and "edge" instead of "computational device" and "communication channel". Naturally, if $\{v, w\} \notin E$, then the nodes v and w cannot communicate directly. However, if there is a node u such that $\{v, u\} \in E$ and $\{u, w\} \in E$, then v and w can communicate via node u, i.e., u can forward a message from v to w (or from w to v). We always assume that the graph G is *connected*, i.e., all nodes can communicate with all other nodes over certain paths. Moreover, all communication links are assumed to be bidirectional, which means that messages can traverse edges in both directions.[7]

For each node v, its *neighborhood* $\mathcal{N}_v$ consists of all nodes that share an edge with v. More formally, for all nodes v, its neighborhood is the set

$$\mathcal{N}_v := \{u \in V \mid \{v, u\} \in E\}.$$

By definition, node v can communicate directly with node w if and only if $w \in \mathcal{N}_v$. A node $w \in \mathcal{N}_v$ is referred to as a *neighbor* or *neighboring node* of v. The size of the neighborhood of v, i.e., the number of adjacent nodes, is commonly referred to as the *degree* of v and denoted by $\delta(v) := |\mathcal{N}_v|$. The *degree* $\Delta(G)$ of the graph G is defined as the largest degree of all nodes $v \in V$:

$$\Delta(G) := \max_{v \in V}\{\delta(v)\}.$$

The *distance* $d(v, w)$ between two nodes v and w is defined as the length of a shortest path between v and w. The formal definition of distance is given by:

$$d(v, w) := \begin{cases} 0 & \text{if } \ v = w \\ 1 + \min_{u \in \mathcal{N}_v}\{d(u, w)\} & \text{else} \end{cases}$$

The *diameter* $D(G)$ of the graph G is the largest distance, in other words, the length of a longest shortest path between any two nodes:

$$D(G) := \max_{v,w \in V}\{d(u, v)\}.$$

Throughout this thesis, the diameter $D(G)$ of the considered graph G is of central importance. Intuitively, the diameter of a graph measures how many times a message needs to be forwarded at most until it arrives at the intended recipient. Instead of writing $\Delta(G)$ and $D(G)$, we simply write Δ and D, respectively, since it is always clear from the context that these measures refer to a given graph G.

All nodes communicate by exchanging messages with their neighbors. Communication is assumed to be *asynchronous* in the sense that the message delays may be variable. In contrast to the conventional definition of asynchronous communication, which merely assumes that messages arrive

[7] In graph theoretical terms, we are only concerned with *simple* and *undirected* graphs.

"eventually", we explicitly bound the delays by defining that each message may be delayed by any value in the range $[0, \mathcal{T}]$ for a certain parameter $\mathcal{T}$. A received message can trigger some local computation and also cause the recipient itself to send messages to some of its neighbors. The parameter $\mathcal{T}$ subsumes the time it takes to perform the local computation that precedes the sending of the message, the maximum time during which a message can be in transit, and also the time it may take at most for the recipient to process the message. This general definition of $\mathcal{T}$ allows us to define that local computations require no time. Although the bound $\mathcal{T}$ is fixed, it is unknown to the nodes and thus cannot be used in an algorithm. Unless otherwise mentioned, all communication is assumed to be reliable, which means that no message is lost after it has been sent, and that every message sent from v to w arrives unaltered at node w after at most $\mathcal{T}$ time. Furthermore, the nodes are always operational, i.e., they never *fail*, and they behave exactly as specified by the algorithm.

Apart from its input, each node v can also store some local variables or partial results, which may have been obtained through exchanging messages with neighboring nodes. It is assumed that each node v knows all its neighbors $w \in \mathcal{N}_v$ and is able to distinguish them, i.e., v knows for each received message which neighbor sent it, and v can distinctly send a message only to a specific neighbor. For example, *unique identifiers*, which are attached to every message, can be used for this purpose. If there is a separate, identifiable communication channel for each neighbor, we can dispense with the use of unique identifiers as v can observe from which channel it received any particular message (and send a response through the same channel). Thus, when processing a message from a specific neighbor $w \in \mathcal{N}_v$, the algorithm executed at v can make use of the information that w is the sender.

The following section provides a few mathematical formulae that are useful in the analysis of the proposed algorithms.

1.4 Mathematical Preliminaries

The goal for each discussed problem is to devise algorithms that solve it as optimally as possible. Since we focus on asymptotic complexities, the results are often stated using the "big oh notation": For any two functions f and g we write $f(n) \in \mathcal{O}(g(n))$ if there are constants c and n_0 such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$. This means that the asymptotic growth of f is upper bounded by the growth of g. The related asymptotic notations such as the "big omega notation" etc., are also used in this thesis. For a formal definition of these notations, the reader is referred to the standard textbooks.

An algorithm that uses randomization, i.e., an algorithm whose actions are not only determined by the initial conditions, but also by the outcome of random "coin tosses", is called a *randomized algorithm*, and an algorithm that does not use randomization is called a *deterministic algorithm.* Since the output of a randomized algorithm depends on the outcome of probabilistic events, the correct result may only be obtained with a certain probability. A randomized algorithm with this property is called a *Monte Carlo* algorithm. If the result is guaranteed to be correct, but the cost to compute it is only bounded with a certain probability, the algorithm is called a *Las Vegas* algorithm. Of course, a Monte Carlo algorithm is only useful if the probability that it succeeds to find the correct solution is large. Similarly, a Las Vegas algorithm is only practical if we have a high probability that the cost to find the right solution is within reasonable bounds. The notion of *high probability* is formalized as follows: If the input size of a problem is n, we say that an event X occurs *with high probability* (w.h.p.) if the probability that X happens is at least $1 - 1/n^\lambda$, where $\lambda \geq 1$ is a constant that is a parameter in the algorithm or in the analysis. This means that the probability that the algorithm fails tends rapidly to zero as the input size of the problem increases.

In order to analyze randomized algorithm, basic statistical tools are often required. The probability that a random variable deviates from the expected value can be bounded using Markov's inequality:

Theorem 1.1 (Markov's Inequality)**.** *For any random variable X, it holds for any $\alpha > 0$ that*

$$\mathbb{P}[|X| \geq \alpha] \leq \frac{\mathbb{E}[|X|]}{\alpha}.$$

Given the variance σ^2 of a random variable, Chebyshev's inequality gives a more precise bound on the deviation from the expected value:

Theorem 1.2 (Chebyshev's Inequality)**.** *Let X be a random variable with finite variance σ^2. It holds for any $\alpha > 0$ that*

$$\mathbb{P}[|X - \mathbb{E}[X]| \geq \alpha] \leq \frac{\sigma^2}{\alpha^2}.$$

Since we often encounter independent identically distributed *Bernoulli trials* $X_1, \ldots, X_n$, we can use Chernoff bounds in order to bound the probability that $X = \sum_{i=1}^n X_i$ deviates from $\mathbb{E}[X]$ by $\delta\mathbb{E}[X]$ for a given $\delta > 0$. In the following, both the lower and the upper tail are given.

Theorem 1.3 (Chernoff Bound (Lower Tail))**.** *Let $X_1, \ldots, X_n$ be independent Bernoulli variables and let $X = \sum_{i=1}^n X_i$ denote the sum of the X_i. For any $\delta \in (0, 1]$, it holds that*

$$\mathbb{P}[X < (1 - \delta)\mathbb{E}[X]] < \left(\frac{e^{-\delta}}{(1 - \delta)^{(1-\delta)}}\right)^{\mathbb{E}[X]} < e^{-\mathbb{E}[X]\delta^2/2}.$$

Theorem 1.4 (Chernoff Bound (Upper Tail)). *Let $X_1, \ldots, X_n$ be independent Bernoulli variables and let $X = \sum_{i=1}^{n} X_i$ denote the sum of the X_i. For any $\delta > 0$, it holds that*

$$\mathbb{P}[X > (1+\delta)\mathbb{E}[X]] < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mathbb{E}[X]}.$$

Moreover, for any $\delta \in (0, 2e-1)$ it holds that

$$\mathbb{P}[X > (1+\delta)\mathbb{E}[X]] < e^{-\mathbb{E}[X]\delta^2/4}.$$

We further use the following fact, which bounds the probability that $X = \sum_{i=1}^{n} X_i$ is greater than a specific constant z.

Fact 1.5. *Let $X_1, \ldots, X_n$ be independent Bernoulli variables with $\mathbb{P}[X_i = 1] = p$ and let $X = \sum_{i=1}^{n} X_i$ denote the sum of the X_i. For any $z \geq 0$, it holds that*

$$\mathbb{P}[X \geq z] = \sum_{i=z}^{n} \binom{n}{i} p^i (1-p)^{n-i} \leq \binom{n}{z} p^z.$$

The intuition behind the expression $\binom{n}{z}p^z$ is that it gives the probability that the event associated with probability p occurs z times out of n independent trials without specifying which event occurred in the remaining $n - z$ trials.

The following combinatorial inequality will also be useful.

Fact 1.6. *For integers $n \geq k \geq 1$, we have that*

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{en}{k}\right)^k.$$

This inequality lower and upper bounds the number of k-element subsets of a set containing n elements.

Part I

Aggregation

Chapter 2

Distributed Aggregation: An Introduction

2.1 Motivation

REGARDLESS of whether data is stored at a single place or distributed on a network, the stored data is only useful if it is easily accessible. Typically, the host of the data places it into a suitable data structure or database for efficient storage and retrieval. There has been a lot of work dedicated to designing tools that enable users to conveniently interact with databases. We can distinguish between two basic forms of interaction, the first is to *query* the database and the second is to *modify* the stored data by either adding, changing, or deleting an entry in the database. We are only concerned with the first form of interaction, i.e., our aim is to acquire information about the state of the system, and we omit any discussion on how data is introduced into the system.

For relational database management systems (RDBMS), the interactive and programming language *SQL* has become a standard tool for both database inquiry and manipulation. Although it is quite an old language, the first version was developed in the early 1970s, it is still widely popular, probably due to its simple command language that can be used not only to query and manipulate the database, but also to perform management and administrative functions. SQL offers several built-in aggregate functions such as:

- `MAX`: Compute the maximum element of some data set.
- `MIN`: Compute the minimum element of some data set.
- `COUNT`: Compute the number of elements in some data set.

- SUM: Compute the sum of all values in some data set.
- AVG: Compute the average of all values in some data set.
- VAR: Compute the variance of all values in some data set.

For example, assuming that the purpose of a distributed system is to monitor the temperatures in some area and these measurements are stored in the data set *Temperature* in a table of measurements called *Measurement*, the following SQL query would return the (current) average temperature:

```
SELECT   AVG(Temperature)
FROM     Measurement
```

Of course, there are many other relevant aggregate functions. Apart from the aforementioned functions, we will also study the following aggregate functions:[1]

- RANK(x): Compute the rank of a given element x in some data set.
- MEDIAN: Compute the median element of some data set.
- SMALLEST(k): Given a parameter k, compute the k^{th} smallest element of some data set.
- LARGEST(k): Given a parameter k, compute the k^{th} largest element of some data set.
- DISTINCT: Compute the number of distinct elements in some data set.
- MODE: Compute the element that occurs most often in some data set.

The *rank* of an element in a given data set $\mathcal{S}$ is its position in $\mathcal{S}$ if the elements are arranged in ascending order. In other words, the k^{th} smallest element of a data set has rank k. Given an element x, the function RANK returns the rank of this element in the data set. If the element x occurs more than once in the data set, the rank is defined as the position of x if all but one copies of x are removed. If x does not occur in the considered data set, we define that the function returns zero.

The function SMALLEST finds an element of a given rank, hence SMALLEST is the *inverse function* of RANK, i.e., $x = \texttt{SMALLEST}(\texttt{RANK}(x))$ and $k = \texttt{RANK}(\texttt{SMALLEST}(k))$.[2]

[1] Note that not all of these functions exist in the standard SQL toolkit, but many of them may be realized through a combination of existing commands. For the sake of simplicity, we assume that all of these functions are part of the SQL language.

[2] A minor blemish is that SMALLEST(k) is not defined for all $k \in \{1, \ldots, N\}$, where N denotes the total number of elements, if $\mathcal{S}$ contains duplicates. Alternatively, we could assign a different rank to each element and define that RANK(x) returns the *range* $[k, k+p-1]$ if there are $k-1$ smaller elements than x and x occurs p times in $\mathcal{S}$. However, in this case SMALLEST is technically not the inverse function of RANK.

These functions can be used, e.g., in order to restrict the set of considered elements. As an example, it may be desirable to compute the average temperature of the ten lowest temperatures:

```
SELECT   AVG(Temperature)
FROM     Measurement
WHERE    RANK(Temperature) <= 10
```

More importantly, ranking information can be used to compute *order statistics*. A prominent example of an order statistic is the *median*. If the size N of the data set is odd, the median is defined as the element for which $(N-1)/2$ elements in the given data set are smaller and $(N-1)/2$ elements are larger. If N is even, the two elements with ranks $N/2$ and $N/2+1$ are both considered median elements throughout this thesis.[3] The median is primarily useful for skewed distribution, where it is a better indicator for central tendency than the average value.[4] The median is further less susceptible to (small) changes in the data set than the mean value and also more robust against perturbation caused by outliers.

If the number of elements N is known, the median can be found by computing the $(N/2)^{th}$ smallest element. Moreover, computing the k^{th} largest element and the $(N-k+1)^{th}$ smallest element is equivalent. Hence it follows that it suffices to be able to compute the function `SMALLEST` efficiently. It is further straightforward how to use this aggregate function to compute percentiles: The tenth percentile, e.g., can be found by applying the function `SMALLEST`$(0.1 \cdot N)$. Note that the function `SMALLEST` may also be preferable to `RANK` when restricting the resulting data set to elements of a particular rank: Assume, as in the example given above, that the result set is to be restricted to the elements whose rank is at most R. Instead of computing the rank of each element in order to determine whether it belongs to the result set, it may be beneficial to compute the element $x =$ `SMALLEST`(R) beforehand and then test for each element x' if $x' \prec x$, where $\prec$ is the binary relation that defines the order of the elements. Due to the great number of fundamental statistics that can be computing using the aggregate function `SMALLEST`, an entire chapter (Chapter 4) is devoted to the analysis of algorithmic bounds to compute it distributively.

Generally, the given data set may contain multiple copies of its elements. In SQL, the keyword "DISTINCT" eliminates all duplicate entries in the data set, thereby enabling the computation of aggregates on a duplicate-free input set. For example, the number of different temperature measurements can be evaluated as follows:

[3] Sometimes the *mean* of these two elements is considered the median of the data set.

[4] A well-known example is the distribution of income: As in many countries a few people earn considerably more money than most people, the median of all salaries is a much better indicator for the economic wealth of society than the average income.

```
SELECT   COUNT(DISTINCT Temperature)
FROM     Measurement
```

Note that the function DISTINCT is equivalent to the SQL operation "COUNT DISTINCT". Determining the number of distinct elements is a fundamental problem in databases [24, 62]. Although we will focus our attention on computing this function, it will also be pointed out how the described technique can be used to determine, e.g., the sum or the average of all *distinct* elements in the data set.

Instead of ignoring duplicates, we may be interested in determining the frequencies of elements. In particular, it may be desirable to find the element that occurs more often than all other elements. The element with the highest frequency is commonly referred to as the *mode*. Apparently, the function MODE, which determines the most frequent element, can also be used to compute the second most frequent element in the given data set by simply excluding the previously computed most frequent element and applying the function to the reduced set again.

All functions mentioned, and combinations thereof, cover a wide range of reasonable aggregation queries.[5] Similarly to how SQL is used to query databases, it would be desirable to have a tool that provides aggregation support for distributed systems where all data items are scattered among the different entities of the system. In order to build a distributed aggregation tool where queries are sent through the distributed system and the result of the query is returned to the machine that issued the query, distributed algorithms for all offered aggregate functions are required. These algorithms ought to be as efficient as possible, which means that they must require little resources, such as bandwidth, and the inquiring machine must obtain the response to the query as quickly as possible. Thus, the objective of the following chapters is to provide expedient algorithms and to analytically prove bounds on their quality.

Before delving into the study of distributed aggregation algorithms, we must clarify how quality is measured by defining an appropriate measure of complexity. Moreover, since all discussed aggregate functions are traditionally categorized into three classes, these classes are also introduced first.

2.2 Model and Definitions

Given a specific aggregate function f and a multiset $\mathcal{S} := \{x_1, \ldots, x_N\}$ consisting of N elements, where all elements x_i, $i \in \{1, \ldots, N\}$, of the multiset $\mathcal{S}$ are distributed arbitrarily among the n nodes of a network graph G, the objective of distributed aggregation is to compute $f(\mathcal{S})$.

[5] The reader is encouraged to think of a natural aggregation (single value result) query that cannot be formulated by a combination of these aggregate functions.

Let X denote the domain of all elements, i.e., each element $x_i \in \mathcal{S}$ is chosen from the set X. We are only concerned with aggregate functions that map the input set to a single value. The range may either be the real numbers $\mathbb{R}$ (the function `COUNT`, e.g., returns the number of elements), or also X (e.g., the function `MAX` returns the largest element in $\mathcal{S}$).[6] Computing functions such as `MAX` or `MEDIAN` is only feasible if there is *total order* on the elements of X. We assume that there is always a total order on the elements and that, given a total order relation $\prec$, the nodes know for any two elements $x, x' \in X$ whether $x \prec x'$ or $x' \prec x$. Since aggregate functions such as `SUM` or `AVG` require that the elements be numeric values, we can assume for these functions that $X = \mathbb{R}$ and thus the traditional smaller-or-equal relation on real numbers can be used.

As mentioned in Section 1.3, the graph G can be any connected, undirected graph consisting of n nodes, which means that the proposed algorithms for certain aggregation functions f must operate efficiently regardless of both the structure of the graph G and the given multiset $\mathcal{S}$. Naturally, there are aggregate functions that can be computed more easily than others. This observation is true not only for distributed aggregation, but also when all data is stored in a single database. In order to quantify the difficulty of computing different aggregate functions, the database community has classified aggregate functions into three categories [23]. Since we adhere to their categorization throughout this thesis, the three categories are now introduced.

Aggregate functions belonging to the first class are called *distributive*. Given a partition $\mathcal{S}_1, \ldots, \mathcal{S}_\ell$ of $\mathcal{S}$,[7] a distributive aggregate function f has the property that the aggregates $f(\mathcal{S}_1), \ldots, f(\mathcal{S}_\ell)$ can be used to compute $f(\mathcal{S})$. Formally, distributive aggregate functions are defined as follows.

Definition 2.1 (Distributive Aggregate Function)**.** *Let $\mathcal{S}$ be a multiset and let $\mathcal{S}_1, \ldots, \mathcal{S}_\ell$ be a partition of $\mathcal{S}$. An aggregate function f is called* distributive *if there is an aggregate function g such that $f(\mathcal{S}) = g(f(\mathcal{S}_1), \ldots, f(\mathcal{S}_\ell))$.*

As the name suggests, distributive aggregate functions can easily be computed distributively since partial solutions can be combined by means of a function g. Distributive aggregate functions are for example `COUNT`, `MAX`, `MIN`, `SUM`, and `RANK`. Apart from `COUNT` and `RANK`, it holds for these functions that the function g that joins the partial aggregates together is the same as the function f.[8] For the aggregate function `COUNT` the function g is simply the aggregate function `SUM`. If we only consider the multisets $\mathcal{S}'_1, \ldots, \mathcal{S}'_r$ that contain element x, the rank of x in $\mathcal{S}$ is $\texttt{RANK}(x, \mathcal{S}) = \texttt{SUM}(\texttt{RANK}(x, \mathcal{S}'_1), \ldots, \texttt{RANK}(x, \mathcal{S}'_r)) - r + 1$.

[6] Of course, it is also possible that $X \subseteq \mathbb{R}$.

[7] $\mathcal{S}_1, \ldots, \mathcal{S}_\ell$ is a *partition* of $\mathcal{S}$ if $\mathcal{S}_i \cap \mathcal{S}_j = \emptyset$ for all $i, j \in \{1, \ldots, \ell\}$, where $i \neq j$, and $\mathcal{S} = \bigcup_{i=1}^{\ell} \mathcal{S}_i$.

[8] For example, $\texttt{MAX}(\mathcal{S}) = \texttt{MAX}(\texttt{MAX}(\mathcal{S}_1), \ldots, \texttt{MAX}(\mathcal{S}_\ell))$.

The second class of aggregate functions consists of the functions that can be computed by combining distributive aggregate functions. If $f(\mathcal{S})$ can be derived from the results of distributive aggregate functions for any multiset $\mathcal{S}$, then f is referred to as an *algebraic* aggregate function.

Definition 2.2 (Algebraic Aggregate Function)**.** *An aggregate function f is called* algebraic*, if it can be computed with a fixed number of distributive aggregate functions.*

The function `AVG`, which computes the average of all elements in $\mathcal{S}$, is an algebraic aggregate function. Once `SUM` and `COUNT` have been computed, we get the average value by simply dividing these values. The function `VAR` is an algebraic aggregate function as well.

Algebraic aggregate functions are by definition not (much) harder to compute than distributive aggregate functions. In both cases it is possible to exploit the fact that sub-aggregates can be merged into the desired aggregate value. The third class distinguishes itself quite clearly from the other classes in this regard. An aggregate function is said to be *holistic* if it is not possible to combine sub-aggregates.

Definition 2.3 (Holistic Aggregate Function)**.** *An aggregate function f is called* holistic*, if there is no constant bound on the size of the storage needed to describe a sub-aggregate.*

Intuitively, a holistic aggregate function is a function that can only be computed by looking at each element individually. Since all functions that cannot be computed by combining sub-aggregates are considered holistic, the classification of aggregate functions into these three categories is exhaustive. The remaining aggregate functions, i.e., `MEDIAN`, `SMALLEST`(k), `LARGEST`(k), `DISTINCT`, and `MODE`, all belong to this class. As mentioned before, since `SMALLEST`$(k) =$ `LARGEST`$(N-k+1)$ and `SMALLEST`$(N/2) =$ `MEDIAN`, computing `SMALLEST` is at least as hard as determining `MEDIAN` or `LARGEST`. It thus suffices to derive algorithms for the function `SMALLEST`, which can also be used to compute the other two aggregate functions. In contrast, the two functions `DISTINCT` and `MODE` require different techniques as we will see in Chapter 5. The fact that sub-aggregates cannot be used directly to compute the final aggregate entails that holistic functions are considerably more difficult to compute than distributive and algebraic aggregate functions.

For any given distributed aggregation problem, no constraint is imposed on the magnitude of the elements in $\mathcal{S}$. However, the maximum size of any single message is restricted: A message may contain solely a constant number of elements—this constant does not depend on the size of the elements—and also at most $\mathcal{O}(\log n + \log N)$ arbitrary additional bits. Recall that n and N denote the number of nodes and the number of elements, respectively. These additional bits may be used, e.g., to include (a constant number of) unique node identifiers in the sent messages, which requires $\mathcal{O}(\log n)$ bits. It is reasonable to assume that a message may further contain $\mathcal{O}(\log N)$ bits as otherwise not even the result of the simple aggregate function `COUNT` can be sent in one message.

In order to constrain the bandwidth consumption, not only the maximum message size but also the number of messages that a node can send in a certain time interval must be bounded. If there is no such limitation, a node may send an unbounded number of messages in an arbitrarily short period of time. For this reason, we define that any node v can only transmit one, but not necessarily the same message to each neighbor, and then v is forced to wait for one time unit before it is allowed to send the next batch of messages to any subset of its neighbors. Note that these "waiting times" entail that the message exchange can be considered to proceed in *communication rounds*.

There are several measures of efficiency of a distributed algorithm. In the first part of this thesis, the main goal is to minimize the time required to compute any of the introduced aggregate functions. For this purpose, we analyze the *time complexity* of our proposed algorithms. The time complexity of a distributed algorithm in an asynchronous model of computation is defined as follows:

Definition 2.4 (Time Complexity)**.** *The* time complexity *of a distributed algorithm is the number of time units from the start of the execution to its completion in the worst case for every legal input and every execution scenario on any graph G.*

In distributed aggregation, the input is simply the distribution of elements on the nodes of the graph G. Recall that any single message is in transit for up to $\mathcal{T}$ time units, but it also may arrive instantaneously, i.e., after zero time. Since $\mathcal{T}$ is a constant that merely re-scales the results linearly, we use the normalized bound $\mathcal{T} := 1$ for ease of presentation. Hence, in the following any message may be delayed by any value in the range $[0, 1]$.

For some algorithms we will also state the *message complexity*, which is the number of messages that need to be sent in the worst case in order to compute the result:

Definition 2.5 (Message Complexity)**.** *The message complexity of a distributed algorithm is the number of messages exchanged in the worst case for every legal input and every execution scenario on any graph G.*

In the following chapter, we start our analysis of distributed aggregation by discussing how aggregate functions that are either algebraic or distributive can be computed efficiently in a distributed manner.

Chapter 3

Algebraic and Distributive Aggregate Functions

ALGEBRAIC and distributive aggregate functions can be computed easily given the aggregates of a partition of the entire data set $\mathcal{S}$. The main question is how a node striving to compute an aggregate function can obtain such sub-aggregates. Evidently, since each node initially knows only its own elements, the nodes must communicate and thereby exchange aggregates of their own elements, and of other elements, in order to compute the final aggregate.

A node v, whose own elements are stored in the multiset S_v, may attempt to compute the desired aggregate value as follows. Whenever a message containing an aggregate value $f(S')$ of some multiset S' is received, v simply broadcasts the sub-aggregate $f(S_v \cup S') = g(f(S_v), f(S'))$ to its neighbors. This scheme is flawed for the following reasons. If the graph G contains *cycles*, a sub-aggregate may repeatedly pass a single node over different paths, which entails that a false aggregate value is computed because its elements S_v are included several times. For example, when computing the sum of all elements, we clearly do not obtain the correct result if the same element is added more than once. If the recipient of a sub-aggregate knows or is able to detect that its own elements have already been included in the obtained aggregate, this problem can be circumvented. Note that storing the information whose local elements have already been included in each message is not an acceptable solution: Given that there are n nodes, and in general the elements of each node may or may not be included in a message, the required maximum message size is at least n bits, which is prohibitory as, apart from a constant number of elements, a message may contain solely $\mathcal{O}(\log n)$ additional bits as defined in Section 2.2. Apparently, for *idempotent* aggregate functions, such as `MAX` and `MIN`, storing this information is not necessary since the repeated application of the aggregate function cannot falsify the result.

The second problem, which also affects idempotent functions, is *termination detection*, i.e., the nodes must be able to determine whether or not the received aggregate value is the final aggregate. As we will see in the following, if a *spanning tree* is used as a simple routing infrastructure on top of the underlying graph G, these problems can be solved quite elegantly.

3.1 Spanning Tree Construction

A spanning tree $T_G = (V_T, E_T)$ of a graph $G = (V, E)$ is a *tree*, i.e., a *cycle-free* graph, that is a subgraph of G for which it holds that $V_T := V$ and for all $v \in V$ there is at least one edge $e \in E_T$ such that $v \in e$. In other words, T_G is a tree that spans all nodes of G. Since a spanning tree does not contain any cycles, there is exactly one *simple path* from each node to every other node in T_G, which renders it an ideal routing infrastructure for our purposes.

Given an arbitrary graph G, the objective is thus to build a spanning tree, which can then be used to route information when computing aggregate functions. Once the spanning tree is constructed, messages are *only* sent over the edges of the spanning tree. As mentioned before, the diameter D of G determines how long it takes at most for any two nodes to exchange information. In order to guarantee that this upper bound is not increased significantly when restricting the information exchange to the edges of the spanning tree T_G, the ratio between the diameter $D(T_G)$ of the spanning tree and D ought to be as small as possible. A natural choice for a spanning tree is a *breadth first search* (BFS) spanning tree, which is defined as follows. A BFS spanning tree T_G is a spanning tree for which it holds that there is a node v' such that the single path in T_G from v' to any other node w is a *shortest path* from v' to w in G. The nice property of a BFS spanning tree is that its diameter is at most twice as large as the diameter of the original graph G. Figure 3.1 illustrates that an arbitrary spanning tree of a graph G may have a much larger diameter than a BFS spanning tree.

A BFS spanning tree can be constructed by using a (distributed) *shortest path algorithm*, which computes shortest paths from a dedicated, but arbitrary node $v' \in V$ to all other nodes. A well-known shortest path algorithm is the Bellman-Ford algorithm [6, 20], which is referred to as algorithm $\mathcal{A}^{tree}$ in the following. For the sake of simplicity, it is assumed that a single node v' initiates the algorithm.[1] The Bellman-Ford algorithm computes shortest paths to this particular node v', inducing a BFS spanning tree. For this purpose, each node v stores the current distance d_v to v' and the current parent p_v, which is the node that is currently known to be the next closer node to v'. Initially, for all nodes $v \in V \setminus \{v'\}$ the distance is unknown and thus set to $d_v := \infty$, and p_v is undefined.

[1] Otherwise, a *leader election algorithm* may be used to determine such a node v'.

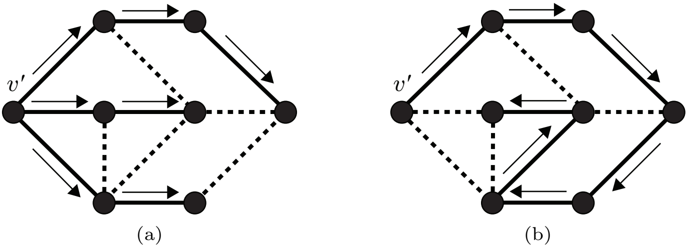

Figure 3.1: The diameter of the BFS spanning tree in Figure (a) is 5, which is less than $2D = 6$. Figure (b) shows that the diameter of an arbitrary spanning tree can be up to $n - 1 = 7$.

Node v', for which $d_{v'} := 0$, initiates the construction of the spanning tree by sending $d_{v'} + 1 = 1$ to its neighbors. The idea behind the algorithm is that whenever a node v receives a message containing a value d_w lower than d_v from a node w, it can reduce its distance to v' in the tree if w is chosen as its parent. After updating p_v to w and setting $d_v := d_w$, the other neighbors are informed that their distance to v' is $d_w + 1 = d_v + 1$ via v. The steps of algorithm $\mathcal{A}^{tree}$ are summarized in Algorithm 3.1.

The following theorem states that algorithm $\mathcal{A}^{tree}$ is correct, i.e., $\mathcal{A}^{tree}$ computes a BFS spanning tree; moreover, it gives upper bounds on the time and message complexity of $\mathcal{A}^{tree}$.

Theorem 3.1. *Algorithm $\mathcal{A}^{tree}$ computes a BFS spanning tree. The time complexity of $\mathcal{A}^{tree}$ is $\mathcal{O}(D)$ and the message complexity is $\mathcal{O}(n|E|)$.*

Proof. The time complexity is proved by induction. We claim that after at most i time units, each node v at distance i to v' stores the value $d_v = i$ for all $i \in \{0, \ldots, D\}$. For any node v at distance at most i, this implies that its parent p_v is part of a shortest path between v' and v as desired. Node v' stores 0 at time 0 and there is no other node at zero distance. We can thus assume that our claim holds for all nodes at a distance of at most i'. A node v at distance $i' + 1$ has a neighbor whose distance to v' is i', which, according to the induction hypothesis, knows that its distance is i' by time i' and sends $i' + 1$ to its neighbors. Thus, after at most one time unit, i.e., by time $i' + 1$, node v receives and stores the value $i' + 1$, which proves the claim.

Since at the latest at time D the last nodes are informed about their distance to v', the last messages are sent not later than at time D. These messages arrive at the latest at time $D+1 \in \mathcal{O}(D)$, which proves the claimed bound on the time complexity.

Algorithm 3.1 $\mathcal{A}^{tree}$: Node v received a message containing d_w from node $w \in \mathcal{N}_v$.

```
if $d_w < d_v$ then
    $d_v := d_w$
    $p_v := w$
    send $d_v + 1$ to all $u \in \mathcal{N}_v \setminus \{w\}$
end if
```

Since each node except v' has exactly one parent, the constructed graph is a spanning tree and given that each parent is part of a shortest path, the spanning tree is a BFS spanning tree rooted at v'. Hence it follows that the algorithm is correct.

The message complexity follows from the observation that each node v can reduce its variable d_v at most $n-1$ times, each time it sends a message to all its neighbors. In total, the message complexity is thus upper bounded by $\sum_{v \in V}(n-1)\delta(v) \in \mathcal{O}(n|E|)$. □

Once the spanning tree construction is complete, it can be used to route aggregate values. If most aggregation queries originate from the same node v, then v should be chosen as the node initiating the algorithm in order to ensure that the shortest paths to v are used. However, if all nodes are equally likely to issue queries, no node can be singled out as a suitable initiator. This is not crucial since the direct path from a node u to any other node w on *any* BFS spanning tree is at most twice as long as a shortest path from u to w on the original graph G as mentioned before.

While the time complexity of algorithm $\mathcal{A}^{tree}$ is optimal if a single node initiates the algorithm,[2] its shortcoming is that the nodes do not know when the construction of the spanning tree is complete because the nodes do not have an upper bound on the message delay. For this purpose, an adequate termination detection algorithm can be run in parallel. Detecting termination in this context is quite tricky because the termination detection algorithm must also involve some sort of message propagation through the network. Naturally, the constructed tree can be used to route such messages, but one has to be careful since the tree may change while the algorithm tries to detect termination. The termination detection will succeed, however, once the construction is complete, which is certainly the case after at most D time. Alternatively, another shortest path algorithm that either detects termination automatically or can be adapted easily to incorporate termination detection may be employed. An example of such an algorithm is Dijkstra's shortest path algorithm [15]. The algorithm operates in phases, at the end of each phase $i \in \{1, \ldots, D\}$, the partial spanning tree contains exactly all the nodes

[2] It clearly takes at least D time in the worst case to construct a spanning tree because it may already take D time to simply propagate a message to all nodes.

that are within distance at most i to v'. In order to add the nodes at distance $i+1$ to the spanning tree in phase $i+1$, a *discover* message, initiated by v', is sent along the edges of the already constructed spanning tree. The leaves of this tree, i.e., the nodes at distance i, can then inquire their neighbors whether they are already part of the spanning tree. Any neighbor that is not part of the tree is a node at distance $i+1$ and is added to the tree by accepting any inquiring node as its parent. Once each leaf has received a response from all its neighbors, a response is forwarded back towards v', which considers phase $i+1$ complete as soon as it has received a response from each of its neighbors.

Note that if the responses contain the information whether a new node has been added in the current phase or not, v' can determine easily at the end of a phase whether or not another phase is required: Once no more nodes have been added to the spanning tree in a certain phase, its construction must be complete and termination has been detected. The disadvantage of this algorithm is that each phase i costs $\mathcal{O}(i)$ time because messages are sent back and forth along the partially constructed tree, which implies that the time complexity is $\mathcal{O}(D^2)$. Thus, Dijkstra's algorithm is substantially slower in the worst case.

If a spanning tree has to be constructed only once, the increased time complexity may not be an issue. In this case, it might be desirable to use an algorithm such as Dijkstra's shortest path algorithm. The same spanning tree can be used if the graph G is static, i.e., if the structure of G does not change. Consequently, it is advisable to use algorithm $\mathcal{A}^{tree}$ if the spanning tree has to be constructed repeatedly due to frequent topological changes of G, e.g., caused by arriving (or departing) nodes or link discoveries (or failures). Since the focus is solely on how to efficiently compute aggregates on a particular (static) graph G, the problem of determining if or when a new spanning tree has to be constructed is not further discussed. For the sake of simplicity, it is assumed in the following that G does not change—at least for the entire duration of the aggregation process—and that *any* BFS spanning tree T_G on which messages can be routed has been pre-computed.

3.2 Aggregation on the Spanning Tree

Since all messages must be sent strictly over edges of the spanning tree $T_G = (V, E_T)$, each node v may only communicate with a neighbor $w \in \mathcal{N}_v$ if the edge $\{v, w\}$ is an edge of the spanning tree. Let $\mathcal{N}_v(T_G) \subseteq \mathcal{N}_v$ denote the set of v's neighbors that v is allowed to communicate with directly, i.e.,

$$\mathcal{N}_v(T_G) := \{w \in \mathcal{N}_v \mid \{v, w\} \in E_T\}.$$

As any algorithm to compute distributive aggregate functions can also be used to compute algebraic aggregate functions, we focus on the computation of distributive aggregate functions first and then describe how the same

Algorithm 3.2 $\mathcal{A}^{dist}$: Node v received a message COMPUTE(f) from node $w \in \mathcal{N}_v$.

```
if $C_v \neq \emptyset$ then
```

```
$p_v := w$
$C_v := \mathcal{N}_v(T_G)/\{w\}$
if $C_v \neq \emptyset$ then
    send COMPUTE($f$) to all $u \in C_v$
    wait until $f(S_1), \dots, f(S_\ell)$ received from all $u \in C_v$   ($\ell = |C_v|$)
    send $g(f(S_1), \dots, f(S_\ell), f(S_v))$ to $p_v$
else
    send $f(S_v)$ to $p_v$
end if
```

technique can be applied to algebraic aggregate functions. It is assumed that each node knows the accompanying function g that allows the combination of sub-aggregates for each distributive aggregate function f. The proposed algorithm, denoted by $\mathcal{A}^{dist}$, operates as follows. If a node v' wants to compute an aggregate $f(\mathcal{S})$, for a distributive aggregate function f, v' initiates the process by sending a message COMPUTE(f) to its neighbors in the spanning tree. Upon receiving such a message from a node w, the recipient v sets its temporary parent variable p_v to w and considers all other neighbors in the spanning tree its (temporary) children C_v. Afterwards, v forwards this message to all nodes in C_v, which will set v to be their current parent etc. For the sake of simplicity, we assume that there is just one aggregation request in the network. It is easy to see that several concurrent requests can also be served if each node stores separate parent/children relations for each request, which can be deleted after the corresponding aggregation task has been completed. A node v that does not have any children to forward the message COMPUTE(f) to (i.e., whose set C_v is empty) applies the aggregate function f to its set of elements S_v and returns the result to its parent p_w. Once a parent w has received its children's aggregates $f(S_1), \dots, f(S_\ell)$, where $\ell := |C_w| \geq 1$, it computes the aggregate $f(S_w)$ of its own elements and forwards $g(f(S_1), \dots, f(S_s), f(S_w))$, which is, by definition, exactly the aggregate function value of all elements $S_1 \cup \dots \cup S_\ell \cup S_w$ in the subtree rooted at w, to its parent. Upon receiving the aggregates from all its children, the node v' that initiated the process can then compute $f(\mathcal{S})$ by applying the function g to the sub-aggregates (including the aggregate value of its own elements). The entire algorithm $\mathcal{A}^{dist}$ is given in Algorithm 3.2.

Since all elements are added exactly once, algorithm $\mathcal{A}^{dist}$ computes the correct aggregate value. The following theorem bounds its time complexity.

Theorem 3.2. *Given a BFS spanning tree T_G of the network graph G of diameter D, the time complexity of algorithm $\mathcal{A}^{dist}$ to compute any distributive aggregate function is $\mathcal{O}(D)$.*

Proof. Let v' be the node that starts the aggregation. If the maximum distance from v' to any other node on the spanning tree is $D_{v'}$, then it takes at most $D_{v'}$ time until the message COMPUTE(f) reaches all the leaves of the spanning tree. Once the leaves receive the message, they compute the aggregate value of their local elements and send it to their respective parent. Thus, after at most $D_{v'}+1$ time, each node at distance $D_{v'}-1$ can compute the aggregate value using the received sub-aggregates and its own elements and send this value to its parent. Inductively, after at most $D_{v'}+i$ time, any node at distance $D_{v'}-i$ has received the sub-aggregates from all its children. Thus, the algorithm terminates after at most $2D_{v'}$ time. The maximum distance $D_{v'}$ is upper bounded by $D(T_G)$, the diameter of the spanning tree. Since T_G is a BFS spanning tree, it holds that $D(T_G) \leq 2D$. Hence it follows that the time complexity is upper bounded by $4D \in \mathcal{O}(D)$. □

Note that the message complexity is $2(n-1) \in \mathcal{O}(n)$ as exactly two messages are sent over each edge of the spanning tree T_G.

If the goal is to compute an algebraic aggregate functions, such as `AVERAGE`, algorithm $\mathcal{A}^{dist}$ can be adapted as follows. Let $[f_1,\ldots,f_c]$ be the distributive aggregate functions required to compute an algebraic aggregate function f_a.[3] As c is a constant by definition, each node v can pack the aggregate values $[f_1(S),\ldots,f_c(S)]$, where S is the set of elements in the subtree rooted at v, into a single message, which incurs only a constant increase of the message size. The only required modification of algorithm $\mathcal{A}^{dist}$ is that each function f_i has to be handled separately and each result is put into the same message that is forwarded to the current parent. The node initiating the aggregation can then compute the aggregate function f_a by combining the aggregate values $f_1(\mathcal{S}),\ldots,f_c(\mathcal{S})$. Since the algorithm essentially remains the same, the time complexity does not change.

One might wonder if there is a more efficient way to compute aggregates. The following theorem states that in general this is not possible, implying that $\mathcal{A}^{dist}$ is *asymptotically optimal*, i.e., its time complexity is optimal up to a constant factor.

Theorem 3.3. *Any algorithm $\mathcal{A}$ that computes $f(\mathcal{S})$ for any aggregate function f, where the elements of $\mathcal{S}$ are distributed on a graph G of diameter D, has a time complexity of $\Omega(D)$.*

Proof. Consider two nodes v and w at distance $d(v,w)=D$. If v wants to compute $f(\mathcal{S})$, it can only do so if it receives some information from w. Since it takes at least D time in the worst case until v receives the first message from w, the claimed time complexity follows. □

The technique to start the aggregation process at the leaves of a spanning tree and forward partial aggregates towards a particular node is well-known

[3] For example, for the function `AVERAGE` this set is [`SUM`,`COUNT`].

(see, e.g., [49]) and commonly referred to as a *convergecast* operation. In a convergecast the sub-aggregates are computed in the network itself, as opposed to centrally collecting the elements individually in order to compute the aggregate. Whenever partial aggregation results are computed in the network, we speak of *in-network aggregation*. Convergecast is a prototypical in-network aggregation technique, which is used in various systems to compute algebraic and distributive aggregation functions. For example, there are several data aggregation systems for wireless sensor networks based on this simple principle [39, 64, 65].

Algebraic and distributive aggregate functions can also be computed using other approaches. Astrolabe [61], a monitoring and management system for distributed applications based on peer-to-peer technology, uses gossiping techniques to compute the desired aggregates.

