DISS. ETH NO. 20240

Coping with Selfishness in Distributed Systems: Mechanism Design in Multi-Core and Peer-to-Peer Systems

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

RAPHAEL PAUL EIDENBENZ

MSc ETH INFK, ETH Zürich

born 13.09.1980

citizen of

Zürich (ZH)

accepted on the recommendation of

Prof. Roger Wattenhofer, examiner
Prof. Dov Monderer, co-examiner
Prof. Karl Aberer, co-examiner

2012

Abstract

Distributed systems with autonomous and self-interested participants often exhibit deficiencies due to selfishness of its participants. Mechanism design is the discipline that optimizes systems by taking selfish behavior into account.

In the first part of this thesis, we study how a mechanism designer can influence games by promising payments to the players. We first investigate the cost of implementing a desirable behavior. Whereas a mechanism designer can decide efficiently whether strategy profiles can be implemented at no cost at all computing an optimal implementation is generally NP-hard. Second, we introduce and analyze the concept of *leverage* in a game. The leverage captures the benefits that a benevolent or a malicious mechanism designer can achieve within economic reason, i.e., by taking the implementation cost into account. Mechanism designers can often manipulate games and change the social welfare by a larger extent than the amount of money invested. Unfortunately, computing the leverage is generally intractable as well.

In the second part of this thesis, we study the incentives exhibited by transactional memory systems. We find that with most current contention managers, transactional memory systems do not incentivize good programming practice, i.e, programmers are encouraged to make transactions coarse rather than fine-grained. We show how Timestamp-like contention managers can be modified so as to feature beneficial incentives. In general, however, priority-based conflict resolution policies are prone to be exploited by selfish programmers. In contrast, a simple manager that resolves conflicts at random is compatible with good-programming incentives.

In the third part of this thesis, we investigate the potential of barter across swarms and along cycles of interest to boosting the market liquidity of tit-for-tat based peer-to-peer file sharing systems. By means of simulations, we find that the proposed measures shorten the median download completion time by more than one third. Furthermore, we study the problem of guiding participants of an established system to using an improved system in a smooth transition. As an entailed problem, we discuss how a conspiring peer can safely determine a connected peer's type, i.e., how can she learn whether the connected peer is a conspirer or a regular peer without giving away her conspiring identity in the latter case. Our solution is a steganographic handshake. Finally, we look at the problem of how a conspirer can broadcast a message secretly to all fellow conspirers in a monitored environment. For several levels of monitoring, we propose distributed and efficient algorithms that transmit hidden information by varying the block request sequence meaningfully.

Zusammenfassung

Verteilte Systeme mit autonomen, eigennützigen Teilnehmern weisen oft Mängel auf, die auf den Egoismus ihrer Teilnehmer zurückzuführen sind. Mechanismus-Design ist die Disziplin, welche Systeme optimiert unter Einbezug egoistischen Verhaltens.

Im ersten Teil dieser Dissertation gehen wir der Frage nach, wie ein Mechanismus-Designer durch das Anbieten von Zahlungen Spiele beeinflussen kann. Als Erstes untersuchen wir die Implementationskosten gewünschten Verhaltens. Während ein Mechanismus-Designer effizient entscheiden kann, ob Strategieprofile kostenlos implementiert werden können, ist das Berechnen einer optimalen Implementation NP-hart. Als Zweites führen wir das Konzept von *Leverage* ("Hebelwirkung") in Spielen ein. Leverage beziffert den Nutzen, den sich ein wohlwollender oder ein böswilliger Mechanismus-Designer unter Einbezug der Implementationskosten erwirtschaften kann. Oft kann ein Spiel so manipuliert werden, dass das resultierende Gemeinwohl grösser ist als das investierte Geld. Leider ist auch die maximale Leverage schwer zu berechnen.

Im zweiten Teil dieser Dissertation untersuchen wir die Anreize in Transactional-Memory-Systemen. Wir finden heraus, dass die meisten gängigen Transactional-Memory-Systeme Anreize zu schlechter Programmierpraxis schaffen, d.h. Programmierer werden ermutigt Transaktionen lang statt kurz zu machen. Wir zeigen wie man Timestamp-artige Contention-Manager modifizieren muss, um Anreize für eine gute Programmierpraxis zu schaffen. Im Allgemeinen sind prioritätsbasierte Contention-Manager jedoch anfällig gegenüber egoistischem Verhalten.

Im dritten Teil dieser Dissertation untersuchen wir das Potenzial von schwarmübergreifendem Tauschhandel entlang Interessenszyklen, um die Liquidität eines auf Tit-For-Tat basierten Peer-to-Peer Filesharingsystems zu erhöhen. Mittels Simulationen finden wir heraus, dass die Einführung derartigen Tauschhandels die Dauer eines Downloads im Median um mehr als einen Drittel verkürzt. Überdies zeigen wir, wie man Teilnehmer eines etablierten Systems mittels einer glatten Übergangsphase dazu bringen kann, ein verbessertes System zu benutzen. Als Folgeproblem betrachten wir die Frage, wie ein konspirierender Peer gefahrlos den Typ eines verbundenen Peers feststellen kann, d.h. wie er erfahren kann, ob der verbundene Peer auch ein Verschwörer ist oder nicht, ohne dabei in letzterem Fall seinen eigenen Typ preiszugeben. Unsere Lösung ist ein steganographischer Handshake. Schliesslich untersuchen wir, wie ein Verschwörer in einem überwachten Netzwerk eine geheime Nachricht an alle Verschwörer übermitteln kann. Für verschiedene Überwachungsstufen schlagen wir effiziente, verteilte Algorithmen vor, welche Information in der Reihenfolge der Block-Requests verstecken.

Acknowledgements

During the four years of my Ph.D. at ETH Zurich I had the chance to work with several inspiring researchers. Without their support, this thesis would not have been possible.

First and foremost, I would like to thank my advisor, Prof. Roger Wattenhofer, for his guidance and his ability, to point me to interesting problems. I am also deeply indebted to my two co-examiners, Prof. Karl Aberer and Prof. Dov Monderer, who served on my committee board and provided me with illuminative remarks to improve the final version of this thesis. I am especially grateful to Stefan Schmid and Yvonne-Anne Pignolet who shared their interest in mechanism design with me in the course of several research projects, including my master thesis. Furthermore, they unselfishly enabled me to join them for ISAAC in Japan with a lasting effect. Their contagious enthusiasm for research has been an inspiration. Another big thank you goes out to Thomas Locher, who I had the pleasure to collaborate with in several projects. His high standards of scientific writing were of great avail, and his lessons in Chinese characters or NBA team names will remain unforgotten.

I would also like to thank all my co-workers at the Distributed Computing Group that contributed to a great research environment: my office mate, Johannes Schneider, who shared my alpine heritage, and with whom I went on adventures in Canada and Hawaii. My other two office mates, Christoph Lenzen, who took care of Stefan's plant, and Klaus-Tycho Förster, who provided me with sweets on gray winter days. I thank Roland Flury and Jasmin Smula for pumping lemmas together, Barbara Keller and Samuel Welten for disseminating knowledge on distributed systems with me, Yuval Emek for educating me on spectral graph theory, Remo Meier for his Java almightiness, Michael Kuhn for mapping music, Philipp Sommer for co-organizing the Tuscany retreat, Nicolas Burri for sharing the passion for coffee, Olga Goussevskaia and Pascal von Rickenbach for ever-interesting chats, Tobias Langner for administering, Stephan Holzer for making adfocs bearable, Jochen Seidel for his roll, Jara Uitto for making my Ph.D. defense an even more memorable day, and Philipp Brandes for taking over the PODC lecture. Furthermore, I would like to thank the friendly and competent administrative staff at TIK: Roland Mathis, Tanja Lantz, Beat Futterknecht, Thomas Steingruber, Damian Friedli; as well as my students: Yu Li, Christian Decker, David Stolz, Roger Odermatt, Mathias Karlsson, Michael König, Pascal Studerus, Damiano Boppart, and Erwin Herrsche. Most of their engaged work during their theses under my supervision has contributed to my thesis in one way or another.

Finally, I am very grateful for the constant support of my friends and my family, my sister Eva-Maria, my brother Stephan, his wife Seraina, their lovely children, Tayra, Yaris, Lyas and Malin, my mother, Silvia, and especially my dad, Hans, who unfortunately could not see the day of my PhD graduation. Their firm confidence in my abilities has always been reassuring. I want to thank my wonderful girlfriend, Neli, for her love, which has overcome 10'000 kilometers so easily. At last, I want to thank the members of my two bands, Mills Embargo and Heizkörper, who have become a second family to me.

Contents

Chapter 1

Introduction

Most societies and distributed systems in which multiple agents interact to achieve common or individual goals exhibit an intricate socio-economic complexity. Understanding and formally describing the socio-economic interplays between autonomous and rational (or selfish) participants of a distributed system is the subject of game theory. The realm of game theory cuts across a wide variety of fields such as biology, economics, politics, or computer science. Its powerful tools allow for analyses of incentive structures present in distributed systems. In many cases, such a game-theoretic analysis yields the insight that the respective system suffers inefficiencies due to effects of selfishness. To quantify the deficiency caused by selfishness, the performance loss of a distributed system consisting of selfish participants is computed with respect to the performance of an optimal reference system where all participants collaborate perfectly. If this performance loss—often referred to as price of anarchy—shows to be large this indicates that the protocol should be extended by a mechanism that sets stronger incentives and thus encourages the participants to cooperate.

If we define game theory with the words of Robert Aumann as a "sort of umbrella or 'unified field' theory for the rational side of social science, where 'social' is interpreted broadly, to include human as well as non-human players (computers, animals, plants)" then we should define *mechanism design*—often referred to as *inverted* game theory, or *applied* game theory—as the social engineering discipline that harnesses the rationality of human and non-human players to achieve a desired design goal. In a society, the typical goal of mechanism design is to optimize social welfare, e.g., taxation policies aim at redistributing wealth so as to guarantee decent minimum standards of living. In the realm of networked computer systems, where the players are computers, programmed and used by humans, the typical goal is to optimize

the system performance.

Hence, mechanism design is essentially an optimization technique that takes the behavior of the system participants into account. A natural first step of such an optimization of an existing system is to analyze the present incentive structure by the means of game theory. If the system indeed proves to exhibit incentive-related deficiencies the system is enhanced with a mechanism in a second step to improve the incentives offered for cooperation, if possible. In some cases, improving the incentives requires a rigorous modification of the existing system. As a consequence, the performance of the entire system has to be re-evaluated and compared to the performance of the precursor to assess the quality of the optimization.

Greatly motivated by the advent of large networked systems like the Internet, game theory has been used to analyze the impact of selfishness for the last few decades; in some cases, like with peer-to-peer file sharing, the design of mechanisms has led to substantial performance improvements of the respective systems.

This thesis consists of three parts that all investigate aspects of mechanism design: in a theoretical first part, we examine a mechanism designer's possibilities of influencing games with payments; in the second and third part, we analyze the incentives of transactional memory systems, evaluate a mechanism for peer-to-peer file sharing, and give solutions to some entailed practical problems.

1.1 Mechanism Design with Payments

In many distributed systems, a mechanism designer cannot change the rules of interactions. However, she may be able to influence the players' behavior by offering payments for certain outcomes. On this account, we consider a mechanism designer whose power is based on her monetary assets and her creditability, i.e., the players trust her to pay the promised payments. Thus, a certain subset of outcomes is implemented in a given game if the additional non-negative payments make it rational for players to choose one of the desired outcomes. A designer faces the following optimization problem: How can the desired outcome be implemented at minimal cost? We tackle this question by analyzing the complexity of this task. Surprisingly, it is sometimes possible to improve (or worsen) the performance of a given system merely by creditability, i.e., without any payments at all: promising payments for other profiles can function as some sort of insurance upon which players choose a better strategy, ending up in a profile where eventually no payments are made.

Whether a mechanism designer is willing to invest the cost of implementing a desired outcome often depends on how much better the implemented

outcome is than the original outcome. If the social welfare gain does not exceed the implementation cost the mechanism designer might decide not to influence the game at all. In many games, however, manipulating the players' utility is profitable.

Part I, covering mechanism design with payments, is based on work published at the International Conference on Combinatorial Optimization and Applications [34], the International Symposium on Algorithms and Computation [33], and the International Game Theory Review [35].

1.2 Mechanism Design in Multi-Core Systems and Peer-to-Peer Systems

Transactional memory denotes a paradigm for multicore computing that was promoted over the last few years by researchers and parts of the industry. The basic idea is to let software developers wrap critical code sections, i.e., sections with accesses to shared data structures, into transactions, rather than to have them declare explicit locks of certain memory objects. In the light of the new computing era where clock rates of processors do not speed up anymore, but the number of cores grows instead, transactional memory systems promise to offer an easier handling of concurrency than the traditional techniques. Software engineers agree that it is crucial to ease the process of developing concurrent code if the distributed computing power of new processors is to be harnessed properly. Indeed, after Intel, IBM, and Sun founded a drafting group for the development of a transactional memory specification in 2009, IBM presented the first commercial processor with hardware support for transactional memory, named PowerPC A2, at the Supercomputing conference in November 2011. Intel announced a Haswell processor with hardware transactional memory to enter the markets in 2013.

The recent developments raise hope that the future will see unimagined applications that utilize an ever growing computing power. However, no research work, or practical work has investigated into the question of whether the proposed systems exhibit the right incentives to programmers that write code for such processors. We believe this is a severe neglect with potentially fatal consequences: the different programs running on a multicore machine, or the threads of one program each of which runs on a different core, compete for shared resources, e.g. shared memory. Thus, also multicore systems are susceptible to selfish behavior. We analyze the incentives exhibited by current transactional memory systems in Part II of this thesis.

The peer-to-peer (p2p) paradigm denotes the architecture of networked applications where all participants have equal privileges and equal duties. As opposed to client-server architecture, where one or several designated servers serve requests of clients, peers in a p2p system both consume services and

provide services. A major challenge of p2p computing is the coordination of peers without a central infrastructure. The lack of a central authority also makes it harder to enforce compliance with the protocol. Whereas in a centralized system, the central authority can usually detect and punish non-compliant participants, e.g., by excluding them from the system, it needs a lot more effort and coordinated action to trace and punish non-compliant behavior in decentralized systems as the work done by the central authority has to be emulated by the peers. Moreover, the client users of a client-server system usually have no incentive to deviate from the protocol, since they are not supposed to offer the servers a return service in the first place.[1] Selfish participants of a p2p system, however, may try to profit from the service of others without serving requests themselves.

The analysis of the incentives exhibited by early p2p file-sharing systems disclosed that it was easy to cheat the protocol and download content without uploading was feasible. More so, the lack of incentives to contribute led to significant deficiencies and a degraded performance of systems like Kazaa. The designers of a next generation p2p file-sharing system, BitTorrent, took the selfishness of the users into account and implemented a mechanism with the protocol that rewards a higher level of contribution with faster downloads.

With its high popularity—a share of up to 70% of the Internet traffic is accounted to BitTorrent—BitTorrent is arguably the most prominent example of successful mechanism design. Although BitTorrent sets beneficial incentives to a certain extent, the selfish client BitTyrant [75] and the free-riding client BitThief [60] prove that also BitTorrent may suffer an inefficiency due to selfishness. This observation raises hope that further improvements of the incentive structures encourage peers to contribute more, and thus increase the effectiveness of p2p file-sharing systems. Barter is one obvious way to impose rather strict incentives. A tit-for-tat trading of data parts forces peers to upload approximately as much as they download. However, such a strict trading regime entails further challenges as it requires peers to be mutually interested in data parts of each other in order to trade. Put differently, whereas with traditional BitTorrent clients, peers often give out data for free, introducing strict barter decreases the market liquidity of a system. In Part III of this dissertation, we study to what extent this challenge can be met by introducing barter along cycles of interest, and barter across swarms, both measures that increase the market liquidity. We then discuss the problem of replacing an established file-sharing system with a newer, potentially more effective protocol. As an example, we explain the solution taken in our

[1] An exception are services where clients are supposed to reimburse the server. Selfish clients may try to get the service without paying. For example, a user of a paid video streaming service might try to receive an unsubscribed video stream for free. Browser plugins that filter out advertisements of websites are another example where the user does not "pay" for viewing the site's content in terms of exposure to the advertisements.

BitThief project that includes a steganographic method to reveal a connected peer's type (BitThief client or other Bittorrent client) safely, i.e., without revealing the type to non-BitThief clients. Finally, we consider the question of how a conspiring subgroup of peers in a p2p network can find each other and coordinate their action without provoking suspicion among other peers or an authority that monitors the network.

Part II on multi-core systems presents work that has been published at the International Symposium on Algorithms and Computation [33] and in the Journal of Theoretical Computer Science [37]. Part III on peer-to-peer systems is based on work published at the International Conference on Computer Communications [32] and on work currently under submission [31].

Part I

Harnessing Games with Payments

Chapter 2

An Introduction to Game Theory and Mechanism Design

Selfishness can be logical (. . .) or practical. The logical egoist considers it unnecessary to verify his adjudgement with the intellect of others (. . .). [The practical egoist] does not see any value in things other than those from which he benefits.
Immanuel Kant, (1724 - 1804)

Game theory can be described as the attempt to mathematically capture behavior in strategic situations (games), in which an individual's success in making choices depends on the choices of others. A classic example of a game is the situation often referred to as *Chicken*: two cars are approaching an intersection, when the drivers realize that if both continue their speed they will crash into each other. Given that there is no rule which car has to yield to the other, both drivers have two options available: either stop and yield to the other car, or drive on and hope the other will yield. Concentrating on these two options, we can describe this situation as a game played by the drivers where each of them has to pick one of two options—usually called *strategies* in game theory. Depending on which strategies the drivers pick, the game can have four different outcomes: if Driver 1 yields and Driver 2 drives on Driver 1 has a low, or even a negative payoff, as she loses time and gas, and Driver 2 has a high payoff. If Driver 1 drives on and Driver 2 stops Driver 1 has a high payoff and Driver 2 has a low payoff. If both drivers try yield to the other they both have a low payoff because they both stop the

	stop	go
stop	0 0	-1 +1
go	+1 -1	-10 -10

Figure 2.1: Chicken. *An anti-coordination 2-player game.*

car. If both drivers do not throttle their speed they crash and incur a high damage, i.e., a high negative payoff.

Such games with two players can be conveniently written as a bi-matrix, a matrix where each entry consists of a pair of values. Chicken is depicted in Figure 2.1. Each row of the bi-matrix refers to a strategy available to Player 1, and each column refers to a strategy available to Player 2. The first value of an entry i, j is the payoff for Player 1 given that Player 1 picks strategy i and Player 2 picks strategy j. The second value of an entry is the payoff for Player 2 if the game yields the corresponding outcome. Note that for games with more than 3 players it is already more difficult to give a concise representation as we need to describe multi-dimensional matrices with multi-valued entries. For the sake of presentation, we will thus primarily provide examples of two player games in this introduction.