All practical systems have in common that they merely implement algebraic and distributive aggregate functions. To the best of our knowledge, there is no system that incorporates support for *holistic* aggregation queries. As discussed in the previous chapter, there are several important holistic aggregate functions, and it is essential to understand how to compute them efficiently. The goal of the following chapters is thus to shed light on the complexity of computing some of the most prominent examples of holistic aggregate functions in a distributed manner.

Chapter 4

Distributed Selection

$\mathscr{F}$INDING the k^{th} smallest element among a set of N elements is a classic problem which has been extensively studied in the past approximately 30 years, both in a distributed and a non-distributed setting. The problem of finding the median, i.e., the element for which half of all elements are smaller and the other half is larger, is a special case of the k-selection problem which has also received a lot of attention. Blum et al. [9] proposed the first deterministic sequential algorithm that, given an array of size N, computes the k^{th} smallest element in $\mathcal{O}(N)$ time. Their algorithm partitions the N elements into roughly $N/5$ groups of 5 elements and determines the median element of each group. The median of these $N/5$ medians is then computed recursively. While this median of medians is not necessarily the median among all N elements, it still partitions all elements well enough in that at least (roughly) 30% of all elements are smaller, and also at least 30% are larger. Thus, at least 30% of all elements can be excluded and the algorithm can be applied recursively to the remaining elements. A careful analysis of this algorithm reveals that only $\mathcal{O}(N)$ operations are required in total. Subsequently, Schönhage et al. [57] developed an algorithm requiring fewer comparisons in the worst case.

As far as distributed k-selection is concerned, a rich collection of algorithms has been amassed for various models over the years. A lot of work focused on special graphs such as stars and complete graphs [41, 52, 54, 56]. The small graph consisting of two connected nodes where each node knows half of all N elements has also been studied and algorithms with a time complexity of $\mathcal{O}(\log N)$ have been presented [13, 40, 51]. It has been shown that this result is tight for deterministic algorithms in a restricted model [51]. For the sake of simplicity, it is often assumed that each node holds exactly one element, i.e., $n = N$, when considering more general graphs consisting of n nodes. In the following, all results are given with respect to this simplified model. Frederickson [21] proposed algorithms for rings, meshes, and also complete binary trees whose time complexities are $\mathcal{O}(n)$, $\mathcal{O}(\sqrt{n})$, and $\mathcal{O}(\log^3 n)$, respectively. Several algorithms, both of deterministic [44, 46, 58] and probabilistic nature [53, 55, 58], have also been devised for arbitrary connected graphs. For some of these deterministic algorithms it is assumed that all elements are numeric and that the elements can be represented using $\mathcal{O}(\log n)$ bits. In this case, simply applying binary search results in a time complexity of $\mathcal{O}(D \log x_{max}) \subseteq \mathcal{O}(D \log n)$, where x_{max} denotes the largest numeric element [46]. By exponentially increasing the initial guess of $x_k = 1$, i.e., the k^{th} smallest element is assumed to be 1, and then applying binary search once the guess becomes too large, the solution can be found in $\mathcal{O}(D \log x_k) \subseteq \mathcal{O}(D \log n)$ time [44]. Of course, it is desirable to find the k^{th} smallest element efficiently without restricting the domain of the elements. The only non-restrictive deterministic k-selection algorithm for general graphs with a sublinear time complexity in the number of nodes is due to Shrira et al. [58]. Their adaptation of the classic sequential algorithm by Blum et al. for a distributed setting has a worst-case running time of $\mathcal{O}(Dn^{0.9114})$. In the same work, a randomized algorithm for general graphs is presented. The algorithm simply inquires a random node for its element and uses this guess to narrow down the number of elements that may still be the k^{th} smallest element. The expected time complexity of this algorithm is shown to be $\mathcal{O}(D \log n)$. It has also been studied how *gossiping* can be used for distributed k-selection: Kempe et al. [29] proposed a gossip-based algorithm that, with probability at least $1 - \varepsilon$, computes the k^{th} smallest element within $\mathcal{O}((\log n + \log \frac{1}{\varepsilon})(\log n + \log\log \frac{1}{\varepsilon}))$ rounds of communication on a complete graph. If the number of elements N is much larger than the number of nodes, the problem can be reduced to the problem where each node has exactly one element in $\mathcal{O}(D \log\log \min\{k, N-k+1\})$ expected time using the algorithm proposed by Santoro et al. [53, 55]. However, their algorithm depends on a particular distribution of the elements on the nodes. Patt-Shamir [46] showed that the median can be approximated very efficiently, again subject to the constraint that each element can be uniquely encoded using $\mathcal{O}(\log n)$ bits.

In this chapter, we shed some new light on the problem of distributed selection. In particular, we show that distributed selection is strictly harder than convergecast by proving a lower bound of $\Omega(D \log_D n)$ on the time complexity in Section 4.2. This result formally confirms the preconception that k-selection cannot be supported by a simple convergecast operation. In addition, in Section 4.1.1, a novel *Las Vegas algorithm* is presented whose time complexity is $\mathcal{O}(D \log_D n)$ with high probability. Given the lower bound of $\Omega(D \log_D n)$, it follows that the time complexity of this algorithm is asymptotically optimal. As a third result, in Section 4.1.2, the algorithm is derandomized, yielding a deterministic distributed selection algorithm with a time complexity of $\mathcal{O}(D \log_D^2 n)$, which constitutes a substantial improvement over prior art.

4.1 Algorithms

Throughout this chapter, it is assumed that each node v_i holds exactly one element x_i, i.e., $\mathcal{S} = \{x_1, \ldots, x_n\}$. Note that this simplification is only introduced for ease of presentation. The proposed algorithms can easily be generalized to the case where both the cardinality and the distribution of the elements of $\mathcal{S}$ among the n nodes are arbitrary. Recall that it is further assumed that the network graph G is static and that a BFS spanning tree T_G has been computed beforehand. Finally, we assume that all nodes know the diameter D of the graph. This assumption is not critical as the diameter can be computed in $\mathcal{O}(D)$ time. Without loss of generality, we can assume that all elements x_i are unique. If two elements x_i and x_j were equal, node identifiers, for example, could be used as tiebreakers.

As both proposed algorithms are *iterative* in that they continuously reduce the set of elements that may be the k^{th} smallest element in $\mathcal{S}$, we need to distinguish between nodes holding elements that are still of interest from the other nodes. Henceforth, the nodes whose elements may be the k^{th} smallest element are referred to as *candidate nodes* or *candidates*. The reduction of the search space by a certain factor is called a *phase* of the algorithm. The number of candidate nodes in phase i is denoted by $n^{(i)}$. This pattern is quite natural for k-selection and used in all other proposed algorithms, including the non-distributed algorithms. The best known deterministic distributed algorithm for general graphs uses the well-known *median-of-median* technique, resulting in a time complexity of $\mathcal{O}(Dn^{0.9114})$ for a constant group size. A straightforward modification of this algorithm in which the group size in each phase i is set to $\mathcal{O}(\sqrt{n^{(i)}})$ results in a much better time complexity. It can be shown that the time complexity of this variant of the algorithm is bounded by $\mathcal{O}(D(\log n)^{\log\log n + \mathcal{O}(1)})$. However, since our proposed algorithm is substantially better, we will dispense with the analysis of this median-of-median-based algorithm. Due to the more complex nature of the deterministic algorithm, the proposed randomized algorithm is discussed first.

4.1.1 Randomized Algorithm

While it is somewhat intricate to compute the k^{th} smallest element efficiently and deterministically, it is remarkably simple to come up with a fast randomized algorithm. An apparent solution, proposed by Shrira et al. [58], is to choose a node randomly and take its element as an initial guess. After computing the number of nodes with smaller and larger elements, it is likely that a considerable fraction of all nodes no longer need to be considered. By iterating this procedure on the remaining candidate nodes, the k^{th} smallest element can be found quickly for all k.

A node can be chosen randomly using the following scheme: A message indicating that a random element is to be selected is sent along a random path in the spanning tree starting at the node v initiating the aggregation. If v is a candidate and it has ℓ children $v_1, \ldots, v_\ell$, where child v_i is the root of a subtree with n_i candidate nodes including itself, v chooses its own element with probability $1/(1 + \sum_{j=1}^{\ell} n_j)$. Otherwise, it sends a message to one of its children. The message is forwarded to node v_i with probability $n_i/(1 + \sum_{j=1}^{\ell} n_j)$ for all $i \in \{1, \ldots, \ell\}$. Naturally, if v is not a candidate, it cannot choose its own element. In this case, v forwards the message to v_i with probability $n_i / \sum_{j=1}^{\ell} n_j$ for all $i \in \{1, \ldots, \ell\}$. Any recipient of such a message proceeds in the same manner until one candidate chooses its own element and sends this element to the initiating node. It is easy to see that this scheme selects a node uniformly at random and that it requires at most $4D$ time because the times to reach any node and to report back are both bounded by $2D$ since the distance between any two nodes in the BFS spanning tree T_G is at most $2D$. Note that after each phase the probabilities change as they depend on the altered number of candidate nodes remaining in each subtree. However, having determined the new interval in which the solution must lie, the number of nodes satisfying the new predicate in all subtrees can again be computed in $4D$ time.

This straightforward procedure yields an algorithm that finds the k^{th} smallest element in $\mathcal{O}(D \log n)$ expected time as $\mathcal{O}(\log n)$ phases suffice in expectation to narrow down the number of candidates to a small constant. It can even be shown that the time required is $\mathcal{O}(D \log n)$ with high probability.[1] The key observation to improve this algorithm is that picking a node randomly always takes $\mathcal{O}(D)$ time, therefore several random elements ought to be chosen in a single phase in order to further reduce the number of candidate nodes [31]. The method to select a single random element can easily be modified to enable the selection of several random elements by including the number of needed random elements in the request message. A node receiving such a message locally determines whether its own element is chosen, and also how many random elements each of its children's subtrees must provide. Subsequently, it forwards the requests to all of its children whose subtrees must provide at least one random element. Note that all random elements can be found in $2D$ time independent of the requested number of random elements, but due to the restriction that only a constant number of elements can be packed into a single message, it is likely that not all elements can propagate back to the node v that initiated the aggregation in $2D$ time. However, all elements still arrive at node v in $\mathcal{O}(D)$ time if the number of random elements is upper bounded by $\mathcal{O}(D)$.

Pseudo-code for the algorithm $\mathcal{A}_{rnd}^{sel}$ which determines the k^{th} smallest element by making use of a larger number of random elements is given in

[1] Recall that *with high probability* means with probability at least $1 - \frac{1}{n^\lambda}$ for a parameter $\lambda \geq 1$.

Algorithm 4.1 $\mathcal{A}^{sel}_{rnd}$: Compute the k^{th} smallest element using r random elements in each phase.

```
$x_{min} := -\infty$; $x_{max} := \infty$
repeat
    $\{x_1, \ldots, x_r\} :=$ getRandomElementsInRange($r, (x_{min}, x_{max})$)
    $x_0 := x_{min}$; $x_{r+1} := x_{max}$
    for $i := 1, \ldots, r+1$ in parallel do
        $c_i :=$ countElementsInRange($(x_{i-1}, x_i]$)
    end for
    $j := \min\left\{\ell \in \{1, \ldots, r+1\} \mid \sum_{i=1}^{\ell} c_i \geq k\right\}$
    if $\sum_{i=1}^{j} c_i = k$ then
        $k := 0$
    else
        $k := k - \sum_{i=1}^{j-1} c_i$
    end if
    $x_{min} := x_{j-1}$; $x_{max} := x_j$
until $c_j \leq r$ or $k = 0$
if $k = 0$ then
    return $x_j$
else
    $\{x_1, \ldots, x_s\} :=$ getElementsInRange($(x_{min}, x_{max})$)
    return $x_k$
end if
```

Algorithm 4.1. The algorithm $\mathcal{A}^{sel}_{rnd}$ uses two parameters, the number of random elements r considered in each phase, and the position of interest k. The function *getRandomElementsInRange* collects r random elements in a given range (x_{min}, x_{max}). As mentioned before, this operation includes in a first step the counting of nodes whose elements lie in the specified interval. Once the number of candidate nodes in each subtree has been determined, the r random elements are selected and reported back to the initiator v. Hence, this function call overall takes $\mathcal{O}(D + r)$ time. After acquiring the random elements, which are ordered such that $x_1 \prec \ldots \prec x_r$, the number of elements c_i in the intervals $(x_{i-1}, x_i]$ are counted using the function *countElementsInRange*. All these counting requests can be sent one after the other, thus there is no need to wait for one single counting operation to complete. By counting the nodes in each interval in parallel, the time complexity of this operation is again $\mathcal{O}(D + r)$. Afterwards, the interval (x_{j-1}, x_j) in which the desired element is to be found is determined and k is updated accordingly. These steps are repeated until the solution is found, i.e., $k = 0$, or the interval contains at most r elements, in which case all elements can be collected in $\mathcal{O}(D + r)$ time and the solution can be computed locally.

It is evident that the number of iterations determines the overall time complexity as each operation can be performed in $\mathcal{O}(D)$ time provided that $r \in \mathcal{O}(D)$. The following lemma states how many phases are required in order to find the solution with high probability.

Lemma 4.1. *If the diameter of G is $D \geq 2$ and $r := \lceil 8\lambda D \rceil$, where $\lambda \geq 1$, then algorithm $\mathcal{A}^{sel}_{rnd}$ determines the k^{th} smallest element in less than $4 \log_D n$ phases w.h.p.*

Proof. First, we compute an upper bound on the probability that after any phase i the k^{th} smallest element is in a fraction of size at least $c\frac{\log D}{D}$ times the size of the fraction after phase $i-1$ for a suitable constant c, i.e., $n^{(i)} \geq n^{(i-1)}\frac{c \log D}{D}$.[2] Let $\hat{x}_1 \prec \ldots \prec \hat{x}_n$ denote the total order of all elements. The size $n^{(i)}$ is determined by the smallest element larger than $\hat{x}_k$ and the largest element smaller than $\hat{x}_k$ that are chosen randomly in phase i. Assume that there are at least $\lfloor \frac{c \log D}{2D} n^{(i-1)} \rfloor$ candidates whose elements are larger than $\hat{x}_k$ after phase $i-1$. In this case, the probability that none of the elements $\hat{x}_{k+1}, \ldots, \hat{x}_{k+\lfloor \frac{c \log D}{2D} n^{(i-1)} \rfloor}$ are among the $r = \lceil 8\lambda D \rceil$ random elements, which implies that there are at least $\lfloor \frac{c \log D}{2D} n^{(i-1)} \rfloor$ candidates whose elements are larger than $\hat{x}_k$ after phase i, is

$$\left(1 - \frac{\lfloor \frac{c \log D}{2D} n^{(i-1)} \rfloor}{n^{(i-1)}}\right)^{\lceil 8\lambda D \rceil}.$$

Apparently, there are less than $\lfloor \frac{c \log D}{2D} n^{(i-1)} \rfloor$ candidates whose elements are larger than $\hat{x}_k$ after phase i if there are already less than $\lfloor \frac{c \log D}{2D} n^{(i-1)} \rfloor$ such candidates after phase $i-1$. The same argument holds for the probability that there are at least $\lfloor \frac{c \log D}{2D} n^{(i-1)} \rfloor$ candidates whose elements are smaller than $\hat{x}_k$ after phase i and thus, by virtue of a union-bound argument, we have that

$$\begin{aligned} p := \mathbb{P}\left[n^{(i)} \geq n^{(i-1)} \frac{c \log D}{D}\right] &\leq 2\left(1 - \frac{\lfloor \frac{c \log D}{2D} n^{(i-1)} \rfloor}{n^{(i-1)}}\right)^{\lceil 8\lambda D \rceil} \\ &< 2\left(1 - \frac{\frac{c \log D}{2D} n^{(i-1)} - 1}{n^{(i-1)}}\right)^{8\lambda D} \\ &< 2\left(1 - \frac{4\lambda c \log D - 1}{8\lambda D}\right)^{8\lambda D} \\ &\leq 2e^{-4\lambda c \log D + 1}. \end{aligned} \tag{4.1}$$

We used the fact that $r = \lceil 8\lambda D \rceil < n^{(i-1)}$ as otherwise the algorithm would simply collect all remaining elements in phase i. We call phase i *successful* if $n^{(i)} < n^{(i-1)}\frac{c \log D}{D}$. By setting $c := \frac{6}{5}$, less than $3 \log_D n$ successful

[2] Note that the base of the logarithm is always 2 unless otherwise noted.

phases are required to find $\hat{x}_k$ as it holds for all $D \geq 1$ that $\left(\frac{c \log D}{D}\right)^3 < \frac{1}{D}$ for this choice of c.

Let the random variable $\mathcal{U}(\tau)$ denote the number of *unsuccessful* phases out of τ phases in total. The probability that $\tau := 4 \log_D n$ phases do not suffice is therefore at most

$$\begin{aligned}
\mathbb{P}[\mathcal{U}(\tau) \geq \log_D n] \quad &= \quad \sum_{i=\log_D n}^{\tau} \binom{\tau}{i} p^i (1-p)^{\tau - i} \\
&\overset{\tau = 4\log_D n}{\leq} \quad \binom{4 \log_D n}{\log_D n} p^{\log_D n} && (4.2) \\
&\overset{(4.1)}{<} \quad (4e)^{\log_D n} \left(2e^{-4\lambda c \log D + 1}\right)^{\log_D n} && (4.3) \\
&= \quad \left(e^{-(4\lambda c \log D - 2 - \ln 8)}\right)^{\log_D n} \\
&< \quad \left(\frac{1}{D^\lambda}\right)^{\log_D n} = \frac{1}{n^\lambda}.
\end{aligned}$$

Inequality (4.2) follow immediately from Fact 1.5, and Inequality (4.3) holds due to Fact 1.6. The last inequality holds because $4\lambda c \frac{\ln D}{\ln 2} - 2 - \ln 8 > \lambda \ln D$ for $c = \frac{6}{5}$ and $D \geq 2$. Hence, with high probability, the algorithm terminates after less than $4 \log_D n$ phases. □

As the number of phases is $\mathcal{O}(\log_D n)$ with high probability, we immediately get the following result.

Theorem 4.2. *If the diameter of G is $D \geq 2$ and $r := \lceil 8\lambda D \rceil$, where $\lambda \geq 1$, the time complexity of algorithm $\mathcal{A}_{rnd}^{sel}$ is $\mathcal{O}(D \log_D n)$ w.h.p.*

This algorithm is considerably faster than the algorithm selecting only a single random element in each phase. In Section 4.2, we prove that no deterministic or probabilistic algorithm can be better asymptotically, i.e., algorithm $\mathcal{A}_{rnd}^{sel}$ is asymptotically optimal.

4.1.2 Deterministic Algorithm

The difficulty of deterministic iterative algorithms for k-selection lies in the selection of elements that provably lead to a reduction of the search space in each phase. Once these elements have been found, the reduced set of candidate nodes can be determined in the same way as in the randomized algorithm. Therefore, the only difference between the two algorithms is the way these elements are chosen. While the function *getRandomElementsInRange* performs this task in the randomized algorithm $\mathcal{A}_{rnd}^{sel}$, a suitable counterpart for the deterministic algorithm, referred to as $\mathcal{A}_{det}^{sel}$, has to be derived.

A simple idea to go about this problem is to start sending up elements from the leaves of the spanning tree, accumulating the elements from all children at the inner nodes, and then forwarding a selection of at most q elements to the parent. If an inner node v_i receives q elements from each of its $\delta(v_i) - 1$ children in the spanning tree, the q elements that partition all $(\delta(v_i) - 1)q$ nodes into $q + 1$ segments of approximately equal size ought to be found and forwarded. However, in order to find these elements, the number of elements in each segment has to be counted starting at the leaves. Since this counting has to be repeated in each step along the path to the root, it takes $\mathcal{O}(D(D + \Delta q))$ time to find a useful partitioning into $q + 1$ roughly equally sized segments. This approach suffers from several drawbacks: It takes $\mathcal{O}(D^2)$ time just to find a partitioning, and the time complexity depends on the structure of the spanning tree.

The proposed algorithm $\mathcal{A}_{det}^{sel}$ solves these issues in the following manner. In any phase i, the algorithm splits the entire graph into $\mathcal{O}(\sqrt{D})$ groups, each of size $\mathcal{O}(n^{(i)}/\sqrt{D})$. Recursively, in each of those groups a particular node initiates the same partitioning into $\mathcal{O}(\sqrt{D})$ groups as long as the group size is larger than $\mathcal{O}(\sqrt{D})$. Groups of size at most $\mathcal{O}(\sqrt{D})$ simply report all their elements to the node that initiated the grouping at this recursion level. Once such an initiating node v has received all $\mathcal{O}(\sqrt{D})$ elements from each of the $\mathcal{O}(\sqrt{D})$ groups it created, it sorts those $\mathcal{O}(D)$ elements, and subsequently issues a request to count the nodes in each of the $\mathcal{O}(D)$ intervals induced by the received elements. Assume that all the groups created by node v together contain $n_v^{(i)}$ nodes in phase i. The intervals can locally be merged into $\mathcal{O}(\sqrt{D})$ intervals such that each interval contains at most $\mathcal{O}(n_v^{(i)}/\sqrt{D})$ nodes. These $\mathcal{O}(\sqrt{D})$ elements are recursively sent back to the node that created the group to which node v belongs. Upon receiving the $\mathcal{O}(D)$ elements from its $\mathcal{O}(\sqrt{D})$ groups and counting the number of nodes in each interval, the root can initiate phase $i + 1$ for which it holds that $n^{(i+1)} \in \mathcal{O}(n^{(i)}/\sqrt{D})$ [31]. The procedure *getPartitionInRange* that computes this partitioning is given in Algorithm 4.2.

We will now study each part of the function *getPartitionInRange* in greater detail. In a first step, groups are created using the function *createGroups*. This operation additionally determines the number of remaining candidate nodes in each subtree in phase i by accumulating the updated counters starting at the leaves of the tree: The leaf nodes return 0 if their elements do not satisfy the predicate of the current phase, and 1 otherwise. Simultaneously, the groups are built using the following procedure. Any inner node v with children $v_1, \ldots, v_p$ whose subtrees contain $n_1^{(i)}, \ldots, n_p^{(i)}$ candidates in phase i creates groups $g_1, \ldots, g_z$, where $g_j \subseteq \{1, \ldots, p\}$ for all $j \in \{1, \ldots, z\}$, $\bigcup_{j \in \{1,\ldots,z\}} g_j = \{1, \ldots, p\}$, and $g_j \cap g_m = \emptyset$ for all $j, m \in \{1, \ldots, z\}$. The size of a group g_j is defined as $s(g_j) := \sum_{c \in g_j} n_c^{(i)}$. The groups are created such that $s(g_j) \leq n^{(i)}/\sqrt{D}$ for all j, and z is minimal. Unless the size of each

Algorithm 4.2 getPartitionInRange($q, (x_{min}, x_{max})$): Compute $m \leq q$ elements that appropriately partition all elements in the range (x_{min}, x_{max}).

```
$n' :=$ countElementsInRange($(x_{min}, x_{max})$)
**if** $n' > q$ **then**
    $\{v_1, \ldots, v_\ell\} :=$ createGroups($q, (x_{min}, x_{max})$)
    **for** $i := 1, \ldots, \ell$ **in parallel do**
        $\{x_{i1}, \ldots, x_{im}\} :=$ getPartitionInRange($q, (x_{min}, x_{max})$) from $v_i$
    **end for**
    $\mathcal{X} := \bigcup_{i=1,\ldots,\ell}\{x_{i1}, \ldots, x_{im}\}$
    $\{x_1, \ldots, x_s\} :=$ sort($\mathcal{X}$)
    $x_0 := x_{min}$; $x_{s+1} := x_{max}$
    **for** $i := 1, \ldots, s+1$ **in parallel do**
        $c_i :=$ countElementsInRange($(x_{i-1}, x_i]$)
    **end for**
    $\{x'_1, \ldots, x'_m\} :=$ reduce($\{x_1, \ldots, x_s\}, \{c_1, \ldots, c_{s+1}\}$)
**else**
    $\{x'_1, \ldots, x'_m\} :=$ getElementsInRange($(x_{min}, x_{max})$)
**end if**
**return** $\{x'_1, \ldots, x'_m\}$
```

group is exactly $\lfloor n^{(i)}/\sqrt{D}\rfloor$, node v adds itself to any group whose size is less than $\lfloor n^{(i)}/\sqrt{D}\rfloor$ if its element is still of interest. If v cannot join any group, it becomes the single element of an additional group. Node v selects for each group g_j, with the exception of one group $g' \in \{g_1, \ldots, g_z\}$, a *leader* which is any node v_m where $m \in g_j$ (or possibly v itself if it joined this group). The group g', for which it must hold that $s' := s(g') < \lfloor n^{(i)}/\sqrt{D}\rfloor$, is the *incomplete group* which might increase in size further up the spanning tree.[3] For this purpose, s' is propagated to the parent of v indicating that there is a group of size s' that can be enlarged. If the size of each group happens to be exactly $\lfloor n^{(i)}/\sqrt{D}\rfloor$, node v sends $s' = 0$ to its parent, which signifies that all the nodes in this subtree are already perfectly matched. Note that, while inner nodes can act as relay nodes for group communication, any edge is used strictly by at most one group. Subsequently, all nodes are informed about their group membership. Once the root is reached, the incomplete group becomes a complete group since it cannot be enlarged anymore, and a leader is selected for this group as well. All the selected leaders $v_1, \ldots, v_\ell$ report to the root, which concludes the operation *createGroups*.

Figure 4.1 depicts an example where node v creates three groups. Node v_1 becomes the leader of the first group, node v declares itself the leader of

[3] Note that any group containing less than $\lfloor n^{(i)}/\sqrt{D}\rfloor$ nodes may be selected as the incomplete group. This decision does not have an impact on the asymptotic time complexity of the algorithm.

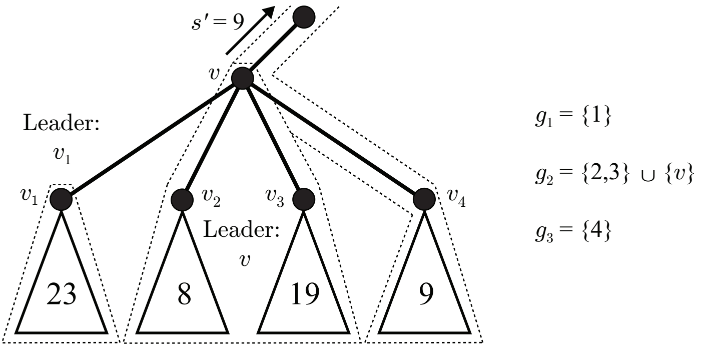

Figure 4.1: The maximum group size in phase i of this example is $\lfloor n^{(i)}/\sqrt{D} \rfloor = 28$. The first group g_1 consists only of the incomplete group rooted at v_1, while g_2 consists of the two incomplete groups rooted at v_2 and v_3. Node v joins g_2 and becomes its leader. Information about the incomplete group g' rooted at v_4 is forwarded to the parent of v.

the second group, and the third group, which consists of nine candidates, is the incomplete group, which might be merged with other incomplete groups by the parent of v.

Once these groups are set up, *getPartitionInRange* is called recursively at each leader $v_i \in \{v_1, \ldots, v_\ell\}$ in order to further partition the groups. The return value of this function call at a particular node v_i is a subset of the elements stored at candidate nodes belonging to the group of which v_i is the leader. If its group consists of at most q candidates, all elements are returned to the node that issued this request for a further partitioning. If the group is larger, the resulting sets of the recursive calls are accumulated and sorted using the function *sort*. Afterwards, the function *countElementsInRange* counts the number of elements belonging to candidate nodes that are part of this group for each induced interval. Subsequently, the function *reduce* merges adjacent intervals as long as each interval contains at most $n_v^{(i)}/\sqrt{D}$ elements, and the reduced set of elements $\{x'_1, \ldots, x'_m\} \subset \{x_1, \ldots, x_{s+1}\}$ inducing these new intervals is returned. The following lemma bounds the number of elements returned and the number of elements in each interval.

Lemma 4.3. *If $q := 2\sqrt{D}$,* getPartitionInRange *executed at any leader v in phase i returns a set of $m \leq q$ elements which induce intervals containing at most $n_v^{(i)}/\sqrt{D}$ elements each.*

Proof. We will prove the lemma by induction. Both claims are obviously true if the group of node v is of size at most $q = 2\sqrt{D}$. Assume now that the group contains more than q candidates, and that, by the induction hypothesis, both claims hold for all ℓ returned sets of elements. Let $n_{v_j}^{(i)}$ denote the number of elements in the group of which node v_j is the leader in phase i. As any interval $(x_{j-1}, x_j]$, for $j \in \{1, \ldots, s+1\}$, contains elements from at most one interval from each of the ℓ groups, we have that the total number of elements in this interval is bounded by $\sum_{j=1}^{\ell} n_{v_j}^{(i)}/\sqrt{D} \leq n_v^{(i)}/\sqrt{D}$. If two adjacent intervals together contain less than $n_v^{(i)}/\sqrt{D}$ elements, they are combined into one interval. Let z denote the number of intervals after the intervals have been merged. Once no more intervals can be merged, any two consecutive segments contain more than $n_v^{(i)}/\sqrt{D}$ elements and it therefore holds that $2n_v^{(i)} = 2\sum_{j=1}^{\ell} n_{v_j}^{(i)} > (z-1)n_v^{(i)}/\sqrt{D}$. Hence it follows that $z < 2\sqrt{D} + 1$. As the values x_{min} and x_{max} define the entire considered interval, $m = z - 1$ elements are required to specify the boundaries of the z intervals, which concludes the proof. □

The root uses the elements returned by *getPartitionInRange* to narrow down the range of potential candidates as described in Section 4.1.1. Algorithm $\mathcal{A}_{det}^{sel}$ takes two parameters, the number q of requested elements and the value k. We are now in the position to prove the following theorem.

Theorem 4.4. *If the diameter of G is $D \geq 2$ and $q := 2\sqrt{D}$, the time complexity of algorithm $\mathcal{A}_{det}^{sel}$ is $\mathcal{O}\left(D \log_D^2 n\right)$.*

Proof. It follows from Lemma 4.3 that $n^{(i+1)} \leq n^{(i)}/\sqrt{D}$, thus the number of phases is bounded by $\mathcal{O}(\log_D n)$. The groups can be created in $\mathcal{O}(D)$ time. The only other non-local operations are the collection of the sets from all subgroups and the counting of all elements in each interval within the entire group. Since the number of groups is upper bounded by $2\sqrt{D}$ and each group returns at most $2\sqrt{D}$ elements according to Lemma 4.3, at most $4D$ elements have to be collected at the initiating node, which can be done in $\mathcal{O}(D)$ time. Consequently, the number of elements in each interval can also be counted in $\mathcal{O}(D)$ time. Let $T : \mathbb{N} \to \mathbb{N}$ where $T(n)$ denotes the time complexity of *getPartitionInRange* if there are n candidate nodes. We have that $T(n) \leq T(n/\sqrt{D}) + cD$ for a suitable constant c, implying that $T(n) \in \mathcal{O}(D \log_D n)$. The time complexity of $\mathcal{A}_{det}^{sel}$ is therefore upper bounded by $\mathcal{O}(D \log_D^2 n)$. □

For many networks this running time is strictly below the time required to collect all elements at one node. In fact, if $D \in \Omega(n^c)$ for a constant $c \leq 1$, the time complexity of $\mathcal{A}_{det}^{sel}$ is $\mathcal{O}(D)$ and thus asymptotically the same as the complexity of a simple convergecast.

4.2 Lower Bound

In this section, a lower bound on the time complexity for *generic* distributed selection algorithms is proved which shows that the time complexity of the simple randomized k-selection algorithm of Section 4.1.1 is asymptotically optimal for most values of k. Informally, we call a selection algorithm *generic* if it does not exploit the structure of the element space except for using the fact that there is a global order on all the elements. Formally, this means that the only access to the structure of the element space is by means of the comparison function. Equivalently, we can assume that all elements assigned to the nodes are fixed but that the ordering of elements belonging to different nodes is determined by an adversary and initially not known to the nodes. A simpler *synchronous communication model* where time is divided into rounds and in every round each node can send a message to each of its neighbors is used for the lower bound. Note that since the synchronous model is more restrictive than the asynchronous model, a lower bound for the synchronous model directly carries over to the asynchronous model. We show that if in any round only $B \geq 1$ elements can be transmitted over any edge, such an algorithm needs at least $\Omega(D \log_D n)$ rounds to find the median with reasonable probability. A lower bound for finding an element with an arbitrary rank k can then be derived by using a simple reduction.

The lower bound is proved in two steps. First, we prove a lower bound of $\Omega(\log_B n)$ on the time complexity for protocols between two nodes where each node starts with half of the elements. In a second step, a graph $T(D)$ for every diameter $D \geq 3$ is constructed such that every median algorithm on $T(D)$ can be simulated by two nodes to compute the median in a two-party protocol. For the sake of simplicity, the lower bound is proved for deterministic algorithms. *Yao's principle* [63] can then be applied in order to obtain a lower bound for randomized algorithms.

Let u and v denote the two nodes in a two-party protocol. Furthermore, let $u_0 \prec u_1 \prec \cdots \prec u_{N-1}$ and $v_0 \prec v_1 \prec \cdots \prec v_{N-1}$ be the elements stored by u and v, respectively, where $N \geq 1$ and $\prec$ denotes the global order according to which the median is to be found. The two sets of elements are denoted by $S_u := \{u_0, \ldots, u_{N-1}\}$ and $S_v := \{v_0, \ldots, v_{N-1}\}$ in the following. Without loss of generality, it can be assumed that no element occurs twice, i.e., there is a total of $2N$ distinct elements. Each message $M = (S, Z)$ between the two nodes is further assumed to contain a set S of at most B elements and some arbitrary additional information Z. If M is a message from u to v, then Z can be everything that can be computed from the results of the comparisons between all the elements u has seen so far, as well as all the additional information u has received so far. The only restriction on Z is that it cannot be used to transmit information about additional elements. We call a protocol between u and v that only sends messages of the form $M = (S, Z)$ as described above a *generic two-party protocol*.

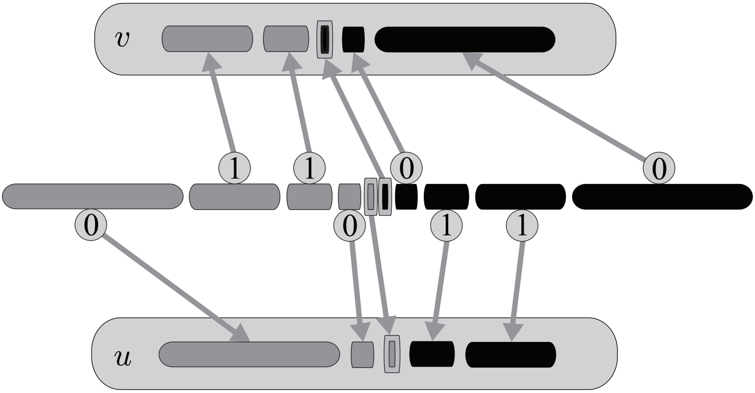

Figure 4.2: The $2N$ elements minus the two medians are assigned to u and v according to $\ell = \log N$ independent Bernoulli variables $Y_0, \ldots, Y_{\ell-1}$. One of the two medians is assigned to u and the other to v.

Of course, the time needed to compute the median in the above model depends on how the $2N$ elements are distributed among u and v. Therefore, this distribution or, equivalently, the outcome of comparisons between elements $u_i \in S_u$ and $v_j \in S_v$, needs to be determined first. The general idea is the following. N different distributions of elements are constructed (i.e., N different orders between the elements in S_u and S_v) in such a way that the N distributions result in N different median elements. If the distribution is chosen uniformly at random from these N distributions, then the probability to reduce the number of possible distributions in each communication round by more than a factor of λB is exponentially small in λ [31].

For simplicity, assume that $N = 2^\ell$ is a power of 2. Let $Y_0, \ldots, Y_{\ell-1} \sim$ Bernoulli(1/2) be ℓ independent Bernoulli variables, i.e., all Y_i take values 0 or 1 with equal probability. The distribution of the $2N$ elements among u and v is determined by the values of $Y_0, \ldots, Y_{\ell-1}$. If $Y_{\ell-1} = 0$, the $N/2$ smallest of the $2N$ elements are assigned to u and the $N/2$ largest elements are assigned to v. If $Y_{\ell-1} = 1$, it is the other way round. In the same way, the value of $Y_{\ell-2}$ determines the assignment of the smallest and largest $N/4$ of the remaining elements: If $Y_{\ell-2} = 0$, u gets the elements with ranks $N/2 + 1, \ldots, 3N/4$ and v gets the elements with ranks $5N/4 + 1, \ldots, 3N/2$ among all $2N$ elements. The remaining elements are recursively assigned in the same way depending on the values of $Y_{\ell-3}, \ldots, Y_0$ until only the two elements with ranks N and $N + 1$ (i.e., the two median elements) remain. The element with rank N is assigned to u and the element with rank $N+1$ is assigned to v. See Figure 4.2 for an illustration of this distribution process.

Formally, the resulting global order can be defined as follows. Let u_α be an element of u and let v_β be an element assigned to v. Recall that u_α is the $(\alpha+1)^{th}$ smallest element of u and that v_β is the $(\beta+1)^{th}$ smallest element of v. Let $\alpha_{\ell-1}\dots\alpha_0$ and $\beta_{\ell-1}\dots\beta_0$ be the binary representations of α and β, i.e., $\alpha=\sum_{i=0}^{\ell-1}\alpha_i 2^i$ and $\beta=\sum_{i=0}^{\ell-1}\beta_i 2^i$. Assume that there is an index i for which $\alpha_i = Y_i$ or $\beta_i \neq Y_i$. Let i^* be the largest such index. If $Y_{i^*}=0$, it holds that $u_\alpha \prec v_\beta$, whereas if $Y_{i^*}=1$, we have that $v_\beta \prec u_\alpha$. If there is no index i for which $\alpha_i = Y_i$ or $\beta_i \neq Y_i$, then $u_\alpha \prec v_\beta$. In this case, u_α and v_β are the elements with ranks N and $N+1$ among all $2N$ elements, i.e., u_α and v_β are the median elements. Since the median elements u_α and v_β are exactly those elements for which $\alpha_i \neq \beta_i = Y_i$ for all $i \in \{0,\dots,\ell-1\}$, it immediately follows that finding the median is equivalent to determining the values of Y_i for all i.

Let $\mathcal{A}$ be a deterministic, generic two-party algorithm between u and v that computes the median. Consider the state of u and v after the first t rounds of an execution of $\mathcal{A}$. Let $S_{uv}(t) \subseteq S_u$ and $S_{vu}(t) \subseteq S_v$ be the sets of elements that u and v have sent to each other in the first t rounds. After t rounds, everything u and v can locally compute has to be a function of the results of comparisons between elements in $S_u \cup S_{vu}(t)$ and of comparisons between elements in $S_v \cup S_{uv}(t)$, respectively. Note that except for the elements themselves, everything u and v can send to each other can be computed from comparisons between elements within these two sets. Let us therefore define the combined state $\mathbf{state}_{u,v}(t)$ of u and v at time t as the partial order induced by comparing all pairs of elements in $S_u \cup S_{vu}(t)$ and by comparing all pairs of elements in $S_v \cup S_{uv}(t)$. Since knowledge of the median implies that the values of Y_i for all i are also known, it follows that if u and v know the median after t rounds, the values of all Y_i can be computed as a function of $\mathbf{state}_{u,v}(t)$. For an element $u_\alpha \in S_u$ let

$$I(u_\alpha) := \max\bigl\{i \in \{0,\dots,\ell-1\} \big| Y_i = \alpha_i\bigr\},$$

where α_i is defined as above. If there is no i for which $Y_i = \alpha_i$, we define that $I(u_\alpha) := -1$. Similarly, for an element $v_\beta \in S_v$ let

$$J(v_\beta) := \max\bigl\{j \in \{0,\dots,\ell-1\} \big| Y_j \neq \beta_j\bigr\}.$$

Again, we define that $J(v_\beta) := -1$ if $Y_j = \beta_j$ for all j. The following lemma quantifies how much can be deduced about the values of the random variables Y_i from a given combined state $\mathbf{state}_{u,v}(t)$.

Lemma 4.5. *Let $I^*(t) := \min_{u_\alpha \in S_{uv}(t)} I(u_\alpha)$, $J^*(t) := \min_{v_\beta \in S_{vu}(t)} J(v_\beta)$, and $H^*(t) := \min\{I^*(t), J^*(t)\}$. The combined state $\mathbf{state}_{u,v}(t)$ of u and v at time t is statistically independent of Y_i for $i < H^*(t)$.*

Proof. We prove that $\mathbf{state}_{u,v}(t)$ can be computed if Y_i is known for all $H^*(t) \le i \le \ell-1$. The lemma then follows because the random variables $Y_0,\dots,Y_{\ell-1}$ are independent.

In order to prove that $\mathbf{state}_{u,v}(t)$ can be computed from the values of Y_i for $i \geq H^*(t)$, it has to be shown that the results of all comparisons between two elements in $S_u \cup S_{vu}(t)$ and between two elements in $S_v \cup S_{uv}(t)$ can be deduced from the knowledge of these Y_i. For this purpose, consider an element $u_\alpha \in S_u$ and an element $v_\beta \in S_{vu}(t)$. Assume that there is an index i for which $\alpha_i = Y_i$ or $\beta_i \neq Y_i$ and let i^* be the largest index for which this holds. In this case, we have that $u_\alpha \prec v_\beta$ if $Y_{i^*} = 0$ and $v_\beta \prec u_\alpha$ if $Y_{i^*} = 1$, i.e., the outcome of the comparison between u_α and v_β is determined by the value of Y_{i^*}. However, by the definition of $J(v_\beta)$, we have $i^* \geq J(v_\beta) \geq J^*(t) \geq H^*(t)$. If there is no index i for which $\alpha_i = Y_i$ or $\beta_i \neq Y_i$, then $H^*(t) = -1$ and the lemma trivially holds. Symmetrically, it can be shown that every comparison between two elements in $S_v \cup S_{uv}(t)$ is determined by the value of a variable $Y_{i'}$ with $i' \geq H^*(t)$ if $H^*(t) \geq 0$, which concludes the proof. □

Based on Lemma 4.5, we are able to prove a lower bound on the complexity to find the element of rank k by means of a generic two-party protocol in the case where each node starts with N elements.