Chicken is a so-called *anti-coordination* game since each player would prefer to do the opposite of the option chosen by the other player: if Driver 2 knew that Driver 1 stops she could decide to drive on without any problem, thus achieving the highest valued outcome for her. On the other hand, if Driver 2 knew that Driver 1 does not stop she would probably rather stop and prevent a crash, i.e., in terms of payoffs, she would rather incur cost of 1 instead of 10. Note that for the above argumentation we already assumed a basic *rationality* of the two involved players. Even though it seems a strange choice not to stop and to knowingly provoke a crash there might be real life instances of the Chicken game where a player still prefers this choice to yielding to the other driver. For the analyses of games in the following, we assume that players behave *rationally* in the sense that they always try to maximize their payoff, and *selfishly* in the sense that they do not care about the payoffs, or costs eventually collected or incurred by others. Hence, we do not consider any altruistic emotions that might lead people to opt for a strategy that does not maximize their payoff, but increases the payoff of others instead. We also disregard feelings of pride, or anger that might lead players to deviate from the strategy with optimal payoff, as it might happen in the Chicken game, for instance.[1] Indifference towards the payoff that

[1] An act of brinkmanship is to put up with an uncontrollable risk. Such uncontrollable

	concert	football
concert	10 3	1 1
football	-10 -10	-5 10

Figure 2.2: Distorted Battle of the Sexes. *The row player, Judy, dislikes football to such an extent that she would rather attend the concert alone.*

---

others receive does not prevent a player from taking the payoffs of others into account. On the contrary, a rational player tries to predict the behavior of others with all information about the game available, and chooses a strategy based on these predictions. To illustrate the reasoning of a player, consider the following distorted variant of the game known as *Battle of the Sexes (DBoS)*: a couple—let us call them Judy and Jim—makes plans for an evening out. Consulting the event calendar, they learn that the only two events that are scheduled that evening are a football match and a classical concert. Jim prefers going to the football match, but he would still rather attend the concert than spending the evening without Judy. Judy on the other hand, dislikes football[2] so much that she would rather go to the concert alone than to join Jim for the football match. See Figure 2.2 for the bi-matrix representation of the game. The reasoning for Judy is that no matter what Jim opts for, she will be better off by attending the concert. As Jim knows Judy's preferences, he predicts that Judy will go to the concert, thus, going to the concert is the better option also for him. Following this rationale, we can predict that Judy and Jim will go to the concert together.

The discipline of game theory consists of two basic activities: modelling situations as games, and modelling the behavior of the players so as to predict the outcome of games. Thereby, the rationality and selfishness assumption serves as a basis for modelling behavior. One assumption shared by most game-theoretic work, if not all, is that a player never opts for a strategy that is *dominated* by another strategy, i.e., a strategy for which there exists another strategy that is always better regardless of the other players' choices. In DBoS, for instance, Judy's option to attend the concert dominates the option of going to the football match. Moreover, attending the concert is called a *dominant* strategy as it dominates all other strategies available to Judy. In games where every player has exactly one dominant strategy, this assumption suffices to predict one unique outcome of the game, namely the

---

risk is taken by Corey Allen's character in the famous "chickie run" scene from the film *Rebel Without a Cause*, when he cannot escape from the car and dies in the crash.

[2]Pardon the gender stereotype.

outcome where all participants play their dominant strategy. Prediction rationales such as the one yielded by the assumption that players never choose dominated strategies are called *solution concepts*. Note that the *dominant strategy solution concept* predicts a unique solution only in games where all participants have a dominant strategy. In other games, however, it only decreases the solution space to outcomes containing all dominant strategies. In the DBoS game, for instance, it decreases the solution space to the outcomes where Judy opts for the concert, but it does not make any prediction about Jim's choice of strategy. In the Chicken game, neither of the drivers has a dominant strategy available.

There are several proposals of solution concepts that make additional assumptions to further reduce the solution space. The most famous is probably the *Nash equilibrium*, named after John F. Nash who proposed it in 1950 [70]. A Nash Equilibrium (NE) is a strategy profile (a possible outcome of the game) for which it holds that no player can improve her payoff by unilaterally changing her strategy. The Nash equilibrium in the DBoS game is the strategy profile where both Judy and Jim go to the concert. The Chicken game, for instance, has two Nash equilibria: the two outcomes where one driver stops and the other does not.

As a solution concept, the highly acclaimed Nash equilibrium is especially suited for games that are played more than once, or for games where the participants are allowed to discuss their strategies before playing. In systems where the same game is repeated over and over, Nash equilibrium points are particularly important because they represent *stable* states of the systems, i.e., a state where no participant has an incentive to unilaterally change the current strategy. For games with more than one Nash equilibrium, the Nash equilibrium solution concept does not make any prediction about which Nash equilibrium is most likely to be the outcome of the game. Other solutions concepts, not considered in this dissertation, include correlated equilibria [13], mixed Nash equilibria [71], and sequential elimination of dominated strategies.

Although Nash equilibria and dominant strategy profiles are desirable game outcomes in the sense that they are stable solutions, they do not necessarily coincide with states that exhibit the largest "social value". This fact is prominently illustrated by the game called *Prisoner's Dilemma* (cf. Figure 2.3): Two bank robbers are arrested by the police. The policemen have insufficient evidence for convicting them of robbing a bank, but they could charge them with a minor crime. Cleverly, the policemen interrogate each suspect separately and offer both of them the same deal. If one testifies to the fact that his accomplice has participated in the bank robbery, they do not charge him for the minor crime. If one robber testifies and the other remains silent, the former goes free and the latter receives a three-year sentence for

	silent	testify
silent	3 3	0 4
testify	4 0	1 1

Figure 2.3: Prisoner's Dilemma. *Each player can either remain* silent *or* testify *to help convicting the other player. Payoffs are expressed in terms of* saved *years in prison.*

---

robbing the bank and a one-year sentence for committing the minor crime. If both betray the other, each of them will get three years for the bank robbery. If both remain silent, the police can convict them for the minor crime only and they get one year each.

In the Prisoner's Dilemma game, no matter which strategy the other player chooses, a bank robber is better off by testifying, i.e., testifying is a dominant strategy for both bank robbers. The strategy profile where both players testify is a dominant outcome, and thus also a Nash equilibrium. Hence, selfish and rational bank robbers will end up testifying against each other, and go to jail for three years each. From the players' point of view this outcome is particularly tragic, because if both of them remained silent they would spend only one year in prison, thus save two years each.

In the following, we will show that by offering payments to the players that can depend on the strategy profile chosen by the players, an external entity, a mechanism designer, can influence the outcome of the game. For some games, this external *mechanism designer* can lead the players to choose an outcome that yields much better payoffs at only minor incurring cost—sometimes at no cost at all.

2.1 Al Capone and the Prisoners Dilemma

Consider the following extension to the Prisoner's Dilemma game: Let us add another well-available option to the strategy set of both bank robbers, namely the option to completely *confess* to the bank robbery and thus supply the police with evidence to convict both criminals for a four-year sentence. The bi-matrix of the extended Prisoner's Dilemma game G is depicted in Figure 2.4. Note that payoffs are again expressed in terms of *saved* years compared to the maximum of four years. A short game-theoretic analysis shows that both player's best strategy is still to testify. Thus, the prisoners will betray each other and both get charged a three-year sentence.

Now, let the two bank robbers be members of the *Al Capone gang*. Thus,

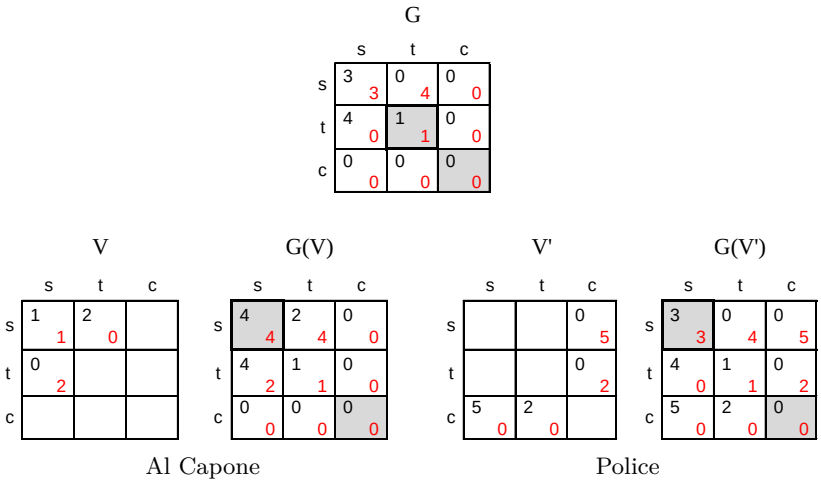

Figure 2.4: *Extended prisoners' dilemma: G shows the prisoners' initial payoffs, where payoff values equal* saved years. *The first strategy is to remain silent (s), the second to testify (t) and the third to confess (c). Nash equilibria are colored gray, and non-dominated strategy profiles have a bold border. The left bi-matrix V shows Mr. Capone's offered payments which modify G to the game $G(V)$. By offering payments V', the police implements the strategy profile (c, c). As $V_1(c, c) = V_2(c, c) = 0$, payments V' implement (c, c) for free.*

Mr. Capone, the gang leader, has an interest in the outcome of the game, since he wants to dispense with the two gang members for a period as short as possible. Furthermore, let Mr. Capone get a chance to take influence on his employees' decisions. Before they take their decision, Mr. Capone calls each of them and promises that if they both remain silent, they will receive money compensating for one year in jail—for this scenario, we presume that time really is money. Furthermore, if one remains silent and the other betrays him, Mr. Capone will pay the former money worth two years in prison (cf. V in Figure 2.4). Thus, Mr. Capone creates a new situation for the two criminals where remaining silent is the most rational behavior. We say, Mr. Capone has *implemented* the outcome where both criminals stay silent. Thereby, he has saved his gang an accumulated two years in jail.

Let us also consider a slightly different scenario where, after the police officers have made their offer to the prisoners, their commander-in-chief de-

vises an even more promising plan. He offers each criminal to drop two years of the four-year sentence if he confesses the bank robbery and his accomplice betrays him. Moreover, if he confesses and the accomplice remains silent they would let him go free and even reward his honesty with a share of the booty (worth going to prison for one year). However, if both suspects confess the robbery, they will spend four years in jail. In this new situation, it is most rational for a prisoner to confess. Consequently, the commander-in-chief implements the best outcome from his point of view without dropping any sentence and he increases the accumulated years in prison by two.

From Mr. Capone's point of view, implementing the outcome where both prisoners keep quiet results in four saved years for the robbers. By subtracting the implementation cost, the equivalent to two years in prison, from the saved years, we see that this implementation yields a benefit of two years for the Capone gang. We say that the *leverage* of the strategy profile where both prisoners play s is two. For the police, the leverage of the strategy profile where both prisoners play c is two, since the implementation costs nothing and increases the years in prison by two. Since implementing c reduces the players' gain, we say the strategy profile where both play c has a *malicious leverage* of two. In the described scenario, Mr. Capone and the commander-in-chief solve the optimization problem of finding the game's strategy profile(s) which bear the largest leverage, or malicious leverage respectively, and therewith the problem of implementing the corresponding outcome at optimal cost.

In the remainder of Part I of this dissertation, we study how a mechanism designer can influence games by promising payments to the players depending on their mutual choice of strategies. In Chapter 3, we investigate the cost of implementing a desirable behavior and present algorithms to compute this cost. Whereas a mechanism designer can decide efficiently whether strategy profiles can be implemented at no cost at all our complexity analysis indicates that computing an optimal implementation is generally NP-hard. In Chapter 4, we introduce and analyze the concept of *leverage* in a game. The leverage captures the benefits that a benevolent or a malicious mechanism designer can achieve by implementing a certain strategy profile region within economic reason, i.e., by taking the implementation cost into account. Mechanism designers can often manipulate games and change the social welfare by a larger extent than the amount of money invested. Unfortunately, computing the leverage turns out to be intractable as well in the general case. In the remainder of this chapter, we introduce our basic model of game theory and implementation.

2.2 Model

2.2.1 Game Theory

A finite *strategic game* can be described by a tuple

$$G = (N, X, U),$$

where $N = \{1, 2, \ldots, n\}$ is the set of *players* and each player $i \in N$ can choose a *strategy* (action) from the set X_i. The product of all the individual players' strategies is denoted by

$$X := X_1 \times X_2 \times \ldots \times X_n.$$

In the following, a particular outcome $x \in X$ is called *strategy profile* and we refer to the set of all other players' strategies of a given player i by

$$X_{-i} = X_1 \times \ldots \times X_{i-1} \times X_{i+1} \times \ldots \times X_n.$$

An element of X_i is denoted by x_i, and similarly, $x_{-i} \in X_{-i}$; we may write x_i, x_{-i} to denote strategy profile $x \in X$ where player i plays x_i and all other players play according to x_{-i}. Finally,

$$U = (U_1, U_2, \ldots, U_n)$$

is an n-tuple of *payoff functions* (utilities), where $U_i : X \to \mathbb{R}$ determines player i's payoff arising from the game's outcome. The *social gain* of a game's outcome is given by the sum of the individual players' payoffs at the corresponding strategy profile x, i.e.,

$$gain(x) := \sum_{i=1}^{n} U_i(x).$$

Let $x_i, x_i' \in X_i$ be two strategies available to Player i. We say that x_i *dominates* x_i' iff $U_i(x_i, x_{-i}) \geq U_i(x_i', x_{-i})$ for every $x_{-i} \in X_{-i}$ and there exists at least one x_{-i} for which a strict inequality holds. x_i is the *dominant* strategy for player i if it dominates every other strategy $x_i' \in X_i \backslash \{x_i\}$. x_i is a *non-dominated* strategy if no other strategy dominates it. We denote the set of non-dominated strategy profiles by

$$X^* = X_1^* \times \ldots \times X_n^*,$$

where X_i^* is the set of non-dominated strategies available to the individual player i. A *strategy profile set*—also called *strategy profile region*—$O \subseteq X$ of G is a nonempty subset of all strategy profiles X, i.e., a region in the payoff matrix consisting of one or multiple strategy profiles. Similarly to X_i and X_{-i}, we define

$$\begin{aligned} O_i &:= \{x_i \mid \exists\ x_{-i} \in X_{-i} \text{ s.t. } (x_i, x_{-i}) \in O\} \text{ and} \\ O_{-i} &:= \{x_{-i} \mid \exists\ x_i \in X_i \text{ s.t. } (x_i, x_{-i}) \in O\}. \end{aligned}$$

2.2.2 Mechanism Design with Payments

Our model is based on the classic assumption that players are rational and always choose a non-dominated strategy. Additionally, we assume that players do not collude. We examine the impact of payments to players offered by a *reliable mechanism designer* (an interested third party) who seeks to influence the outcome of a game. It is assumed that the mechanism designer has complete knowledge of the players' utilities. By *reliable* we mean that the owed payments will always be acquitted. Note that this differs from standard mechanism design where a designer (e.g., a government) defines an interaction for self-motivated parties that will allow it to obtain some desired goal (such as maximizing revenue or social welfare) taking the agents' incentives into account, see also the discussion in [66]. In many distributed systems, unfortunately, interested parties cannot control the rules of interactions. A network manager for example cannot simply change the communication protocols in a given distributed systems in order to lead to desired behaviors, and a broker cannot change the rules in which goods are sold by an agency auctioneer to the public.

The payments promised by the mechanism designer are described by a tuple of non-negative *payment functions*

$$V = (V_1, V_2, \ldots, V_n), \text{ where } V_i : X \to \mathbb{R}^+,$$

i.e., the payments for player i depend on the strategy Player i selects as well as on the choices of all other players. Thereby, we assume that the players trust the mechanism designer to finally pay the promised amount of money. The original game $G = (N, X, U)$ is modified to a game $G(V) := (N, X, [U + V])$ by the payments V, where

$$[U + V]_i(x) = U_i(x) + V_i(x),$$

that is, each player i obtains the payments of V_i in addition to the payoffs of U_i. The players' choice of strategies changes accordingly: Each player now selects a non-dominated strategy in $G(V)$. Henceforth, the set of non-dominated strategy profiles of $G(V)$ is denoted by $X^*(V)$, and $V(x)$ denotes the sum of all payments offered to the players when x is the game's outcome,

$$V(x) = \sum_{i=1}^{n} V_i(x).$$

Observe that we have made two implicit assumptions: The mechanism designer can observe the actions chosen by the players and the players can determine the payoffs of all their strategies and compute the best strategy among them.

Chapter 3

Implementation Cost & Complexity

The mechanism designer's objective is to bring the players to choose a certain outcome of the game for as little payments as possible. Monderer and Tennenholtz introduced the notion of k-implementations in [66] to denote mechanisms that manipulate the players' behavior with payments of total value at most k. For the smallest implementable units of a game, singletons, they derived a closed formula for the minimal costs k needed to implement it. This formula builds on the fact that in order to implement a strategy profile $z \in X$, for each player i, strategy z_i must be the dominant strategy for i in the game $G(V)$ that combines the original payoffs with the offered payments. To achieve dominance, $U_i(z) + V_i(z)$ must be at least as large as any payoff $U_i(x_i, z_{-i})$ of any other strategy $x_i \in X_i$, all other payments $V_i(z_i, x_{-i})$ can be chosen high enough to yield $U_i(z_i, x_{-i}) + V_i(z_i, x_{-i}) > U_i(x_i, x_{-i})$ for all $x_i \neq z_i, x_{-i} \neq z_{-i}$. In fact, this is exactly what Mr. Capone and the commander-in-chief achieve with the payments v, and V' respectively in the extended prisoner's dilemma of Figure 2.4.

Theorem 3.1 ([66]). *Let $G = (N, X, U)$ be a game with at least two strategies for every player. Every strategy profile z has an implementation V, and its implementation cost amounts to*

$$k(z) = \sum_{i=1}^{n} \max_{x_i \in X_i} \left(U_i(x_i, z_{-i}) - U_i(z_i, z_{-i}) \right).$$

Furthermore, observe that z constitutes a Nash equilibrium if and only if it holds for every player $i \in N$, $\max_{x_i \in X_i}(U_i(x_i, z_{-i}) - U_i(z_i, z_{-i})) = 0$. As a corollary to Theorem 3.1 we get that a strategy profile z is a Nash equilibrium if and only if z has a 0-implementation. This remarkable result

$G =$

20 0	11 9	15 15	15 15
11 9	20 0	15 15	15 15
19 10	10 19	9 11	0 20
10 19	19 10	0 20	9 11

$V =$

0 ∞	0 ∞	0 0	0 0
0 ∞	0 ∞	0 0	0 0
1 1	1 1	∞ 0	∞ 0
1 1	1 1	∞ 0	∞ 0

Figure 3.1: *2-player game where O 's optimal implementation V yields a region $|X^*(V)| > 1$.*

by [66] implies that some outcomes can be implemented without spending anything.[1]

Note that in general, there are strategy profile regions for which it is cheaper to implement the entire region rather than a singleton within that region. Consider the game G in Figure 3.1. In game G, each singleton o in the region O consisting of the four bottom left profiles has cost $k(o) = 11$ whereas V implements O at cost 2: in the game $G(V)$ induced by payments V, the first two strategies of both players are dominated by the third or the fourth strategy respectively. Thus, the non-dominated strategy profiles $X^*(V)$ in game $G(V)$ corresponds to O. No matter which of the profiles in O will eventually be the outcome of the game, the mechanism designer incurs cost of only 2. The game G can be generalized to an arbitrarily large difference in the implementation cost between a singleton and a region in the worst case, e.g., by increasing all payoffs in G that are larger than 15 by the same amount. The example of Figure 3.1 shows that it can be worthwhile for a mechanism designer not to be too restrictive in what should be implemented. If several outcomes are acceptable, and not just a singleton, cheaper implementations may exist. Lower implementation cost can be traded for the uncertainty about which of the outcome in the region will be picked by the players.