Theorem 4.6. *Let $h := \min\{k, 2N-k\}$. Every, possibly randomized, generic two-party protocol needs at least $\Omega(\log_B h)$ rounds to find the element with rank k in expectation and with probability at least $1 - 1/h^\delta$ for any constant $\delta < 1/2$.*

Proof. For simplicity, assume that B is a power of 2. In a first step, the lower bound to find the median, i.e., $k = N$, is proved. For the state after t rounds, let again $I^*(t) = \min_{u_\alpha \in S_{uv}(t)} I(u_\alpha)$, $J^*(t) = \min_{v_\beta \in S_{vu}(t)} J(v_\beta)$, and $H^*(t) = \min\{I^*(t), J^*(t)\}$ as defined above. Initially, it holds that $H^*(0) = \ell$. Assume that a given protocol $\mathcal{A}$ needs T rounds to find the median, i.e., $H^*(T) = 0$. The progress of round t is defined as $\mathit{progress}(t) := H^*(t) - H^*(t-1)$. In the following, we show that the progress of every round is at best geometrically distributed:

$$\forall t: \ \mathbb{P}\big[\mathit{progress}(t) \geq \xi\big] \leq \frac{2B}{2^\xi}. \tag{4.4}$$

Consider $\mathbf{state}_{u,v}(t-1)$ and $\mathbf{state}_{u,v}(t)$. By the definition of $H^*(t)$, in round t, either one of the B elements u_α that u sends to v satisfies $I(u_\alpha) = H^*(t)$ or one of the B elements v_β that v sends to u satisfies $J(v_\beta) = H^*(t)$. If $I(u_\alpha) = H^*(t)$, the $(\ell - H^*(t))$-highest priority bits of the base-2 representation of α equal $Y_{H^*}, \ldots, Y_{\ell-1}$. Similarly, if $I(v_\beta) = H^*(t)$, the $(\ell - H^*(t))$-highest priority bits of the base-2 representation of β equal the complements of $Y_{H^*}, \ldots, Y_{\ell-1}$. Therefore, at least one of the $2B$ elements sent in round t contains all information about the values of Y_i for $i \geq H^*(t)$. Since the combined state $\mathbf{state}_{u,v}(t-1)$ after round $t-1$ is independent of the random variables Y_i for $H^*(t) \leq i < H^*(t-1)$ according to Lemma 4.5, the probability that $H^*(t) \leq H^*(t-1) - \xi$ is at most $2B/2^\xi$, which proves Inequality (4.4).

Let $P_t := \sum_{i=1}^{t} \mathit{progress}(t)$. If $\mathcal{A}$ finds the median in T rounds, we have that $P_T = \ell$. For any $t \neq t'$, the random variables $\mathit{progress}(t)$ and $\mathit{progress}(t')$ are not independent. However, according to Lemma 4.5 and the above observation, it is possible to upper bound the random variables $\mathit{progress}(t)$ by independent random variables Z_t where $\mathbb{P}[Z_t = \log 2B + i - 1] = 1/2^i$ for $i \geq 1$. That is, independent random variables Z_t can be defined such that $\mathit{progress}(t) \leq Z_t$, and for which

$$\forall t:\ \mathbb{P}\big[Z_t \geq \xi\big] \leq \frac{2B}{2^\xi}. \tag{4.5}$$

The probability that the number of rounds T to compute the median is at most some value τ can be upper bounded by using a generalized Chernoff-type argument:

$$\begin{aligned}
\mathbb{P}[T \leq \tau] &= \mathbb{P}\left[\sum_{t=1}^{\tau} \mathit{progress}(t) \geq \ell\right] \\
&\leq \mathbb{P}\left[\sum_{t=1}^{\tau} Z_t \geq \ell\right] \\
&\overset{\gamma \geq 0}{=} \mathbb{P}\left[e^{\gamma \cdot \sum_{t=1}^{\tau} Z_t} \geq e^{\gamma \cdot \ell}\right] \\
&\leq \frac{\mathrm{E}\left[e^{\gamma \cdot \sum_{t=1}^{\tau} Z_t}\right]}{e^{\gamma \cdot \ell}} \;=\; \frac{\mathrm{E}\left[\prod_{t=1}^{\tau} e^{\gamma \cdot Z_t}\right]}{e^{\gamma \cdot \ell}} \qquad (4.6) \\
&= \frac{\prod_{t=1}^{\tau} \mathrm{E}\left[e^{\gamma \cdot Z_t}\right]}{e^{\gamma \cdot \ell}} \qquad (4.7) \\
&= \frac{1}{e^{\gamma \cdot \ell}} \cdot \prod_{t=1}^{\tau} \sum_{\xi=0}^{\infty} \mathbb{P}[Z_t = \xi] \cdot e^{\gamma \cdot \xi} \\
&= \frac{1}{e^{\gamma \cdot \ell}} \cdot \prod_{t=1}^{\tau} \left(\mathbb{P}[Z_t \geq 0] + \sum_{\xi=1}^{\infty} \mathbb{P}[Z_t \geq \xi] \cdot \left(e^{\gamma \cdot \xi} - e^{\gamma \cdot (\xi - 1)}\right) \right) \\
&\leq \frac{1}{e^{\gamma \cdot \ell}} \cdot \left(1 + \sum_{\xi=1}^{\log(2B)} \left(e^{\gamma \cdot \xi} - e^{\gamma \cdot (\xi-1)}\right) \right. \\
&\qquad \left. + \sum_{\xi=\log(2B)+1}^{\infty} \frac{2B}{2^\xi} \cdot \left(e^{\gamma \cdot \xi} - e^{\gamma \cdot (\xi-1)}\right) \right)^{\tau} \qquad (4.8) \\
&= \frac{1}{e^{\gamma \cdot \ell}} \cdot \left(e^{\gamma \cdot \log(2B)} + \sum_{\xi=\log(2B)+1}^{\infty} \frac{2B}{2^\xi} \cdot e^{\gamma \cdot \xi} \cdot \left(1 - \frac{1}{e^\gamma}\right) \right)^{\tau}.
\end{aligned}$$

Inequality (4.6) is obtained by applying Markov's inequality (Theorem 1.1), Equation (4.7) follows from the independence of different Z_t, and Inequality (4.8) is a consequence of Inequality (4.5). By setting $\gamma := \ln(\sqrt{2})$ we obtain

$$\begin{aligned}\mathbb{P}[T \leq \tau] &\leq \frac{1}{2^{\ell/2}} \cdot \left(\sqrt{2B} + \frac{2\sqrt{2B}}{2-\sqrt{2}} \cdot \left(1 - \frac{1}{\sqrt{2}}\right)\right)^{\tau} \\ &= \frac{(8B)^{\tau/2}}{2^{\ell/2}}.\end{aligned}$$

If we set $\tau := \log_{8B}(N)/c$ for a constant $c > 1$, we get that

$$\begin{aligned}\mathbb{P}\left[T \leq \frac{1}{3c} \cdot \log_{2B}(N)\right] &\overset{B\geq 1}{\leq} \mathbb{P}\left[T \leq \frac{\log_{8B}(N)}{c}\right] \\ &\leq \frac{(8B)^{1/2 \cdot \log_{8B}(N)/c}}{2^{\log(N)/2}} \\ &= \frac{1}{N^{\frac{1-1/c}{2}}},\end{aligned}$$

which proves the claimed lower bound for finding the median by means of a deterministic algorithm. The lower bound for randomized algorithms follows immediately from Yao's principle. Note that the lower bound for randomized algorithms could also be proved directly in exactly the same way as the deterministic lower bound. However, this would require to include randomness in all the definitions.

In order to obtain the lower bound for selecting an element with arbitrary rank $k < N$, we show that finding the element with rank $k < N$ is at least as hard as finding the median if both nodes start with k instead of N values. For this purpose, elements $u_1, \ldots, u_k$ and $v_1, \ldots, v_k$ are assigned to u and v as described above in such a way that finding the median of the $2k$ elements is difficult. The remaining elements are assigned such that $u_i \prec u_j$, $u_i \prec v_j$, $v_i \prec u_j$, and $v_i \prec v_j$ for all $i \leq k$ and $j > k$. If $k > N$, we get the lower bound by lower bounding the time to find the k^{th} smallest element with respect to the complementary order relation. □

Based on the above lower bound for generic two-party protocols, a lower bound for generic selection algorithms on general graphs can be proved. In the following, we again assume that every node possesses one element and that the objective is to find the k^{th} smallest of these n elements. In every round, every node can send B elements to each of its neighbors. The proof of the following theorem shows how to reduce the problem on general graphs to the two-party problem.

Theorem 4.7. *For every $n \geq 3$ there is a graph $G(D)$ of diameter D consisting of n nodes such that $\Omega(D \log_D \min\{k, n-k\})$ rounds are needed to find the k^{th} smallest element in expectation and with probability at least $1/(\min\{k, n-k\})^\delta$ for every constant $\delta < 1/2$. In particular, finding the median requires at least $\Omega(D \log_D n)$ time.*

Proof. It can certainly be assumed that $n \in \omega(D)$ since even finding the median of three values requires $\Omega(D)$ rounds. Without loss of generality, we can also assume that $k \leq n/2$.[4]

For the sake of simplicity, assume that $n - D$ is an odd number. Let $N := (n - D + 1)/2$. The graph $G(D)$ is defined as follows: $G(D)$ consists of two nodes u and v that are connected by a path of length $D - 2$ (i.e., this path contains $D - 1$ nodes including u and v). In addition, there are nodes $u_0, \ldots, u_{N-1}$ and $v_0, \ldots, v_{N-1}$ such that u_i is connected to u and v_i is connected to v for all $i \in \{0, \ldots, N-1\}$. Assume that only the leaf nodes u_i and v_i for $i \in \{0, \ldots, N-1\}$ hold elements and that the goal is to find the k^{th} smallest of these $2N$ elements. We can simply assign "dummy elements" that are larger than these $2N$ elements to all other nodes. Since only the leaves start with an element, it can further be assumed that all leaves $u_0, \ldots, u_{N-1}$ send their elements to u and all leaves $v_0, \ldots, v_{N-1}$ send their elements to v in the first round as this is the only possible useful communication. Thereby the problem reduces to finding the k^{th} smallest element of $2N$ elements on a path of length $D - 2$ where initially each of the two end nodes u and v of the path holds N elements. The graph $G(D)$ and the reduction is depicted in Figure 4.3. Note that the leaf nodes u_i and v_i of $G(D)$ do not need to further participate in a distributed selection protocol since u and v know everything their respective leaves know and can locally simulate all actions of their leaf nodes.

Let w be any node on the path and let $M_w(t)$ be a message that w sends in round t. Since we consider deterministic algorithms and because only u and v initially hold elements, $M_w(t)$ can be computed when knowing the elements that u and v sent to their neighbors in rounds prior to t. In fact, since information needs time $d(w, w')$ to be transmitted from a node w to a node w', $M_w(t)$ can be computed when knowing all elements u sends in rounds prior to $t - d(u, w) + 1$ and all elements v sends in rounds prior to $t - d(v, w) + 1$. In particular, u and v can compute their own messages of round t when knowing the elements sent by v and u in rounds prior to $t - (D - 2) + 1$, respectively.

Consider the following alternative model: A round lasts $D - 2$ time units. In every round, u and v can send a message containing $D - 2$ elements to all other nodes of the path (they need to send the same message to all nodes). According to the above observation, u and v can send all messages and locally

[4] For $k > n/2$, the theorem follows by symmetry.

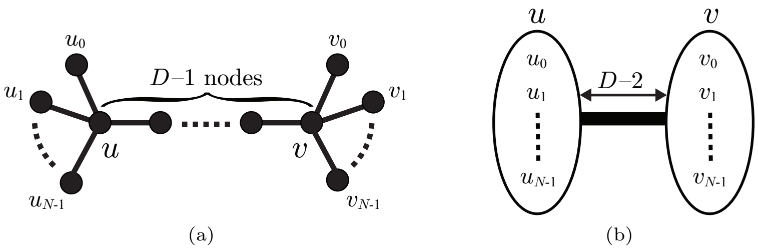

Figure 4.3: The graph $G(D)$ is depicted in Figure (a). Once u has received the elements from the nodes $u_0, \dots, u_{N-1}$ and v has obtained the elements from the nodes $v_0, \dots, v_{N-1}$, the problem reduces to finding the k^{th} smallest element among these $2N$ elements stored by u and v, where the distance between u and v is $d(u, v) = D - 2$, see Figure (b).

compute all communication on the path of rounds r for $(i-1) \cdot (D-2) < r \leq i \cdot (D-2)$ in round i of the alternative model. Thus, if it takes r rounds in the original model to find the k^{th} smallest element, then it takes $i \in \Theta(r/D)$ rounds in the alternative model. Apparently, the time needed in the alternative model is exactly the time needed by a two-party protocol if every round lasts $D-2$ time units and if in every round $D-2$ elements can be transmitted. Since $n \in \omega(D)$ and therefore $2N = n(1 - o(1))$, Theorem 4.6 can be applied, which implies that the time complexity in the alternative model is $\Omega(\log_D \min\{k, n-k\})$. Hence it follows that the time complexity in the original model is lower bounded by $\Omega(D \log_D \min\{k, n-k\})$. □

Not only the algorithms but also the lower bound can be generalized to the more general setting where each node holds an arbitrary number of elements. If the total number of elements in the system is $N \geq n$, we then obtain a lower bound of $\Omega(D \log_D \min\{k, N-k\})$ on the time complexity to find the k^{th} smallest element.

This result demonstrates that there are holistic functions that can be computed quite efficiently in a distributed manner, albeit not as efficiently as algebraic and distributive aggregate functions. Distributed k-selection is thus not a truly holistic function in the sense that there are solutions that find the k^{th} smallest element more efficiently than by simply gathering all elements at one place. Unfortunately, as the following chapter reveals, this is not true for other relevant holistic functions. Since the time complexity to compute these functions is large, one must resort to approximation algorithms.

Chapter 5

Holistic Aggregate Functions Beyond Selection

SINCE the frequency of elements plays a pivotal role in this chapter, we abandon the assumption that $x_i \neq x_j$ for all $x_i, x_j \in \mathcal{S}$. Moreover, each node may hold any number of elements. Thus, we again consider the general setting where all $|\mathcal{S}| = N$ elements are arbitrarily distributed among the n nodes and where each element may occur multiple times.

We introduce the following notation: Let ϕ_i be the frequency of element x_i in $\mathcal{S}$ for all $x_i \in X$, and let σ denote the number of *distinct elements* in $\mathcal{S}$. For any given multiset $\mathcal{S}$ the elements are ordered in decreasing order of their frequency, i.e., we have that $\phi_1 \geq \phi_2 \geq \ldots \geq \phi_\sigma$. The *frequency moments* of $\mathcal{S}$ are defined as follows.

Definition 5.1 (Frequency Moments)**.** *The ℓ^{th}* frequency moment *F_ℓ of a multiset $\mathcal{S}$ containing ϕ_i elements of type $x_i \in X$ is defined as $F_\ell = \sum_{i=1}^{\sigma} \phi_i^\ell$.*

Observe that $\sigma = F_0$ is exactly the number of distinct elements in the multiset, and $F_1 = \sum_{i=1}^{\sigma} \phi_i = N$ is the total number of elements. Hence, F_0 is the result of the aggregate function DISTINCT and F_1 is the result of COUNT. While COUNT can be computed easily and efficiently as shown in Section 3.2, the complexity of computing the number of distinct elements has not been discussed yet. Note that DISTINCT is simply the *duplicate insensitive* version of COUNT as computing the number of distinct elements is the same as COUNT without counting any element more than once.

Unfortunately, a simple reduction from the *set disjointness problem* reveals that in general the exact solution of DISTINCT cannot be computed efficiently: If two nodes hold $N/2$ unique elements each, a well-known communication complexity result states that the two nodes must send $\Omega(N)$ bits in order to determine whether or not the two sets are disjoint even when using randomization [28, 32, 50]. If an algorithm is able to determine the number of distinct elements in this scenario by communicating $o(N)$ bits, the result implies that the two sets are disjoint if and only if the number of distinct elements is N. Thus, such an algorithm would also solve the set disjointness problem by sending $o(N)$ bits, which entails that $\Omega(N)$ bits need to be exchanged in order to compute DISTINCT [46]. Given an upper bound on the message size of b bits, this bit complexity implies a lower bound on the time complexity of $\Omega(N/b)$. Fortunately, while there is no efficient algorithm to find the exact solution, an accurate estimate can be computed efficiently as we will see in Section 5.1.

Naturally, one might also be interested in the complexity of computing the frequency moments F_ℓ for $\ell \geq 2$. In a *streaming model*, where a single entity processes all elements sequentially and only one pass through all elements is allowed, the problem of minimizing the *space complexity*, i.e., the number of bits that this entity must store locally in order to approximate the frequency moments F_ℓ as accurately as possible, is well understood. For any $\ell \neq 1$, computing a constant-factor approximation of F_ℓ deterministically requires $\Omega(N)$ space, and any randomized algorithm that computes the exact solution also has a space complexity of $\Omega(N)$ [1]. These negative results entail that one can only hope to approximate F_ℓ well using randomization. While F_0 and F_2 can be approximated using little space, it has been shown in a series of contributions that in order to obtain an estimate $\hat{F}_\ell$ that differs from F_ℓ merely by a constant factor requires $\Omega(N^{1-2/\ell})$ space for any $\ell > 2$ [1, 3, 11]. This result has been complemented by algorithms that compute an estimate $\hat{F}_\ell$ for which it holds that $|\hat{F}_\ell - F_\ell| \leq \varepsilon F_\ell$ with constant probability and have a space complexity of $\tilde{\mathcal{O}}(N^{1-2/\ell})$ for any $\ell > 2$ and a constant $\varepsilon > 0$ [7, 27].[1]

In the second part of this chapter, the time complexity to compute the element that occurs most often is studied. This element is commonly referred to as the *mode*. Note that the frequency moment F_∞, which is usually defined as $F_\infty := \lim_{\ell\to\infty}(F_\ell)^{1/\ell} = \max_{1\leq i\leq\sigma} \phi_i = \phi_1$, is the frequency of the mode. In the streaming model, the above result states that approximating F_∞ requires to store $\Omega(N)$ bits. In our distributed computing model, a lower bound to compute F_∞ follows again by reduction from the set disjointness problem: If an algorithm computes F_∞ by exchanging $o(N)$ bits, it also solves the set disjointness problem by sending $o(N)$ bits as the sets are disjoint if and only if $F_\infty = 1$. Thus, the time complexity of a distributed algorithm to compute F_∞ is lower bounded by $\Omega(N/b)$. Apparently, any algorithm that determines the mode x_1 must also have a time complexity of $\Omega(N/b)$, as the frequency F_∞ of the mode can be computed trivially by counting the number of occurrences of x_1 in $\mathcal{O}(D)$ time, independent of N, once the mode x_1 is known. In fact, given that it may take D time to route the solution to the node that is interested in the mode, the time complexity is also lower bounded by D, which implies that the time complexity is $\Omega(N/b + D)$. Unfortunately, while

[1] Naturally, the space complexity depends on the constant ε.

the number of distinct elements can be approximated accurately and efficiently, any approximation algorithm that computes the aggregate function `MODE` and that returns an element whose frequency is at most a constant times smaller than the true mode has a time complexity of $\Omega(N/b)$ as we will see in Section 5.2.5. Given this result for arbitrary distributions of the elements, one can only hope to get an efficient algorithm for specific distributions. In Section 5.2, upper and lower bounds on the time complexity of computing the mode are derived that take the frequency distribution of the elements into account. The proposed algorithm has the interesting property that it requires estimates for both F_0—the algorithm presented in Section 5.1 can be used for this purpose—and the second frequency moment F_2, which must also be approximated in a distributed fashion.

5.1 Number of Distinct Elements

The most basic approach to approximate the number of distinct elements is to acquire a sufficiently large random *sample* of all elements and use this sample to compute an approximation. The database (and statistics) community proposed numerous sampling-based estimators of F_0 of an attribute in a relation.[2] Since the size of the required sample depends on the distribution of the elements, there has been a lot of work on designing more (space) efficient algorithms that achieve a low approximation ratio with bounded probability [1, 4, 19, 22]. More specifically, the goal is to compute an estimate $\hat{F_0}$ that deviates from F_0 by at most εF_0 with probability at least $1-\delta$ for two parameters $\varepsilon, \delta > 0$. If an algorithm meets this requirement, we say that it *(ε, δ)-estimates* F_0.

Definition 5.2 ((ε, δ)-estimator)**.** *For any $\varepsilon, \delta \in (0,1)$, a randomized algorithm $\mathcal{A}$ (ε, δ)-*estimates *the correct solution y of a problem if the output of $\mathcal{A}$ is a random variable Y for which it holds that*

$$\mathbb{P}[|Y-y| > \varepsilon y] < \delta.$$

If the elements are all chosen from a domain of size m, an algorithm can (ε, δ)-estimate F_0 by storing $\Theta(\log m)$ bits.[3] It has even been shown that F_0 can be approximated by storing an auxiliary memory of merely $\mathcal{O}(\log \log m)$ bits [17]. The next section shows that F_0 can also be approximated accurately and efficiently in the distributed computing model by means of a simple randomized algorithm.

[2] See, e.g., [24] for a brief introduction and an empirical comparison.

[3] If the number of elements N is larger than m, hashing can be used to reduce the description of the elements to $\mathcal{O}(\log m)$ bits. Thus, the number of elements is not crucial.

5.1.1 Algorithm

The presented algorithm $\mathcal{A}^{F_0}$ is a simple adaptation of a streaming algorithm [4]. All algorithms in this chapter make extensive use of random *hash functions* $h : X \to I$ that map the elements to an appropriate image I. The hash functions have in common that they map all elements to an element in I uniformly at random, i.e., for all h taken from such a family of hash functions $\mathcal{H}$ it holds that $\mathbb{P}[h(x) = i] = 1/|I|$ for all $x \in X$ and all $i \in I$.

Algorithm $\mathcal{A}^{F_0}$ uses hash functions that are *injective* with high probability. The number of distinct elements is clearly upper bounded by the total number of elements N. If $|I| = N^{\lambda+2}$, $\lambda \geq 1$, for all $h \in \mathcal{H}$, then the probability that any hash function h is injective is at least

$$\begin{aligned}
\mathbb{P}[h \text{ is injective}] &= 1 \cdot \left(1 - \frac{1}{|I|}\right) \cdot \left(1 - \frac{2}{|I|}\right) \cdots \left(1 - \frac{N-1}{|I|}\right) \\
&\geq 1 \cdot e^{-\frac{2}{|I|}} \cdot e^{-\frac{2\cdot 2}{|I|}} \cdots e^{-\frac{2\cdot(N-1)}{|I|}} \\
&= e^{-\frac{N(N-1)}{|I|}} \\
&\overset{|I|=N^{\lambda+2}}{>} e^{-\frac{1}{N^\lambda}} \geq 1 - \frac{1}{N^\lambda}.
\end{aligned}$$

We used that $e^{-2y} \leq 1 - y$ for all $y \in [0, 0.79\ldots)$ and $e^y \geq 1 + y$ for all $y \in \mathbb{R}$. If the cardinality of I is $N^{\lambda+2}$, encoding the hash values requires $\lambda+2$ times more bits than the encoding of the N elements. Since a small constant can be chosen for λ, any message may also contain a constant number of hash values without increasing its size substantially. It is further assumed that there is a known total order on the hash values, i.e., there is an order relation $\prec$ such that for all $i, i' \in I$ the nodes know whether $i \prec i'$ or $i' \prec i$.

The algorithm first computes two parameters $s := \lceil 16/\varepsilon^2 \rceil$ and $r := 2\lceil \log_{5/2}(2/\delta) \rceil + 1$. Subsequently, $\mathcal{A}^{F_0}$ chooses a hash function $h_1 \in \mathcal{H}$ and executes the subroutine *getMinimumHashed*(h_1, s), which returns the s smallest hash values of all elements in the network when all elements are hashed using the hash function h_1. This subroutine works basically the same way as algorithm $\mathcal{A}^{dist}$ (Algorithm 3.2) for the aggregate function `MIN`. Apart from the fact that only the hash values are considered (and not the elements themselves), the main difference is that *getMinimumHashed* does not only return the smallest, but the s smallest values. This generalization can be accomplished easily: If a leaf node holds more than s elements, it sends the s smallest hash values in increasing order to its parent, otherwise it simply sends all values. Upon receiving at least one hash value from each child, an inner node always forwards the smallest value among the hash values of its children and its own hash values that it has not already sent. Let $i_{11}, \ldots, i_{1s}$ denote the s smallest hash values when the hash function h_1 is applied. The algorithm takes the largest among these s hash values i_{1s} and computes an estimate $\hat{F}_0^{(1)} := s|I|/i_{1s}$. This procedure is repeated for randomly chosen

Algorithm 5.1 $\mathcal{A}^{F_0}$: Given the parameters s and r, compute the number of distinct elements.

```
for $j := 1, \ldots, r$ in parallel do
    $[i_{j1}, \ldots, i_{js}] :=$ getMinimumHashed$(h_j, s)$
    $\hat{F}_0^{(j)} := s|I|/i_{js}$
end for
return median$([\hat{F}_0^{(1)}, \ldots, \hat{F}_0^{(r)}])$
```

hash functions $h_2, \ldots, h_r$ *in parallel*, i.e., after forwarding h_1 and s to all children, $\mathcal{A}^{F_0}$ sends h_2 in the next step and then h_3 etc. Since the r computations are independent, there is no need to wait for a single computation to terminate. Once the r^{th} computation is complete, the initiating node has r estimates $\hat{F}_0^{(1)}, \ldots, \hat{F}_0^{(r)}$ from which it selects the (unique) median as the final estimate $\hat{F}_0$. The steps of $\mathcal{A}^{F_0}$ are summarized in Algorithm 5.1. It remains to prove that $\mathcal{A}^{F_0}$ (ε, δ)-estimates the number of distinct elements F_0 as desired.

5.1.2 Analysis

For the sake of simplicity, it is assumed that $\varepsilon \leq 1/2$. This is not a severe restriction as basically the same techniques can be used for any $\varepsilon < 1$ by using slightly different arguments.[4] The following theorem states the main result of this section.

Theorem 5.3. *If $s := \lceil 16/\varepsilon^2 \rceil$ and $r := 2\lceil \log_{5/2}(2/\delta) \rceil + 1$, algorithm $\mathcal{A}^{F_0}$ (ε, δ)-approximates the number of distinct elements F_0 on any connected graph G of diameter D. The time complexity of $\mathcal{A}^{F_0}$ is*

$$\mathcal{O}\left(D + \left(\frac{1}{\varepsilon^2}\right) \log\left(\frac{1}{\delta}\right)\right).$$

Proof. First, we prove that each $\hat{F}_0^{(j)}$ lies in the range $[(1-\varepsilon)F_0, (1+\varepsilon)F_0]$ with constant probability. If $\hat{F}_0^{(j)} > (1+\varepsilon)F_0$, we have that $i_{js} < \frac{s|I|}{(1+\varepsilon)F_0}$, i.e., there are at least s elements that hashed to a value smaller than

$$\frac{s|I|}{(1+\varepsilon)F_0} \leq \frac{(1-\varepsilon/2)s|I|}{F_0}.$$

The probability for each element to be hashed to a value below $(1 - \varepsilon/2)s|I|/F_0$ is upper bounded by $(1-\varepsilon/2)s/F_0$. Let $x_1, \ldots, x_\sigma$ denote the $\sigma = F_0$ distinct elements that occur in $\mathcal{S}$. Furthermore, let the random variable Y_i be 1 if x_i is hashed to a value below $(1-\varepsilon/2)t|I|/F_0$ and 0 otherwise.

[4] Moreover, one may not wish to get an estimate that is more than 50% off the correct value anyway.

The random variable $Y := \sum_{i=1}^{F_0} Y_i$ is the sum of these random variables Y_i. Since all Y_i are independent, identically distributed (i.i.d.) Bernoulli trials, it holds that $Var(Y) \leq \mathbb{E}[Y] \leq (1-\varepsilon/2)s$. The probability that $\hat{F}_0^{(j)} > (1+\varepsilon)F_0$ is upper bounded by the probability that Y is larger than s. Chebyshev's inequality (Theorem 1.2) and the fact that $s \geq 16/\varepsilon^2$ imply that this probability is at most

$$\mathbb{P}[Y > s] \leq \mathbb{P}[|Y - \mathbb{E}[Y]| > \varepsilon s/2] \leq \frac{(1-\varepsilon/2)s}{\varepsilon^2 s^2/4} < \frac{4}{\varepsilon^2 s} \leq \frac{1}{4}.$$

If $\hat{F}_0^{(j)} < (1-\varepsilon)F_0$, it holds that $i_{js} > \frac{s|I|}{(1-\varepsilon)F_0}$, i.e., there are less than s hash values that are smaller than

$$\frac{s|I|}{(1-\varepsilon)F_0} \leq \frac{(1+2\varepsilon)s|I|}{F_0}$$

because $\varepsilon \leq 1/2$. Let the random variable Z_i be 1 if the element x_i is hashed to a value below $(1+2\varepsilon)s|I|/F_0$ and 0 otherwise. The random variable Z is defined as $Z := \sum_{i=1}^{F_0} Z_i$ for which it holds that $Var(Z) \leq \mathbb{E}[Z] \leq (1+2\varepsilon)s$. Moreover, we have that

$$\frac{s|I|}{(1-\varepsilon)F_0} \geq \frac{(1+\varepsilon)s|I|}{F_0}.$$

Given that the probability to hash to any $i \in I$ is $1/|I|$, it holds that $\mathbb{E}[Z_i] \geq (1+\varepsilon)s/F_0 - 1/|I|$. By definition, we have that $s \geq 16/\varepsilon^2$ and $\varepsilon \leq 1/2$, and thus $\varepsilon s \geq 32$. This observation together with the definition that $|I| = N^{\lambda+2} \geq N \geq F_0$ imply that

$$\frac{1}{|I|} \leq \frac{1}{F_0} = \frac{\varepsilon s}{F_0} \cdot \frac{1}{\varepsilon s} \leq \frac{\varepsilon s}{32 F_0}.$$

Thus, we get that $\mathbb{E}[Z] \geq (1 + 31/32\varepsilon)s$. Since $\hat{F}_0^{(j)}$ is smaller than $(1-\varepsilon)F_0$ only if Z is smaller than s, we get the following bound on the probability that $\hat{F}_0^{(j)}$ is too small:

$$\mathbb{P}[Z < s] \leq \mathbb{P}[|Z - \mathbb{E}[Z]| > 31/32\varepsilon s] \leq \frac{(1+2\varepsilon)s}{(\frac{31}{32})^2 \varepsilon^2 s^2} \leq \frac{2}{(\frac{31}{32})^2 16} < \frac{1}{7}.$$

Hence, the probability that $\hat{F}_0^{(j)}$ is not within the desired range is less than $1/4 + 1/7 < 2/5$. Consider the final estimate $\hat{F}_0$, which is the median of all estimates $\hat{F}_0^{(1)}, \ldots, \hat{F}_0^{(r)}$. If the median does not lie in the desired range, e.g., because $\hat{F}_0 < (1-\varepsilon)F_0$, then all the smaller estimates $\hat{F}_0^{(1)}, \ldots, \hat{F}_0^{((r-1)/2)}$ are also too small. Since all estimates are independent, the probability that all of these $(r-1)/2 = \lceil \log_{5/2}(2/\delta) \rceil$ estimates are too small is upper bounded by $(2/5)^{\lceil \log_{5/2}(2/\delta) \rceil} \leq \delta/2$. The same argument also applies if $\hat{F}_0 > (1+\varepsilon)F_0$.

Thus, by means of a union bound, $\hat{F}_0$ does not deviate from F_0 by more than εF_0 with probability at least $1 - \delta$ as desired.

Let $D(T_G)$ again denote the diameter of the BFS spanning tree T_G that is used as a routing infrastructure. Each leaf v is informed about its task to compute s hash values and forward them to its parent at the latest after $D(T_G)$ time. Once v has sent these values, it can immediately send the s hash values for the next hash function as the information about which hash function is to be used next must have arrived in the meantime.[5] Thus, each leaf node has sent all requested values at the latest at time $D(T_G) + rs$. Its parent receives the smallest values from all its children by time $D(T_G) + 1$ and can forward the smallest value in its subtree at time $D(T_G) + 2$. After $D(T_G) + 1 + rs$ time, it has forwarded all rs hash values. Inductively, we get that the s smallest values for all r hash functions must have arrived at the initiating node after $2D(T_G) + rs \leq 4D + rs \in \mathcal{O}(D + (1/\varepsilon^2)\log(1/\delta))$ time, which proves the claimed bound on the time complexity. □

Note that an estimate $\hat{F}_0 \in [(1-\varepsilon)F_0, (1+\varepsilon)F_0]$ for any constant ε can be found in $\mathcal{O}(D + \log N)$ time w.h.p.

5.1.3 Discussion

While the exact solution for the aggregate function `DISTINCT` cannot be computed efficiently, we saw that F_0 can be approximated quite well. Instead of counting the number of distinct elements, it may be desirable to compute other aggregates without considering any element more than once. An example for such an aggregate function is the sum of all distinct elements. A reduction from the set disjointness problem and the observation that routing information may take D time again reveal that the time complexity of the *distinct summation problem* is also $\Omega(D + N/b)$, which implies that one can only approximate the correct value. A straightforward solution is to hash each element x_j, where x_j is interpreted as a natural number,[6] to x_j hash values $i_j^{(1)}, \ldots, i_j^{(x_j)}$. If $i_j^{(k)} \neq i_m^{(\ell)}$ for all elements $x_j, x_m \in \mathcal{S}$, and all $k \in \{1, \ldots, x_j\}$ and $\ell \in \{1, \ldots, x_m\}$, then the number of distinct hash values is exactly the sum of all distinct elements, which can be approximated using algorithm $\mathcal{A}^{F_0}$. This simple approach has several drawbacks. First, the local computations are no longer negligible as each node must compute x_j hash values for each locally stored element x_j. Moreover, if $x_{\max}$ is the largest element in $\mathcal{S}$, the total number of hash values is at most $N \cdot x_{\max}$, which entails that the time complexity to approximate the sum of distinct elements

[5] If we cannot send messages over an edge in both directions at the same time, the algorithm can simply distribute information about all r hash functions in $D(T_G) + r$ time first.

[6] If the elements are not natural numbers, an injective function can be used to map all elements to natural numbers.

w.h.p. is $\mathcal{O}(D + \log N + \log x_{\max})$. This complexity is unacceptable if $x_{\max}$ is large, e.g., $x_{\max} \in \Omega(2^N)$.

A solution to this problem is to use *counting sketches* [19], which are bitmaps $\mathcal{B}$ of length $\ell \in \mathcal{O}(\log F_0) \subseteq \mathcal{O}(\log N)$. There is a 1 in the bitmap at position p if there is an element whose random hash value starts with $p-1$ zero bits and its first 1 is at position p. The largest index in the bitmap that stores a 1 is approximately $\log F_0$. Note that two different bitmaps $\mathcal{B}_1, \mathcal{B}_2$, which are induced by two different sets of elements, can be combined easily: If $\mathcal{B}_i[p]$ is the bit at position p in a $\mathcal{B}_i$, then the bits of the resulting bitmap $\mathcal{B}$ are simply set to $\mathcal{B}[p] := \mathcal{B}_1[p] \vee \mathcal{B}_2[p]$ for all $p \in \{1, \ldots, \ell\}$. Instead of hashing each element x_j to x_j hash values, roughly the first $\log x_j$ bits are set to 1 immediately, as these values are set to 1 in the hashing process w.h.p. The remaining bits are then set according to the hash values of a random sample. It can be shown that all x_j hash values can be "inserted" into the bitmap in $\mathcal{O}(\log^2 x_j)$ local computational steps [14]. By aggregating $\mathcal{O}(\log N)$ of these bitmaps and choosing again the median as the final estimate, the initiating node gets an estimate of the sum of all distinct elements that is at most a factor $1 + \mathcal{O}(\varepsilon)$ off the correct value w.h.p. in $\mathcal{O}(D + \log N)$ time. Apparently, this scheme offers an alternative solution to the problem of computing the number of distinct elements. Note that it is not required to send the entire bitmap as only the largest index that contains a 1 is used as an estimate for $\log F_0$. Thus, it suffices to send $\mathcal{O}(\log \log N)$ bits for each estimate.[7] Given the estimate for the sum of distinct elements and an estimate for F_0, we immediately get an estimate for the average of all distinct elements. Furthermore, by hashing each element x_j to x_j^2 hash values, still in polylogarithmic time, an estimate for the sum of all distinct *squared* elements can be obtained. This estimate can then be used to approximate the variance of the distinct elements.

As a final note, the median of all distinct elements can be approximated using the technique introduced in Chapter 4. In each recursive step, the number of distinct elements within particular intervals must be approximated and these values determine where the median lies (with a certain probability). Note that counting the number of distinct elements in m intervals costs $\mathcal{O}(D + m \log N)$ time, which entails it is best to choose only a constant number of random elements in each phase if $D \in \mathcal{O}(\log N)$. In this case, the time complexity to approximate the median is $\mathcal{O}(\log^2 N)$ w.h.p. Naturally, if $D \geq c \cdot \log N$ for a sufficiently large constant c,[8] $\Theta(D/\log N)$ random elements can be chosen in each phase, which results in a time complexity of $\mathcal{O}(D \log_{D/\log N} N)$.

[7] An alternative approach to computing the sum of distinct elements is to use a *range-efficient* algorithm to compute F_0 [5, 48], where each element is hashed to a certain *interval* $[i_1, i_2]$ instead of hashing the elements to random hash values in a particular domain. This technique is not further discussed.

[8] Of course, the integer c depends on the parameters s and r used in algorithm $\mathcal{A}^{F_0}$.

5.2 Mode: The Most Frequent Element

As mentioned before, the time complexity to find the most frequent element grows linearly with the number of elements. In the first section, a deterministic algorithm is discussed, which is essentially asymptotically optimal. Since the *skewness* of the distribution can reasonably be expected to affect the efficiency of an algorithm to compute the mode, a randomized algorithm is presented in the subsequent section whose time complexity depends on the frequency distribution of the elements [30]. Skewed distributions naturally arise in various contexts. For example, the frequencies of terms on web pages, files in file-sharing networks etc., are distributed according to a *power law* [10, 42, 59]. In Section 5.2.4, we illustrate that the time complexity of the algorithm is fairly low for such distributions. Moreover, in Section 5.2.5 a lower bound is proved that takes the frequency distribution into account.

5.2.1 Deterministic Algorithm

There is a straightforward deterministic algorithm to find the mode executed on the pre-computed spanning tree T_G. Again, it is assumed that there is a total order on all the elements, i.e., $x_1 \succ \ldots \succ x_\sigma$ for all $\sigma = F_0$ distinct elements occurring in $\mathcal{S}$. The algorithm starts at the leaves of the tree which send element-frequency pairs $\langle x_i, \phi_i \rangle$ to their parents in increasing order, with respect to the order relation $\succ$ on the elements, starting with the smallest element that they possess. Any inner node v stores these pairs received from its children and sums up the frequencies for each distinct element. Node v forwards $\langle x_i, \phi_i \rangle$, where ϕ_i is the accumulated frequency of x_i in the subtree rooted at v, to its parent as soon as v has received at least one pair $\langle x_j, \phi_j \rangle$ from each of its children such that $x_j \succ x_i$ or $x_j = x_i$. Any node v sends $\langle x_i, \phi_i \rangle$ to its parent at time $t \leq h + i$ where h is the height of the subtree rooted at v. This claim clearly holds for the leaves as each leaf can send the i^{th} smallest element x_i at the latest at time i. Inductively, a node v thus receives at least the i^{th} smallest element after $h + i - 1$ time, after which it can forward the element including the accumulated frequency to its parent. Observe that there is no congestion as node v has already sent all smaller element-frequency pairs in earlier rounds. Thus, the algorithm terminates after at most $\mathcal{O}(D + F_0)$ time. Note that this algorithm does not only compute the mode x_1 and its frequency ϕ_1, but also the frequencies of all other elements.

5.2.2 Randomized Algorithm

For the sake of simplicity, we assume that the nodes know the frequency moments F_0 and F_2, as well as the frequency ϕ_1 of the mode when describing the algorithm, which we will refer to as $\mathcal{A}^{mode}$. While an estimate of F_0 can

Algorithm 5.2 countElementsInBins(h): Given a hash function h, node v computes and forwards the number of elements that map to -1 and 1 in its subtree, including its own elements $x_1, \ldots, x_\ell$.

```
$c_0 := |\{x_i \in \{x_1, \ldots, x_\ell\} \mid h(x_i) = -1\}|$
$c_1 := |\{x_i \in \{x_1, \ldots, x_\ell\} \mid h(x_i) = 1\}|$
if $\mathcal{N}_v(T_G) \setminus \{p_v\} = \emptyset$ then
    send $\langle c_0, c_1 \rangle$ to $p_v$
else
    for all $v_j \in \mathcal{N}_v(T_G) \setminus \{p_v\}$ in parallel do
        $\langle c_0^{(j)}, c_1^{(j)} \rangle :=$ countElementsInBins($h$)
    end for
    send $\langle c_0, c_1 \rangle + \sum_{v_j \in \mathcal{N}_v(T_G) \setminus \{p_v\}} \langle c_0^{(j)}, c_1^{(j)} \rangle$ to $p_v$
end if
```

be computed using $\mathcal{A}^{F_0}$, an estimate of F_2 can also be obtained efficiently, and ϕ_1 can be approximated in parallel to the computation of the mode as we will see.

Algorithm $\mathcal{A}^{mode}$ also uses hash functions, but its hash functions are not injective: The image I of all hash functions solely consists of the two values -1 and 1, i.e., each hash function maps every element to one of the two values with equal probability. We can think of the mapping process as putting each element in one of two bins. The basic idea behind algorithm $\mathcal{A}^{mode}$ is the following. Each node in the graph stores a local counter $c(x_i)$ for each of its elements $x_1, \ldots, x_\ell$ and two counters c_0 and c_1. All counters are initially set to zero. For a certain hash function h, the algorithm computes the total number c_0 and c_1 of all elements that hash to -1 and 1, respectively. In other words, the algorithm determines how many elements end up in each of the two bins "-1" and "1". The nodes use these values to increment each counter $c(x_i)$ for all local elements $x_1, \ldots, x_\ell$ by the number of elements that have been mapped to the same bin as element x_i. The idea is to determine the mode by repeating this procedure using different hash functions. Since the mode is likely to end up in the larger bin more often than the other elements, the counter $c(x_1)$ will increase faster than the counters of the other elements. After a first phase, which reduces the set of candidates for the mode, the frequency of each remaining candidate is computed separately in a second phase. The time complexity is bounded by the time required to find a small set of candidates and by the time to check these candidates.