In the following, we therefore investigate the implementations of strategy profile regions. How difficult is it to find the optimal implementation for a given region? How can a mechanism designer compute the payments? And what are the optimal cost? We consider two scenarios leading to two kinds of implementation cost: *worst-case implementation cost* and *uniform implementation cost*.

[1] For a discussion of exact 0-implementations of profile *sets*, we refer the reader to [34].

3.1 Worst-Case Implementation Cost

In a first step, we study a perfect common knowledge scenario where all players know all strategy spaces X and payoff functions U, and the players are aware that the other players know X and U. Moreover, the mechanism designer calculates with the maximum possible payments for a desired outcome (*worst-case implementation cost*). For a desired strategy profile set O, we say that payments V *implement* O if $\emptyset \subsetneq X^*(V) \subseteq O$. V is called (worst-case) *k-implementation* if, in addition

$$V(x) \leq k, \quad \forall\, x \in X^*(V).$$

That is, the players' non-dominated strategies are within the desired strategy profile, and the payments do not exceed k for any possible outcome. Moreover, V is an *exact k-implementation* of O if all strategies of O are non-dominated in the resulting game, i.e.,

$$X^*(V) = O \text{ and } V(x) \leq k \;\forall x \in X^*(V).$$

The *cost* $k(O)$ of implementing O is the greatest lower bound of all non-negative numbers q for which there exists a q-implementation. Depending on the game and the region O to implement, there either exists an implementation that reaches the cost exactly, or one that reaches $k(O)$ up to an arbitrarily small positive number. We refer to such implementations as *optimal* implementations. That is, V is an *optimal implementation* of O if V implements O and $\max_{x \in X^*(V)} V(x) = k(O)$ for games and target regions where such a V exists, or if V implements O and $\max_{x \in X^*(V)} V(x) = k(O)+\epsilon$ for arbitrarily small $\epsilon > 0$ in the general case. The cost $k^*(O)$ of implementing O exactly is the greatest lower bound of all non-negative numbers q for which there exists an exact q-implementation of O. V is an *optimal exact implementation* of O if it implements O exactly and requires cost $k^*(O)$, or $k^*(O) + \epsilon$ for games and target regions where no exact implementation reaches the lower bound. The set of all implementations of O will be denoted by $\mathcal{V}(O)$, and the set of all exact implementations of O by $\mathcal{V}^*(O)$. Finally, a strategy profile set $O = \{z\}$ of cardinality one—consisting of only one strategy profile—is called a *singleton*. Clearly, for singletons it holds that non-exact and exact k-implementations are equivalent. For simplicity's sake we often write z instead of $\{z\}$. Observe that only subsets of X which are in $2^{X_1} \times 2^{X_2} \times \ldots \times 2^{X_n}$, i.e., the Cartesian product of subsets of the players' strategies, can be implemented exactly. We call such a subset of X a *rectangular strategy profile set*.[2] In conclusion, for the worst-case implementation cost, we have the following definitions.

[2]Note that within our model where payments are made to individual players in different profiles, non-dominated profile sets will always be rectangular.

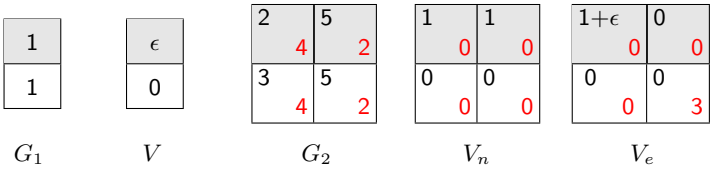

Figure 3.2: ϵ-Payments. *To implement the upper strategy profile in the one-player game G_1, the mechanism designer has to pay an arbitrarily small $\epsilon > 0$ to make the first strategy dominate the second. In game G_2, the region O consisting of the upper two strategy profiles has (exact) implementation cost $k(O) = k^*(O) = 1$. While the optimal exact implementation V_e needs ϵ-payments, the optimal non-exact implementation V_n can do without.*

Definition 3.2 (Worst-Case Cost and Exact Worst-Case Cost)**.** *The* worst-case implementation cost $k(O)$ *of a strategy profile set O is defined as*

$$k(O) := \inf_{V \in \mathcal{V}(O)} \left\{ \max_{z \in X^*(V)} V(z) \right\}.$$

A strategy profile set O has exact worst-case implementation cost

$$k^*(O) := \inf_{V \in \mathcal{V}^*(O)} \left\{ \max_{z \in X^*(V)} V(z) \right\}.$$

Note that we need to define the implementation cost by an infimum over all implementations of region O, as for some instances of games and target regions, ϵ-payments might be necessary to implement O. Figure 3.2 provides examples of such a problem instance. However, many instances allow an implementation that reaches the cost exactly. In particular, for a player i, whenever $O_{-i} \subsetneq X_{-i}$ a mechanism designer can use high payments for profiles $(o_i, \bar{o}_{-i})$ where $\bar{o}_{-i} \in X_{-i} \setminus O_{-i}$ to make it unnecessary to offer payments $V_i(o_i, o_{-i})$ to player i so that $[U + V]_i(o_i, o_{-i})$ exceeds the largest payoff $U_i(\bar{o}_i, o_{-i}), \bar{o}_i \in X_i \setminus O_i$.

Theorem 3.3. *For a game G and a rectangular target region $O \in X$, it holds that if $O_{-i} \subsetneq X_{-i}$ for all $i \in N$ then there exist payments V that implement O with cost equal to $k(O)$, and there exist payments V' that implement O exactly with cost $k^*(O)$.*

Proof. Let V be an optimal implementation of O. It must hold for every $i \in N$ and every strategy $o_i \in O_i$ that the set of strategies $\bar{o}_i \in X_i \setminus O_i$ that are dominated by o_i in $G(V)$ is optimal, otherwise V would not be optimal. For any pair of strategies $(o_i, \bar{o}_i)$ where o_i dominates $\bar{o}_i$ in $G(V)$, it must hold

that $[U+V]_i(o_i, x_{-i}) \geq [U+V]_i(\bar{o}_i, x_{-i})$ and strict inequality must hold for at least one $x_{-i} \in X_{-i}$. ϵ-payments are only needed to break ties between $\bar{o}_i$ and o_i if it would hold otherwise that $[U+V]_i(o_i, x_{-i}) = [U+V]_i(\bar{o}_i, x_{-i})$ for all x_{-i}. Since adding an ϵ-payment to one strategy profile (o_i, x_{-i}) is sufficient and there is at least one such profile outside O, there are no ϵ-payments needed inside O. Hence, among the profiles in $X^*(V) \subseteq O$ that exhibit cost equal to $\max_{z \in X^*(V)} V(z)$ there is at least one for which V does not contain any ϵ-payments. Otherwise the cost yielded by V could be improved by ϵ, contradicting optimality. Therefore, $k(O) = \max_{z \in X^*(V)} V(z)$.

The same argument proves the claim for the cost of exact implementations with the only difference that V be an optimal exact implementation of O, and $X^*(V) = O$. □

We would like to stress that Theorem 3.3 describes a one-sided implication: all problem instances with the described property have implementations that reach the cost, or the exact cost, without ϵ-payments; there are, however, games and target regions O where there exist $i \in N$ for which $O_{-i} = X_{-i}$, and an optimal implementation of O does still not need ϵ-payments (cf. game G_2 in Figure 3.2 for instance).

We now begin with studying exact implementations where the mechanism designer aims at implementing an *entire* strategy profile region. Exact region implementations are computationally cheaper to find compared to general region implementations, as calculating and comparing all the possible subregions is time-consuming. Subsequently, we examine general k-implementations.

3.1.1 Exact Implementation

Recall that the matrix V is an exact k-implementation of a strategy region O iff $X^*(V) = O$ and $\sum_{i=1}^{n} V_i(x) \leq k \ \forall x \in X^*(V)$, i.e. each strategy O_i is part of the set of player i's non-dominated strategies for all Players i. We present the first correct algorithm to find such implementations.[3]

Recall that in our model each player classifies the strategies available to her as either dominated or non-dominated. Thereby, each dominated strategy $x_i \in X_i \backslash X_i^*$ is dominated by at least one non-dominated strategy $x_i^* \in X_i^*$. In other words, a game determines for each player i a relation M_i^G from dominated to non-dominated strategies,

$$M_i^G \subseteq X_i \backslash X_i^* \times X_i^*,$$

where $(x_i, x_i^*) \in M_i^G$ states that $x_i \in X_i \backslash X_i^*$ is dominated by $x_i^* \in X_i^*$. See Figure 3.3 for an example. When implementing a strategy profile region O

[3] Monderer and Tennenholtz present a polynomial algorithm in [66] that is not correct (cf. [34]).

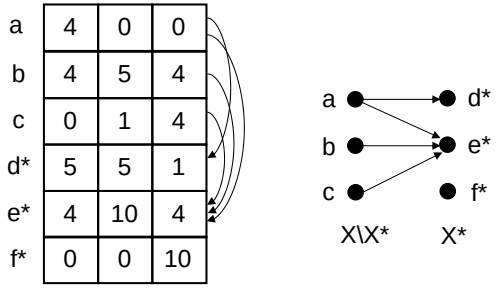

Figure 3.3: *Game from a single player's point of view with the corresponding relation of dominated ($X_i \backslash X_i^* = \{a, b, c\}$) to non-dominated strategies ($X_i^* = \{d^*, e^*, f^*\}$).*

---

exactly, the mechanism designer creates a modified game $G(V)$ with a new relation $M_i^V \subseteq X_i \setminus O_i \times O_i$ such that all strategies outside O_i map to at least one strategy in O_i. Therewith, the set of all newly non-dominated strategies of player i must equal O_i. Every $V \in \mathcal{V}^*(O)$ determines a set of relations $M^V := \{M_i^V : i \in N\}$. Vice versa, if we are given the relation set $M^{V'}$ of an optimal exact implementation V' we can compute, not necessarily V', but a similar implementation V with minimal payments and the same relation $M^V = M^{V'}$, i.e., given an "optimal" relation set we can find an optimal exact implementation. As an illustrating example, assume an optimal relation set for G with $(x_{i1}^*, o_i) \in M_i^G$ and $(x_{i2}^*, o_i) \in M_i^G$. Thus, we can compute V such that o_i must dominate x_{i1}^* and x_{i2}^* in $G(V)$, namely, the condition

$$U_i(o_i, o_{-i}) + V_i(o_i, o_{-i}) \geq \max_{s \in (x_{i1}^*, x_{i2}^*)} \left(U_i\left(s, o_{-i}\right) + V_i\left(s, o_{-i}\right)\right)$$

must hold for all $o_{-i} \in O_{-i}$. In an optimal implementation, Player i need not be offered payments for strategy profiles of the form $(\bar{o}_i, x_{-i})$ where $\bar{o}_i \in X_i \backslash O_i,\ x_{-i} \in X_{-i}$. Hence, the condition above implies that

$$V_i(o_i, o_{-i}) = \max\left\{0, \max_{s \in \{x_{i1}^*, x_{i2}^*\}} U_i(s, o_{-i}) - U_i(o_i, o_{-i})\right\}.$$

If $V' \in \mathcal{V}^*(O)$ is an optimal exact implementation then payments V consti-

tute an optimal exact implementation of O as well if it holds for any player i:

$$\begin{aligned} V_i(\bar{o}_i, x_{-i}) &= 0, \\ V_i(o_i, \bar{o}_{-i}) &= \infty, \\ V_i(o_i, o_{-i}) &= \max\left\{0, \max_{s \in S_i(o_i)} U_i(s, o_{-i}) - U_i(o_i, o_{-i})\right\}, \end{aligned}$$

where $S_i(o_i) := \{s \in X_i \setminus O_i \mid (s, o_i) \in M_i^{V'}\}$ are the strategies of player i that are dominated by o_i in $G(V')$. Therefore, the problem of finding an optimal exact implementation V of O corresponds to the problem of finding an optimal set of relations $M_i^V \subseteq X_i \setminus O_i \times O_i$.

Algorithm $\mathcal{ALG}_{exact}$ (cf. Algorithm 3.1) exploits this fact and constructs an implementation V for all possible relation sets, checks the cost that V would entail, and returns the lowest cost. The computation is done by recursively calling a subroutine *ExactK(V, i)*. Subroutine *ExactK(V, i)* returns the minimum cost incurred by any exact implementation $V' \in \mathcal{V}^*(O)$ that only adds additional payments to V, i.e., $V_i'(x) \geq V_i(x)$ for all $i \in N$, $x \in X$. Parameter i marks the current player in the recursion, i.e., payments V already yield the desired set of non-dominated strategies for the players in $\{i+1, \ldots, n\}$. Note that V has reference semantics in Algorithm 3.1.

Theorem 3.4. *$\mathcal{ALG}_{exact}$ computes a rectangular strategy profile region's optimal exact implementation cost in time*

$$\mathcal{O}\left(|X|^2 \max_{i \in N}(|O_i|^{n|X_i^* \setminus O_i|-1}) + n|O| \max_{i \in N}(|O_i|^{n|X_i^* \setminus O_i|})\right).$$

Proof. $\mathcal{ALG}_{exact}$ is correct as it checks all possible relations in the relation set M^V recursively by calling the subroutine *ExactK* in Line 7. Therefore, it must find the relation set which corresponds to an implementation with optimal cost. For each relation set, an implementation is computed that guarantees that all strategies outside O are dominated, and whose cost are optimal with respect to the relations (Line 5). The input requirement on the target region O guarantees that for any $i \in N$, $X_{-i} \setminus O_{-i}$ is nonempty. Thus, no strategy $o_i \in O_i$ can be dominated since the payoff for a profile $(o_i, \bar{o}_{-i})$ with $\bar{o}_{-i} \in X_{-i} \setminus O_{-i}$ equals ∞ in the resulting game (Line 2). Hence, the payments V are indeed an exact implementation of O, and optimal with respect to the corresponding relation set when returning the incurred cost in Line 19.

It remains to prove the algorithm's runtime. Computing the non-dominated region X^* by checking for each strategy whether it is dominated takes time $\sum_{i=1}^{n} \binom{|X_i|}{2} |X_{-i}| = O(n|X|^2)$. The complexity of this computation asymptotically dominates the runtime required by Lines 1 and 2. We

Algorithm 3.1 Exact k-Implementation ($\mathcal{ALG}_{exact}$)

Input: Game G, rectangular region O with $O_{-i} \subsetneq X_{-i} \forall\ i$
Output: $k^*(O)$

$V_i(x) := 0$, $W_i(x) := 0\ \forall x \in X\ ,\ i \in N$;
$V_i(o_i, \bar{o}_{-i}) := \infty\ \forall i \in N,\ o_i \in O_i\ ,\ \bar{o}_{-i} \in X_{-i} \backslash O_{-i}$;
compute X^*;
return ExactK(V, n);

***ExactK(V, i)*:**
Input: payments V, current player i
Output: $k^*(O)$ for $G(V)$

if $|X_i^*(V) \backslash O_i| > 0$ **then**
  $s :=$ any strategy in $X_i^*(V) \backslash O_i$; $k_{best} := \infty$;
  for all $o_i \in O_i$ **do**
    for all $o_{-i} \in O_{-i}$ **do**
      $W_i(o_i, o_{-i}) := \max(0, U_i(s, o_{-i}) - (U_i(o_i, o_{-i}) + V_i(o_i, o_{-i})))$;
    end for
    $k :=$ ExactK($V + W$, i);
    if $k < k_{best}$ **then**
      $k_{best} := k$;
    end if
    for all $o_{-i} \in O_{-i}$ **do**
      $W_i(o_i, o_{-i}) := 0$;
    end for
  end for
  return k_{best};
else if $i > 1$ **then**
  return $ExactK(V, i-1)$;
else
  return $\max_{o \in O} \sum_i V_i(o)$;
end if

next examine the complexity of subroutine *ExactK*. Computing Line 1 costs $|X|^2$, the two for-loops in Lines 3 and 4 are executed $|O|$ times, and *ExactK* is called $|O_i|$ times (Line 6). Hence, we derive the following (asymptotic) recursive equations for the runtime $T_i(\ell)$ for $ExactK(V, i)$ if i has yet ℓ strategies to dominate:

$$T_i(\ell) = \begin{cases} |X|^2 + |O| + |O_i|T_i(\ell - 1) & \text{if } (0 < \ell < |X_i^* \setminus O_i|) \wedge (i \in N) \\ T_{i-1}(|X_{i-1}^* \setminus O_{i-1}|) & \text{if } \ell = 0 \wedge i \in N \\ n|O| & \text{if } \ell = 0 \wedge i = 0 \end{cases}$$

For $\ell_i = |X_i^* \setminus O_i|$, we obtain $T_i(\ell_i) = |O_i|^{\ell_i - 1}|X|^2 + |O_i|^{\ell_i} T_{i-1}(\ell_{i-1})$ if $i > 1$. Let $a_i = |O_i|^{\ell_i - 1}|X|^2, b_i = |O_i|^{\ell_i}$ and $a = \max_{i \in N} a_i, b = \max_{i \in N} b_i$; hence

$$\begin{aligned} T_i(\ell_i) &= a_i + b_i T_{i-1}(\ell_{i-1}) \\ &= a \left[\sum_{j=1}^{i} \prod_{k=1}^{j-1} b_k \right] + \left[\prod_{k=1}^{i} b_k \right] T_1(0) \\ &= a \sum_{j=1}^{i} b^{j-1} + b^i n|O| \\ &= a b^{i-1} + b^i n|O| \end{aligned}$$

and the claim follows. □

Note that $\mathcal{ALG}_{exact}$ has a large time complexity. In fact, a faster algorithm for this problem, called *Optimal Perturbation*, has been proposed in [66]. However, the Optimal Perturbation algorithm does not always compute the correct cost, as we have shown in [34]. In contrast, while analyzing games for which the Optimal Perturbation Algorithm fails, we observed that the problem is seemingly, inherently hard. We conjecture that deciding whether a k-exact implementation exists is NP-hard. Although we did not succeed in proving NP-hardness we have reason to believe so as we can show the arguably easier, and closely related problem of finding the exact uniform implementation cost of a strategy region to be NP-hard (Theorem 3.8).