We will now study each step of the algorithm in greater detail. The initiating node selects r_1 hash functions $h_1, \ldots, h_{r_1}$ where $h_i : X \to \{-1, 1\}$. The parameter r_1 will be determined later in the analysis of the algorithm. In the following, the number of elements that hashed to the two bins is represented as a tuple $\langle c_0, c_1 \rangle$, where c_i denotes the number of elements that have

Algorithm 5.3 $\mathcal{A}^{mode}$: Given the parameters r_1 and r_2, compute the most frequent element.

```
mode := $\emptyset$; $\phi^{mode} := 0$
*Phase (1):*
**for** $i := 1, \ldots, r_1$ **in parallel do**
    $\langle c_0, c_1 \rangle$ := countElementsInBins($h_i$)
    distribute($\langle c_0, c_1 \rangle$)
**end for**
*Phase (2):*
$\{x_1, \ldots, x_{r_2}\}$ := getPotentialModes($r_2$)
**for** $i := 1, \ldots, r_2$ **in parallel do**
    $\phi_i$ := getFrequency($x_i$)
    **if** $\phi_i > \phi^{mode}$ **then**
        mode := $x_i$; $\phi^{mode} := \phi_i$
    **end if**
**end for**
**return** mode
```

been mapped to bin $i \in \{0, 1\}$. These tuples are accumulated by means of a convergecast on the spanning tree using the subroutine *countElementsInBins*, which is parameterized by the chosen hash function h: Once the leaves have received information about the hash function, their elements $x_1, \ldots, x_\ell$ are hashed and put into one of the two bins, i.e., c_0 is set to the number of elements that mapped to the first bin and c_1 is set to the number of the remaining elements. This tuple is sent to the parent node, which accumulates the tuples from all its children and adds its own tuple. The resulting tuple is forwarded recursively towards the initiating node. As in Chapter 3, let p_v denote the temporary parent for this computation, i.e., v received the information about the hash functions from p_v. Recall that $\mathcal{N}_v(T_G)$ is the set of v's neighbors in the pre-constructed spanning tree T_G. The sum of two tuples $\langle c'_0, c'_1 \rangle$ and $\langle c''_0, c''_1 \rangle$ is defined as $\langle c_0, c_1 \rangle$, where $c_i = c'_i + c''_i$ for $i \in \{0, 1\}$. This subroutine is summarized in Algorithm 5.2.

Once the root has computed the final tuple $\langle c_0, c_1 \rangle$, this tuple is sent to all nodes along the edges of the spanning tree. Any node that receives this *distribute* message forwards it to its children and updates its local counters according to the following rule: For all elements x_i that mapped to the larger of the two bins, its counter $c(x_i)$ is increased by $|c_0 - c_1|$. This procedure is repeated for all other chosen hash functions $h_2, \ldots, h_{r_1}$. Note that all steps can be carried out *in parallel* for all r_1 hash functions, i.e., the initiating node can issue one of the r_1 procedure calls in each communication round. Once the r_1 results have been obtained and the tuples have all been distributed, Phase (1) of the algorithm is completed.

In the second phase, the r_2 elements—the parameter r_2 will also be specified later—with the largest counters are accumulated at the root using the procedure *getPotentialModes*. In this procedure, the nodes always forward the element x_i, including $c(x_i)$, if its counter is the largest among all those whose element has not yet been sent. Moreover, an element is only forwarded if its counter is among the r_2 largest counters ever forwarded to the parent node. Once these elements arrive at the root, the root issues a request to count the individual frequencies of all those elements, and the element with the largest frequency is returned as the mode. The entire algorithm is given in Algorithm 5.3.

5.2.3 Analysis of $\mathcal{A}^{mode}$

In Phase (1), $\mathcal{A}^{mode}$ executes r_1 iterations, where a different hash function $h_i \in \{h_1, \dots, h_{r_1}\}$ assigns all the elements $x_j \in \mathcal{S}$ to one of the two bins in each iteration. In the first step, the number r_1 of required hash functions to substantially reduce the set of candidates for the mode is determined. The following helper lemma will be useful.

Lemma 5.4. *For $i = 1, \dots, \sigma$, let Y_i be a random variable for which*

$$Y_i = \begin{cases} y_i, & \text{with } \mathbb{P} = 1/2 \\ -y_i, & \text{with } \mathbb{P} = 1/2 \end{cases}$$

and let all variables $Y_1, \dots, Y_\sigma$ be independent. If $Y = \sum_{i=1}^{\sigma} Y_i$ and $F_2[Y] = \sum_{i=1}^{\sigma} y_i^2$ is the second frequency moment of a set with frequencies $y_1, \dots, y_\sigma$, then it holds that

$$\mathbb{P}\left[Y \geq \lambda\sqrt{F_2[Y]}\right] \leq e^{-\lambda^2/2}.$$

Proof.

$$\begin{aligned} \mathbb{P}\left[Y \geq \lambda\sqrt{F_2[Y]}\right] & \overset{\gamma>0}{\leq} \frac{\mathbb{E}\left[e^{\gamma Y}\right]}{e^{\gamma\lambda\sqrt{F_2[Y]}}} = \frac{\prod_{i=1}^{\sigma}\mathbb{E}\left[e^{\gamma Y_i}\right]}{e^{\gamma\lambda\sqrt{F_2[Y]}}} \qquad (5.1) \\ & = \frac{\prod_{i=1}^{\sigma}\frac{e^{\gamma y_i}+e^{-\gamma y_i}}{2}}{e^{\gamma\lambda\sqrt{F_2[Y]}}} = \frac{\prod_{i=1}^{\sigma}\cosh(\gamma y_i)}{e^{\gamma\lambda\sqrt{F_2[Y]}}} \\ & \leq \frac{\prod_{i=1}^{\sigma} e^{\gamma^2 y_i^2/2}}{e^{\gamma\lambda\sqrt{F_2[Y]}}} = \frac{e^{\gamma^2\sum_{i=1}^{\sigma} y_i^2/2}}{e^{\gamma\lambda\sqrt{F_2[Y]}}} \\ & \leq e^{-\lambda^2/2}. \end{aligned}$$

Inequality (5.1) follows again from Markov's inequality (Theorem 1.1). Furthermore, we used that $(e^x + e^{-x})/2 = \cosh(x) \leq e^{x^2/2}$. Finally, the last step follows by setting $\gamma := \lambda/\sqrt{F_2[Y]}$. Note that $F_2[Y] = \mathrm{Var}(Y)$. □

The goal of Phase (1) is to reduce the set of elements that are potentially the mode to a small set of size r_2. The following lemma bounds the number r_1 of hash functions required to ensure that the counter of the mode is larger than the counter of a large fraction of all elements.

Lemma 5.5. *If $r_1 := 32\lceil (F_2/\phi_1^2)\ln(2F_0/\varepsilon)\rceil$, then it holds for all x_i for which $\phi_i < \phi_1/2$ that $c(x_i) < c(x_1)$ with probability at least $1-\varepsilon$.*

Proof. First, only the events where the mode x_1 and the element x' with the maximum frequency among all elements whose frequency is less than half of the frequency ϕ_1 of the mode are put into different bins are considered. All other elements are added randomly to one of the bins, and this procedure is repeated r_1 times. Alternatively, we can say that there are $(\sigma-2)r_1$ elements $\alpha_1,\ldots,\alpha_{(\sigma-2)r_1}$ that are placed randomly into the two bins. It holds that $\sum_{i=1}^{(\sigma-2)r_1}\alpha_i^2 < r_1 \cdot F_2$. Before the elements $\alpha_1,\ldots,\alpha_{r_1(\sigma-2)}$ are put into the bins, it holds that $c(x_1) > c(x') + r_1 \cdot \phi_1/2$. In order to ensure that $c(x_1) > c(x')$ after all elements have been placed in one of the bins, the probability that the other elements offset this imbalance of at least $r_1 \cdot \phi_1/2$ must be small. Let the Bernoulli variable Z_i indicate into which bin the element α_i is placed. In particular, if $Z_i = -1$, the element is put into the bin where the mode is, and if $Z_i = 1$, the element is placed into the other bin. By setting $r_1 := 8\lceil (F_2/\phi_1^2)\ln(2F_0/\varepsilon)\rceil$ and applying Lemma 5.4 we get that

$$\begin{aligned}\mathbb{P}\left[\sum_{i=1}^{(\sigma-2)r_1}\alpha_i Z_i \geq r_1\cdot\phi_1/2\right] &< e^{-\frac{r_1^2\left(\frac{\phi_1}{2}\right)^2}{2\sum_{i=1}^{(\sigma-2)r_1}\alpha_i^2}}\\ &< e^{-\frac{r_1\phi_1^2}{8F_2}}\\ &\leq \frac{\varepsilon}{2F_0}.\end{aligned}$$

In order to ensure that the elements x_1 and x' are often placed into different bins, the number of rounds is increased to $r_1 := 32\lceil (F_2/\phi_1^2)\ln(2F_0/\varepsilon)\rceil$. Let the random variable $\mathcal{U}$ denote the number of times that the two elements are placed into the same bin. The Chernoff bound for the upper tail (Theorem 1.4), implies that the probability that $32\lceil (F_2/\phi_1^2)\ln(2F_0/\varepsilon)\rceil$ rounds do not suffice to bound the probability of failure to $\varepsilon/(2F_0)$ because x_1 and x' are put into different bins less than $8\lceil (F_2/\phi_1^2)\ln(2F_0/\varepsilon)\rceil$ times, is bounded by

$$\begin{aligned}\mathbb{P}[\mathcal{U} > 24\lceil (F_2/\phi_1^2)\ln(2F_0/\varepsilon)\rceil] &= \mathbb{P}\left[\mathcal{U} > \left(1+\frac{1}{2}\right)\mathbb{E}[\mathcal{U}]\right]\\ &< e^{-(F_2/\phi_1^2)\ln(2F_0/\varepsilon)}\\ &< \frac{\varepsilon}{2F_0}.\end{aligned}$$

Thus, the probability that $c(x_1) > c(x')$ is at least $1 - \varepsilon/F_0$. Let $\Upsilon = \{x_i : \phi_i < \phi_1/2\}$ denote the set of all elements for which we want to prove that their counters are lower than the counter of the mode. The probability that any element x^* in this set has a counter larger than the mode is

$$\mathbb{P}[\exists x^* \in \Upsilon : c(x^*) > c(x_1)] < \sum_{x \in \Upsilon} \mathbb{P}[c(x) > c(x_1)] < F_0 \cdot (\varepsilon/F_0) = \varepsilon,$$

which concludes the proof. □

Given this bound on the number of hash functions, we are now in the position to prove the following theorem.

Theorem 5.6. *If* $r_1 := 32\lceil (F_2/\phi_1^2)\ln(2F_0/\varepsilon)\rceil$ *and* $r_2 := \lceil 4F_2/\phi_1^2\rceil$*, the time complexity of* $\mathcal{A}^{mode}$ *to compute the mode with probability at least* $1-\varepsilon$ *on any connected graph* G *of diameter* D *is*

$$\mathcal{O}\left(D + \frac{F_2}{\phi_1^2}\log\frac{F_0}{\varepsilon}\right).$$

Proof. In Phase (1) of the algorithm, r_1 hash functions are applied to all elements and the sum of elements mapped to each bin is accumulated. Algorithm $\mathcal{A}^{mode}$ is similar to $\mathcal{A}^{F_0}$ in that it handles all these hash functions in parallel as opposed to computing the resulting bin sizes for each hash function sequentially, i.e., the number of communication rounds required is upper bounded by $\mathcal{O}(D + r_1)$ and not $\mathcal{O}(D \cdot r_1)$. Each result is distributed back down the tree in order to allow each node to update the counters of its elements, which requires $\mathcal{O}(D)$ time. Hence, the time complexity of the first phase is bounded by $\mathcal{O}(D + r_1)$.

In Phase (2), the r_2 elements with the largest counters are accumulated at the root, and the element with the highest number of occurrences out of this set is returned as the mode. The procedure *getPotentialModes* performs this operation in $\mathcal{O}(D + r_2)$ time, as the i^{th} largest value arrives at the root after at most $D(T_G) + i$ time. Naturally, the frequency of r_2 elements can also be determined in $\mathcal{O}(D + r_2)$ time, and thus the entire phase requires $\mathcal{O}(D + r_2)$ time.

The parameter r_2 has to be large enough to ensure that the mode is in fact in this set of elements with high probability. Let $\bar{\Upsilon} = \{x_i : \phi_i \geq \phi_1/2\}$ be the complement of Υ. According to Lemma 5.5, if $r_1 := \lceil 32(F_2/\phi_1^2)\ln(2F_0/\varepsilon)\rceil$, then the mode is in $\bar{\Upsilon}$ with probability at least $1 - \varepsilon$. We have that $|\bar{\Upsilon}|(\phi_1/2)^2 \leq \sum_{x_i \in \bar{\Upsilon}} \phi_i^2 \leq F_2$, implying that $|\bar{\Upsilon}| \leq 4F_2/\phi_1^2$. Thus, by setting $r_2 := \lceil 4F_2/\phi_1^2\rceil$, both phases complete after $\mathcal{O}(D + (F_2/\phi_1^2)\log(F_0/\varepsilon))$ time, and the mode is found with probability $1 - \varepsilon$. □

An (ε, δ)-estimator of F_2 can be computed using a mixture of the techniques introduced for $\mathcal{A}^{dist}$ and $\mathcal{A}^{mode}$: As in algorithm $\mathcal{A}^{mode}$, hash functions that map elements to -1 and 1 are used. Instead of using two bins, a single value c_1 is computed distributively, again using a simple convergecast, which in the end stores $c_1 = \sum_{x \in \mathcal{S}} h_1(x)$ for a given hash function h_1. This step is repeated for hash functions $h_2, \ldots, h_s$, where $s := \lceil 16/\varepsilon^2 \rceil$ as in algorithm $\mathcal{A}^{dist}$. The *average* of the squared values $\frac{1}{s}\sum_{i=1}^{s} c_i^2$ is then chosen as an estimate $\hat{F}_2$ for F_2. It can be shown that $\mathbb{E}[c_i^2] = F_2$ and that $Var(c_i^2) = 2F_2^2$ for any such counter c_i [1]. Since $\hat{F}_2$ is the average of s independent random variables, the variance reduces to $Var(\hat{F}_2) = 2F_2^2/s \leq \varepsilon^2 F_2^2/8$. Using Chebyshev's inequality (Theorem 1.2), we get that the probability that $\hat{F}_2$ deviates from F_2 by more than εF_2 is bounded by

$$\mathbb{P}[|\hat{F}_2 - F_2| > \varepsilon F_2] \leq \frac{Var(\hat{F}_2)}{\varepsilon^2 F_2^2} \leq \frac{1}{8}.$$

As in Section 5.1, this probability can be reduced to any value δ by computing $\mathcal{O}(\log(1/\delta))$ estimates and using the median as the final estimate. The time complexity is obviously again $\mathcal{O}(D + (1/\varepsilon^2)\log(1/\delta))$. Thus, estimators of both $\hat{F}_0$ and $\hat{F}_2$ can be computed beforehand.

An estimator $\hat{\phi}_1$ for ϕ_1 can be obtained as follows. We know that after counting the elements in the two bins for $r_1 = 32\lceil (F_2/\phi_1^2)\ln(2F_0/\varepsilon)\rceil$ different hash functions, the first phase of the algorithm may terminate. After each distribution of $\langle c_0, c_1 \rangle$ we determine the frequency ϕ_i of the element x_i whose counter is currently the largest in $\mathcal{O}(D)$ time and use ϕ_i as the new estimator for $\hat{\phi}_1$ if it exceeds the largest previously encountered frequency. The bin sizes for another hash function are computed as long as $32\lceil (\hat{F}_2/\hat{\phi}_1^2)\ln(2\hat{F}_0/\varepsilon)\rceil < r$, where r denotes the number of hash functions that have been used so far. Once this inequality does no longer hold, we can conclude that the algorithm must have used a sufficient number of hash functions, as $\hat{\phi}_1 \leq \phi_1$. Note that the algorithm does not run much longer than needed, since $\hat{\phi}_1 \geq \phi_1/2$ after $32\lceil (\hat{F}_2/\hat{\phi}_1^2)\ln(2\hat{F}_0/\varepsilon)\rceil \geq r_1$ rounds with probability at least $1 - \varepsilon$.

5.2.4 Power Law Distributions

A widely studied distribution is the *power-law distribution* $p(x) \propto x^{-\alpha}$ for some constant $\alpha > 0$ [10, 42, 59]. The frequencies can be normalized so that $\phi_i := 1/i^{\alpha}$. The second frequency moment $F_2 = \sum_{i=1}^{\sigma} \phi_i^2$ strongly depends on the constant α: It holds that $F_2 \in \Theta(\sigma^{1-2\alpha})$ for $\alpha < 1/2$, $F_2 \in \Theta(\log \sigma)$ for $\alpha = 1/2$, and $F_2 \in \Theta(1)$ for $\alpha > 1/2$. Since $\phi_1 = 1$, the time complexity T of $\mathcal{A}^{mode}$ for power-law distributions is:

$$T \in \begin{cases} \mathcal{O}\left(D + F_0^{1-2\alpha} \cdot \log(F_0/\varepsilon)\right) & \text{if } \alpha < 1/2 \\ \mathcal{O}\left(D + \log F_0 \cdot \log(F_0/\varepsilon)\right) & \text{if } \alpha = 1/2 \\ \mathcal{O}\left(D + \log(F_0/\varepsilon)\right) & \text{if } \alpha > 1/2 \end{cases}$$

If $\alpha < 1/2$, $\mathcal{A}^{mode}$ needs polynomial time, whereas for $\alpha \geq 1/2$, the mode can be found in polylogarithmic time. The mode can be determined in $\mathcal{O}(D + \log N)$ time w.h.p. if $\alpha > 1/2$. In this case, the asymptotic time complexity to compute the mode (w.h.p.) is the same as the complexity to obtain the required approximations of both the number F_0 of distinct elements and the second frequency moment F_2.

5.2.5 Lower Bound

In this section, a lower bound on the time complexity to compute the mode is proved that matches the time complexity of algorithm $\mathcal{A}^{mode}$ up to logarithmic factors. A lower bound on the bit complexity of a variation of the set disjointness problem is used to prove the bound: Assume that there are d nodes $v_1, \ldots, v_d$ that hold sets $S_1, \ldots, S_d$ of elements whose cardinality is $|S_1| = \ldots = |S_d| = s$, i.e., the total number of elements is $N = sd$. Either the sets are pairwise disjoint or there is a single element s such that $S_i \cap S_j = \{x\}$ for all $1 \leq i < j \leq d$. In a series of work it has been shown that the number of bits that need to be exchanged in this scenario in order to find this particular element x is at least $\Omega(s/\log d) = \Omega(N/(d \log d))$ [1, 3, 11]. The following theorem states this result.

Theorem 5.7. *If each node $v_i \in \{v_1, \ldots, v_d\}$ holds $s \geq d$ elements, the total number of bits that the nodes have to communicate in order to distinguish between the case where their sets are pairwise disjoint and the case where they intersect in exactly one element is $\Omega(s/\log d)$. This lower bound also holds for randomized algorithms with error probability $\varepsilon < 1/2$.*

This theorem enables us to show that there is a graph G of diameter D and a frequency distribution with maximum frequency ϕ_1 and second frequency moment F_2 such that the time complexity to compute the mode is $\Omega(D + F_2/(\phi_1^2 b \log \phi))$. Recall that b denotes the number of bits that can be transmitted in a single message.

Theorem 5.8. *For any $\phi_1, F_2 \in \mathbb{N}$, where $F_2 \geq \phi_1^2 > 1$, there is a frequency distribution $\phi_1 > \phi_2 \geq \phi_3 \geq \ldots \geq \phi_\sigma$ of elements whose second frequency moment is $F_2 = \sum_{i=1}^{\sigma} \phi_i^2$ such that for every $D > 1$ there is a graph G of diameter D and a distribution of the elements among the nodes of G such that the time complexity to compute the mode is*

$$\Omega\left(D + \frac{F_2}{\phi_1^2 b \log \phi_1}\right),$$

where b is the maximum number of bits that can be sent in a single message.

Proof. Setting $\phi_2 := \phi_3 := \ldots := \phi_\sigma := 1$ results in a second frequency moment of $F_2 = \sum_{i=1}^{\sigma} \phi_i^2 = \phi_1^2 + \sigma - 1$. Let σ' be the largest integer $\sigma' \leq \sigma$ such that $\sigma' - 1$ is a multiple of ϕ_1, i.e., we have that $\sigma' > \sigma - \phi_1$. Assume that the algorithm knows that the elements $x_{\sigma'+1}, \ldots, x_\sigma$ need not be considered. In this case, it remains to solve the problem for the frequency distribution $\phi_1, \ldots, \phi_{\sigma'}$. Note that finding the mode for this distribution can be at most as hard as finding the mode for the frequency distribution $\phi_1, \ldots, \phi_\sigma$ because additional information cannot make the problem harder. Without loss of generality, it can further be assumed that $F_2 \geq 2\phi_1^2$ since otherwise the statement of the theorem reduces to the trivial bound of $\Omega(D)$. This assumption implies that $\sigma - 1 \geq \phi_1^2$ and thus $\sigma' > \phi_1^2 - \phi_1 + 1$.

Let G be a star graph consisting of $n = \phi_1 + 1$ nodes, i.e., G consists of an inner node v of degree $\delta(v) = \phi_1$ and ϕ_1 leaves $v_1, \ldots, v_{\phi_1}$. There is a total of $N = \phi_1 + \sigma' - 1$ elements; element x_1 occurs ϕ_1 times, all other $\sigma' - 1$ elements occur only once. These elements are distributed among the ϕ_1 leaf nodes such that every leaf node receives x_1 exactly once and $(\sigma' - 1)/\phi_1$ of the other elements. Let S_i be the set of elements of leaf node v_i. Since it holds that $S_i \cap S_j = \{x_1\}$ for any $i \neq j$ and $s \geq d$, i.e., $1+(\sigma'-1)/\phi_1 \geq \phi_1$ due to the assumption that $F_2 \geq 2\phi_1^2$, Theorem 5.7 can be applied, which states that the total number of bits the nodes $v_1, \ldots, v_{\phi_1}$ have to communicate in order to find x_1 is $\Omega(s/\log\phi_1) = \Omega(\sigma'/(\phi_1 \log\phi_1))$. This reduction holds for the following reason. Any algorithm $\mathcal{A}$ that computes x_1 can be used to solve the set disjointness problem on d nodes: If $\mathcal{A}$ terminates without returning a value x_1, we know that the sets $S_1, \ldots, S_d$ are pairwise disjoint. If $\mathcal{A}$ returns a value x_1, a simple test reveals whether $x_1 \in S_i$ for all $i \in \{1, \ldots, d\}$ by exchanging a logarithmic number of additional bits. The facts that $\sigma' > \sigma - \phi_1$ and $\sigma > \phi_1$ imply that $\sigma' > \sigma(1 - 1/\phi_1)$, and thus $\Omega(\sigma'/(\phi_1 \log\phi_1)) = \Omega(\sigma/(\phi_1 \log\phi_1))$. Hence, since at least one of the leaf nodes has to send at least $\Omega(\sigma/(\phi_1^2 \log\phi_1))$ bits and each message may contain at most b bits, the time complexity is at least $\Omega(\sigma/(\phi_1^2 b \log\phi_1))$.

A graph G of diameter D on which the lower bound becomes $\Omega(D + \sigma/(\phi_1^2 b \log\phi_1))$ can be obtained by replacing each edge in G by a path of length $D/2$. A lower bound on the time complexity of $\Omega(D + F_2/(\phi_1^2 b \log\phi_1))$ follows because $\sigma > \phi_1^2$ and thus $F_2 = \phi_1^2 + \sigma - 1 < 2\sigma$. □

The ratio between the proven upper and lower bound to compute the mode is $\mathcal{O}(b \log\phi_1 \log(F_0/\varepsilon))$. Given that both ϕ_1 and F_0 are upper bounded by N, this ratio is (only) polylogarithmic in N if $b \in \mathcal{O}(\log N)$. More importantly, the lower bound also reveals that the time complexity of any distributed algorithm that is able to find an element whose frequency is at most a specific constant $c > 1$ lower than the frequency of the mode is linear in N: If $\phi_1 \in \Theta(c)$, we get a lower bound of $\Omega(N/(bc \log c))$, where we used the simple fact that $F_2 \geq N$. This result implies that there is no distributed algorithm that (ε, δ)-estimates the frequency of the mode and that has a time complexity that is substantially better than $\mathcal{O}(N)$ for any reasonable choice of the maximum number of bits that can be sent in a single message.

5.2.6 Discussion

All studied aggregate functions have in common that the result is only a single element or value. One reason to study only such aggregate functions is that the time complexity to acquire a particular set of k elements is inherently at least $\Omega(D + k)$, which implies that k must be small lest the time complexity becomes too large. Nevertheless, one could still be interested in obtaining solutions to problems whose result is a set of elements. A variation of the problem to compute the mode is to find a set of k elements such that the frequency ϕ_i of every element x_i in this set is at least $(1 - \varepsilon)\phi_k$ for a parameter ε. There is an algorithm for the streaming model that solves this problem with probability $1 - \delta$ whose space complexity is (roughly) $\mathcal{O}((k + F_2/(\varepsilon\phi_k)^2)\log(N/\delta))$ [12]. In fact, the algorithm achieves the stronger guarantee that every element x_i whose frequency is greater than $(1 + \varepsilon)\phi_k$ is in the set. The simple idea behind the algorithm is to compute a counter $c = \sum_{x \in \mathcal{S}} h(x) = \sum_{i=1}^{\sigma} h(x_i)\phi_i$, where h is again a random hash function that maps all elements to either -1 or 1. Note that the same counter is used to estimate F_2. This counter c can be used to estimate the frequency of any element x_i because

$$\begin{aligned}\mathbb{E}[c \cdot h(x_i)] &= \mathbb{E}\left[\left(\sum_{j=1}^{\sigma} h(x_i)\phi_i\right) h(x_i)\right] \\ &= \mathbb{E}[\phi_i h(x_i)^2] + \sum_{j=1, j\neq i}^{\sigma} \mathbb{E}[\phi_j h(x_j) h(x_i)] \\ &= \phi_i.\end{aligned}$$

As the variance is large, several estimates are again required in order to get the desired result with probability $1 - \delta$. This algorithm can be transformed into a distributed algorithm that finds the mode with basically the same time complexity as $\mathcal{A}^{mode}$. Thus, this simple technique offers an alternative solution to the problem of computing the mode. An advantage of this algorithm over $\mathcal{A}^{mode}$ is that the used hash functions must only be *pairwise independent*, i.e., any two functions h_i and h_j are independent, but any three or more hash functions may not be independent. Throughout this chapter, it has been assumed that all hash functions are independent. Note, however, that the same asymptotic results can also be proved using weaker random hash functions [1, 4, 30]. Roughly speaking, these results indicate that pseudorandom hash functions suffice to achieve good bounds.

While it is somewhat misleading and thus not quite appropriate to consider distributed selection a holistic aggregation problem in the distributed computation model, because efficient algorithms for this problem exist, we found that, e.g., computing the number F_0 of distinct elements proved to be more difficult in that the time complexity to compute the correct solution is $\Omega(N/b)$, i.e., the straightforward strategy to simply collect all elements locally is basically the best possible strategy.[9] A problem with this property, we might say, deserves to be called a holistic aggregate function. However, while F_0 can be approximated easily and efficiently, there are problems, such as computing the frequency of the mode, for which any constant-factor approximation algorithm has a time complexity of $\Omega(N/b)$, which implies that there are even harder problem classes and that referring to all of these aggregation problems as holistic aggregate functions is an imprecise categorization in the distributed computation model. Of course, one possible categorization would be to define classes exactly according to the problems discussed in this thesis, i.e., apart from the class of problems whose time complexity is polylogarithmic in n or N (and linear in D), there is a class of problems that have a time complexity of $\tilde{\Omega}(N)$ but can be approximated efficiently, and a class for problems that (essentially) cannot even be approximated in sublinear time. The interested reader is encouraged to think about ways to refine this rudimentary categorization. This concludes our discussion of distributed aggregation.

[9] Technically, since $b \in \mathcal{O}(\log N)$ if $n \in \mathcal{O}(N)$, the lower bound only states that the time complexity is $\Omega(N/\log N)$.

Part II

Synchronization

Chapter 6

Clock Synchronization: An Introduction

6.1 Motivation

COORDINATION among the participants of a distributed system is often required in order to solve computational problems in a distributed fashion. One of the most basic coordination primitives is *time*, i.e., coordination can be achieved by establishing a common notion of time in the distributed system. However, an intrinsic problem of distributed systems is that the participants do not have access to global parameters. In other words, we cannot assume that all participants have access to a global clock that provides the network with real time information. A natural solution to this problem is to equip each participant with its own clock according to which it may execute the scheduled tasks. As mentioned in Section 1.2, the problem of this approach is that clocks drift apart since they run at slightly different and potentially variable clock rates, which implies that the clocks have to be *synchronized* from time to time. To this end, a *clock synchronization algorithm* must be used to correct the *clock skews*, i.e., the differences between the clock values caused by the different clock rates. In order to detect clock skews, any clock synchronization algorithm requires that timing information is exchanged among the participants.

By exchanging synchronization messages and manipulating its clock according to the received timing information, each participant synchronizes its clock with all participants with which it shares a communication link. Since these participants have communication channels to other participants, updates on current clock values may propagate through the entire network. This propagation ought to ensure that the clock skew between *any* two participants in the network is small even if they cannot communicate directly. Although minimizing the clock skew in the entire network is clearly desirable, for several distributed applications or protocols the primary requirement is that the clocks between participants that share a communication link are always synchronized as well as possible. A prominent example for shared medium networks is *time division multiple access* (TDMA), a channel access method that allows the participants to share the same channel by assigning disjoint time slots to the participants during which they may send their messages. Apparently, these time slots must be well synchronized in order to avoid simultaneous transmissions. Moreover, since the participants that are far away from each other use different channels, strong guarantees on the degree of synchronization to their clocks are not mandatory. Examples of distributed applications that require each participant to be well synchronized primarily with all participants in its vicinity include monitoring or tracking applications, where some event is only registered locally in some part of the network. The following simple example further illustrates this point. Consider a wireless sensor network whose main objective is to track the movement of a particular object (or several objects) and determine its speed along the way. For this purpose, the sensor nodes store proximity information together with a timestamp if the object is close. Given the recordings of the sensor nodes, locally well synchronized clocks suffice to reconstruct the path of the object even if there is a large skew between clocks of distant nodes in the network. On the contrary, if the clock skew between near sensor nodes is large, the movement cannot be tracked reliably.[1] The same reasoning applies to the measurement of the speed of the object. Generally, the requirement that clocks between participants that share a communication link are well synchronized is reasonable whenever occurrences of events are only of local importance and do not bear any (immediate) significance for participants that are not close-by. Of course, an ideal clock synchronization algorithm guarantees that the clock skew between participants sharing a communication link and between any two participants is as small as possible.

Synchronizing clocks is challenging due to the variable (and uncontrollable) delay between the time when synchronization messages are sent and the time when the recipients are able to process them. If the sender of a message attaches information about its current local time, i.e., the time on its own clock, the recipient receives this message after it has been delayed for an unknown time. The unknown delay renders it impossible for the recipient to correctly determine the current time at the sender. Moreover, even if the recipient knew *exactly* how long the message was in transit, it still could not figure out the current time at the sender since the receiver does not know the rate of progress of the sender's clock from the moment the message was sent until it was received. These simple observations lead to the conclusion that synchronizing clocks perfectly is in general impossible. Given this negative

[1] For general distributed systems, *locality* refers to the ability of nodes to communicate directly. Note that in the context of sensor networks, locality is tantamount to physical proximity as nodes can communicate directly if and only if they are physically close.

result, an obvious question is how large the clock skews can become, both between directly connected participants and between participants that do not share a communication link, regardless of the strategy that is employed to correct them. However, the main objective must be to find an algorithm guaranteeing that the clock skews are as small as possible at all times. Of course, it is desirable that the guaranteed upper bounds on the clock skews match the lower bounds, i.e., the clock skews that cannot be prevented by any algorithm.

Since timing information is conveyed only through message exchange and the clocks may drift apart as long as no new information is received, the *message frequency* has an impact on the achievable bounds on the clock skews. If the algorithm may trigger synchronization messages itself, it can influence the accuracy of the information that any participant has about the clock values of the other participants directly. However, if the algorithm may only attach timing information to messages that are sent by other applications, a technique commonly referred to as "piggybacking", the accuracy depends on how often the other applications communicate. Naturally, the goal is to bound the clock skews even if the information about other clock values in the network is outdated to a certain extent. If it is possible to achieve strong bounds on the clock skews while keeping the message frequency low, then piggybacking can be employed as long as the other applications communicate reasonably often.

Apparently, clock synchronization is a problem with many parameters including the clock drift rates, the message delays, and the message frequency; all of which influence the feasible degree of synchronization. There are several optimization criteria that can be considered, such as minimizing the clock skews between directly connected participants and participants that can only communicate if the messages are relayed by other participants. Another desirable property is that the clock synchronization algorithm ensures that the clock values are never changed abruptly when new timing information is received, i.e., the clocks ought to run smoothly at all times. Such a restriction clearly inhibits the ability of a clock synchronization algorithm to react to clock skews, which renders the problem of keeping the clocks synchronized even harder. Moreover, the clock synchronization problem can be studied in different models; the most prominent models are discussed in the subsequent section.

After introducing the relevant definitions, the goal of the following chapters is to provide an understanding of how the clock skew bounds depend on the various parameters. In particular, after illustrating the difficulty of the problem by analyzing several simple algorithms in Chapter 7, an algorithm that achieves strong bounds on the worst-case clock skews for several models is presented in Chapter 8, followed by a study of how much skew between the clocks is inevitable in a worst-case scenario in Chapter 9.

6.2 Model and Definitions

As in the previous part, the distributed system is modeled as an arbitrary connected graph $G = (V, E)$ of diameter D, where nodes represent computational devices and edges represent bidirectional communication links. We no longer impose the normalization that each message arrives after at most 1 time unit, i.e., it is assumed that there is a constant $\mathcal{T} > 0$ such that for any sent message the time that passes until the recipient can act upon it may be any value in the range $[0, \mathcal{T}]$. The upper bound $\mathcal{T}$ is referred to as the *maximum delay* in the following. While the bound $\mathcal{T}$ is unknown to the algorithm, we assume that the nodes know an upper bound $\hat{\mathcal{T}} \in \mathcal{O}(\mathcal{T})$ on the maximum delay. This assumption is not critical as we will point out in Section 8.2.3.

Each node v is equipped with a *hardware clock* H_v whose value at *real time* t is denoted by $H_v(t)$, i.e., $H_v : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ is a strictly monotonically increasing function. For the sake of simplicity, it is assumed that all nodes start their clocks at real time $t = 0$. The value of the hardware clock of v is 0 at time 0 and $H_v(t) := \int_0^t h_v(\tau)\, d\tau$ afterwards, where $h_v(\tau)$ is the *hardware clock rate* of v at time τ. The clock rates can vary over time, but there is a constant $\varepsilon \in (0, 1)$ such that the following condition holds.

$$\forall v \in V\ \forall t : 1 - \varepsilon \leq h_v(t) \leq 1 + \varepsilon.$$

The parameter ε determines the largest possible clock drift between any two hardware clocks, which is 2ε if the hardware clock rate of one clock is $1 + \varepsilon$ and the clock rate of the other is $1 - \varepsilon$. While the exact value of ε is unknown, it is assumed that the nodes know an upper bound $\hat{\varepsilon} \in \mathcal{O}(\varepsilon)$ that is strictly smaller than 1.

We assume that no node v can manipulate its hardware clock, i.e., v can only read its hardware clock value $H_v(t)$ at any time t. Since the hardware clocks cannot be modified, each node computes a *logical clock value* which is based on its hardware clock and the information it received about its neighbors' logical clock values. Thus, each node v additionally has a *logical clock* L_v, which is also a function $L_v : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ whose value is 0 at time 0 as well. The goal is to minimize the *skew* between the logical clocks.

As mentioned before, it is desirable that the clocks always behave normally in the sense that the logical clock values may not change dramatically in a short time, which means that a clock synchronization algorithm must strive to keep the logical clock rates within pre-specified bounds at all times. Formally, for certain constants $\beta > \alpha > 0$, the following condition must be satisfied.

$$\forall v \in V\ \forall t < t' :\ \alpha(t' - t) \leq L_v(t') - L_v(t) \leq \beta(t' - t). \tag{6.1}$$

Increasing (or lowering) the clock rates of the logical clocks allows the nodes to correct clock skews in the network. Ideally, the logical clocks behave just like the hardware clocks even in the presence of clock skews, albeit with a slightly worse clock drift than the drift of the hardware clocks, i.e., $\alpha \in 1-\mathcal{O}(\varepsilon)$ and $\beta \in 1+\mathcal{O}(\varepsilon)$. Obviously, Condition (6.1) implies that clocks are not allowed to run backwards. Since this condition severely restricts the actions an algorithm can take, the simple algorithms discussed in Chapter 7 only satisfy the weaker condition that the minimum progress rate is lower bounded by a certain $\alpha > 0$, i.e., the maximum clock rate is unbounded. This simplification allows a node to increase its clock value instantaneously once it notices that it has fallen behind. If an algorithm is allowed to modify the value of its clock at discrete points in time, the interpretation of $L_v(t)$ at times when the clock value changes has to be clarified: At any time t when the clock value is increased, $L_v(t)$ is defined as the value *after* the algorithm has handled all events that occur at time t.[2] The same definition generally applies to all local variables that are modified at time t, also for algorithms that do not increase the logical clock value instantaneously.

Moreover, although all clocks may continually drift away from real time, an algorithm ought to ensure that the logical clock values are always within a linear envelope of real time. Therefore, it is further required that any algorithm satisfies the following condition.

$$\forall v \in V\ \forall t:\ (1-\varepsilon)t \leq L_v(t) \leq (1+\varepsilon)t. \tag{6.2}$$

It is easy to see that a better bound on the accuracy with respect to real time cannot be achieved in the absence of an external timer.

A clock synchronization algorithm $\mathcal{A}$ executed at node v specifies how the logical clock $L_v(t)$ of node v is adapted based on its hardware clock and the information received from its neighbors up to time t. Ideally, an algorithm satisfies both Condition (6.1) and Condition (6.2). Given an algorithm $\mathcal{A}$, an *execution* specifies the delays of all messages and also the hardware clock rates of all nodes at each point in time when $\mathcal{A}$ is executed on a given graph G. One optimization criterion is to minimize the worst-case clock skew that can occur between any two nodes in the graph G, which is referred to as the *global skew* in the following. Minimizing the worst-case clock skew between neighboring nodes in G, which is called the *local skew*, is the second fundamental optimization criterion [18]. The formal definitions of the global and the local skew are as follows:

Definition 6.1 (Global Skew)**.** *Given a connected graph $G = (V, E)$ and a clock synchronization algorithm $\mathcal{A}$, the* global skew *is defined as*

$$\sup_{\mathcal{E},\, v\in V,\, w\in V,\, t} \{L_v(t) - L_w(t)\},$$

where $\mathcal{E}$ is any execution of $\mathcal{A}$ on G.

[2] If there is more than one event at time t, the events may be handled in an arbitrary order.

Definition 6.2 (Local Skew)**.** *Given a connected graph $G = (V, E)$ and a clock synchronization algorithm $\mathcal{A}$, the* local skew *is defined as*

$$\sup_{\mathcal{E},\, v\in V,\, w\in\mathcal{N}_v,\, t} \{L_v(t) - L_w(t)\}\,,$$

where $\mathcal{E}$ is any execution of $\mathcal{A}$ on G.

As mentioned before, the goal of an algorithm $\mathcal{A}$ is to ensure the best possible bounds on both the global and the local skew on any graph G. Ideally, the algorithm guarantees strong bounds on the clock skews even if the nodes exchange information about their current state infrequently, i.e., the algorithm has a low *message frequency*, which is formally defined as follows.

Definition 6.3 (Message Frequency)**.** *The* message frequency *is the maximum number of messages that are sent in one time unit in the worst case for every legal input and every execution scenario.*

It is not only important to bound the number of sent messages, but also how much information each message must carry. Thus, the *maximum message size* also needs to be considered.

In our model, the nodes have to synchronize their clocks without access to a source of real time. This synchronization problem is called *internal synchronization* (see, e.g., [2, 26, 33]). The counterpart to this problem is *external synchronization* (see, e.g., [43, 45, 47]), where it is assumed that there is a dedicated node that provides the network with real time information, and the objective is to synchronize all clocks to this reference time. Although we focus on internal synchronization, the implications of the results for external synchronization are also discussed. Another prominent clock synchronization problem is synchronizing clocks in the presence of *faulty nodes* (see, e.g., [16, 25, 38]). While the algorithms introduced in the following chapters do not consider node or link failures, we will point out that simple modifications to the algorithm discussed in Chapter 8 enable it to cope with node and link failures as well.

Chapter 7

Simple Synchronization Algorithms

MOST work on the fundamental problem of synchronizing clocks in distributed systems primarily focuses on deriving techniques to bound the skew that may occur between any two clocks (see, e.g., [37, 45, 47, 60]). Surprisingly, while simple algorithms suffice to bound the worst-case clock skew between any two nodes, it is considerably more challenging to come up with algorithms guaranteeing an upper bound on the skew between the clocks of neighboring nodes that is substantially better than the bound on the skew between any two clocks. This is quite counterintuitive at first glance because one would expect that it is easy to maintain a small clock skew between the nodes that are able to exchange messages directly.

In this chapter, the bounds on the global and the local skew of a few straightforward synchronization algorithms are discussed. These algorithms have in common that they all satisfy Condition (6.2) and also the weak version of Condition (6.1) that does not impose any restrictions on the maximum clock rate (i.e., $\beta = \infty$). Thus, the algorithms ensure that the logical clock values are always within a linear envelope of real time and the minimum progress rate of each clock is at least a specific constant $\alpha > 0$.

In order to guarantee that all nodes periodically receive updates about their neighbor's clock values, we assume that every node automatically transmits a message containing its logical clock value to all neighboring nodes at the latest after its hardware clock advanced by $H_0 > 0$ since the last time when messages were sent. For example, each node may send a message to its neighbors whenever its hardware clock value reaches the next integer multiple of H_0. Of course, an algorithm may trigger the exchange of additional messages at different times. The parameter H_0 is important because it has a direct influence on the message frequency.[1]

[1] The parameter even determines the message frequency if messages are *only* sent when the hardware clock value has increased by H_0 since the last message was sent.

The simplest approach is probably to increase the logical clock value at the hardware clock rate and set the logical clock value immediately to L if a value L greater than the current logical clock value is received [33]. The updated clock value L is then forwarded to all neighbors immediately. Since the logical clock is always set to the largest known clock value, this algorithm is referred to as $\mathcal{A}^{max}$. Evidently, algorithm $\mathcal{A}^{max}$ satisfies Condition (6.2) because the minimum progress rate is the minimum hardware clock rate, which is lower bounded by $1-\varepsilon$, and no clock value is ever set to a value that exceeds the largest known clock value, whose progress rate is upper bounded by $1+\varepsilon$. The fact that the minimum progress rate is the hardware clock rate implies that Condition (6.1) is also satisfied for $\alpha = 1-\varepsilon$.

As mentioned before, the main problem of any synchronization algorithm is that messages may be delayed. Let the variable $L_v^w(t)$ denote v's estimate of w's clock value at real time t. These variables L_v^w are initialized to zero at time $t=0$. If a node v receives a new estimate $L_v^w = L_v^w(t)$ of w's clock value at a time t, this message may have been sent $\mathcal{T}$ time ago. If the clock skew was $\mathcal{T}$ at the time $t-\mathcal{T}$ when the message was sent and the logical clock of v increased by $\mathcal{T}$ until t, then the received estimate L_v^w is exactly v's clock value at time t. Hence, v does not have a reason to increase its clock value although the clock skew is (still) $\mathcal{T}$. Consequently, if a skew of $\mathcal{T}$ can be induced between any two neighboring nodes, no node may ever receive a clock value larger than its own and the global skew may reach $D\mathcal{T}$. In fact, Chapter 9 reveals that there is an execution of any algorithm on any graph G of diameter D that induces a global skew of roughly $D\mathcal{T}$. Not surprisingly, this skew occurs between two nodes v and w at distance $d(v,w) = D$.

The following example illustrates that $\mathcal{A}^{max}$ is a poor clock synchronization algorithm in that the clock skew between neighboring nodes can become large. Assume that the skew between two nodes v and w is $D\mathcal{T}$ at a certain time, where v is the node with the larger clock value, and that the message delays are suddenly reduced to zero, apart from the delays of the messages that the nodes $u \in \mathcal{N}_w$ send to w. In this scenario, all nodes except w increase their clock values instantaneously to v's clock value, which causes a clock skew of $D\mathcal{T}$ between w and all of its neighbors. The local skew is thus exactly as large as the global skew, even if ε and H_0 are arbitrarily small, i.e., the clocks drift at an arbitrarily small rate and messages are exchanged arbitrarily often. Since w receives the update after at most $\mathcal{T}$ time, one might think that the clock skew is $\Omega(D\mathcal{T})$ only for a short time after which the local skew is again reduced to a small constant. Unfortunately, the assumption that this problem is transient is not true: Instead of increasing all clock values to L_v, the message delays can be manipulated so that the skew between w and its neighboring nodes becomes only $D\mathcal{T}/c$ for some $c \in \mathbb{N}$. Once w increases its clock value by roughly $D\mathcal{T}/c$ after at most $\mathcal{T}$ time, the neighboring nodes may learn about clock values that are again $D\mathcal{T}/c$ larger due to reduced message delays. This simple step can be repeated c times, i.e., the clock skew remains approximately $D\mathcal{T}/c$ for $c\mathcal{T}$ time. In particular, by setting $c := \lfloor\sqrt{D}\rfloor$, the local skew is about $\sqrt{D}\mathcal{T}$ for roughly $\sqrt{D}\mathcal{T}$ time.

The problem with algorithm $\mathcal{A}^{max}$ is that solely the neighbor with the largest clock value is considered. This observation indicates that a node must also take the clock values of its other neighbors into account in order to achieve a better bound on the local skew.

7.1 Averaging Algorithm

A straightforward approach is to try to balance the clock skew between the own clock and the clocks of the neighboring nodes. In an attempt to minimize the maximum clock skew to some specific neighboring nodes, a node v may set its clock to the *average* value of these neighbors' clocks. Naturally, v can only set its clock to this average value if it exceeds its own clock value, otherwise v's clock value would become smaller, a violation of Condition (6.1). However, the algorithm that compares the average of the (estimated) clock values of *all* neighbors to its own clock value performs poorly as the following simple example shows. Assume that the clock of a node v always runs at the clock rate $1+\varepsilon$ and the clock of its sole neighbor w has a progress rate of 1 at all times. Node w has $n-2$ other neighbors $u_1, \dots, u_{n-2}$ that are only connected to w and whose clocks run at a clock rate of 1 as well, i.e., the graph is a star and w is the center node. If the message delays are always $\mathcal{T}$, w receives clock values that are $\mathcal{T}$ smaller than its own from all u_i, $i \in \{1, \dots, n-2\}$, and vice versa. Once the clock skew between v and w reaches $(n-1)\mathcal{T}$ at a time t, node w has received a clock value from v that is at most $(n-2)\mathcal{T}$ larger than its own due to the message delay $\mathcal{T}$. Since the estimates of all other $n-2$ neighbors are at least $\mathcal{T}$ smaller than its own clock value, the average of the estimated clock values of all the neighbors is still smaller than w's own clock value, implying that w does not increase its clock at time t or any earlier time. The global and the local skew can thus become $\Omega(n\mathcal{T})$ although the diameter of the graph is only 2. The clock skews when algorithm $\mathcal{A}^{max}$ is used on this graph are upper bounded by $\mathcal{O}(\mathcal{T})$ even if H_0 is large as we will see in the subsequent section.

A crucial observation is that it is not important how many neighbors seemingly have smaller clock values, but how large the clock skew appears to be to the neighboring node that is behind the most. Thus, instead of computing the average of all clock values, a node only determines the values of the two variables $\Lambda_v^{\uparrow}$ and $\Lambda_v^{\downarrow}$, which are defined as follows:

$$\begin{aligned}\Lambda_v^{\uparrow} &:= \max_{u \in \mathcal{N}_v}\{L_v^u - L_v\}\\ \Lambda_v^{\downarrow} &:= \max_{u \in \mathcal{N}_v}\{L_v - L_v^u\}\end{aligned}$$

If these variables are positive, $\Lambda_v^{\uparrow}$ denotes v's estimate of the largest clock

Algorithm 7.1 $\mathcal{A}^{avg}$: Compute the new clock value L_v after receiving L^w from node $w \in \mathcal{N}_v$.

```
if $L^w > L_v^w$ then
    $L_v^w := L^w$
    $\Lambda_v^{\uparrow} := \max_{u \in \mathcal{N}_v}\{L_v^u - L_v\}$
    $\Lambda_v^{\downarrow} := \max_{u \in \mathcal{N}_v}\{L_v - L_v^u\}$
    if $\Lambda_v^{\uparrow} > \Lambda_v^{\downarrow}$ then
        $L_v := L_v + (\Lambda_v^{\uparrow} - \Lambda_v^{\downarrow})/2$
        send $L_v$ to all $u \in \mathcal{N}_v$
    end if
end if
```

skew to a neighbor whose clock is ahead and $\Lambda_v^{\downarrow}$ is v's estimate of the largest clock skew to a neighbor whose clock is assumed to be behind. Whenever any node v obtains a message containing a clock value L^w from a node $w \in \mathcal{N}_v$ that is larger than the current estimate L_v^w, the estimate L_v^w is updated, and v determines whether the average of the largest and the smallest clock value among its neighbors' clock values is larger than its own clock value, i.e., whether $\Lambda_v^{\uparrow} > \Lambda_v^{\downarrow}$. If the average is indeed larger, v sets its logical clock to the average $L_v + (\Lambda_v^{\uparrow} - \Lambda_v^{\downarrow})/2$ and immediately sends its new clock value to its neighbors. This simple algorithm $\mathcal{A}^{avg}$ is given in Algorithm 7.1.

Unfortunately, while algorithm $\mathcal{A}^{avg}$ is better than the simple averaging algorithm described before, it fails to achieve better bounds on the clock skews than $\mathcal{A}^{max}$. On the contrary, the skew can become substantially larger when $\mathcal{A}^{avg}$ is executed on the simple graph $G^{list} := (V^{list}, E^{list})$, where $V^{list} = \{v_0, \ldots, v_D\}$ and $E^{list} = \{\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{D-1}, v_D\}\}$, regardless of when and how often messages are exchanged.

Theorem 7.1. *There is an execution of $\mathcal{A}^{avg}$ on the graph G^{list} of diameter D that causes a global skew of $D^2\mathcal{T}$.*

Proof. For all $i \in \{0, \ldots, D\}$, define $t_i := i^2\mathcal{T}/\varepsilon$. The message delays in the considered execution are always $\mathcal{T}$. The execution changes the progress rates of the hardware clocks at times $t_1, \ldots, t_D$ as follows. For all $i \in \{0, \ldots, D-1\}$, the hardware clock rate of v_j's clock at any time $t \in [t_i, t_{i+1})$ is defined as:

$$h_{v_j}(t) := \begin{cases} 1 + \varepsilon & \text{if } \ j > i \\ 1 & \text{else} \end{cases}$$

Since the logical clocks increase at the hardware clock rates, the logical clock value of v_i at any time t is

$$L_{v_i}(t) = \begin{cases} (1 + \varepsilon)t & \text{if } \ t < t_i \\ \varepsilon t_i + t & \text{else} \end{cases}$$

unless v_i receives a message that causes it to set its clock to a larger value. If no node ever decides to set its clock to a larger value, the clock skew between v_i and v_{i-1} at time t_i is

$$L_{v_i}(t_i) - L_{v_{i-1}}(t_i) = \varepsilon(t_i - t_{i-1}) = (2i-1)\mathcal{T}.$$

Given that the hardware clock rate of both v_i's and v_{i-1}'s clock are 1 after t_i by definition, the clock skew remains $(2i-1)\mathcal{T}$. The clock skew between v_D and v_0 at time t_D is thus

$$L_{v_D}(t_D) - L_{v_0}(t_D) = \sum_{j=1}^{D}(2j-1)\mathcal{T} = D^2\mathcal{T}.$$

It remains to show that no node ever increases its clock value due to a received message. Assume for the sake of contradiction that node v_i is the first node that receives a message that causes it to increase its clock value instantaneously at a time $\bar{t}$. Let $L'_{v_i}(\bar{t})$ denote the clock value of v_i before the message is processed.[2] Note that $L^w_{v_i}(\bar{t}) \leq L_w(\bar{t}-\mathcal{T})$ for any $w \in \mathcal{N}_{v_i}$ because each messages is delayed by $\mathcal{T}$ and the last message may have arrived earlier than at time $\bar{t}$.

As v_D's clock value is the largest at time $\bar{t}$, we know that $i \neq D$. Moreover, v_i cannot be v_0: The minimum progress rate of v_1's clock is 1 and the execution only builds up a clock skew of $\mathcal{T}$ between the two nodes, which implies that $L^{v_1}_{v_0}(\bar{t}) \leq L_{v_1}(\bar{t}-\mathcal{T}) \leq L_{v_1}(\bar{t}) - \mathcal{T} \leq L'_{v_0}(\bar{t})$. Thus, v_0 cannot receive a larger clock value than its own from v_1 at time $\bar{t}$, and consequently v_0 does not increase its clock value at time $\bar{t}$ due to a received message. We conclude that v_i must be a node in the set $\{v_1, \ldots, v_{D-1}\}$.

Since all clock values increased at their own hardware clock rate until $\bar{t}$, we further know that

$$\begin{aligned}
\Lambda^{\uparrow}_{v_i}(\bar{t}) &= \max\{L^{v_{i+1}}_{v_i}(\bar{t}), L^{v_{i-1}}_{v_i}(\bar{t})\} - L'_{v_i}(\bar{t}) \\
&\leq \max\{L_{v_{i+1}}(\bar{t}-\mathcal{T}), L_{v_{i-1}}(\bar{t}-\mathcal{T})\} - L'_{v_i}(\bar{t}) \\
&= L_{v_{i+1}}(\bar{t}-\mathcal{T}) - L'_{v_i}(\bar{t})
\end{aligned}$$

and

$$\begin{aligned}
\Lambda^{\downarrow}_{v_i}(\bar{t}) &= L'_{v_i}(\bar{t}) - \min\{L^{v_{i+1}}_{v_i}(\bar{t}), L^{v_{i-1}}_{v_i}(\bar{t})\} \\
&\geq L'_{v_i}(\bar{t}) - \min\{L_{v_{i+1}}(\bar{t}-\mathcal{T}), L_{v_{i-1}}(\bar{t}-\mathcal{T})\} \\
&= L'_{v_i}(\bar{t}) - L_{v_{i-1}}(\bar{t}-\mathcal{T}),
\end{aligned}$$

which implies that

$$\Lambda^{\uparrow}_{v_i}(\bar{t}) - \Lambda^{\downarrow}_{v_i}(\bar{t}) \leq L_{v_{i+1}}(\bar{t}-\mathcal{T}) + L_{v_{i-1}}(\bar{t}-\mathcal{T}) - 2L'_{v_i}(\bar{t}). \qquad (7.1)$$

[2] This variable is needed because the clock value changes at time $\bar{t}$ by definition and $L_{v_i}(\bar{t})$ denotes the value *after* the increase as defined in Section 6.2.

If $\bar{t} < t_i$, the clock value of v_i at time $\bar{t}$ is $L'_{v_i}(\bar{t}) = (1+\varepsilon)\bar{t}$, and the estimates $L_{v_i}^{v_{i+1}}(\bar{t})$ and $L_{v_i}^{v_{i-1}}(\bar{t})$ are upper bounded by $L_{v_{i+1}}(\bar{t} - \mathcal{T}) \leq (1+\varepsilon)(\bar{t} - \mathcal{T})$ and $L_{v_{i-1}}(\bar{t} - \mathcal{T}) \leq (1+\varepsilon)(\bar{t} - \mathcal{T})$, respectively. Inequality (7.1) states that in this case $\Lambda^{\uparrow}_{v_i}(\bar{t}) - \Lambda^{\downarrow}_{v_i}(\bar{t}) \leq 0$.

If $t_i \leq \bar{t} < t_{i+1} + \mathcal{T}$, we have that $L'_{v_i}(\bar{t}) = \varepsilon t_i + \bar{t}$. The estimate $L_{v_i}^{v_{i+1}}(\bar{t})$ is certainly upper bounded by $(1+\varepsilon)(\bar{t} - \mathcal{T})$ and the estimate $L_{v_i}^{v_{i-1}}(\bar{t} - \mathcal{T})$ is at most $\varepsilon t_{i-1} + \bar{t} - \mathcal{T}$ because $\bar{t} - \mathcal{T} \geq t_i - \mathcal{T} > t_{i-1}$. Hence, according to Inequality (7.1) we have that

$$\begin{aligned}\Lambda^{\uparrow}_{v_i}(\bar{t}) - \Lambda^{\downarrow}_{v_i}(\bar{t}) &\overset{(7.1)}{\leq} (1+\varepsilon)(\bar{t} - \mathcal{T}) + \varepsilon t_{i-1} + \bar{t} - \mathcal{T} - 2(\varepsilon t_i + \bar{t}) \\ &< \varepsilon(t_{i+1} - 2t_i + t_{i-1}) - 2\mathcal{T} = 0.\end{aligned}$$

The last case is that $\bar{t} \geq t_{i+1} + \mathcal{T}$. Since the clock rate of all three nodes is 1 at any time $t \geq \bar{t} - \mathcal{T}$, it holds that

$$\begin{aligned}\Lambda^{\uparrow}_{v_i}(\bar{t}) - \Lambda^{\downarrow}_{v_i}(\bar{t}) &\overset{(7.1)}{\leq} \varepsilon t_{i+1} + \bar{t} - \mathcal{T} + \varepsilon t_{i-1} + \bar{t} - \mathcal{T} - 2(\varepsilon t_i + \bar{t}) \\ &= \varepsilon(t_{i+1} - 2t_i + t_{i-1}) - 2\mathcal{T} = 0.\end{aligned}$$

We conclude that $\Lambda^{\uparrow}_{v_i}(\bar{t}) \leq \Lambda^{\downarrow}_{v_i}(\bar{t})$ in all cases, which implies that v_i does not set its clock to a larger value at time $\bar{t}$, a contradiction to the definition of $\bar{t}$. □

When algorithm $\mathcal{A}^{avg}$ is used on the graph G^{list}, it is thus possible that the *average* clock skew between neighboring nodes becomes $\Omega(D\mathcal{T})$. Moreover, the nodes never increase their clock values due to received messages at any time in the execution described in the proof, which implies that the large clock skews remain in the network. The problem of $\mathcal{A}^{avg}$ appears to be that the nodes do not increase their clock values quickly enough if one neighbor lags behind. However, it is not possible to simply "ignore" this particular neighbor, as otherwise the algorithm behaves like $\mathcal{A}^{max}$ when executing the algorithm on the example graph G^{list} because each node $v \in V^{list}$ has at most two neighbors. As we discussed before, $\mathcal{A}^{max}$ has the undesirable property that the skew between neighboring nodes may become $\Omega(D\mathcal{T})$ on any graph G. Thus, the solution must be to use the information about the neighbors' clock values differently.

7.2 Blocking Algorithm

A different approach to bounding the local skew is to set the logical clock to the maximum clock value, just like $\mathcal{A}^{max}$, subject to the condition that the (estimated) clock value of no neighbor is more than a specific value $\kappa > 0$ behind [36]. In other words, any node actively sets its clock to a value that is at most κ larger than the smallest estimate of any of its neighbors' clock values. This rule implies that a node never increases its clock value as long as $\Lambda_v^{\downarrow} \geq \kappa$, i.e., any further increase is "blocked" until a received message indicates that the node that is lagging behind the most has caught up. Intuitively, the parameter κ must be larger than $\mathcal{T}$ for the following reason. If all messages are delayed by $\mathcal{T}$, a node v may erroneously assume that a neighbor's clock is $\mathcal{T}$ behind even if the skew between its own and this neighbor's clock is in fact zero. By setting κ to a value greater than $\mathcal{T}$, we ensure that v still increases its clock value instantaneously upon receiving a larger clock value from another neighbor in this situation. The algorithm $\mathcal{A}^{block}$, which is based on this principle, is now discussed in detail.

7.2.1 Algorithm $\mathcal{A}^{block}$

As in the preceding section, the clock values increase at the hardware clock rate in the absence of new information about the neighbors' clock values. Apart from the clock value L_v, each message further contains v's estimate L_v^{max} of the largest clock value in the network, which is also initialized to 0 at time $t = 0$. Instead of updating the estimates L_v^w for all $w \in \mathcal{N}_v$ only if a larger clock value L^w is received from w, all estimates L_v^w are increased at the hardware clock rate of v as well, which means that all nodes assume that the clock values of their neighboring nodes increase at their own hardware clock rate. The same rule applies to the estimate L_v^{max} of the largest clock value in the network. This approach has a considerable advantage: The logical clock value of w potentially increases by $(1+\varepsilon)H_0$ until it sends the next message to v even if w never sets its clock to a larger value in this time interval. If v only updates the estimate L_v^w when a new message is received from w, the error in the estimate increases by $(1+\varepsilon)H_0$ as well. However, the error can increase by at most $2\varepsilon H_0$ if L_v^w is increased at v's hardware clock rate because the progress of v's hardware clock in the same interval is at least $(1-\varepsilon)H_0$. This simple trick renders it possible to set H_0 to a large value, resulting in a low message frequency without increasing the clock skew bounds significantly.

Upon receiving a tuple $\langle L^w, L_w^{max} \rangle$ from a neighbor w,[3] node v first updates its variable L_v^{max} by setting it to L_w^{max} if this value exceeds the old value. The algorithm strives to disseminate new information about the largest clock value in the network as quickly as possible. For this purpose, the flag *send*, which indicates whether a message has to be sent to all neighbors after the received message has been processed, is set to *true* if the received estimate of the largest clock value is larger than the old estimate. Subsequently, L_v^w is set to the received clock value L^w if this value is greater than the largest clock value ℓ_v^w that v has ever received from w before. Since messages that are sent later may arrive earlier due to different message delays,

[3] Technically, a node v might receive several messages at the same time. These messages can be processed in an arbitrary order.

Algorithm 7.2 $\mathcal{A}^{block}$: Compute the new clock value L_v after receiving $\langle L^w, L_w^{max}\rangle$ from node $w \in \mathcal{N}_v$.

```
if $L_w^{max} > L_v^{max}$ then
    $send := true$
    $L_v^{max} := L_w^{max}$
end if
if $L^w > \ell_v^w$ then
    $L_v^w := L^w$; $\ell_v^w := L^w$
end if
$\Lambda_v^{\downarrow} := \max_{u \in \mathcal{N}_v}\{L_v - L_v^u\}$
$R_v := \min\{L_v^{max} - L_v, \kappa - \Lambda_v^{\downarrow}\}$
if $R_v > 0$ then
    $L_v := L_v + R_v$
    $\Lambda_v^{\downarrow} := \Lambda_v^{\downarrow} + R_v$
end if
if $R_v > 0$ or $send = true$ then
    $send := false$
    send $\langle L_v, L_v^{max}\rangle$ to all $u \in \mathcal{N}_v$
end if
```

this test is necessary in order to ensure that L_v^w is based on the most current information about w's clock value. Afterwards, the estimated clock skew $\Lambda_v^{\downarrow}$ to the node that is behind the most is computed. The clock value L_v is then increased by at most $L_v^{max} - L_v$, i.e., nodes never set their clocks to a value larger than the estimated maximum clock value. Additionally, a node is merely allowed to increase its clock value by at most $\kappa - \Lambda_v^{\downarrow}$, which ensures that the clock value is increased to a value that is at most κ larger than the estimated clock value of any neighbor. If v can increase its clock by a positive value, L_v is set to the new value, and $\Lambda_v^{\downarrow}$ is adjusted. Finally, a message is sent to all neighbors if either v sets its clock to a larger value or the estimate of the maximum clock value has increased. The steps of algorithm $\mathcal{A}^{block}$ are summarized in Algorithm 7.2.

In the following, we assume that messages are sent automatically whenever the estimate L_v^{max} (instead of the hardware clock value H_v) reaches the next integer multiple of H_0. The advantage of this strategy is that it helps to restrict the number of messages that are exchanged: If a message is sent when H_v reaches the next multiple of H_0, any node v may receive messages from its neighbors in such an order that each message contains a larger estimate of the maximum clock value in the network, which causes v to send messages to all neighbors for each received message although the differences between these estimates may be arbitrarily small. However, all nodes only send messages to their neighbors once for each integer multiple of H_0 reached by L_v^{max} if L_v^{max} is used for this purpose.

It is worth noting that the rule that R_v can be at most $\kappa - \Lambda_v^{\downarrow}$ does not guarantee that a node never increases its clock value instantaneously if a neighbor's clock is κ behind; however, since any node receives a message from each neighbor after at most $\mathcal{T} + H_0/(1-\varepsilon)$ time has passed since the last message arrived at a time when $\Lambda_v^{\downarrow} < \kappa$ and any instantaneous increase $R_v > 0$ also causes $\Lambda_v^{\downarrow}$ to increase by R_v, the difference between the actual clock skew between v and the neighbor w that is behind the most and $\Lambda_v^{\downarrow}$ can only grow due to a faster hardware clock. Given that the minimum progress rate is $1-\varepsilon$, the clock skew can increase by at most $2\varepsilon(\mathcal{T} + H_0/(1-\varepsilon))$. Thus, any node v can increase its clock value instantaneously only to a value that is less than

$$\kappa + 2\varepsilon(\mathcal{T} + H_0/(1-\varepsilon))$$

larger than the clock value of any of its neighbors.

As one might expect, the parameters κ and H_0 have an impact on the achievable bounds on both the global and the local skew. The dependency between these parameters and the bounds on the clock skew are examined next.

7.2.2 Analysis of $\mathcal{A}^{block}$

A node may increase its clock value quickly as long as it is not blocked. Let $\mathcal{R}_v(t_1, t_2)$ denote the amount by which the logical clock increase exceeds the increase of the hardware clock:

Definition 7.2. $\forall t_1 \leq t_2 : \mathcal{R}_v(t_1,t_2) := (L_v(t_2) - L_v(t_1)) - (H_v(t_2) - H_v(t_1))$.

Since nodes no longer react to incoming messages as long as they are blocked, one might fear that the global skew may again become large. Fortunately, this is not the case if κ is sufficiently large. In fact, the following upper bound does not only hold for algorithm $\mathcal{A}^{block}$, but for any algorithm that belongs to the family $\mathfrak{A}$ of clock synchronization algorithms with the following properties.

1. The logical clock value of each node v always increases at a rate of at least $1-\varepsilon$.

2. Any node v increases the estimates of L_v^w and L_v^{max} at its own hardware clock rate and updates them when new information is received.

3. No node v sets its clock to a value greater than L_v^{max}.

4. Any node v immediately forwards the estimate L_w^{max} to all (other) neighbors when it receives an estimate $L_w^{max} > L_v^{max}$ from a node w.

5. Any node v sends $\langle L_v, L_v^{max} \rangle$ to all neighbors if L_v^{max} reaches the next integer multiple of H_0, for an arbitrary parameter $H_0 > 0$.

6. Any node v increases its logical clock at least at the rate $1+\varepsilon$ if $L_v < L_v^{max}$ and $\Lambda_v^{\downarrow} < \kappa_0 := (1+\varepsilon)\mathcal{T} + 2\varepsilon H_0/(1+\varepsilon)$.

It is easy to see that Algorithm $\mathcal{A}^{block}$ belongs to the family $\mathfrak{A}$ when choosing any $\kappa \geq \kappa_0 = (1+\varepsilon)\mathcal{T} + 2\varepsilon H_0/(1+\varepsilon)$: Obviously, algorithm $\mathcal{A}^{block}$ has the first five properties. Line 9 of Algorithm 7.2 states that L_v is set either to L_v^{max} or to a value such that $\Lambda_v^{\downarrow} \geq \kappa \geq \kappa_0$, i.e., $\mathcal{A}^{block}$ also has the last property. Note that we may consider $\mathcal{A}^{max}$ to be a member of the family $\mathfrak{A}$ as well even if it does not store any estimates L_v^w (or computes $\Lambda_v^{\downarrow}$), because $L_v = L_v^{max}$ is increased at the hardware clock rate, and both L_v and L_v^{max} are set to L_w^{max} when an estimate L_w^{max} greater than L_v^{max} is received.

Since the minimum progress rate of each node is $1-\varepsilon$, any algorithm $\mathcal{A} \in \mathfrak{A}$ satisfies the weak version of Condition (6.1). We define the *virtual clock*

$$L^{max}(t) := \max_{v \in V}\{L_v^{max}(t)\},$$

which increases at a rate of at most $1+\varepsilon$ when an algorithm $\mathcal{A} \in \mathfrak{A}$ is used because the estimates L_v^{max} are increased at the hardware clock rates. Moreover, it holds that $L_v(t) \leq L_v^{max}(t)$ at all times t because no node sets its clock to a value larger than L_v^{max} and L_v increases at the hardware clock rate whenever $L_v = L_v^{max}$. These observations and the fact that the minimum logical clock rate is $1-\varepsilon$ imply that any $\mathcal{A} \in \mathfrak{A}$, and in particular $\mathcal{A}^{block}$, satisfies Condition (6.2). The following theorem bounds the global skew of any such algorithm.

Theorem 7.3. *The global skew when executing any algorithm $\mathcal{A} \in \mathfrak{A}$ on any graph G of diameter D is upper bounded by*

$$\mathcal{G} := (1+\varepsilon)D\mathcal{T} + \frac{2\varepsilon}{1+\varepsilon}H_0.$$

Proof. Instead of bounding the clock skew between the nodes directly, we show that $L^{max}(t) - L_v(t) \leq \mathcal{G}$ for all nodes $v \in V$ and times t. As $L^{max}(t) \geq L_v(t)$ for all nodes $v \in V$ at all times t, this statement proves the theorem.

For the sake of contradiction, assume that $\bar{t}$ is the infimum of all times when the clock skew between L^{max} and the clock value L_v of some node v exceeds $\mathcal{G}$. Since L^{max} increases continuously and since all nodes never reduce their clock values, it holds that

$$L^{max}(\bar{t}) - L_v(\bar{t}) = \mathcal{G}. \tag{7.2}$$

First, assume that $\Lambda_v^{\downarrow}(\bar{t}) < \kappa_0$. Let L be the largest estimate of the maximum clock value that v receives at the latest at time $\bar{t}$ and that is an integer multiple of H_0, i.e., it has originally been sent due to the fifth property of the family $\mathfrak{A}$.[4] Moreover, let t_s and $t_r \leq \bar{t}$ be the times when L is first sent

[4] It is possible that v receives a larger estimate at a time $t \leq \bar{t}$ if the considered algorithm $\mathcal{A}$ also sends messages because of other events.

and when it is received by v, respectively.[5] Since t_s is the first time when L was sent, we have that $L = L^{max}(t_s)$. Assume that $t_s \geq \bar{t} - D\mathcal{T}$. As L_v^{max} increases at least at the rate $1-\varepsilon$, we get that

$$\begin{aligned} L_v^{max}(\bar{t}) &\geq L + (1-\varepsilon)(\bar{t} - t_r) \\ &= L^{max}(t_s) + (1-\varepsilon)(\bar{t} - t_r) \\ &\geq L^{max}(\bar{t}) - (1+\varepsilon)(\bar{t} - t_s) + (1-\varepsilon)(\bar{t} - t_r) \qquad (7.3) \\ &\overset{(7.2)}{\geq} L_v(\bar{t}) + \mathcal{G} - (1+\varepsilon)D\mathcal{T} \\ &> L_v(\bar{t}). \end{aligned}$$

Inequality (7.3) uses that the progress rate of L^{max} is upper bounded by $1+\varepsilon$. The message may have been sent earlier at a time $t_s = \bar{t} - D\mathcal{T} - \gamma$ for a value $\gamma \in (0, H_0/(1-\varepsilon))$. Note that $t_s \leq \bar{t} - D\mathcal{T} - H_0/(1-\varepsilon)$ is not possible as the progress of the hardware clock until $\bar{t} - D\mathcal{T}$ is at least H_0 because the hardware clock rates are always at least $1-\varepsilon$. In this case, the fifth property states that a message must be sent at the latest at time $\bar{t} - D\mathcal{T}$, which, according to the fourth property, is forwarded towards v unless some node on the path has already sent a larger estimate of the maximum clock value earlier. Since the distance to v is upper bounded by D and the message delays are at most $\mathcal{T}$, node v would receive this message at the latest at time $\bar{t}$, contradicting the assumption that L is the largest estimate that arrives until $\bar{t}$. More generally, if h^{max} denotes the average clock rate of L^{max} in the interval $[t_s, \bar{t} - D\mathcal{T}]$, it holds that $\gamma h^{max} < H_0$, otherwise L is not the largest estimate of the maximum clock value that arrives at v at the latest at time $\bar{t}$. Since $t_r \leq t_s + D\mathcal{T}$ and thus $\bar{t} - t_r \geq \gamma$, it holds in this case that

$$\begin{aligned} L_v^{max}(\bar{t}) &\geq L + (1-\varepsilon)(\bar{t} - t_r) \\ &\geq L^{max}(t_s) + (1-\varepsilon)\gamma \\ &\geq L^{max}(\bar{t}) - (1+\varepsilon)D\mathcal{T} - h^{max}\gamma + (1-\varepsilon)\gamma \\ &\overset{(7.2)}{=} \mathcal{G} + L_v(\bar{t}) - (1+\varepsilon)D\mathcal{T} - \gamma(h^{max} - (1-\varepsilon)) \\ &> \mathcal{G} + L_v(\bar{t}) - (1+\varepsilon)D\mathcal{T} - \frac{2\varepsilon}{1+\varepsilon}H_0 \qquad (7.4) \\ &= L_v(\bar{t}). \end{aligned}$$

Inequality (7.4) exploits that $\gamma(h^{max} - (1-\varepsilon)) = \gamma h^{max}\left(1 - \frac{1-\varepsilon}{h^{max}}\right) < \frac{2\varepsilon}{1+\varepsilon}H_0$ for all $\gamma \in (0, H_0/(1-\varepsilon))$ as $\gamma h^{max} < H_0$ and $h^{max} \leq 1+\varepsilon$. Thus, in both cases it holds that $L_v(\bar{t}) < L_v^{max}(\bar{t})$. The sixth property states that any logical clock increases at least at the rate $1+\varepsilon$ if $\Lambda_v^{\downarrow} < \kappa_0$ and $L_v < L_v^{max}$. Since L^{max} increases at most at the clock rate $1+\varepsilon$, $\bar{t}$ cannot be

[5] Note that t_s and t_r are not (necessarily) the send and receive event of the same message. The estimate L of the maximum clock value in the network may be forwarded to v along a path of nodes.

the infimum of all times when the clock skew between L^{max} and L_v exceeds $\mathcal{G}$, a contradiction to the definition of $\bar{t}$.

It remains to study the case that $\Lambda_v^{\downarrow}(\bar{t}) \geq \kappa_0$. Let $w \in \mathcal{N}_v$ be any node such that $L_v(\bar{t}) - L_v^w(\bar{t}) \geq \kappa_0$, and let t_s denote the time when w sent the message containing the largest clock value that v received at a time $t_r \leq \bar{t}$. It holds that $L_v(t_r) - L_v^w(t_r) + \mathcal{R}_v(t_r, \bar{t}) = L_v(\bar{t}) - L_v^w(\bar{t}) \geq \kappa_0$ because L_v increases by exactly $\mathcal{R}_v(t_r, \bar{t})$ more than the hardware clock and thus also $\mathcal{R}_v(t_r, \bar{t})$ more than the estimate L_v^w. Hence, since the message received at time t_r contains the clock value $L_w(t_s)$, which implies that $L_v^w(t_r) = L_w(t_s)$, we know that

$$L_w(t_s) \leq L_v(t_r) + \mathcal{R}_v(t_r, \bar{t}) - \kappa_0. \tag{7.5}$$

Furthermore, it holds that

$$\begin{aligned} L_v(\bar{t}) &= L_v(t_r) + (H_v(\bar{t}) - H_v(t_r)) + \mathcal{R}_v(t_r, \bar{t}) \\ &\geq L_v(t_r) + (1-\varepsilon)(\bar{t} - t_r) + \mathcal{R}_v(t_r, \bar{t}) \qquad (7.6) \end{aligned}$$

If we assume that $t_s \geq \bar{t} - \mathcal{T}$, the skew between L^{max} and L_w at time t_s is at least

$$\begin{aligned} L^{max}(t_s) - L_w(t_s) &\overset{(7.5)}{\geq} L^{max}(\bar{t}) - (1+\varepsilon)(\bar{t} - t_s) - L_v(t_r) \\ &\quad -\mathcal{R}_v(t_r, \bar{t}) + \kappa_0 \\ &\overset{(7.6)}{\geq} L^{max}(\bar{t}) - L_v(\bar{t}) + \kappa_0 \\ &\quad -(1+\varepsilon)(\bar{t} - t_s) + (1-\varepsilon)(\bar{t} - t_r) \\ &= L^{max}(\bar{t}) - L_v(\bar{t}) + \kappa_0 \\ &\quad -(1-\varepsilon)(t_r - t_s) - 2\varepsilon(\bar{t} - t_s) \\ &\overset{(7.2)}{\geq} \mathcal{G} + \kappa_0 - (1+\varepsilon)\mathcal{T} > \mathcal{G}. \end{aligned}$$

If $t_s = \bar{t} - \mathcal{T} - \gamma$ for a value $\gamma \in (0, H_0/(1-\varepsilon))$ as before, we again have that $h_w\gamma < H_0$, where h_w now denotes the average clock rate of w in the time interval $[t_s, \bar{t} - \mathcal{T}]$. The same arguments as in the previous cases reveal that

$$\begin{aligned} L^{max}(\bar{t} - \mathcal{T}) - L_w(\bar{t} - \mathcal{T}) &\geq L^{max}(\bar{t}) - (1+\varepsilon)\mathcal{T} - (L_w(t_s) + h_w\gamma) \\ &\overset{(7.5)}{\geq} L^{max}(\bar{t}) - L_v(t_r) - \mathcal{R}_v(t_r, \bar{t}) \\ &\quad +\kappa_0 - (1+\varepsilon)\mathcal{T} - h_w\gamma \\ &\overset{(7.6)}{\geq} L^{max}(\bar{t}) - L_v(\bar{t}) + (\bar{t} - t_r)(1-\varepsilon) \\ &\quad +\kappa_0 - (1+\varepsilon)\mathcal{T} - h_w\gamma \\ &\overset{\bar{t}-t_r \geq \gamma}{\geq} \mathcal{G} + \kappa_0 - (1+\varepsilon)\mathcal{T} - \gamma(h_w - (1-\varepsilon)) \\ &\overset{\gamma h_w < H_0}{>} \mathcal{G} + \kappa_0 - (1+\varepsilon)\mathcal{T} - \frac{2\varepsilon}{1+\varepsilon} H_0 \geq \mathcal{G}. \end{aligned}$$

Thus, in either case the skew between L^{max} and L_w exceeded $\mathcal{G}$ at a time not later than $\bar{t}$. This is a contradiction as $L^{max}(t) - L_v(t) \leq \mathcal{G}$ for all $v \in V$ at all times $t < \bar{t}$ because $\bar{t}$ is the infimum of all times when this bound is violated, and $L^{max}(\bar{t}) - L_v(\bar{t}) = \mathcal{G}$ due to the fact that L^{max} is a continuous function. □

This result implies that the same upper bound $\mathcal{G}$ on the global skew applies to both $\mathcal{A}^{block}$ and $\mathcal{A}^{max}$. The following theorem shows that $\mathcal{A}^{block}$ guarantees a better bound on the local skew than $\mathcal{A}^{max}$.

Theorem 7.4. *If $\kappa \geq 2(1+\varepsilon)\left(\mathcal{T} + \frac{H_0}{1-\varepsilon}\right)$, the local skew when executing algorithm $\mathcal{A}^{block}$ on any graph G of diameter D is upper bounded by*

$$\kappa + \frac{8\varepsilon}{\kappa}\mathcal{G}\left(\mathcal{T} + \frac{H_0}{1-\varepsilon}\right).$$

Proof. Define that $\mathcal{T}' := \mathcal{T} + \frac{H_0}{1-\varepsilon}$. If a node receives a message from a neighbor at time t, the next message from the same neighbor will arrive at the latest at time $t + \mathcal{T}'$ because each node sends a message after at most H_0 time measured using its hardware clock and the hardware clock rate of each clock is lower bounded by $1-\varepsilon$. Assume for the sake of contradiction that there is a time $\bar{t}$ and two nodes v_0 and v_1 such that

$$L_{v_0}(\bar{t}) - L_{v_1}(\bar{t}) \geq \kappa + \frac{8\varepsilon}{\kappa}\mathcal{G}\mathcal{T}'. \tag{7.7}$$

Define that $t_i := \bar{t} - (i-1)\mathcal{T}'$ for all $i \in \mathbb{N}$. Furthermore, let i^* be the largest integer such that $i^* \leq \frac{2}{\kappa}\mathcal{G}$. We can assume that $i^* \geq 2$, as otherwise it holds that $\mathcal{G} < \kappa$ and the theorem follows immediately from Theorem 7.3. As argued before, a node can increase its clock value instantaneously at most to a value that is $\kappa + 2\varepsilon(\mathcal{T} + H_0/(1-\varepsilon)) = \kappa + 2\varepsilon\mathcal{T}'$ larger than the clock value of any neighbor. Since the clock skew at time $\bar{t}$ is at least $\kappa + \frac{8\varepsilon}{\kappa}\mathcal{G}\mathcal{T}'$, node v_0 must have built up the additional skew of $\frac{8\varepsilon}{\kappa}\mathcal{G}\mathcal{T}' - 2\varepsilon\mathcal{T}'$ by means of a faster hardware clock rate, i.e., v_0 is blocked at least since time $\bar{t} - (\frac{8\varepsilon}{\kappa}\mathcal{G}\mathcal{T}' - 2\varepsilon\mathcal{T}')/(2\varepsilon) = \bar{t} - (\frac{4}{\kappa}\mathcal{G} - 1)\mathcal{T}' \leq \bar{t} - (2i^* - 1)\mathcal{T}' = t_{2i^*}$, which implies that

$$L_{v_0}(t) \geq L_{v_0}(\bar{t}) - (1+\varepsilon)(\bar{t} - t) \tag{7.8}$$

for any $t \in [t_{2i^*}, \bar{t}]$.

We claim that there is a series of nodes $v_1, \ldots, v_{i^*}$ such that for each v_i, $i \in \{1, \ldots, i^*\}$, it holds that

$$L_{v_0}(\bar{t}) - L_{v_i}(t_i) > i\kappa \tag{7.9}$$

and v_i is blocked at time t_i, i.e., $\Lambda^{\downarrow}_{v_i}(t_i) \geq \kappa$. Furthermore, the distance between v_0 and v_i is upper bounded by i.

The clock skew at time $\bar{t}$ between the nodes v_0 and v_1, for which $d(v_0, v_1) = 1$, is greater than κ according to Inequality (7.7). Moreover, node v_1 received a message at the latest at time $\bar{t}$ from v_0 containing a clock value greater than $L_{v_0}(\bar{t} - \mathcal{T}') \geq L_{v_0}(\bar{t}) - (1+\varepsilon)\mathcal{T}'$, which implies that

$$L_{v_1}^{max}(\bar{t}) > L_{v_0}(\bar{t}) - (1+\varepsilon)\mathcal{T}' \overset{(7.7)}{>} L_{v_1}(\bar{t}) + \kappa - (1+\varepsilon)\mathcal{T}' > L_{v_1}(\bar{t}).$$

Thus, it must hold that $\Lambda_{v_1}^{\downarrow}(t_1) = \Lambda_{v_1}^{\downarrow}(\bar{t}) \geq \kappa$, which proves the claim for $i = 1$.