Conjecture 3.5. *Finding an optimal exact implementation of a strategy region is* NP*-hard.*

The study of exact implementation cost was introduced by Monderer and Tennenholtz [66] primarily because it seems easier to compute the exact implementation cost of a region O than its non-exact cost. Computing the non-exact cost of O implicitly computes at least the optimal subregion's exact cost, potentially the exact cost of all subsets of O since the algorithm has

to discover that no other subregion has lower implementation cost. Unfortunately, although we experienced that computing exact cost is computationally easier than computing non-exact cost, it still seems infeasible to do so in polynomial time.

3.1.2 Non-Exact Implementation

In contrast to exact implementations, where the complete set of strategy profiles O must be non-dominated, the additional payments in non-exact implementations only have to ensure that a *subset* of O is the newly non-dominated region. Obviously, it matters which subset this is. Knowing that a subset $O' \subseteq O$ bears optimal cost, we could find $k(O)$ by computing $k^*(O')$. As we conjectured that computing exact cost is in NP we get the following:

Conjecture 3.6. *Finding an optimal implementation of a strategy region is* NP*-hard.*

Apart from the fact that finding an optimal implementation includes solving the—believed to be NP-hard—optimal exact implementation cost problem for at least one subregion of O, finding this subregion might also be NP-hard even if the exact implementation cost problem shows to be in P since there are exponentially many possible subregions. In fact, a reduction from the SAT problem is presented in [66]. The authors show how to construct a 2-person game in polynomial time given a CNF formula such that the game has a 2-implementation if and only if the formula has a satisfying assignment. However, their proof is not correct: While there indeed exists a 2-implementation for every satisfiable formula, it can be shown that 2-implementations also exist for non-satisfiable formulas. E.g., strategy profiles $(x_i, x_i) \in O$ are always 1-implementable. Unfortunately, we were not able to correct their proof. Moreover, we conjecture the problem to be NP-hard, i.e., we believe that no algorithm can do much better than performing a brute force computation of the exact implementation cost (cf. Algorithm 3.1) of all possible subsets, unless NP = P. Note that we give a reduction from SET COVER for the uniform implementation cost in the following section.

3.2 Uniform Implementation Cost

In the *uniform* model, we assume a scenario of imperfect knowledge, i.e., a player i is aware of all strategy spaces X, but the player only knows her own utilities U_i rather than all players' utilities U. Without having any indication of what the others will play, we presume a player chooses one of the non-dominated strategies uniformly at random. This is opposed to a perfect knowledge scenario where a player could still take into account

rational choices of the other players and thus eliminate also certain non-dominated strategies. As we have seen in the DBoS example in Figure 2.2, if Jim is certain that Judy will choose to go to the classical concert anyway, it does not make sense for him to choose *football* even though *football* is a non-dominated strategy. The worst-case model accounts for such rational, secondary reasoning based on perfect knowledge by assuming the most costly outcome. It thus provides a lower bound on the power of a mechanism designer. With imperfect knowledge, reasoning based on the payoffs of others is infeasible, and mixing among the non-dominated pure strategies uniformly at random seems a natural strategy. As a consequence, the uniform behavior yields a uniform probability distribution over the non-dominated strategy profiles. All strategy profiles in the non-dominated region $X^*(V)$ are played with the same probability, and the mechanism designer can calculate an expected implementation cost for payments V.

Note that the assumption of a uniform distribution can be modeled either on the level of the players or on the level of the mechanism designer. We either presume the players to adopt a certain behavior or we presume the mechanism designer to make some assumptions on the players' behavior. The argument supporting the uniform assumption stated above reasons on the level of the players' behavior. To reason on the latter level we could think of the mechanism designer as willing to take risks and presume her to anticipate uniform rather than worst case costs regardless of the scope of information available to the players.

We define the uniform cost of an implementation V as the *average* of all strategy profiles' possible cost in $X^*(V)$.

Definition 3.7 (Uniform Cost and Exact Uniform Cost)**.** *A strategy profile set O has* uniform implementation cost

$$k_{UNI}(O) := \inf_{V \in \mathcal{V}(O)} \left\{ \operatorname*{avg}_{z \in X^*(V)} V(z) \right\},$$

where avg *is defined as* $\operatorname{avg}_{x \in X} f(x) := 1/|X| \cdot \sum_{x \in X} f(x)$*. A strategy profile set O has* exact uniform implementation cost

$$k^*_{UNI}(O) := \inf_{V \in \mathcal{V}^*(O)} \left\{ \operatorname*{avg}_{z \in X^*(V)} V(z) \right\}.$$

Similarly to the worst-case implementation cost, we have to define the uniform cost by an infimum over all implementations of region O, as for some instances of games and target regions, ϵ-payments might be necessary to implement O. We thus again consider payments V an *optimal implementation* of O if $\operatorname{avg}_{z \in X^*(V)} V(z) = k_{UNI}(O)$ for games and target regions where such a V exists or if $\operatorname{avg}_{z \in X^*(V)} V(z) = k_{UNI}(O) + \epsilon$ for arbitrarily small

$\epsilon > 0$ in the general case. Note that it also holds that if $O_{-i} \subsetneq X_{-i} \ \forall\, i \in N$ then there are optimal implementations that reach $k_{UNI}(O)$ exactly.

As promised, we show in the following that it is NP-hard to find implementations that yield optimal cost. Moreover, it is NP-hard to compute the uniform implementation cost for both the non-exact and the exact case. We devise game configurations which reduce SET COVER to the problem of finding an implementation of a strategy profile set with optimal uniform cost.

Theorem 3.8. *In games with at least two players, the problem of finding an* exact implementation *of a strategy profile set yielding optimal uniform cost is* NP*-hard.*

Proof. For a given universe $\mathcal{U}$ of l elements $\{e_1, e_2, \ldots, e_l\}$ and m subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, with $S_i \subset \mathcal{U}$, SET COVER is the problem of finding the minimal collection of S_i's which contains each element $e_i \in \mathcal{U}$. We assume without loss of generality that $\nexists (i \neq j) : S_i \subseteq S_j$, and that the SET COVER problem has a solution. Given a SET COVER problem instance $SC = (\mathcal{U}, \mathcal{S})$, we can efficiently construct a game $G = (N, X, U)$ with 2 players, $N = \{1, 2\}$, and strategy sets

$$\begin{aligned} X_1 &= \{e_1, e_2, \ldots, e_l, s_1, s_2, \ldots, s_m\}, \text{ and} \\ X_2 &= \{e_1, e_2, \ldots, e_l, d, r\}. \end{aligned}$$

Each strategy e_j corresponds to an element $e_j \in \mathcal{U}$, and each strategy s_j corresponds to a set S_j. Player 1's payoff function U_1 is defined as follows:

$$\begin{aligned} U_1(e_i, e_j) &:= \begin{cases} m+1 & \text{if } i = j, \\ 0 & \text{otherwise,} \end{cases} \\ U_1(s_i, e_j) &:= \begin{cases} m+1 & \text{if } e_j \in S_i, \\ 0 & \text{otherwise,} \end{cases} \\ U_1(e_i, d) &:= 1, \\ U_1(s_i, d) &:= 0, \\ U_1(x_1, r) &:= 0 \ \ \forall\, x_1 \in X_1. \end{aligned}$$

Player 2 has a payoff of 0 when playing r and 1 otherwise. See Figure 3.4 for an example. In such games, strategies e_j are not dominated for Player 1 because in column d, it holds that $U_1(e_j, d) > U_1(s_i, d)$ for all $i \in \{1, \ldots, m\}$. The set O we would like to implement is

$$O = \{(x_1, x_2) \mid x_1 = s_i \wedge (x_2 = e_i \vee x_2 = d)\}.$$

	e_1	e_2	e_3	e_4	e_5	d	r
e_1	5	0	0	0	0	1	0
e_2	0	5	0	0	0	1	0
e_3	0	0	5	0	0	1	0
e_4	0	0	0	5	0	1	0
e_5	0	0	0	0	5	1	0
s_1	5	0	0	5	0	0	0
s_2	0	5	0	5	0	0	0
s_3	0	5	5	0	5	0	0
s_4	5	5	5	0	0	0	0

Figure 3.4: *Payoff matrix for Player 1 in a game which reduces the SET COVER problem instance $SC = (\mathcal{U}, \mathcal{S})$ where $\mathcal{U} = \{e_1, e_2, e_3, e_4, e_5\}$, $\mathcal{S} = \{S_1, S_2, S_3, S_4\}$, $S_1 = \{e_1, e_4\}, S_2 = \{e_2, e_4\}, S_3 = \{e_2, e_3, e_5\}, S_4 = \{e_1, e_2, e_3\}$ to the problem of computing $k^*_{UNI}(O)$. The optimal exact implementation V of O in this sample game adds a payment V_1 of 1 to the strategy profiles (s_1, d) and (s_3, d), implying that the two sets S_1 and S_3 cover $\mathcal{U}$ optimally.*

We will now show that an optimal exact implementation V of region O has positive payments inside O only in column d, and zero payments otherwise.

By setting $V_1(s_i, d)$ to 1 and $V_1(s_i, r) > 0$ strategy s_i dominates all strategies e_i that correspond to an element in S_i. Hence, since the SET COVER instance is guaranteed to have a solution, a valid, potentially suboptimal exact implementation can be found by setting $V_1(s_i, d) := 1$ and $V_1(s_i, r) > 0$ for all s_i: all strategies e_j are dominated and V implements O exactly with uniform cost $\text{avg}_{o \in O} V(o) = m/|O|$. No payments need to be offered to Player 2 as her only dominated strategy is already r, as desired. If an implementation had a positive payment for Player 1 for any strategy profile of the form (s_i, e_j), it would cost at least $m + 1$ to have an effect. However, a positive payment greater than m yields a larger uniform cost. Thus, an optimal V sets all payments $V_1(s_i, e_j)$ to 0.