Assume that the claim holds for node v_i, $i < i^*$, i.e., $\Lambda_{v_i}^{\downarrow}(t_i) \geq \kappa$. Let $v_{i+1} \in \mathcal{N}_{v_i}$ be the node that maximizes $\Lambda_{v_i}^{\downarrow}(t_i)$. We have that $L_{v_{i+1}}(t_{i+1}) \leq L_{v_i}(t_i) - \kappa$, as otherwise v_i receives a message at the latest at time t_i containing a clock value larger than $L_{v_i}(t_i) - \kappa$. Since v_{i+1} maximizes $\Lambda_{v_i}^{\downarrow}(t_i)$, v_i would not be blocked at time t_i, a contradiction. Consequently, it holds that

$$L_{v_0}(\bar{t}) - L_{v_{i+1}}(t_{i+1}) \geq L_{v_0}(\bar{t}) - (L_{v_i}(t_i) - \kappa) \overset{(7.9)}{>} (i+1)\kappa. \tag{7.10}$$

As $d(v_0, v_i) \leq i$ and $v_{i+1} \in \mathcal{N}_{v_i}$, it follows that $d(v_0, v_{i+1}) \leq i+1$. It remains to verify that v_{i+1} is blocked, i.e., that $L_{v_{i+1}}^{max}(t_{i+1}) > L_{v_{i+1}}(t_{i+1})$. Given that $d(v_0, v_{i+1}) \leq i+1$, it takes at most $(i+1)\mathcal{T}$ time for a clock value from v_0 to be forwarded to v_{i+1}, which implies that $L_{v_{i+1}}^{max}(t_{i+1}) \geq L_{v_0}(t_{2(i+1)})$. Since $i+1 \leq i^*$ and thus $t_{2(i+1)} \geq t_{2i^*}$, we have that

$$\begin{array}{rcl}
L_{v_{i+1}}^{max}(t_{i+1}) & \geq & L_{v_0}(t_{2(i+1)}) \\
& = & L_{v_0}(\bar{t} - (2i+1)\mathcal{T}') \\
& \overset{(7.8)}{\geq} & L_{v_0}(\bar{t}) - (1+\varepsilon)(2i+1)\mathcal{T}' \\
& \overset{(7.10)}{>} & L_{v_{i+1}}(t_{i+1}) + (i+1)\kappa - (1+\varepsilon)(2i+1)\mathcal{T}' \\
& > & L_{v_{i+1}}(t_{i+1}) + (2i+1)\left(\frac{\kappa}{2} - (1+\varepsilon)\mathcal{T}'\right) \\
& \geq & L_{v_{i+1}}(t_{i+1}),
\end{array}$$

which proves the claim.

Since v_{i^*} is blocked at time t_{i^*}, there is a node $v_{i^*+1} \in \mathcal{N}_{v_{i^*}}$ for which it holds that $L_{v_{i^*+1}}(t_{i^*+1}) \leq L_{v_{i^*}}(t_{i^*}) - \kappa$, implying that

$$L_{v_0}(\bar{t}) - L_{v_{i^*+1}}(t_{i^*+1}) \overset{(7.9)}{>} (i^*+1)\kappa.$$

The clock value of v_0 at time $t_{i^*+1} \geq t_{2i^*}$ is at least

$$L_{v_0}(t_{i^*+1}) = L_{v_0}(\bar{t} - i^*\mathcal{T}') \overset{(7.8)}{\geq} L_{v_0}(\bar{t}) - (1+\varepsilon)i^*\mathcal{T}'.$$

These two inequalities show that the clock skew between v_0 and v_{i^*+1} at time t_{i^*+1} is lower bounded by

$$\begin{aligned} L_{v_0}(t_{i^*+1}) - L_{v_{i^*+1}}(t_{i^*+1}) &> (i^*+1)\kappa - (1+\varepsilon)i^*\mathcal{T}' \\ &> (i^*+1)(\kappa - (1+\varepsilon)\mathcal{T}') \\ &\geq (i^*+1)\frac{\kappa}{2} \\ &> \frac{2\mathcal{G}}{\kappa}\frac{\kappa}{2} = \mathcal{G}. \end{aligned}$$

In the last inequality we used that i^* is the largest integer that is at most $2\mathcal{G}/\kappa$, which implies that $i^* > 2\mathcal{G}/\kappa - 1$. Thus, the assumption that there are neighboring nodes with a clock skew of at least $\kappa + \frac{8\varepsilon}{\kappa}\mathcal{G}\mathcal{T}'$ implies that the skew between two nodes must have been larger than $\mathcal{G}$ at an earlier time, a contradiction to Theorem 7.3. □

Since ε and $\mathcal{T}$ are unknown, κ must be set to at least $2(1+\hat{\varepsilon})(\hat{\mathcal{T}}+H_0/(1-\hat{\varepsilon}))$. According to Theorem 7.4 and the assumptions that $\hat{\varepsilon} \in \mathcal{O}(\varepsilon)$ and $\hat{\mathcal{T}} \in \mathcal{O}(\mathcal{T})$, the local skew is upper bounded by $\mathcal{O}(\mathcal{T}(1+\varepsilon D) + H_0/(1-\varepsilon))$ when setting κ to this minimum value. Note that this result is an improvement over the bound of algorithm $\mathcal{A}^{max}$ as the local skew tends to $\mathcal{O}(\mathcal{T})$ if ε and H_0 tend to zero. Recall that the local skew of $\mathcal{A}^{max}$ is $\Omega(D\mathcal{T})$ independent of both ε and H_0.

7.2.3 Discussion

It is not hard to see that the analysis in the preceding section is asymptotically tight if κ is chosen as small as possible and H_0 tends to zero, i.e., there are executions that induce a local skew of $\Omega(\mathcal{T}(1+\varepsilon D))$ when algorithm $\mathcal{A}^{block}$ is used and $\kappa \in \Theta(\mathcal{T})$. As we will see in Chapter 9, a global skew of $\Omega(D\mathcal{T})$ cannot be prevented. Once a global skew of $\Omega(D\mathcal{T})$ has been introduced into the network, the message delays are reduced to zero except for the delays of the message sent to the node v_0 with the currently smallest clock value. The reduced message delays entail that the neighbors of v_0 acquire a clock value that is κ larger than v_0's clock value. Their neighbors in turn have clock values that are κ larger etc., resulting in paths $v_0, \ldots, v_k$ in which each node v_i has a clock value that is approximately κ larger than the clock value of v_{i-1} for all $i \in \{1, \ldots, k\}$. The length of such a path is bounded by $\Omega(D\mathcal{T}/\kappa) = \Omega(D)$. If the message delays are increased to $\mathcal{T}$ again at this point, node v_{k-1} only increases its clock value at its hardware clock rate until it receives a larger clock value from node v_{k-2}, which has to wait for a larger clock value from v_{k-3} etc. Since the message delays are $\mathcal{T}$ and $k \in \Omega(D)$, it takes $\Omega(D\mathcal{T})$ time until an update propagates from v_0 to v_{k-1}. The clock skew between v_k and v_{k-1} increases to $\Omega(\mathcal{T} + \varepsilon D\mathcal{T})$ if during this time the hardware clock rate of v_k is $1+\varepsilon$ and the hardware clock rate of all other nodes is $1-\varepsilon$.

In practical distributed systems, the diameter D of the network is much smaller than $1/\varepsilon$. For example, if $\varepsilon = 10^{-5}$, the clocks gain or lose roughly one second per day, which is roughly the accuracy of a regular quartz clock. Clearly, there is no practical network with a diameter of 10^5. If D is smaller than $1/\varepsilon$, the local skew is in fact bounded by $\mathcal{O}(\mathcal{T})$ if $\mathcal{A}^{block}$ is used to synchronize the clocks, $\kappa \in \Theta(\mathcal{T})$, and $H_0/(1-\hat{\varepsilon}) \in \mathcal{O}(\mathcal{T})$. However, if the diameter and the clock drifts are sufficiently large, the skew between neighboring nodes is dominated by the term $\frac{8\varepsilon}{\kappa}\mathcal{G}(T + H_0/(1-\varepsilon))$, i.e., the local skew still grows linearly with the network diameter D.

There is a simple trick to avoid this linear dependency: By setting

$$\kappa := \max\left\{2(1+\hat{\varepsilon})\left(\hat{\mathcal{T}} + \frac{H_0}{1-\hat{\varepsilon}}\right), \sqrt{8\hat{\varepsilon}\hat{\mathcal{G}}\left(\hat{\mathcal{T}} + \frac{H_0}{1-\hat{\varepsilon}}\right)}\right\},$$

where $\hat{\mathcal{G}} \geq \mathcal{G}$ is the estimated global skew based on $\hat{\varepsilon} \in \mathcal{O}(\varepsilon)$ and $\hat{\mathcal{T}} \in \mathcal{O}(\mathcal{T})$, the local skew reduces to

$$\mathcal{O}\left(\mathcal{T} + \frac{H_0}{1-\varepsilon} + \sqrt{\varepsilon D\left(\mathcal{T} + \frac{H_0}{1-\varepsilon}\right)}\right).$$

Note that the nodes need to know the diameter D in order to compute $\hat{\mathcal{G}}$. If the term $H_0/(1-\varepsilon)$ is upper bounded by $\mathcal{O}(\mathcal{T})$, this choice of κ ensures that the local skew is bounded by $\mathcal{O}(\mathcal{T} + \sqrt{\varepsilon D \mathcal{T}})$, i.e., the local skew merely grows with the square root of the diameter D. In order to further improve the asymptotic behavior, a more sophisticated technique is needed, which is the subject of the following chapter.

Chapter 8

Optimal Clock Synchronization

BOTH algorithm $\mathcal{A}^{avg}$ and $\mathcal{A}^{block}$ essentially incur large clock skews between neighboring nodes because they increase the clock values too slowly. The problem of $\mathcal{A}^{avg}$ is that all clocks permanently run at their hardware clock rate as long as each node v assumes that there is a neighbor whose clock is behind more than any other neighbor's clock is ahead, i.e., $\Lambda_v^{\downarrow} \geq \Lambda_v^{\uparrow}$. While algorithm $\mathcal{A}^{block}$ increases the clock values as much as possible as long as no neighbor is (assumed to be) κ or more behind, the clock value only increases at the hardware clock rate once this threshold is reached.

The main challenge is to find an algorithm that is more aggressive, but does not increase the clock values too quickly. Recall that algorithms that increase the clock values too quickly, such as $\mathcal{A}^{max}$, also incur a large local skew. The basic strategy of the algorithm discussed in this chapter is the following. Any node v sets its clock to the largest value that neither exceeds the estimated maximum clock value in the network nor induces a clock skew larger than κ to any neighbor, just as dictated by algorithm $\mathcal{A}^{block}$. However, if v assumes that the clock of a neighbor is more than κ ahead, v is allowed to further increase its clock value instantly as long as $\Lambda_v^{\downarrow} \leq 2\kappa$. If a neighbor is assumed to be at least 2κ behind, v is blocked until the clock skew to a node that is ahead exceeds 2κ. In this case, v may again increase its clock value instantly subject to the constraint that $\Lambda_v^{\downarrow} \leq 3\kappa$ etc. This technique is more aggressive as we will see, but it also ensures that the nodes are blocked long enough for the nodes with the smallest clock values to catch up [35].[1]

Surprisingly, this algorithm, henceforth referred to as $\mathcal{A}^{opt}$, achieves a much better bound on the local skew even if the maximum progress rate is bounded, i.e., $\mathcal{A}^{opt}$ satisfies Condition (6.1). Moreover, the message frequency can be kept low without increasing the skew bounds substantially.

[1] A slightly different technique, which has been proposed earlier, basically achieves the same asymptotic bounds on the clock skews [34].

Algorithm 8.1 $\mathcal{A}^{opt}$: Message $\langle L^w, L^{max}_w\rangle$ received from node $w \in \mathcal{N}_v$.

```
if $L^{max}_w > L^{max}_v$ then
    $L^{max}_v := L^{max}_w$
    send $\langle L_v, L^{max}_v\rangle$ to all $u \in \mathcal{N}_v$
end if
if $L^w > \ell^w_v$ then
    $L^w_v := L^w$; $\ell^w_v := L^w$
    $\Lambda^{\uparrow}_v := \max_{u\in\mathcal{N}_v}\{L^u_v - L_v\}$
    $\Lambda^{\downarrow}_v := \max_{u\in\mathcal{N}_v}\{L_v - L^u_v\}$
end if
setClockRate()
```

8.1 Algorithm

The algorithm $\mathcal{A}^{opt}$ stores the same local variables L^w_v, ℓ^w_v, for all $w \in \mathcal{N}_v$, and L^{max}_v as algorithm $\mathcal{A}^{block}$, which are initialized to zero at time $t = 0$. Furthermore, L^{max}_v and the estimates L^w_v, for all $w \in \mathcal{N}_v$, are still increased at the hardware clock rate. The messages again contain both the logical clock value L_v and the estimate L^{max}_v of the largest clock value in the network. As in algorithm $\mathcal{A}^{block}$, each node v immediately sends a message to all neighbors whenever v obtains an estimate L^{max}_w larger than L^{max}_v in order to ensure that the estimated maximum clock value is disseminated as quickly as possible. Algorithm $\mathcal{A}^{opt}$ further computes the estimates $\Lambda^{\uparrow}_v$ and $\Lambda^{\downarrow}_v$ of the skew to the clocks in its neighborhood that are ahead and behind the most, respectively, just like $\mathcal{A}^{avg}$. In order to satisfy Condition (6.1), the clock value L_v can no longer be increased by a certain value instantaneously, i.e., $\mathcal{A}^{opt}$ can only manipulate the logical clock rate. For this purpose, the subroutine *setClockRate* is called, which adapts the logical clock rate according to the current situation. The actions that each node v takes upon receiving a message $\langle L^w, L^{max}_w\rangle$ from a node w are summarized in Algorithm 8.1.

The steps of the subroutine *setClockRate*, the key ingredient that distinguishes $\mathcal{A}^{opt}$ from the other algorithms, are summarized in Algorithm 8.2. By default, the logical clock runs at the hardware clock rate as well. The subroutine determines if and for how long the logical clock value has to increase more quickly than the hardware clock value (and the local variables) as follows. First, the amount R_v by which v would increase L_v if it were allowed to increase its clock value instantaneously is computed. Roughly speaking, the goal of the subroutine is to ensure that the clock skew to the neighbor whose clock is assumed to be behind the most and the clock skew to the neighbor with the largest estimated clock value are the same integer multiple of κ. The variable R_v is the largest value that satisfies this constraint. More precisely, if $\Lambda^{\uparrow}_v \leq s\kappa$ and $\Lambda^{\downarrow}_v \geq s\kappa$ for some $s \in \mathbb{N}_0$, v is blocked, i.e., $R_v = 0$. If v is

Algorithm 8.2 setClockRate(): Adjust the logical clock rate of the clock L_v according to the current information.

```
1: $R_v := \sup\left\{R \in \mathbb{R} \,\middle|\, \left\lfloor \frac{\Lambda_v^{\uparrow}-R}{\kappa} \right\rfloor \geq \left\lfloor \frac{\Lambda_v^{\downarrow}+R}{\kappa} \right\rfloor \right\}$
2: $R_v := \min\left\{\max\left\{\kappa - \Lambda_v^{\downarrow}, R_v\right\}, L_v^{max} - L_v\right\}$
3: if $R_v > 0$ then
4:     $\rho_v := 1 + \mu$
5:     $H_v^R := H_v + \frac{R_v}{\mu}$   (Reset $\rho_v := 1$ at time $H_v^R$)
6: else
7:     $\rho_v := 1$
8: end if
```

not blocked, $R_v > 0$ is exactly the increase of the clock value that causes v to be blocked. Line 1 of Algorithm 8.2 is a concise formulation of this rule. Although $\mathcal{A}^{opt}$ also strives to balance the skew to the neighbor's clock with the largest estimated clock value and the neighbor whose clock is assumed to be behind the most, similarly to algorithm $\mathcal{A}^{avg}$, the following simple example illustrates that $\mathcal{A}^{opt}$ is more aggressive than $\mathcal{A}^{avg}$. If $\Lambda_v^{\uparrow} = \Lambda_v^{\downarrow} = s\kappa + \frac{\kappa}{2}$ for any $s \in \mathbb{N}$, algorithm $\mathcal{A}^{opt}$ sets R_v to $\frac{\kappa}{2}$, whereas v does not increase its clock value when algorithm $\mathcal{A}^{avg}$ is used. As in algorithm $\mathcal{A}^{block}$, the value R_v may be at least $\kappa - \Lambda_v^{\downarrow}$, because a skew of κ is always tolerated, and at most $L_v^{max} - L_v$, since v must not increase the clock value to a value greater than L_v^{max} (see Line 2 of Algorithm 8.2).

In order to bound the increase of the logical clock, the algorithm dictates that any node's logical clock value may increase at most $1 + \mu$ times faster than its hardware clock value for a given $\mu > 0$. If the computed increase R_v is positive, v sets its *logical clock rate multiplier* ρ_v to $1 + \mu$, which ensures that $L_v(t_2) - L_v(t_1) = (1 + \mu)(H_v(t_2) - H_v(t_1))$ for any time interval $[t_1, t_2]$ where $\rho_v = 1 + \mu$. The clock rate multiplier ρ_v is reset to 1 as soon as the logical clock has made a progress that is R_v larger than the progress of the hardware clock, i.e., ρ_v is set to 1 when the hardware clock value reaches $H_v^R = H_v + R_v/\mu$. Naturally, new information can cause v to set H_v^R to a smaller or a larger value. Of course, if we do not insist on a strict upper bound on the logical clock rate, the computed R_v can simply be added to the logical clock value.

In the following, we assume that each node first sends a message to all of its neighbors at time $t = 0$. Note that the described algorithm only causes each node v to send a message if an estimate $L_w^{max} > L_v^{max}$ is received from a neighbor w. Since it is desirable to bound the message frequency, which is discussed in Section 8.2.2, we again employ the rule that a message is sent automatically whenever the estimate L_v^{max} reaches the next multiple of H_0.

8.2 Analysis

First, we point out that algorithm $\mathcal{A}^{opt}$ also belongs to the family $\mathfrak{A}$ if $\kappa \geq (1+\varepsilon)\mathcal{T} + 2\varepsilon H_0/(1+\varepsilon)$ and $\mu \geq 2\varepsilon/(1-\varepsilon)$. According to the description of $\mathcal{A}^{opt}$ in the preceding section, it follows immediately that $\mathcal{A}^{opt}$ has the first five properties. Lines 2-4 of the subroutine *setClockRate* (Algorithm 8.2) state that the clock rate is at least $1+\mu$ times the hardware clock rate if $\Lambda_v^{\downarrow} < (1+\varepsilon)\mathcal{T} + 2\varepsilon/(1+\varepsilon)H_0 \leq \kappa$ and $L_v < L_v^{max}$. Since the hardware clock rate is at least $1-\varepsilon$ and $\mu \geq 2\varepsilon/(1-\varepsilon)$, the logical clock rate is at least $(1+\mu)(1-\varepsilon) \geq 1+\varepsilon$. This lower bound on μ is quite natural because a smaller bound would imply that the skew between two clocks whose hardware clock rates are $1-\varepsilon$ and $1+\varepsilon$ may grow indefinitely. We conclude that $\mathcal{A}^{opt} \in \mathfrak{A}$, implying that $\mathcal{A}^{opt}$ satisfies Condition (6.2). Furthermore, Condition (6.1) is satisfied for $\alpha = 1-\varepsilon$ and $\beta = (1+\varepsilon)(1+\mu)$.

In our analysis of the local skew, we require that the parameters κ and μ are slightly larger. In particular, for an integer $\sigma \geq 2$ we require that

$$\mu \geq 7\sigma\frac{\varepsilon}{1-\varepsilon}. \tag{8.1}$$

We see that it suffices to set μ to roughly 14ε for any reasonable ε, i.e., the precision of the clocks reduces by merely one order of magnitude while clock skews are corrected. Naturally, μ can also be set to a larger value, which entails that σ may also become larger. The number σ has an impact on the local skew as we will see in Section 8.2.1. As far as the parameter κ is concerned, it is required that

$$\kappa \geq 2((1+\mu)\mathcal{T} + \bar{H}_0), \tag{8.2}$$

where

$$\bar{H}_0 := (2\varepsilon + \mu)H_0. \tag{8.3}$$

The parameter $\bar{H}_0$ is simply introduced to abbreviate the notation. The intuition behind this lower bound is that κ must be large enough to compensate for the inaccuracy of the known clock values of the neighboring nodes due to the maximum delay $\mathcal{T}$. In order to give the algorithm a chance to react to clock skews, the term $\mu\mathcal{T}$ is added. Obviously, the accuracy of the information about neighboring clocks deteriorates if H_0 is set to a large value. Since clock skew can be built up at a rate of at most $\mathcal{O}(\mu)$, the additional skew is bounded by $\mathcal{O}(\mu H_0)$. Therefore, κ must further include the term $\bar{H}_0$ as defined above. The factor of two is due to the fact that any node v might possess outdated information about both the nodes whose clocks are ahead and the nodes whose clocks are behind. Throughout this chapter, it is implicitly assumed that Condition (8.1) and Condition (8.2) are satisfied, and all lemmas and the main theorem make use of this assumption.

A crucial quantity in our analysis is the amount by which the increase of the logical clock value exceeds the minimum increase in the given interval, which is determined by the minimum hardware clock rate $1-\varepsilon$. For this purpose, we introduce the following definition.

Definition 8.1. $\forall t_1 \leq t_2 : \mathcal{I}_v(t_1,t_2) := L_v(t_2) - L_v(t_1) - (1-\varepsilon)(t_2 - t_1)$.

Apparently, it holds that $\mathcal{I}_v$ is *interval additive*:

$$\forall t_1 \leq t_2 \leq t_3 : \mathcal{I}_v(t_1,t_2) + \mathcal{I}_v(t_2,t_3) = \mathcal{I}_v(t_1,t_3).$$

By definition of $\mathcal{I}_v$, we further have that

$$\forall t_1 \leq t_2 : \mathcal{I}_v(t_1,t_2) \geq 0. \tag{8.4}$$

If $\rho_v = 1+\mu$ in a particular interval $[t_1,t_2]$, a node increases its logical clock value faster than its hardware clock value, i.e., $\mathcal{R}_v(t_1,t_2)$ is positive. Since $(1-\varepsilon)(t_2-t_1) \leq H_v(t_2) - H_v(t_1) \leq (1+\varepsilon)(t_2-t_1)$, it holds that

$$\forall t_1 \leq t_2 : \mathcal{R}_v(t_1,t_2) \leq \mathcal{I}_v(t_1,t_2) \leq \mathcal{R}_v(t_1,t_2) + 2\varepsilon(t_2-t_1). \tag{8.5}$$

Before we start our analysis of the worst-case clock skews, we prove that algorithm $\mathcal{A}^{opt}$ has the following essential property.

Lemma 8.2. *If the subroutine* setClockRate *is called at a node v at a time when no message is received, both ρ_v and H_v^R remain unchanged.*

Proof. Assume that the subroutine is called at time t, and let $t' < t$ be the time when the last message arrived at node v. By definition, the increase of the logical clock value in the time interval $(t',t]$ exceeds the increase of the local variables by $\mathcal{R}_v(t',t)$.

If $\mathcal{R}_v(t',t) = 0$, the subroutine *setClockRate* must have determined at time t' that the logical and the hardware clock value have to increase at the same rate. Therefore, if the subroutine *setClockRate* were called at time t, we would have that $R_v(t) \leq 0$ because $\Lambda_v^{\uparrow}$ and $\Lambda_v^{\downarrow}$, and also $L_v^{max} - L_v$, remain unaltered. Thus, the logical clock rate multiplier ρ_v would still be set to 1 after the procedure call at time t. Moreover, H_v^R is not changed at time t.

If $\mathcal{R}_v(t',t) > 0$ it holds that $\Lambda_v^{\uparrow}(t) = \Lambda_v^{\uparrow}(t') - \mathcal{R}_v(t',t)$ and $\Lambda_v^{\downarrow}(t) = \Lambda_v^{\downarrow}(t') + \mathcal{R}_v(t',t)$. The value R_v in Line 1 of Algorithm 8.2 thus reduces by exactly $\mathcal{R}_v(t',t)$. Moreover, $\kappa - \Lambda_v^{\downarrow}$ and $L_v^{max} - L_v$ also reduce by $\mathcal{R}_v(t',t)$ in Line 2, implying that $R_v(t) = R_v(t') - \mathcal{R}_v(t',t)$. If $R_v(t)$ evaluates to zero, i.e., $\mathcal{R}_v(t',t) = R_v(t')$, the hardware clock must have reached H_v^R and the logical clock rate multiplier has been reset to 1 by time t.[2] The logical clock rate multiplier is not set to $1+\mu$, because $R_v(t) \leq 0$, and H_v^R is again not changed at time t. If $R_v(t) > 0$, the logical clock rate

[2] Note that $R_v(t)$ cannot be negative because ρ_v is set to 1 as soon as $\mathcal{R}_v(t',t) = R_v(t)$ and no message is received in the interval $(t',t]$.

multiplier is set to $1+\mu$ until the hardware clock value reaches $H_v^R(t)$. Since $\mathcal{R}_v(t',t) = \mu(H_v(t) - H_v(t'))$, it holds that

$$\begin{aligned} H_v^R(t) = H_v(t) + \frac{R_v(t)}{\mu} &= H_v(t') + \frac{\mathcal{R}_v(t',t)}{\mu} + \frac{R_v(t') - \mathcal{R}_v(t',t)}{\mu} \\ &= H_v(t') + \frac{R_v(t')}{\mu} = H_v^R(t'). \end{aligned}$$

Hence, the logical clock rate multiplier remains $1+\mu$ until the same hardware clock time as before. We conclude that ρ_v and H_v^R remain exactly the same in all cases. □

This property is useful in that it allows us to determine the logical clock rate also at any time when no message is processed: We can simply assume that the variables are updated and the subroutine *setClockRate* is called, which does not cause the logical clock rate to change at any such time.

8.2.1 Clock Skew

Due to the fact that $\mathcal{A}^{opt} \in \mathfrak{A}$, the same bound on the global skew holds.

Corollary 8.3. *The global skew when executing algorithm $\mathcal{A}^{opt}$ on any graph G of diameter D is upper bounded by*

$$\mathcal{G} = (1+\varepsilon)\mathcal{T}D + \frac{2\varepsilon}{1+\varepsilon}H_0.$$

Since the bound on the global skew is already established, we focus our attention on the worst-case clock skew between neighboring nodes. Consider the maximum length of a path with a given average clock skew ΔL between the nodes of this path. The proof of the bound on the local skew relies on the fact that linearly increasing the average clock skew ΔL results in an exponential decrease in the length of the longest possible path that exhibits such an average clock skew. This fact implies that the average skew on paths of length one, i.e., between neighboring nodes, is logarithmically bounded in the diameter D. In particular, we show that the network is always in a *legal state*, which is defined as follows.

Definition 8.4 (Legal State)**.** *Given the integer $\sigma \geq 2$, we say that a network is in a* legal state *at time t, if and only if for all $s \in \mathbb{N}_0$ and all nodes $v, w \in V$ at distance*

$$d(v,w) \geq C_s := \frac{2\mathcal{G}}{\kappa}\sigma^{-s},$$

we have that

$$L_v(t) - L_w(t) \leq d(v,w)\left(s + \frac{1}{2}\right)\kappa.$$

Note that Corollary 8.3 shows that any two nodes v and w at a distance of at least C_0 cannot violate the legal state condition, because $L_v(t) - L_w(t) \le \mathcal{G} \le d(v,w)\frac{\kappa}{2}$ at all times t.

Two lemmas are required in order to prove the main theorem. The first lemma itself requires a simple helper lemma, which bounds the inaccuracy of the estimates L_v^w.

Lemma 8.5. *For all nodes $v \in V$ and $w \in \mathcal{N}_v$ it holds for all times t and $t' := \max\{t - \mathcal{T}, 0\}$ that*

$$L_v^w(t) > L_w(t') - \bar{H}_0. \tag{8.6}$$

Proof. If $t \le \mathcal{T}$, then $L_w(t') = 0$, implying that $L_v^w(t) \ge 0 > L_w(t') - \bar{H}_0$. Thus, we can assume that $t > \mathcal{T}$. Consequently, all nodes have already received a message from all their neighbors. Let t_s denote the time when w sent the largest clock value that v received at the latest at time t, and let $t_r \le t$ be the time when v received this clock value. Since v sets its estimate to the received value at time t_r, it holds that $L_v^w(t_r) = L_w(t_s)$.

If $t_s \ge t'$, we have that

$$L_v^w(t) \ge L_v^w(t_r) = L_w(t_s) \ge L_w(t') > L_w(t') - \bar{H}_0.$$

If $t_s < t'$, it must hold that $H_w(t') - H_w(t_s) < H_0$, as otherwise w sends a message that arrives at v at the latest at time t and that contains a larger clock value than the clock value sent at time t_s. Furthermore, $L_w(t') - L_w(t_s)$ is upper bounded by $(1+\varepsilon)(H_w(t') - H_w(t_s))$. Consequently, it holds that

$$\begin{aligned} \mathcal{I}_w(t_s, t') &= L_w(t') - L_w(t_s) - (1-\varepsilon)(t' - t_s) \\ &\le \left(1 + \mu - \frac{1-\varepsilon}{1+\varepsilon}\right)(H_w(t') - H_w(t_s)) \\ &< (\mu + 2\varepsilon)H_0 \overset{(8.3)}{=} \bar{H}_0. \end{aligned} \tag{8.7}$$

This observation and the fact that L_v^w is increased at the hardware clock rate in the interval $[t_r, t]$ allow us to bound

$$\begin{aligned} L_v^w(t) &\ge L_w(t_s) + (1-\varepsilon)(t - t_r) \\ &= L_w(t') - \mathcal{I}_w(t_s, t') - (1-\varepsilon)(t' - t_s) + (1-\varepsilon)(t - t_r) \\ &\overset{(8.7)}{>} L_w(t') - \bar{H}_0 - (1-\varepsilon)(t_r - t_s) + (1-\varepsilon)(t - t') \\ &\ge L_w(t') - \bar{H}_0. \end{aligned}$$

In the last step, we simply used that $t - t' = \mathcal{T}$ and $t_r - t_s \le \mathcal{T}$. □

The first lemma used in the proof of the main theorem basically states that the larger the clock skew is between two nodes v and w, the faster w can reduce it by increasing its clock value quickly.

Lemma 8.6. *Given $\xi \in \mathbb{R}$ and $s \in \mathbb{N}$, assume that the clock skew between two nodes v and w at time t_0 is*

$$L_v(t_0) - L_w(t_0) = d(v,w)\left(s - \frac{1}{2}\right)\kappa + \xi. \tag{8.8}$$

Define $t_1 := t_0 + \frac{\kappa C_{s-1}}{(1-\varepsilon)\mu} + \mathcal{T}$. If the network is in a legal state at time t_0, it follows that

$$\mathcal{I}_w(t_0, t_1) \geq \xi. \tag{8.9}$$

Proof. Define

$$\Xi(t) := \max_{u \in V}\left\{L_v(t_0) - L_u(t_0) - d(v,u)\left(s - \frac{1}{2}\right)\kappa - \mathcal{I}_u(t_0, t)\right\}.$$

Observe that if $\Xi(t) \leq 0$ holds at any time $t \geq t_0$, then for node w we have that

$$\mathcal{I}_w(t_0, t) \geq L_v(t_0) - L_w(t_0) - d(v,w)\left(s - \frac{1}{2}\right)\kappa = \xi.$$

Furthermore, $\Xi(\cdot)$ is monotonically decreasing, hence showing that $\Xi(t) \leq 0$ for any $t \leq t_1$ proves the lemma. We proceed by deriving an upper bound on $\Xi(t_0)$.

Consider any node $u \in V$. If $d(v,u) \geq C_0$, it holds that $L_v(t_0) - L_u(t_0) \leq d(v,u)\frac{\kappa}{2}$ due to the legal state condition, which was satisfied at time t_0. Since $s \geq 1$, it holds in this case that $L_v(t_0) - L_u(t_0) - d(v,u)\frac{\kappa}{2} \leq 0$, i.e., $\Xi(\cdot)$ cannot become positive because of a node u at distance $d(v,u) \geq C_0$. Hence, we can assume that $d(v,u) < C_0$, i.e., there is an integer $r \geq 1$ such that $d(v,u) \in [C_r, C_{r-1})$. The legal state condition states that

$$L_v(t_0) - L_u(t_0) - d(v,u)\left(s - \frac{1}{2}\right)\kappa \leq d(v,u)(r - s + 1)\kappa. \tag{8.10}$$

The right-hand side of this inequality is at most 0 for $r < s$. If $r \geq s$ it holds that

$$\begin{aligned} d(v,u)(r-s+1)\kappa &< C_{r-1}(r-s+1)\kappa \\ &= \sigma^{s-r} C_{s-1}(r-s+1)\kappa \\ &\leq \kappa C_{s-1} \end{aligned}$$

because $\sigma \geq 2$. Hence it follows that $\Xi(t_0) \leq \kappa C_{s-1}$.

The second step is to prove that $\Xi(\cdot)$ decreases at least at an average rate of $(1-\varepsilon)\mu$ until it reaches zero. In particular, we claim that

$$\Xi(t) \leq \max\{0, \Xi(t_0) - (1-\varepsilon)\mu(t - t_0) + (1-\varepsilon)\mu\mathcal{T}\} \tag{8.11}$$

for all times $t \geq t_0$.

This statement is trivially true for all times $t \leq t_0 + \mathcal{T}$, because in this case we only require that $\Xi(t) \leq \Xi(t_0)$, which follows from the fact that $\Xi(\cdot)$ is monotonically decreasing. Assume for the sake of contradiction that the claim that Inequality (8.11) holds at all times $t \geq t_0$ is false. Let $\bar{t}$ be the infimum of all times $t \geq t_0 + \mathcal{T}$ when $\Xi(t) > \Xi(t_0) - (1-\varepsilon)\mu(t-t_0) + (1-\varepsilon)\mu\mathcal{T}$ and let u be a node that maximizes $\Xi(\bar{t})$. Note that $u \neq v$ because v cannot cause the value of the function $\Xi(\cdot)$ to become positive due to the fact that $\mathcal{I}_v(t_0, t) \geq 0$. Since $\Xi(\cdot)$ is a continuous function, it holds at time $\bar{t}$ that

$$\begin{aligned}
& L_v(t_0) - L_u(t_0) - d(v,u)\left(s - \frac{1}{2}\right)\kappa - \mathcal{I}_u(t_0, \bar{t}) \\
= \quad & \Xi(t_0) - (1-\varepsilon)\mu(\bar{t} - t_0) + (1-\varepsilon)\mu\mathcal{T}. \qquad (8.12)
\end{aligned}$$

Consider any neighbor u' of u that is closer to v than u, i.e., the distance between v and u' is $d(v,u') = d(v,u) - 1$. By definition of $\bar{t}$, Inequality (8.11) holds for node u' at time $t' := \bar{t} - \mathcal{T} \geq t_0$. Thus, we have that

$$\begin{aligned}
& L_v(t_0) - L_{u'}(t_0) - d(v,u')\left(s - \frac{1}{2}\right)\kappa - \mathcal{I}_{u'}(t_0, t') \\
\leq \quad & \Xi(t_0) - (1-\varepsilon)\mu(t' - t_0) + (1-\varepsilon)\mu\mathcal{T} \\
= \quad & \Xi(t_0) - (1-\varepsilon)\mu(\bar{t} - t_0) + 2(1-\varepsilon)\mu\mathcal{T}. \qquad (8.13)
\end{aligned}$$

By subtracting Inequality (8.13) from Equation (8.12), we get that

$$\begin{aligned}
& L_{u'}(t_0) + \mathcal{I}_{u'}(t_0, t') - L_u(t_0) - \mathcal{I}_u(t_0, \bar{t}) \\
\geq \quad & (d(v,u) - d(v,u'))\left(s - \frac{1}{2}\right)\kappa - (1-\varepsilon)\mu\mathcal{T} \\
= \quad & \left(s - \frac{1}{2}\right)\kappa - (1-\varepsilon)\mu\mathcal{T}. \qquad (8.14)
\end{aligned}$$

Since $t' = \bar{t} - \mathcal{T}$, Lemma 8.5 can be used in order to lower bound $L_u^{u'}(\bar{t}) - L_u(\bar{t})$:

$$\begin{aligned}
L_u^{u'}(\bar{t}) - L_u(\bar{t}) \quad & \overset{(8.6)}{>} \quad L_{u'}(t') - \bar{H}_0 - L_u(\bar{t}) \\
& = \quad L_{u'}(t_0) + \mathcal{I}_{u'}(t_0, t') - L_u(t_0) - \mathcal{I}_u(t_0, \bar{t}) \\
& \qquad -(1-\varepsilon)(\bar{t} - t') - \bar{H}_0 \\
& \overset{(8.14)}{\geq} \quad \left(s - \frac{1}{2}\right)\kappa - (1-\varepsilon)\mu\mathcal{T} - (1-\varepsilon)\mathcal{T} - \bar{H}_0 \\
& \overset{(8.2)}{>} \quad (s-1)\kappa.
\end{aligned}$$

This bound and the fact that the estimate of the maximum clock value is at least as large as the estimated clock value of any neighbor imply that $L_u^{max}(\bar{t}) \geq L_u^{u'}(\bar{t}) > L_u(\bar{t}) + (s-1)\kappa \geq L_u(\bar{t})$. Thus, according to Lemma 8.2, the clock rate is $1+\mu$ unless the clock value of a neighboring node is too small. Consider an arbitrary node $u'' \in \mathcal{N}_u$. The distance between v and u'' is at most $d(v,u)+1$. As u'' did not violate the claimed bound at time $t' = \bar{t} - \mathcal{T}$, it holds that

$$\begin{array}{rl} & L_v(t_0) - L_{u''}(t_0) - d(v,u'')\left(s - \frac{1}{2}\right)\kappa - \mathcal{I}_{u''}(t_0,t') \\ \leq & \Xi(t_0) - (1-\varepsilon)\mu(\bar{t} - t_0) + 2(1-\varepsilon)\mu\mathcal{T}. \end{array}$$

By subtracting Equation (8.12), we get that

$$\begin{array}{rl} & L_u(t_0) + \mathcal{I}_u(t_0,\bar{t}) - L_{u''}(t_0) - \mathcal{I}_{u''}(t_0,t') \\ \leq & (d(v,u'') - d(v,u))\left(s - \frac{1}{2}\right)\kappa + (1-\varepsilon)\mu\mathcal{T} \\ \leq & \left(s - \frac{1}{2}\right)\kappa + (1-\varepsilon)\mu\mathcal{T}. \end{array} \tag{8.15}$$

This inequality is used to upper bound $L_u(\bar{t}) - L_u^{u''}(\bar{t})$:

$$\begin{array}{rcl} L_u(\bar{t}) - L_u^{u''}(\bar{t}) & \overset{(8.6)}{<} & L_u(\bar{t}) - L_{u''}(t') + \bar{H}_0 \\ & = & L_u(t_0) + \mathcal{I}_u(t_0,\bar{t}) - L_{u''}(t_0) - \mathcal{I}_{u''}(t_0,t') \\ & & +(1-\varepsilon)(\bar{t} - t') + \bar{H}_0 \\ & \overset{(8.15)}{\leq} & \left(s - \frac{1}{2}\right)\kappa + (1-\varepsilon)\mu\mathcal{T} + (1-\varepsilon)\mathcal{T} + \bar{H}_0 \\ & \overset{(8.2)}{<} & s\kappa. \end{array}$$

As we argued before, it holds that $L_u(\bar{t}) < L_u^{max}(\bar{t})$ because $\Lambda_u^{\uparrow}(\bar{t}) > 0$. Since $L_u(\bar{t}) < L_u^{max}(\bar{t})$, $\Lambda_u^{\uparrow}(\bar{t}) > (s-1)\kappa$, and $\Lambda_u^{\downarrow}(\bar{t}) \leq L_u(\bar{t}) - L_u^{u''}(\bar{t}) < s\kappa$, the subroutine *setClockRate* would set R_u to a positive value at time $\bar{t}$ and the logical clock rate multiplier to $1+\mu$. Thus, according to Lemma 8.2, the value of the logical clock rate multiplier ρ_{v_i} is $1+\mu$ at time $\bar{t}$. This result implies that the rate of increase of $\mathcal{I}_u$ at time $\bar{t}$ is at least $(1+\mu)(1-\varepsilon) - (1-\varepsilon) = (1-\varepsilon)\mu$. Since the rate of $\mathcal{I}_u$ is at least $(1-\varepsilon)\mu$ for any u for which Inequality (8.12) holds, it follows that the rate of $\Xi(\cdot)$ at time $\bar{t}$ is at most $-(1-\varepsilon)\mu$, contradicting the definition that $\bar{t}$ is the infimum of all times when the claim does not hold. Thus, at time $t_1 = t_0 + \frac{\kappa C_{s-1}}{(1-\varepsilon)\mu} + \mathcal{T}$, it holds that $\Xi(t_1) \leq \max\{0, \Xi(t_0) - (1-\varepsilon)\mu(t_1 - t_0) + (1-\varepsilon)\mu\mathcal{T}\} = 0$ as desired. □

The second lemma shows that the clock skew can only increase slowly once it reaches a certain level. More precisely, for any $s \in \mathbb{N}$, we consider the path on which the average clock skew exceeds $s\kappa$ the most. If v is the node with the largest and w is the node with the smallest clock value among all nodes on this path, then v's logical clock runs at the hardware clock rate, i.e., the clock skew between v and w can only grow further at a rate of at most 2ε. Moreover, the clock skew decreases if w increases its clock value quickly. In order to abbreviate the notation, the following definition is introduced.

Definition 8.7. *Given a node $w \in V$ and $s \in \mathbb{N}$, define for any time t*

$$\Psi_w^s(t) := \max_{v \in V} \{L_v(t) - L_w(t) - s\kappa d(v,w)\} \geq 0.$$

Lemma 8.8. *If $\Psi_w^s(t) > 0$ for all $t \in (t_0, t_1]$, then it holds at any time $t \in [t_0, t_1]$ that*

$$\Psi_w^s(t) \leq 2\varepsilon(t - t_0) - \mathcal{I}_w(t_0, t) + \frac{\kappa}{7\sigma} + \Psi_w^s(t_0). \tag{8.16}$$

Proof. Since $\Psi_w^s(\cdot) > 0$ implies that the clock skew between some nodes v and w is at least $\kappa > 2\mathcal{T} > 2\varepsilon\mathcal{T}$, and at most $2\varepsilon\mathcal{T}$ can be built up until time $\mathcal{T}$, t_0 must be a point in time after $\mathcal{T}$, i.e., we can assume that all nodes have already received messages from all their neighbors.

Assume for the sake of contradiction that $\bar{t} \in [t_0, t_1]$ is the infimum of times when the claim is false, i.e., there is a node $v \in V$ such that

$$L_v(\bar{t}) - L_w(\bar{t}) - s\kappa d(v,w) = 2\varepsilon(\bar{t} - t_0) - \mathcal{I}_w(t_0, \bar{t}) + \frac{\kappa}{7\sigma} + \Psi_w^s(t_0). \tag{8.17}$$

Note that $v \neq w$ and thus $d(v,w) \geq 1$ because w cannot cause $\Psi_w^s(\cdot)$ to become positive. Since $d(v,w) \geq 1$, a neighbor $u \in \mathcal{N}_v$ at distance $d(u,w) = d(v,w) - 1$ from w exists. Let t_s denote the time when u sent the largest clock value that v received at a time $t_r \leq \bar{t}$. We need to distinguish between the following two cases.

If $t_s \geq t_0$, Inequality (8.16) was not violated at time t_s, which allows us to bound

$$\begin{aligned}
L_v(\bar{t}) - L_v^u(\bar{t}) \quad &= \quad L_v(t_r) - L_v^u(t_r) + \mathcal{R}_v(t_r, \bar{t}) \\
&\overset{(8.5)}{\geq} \quad L_v(t_r) - L_u(t_s) + \mathcal{I}_v(t_r, \bar{t}) - 2\varepsilon(\bar{t} - t_r) \\
&= \quad L_v(\bar{t}) - L_w(\bar{t}) - (L_u(t_s) - L_w(\bar{t})) \\
&\qquad -(1-\varepsilon)(\bar{t} - t_r) - 2\varepsilon(\bar{t} - t_r) \\
&= \quad L_v(\bar{t}) - L_w(\bar{t}) - (L_u(t_s) - L_w(t_s)) \\
&\qquad +\mathcal{I}_w(t_s, \bar{t}) + (1-\varepsilon)(\bar{t} - t_s) \\
&\qquad -(1-\varepsilon)(\bar{t} - t_r) - 2\varepsilon(\bar{t} - t_r) \qquad (8.18) \\
&\overset{(8.16,8.17)}{\geq} \quad s\kappa(d(v,w) - d(u,w)) + (1-\varepsilon)(t_r - t_s) \\
&\qquad -\mathcal{I}_w(t_0, \bar{t}) + \mathcal{I}_w(t_0, t_s) + \mathcal{I}_w(t_s, \bar{t}) \\
&\qquad -2\varepsilon(\bar{t} - t_r) + 2\varepsilon(\bar{t} - t_0) - 2\varepsilon(t_s - t_0) \\
&= \quad s\kappa + (1+\varepsilon)(t_r - t_s) \geq s\kappa.
\end{aligned}$$

If $t_s < t_0$, note that the time difference $t_s - t_0$ is bounded because each node sends a message to its neighbors at the latest after its hardware clock value increased by H_0, i.e., after at most $H_0/(1-\varepsilon)$ time, since it last sent a message. If $t_s \leq t_0 - H_0/(1-\varepsilon) - \mathcal{T}$, another message, containing a larger clock value, is sent at a time $t'_s \leq t_0 - \mathcal{T}$, which arrives at a time $t'_r \leq t_0$, contradicting the assumption that the message sent at time t_s contains the largest clock value that v receives from u until $\bar{t} \geq t_0$. Hence it follows that $t_s > t_0 - H_0/(1-\varepsilon) - \mathcal{T}$ and thus

$$t_0 - t_r \leq t_0 - t_s < \frac{H_0}{1-\varepsilon} + \mathcal{T} \overset{(8.3)}{<} \frac{\bar{H}_0}{(1-\varepsilon)\mu} + \mathcal{T} \overset{(8.1)}{\leq} \frac{\bar{H}_0}{7\sigma\varepsilon} + \mathcal{T}. \tag{8.19}$$

The clock skew between L_u and L_w at time t_s is bounded by

$$\begin{aligned} L_u(t_s) - L_w(t_s) &= L_u(t_0) - L_w(t_0) - \mathcal{I}_u(t_s,t_0) + \mathcal{I}_w(t_s,t_0) \\ &\leq s\kappa d(u,w) - \mathcal{I}_u(t_s,t_0) + \mathcal{I}_w(t_s,t_0) + \Psi_w^s(t_0) \\ &\overset{(8.4)}{\leq} s\kappa d(u,w) + \mathcal{I}_w(t_s,t_0) + \Psi_w^s(t_0). \end{aligned} \tag{8.20}$$

In this case, the estimated clock skew between v and u at time $\bar{t}$ is

$$\begin{aligned} L_v(\bar{t}) - L_v^u(\bar{t}) &\overset{(8.18)}{\geq} L_v(\bar{t}) - L_w(\bar{t}) - (L_u(t_s) - L_w(t_s)) + \mathcal{I}_w(t_s,\bar{t}) \\ &\quad +(1-\varepsilon)(t_r - t_s) - 2\varepsilon(\bar{t} - t_r) \\ &\overset{(8.17,8.20)}{\geq} s\kappa(d(v,w) - d(u,w)) - \mathcal{I}_w(t_0,\bar{t}) - \mathcal{I}_w(t_s,t_0) \\ &\quad +\mathcal{I}_w(t_s,\bar{t}) + \frac{\kappa}{7\sigma} + (1-\varepsilon)(t_r - t_s) - 2\varepsilon(t_0 - t_r) \\ &\overset{(8.19)}{>} s\kappa + \frac{\kappa}{7\sigma} - 2\varepsilon\left(\mathcal{T} + \frac{\bar{H}_0}{7\sigma\varepsilon}\right) > s\kappa. \end{aligned}$$

In the last step, we used that $\kappa \overset{(8.2)}{>} 2(\mu\mathcal{T} + \bar{H}_0) \overset{(8.1)}{>} 14\sigma\varepsilon\mathcal{T} + 2\bar{H}_0$. Thus, $L_v(\bar{t}) - L_v^u(\bar{t}) \geq s\kappa$ holds in either case. Applying the same arguments to any node $u' \in \mathcal{N}_v$ yields that $L_v^{u'}(\bar{t}) - L_v(\bar{t}) \leq s\kappa$, as all terms in the previous estimates switch signs, except the term $s\kappa$ because $d(v,w) - d(v,u') \geq -1$. Since these estimate holds for any node u', we have that $\Lambda_v^{\uparrow}(\bar{t}) \leq s\kappa$. Moreover, it holds that $\Lambda_v^{\downarrow}(\bar{t}) \geq L_v(\bar{t}) - L_v^u(\bar{t}) \geq s\kappa$, which implies that R_v evaluates to zero in Line 1 of the subroutine *setClockRate* (Algorithm 8.2) if the subroutine is called at time $\bar{t}$. Given that $s \in \mathbb{N}$, we further know that $\kappa - \Lambda_v^{\downarrow}(\bar{t}) \leq \kappa(1-s) \leq 0$. Hence, $R_v(\bar{t}) = 0$ and the logical clock rate is the same as the hardware clock rate at time $\bar{t}$ according to Lemma 8.2. Since the minimum progress rate of any node is $1-\varepsilon$ and the progress rate of v at time $\bar{t}$ is at most $1+\varepsilon$, the clock skew can only increase at the rate 2ε, a contradiction to the assumption that $\bar{t}$ is the infimum of all times when the claim is violated, which concludes the proof. □

We are now in the position to prove the main theorem, which states that the local skew grows logarithmically with the diameter D of the graph.

Theorem 8.9. *The local skew when executing algorithm $\mathcal{A}^{opt}$ on any graph G of diameter D is upper bounded by*

$$\kappa \left(\left\lceil \log_\sigma \frac{2\mathcal{G}}{\kappa} \right\rceil + \frac{1}{2} \right).$$

Proof. Let $\mathcal{G}' \geq \mathcal{G}$ be the number for which $\log_\sigma \frac{2\mathcal{G}'}{\kappa} = \left\lceil \log_\sigma \frac{2\mathcal{G}}{\kappa} \right\rceil$. It is convenient to assume, without loss of generality, that $\mathcal{G}'$ is the bound on the global skew because in this case $C_s \in \mathbb{N}$ for all $s \in \{0, \ldots, s_{max}\}$, where $s_{max} := \log_\sigma \frac{2\mathcal{G}'}{\kappa}$.

By definition, a skew of more than $d(v,w)\left(s_{max} + \frac{1}{2}\right)\kappa$ between the clocks of any two nodes v and w at distance $d(v,w) \geq C_{s_{max}}$ cannot occur as long as the network is in a legal state. Since $C_{s_{max}} = 1$, the claimed bound on the worst-case skew between neighboring nodes can only be violated if the network is not in a legal state. Thus, if the network is always in a legal state, the theorem follows immediately.

Assume for the sake of contradiction that $\bar{t}$ is the infimum of all times when the network is not in a legal state. As argued before, the legal state condition cannot be violated for $s = 0$ as a violation implies that the clock skew between two nodes exceeds $\mathcal{G}' \geq \mathcal{G}$, a contradiction to Corollary 8.3. Hence, two nodes v and w at distance $d(v,w) \geq C_s$ for some $s \in \{1, \ldots, s_{max}\}$ exist such that

$$L_v(\bar{t}) - L_w(\bar{t}) = d(v,w)\left(s + \frac{1}{2}\right)\kappa. \tag{8.21}$$

Define $t_0 := \bar{t} - \frac{\kappa C_{s-1}}{(1-\varepsilon)\mu} - \mathcal{T}$. Since $\kappa C_s \geq \kappa > (1-\varepsilon)\mu\mathcal{T}$, it holds that

$$\bar{t} - t_0 < \frac{(\sigma+1)}{(1-\varepsilon)\mu}\kappa C_s. \tag{8.22}$$

If $\Psi_w^s = 0$ at some point in time in the interval $[t_0, \bar{t}]$, choose the largest $t \in [t_0, \bar{t}]$ such that $\Psi_w^s(t) = 0$. Note that $t < \bar{t}$ because at time $\bar{t}$ we have that

$$\begin{aligned}
\Psi_w^s(\bar{t}) &\geq L_v(\bar{t}) - L_w(\bar{t}) - s\kappa d(v,w) \\
&\overset{(8.21)}{=} \frac{\kappa}{2} d(v,w) \\
&\overset{d(v,w)\geq C_s}{\geq} \frac{\kappa}{2} C_s > 0.
\end{aligned} \tag{8.23}$$

In this case, by applying Lemma 8.8, we get that

$$\begin{aligned}\frac{\kappa}{2}C_s &\overset{(8.23)}{\leq} \Psi_w^s(\bar{t})\\ &\overset{(8.16)}{\leq} 2\varepsilon(\bar{t}-t) - \mathcal{I}_w(t,\bar{t}) + \frac{\kappa}{7\sigma}\\ &\overset{(8.4)}{\leq} 2\varepsilon(\bar{t}-t_0) + \frac{\kappa}{7\sigma}C_s\\ &\overset{(8.22)}{<} 2\varepsilon\frac{\sigma+1}{(1-\varepsilon)\mu}\kappa C_s + \frac{\kappa}{7\sigma}C_s.\end{aligned}$$

This inequality implies that $2\varepsilon\frac{\sigma+1}{(1-\varepsilon)\mu}\kappa C_s > (\frac{1}{2}-\frac{1}{7\sigma})\kappa C_s$ and thus

$$\mu < \frac{28\varepsilon\sigma(\sigma+1)}{(1-\varepsilon)(7\sigma-2)} \overset{\sigma\geq 2}{\leq} 7\sigma\frac{\varepsilon}{1-\varepsilon},$$

a contradiction to Condition (8.1).

Hence, it must hold that $\Psi_w^s(t) > 0$ for all $t \in [t_0, \bar{t}]$. Another application of Lemma 8.8 yields that

$$\begin{aligned}\frac{\kappa}{2}C_s - \Psi_w^s(t_0) &\overset{(8.23)}{\leq} \Psi_w^s(\bar{t}) - \Psi_w^s(t_0)\\ &\overset{(8.16)}{\leq} 2\varepsilon(\bar{t}-t_0) - \mathcal{I}_w(t_0,\bar{t}) + \frac{\kappa}{7\sigma}\\ &\overset{(8.22)}{<} 2\varepsilon\frac{\sigma+1}{(1-\varepsilon)\mu}\kappa C_s - \mathcal{I}_w(t_0,\bar{t}) + \frac{\kappa}{7\sigma}.\end{aligned}$$

By rearranging the terms we get that

$$\Psi_w^s(t_0) > \frac{\kappa}{2}C_s - 2\varepsilon\frac{\sigma+1}{(1-\varepsilon)\mu}\kappa C_s + \mathcal{I}_w(t_0,\bar{t}) - \frac{\kappa}{7\sigma}. \tag{8.24}$$

Since $t_0 = \bar{t} - \frac{\kappa C_{s-1}}{(1-\varepsilon)\mu} - \mathcal{T}$, Lemma 8.6 can be used to lower bound $\mathcal{I}_w(t_0,\bar{t})$. As $\mathcal{I}_w(t_0,\bar{t}) \geq \xi$ if there is *any* node u such that $L_u(t_0) - L_w(t_0) = d(u,w)\left(s-\frac{1}{2}\right)\kappa + \xi$, we have that

$$\begin{aligned}\mathcal{I}_w(t_0,\bar{t}) &\overset{(8.8)}{\geq} \max_{u\in V}\left\{L_u(t_0) - L_w(t_0) - \left(s-\frac{1}{2}\right)\kappa d(u,w)\right\}\\ &\geq \Psi_w^s(t_0)\\ &\overset{(8.24)}{>} \frac{\kappa}{2}C_s - 2\varepsilon\frac{\sigma+1}{(1-\varepsilon)\mu}\kappa C_s + \mathcal{I}_w(t_0,\bar{t}) - \frac{\kappa}{7\sigma}\\ &\geq \left(\frac{1}{2}-\frac{1}{7\sigma}\right)\kappa C_s - 2\varepsilon\frac{\sigma+1}{(1-\varepsilon)\mu}\kappa C_s + \mathcal{I}_w(t_0,\bar{t}).\end{aligned}$$

This result again implies that $2\varepsilon\frac{\sigma+1}{(1-\varepsilon)\mu}\kappa C_s > (\frac{1}{2}-\frac{1}{7\sigma})\kappa C_s$, which leads to the contradiction that $\mu < 7\sigma\frac{\varepsilon}{1-\varepsilon}$. Thus, the network can never leave the legal state, which proves the claimed bound on the local skew. □

Note that this theorem also holds if each node v increases its logical clock value by the value R_v computed in the subroutine *setClockRate* at once instead of raising the logical clock rate: Theorem 8.9 is proved using Lemma 8.6 and Lemma 8.8. Clearly, increasing the clock values instantly is a more aggressive strategy and it is easy to see that Lemma 8.6 still holds. In Lemma 8.8, we found that $R_v = 0$ if the clock skew between two nodes v and w is sufficiently large, which implies that v increases its logical clock value at the hardware clock rate in this situation even if the nodes are allowed to increase the clock values instantaneously.

Since the base of the logarithm σ is the largest integer such that Inequality (8.1) holds (for a given $\mu \geq 14\varepsilon/(1-\varepsilon)$), it follows that $\sigma \in \Theta(\mu/\varepsilon)$. Thus, choosing $\kappa \in \Theta((1+\mu)\mathcal{T} + \mu H_0)$ results in a local skew of

$$\mathcal{O}\left(((1+\mu)\,\mathcal{T} + \mu H_0)\log_{\mu/\varepsilon} D\right).$$

When choosing $\mu \in \Theta(\hat{\varepsilon}) = \Theta(\varepsilon)$ and $H_0 \in \mathcal{O}(\mathcal{T}/\mu) = \mathcal{O}(\mathcal{T}/\varepsilon)$, the local skew is upper bounded by $\mathcal{O}(\mathcal{T}\log D)$. Note that choosing $\mu \in \Theta(\varepsilon)$ entails that the maximum logical clock rate β is upper bounded by $1 + \mathcal{O}(\varepsilon)$. Moreover, if the logical clock rate is allowed to be larger than the hardware clock rate by a constant factor, i.e., $\mu \in \Theta(1)$, and we choose $H_0 \in \mathcal{O}(\mathcal{T})$, the bound on the local skew reduces to $\mathcal{O}(\mathcal{T}\log_{1/\varepsilon} D)$.

8.2.2 Message Frequency & Maximum Message Size

An essential optimization criterion is the frequency of communication that is required to sustain a certain degree of synchronization. As the virtual clock L^{max} increases at most at the rate $1+\varepsilon$, it holds at any time t that $L^{max}(t) \leq (1+\varepsilon)t$. Recall that $\Delta = \max_{v \in V} \delta(v)$ denotes the largest node degree in the network as defined in Section 1.3. A message is only sent whenever the estimate of the largest clock value in the network reaches the next multiple of H_0, which implies that no node sends more than

$$\left(1 + \left\lfloor \frac{(1+\varepsilon)t}{H_0} \right\rfloor\right)\Delta$$

messages up to time t.[3] Furthermore, any node sends messages at the latest after its hardware clock advanced by H_0 since the last send event. These observations imply that the *amortized message frequency*, i.e., the average number of messages sent per time unit, is bounded by $\Theta(\Delta/H_0)$ when algorithm $\mathcal{A}^{opt}$ is used. The bound on the local skew states that choosing $H_0 \in \Theta(\mathcal{T}/\mu)$ still results in a local skew of $\mathcal{O}(\mathcal{T}\log_{\mu/\varepsilon} D)$, i.e., the bound on the worst-case clock skew between neighboring nodes increases merely by a constant factor if the average time between send events is $\Theta(\mathcal{T}/\mu)$. Since

[3] The additional 1 is due to the fact that all nodes send messages at time $t = 0$.

it is possible to choose $\mu \in \Theta(\varepsilon)$, we can get an amortized message frequency of $\Theta(\varepsilon\Delta/\mathcal{T})$. As far as the global skew is concerned, choosing $H_0 \in \Theta(\mathcal{T}/\varepsilon)$ results in worst-case clock skew of $(1+\varepsilon)D\mathcal{T} + \mathcal{O}(\mathcal{T})$, which does not constitute a substantial increase if the diameter D is (sufficiently) large. Thus, assuming that $\varepsilon \ll \mathcal{T}$, the amortized message frequency can be quite low without increasing the asymptotic bounds on both the global and the local skew.

However, a node v may receive up to $\mathcal{O}(\mathcal{G}/H_0)$ messages, all containing a larger estimate of the maximum clock value, in an arbitrarily small period of time. Each of these messages causes v to send messages to all its neighbors, which means that the algorithm does not guarantee a strong upper bound on the message frequency. Note that the estimates of the maximum clock value are distributed quickly in this scenario, which implies that the clock skews may be reduced at a higher rate. For this reason, one might argue that this behavior is in fact desirable. Nevertheless, it may be required that the message frequency is bounded at all times, which can be achieved by forcing nodes to wait for $\Theta(H_0)$ time until they send the next set of messages. This modification ensures that the message frequency is upper bounded by $\mathcal{O}(\Delta/H_0)$. Obviously, the clock skews may increase during this idle time, and this additional clock skew must be compensated for by adding another term in the order of $\Theta(\mu H_0)$ to κ. The drawback of this straightforward modification of algorithm $\mathcal{A}^{opt}$ is that the time it takes to propagate information through the whole network increases by $\mathcal{O}(D\,H_0)$, which entails that the upper bound on the global skew deteriorates to $(1+\varepsilon)D\mathcal{T} + \mathcal{O}(\varepsilon D\,H_0)$ because all nodes locally increase their estimates of the maximum clock value at their hardware clock rate. Thus, this modified algorithm provides a trade-off between the minimum message frequency and the upper bound on the global skew that depends on the duration of the forced idle time.

In order to bound the maximum message size, the nodes cannot exchange the (technically unbounded) clock values themselves. Instead, nodes can communicate the progress of their clocks since they last sent a message. Since the progress may be an arbitrary real number in the range $[H_0, (1+\mu)H_0]$, it cannot be encoded using a bounded number of bits. However, if $\mu \in \mathcal{O}(1)$, the integer part of the progress can be encoded using $\mathcal{O}(\log H_0)$ bits, and the error is at most ε^c, for a constant $c \in \mathbb{N}$, if $\lceil c\log(1/\varepsilon)\rceil$ bits are used to encode the fractional part of the progress. This error is negligible, as the sent clock values may in fact be the real clock values in an execution with slightly different clock rates. As far as the estimate L_v^{max} of any node v is concerned, the increase may be $\Theta(\mathcal{G})$, which requires $\mathcal{O}(\log(\mathcal{T}D))$ bits. This dependency on the diameter can be avoided by limiting the maximum increase that v *informs* its neighbors about in a single message to $\mathcal{O}(H_0)$. Since the messages contain estimates of the maximum clock value that are multiples of H_0, these estimates can be encoded using $\mathcal{O}(1)$ bits. If the estimate is larger than the sent value, v stores the difference and informs its neighbors about the remaining increase, which is still a multiple of H_0, in its subsequent messages. The intuition behind this strategy is that if v receives an estimate of the maximum clock value that is much larger than its own, then the estimates must have propagated quickly. In the scenario where all messages are as slow as possible, this situation would not occur. Thus, the estimates of the maximum clock value in the network that the nodes with the smallest clock values receive are sufficiently large and Theorem 8.3 still holds. We conclude that algorithm $\mathcal{A}^{opt}$ can be implemented in such a way that the maximum message size is bounded by $\mathcal{O}(\log(H_0/\varepsilon))$. Thus, assuming again that $\varepsilon \ll \mathcal{T}$, the foregoing discussion reveals that H_0 can be set to a sufficiently large value such that a small number of bits need to be sent in constant time for any practical maximum degree Δ. Naturally, if all messages must contain globally unique node identifiers, each message requires $\mathcal{O}(\log n)$ additional bits.

8.2.3 Alternative Models

It is important to understand to what extent the described techniques are applicable to other clock synchronization models. The applicability of $\mathcal{A}^{opt}$ to other models that differ from our model in various ways are now briefly discussed.

Throughout this part of the thesis, we assumed that estimates $\hat{\varepsilon}$ and $\hat{\mathcal{T}}$, where $\varepsilon \leq \hat{\varepsilon} \in \mathcal{O}(\varepsilon)$ and $\mathcal{T} \leq \hat{\mathcal{T}} \in \mathcal{O}(\mathcal{T})$, are known to all nodes. Both algorithm $\mathcal{A}^{block}$ and $\mathcal{A}^{opt}$ require these estimates in order to set κ to an appropriate value. Additionally, $\mathcal{A}^{opt}$ needs $\hat{\varepsilon}$ to determine (a lower bound on) μ. As long as the nodes know an upper bound $\hat{\varepsilon} < 1$ on the clock drift rate, κ and μ can be set to values that are large enough to ensure that both Theorem 8.3 and Theorem 8.9 hold. However, if $\hat{\varepsilon}$ may be arbitrarily larger than ε, we can no longer guarantee that the maximum logical clock rate is upper bounded by $1 + \mathcal{O}(\varepsilon)$.[4] We will now briefly illustrate how the maximum delay $\mathcal{T}$ can be estimated dynamically while a clock synchronization algorithm is running. If all nodes acknowledge the reception of messages, the nodes can simply keep track of the longest round-trip time ever measured using their own hardware clock. The estimate $\hat{\mathcal{T}}$ of the maximum (one-way) delay is then determined by multiplying this round-trip time by $1/(1-\hat{\varepsilon})$, which ensures that the computed value is at least the round-trip time measured in *real time* because the hardware clock rate is lower bounded by $1-\varepsilon \geq 1-\hat{\varepsilon}$. Whenever a node sets its estimate to a larger value, the new value is flooded through the network in order to ensure that all nodes (eventually) work with the same estimate. A node that receives an estimate that is larger than its current estimate sets $\hat{\mathcal{T}}$ to the received value and re-computes its parameters. In order to bound the number of updates, $\hat{\mathcal{T}}$ can be rounded up to the next

[4] It is unlikely that this is a major issue in practice as the (maximum) clock drift of the used hardware clocks can be measured fairly accurately.

power of two. Thus, for any constant initial guess, there are at most $\log \mathcal{T}$ updates in the network.

For some distributed systems it is more adequate to define that each message is delayed by a value in the range $[\mathcal{T}_0, \mathcal{T}_0+\mathcal{T}]$, i.e., apart from a jitter of at most $\mathcal{T}$, the delay of each message is $\mathcal{T}_0 \gg \mathcal{T}$. We can assume that the basic delay $\mathcal{T}_0$ is known. It is not hard to see that the lower bounds on both the global and the local skew still hold in this model; the only difference is that the parameter $\mathcal{T}$ in the skew bounds now denotes the maximum jitter (and no longer the maximum delay). The algorithm has to be modified slightly in that the basic delay $\mathcal{T}_0$ must be added to each received clock value, including the estimate of the maximum clock value in the network. Due to the fact that it may take $D(\mathcal{T}_0 + \mathcal{T}) \gg D\mathcal{T}$ time to distribute a new estimate of the maximum clock value to all nodes, the upper bound on the global skew increases by an additive term in the order of $\mathcal{O}(\varepsilon D\mathcal{T}_0)$. Thus, the global skew is still upper bounded by $\mathcal{O}(D\mathcal{T})$ subject to the condition that $\mathcal{T}_0 \in \mathcal{O}(\mathcal{T}/\varepsilon)$. As far as the local skew is concerned, Lemma 8.6 has to be adapted because $\Xi(\cdot)$ now reduces merely at an amortized rate of $\Omega(\min\{\mu, \mathcal{T}/\mathcal{T}_0\})$, which implies that choosing $\mu \in \omega(\mathcal{T}/\mathcal{T}_0)$ does not improve the bound on the local skew anymore. However, if $\mathcal{T} \in \Omega(\varepsilon^c \mathcal{T}_0)$ for a constant $c \in (0,1)$, choosing $\mu \in \Theta(\varepsilon^c)$ and adapting κ accordingly still results in an asymptotically optimal local skew of $\mathcal{O}(\mathcal{T} \log_{1/\varepsilon} D)$. In summary, the skew bounds depend on the unknown part of the message delay in this model, and we get the same asymptotic bounds on the worst-case clock skews as long as the maximum delay is not substantially larger than the maximum jitter.

We further assumed for the sake of simplicity that all nodes start their clocks *synchronously* at time $t = 0$, which is not a realistic assumption because all communication in our model is asynchronous and there is no centralized authority to activate the clocks. However, this assumption is not critical for the following reason. If a single node v activates its clocks and then floods an activation message through the network, the time that passes until the last node activates its clock is bounded by $D\mathcal{T}$. During this time, v can only build up a skew of at most $(1+\varepsilon)D\mathcal{T}$, which is less than the proven bound $\mathcal{G}$ on the global skew in case of a synchronous start. Afterwards, the clock skew can only grow further at a rate of 2ε and the nodes with the smallest clock values continually receive messages containing larger clock values, which suggests that the clock skew cannot become much larger than $(1+\varepsilon)D\mathcal{T}$. In fact, it can be shown the same bound $\mathcal{G}$ on the global skew holds even if the nodes cannot activate their clocks simultaneously [35]. Intuitively, the bound on the local skew also remains the same because it takes at least $\Omega((\mathcal{T}\log D)/\mu)$ time to build up a clock skew of $\Omega(\mathcal{T}\log D)$ between two nodes v and w, and at least all clocks of nodes up to a distance of $\Omega((\log D)/\mu)$ from either v or w have been activated in the meantime. Hence, no node in v's or w's vicinity can be blocked because a neighbor has not yet activated its clock.

Since computational devices typically synchronize their operations internally based on a *clock pulse*, an alternative model is to assume that nodes may only act at such clock pulses. In particular, the clock values can only be increased by discrete values at each clock pulse and the clock value remains unchanged until the next clock pulse. Not surprisingly, the constant idle time between clock pulses causes only a small increase in the bounds on the worst-case clock skews, i.e., the same asymptotic results hold in this model as well [35].

As mentioned in Section 6.2, there is a lot of work on *external* clock synchronization algorithms, where a source of real time is available and the objective is to synchronize all clocks to this source. Let node v_0 be this source of real time, i.e., $H_{v_0}(t) = L_{v_0}(t) = t$. A simple trick allows us to use algorithm A^{opt} to synchronize the clocks in this model as well. The computed logical clock rate is always multiplied by $1/(1+\hat{\varepsilon})$, and the local variables are also increased at the hardware clock rate multiplied by $1/(1+\hat{\varepsilon})$, where $\hat{\varepsilon} \geq \varepsilon$ again denotes the estimate of ε. This rate reduction ensures that all nodes have a clock rate that is upper bounded by 1, and thus all nodes always strive to catch up to v_0, similarly to how every node v tries to increase its clock value to L_v^{max} in the internal clock synchronization model. The main difference is that the minimum logical clock rate is $(1-\varepsilon)/(1+\hat{\varepsilon}) \in 1-\mathcal{O}(\varepsilon)$, which can easily be accounted for when determining μ and κ. Thus, the algorithm still guarantees basically the same clock skew bounds with slightly increased parameters μ and κ. Note that a variant of the linear envelope condition (Condition (6.2)) holds, which states that for all nodes v and all times t we have that $(1-\mathcal{O}(\varepsilon))t \leq L_v(t) \leq t$. Of course, the logical clock values also do not deviate from real time by more than the global skew at any point in time.

A similar technique is applicable if the linear envelope condition is replaced by the more stringent condition that

$$\forall v \in V\, \forall t: \ \min_{w \in V}\{H_w(t)\} \leq L_v(t) \leq \max_{w \in V}\{H_w(t)\},$$

i.e., all logical clock values must always be at least as large as the smallest hardware clock value and at most as large as the largest hardware clock value in the network. Obviously, it also holds that $(1-\varepsilon)t \leq L_v(t) \leq (1+\varepsilon)t$ for all $v \in V$ at all times t if this condition is satisfied because the hardware clock rates are always in the range $[1-\varepsilon, 1+\varepsilon]$. In this case, each node v stores and sends an estimate of the maximum *hardware clock value* H_v^{max} instead of L_v^{max}. In order to guarantee that the estimate H_v^{max} does not exceed the largest hardware clock value in the network, it is only increased at the rate $\frac{1-\hat{\varepsilon}}{1+\hat{\varepsilon}} h_v$ whenever $H_v^{max}(t) > H_v(t)$. Moreover, at any time t when $L_v(t) = H_v^{\max}(t)$, the logical clock rate is the same as the rate of progress of $H_v^{\max}$, which ensures that the logical clock value is always at most H_v^{max}. Hence, the logical clock values are always upper bounded by the largest hardware clock value in the network. As each node increases its logical clock at the normal rate when $L_v(t) = H_v(t)$, we also have that $L_v(t) \geq \min_{w \in V}\{H_w(t)\}$. The minimum and the maximum logical clock rates are $\alpha = \frac{1-\hat{\varepsilon}}{1+\hat{\varepsilon}}(1-\varepsilon) \in 1 - \mathcal{O}(\varepsilon)$ and $\beta = (1+\varepsilon)\frac{1-\hat{\varepsilon}}{1+\hat{\varepsilon}}(1+\mu) < 1+\mu$. The parameters μ and κ can again be adapted for these progress rates, and the bounds on the clock skews remain basically unaltered.

Finally, we briefly discuss how the algorithm can be adapted in order to cope with node or edge failures. Since the algorithm cannot distinguish between node and edge failures, it suffices to consider edge failures. We assume for the sake of simplicity that a failed edge does not reappear. The proposed strategy relies on the observation that the algorithm can dynamically determine an estimate $\hat{\mathcal{T}} \in \mathcal{O}(\mathcal{T})$ of the maximum delay. Every node v stores the times when it last received a message from each of its neighbors. It is possible that a message from a neighbor $w \in \mathcal{N}_v$ is "overdue" according to v's hardware clock, i.e., more than $(1+\hat{\varepsilon})(\hat{\mathcal{T}} + H_0/(1-\hat{\varepsilon}))$ time has passed since v received a message from w. This situation can occur for one of two reasons, either w crashed or the delay is larger than estimated. In both cases, v (temporarily) removes w from its set of neighbors and adjusts its logical clock rate if necessary. If v receives the next message from w at a later point in time, w is added to the set $\mathcal{N}_v$ again, $\hat{\mathcal{T}}$ and κ are updated, and the logical clock rate is reevaluated. Subsequently, the new estimate of the maximum delay is forwarded to all neighbors. The exclusion of w may mistakenly cause v to increase its logical clock rate because w was the only neighbor that prevented v from setting its logical clock rate multiplier to $1+\mu$. However, if v considers the edge $\{v, w\}$ to have failed for ΔT time, it increases its logical clock value by at most $\Delta L = (1+\varepsilon)(1+\mu)\Delta T - (1-\varepsilon)\Delta T = 2\varepsilon\Delta T + (1+\varepsilon)\mu\Delta T$ more than the minimum progress in this time interval, i.e., the clock skew to any neighbor grows by at most ΔL. The bound on the local skew relies on the fact that the average clock skew of a path of a given length can only be a specific multiple of κ. Consider any path containing v with a certain average clock skew $\Lambda \geq \kappa$. Since $\kappa = 2((1+\mu)\mathcal{T} + \bar{H}_0)$, the new estimate of $\mathcal{T}$ causes κ to increase by $\Delta\kappa = 2(1+\mu)\Delta T > (2\varepsilon + (1+\varepsilon)\mu)\Delta T = \Delta L$, which implies that

$$\frac{\Lambda + \Delta L}{\kappa + \Delta\kappa} < \frac{\Lambda}{\kappa}.$$

Thus, the average clock skew is at most the same multiple of κ with respect to the new value of κ. The same argument holds if v no longer increases its logical clock at the rate $1+\mu$ because the node u that maximizes $\Lambda_v^{\uparrow}$ does not respond within the expected amount of time. We conclude that temporarily excluding nodes from the set of neighbors because they do not respond in time does not have an impact on the clock skew bounds with respect to the increased κ, and thus algorithm $\mathcal{A}^{opt}$ can also be used in an environment where edges (or nodes) fail. Apparently, this simple strategy cannot cope with temporary failures since a long-term malfunction of a communication link would cause the nodes to set $\hat{\mathcal{T}}$ to an exceedingly large value. This problem can be tackled if the nodes have some means to detect that they, or the links between them, must have been inoperative so that their recurrence does not cause the nodes to increase their estimate of the maximum delay.

In conclusion, we see that the proposed synchronization techniques are also useful for a wide range of other models as basically the same bounds on the clock skews are guaranteed. It remains to show that these bounds are asymptotically optimal, i.e., there is no algorithm that achieves an asymptotically better bound on either the global or the local skew.

Chapter 9

Lower Bounds

THE lower bounds on both the global and the local skew are proved using *indistinguishability type arguments*. Concretely, we construct *indistinguishable executions* for any given synchronization algorithm $\mathcal{A}$ and any graph G in such a way that at least one of the executions causes large clock skews. Given two executions $\mathcal{E}$ and $\bar{\mathcal{E}}$ of an algorithm $\mathcal{A}$ on a graph G, let $H_v^{\mathcal{E}}(t)$ and $H_v^{\bar{\mathcal{E}}}(t)$ denote the hardware clock values of v at time t in $\mathcal{E}$ and $\bar{\mathcal{E}}$, respectively. The respective logical clock values are denoted by $L_v^{\mathcal{E}}(t)$ and $L_v^{\bar{\mathcal{E}}}(t)$. The following definition formalizes the concept of indistinguishable executions.

Definition 9.1 (Indistinguishability of Executions)**.** *We call $\mathcal{E}$ and $\bar{\mathcal{E}}$* indistinguishable at node v until hardware clock time H, *if v observes the same message pattern with respect to its local time H_v in both $\mathcal{E}$ and $\bar{\mathcal{E}}$ until its hardware clock reaches H. More precisely, if v receives a message at a time t_r when $H_v^{\mathcal{E}}(t_r) \leq H$ in $\mathcal{E}$, it receives an identical message in $\bar{\mathcal{E}}$ at the time $\bar{t}_r$ when $H_v^{\bar{\mathcal{E}}}(\bar{t}_r) = H_v^{\mathcal{E}}(t_r)$, and vice versa. Note that in this situation v behaves the same way in $\mathcal{E}$ and $\bar{\mathcal{E}}$ until local time H, i.e., if $H_v^{\mathcal{E}}(t) = H_v^{\bar{\mathcal{E}}}(\bar{t}) \leq H$, it follows that $L_v^{\mathcal{E}}(t) = L_v^{\bar{\mathcal{E}}}(\bar{t})$ and v sends the same set of messages at times t and $\bar{t}$ in $\mathcal{E}$ and $\bar{\mathcal{E}}$, respectively.*

Throughout this chapter, the construction of indistinguishable executions $\mathcal{E}$ and $\bar{\mathcal{E}}$ is based on a simple principle called *shifting* where both the clock rates and the message delays are "shifted" in such a way that all events occur at the same local times in $\mathcal{E}$ and $\bar{\mathcal{E}}$ [37]. If it is clear from the context, we may omit the specification of the execution in the notation and write, e.g., $H_v(t)$ instead of $H_v^{\mathcal{E}}(t)$.

9.1 Global Skew

In the previous chapter, we assumed that an algorithm only possesses estimates of ε and $\mathcal{T}$. If the estimates are inaccurate, κ is set to a value that is larger than necessary, which has a negative impact on the bound on the local skew. The following theorem gives a lower bound on the global skew that depends on the accuracy of both ε and $\mathcal{T}$.

Theorem 9.2. *Assume that a clock synchronization algorithm $\mathcal{A}$ is equipped with initial parameters $\hat{\varepsilon} \in (0,1)$, and $\hat{\mathcal{T}} \in \mathbb{R}^+$ such that $c_1\hat{\mathcal{T}} \leq \mathcal{T} \leq \hat{\mathcal{T}}$ and $c_2\hat{\varepsilon} \leq \varepsilon \leq \hat{\varepsilon}$ for certain values $c_1, c_2 \in (0,1]$. Define that $\varrho := \min\{\varepsilon, (1-c_2\hat{\varepsilon})/c_1 - 1\} \in [-\varepsilon, \varepsilon]$. If algorithm $\mathcal{A}$ is bound to satisfy Condition (6.2), it cannot avoid a global skew of at least*

$$(1+\varrho)D\mathcal{T}$$

on any graph G of diameter D.

Proof. For the sake of simplicity, we formally allow relative clock drifts of $\varepsilon + \delta\varepsilon$, where $\delta\varepsilon$ is infinitesimally small.[1]

Let $v_0, v_D \in V$ be any two nodes at distance D. Furthermore, define that $\mathcal{T} := c_1\hat{\mathcal{T}}$, $\varepsilon' := c_2\hat{\varepsilon}$, and $\mathcal{T}' := \frac{1+\varrho}{1-\varepsilon'}\mathcal{T}$. Since $\varrho \geq -\varepsilon'$, we have that

$$c_1\hat{\mathcal{T}} = \mathcal{T} \leq \mathcal{T}' \leq \hat{\mathcal{T}}$$
$$c_2\hat{\varepsilon} = \varepsilon' \leq \varepsilon \leq \hat{\varepsilon}.$$

Thus, it is possible that $\mathcal{T}'$ is the real maximum delay and ε' is the real maximum clock drift because both values lie in the legal range according to the definition of c_1 and c_2. Assume that the maximum delay is in fact $\mathcal{T}'$ and the maximum clock drift is ε'. Consider the following two executions:

$\mathcal{E}_1$: The hardware clock rates of all clocks are $1-\varepsilon'$ at all times. The message delays are always $\mathcal{T}'$ from any node $v \in V$ to any node $w \in \mathcal{N}_v$ if $d(v_0, w) = d(v_0, v) - 1$, and 0 otherwise.

$\mathcal{E}_2$: The hardware clock rates of all clocks are $1+\varepsilon'$ at all times. The message delays are $\frac{(1-\varepsilon')}{1+\varepsilon'}\mathcal{T}'$ from node $v \in V$ to node $w \in \mathcal{N}_v$ if $d(v_0, w) = d(v_0, v) - 1$, and 0 otherwise.

Execution $\mathcal{E}_1$ and $\mathcal{E}_2$ are obviously legal executions as both the message delays and the clock drifts are within the legal bounds. Furthermore, $\mathcal{E}_1$ and $\mathcal{E}_2$ are indistinguishable: In execution $\mathcal{E}_1$, if a node v sends a message to w at local time H_v, w receives this message at a time t when $H_w(t) = H_v + (1-\varepsilon')\mathcal{T}'$ if $d(v_0, w) = d(v_0, v) - 1$ and $H_w(t) = H_v$ otherwise. Since the clock rates are faster by a factor of $(1+\varepsilon')/(1-\varepsilon')$ and the message delay of any message that is sent to a node that is closer to v_0 is reduced by the same factor, the nodes receive and send the same messages at the same hardware clock times in execution $\mathcal{E}_2$.

[1] The same result could be obtained, e.g., by replacing ε by $\varepsilon - \delta\varepsilon$ and proving a bound of $(1+\varrho-\mathcal{O}(\delta\varepsilon))D\mathcal{T}$.

No node v can increase its logical clock at a rate lower than its hardware clock rate, as otherwise it violates the linear envelope condition in execution $\mathcal{E}_1$. Likewise, v cannot increase its logical clock faster than its hardware clock because Condition (6.2) would be violated in execution $\mathcal{E}_2$. Thus, in both executions it must hold that $L_v(t) = H_v(t)$ at all times t.

Assume now that $\mathcal{T}$ and ε are the correct upper bounds on the maximum delay and the maximum clock drift, respectively. Consider the following execution:

$\mathcal{E}_3$: The hardware clock rate of $v \in V$ is $1+\varrho+\frac{D-d(v_0,v)}{D}\delta\varepsilon$, where $0 < \delta\varepsilon \ll |\varrho|$ is infinitesimally small. At time $t_0 := \frac{(1+\varrho)D\mathcal{T}}{\delta\varepsilon}$ all hardware clock rates are switched to $1+\varrho$. If a node v sends a message at hardware clock time H_v, the message delay is adjusted in such a way that it is received at time t when $H_w(t) = H_v+(1-\varepsilon')\mathcal{T}'$ if $d(v_0,w) = d(v_0,v)-1$ and $H_w(t) = H_v$ otherwise.

Note that execution $\mathcal{E}_1$ and $\mathcal{E}_3$, and hence also $\mathcal{E}_2$ and $\mathcal{E}_3$, are indistinguishable at each node $v \in V$ by construction. It remains to verify that $\mathcal{E}_3$ is a legal execution. Since $\varrho \in [-\varepsilon, \varepsilon]$ and a clock drift of $\varepsilon + \delta\varepsilon$ is allowed, the clock drifts of all clocks are in the legal range. As far as the message delays are concerned, we have at all times $t \leq t_0$ that $H_w(t) - H_v(t) = \frac{\delta\varepsilon}{D}t \in [0, (1+\varrho)\mathcal{T}] = [0, (1-\varepsilon')\mathcal{T}']$ if $d(v_0,w) = d(v_0,v)-1$. First, consider a message sent from v to w. If $H_w(t) - H_v(t) = (1-\varepsilon')\mathcal{T}'$, then the message delay is set to zero, which ensures that w "sees" exactly a difference of $(1-\varepsilon')\mathcal{T}'$. If $H_w(t) - H_v(t) = 0$, the message must be delayed. However, since the hardware clock rate of each node is at least $1+\varrho$, it takes at most $\frac{(1-\varepsilon')\mathcal{T}'}{1+\varrho} = \mathcal{T}$ time for w to reach the hardware clock value $H_v + (1-\varepsilon')\mathcal{T}'$. Thus in case of $d(v_0,w) = d(v_0,v)-1$ the message delays are always in the range $[0,\mathcal{T}]$. If w sends a message to v, the same arguments apply, but in this case we need that the message delay is set to zero if $H_w(t) - H_v(t) = 0$ and at most $\frac{(1-\varepsilon')\mathcal{T}'}{1+\varrho} = \mathcal{T}$ if $H_w(t) - H_v(t) = (1-\varepsilon')\mathcal{T}'$. Note that if $d(v_0,w) = d(v_0,v)$, then $H_v(t) = H_w(t)$ as in the other two executions, and the message delay remains zero. Finally, the message delays remain in the range $[0,\mathcal{T}]$ at any time $t > t_0$, because all clocks run at the same rate, i.e., the differences between the hardware clock values do not change.

Since the nodes cannot distinguish between any of the three executions, it follows that $L_v(t) = H_v(t)$ for all nodes $v \in V$ at all times t also in $\mathcal{E}_3$. The skew between the clocks L_{v_0} and L_{v_D} in execution $\mathcal{E}_3$ at any time $t \geq t_0$ is

$$t_0\frac{d(v_0,v_D) - d(v_D,v_D)}{D}\delta\varepsilon = (1+\varrho)D\mathcal{T},$$

which proves the stated lower bound on the global skew of any algorithm $\mathcal{A}$ that satisfies Condition (6.2). □

Note that also a randomized algorithm must increase the logical and the hardware clock at the same rate in these executions, implying that randomization does not help to reduce the global skew.

We can conclude from this theorem that the estimates of $\mathcal{T}$ and ε must be extremely accurate in order guarantee a better bound than $(1+\varepsilon)D\mathcal{T}$. However, even if the exact values are known, a global skew of $(1-\varepsilon)D\mathcal{T}$ cannot be prevented subject to the condition that the logical clock values must be within a linear envelope of real time.

Corollary 9.3. *Consider clock synchronization algorithms that satisfy Condition* (6.2)*. No such algorithm without knowledge of a lower bound on ε can avoid a global skew of $D\mathcal{T}$. Furthermore, no such algorithm without knowledge of bounds on $\mathcal{T}$ stronger than $\mathcal{T} \in \left[\frac{1-\varepsilon}{1+\varepsilon}\hat{\mathcal{T}}, \hat{\mathcal{T}}\right]$ can achieve a better bound on the global skew than $(1+\varepsilon)D\mathcal{T}$.*

This corollary implies that any algorithm in the family $\mathfrak{A}$, and in particular $\mathcal{A}^{opt}$, is essentially optimal as far as the global skew is concerned. What is more, it can be shown that a global skew of $D\mathcal{T}/2$ cannot be prevented even if the restriction that the algorithm must satisfy Condition (6.2) is dropped [8]. Thus, the bound on the global skew of $\mathcal{A}^{opt}$ is roughly a factor of two worse than the bound on the global skew of any algorithm whose behavior is not constrained by any additional conditions.

9.2 Local Skew

The proof of the lower bound on the local skew also exploits that specific executions cannot be distinguished. The following lemma, which is a simple variant of the lemma presented in the original work that introduced the problem of bounding the clock skews between neighboring nodes [18], states that for a specific execution $\mathcal{E}$ there is an indistinguishable execution $\bar{\mathcal{E}}$ such that the skew between two nodes v and w is larger in $\bar{\mathcal{E}}$ than in $\mathcal{E}$ at some point in time.

Lemma 9.4. *Consider any clock synchronization algorithm $\mathcal{A}$ executed on any graph $G=(V,E)$ and two nodes v and w. If the hardware clock rates of all nodes are always* 1 *and the message delays are $\mathcal{T}/2$ in an execution $\mathcal{E}$ that starts at a time t_0 and ends at time $t_{\mathcal{E}} := t_0 + \frac{1+\varepsilon/2}{\varepsilon}d(v,w)\mathcal{T}$, there is an execution $\bar{\mathcal{E}}$ that starts at time t_0 and ends at time $t_{\bar{\mathcal{E}}} := t_0 + \frac{1}{\varepsilon}d(v,w)\mathcal{T}$ such that $L_v^{\mathcal{E}}(t_{\mathcal{E}}) = L_v^{\bar{\mathcal{E}}}(t_{\bar{\mathcal{E}}})$ and $L_w^{\mathcal{E}}(t_{\bar{\mathcal{E}}}) = L_w^{\bar{\mathcal{E}}}(t_{\bar{\mathcal{E}}})$.*

Proof. Execution $\bar{\mathcal{E}}$ is defined as follows. The hardware clock rate of any node u at all times $t \in [t_0, t_{\bar{\mathcal{E}}}]$ is given by:

$$h_u = h_u(t) := \begin{cases} 1 + \frac{\varepsilon}{2}\left(1 - \frac{d(v,u)}{d(v,w)}\right) & \text{if } d(v,u) \leq d(v,w) \\ 1 & \text{else} \end{cases}$$

All message delays are adjusted in such a way that for each send or receive event of any node u at a time t in execution $\mathcal{E}$ the same event occurs in execution $\bar{\mathcal{E}}$ at the time $\bar{t} \leq t_{\bar{\mathcal{E}}}$ for which it holds that $H_u^{\mathcal{E}}(t) = H_u^{\bar{\mathcal{E}}}(\bar{t})$, i.e., execution $\mathcal{E}$ and $\bar{\mathcal{E}}$ are indistinguishable at any node u until its hardware clock value reaches $H_u^{\bar{\mathcal{E}}}(t_{\bar{\mathcal{E}}})$ by construction. Since the hardware clock rate of v is $1 + \varepsilon/2$ and $\bar{\mathcal{E}}$ is a factor of $1 + \varepsilon/2$ shorter than $\mathcal{E}$, node v reaches the same hardware clock value at the end of both executions, i.e., $H_v^{\mathcal{E}}(t_{\mathcal{E}}) = H_v^{\bar{\mathcal{E}}}(t_{\bar{\mathcal{E}}})$. Moreover, w's hardware clock always runs at the rate 1 in both executions, which implies that $H_v^{\mathcal{E}}(t_{\bar{\mathcal{E}}}) = H_v^{\bar{\mathcal{E}}}(t_{\bar{\mathcal{E}}})$. Given that the executions are indistinguishable, it follows that $L_v^{\mathcal{E}}(t_{\mathcal{E}}) = L_v^{\bar{\mathcal{E}}}(t_{\bar{\mathcal{E}}})$ and $L_v^{\mathcal{E}}(t_{\bar{\mathcal{E}}}) = L_v^{\bar{\mathcal{E}}}(t_{\bar{\mathcal{E}}})$ as desired.

It has to be verified that $\bar{\mathcal{E}}$ is in fact a valid execution in the sense that all clock rates and message delays are within the legal bounds. The clock rates lie in the interval $[1, 1+\varepsilon/2] \subset [1-\varepsilon, 1+\varepsilon]$ and are thus in the legal range. As far as the message delays are concerned, consider any message sent from a node u_s at time t_s that node u_r received at time $t_r = t_s + \mathcal{T}/2$ in execution $\mathcal{E}$. For the corresponding send and receive events at times $\bar{t}_s$ and $\bar{t}_r \leq t_{\bar{\mathcal{E}}}$ in execution $\bar{\mathcal{E}}$ it holds that

$$\begin{aligned} t_s - t_0 &= h_{u_s} \cdot (\bar{t}_s - t_0) \\ t_r - t_0 &= h_{u_r} \cdot (\bar{t}_r - t_0). \end{aligned}$$

The message delay in execution $\bar{\mathcal{E}}$ is

$$\begin{aligned} \bar{t}_r - \bar{t}_s &= (\bar{t}_r - t_0) - (\bar{t}_s - t_0) \\ &= \frac{t_r - t_0}{h_{u_r}} - \frac{t_s - t_0}{h_{u_s}} \\ &= \frac{h_{u_s}(t_r - t_0) - h_{u_r}(t_s - t_0)}{h_{u_s} h_{u_r}} \\ &= \frac{h_{u_s}(t_r - t_0) - h_{u_r}(t_r - \mathcal{T}/2 - t_0)}{h_{u_s} h_{u_r}} \\ &= \frac{\mathcal{T}}{2h_{u_s}} + \frac{(h_{u_s} - h_{u_r})(\bar{t}_r - t_0)}{h_{u_s}}. \end{aligned} \tag{9.1}$$

If either $d(v, u_s) = d(v, u_r)$ or both the distance from u_s to v and the distance from u_r to v are at least $d(v, w)$, the hardware clocks rates of u_s and u_r are equal, i.e., $h_{u_s} = h_{u_r}$. In this case, the message delay $\bar{t}_r - \bar{t}_s$ in $\bar{\mathcal{E}}$ is $\mathcal{T}/(2h_{u_s}) \in (\mathcal{T}/(2(1+\varepsilon/2)), \mathcal{T}/2] \subset [0, \mathcal{T}]$.

If u_s is closer to v and $d(v, u_s) < d(v, w)$, it holds that $h_{u_s} - h_{u_r} = \varepsilon/(2d(v, w))$.

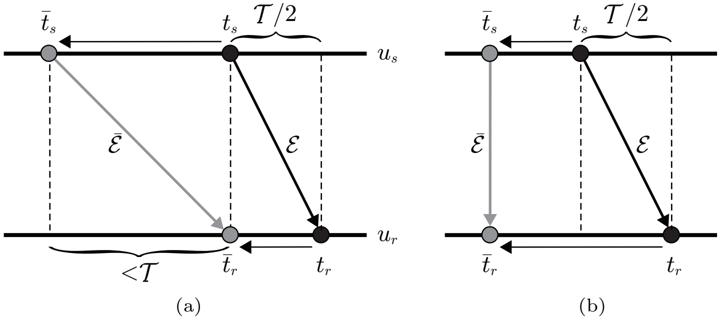

Figure 9.1: In Figure (a), we have that $d(v, u_s) < d(v, u_r) < d(v, w)$, which implies that $h_{u_s} > h_{u_r} > 1$. Since $h_{u_s} > h_{u_r}$, the message delay increases from $\mathcal{T}/2$ in execution $\mathcal{E}$ to almost $\mathcal{T}$ towards the end of execution $\bar{\mathcal{E}}$. If $d(v, u_r) < d(v, u_s) < d(v, w)$, the message delay reduces to close to zero, as depicted in Figure (b). Note that the message delay can only be exactly zero in execution $\bar{\mathcal{E}}$ if $d(v, u_s) = d(v, w)$, i.e., $h_{u_s} = 1$.

As $\bar{t}_r \le t_{\bar{\mathcal{E}}}$, we get that

$$\begin{aligned} 0 \overset{(9.1)}{<} \bar{t}_r - \bar{t}_s &\overset{(9.1)}{=} \frac{\mathcal{T}}{2h_{u_s}} + \frac{\frac{\varepsilon}{2d(v,w)}(\bar{t}_r - t_0)}{h_{u_s}} \\ &\le \frac{\mathcal{T}}{2h_{u_s}} + \frac{\frac{\varepsilon}{2d(v,w)}(t_{\bar{\mathcal{E}}} - t_0)}{h_{u_s}} \\ &= \frac{\mathcal{T}}{2h_{u_s}} + \frac{\mathcal{T}}{2h_{u_s}} < \mathcal{T}. \end{aligned}$$

Analogously, if u_r is closer to v, we have that $h_{u_s} - h_{u_r} = -\varepsilon/(2d(v, w))$ and thus

$$\mathcal{T} \overset{(9.1)}{>} \bar{t}_r - \bar{t}_s \overset{(9.1)}{\ge} \frac{\mathcal{T}}{2h_{u_s}} - \frac{\frac{\varepsilon}{2d(v,w)}(t_{\bar{\mathcal{E}}} - t_0)}{h_{u_s}} = \frac{\mathcal{T}}{2h_{u_s}} - \frac{\mathcal{T}}{2h_{u_s}} = 0.$$

Figure 9.1 illustrates how $d(v, u_s)$ and $d(v, u_r)$ affect the message delays in execution $\bar{\mathcal{E}}$ if $d(v, u_s) < d(v, w)$, $d(v, u_r) < d(v, w)$, and $d(v, u_s) \neq d(v, u_r)$. Since the message delays are always in the legal range $[0, \mathcal{T}]$, $\bar{\mathcal{E}}$ is a legal execution, which concludes the proof. □

The concept of an *extended execution* will be useful in the proof of the theorem.

Definition 9.5 (Extended Executions)**.** *Given an execution $\mathcal{E}$ running from time t_1 to t_2, $\mathcal{E}$ can be extended by specifying hardware clock rates and message delays in a time interval $[t_2, t_3]$. We will refer to this extension as an* *execution $\mathcal{E}'$ running from time t_2 until time t_3. Note that $\mathcal{E}'$ inherits the state of the system at time t_2, i.e., the state of all nodes and any pending messages sent in $\mathcal{E}$ that did not reach their destination until time t_2.*

The theorem states that, for any algorithm $\mathcal{A}$ executed on any graph G, the local skew is lower bounded by $\Omega\left(\mathcal{T}\log_b D\right)$ where the base b depends on α, β, and ε. Recall that α and β denote the minimum and the maximum logical clock rate, respectively.

Theorem 9.6. *Define $b := \left\lceil \frac{4(\beta-\alpha)(1+\varepsilon/2)}{\alpha\varepsilon} \right\rceil$. No clock synchronization algorithm $\mathcal{A}$ can prevent a local skew of*

$$\frac{\lfloor \log_b D \rfloor + 2}{4}\alpha\mathcal{T} \in \Omega\left(\alpha\mathcal{T}\left(1 + \log_{(\beta-\alpha)/(\alpha\varepsilon)} D\right)\right)$$

on any graph G of diameter D.

Proof. Define $D' := b^{\lfloor \log_b D \rfloor} \leq D$ and let $t_{\mathcal{E}}$ denote the time when an execution $\mathcal{E}$ ends. We claim that for any k, where $0 \leq k \leq \log_b D'$, there are two nodes v_k and w_k at distance $d(v_k, w_k) = D'/b^k$ such that the clock skew between these nodes at the end of an execution $\bar{\mathcal{E}}_k$ is

$$L_{v_k}^{\bar{\mathcal{E}}_k}(t_{\bar{\mathcal{E}}_k}) - L_{w_k}^{\bar{\mathcal{E}}_k}(t_{\bar{\mathcal{E}}_k}) \geq \frac{k+2}{4}\alpha d(v_k, w_k)\mathcal{T}. \tag{9.2}$$

Consider any two nodes v_0 and w_0 at distance $d(v_0, w_0) = D'$. The execution $\mathcal{E}_0$ starts at time 0 and ends at time $t_{\mathcal{E}_0} := \frac{1+\varepsilon/2}{\varepsilon}D'\mathcal{T}$. All messages delays are $\mathcal{T}/2$ and the hardware clock rates are 1. Without loss of generality, assume that $L_{v_0}^{\mathcal{E}_0}(t_{\mathcal{E}_0}) \geq L_{w_0}^{\mathcal{E}_0}(t_{\mathcal{E}_0})$. According to Lemma 9.4, there is an execution $\bar{\mathcal{E}}_0$ that ends at time $t_{\bar{\mathcal{E}}_0} = \frac{1}{\varepsilon}d(v_0, w_0)\mathcal{T}$ for which it holds that $L_{v_0}^{\mathcal{E}_0}(t_{\mathcal{E}_0}) = L_{v_0}^{\bar{\mathcal{E}}_0}(t_{\bar{\mathcal{E}}_0})$ and $L_{w_0}^{\mathcal{E}}(t_{\bar{\mathcal{E}}_0}) = L_{w_0}^{\bar{\mathcal{E}}_0}(t_{\bar{\mathcal{E}}_0})$. Since the minimum clock rate is α, we have that $L_{w_0}^{\mathcal{E}_0}(t_{\mathcal{E}_0}) - L_{w_0}^{\mathcal{E}_0}(t_{\bar{\mathcal{E}}_0}) \geq \frac{\alpha}{2}d(v_0, w_0)\mathcal{T}$. The clock skew between v_0 and w_0 at the end of execution $\bar{\mathcal{E}}_0$ is thus at least

$$\begin{aligned}
L_{v_0}^{\bar{\mathcal{E}}_0}(t_{\bar{\mathcal{E}}_0}) - L_{w_0}^{\bar{\mathcal{E}}_0}(t_{\bar{\mathcal{E}}_0}) \quad &= \quad L_{v_0}^{\mathcal{E}_0}(t_{\mathcal{E}_0}) - L_{w_0}^{\mathcal{E}_0}(t_{\bar{\mathcal{E}}_0}) \\
&\geq \quad L_{v_0}^{\mathcal{E}_0}(t_{\mathcal{E}_0}) - L_{w_0}^{\mathcal{E}_0}(t_{\mathcal{E}_0}) + \frac{\alpha}{2}d(v_0, w_0)\mathcal{T} \\
&\geq \quad \frac{\alpha}{2}d(v_0, w_0)\mathcal{T},
\end{aligned}$$

which proves the claim for $k = 0$.

Assume that the claim is true for k, where $k < \log_b D'$. Given such an execution $\bar{\mathcal{E}}_k$, we extend it by an execution $\mathcal{E}^{k+1}$ that starts at time $t_{\bar{\mathcal{E}}_k}$ and ends at time $t_{\mathcal{E}^{k+1}} := t_{\bar{\mathcal{E}}_k} + \frac{1+\varepsilon/2}{\varepsilon}\frac{D'}{b^{k+1}}\mathcal{T}$. Note that $\mathcal{E}^{k+1}$ inherits all the messages that were still in transit at the end of execution $\bar{\mathcal{E}}_k$. We simply define that all these messages arrive when execution $\mathcal{E}^{k+1}$ starts, i.e., at time $t_{\bar{\mathcal{E}}_k}$. The hardware clock rate of each node is 1 and the message delays are always $\mathcal{T}/2$ in execution $\mathcal{E}^{k+1}$. The clock skew at time $t_{\mathcal{E}^{k+1}}$ between the nodes v_k and w_k for which Inequality (9.2) holds at time $t_{\bar{\mathcal{E}}_k}$ is at least

$$\begin{aligned}
L_{v_k}^{\mathcal{E}^{k+1}}(t_{\mathcal{E}^{k+1}}) - L_{w_k}^{\mathcal{E}^{k+1}}(t_{\mathcal{E}^{k+1}}) &\geq (L_{v_k}^{\bar{\mathcal{E}}_k}(t_{\bar{\mathcal{E}}_k}) + \alpha(t_{\mathcal{E}^{k+1}} - t_{\bar{\mathcal{E}}_k})) \\
&\quad -(L_{w_k}^{\bar{\mathcal{E}}_k}(t_{\bar{\mathcal{E}}_k}) + \beta(t_{\mathcal{E}^{k+1}} - t_{\bar{\mathcal{E}}_k})) \\
&\overset{(9.2)}{\geq} \frac{k+2}{4}\alpha d(v_k, w_k)\mathcal{T} \\
&\quad -(\beta-\alpha)\frac{1+\varepsilon/2}{\varepsilon}\frac{D'}{b^{k+1}}\mathcal{T} \\
&= \frac{k+2}{4}\alpha d(v_k, w_k)\mathcal{T} \\
&\quad -(\beta-\alpha)\frac{1+\varepsilon/2}{\varepsilon}\frac{d(v_k, w_k)}{b}\mathcal{T} \\
&\geq \frac{k+1}{4}\alpha d(v_k, w_k)\mathcal{T}.
\end{aligned}$$

Consequently, there must be two nodes v_{k+1} and w_{k+1} at distance $d(v_{k+1}, w_{k+1}) = d(v_k, w_k)/b = D'/b^{k+1}$ for which it holds that

$$L_{v_{k+1}}^{\mathcal{E}^{k+1}}(t_{\mathcal{E}^{k+1}}) - L_{w_{k+1}}^{\mathcal{E}^{k+1}}(t_{\mathcal{E}^{k+1}}) \geq \frac{k+1}{4}\alpha d(v_{k+1}, w_{k+1})\mathcal{T}. \qquad (9.3)$$

We can apply Lemma 9.4 again for execution $\mathcal{E}^{k+1}$ and get an execution $\bar{\mathcal{E}}^{k+1}$ of duration $\frac{1}{\varepsilon}d(v_{k+1}, w_{k+1})\mathcal{T}$ for which it holds that $L_{v_{k+1}}^{\mathcal{E}^{k+1}}(t_{\mathcal{E}^{k+1}}) = L_{v_{k+1}}^{\bar{\mathcal{E}}^{k+1}}(t_{\bar{\mathcal{E}}^{k+1}})$ and $L_{w_{k+1}}^{\mathcal{E}^{k+1}}(t_{\bar{\mathcal{E}}^{k+1}}) = L_{w_{k+1}}^{\bar{\mathcal{E}}^{k+1}}(t_{\bar{\mathcal{E}}^{k+1}})$. Let $\bar{\mathcal{E}}_{k+1}$ be the execution $\bar{\mathcal{E}}_k$ extended by $\bar{\mathcal{E}}^{k+1}$. Since it holds that $L_{w_{k+1}}^{\mathcal{E}^{k+1}}(t_{\mathcal{E}^{k+1}}) - L_{w_{k+1}}^{\mathcal{E}^{k+1}}(t_{\bar{\mathcal{E}}_{k+1}}) \geq \frac{\alpha}{2}d(v_{k+1}, w_{k+1})\mathcal{T}$, the clock skew at time $t_{\bar{\mathcal{E}}_{k+1}} = t_{\bar{\mathcal{E}}^{k+1}}$ between v_{k+1} and w_{k+1} in execution $\bar{\mathcal{E}}_{k+1}$ is

$$\begin{aligned}
L_{v_{k+1}}^{\bar{\mathcal{E}}_{k+1}}(t_{\bar{\mathcal{E}}_{k+1}}) - L_{w_{k+1}}^{\bar{\mathcal{E}}_{k+1}}(t_{\bar{\mathcal{E}}_{k+1}}) &= L_{v_{k+1}}^{\mathcal{E}^{k+1}}(t_{\mathcal{E}^{k+1}}) - L_{w_{k+1}}^{\mathcal{E}^{k+1}}(t_{\bar{\mathcal{E}}_{k+1}}) \\
&\geq L_{v_{k+1}}^{\mathcal{E}^{k+1}}(t_{\mathcal{E}^{k+1}}) - L_{w_{k+1}}^{\mathcal{E}^{k+1}}(t_{\mathcal{E}^{k+1}}) \\
&\quad +\frac{\alpha}{2}d(v_{k+1}, w_{k+1})\mathcal{T} \\
&\overset{(9.3)}{\geq} \frac{(k+1)+2}{4}\alpha d(v_{k+1}, w_{k+1})\mathcal{T},
\end{aligned}$$

which proves the claim. If $k = \lfloor \log_b D \rfloor$, the distance between the considered nodes is 1, i.e., v_k and w_k are neighboring nodes. The local skew is thus at least $\frac{\lfloor \log_b D \rfloor + 2}{4}\alpha\mathcal{T}$. □

Note that randomization again does not help as we are only concerned with the *existence* of executions that cause a large local skew. In other words, given a randomized algorithm $\mathcal{A}$, an adversary may not be able to construct an execution that results in a large clock skew between some neighboring nodes, but an execution of $\mathcal{A}$ on any graph G still exists such that the local skew reaches $\frac{\lfloor \log_b D \rfloor + 2}{4}\alpha\mathcal{T}$. In that sense, randomized algorithms do not have an advantage over deterministic algorithms.

It is further important to see that the maximum clock skew among all neighboring nodes can be $\Omega(\mathcal{T} \log_b D)$ for more than a constant period of time. The proof of Theorem 9.6 reveals that, e.g. for $k = \frac{1}{2}\log_b D'$, the average clock skew on a path of length $\Theta(\sqrt{D})$ is half the local skew that is built up between two neighbors until the end of the constructed execution. Since it takes $\Theta(\sqrt{D}\mathcal{T})$ time to increase the clock skew to $\frac{\lfloor \log_b D \rfloor + 2}{4}\alpha\mathcal{T}$, the maximum clock skew among all neighboring nodes is $\Omega(\mathcal{T} \log_b D)$ for $\Theta(\sqrt{D}\mathcal{T})$ time. More generally, for any constant $c < 1$, the average clock skew on some path, and thus also the maximum clock skew among all neighboring nodes, is $\Omega(\frac{1}{c}\log_b D)$ for $\Theta(D^{1-c}\mathcal{T})$ time. Furthermore, it is evident from the proof that Theorem 9.6 still holds if the nodes are allowed to increase their logical clock values instantaneously, but the maximum *average* logical clock rate over any time interval of length $\Omega(\mathcal{T}/\varepsilon)$ is upper bounded by β.

If we demand that the logical clocks run roughly at the same rates as the hardware clocks, e.g., $\alpha \in 1 - \mathcal{O}(\varepsilon)$ and $\beta \in 1 + \mathcal{O}(\varepsilon)$, we get that $b \in \mathcal{O}(1)$ and thus a lower bound of $\Omega(\mathcal{T} \log D)$, which matches the upper bound of algorithm $\mathcal{A}^{opt}$ when $\mu \in \Theta(\varepsilon)$ and $H_0 \in \mathcal{O}(\mathcal{T}/\varepsilon)$. Similarly, if we allow a logical clock rate that is a constant times larger than real time, i.e., $\beta \in \Theta(1)$, the lower bound reduces to $\Omega(\mathcal{T} \log_{1/\varepsilon} D)$. Algorithm $\mathcal{A}^{opt}$ guarantees an upper bound on the local skew of $\mathcal{O}(\mathcal{T} \log_{1/\varepsilon} D)$ when choosing $\mu \in \Theta(1)$ and $H_0 \in \mathcal{O}(\mathcal{T})$. In both cases, we see that $\mathcal{A}^{opt}$ is asymptotically optimal. One might wonder if a better asymptotical bound on the local skew can be achieved if there is no constraint on the maximum logical clock rate. Intuitively, an unbounded maximum clock rate cannot help much, because the clock skew increases more quickly at higher rates [18]. In fact, it can be shown that the lower bound is still $\Omega(\mathcal{T} \log_{1/\varepsilon} D)$ [35], which implies that there is no benefit if the algorithm is allowed to increase the logical clock values arbitrarily fast.

Thus, algorithm $\mathcal{A}^{opt}$ guarantees upper bounds on both the global and the local skew that are asymptotically optimal for any reasonable choice of parameters. Moreover, the bound on the global skew is basically *tight* if Condition (6.2) must be satisfied. These results and the observation that the same asymptotic bounds hold for several other clock synchronization models, as discussed in Section 8.2.3, suggest that the techniques used by algorithm $\mathcal{A}^{opt}$ to bound the (worst-case) clock skews between the nodes of a distributed system are essentially optimal. This concludes our discussion of clock synchronization.

Chapter 10

Conclusion

DUE to the broad range of applicability of distributed systems, there is an unprecedented and growing interest in the complexity of distributed applications. In this thesis, the complexity of two fundamental problems in distributed systems, computing aggregate functions and synchronizing clocks in a distributed manner, have been studied.

In the first part of this thesis, we assumed that a set of elements is scattered arbitrarily among the devices of the distributed system, and the goal is to compute the result of an aggregate function when applied to this set of elements. While the results of aggregate functions that are either *distributive* or *algebraic* can be computed efficiently in a distributed manner by means of a simple convergecast, more sophisticated algorithms are required for *holistic* aggregate functions. Therefore, we focused mainly on holistic functions and gained new insights into the complexity of computing a few well-known holistic functions distributively. In particular, we proved that the k^{th} smallest element indeed cannot be computed as efficiently as aggregate functions that are not holistic, even if the algorithm is allowed to use randomization. On the other hand, we showed that the k^{th} smallest element can still be found fairly efficiently by providing both a randomized algorithm, whose time complexity matches the proved lower bound, and a deterministic k-selection algorithm that achieves a substantially better bound on the time complexity than the simple algorithm that collects all elements locally. By contrast, other holistic aggregate functions, such as finding the *mode*, i.e., the most frequent element, cannot be computed efficiently by any deterministic algorithm. What is more, there is no efficient randomized algorithm that merely computes a constant-factor approximation of the frequency of the mode. However, these results only hold in a worst-case scenario. We showed that it is nonetheless possible to compute the mode quickly for distributions of elements that are common in practice.

In general, one might wonder whether the presented techniques are primarily of theoretical interest or whether they may actually have a notable impact on the complexity of computing aggregate functions in real networks and thus prove to be relevant for practical applications. Apparently, a real network may impose several additional constraints that must be considered. In wireless sensor networks, for example, there are various problems such as interference and transient communication links etc. that need to be handled. Nevertheless, we hope that some of our results and techniques may eventually find their way into several application areas, providing aggregation support for, e.g., streaming databases or multi-core architectures.

In the second part of this thesis, we studied several techniques to bound the skew between the clocks of the computational devices in a distributed system. The main result is an algorithm that guarantees that the worst-case skew between any two clocks is a function that depends linearly on the maximum message delay and on the diameter of the network. As far as the devices that share a direct communication channel are concerned, the worst-case clock skew also depends linearly on the maximum message delay, but only logarithmically on the network diameter. In light of the lower bounds on the clock skews presented in Chapter 9, these bounds are asymptotically optimal. Surprisingly, the algorithm achieves these bounds at a low message frequency, even if it is not allowed to change the clock rates substantially in order to compensate for the observed clock skews, i.e., a minute change of the clock rate at the right time suffices to keep the clock skews essentially as small as possible.

The practical relevance of these results are twofold. First, we showed that clock skews can become fairly large if simple strategies, such as averaging the clock values, are used to synchronize the clocks. These results are not only of theoretical interest as algorithms based on such strategies will likely perform poorly in real networks. Second, if the parameters of the proposed algorithm are set to appropriate values, the algorithm achieves small bounds on the clock skews also in real networks, i.e., the asymptotic results do not hide any large constants that would render the algorithm unfit for practical use. Moreover, we illustrated that the presented techniques can be used for various other clock synchronization models and under different constraints. This generality and flexibility further indicates that state-of-the-art clock synchronization protocols could potentially be improved by applying some of the techniques introduced in this thesis.

Bibliography

[1] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. *Journal of Computer Systems and Sciences*, 58(1):137–147, 1999.

[2] H. Attiya, A. Herzberg, and S. Rajsbaum. Optimal Clock Synchronization under Different Delay Assumptions. *SIAM Journal on Computing*, 25(2):369–389, 1996.

[3] Z. Bar-Yossef, T. S. Jayram, R. Kumar, and D. Sivakumar. An Information Statistics Approach to Data Streams and Communication Complexity. *Journal of Computer and System Sciences*, 68(4):702–732, 2004.

[4] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting Distinct Elements in a Data Stream. In *Proc. 6th International Workshop on Randomization and Approximation Techniques (RANDOM)*, pages 1–10, 2002.

[5] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in Streaming Algorithms, with an Application to Counting Triangles in Graphs. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 623–632, 2002.

[6] R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[7] L. Bhuvanagiri, S. Ganguly, D. Kesh, and C. Saha. Simpler Algorithm for Estimating Frequency Moments of Data Streams. In *Proc. 17th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 708–713, 2006.

[8] S. Biaz and J. Lundelius Welch. Closed Form Bounds for Clock Synchronization under Simple Uncertainty Assumptions. *Information Processing Letters*, 80(3):151–157, 2001.

[9] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time Bounds for Selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[10] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. 18th IEEE Conference on Computer Communications (INFOCOM)*, pages 126–134, 1999.

[11] A. Chakrabarti, S. Khot, and X. Sun. Near-Optimal Lower Bounds on the Multi-Party Communication Complexity of Set Disjointness. In *Proc. 18th IEEE Conference on Computational Complexity (CCC)*, pages 107–117, 2003.

[12] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[13] F. Y. L. Chin and H. F. Ting. An Improved Algorithm for Finding the Median Distributively. *Algorithmica*, 2(1):235–249, 1987.

[14] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate Aggregation Techniques for Sensor Databases. In *Proc. 20th International Conference on Data Engineering (ICDE)*, pages 449–460, 2004.

[15] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[16] D. Dolev, J. Y. Halpern, and R. H. Strong. On the Possibility and Impossibility of Achieving Clock Synchronization. *Journal of Computer and System Sciences*, 32(2):230–250, 1986.

[17] M. Durand and P. Flajolet. LogLog Counting of Large Cardinalities. In *Proc. European Symposium on Algorithms (ESA)*, pages 605–617, 2003.

[18] R. Fan and N. Lynch. Gradient Clock Synchronization. *Distributed Computing*, 18(4):255–266, 2006.

[19] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[20] L. R. Ford Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[21] G. N. Frederickson. Tradeoffs for Selection in Distributed Networks. In *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 154–160, 1983.

[22] P. B. Gibbons and S. Tirthapura. Estimating Simple Functions on the Union of Data Streams. In *Proc. 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 281–291, 2001.

[23] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[24] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *Proc. 21st International Conference on Very Large Data Bases (VLDB)*, pages 311–322, 1995.

[25] J. Y. Halpern, , B. Simons, R. Strong, and D. Dolev. Dynamic Fault-Tolerant Clock Synchronization. *Journal of the ACM*, 42(1):143–185, 1995.

[26] J. Y. Halpern, N. Megiddo, and A. A. Munshi. Optimal Precision in the Presence of Uncertainty. *Journal of Complexity*, 1:170–196, 1985.

[27] P. Indyk and D. Woodruff. Optimal Approximations of the Frequency Moments of Data Streams. In *Proc. 37th ACM Symposium on Theory of Computing (STOC)*, pages 202–208, 2005.

[28] B. Kalyanasundaram and G. Schnitger. The Probabilistic Communication Complexity of Set Intersection. *SIAM Journal on Discrete Mathematics*, 5(4):545–557, 1992.

[29] D. Kempe, A. Dobra, and J. Gehrke. Gossip-Based Computation of Aggregate Information. In *Proc. 44th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 482–491, 2003.

[30] F. Kuhn, T. Locher, and S. Schmid. Distributed Computation of the Mode. In *Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 15–24, 2008.

[31] F. Kuhn, T. Locher, and R. Wattenhofer. Tight Bounds for Distributed Selection. In *Proc. 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–153, 2007.

[32] E. Kushilevitz and N. Nisan. *Communication Complexity.* Cambridge University Press, 1997.

[33] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 27(7):558–565, 1978.

[34] C. Lenzen, T. Locher, and R. Wattenhofer. Clock Synchronization with Bounded Global and Local Skew. In *Proc. 49th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 509–518, 2008.

[35] C. Lenzen, T. Locher, and R. Wattenhofer. Tight Bounds for Clock Synchronization. In *Proc. 28th ACM Symposium on Principles of Distributed Computing (PODC)*, 2009.

[36] T. Locher and R. Wattenhofer. Oblivious Gradient Clock Synchronization. In *Proc. 20th International Symposium on Distributed Computing (DISC)*, pages 520–533, 2006.

[37] J. Lundelius and N. Lynch. An Upper and Lower Bound for Clock Synchronization. *Information and Control*, 62(2-3):190–204, 1984.

[38] J. Lundelius Welch and N. Lynch. A New Fault-Tolerant Algorithm for Clock Synchronization. *Information and Computation*, 77(1):1–36, 1988.

[39] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, 2002.

[40] S. L. Mantzaris. On "An Improved Algorithm for Finding the Median Distributively". *Algorithmica*, 10(6):501–504, 1993.

[41] J. M. Marberg and E. Gafni. An Optimal Shout-Echo Algorithm for Selection in Distributed Sets. In *Proc. 23rd Annual Allerton Conference on Communication, Control, and Computing*, 1985.

[42] M. Mitzenmacher. A Brief History of Generative Models for Power Law and Lognormal Distributions. *Internet Mathematics*, 1(2):226–251, 2004.

[43] Y. Moses and B. Bloom. Knowledge, Timed Precedence and Clocks. In *Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 294–303, 1994.

[44] A. Negro, N. Santoro, and J. Urrutia. Efficient Distributed Selection with Bounded Messages. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):397–401, 1997.

[45] R. Ostrovsky and B. Patt-Shamir. Optimal and Efficient Clock Synchronization under Drifting Clocks. In *Proc. 18th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 3–12, 1999.

[46] B. Patt-Shamir. A Note on Efficient Aggregate Queries in Sensor Networks. *Theoretical Computer Science*, 370(1-3):254–264, 2007.

[47] B. Patt-Shamir and S. Rajsbaum. A Theory of Clock Synchronization. In *Proc. 26th ACM Symposium on Theory of Computing (STOC)*, pages 810–819, 1994.

[48] A. Pavan and S. Tirthapura. Range-Efficient Computation of F_0 over Massive Data Streams. *SIAM Journal on Computing*, 37(2):359–379, 2007.

[49] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications, 2000.

[50] A. A. Razborov. On the Distributional Complexity of Disjointness. *Theoretical Computer Science*, 106(2):385–390, 1992.

[51] M. Rodeh. Finding the Median Distributively. *Journal of Computer and System Sciences*, 24(2):162–166, 1982.

[52] D. Rodem, N. Santoro, and J. B. Sidney. Shout-Echo Selection in Distributed Files. *Networks*, 16(1):235–249, 1986.

[53] N. Santoro, M. Scheutzow, and J. B. Sidney. On the Expected Complexity of Distributed Selection. *Journal of Parallel and Distributed Computing*, 5(2):194–203, 1988.

[54] N. Santoro and J. B. Sidney. Order Statistics on Distributed Sets. In *Proc. 20th Annual Allerton Conference on Communication, Control, and Computing*, pages 251–256, 1982.

[55] N. Santoro, J. B. Sidney, and S. J. Sidney. A Distributed Selection Algorithm and Its Expected Communication Complexity. *Theoretical Computer Science*, 100(1):185–204, 1992.

[56] N. Santoro and E. Suen. Reduction Techniques for Selection in Distributed Files. *IEEE Transactions on Computers*, 38(6):891–896, 1989.

[57] A. Schönhage, M. S. Paterson, and N. Pippenger. Finding the Median. *Journal of Computer and System Sciences*, 13:184–199, 1976.

[58] L. Shrira, N. Francez, and M. Rodeh. Distributed k-Selection: From a Sequential to a Distributed Algorithm. In *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 143–153, 1983.

[59] H. A. Simon. On a Class of Skew Distribution Functions. *Biometrika*, 42(3-4):425–440, 1955.

[60] T. K. Srikanth and S. Toueg. Optimal Clock Synchronization. *Journal of the ACM*, 34(3):626–645, 1987.

[61] R. van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003.

[62] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A Linear-Time Probabilistic Counting Algorithm for Database Applications. *ACM Transactions on Database Systems*, 15(2):208–229, 1990.

[63] A. Yao. Probabilistic Computations: Toward a Unified Measure of Complexity. In *Proc. 18th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 222–227, 1977.

[64] Y. Yao and J. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *ACM SIGMOD Record*, 31(3):9–18, 2002.

[65] J. Zhao, R. Govindan, and D. Estrin. Computing Aggregates for Monitoring Wireless Sensor Networks. In *Proc. 1st IEEE International Workshop on Sensor Network Protocols and Applications (SNPA)*, pages 139–148, 2003.

Curriculum Vitae

March 20, 1980	Born in Zurich, Switzerland
1987–2000	Primary and high schools in Oberlunkhofen AG, Bremgarten AG, and Wohlen AG, Switzerland
2000–2005	Studies in computer science, ETH Zurich, Switzerland
2006–2009	Ph.D. student, research and teaching assistant, Distributed Computing Group, Prof. Dr. Roger Wattenhofer, ETH Zurich, Switzerland
February 2009	Ph.D. degree, Distributed Computing Group, ETH Zurich, Switzerland Advisor: Prof. Dr. Roger Wattenhofer Co-examiners: Prof. Dr. Nancy Lynch, MIT, Massachusetts, USA Prof. Dr. Christian Scheideler, TUM, München, Germany

Publications

In the following, all publications that I co-authored during my time with the Distributed Computing Group at ETH Zurich are listed.

1. Gradient Clock Synchronization in Dynamic Networks.
   Fabian Kuhn, Thomas Locher, Rotem Oshman. In *Proc. 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, August 2009.

2. Tight Bounds for Clock Synchronization.
   Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. In *Proc. 28th ACM Symposium on the Principles of Distributed Computing (PODC)*, August 2009.

3. Robust Live Media Streaming in Swarms.
   Thomas Locher, Remo Meier, Stefan Schmid, and Roger Wattenhofer. In *Proc. 19th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, June, 2009.

4. Clock Synchronization with Bounded Global and Local Skew.
   Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. In *Proc. 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, October 2008.

5. Distributed Selection: A Missing Piece of Data Aggregation.
   Fabian Kuhn, Thomas Locher, and Roger Wattenhofer. In *Communications of the ACM*, September 2008.

6. Distributed Computation of the Mode.
   Fabian Kuhn, Thomas Locher, and Stefan Schmid. In *Proc. 27th ACM Symposium on the Principles of Distributed Computing (PODC)*, August 2008.

7. Sensor Networks Continue to Puzzle: Selected Open Problems.
   Thomas Locher, Pascal von Rickenbach, and Roger Wattenhofer. In *Proc. 9th International Conference on Distributed Computing and Networking (ICDCN)*, January 2008.

8. Push-to-Pull Peer-to-Peer Live Streaming.
   Thomas Locher, Remo Meier, Stefan Schmid, and Roger Wattenhofer. In *Proc. 21st International Symposium on Distributed Computing (DISC)*, September 2007.

9. Rescuing Tit-for-Tat with Source Coding.
   Thomas Locher, Stefan Schmid, and Roger Wattenhofer. In *Proc. 7th IEEE International Conference on Peer-to-Peer Computing (P2P)*, September 2007.

10. Tight Bounds for Distributed Selection.
    Fabian Kuhn, Thomas Locher, and Roger Wattenhofer. In *Proc. 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2007.

11. Free Riding in BitTorrent is Cheap.
    Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. In *Proc. 5th Workshop on Hot Topics in Networks (HotNets)*, November 2006.

12. Oblivious Gradient Clock Synchronization.
    Thomas Locher and Roger Wattenhofer. In *Proc. 20th International Symposium on Distributed Computing (DISC)*, September 2006.

13. eQuus: A Provably Robust and Locality-Aware Peer-to-Peer System.
    Thomas Locher, Stefan Schmid, and Roger Wattenhofer. In *Proc. 6th IEEE International Conference on Peer-to-Peer Computing (P2P)*, September 2006.

14. Received-Signal-Strength-Based Logical Positioning Resilient to Signal Fluctuation.
    Thomas Locher, Roger Wattenhofer, and Aaron Zollinger. In *Proc. 1st ACIS International Workshop on Self-Assembling Wireless Sensor Networks (SAWN)*, May 2005.