By setting $V_1(s_i, d)$ to 1, s_i dominates the strategies e_j that correspond to the elements in S_i, due to our construction. For a strategy profile (s_i, d) it only makes sense to offer a payment of 1 or 0: A larger payment would

increase the cost, a payment smaller than 1 has no effect. Thus, an optimal implementation minimizes the number of 1s in column d while dominating all Player 1-strategies e_i. This can be achieved by selecting those rows s_i, i.e., setting $V_1(s_i, d) := 1$, that form a minimal covering set and as such all strategies e_i of player 1 are dominated at minimal cost.

Consequently, an optimal solution Q for the SET COVER problem can be derived from an optimal exact implementation V of O in the corresponding game by setting $Q := \{S_i \mid V_1(s_i, d) = 1\}$.

The shown reduction can be generalized for $n > 2$ by adding players with only one strategy and zero payoffs in all strategy profiles. □

Corollary 3.9. *In games with at least two players, the problem of computing the* exact uniform implementation cost $k^*_{UNI}(O)$ *of a strategy profile set O is* NP*-hard.*

Proof. We prove the corollary by giving a reduction from the SET COVER *decision* problem: Given a universe $\mathcal{U}$, sets $\mathcal{S}$, and an integer k, decide whether there is a set covering of size k or less. From any given SET COVER instance, we construct a game as described in the proof of Theorem 3.8. If we know $k^*_{UNI}(O)$ in that game we can also decide whether there is a set covering, namely iff $k \geq k^*_{UNI}(O) \cdot (l+1)m$. Thus, if $k^*_{UNI}(O)$ could be computed in polynomial time then SET COVER could also be decided in polynomial time, which is a contradiction, unless $\mathsf{P} = \mathsf{NP}$. □

Theorem 3.10. *In games with at least three players, the problem of finding a* non-exact implementation *of a strategy profile set yielding optimal uniform cost is* NP*-hard.*

Proof. We give a similar reduction of SET COVER to the problem of computing $k_{UNI}(O)$ by extending the setup we used for proving the exact case. We add a third player and show NP-hardness for $n = 3$ first and indicate how the reduction can be adapted for games with $n > 3$. Given a SET COVER problem instance $SC = (\mathcal{U}, \mathcal{S})$, we can construct a game $G = (N, X, U)$ where $N = \{1, 2, 3\}$, and the strategies for the players are

$$\begin{aligned} X_1 &= \{e_1, e_2, \ldots, e_l, s_1, s_2, \ldots, s_m\}, \\ X_2 &= \{e_1, e_2, \ldots, e_l, s_1, s_2, \ldots, s_m, d, r\}, \text{ and} \\ X_3 &= \{a, b\}. \end{aligned}$$

Again, each strategy e_j corresponds to an element $e_j \in \mathcal{U}$, and each strategy s_j corresponds to a set S_j. In the following, we use '_' in profile vectors as a

placeholder for any possible strategy. Player 1's payoff function U_1 is defined as follows:

$$\begin{aligned}
U_1(e_i, e_j, _) &:= \begin{cases} (m+l)^2 & \text{if } i = j, \\ 0 & \text{otherwise} \end{cases} \\
U_1(e_i, s_j, _) &:= 0, \\
U_1(s_i, e_j, _) &:= \begin{cases} (m+l)^2 & \text{if } e_j \in S_i, \\ 0 & \text{otherwise,} \end{cases} \\
U_1(s_i, s_j, _) &:= \begin{cases} 0 & \text{if } i = j, \\ (m+l)^2 & \text{otherwise} \end{cases} \\
U_1(e_i, d, _) &:= 1 \\
U_1(s_i, d, _) &:= 0 \\
U_1(_, r, _) &:= 0
\end{aligned}$$

Player 2 has a payoff of $(m+l)^2$ for any strategy profile of the form $(s_i, s_i, \_)$ and 0 for any other strategy profile. Player 3 has a payoff of $m + l + 2$ for strategy profiles of the form (s_i, s_i, b), a payoff of 2 for profiles (s_i, e_i, b) and profiles $(s_i, s_j, b), i \neq j$, and a payoff of 0 for any other profile. The set O we would like to implement is

$$O := \{(x_1, x_2, x_3) \mid x_1 = s_i \ \wedge \ (x_2 = e_i \ \vee \ x_2 = s_i \ \vee \ x_2 = d) \ \wedge \ (x_3 = a)\}.$$

See Figure 3.5 for an example. First, note the fact that any implementation of O will have $V_3(o_1, o_2, a) \geq U_3(o_1, o_2, b)$, in order to leave player 3 no advantage playing b instead of a. In fact, setting $V_3(o_1, o_2, a) = U_3(o_1, o_2, b)$ suffices. (Setting any $V_3(a, \bar{o}_{-3}) > U_3(b, \bar{o}_{-3})$ where $\bar{o}_{-3}$ is outside O lets Player 3 choose strategy a.) Also note that for Player 2, O_2 can be made non-dominated without offering any payments inside O, e.g., set $V_2(e_i, e_j, \_) := 1$ and $V_2(e_i, d, \_) := 1$.

Similar to the proof of the exact case, we claim that if and only if Q is an optimal solution for a SET COVER problem then there exists an optimal exact implementation V of O in the corresponding game: implementation V *selects* a row s_i, i.e., $V_1(s_i, d, a) = 1$, if $S_i \in Q$ and does not select s_i, i.e., $V_1(s_i, d, a) = 0$, otherwise. All other payments V_1 inside O are 0. Player 2's payments $V_2(o)$ are 0 for all $o \in O$ and Player 3's payments are set to $V_3(o_1, o_2, a) = U_3(o_1, o_2, b)$. A selected row s_i contributes

$$cost_{s_i} = \frac{3(l+m)+1}{l+m+1}.$$

A non-selected row s_j contributes cost

$$cost_{s_j} = \frac{3(l+m)}{l+m+1} < cost_{s_i}.$$

	e_1	e_2	e_3	e_4	e_5	s_1	s_2	s_3	s_4	d	r
e_1	81 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	1 0	0 0
e_2	0 0	81 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	1 0	0 0
e_3	0 0	0 0	81 0	0 0	0 0	0 0	0 0	0 0	0 0	1 0	0 0
e_4	0 0	0 0	0 0	81 0	0 0	0 0	0 0	0 0	0 0	1 0	0 0
e_5	0 0	0 0	0 0	0 0	81 0	0 0	0 0	0 0	0 0	1 0	0 0
s_1	81 0	0 0	0 0	81 0	0 0	0 81	81 0	81 0	81 0	0 0	0 0
s_2	0 0	81 0	0 0	81 0	0 0	81 0	0 81	81 0	81 0	0 0	0 0
s_3	0 0	81 0	81 0	0 0	81 0	81 0	81 0	0 81	81 0	0 0	0 0
s_4	81 0	81 0	81 0	0 0	0 0	81 0	81 0	81 0	0 81	0 0	0 0

	e_1	e_2	e_3	e_4	e_5	s_1	s_2	s_3	s_4	d	r
e_1	0	0	0	0	0	0	0	0	0	0	0
e_2	0	0	0	0	0	0	0	0	0	0	0
e_3	0	0	0	0	0	0	0	0	0	0	0
e_4	0	0	0	0	0	0	0	0	0	0	0
e_5	0	0	0	0	0	0	0	0	0	0	0
s_1	2	2	2	2	2	11	2	2	2	0	0
s_2	2	2	2	2	2	2	11	2	2	0	0
s_3	2	2	2	2	2	2	2	11	2	0	0
s_4	2	2	2	2	2	2	2	2	11	0	0

Figure 3.5: *Payoff matrix for Player 1 and Player 2 given Player 3 chooses a and payoff matrix for Player 3 when she plays strategy b in a game that reduces a SET COVER instance $SC = (\mathcal{U}, \mathcal{S})$ where $\mathcal{U} = \{e_1, e_2, e_3, e_4, e_5\}$, $\mathcal{S} = \{S_1, S_2, S_3, S_4\}$, $S_1 = \{e_1, e_4\}$, $S_2 = \{e_2, e_4\}$, $S_3 = \{e_2, e_3, e_5\}$, $S_4 = \{e_1, e_2, e_3\}$ to the problem of computing $k_{UNI}(O)$. Every implementation V of O in this game needs to add any positive payment in the second matrix to V_3, i.e. $V_3(x_1, x_2, a) = U_3(x_1, x_2, b)$, in order to convince player 3 of playing strategy a. An optimal implementation adds a payment V_1 of 1 to the strategy profiles (s_1, d, a) and (s_3, d, a), implying that the two sets S_1 and S_3 cover $\mathcal{U}$ optimally in the corresponding SET COVER problem.*

Thus, including non-selected rows in $X^*(V)$ can be profitable. Selecting all rows s_i yields a correct implementation of O with uniform cost of

$$\underset{i=1}{\overset{m}{\text{avg}}}\, cost_{s_i} = \frac{3(l+m)+1}{l+m+1} < 3.$$

Hence, it must hold that $k^*(O) < 3$.

In fact, the game's payoffs are chosen such that it is not worth implementing any set smaller than O. We show for every proper rectangular subset Φ of O that its exact uniform implementation cost is strictly larger than that of O, i.e., $k^*_{UNI}(\Phi) > k^*_{UNI}(O)$. Assume for the sake of contradiction that there exists such a set $\Phi \subsetneq O$ yielding lower cost. Let $\alpha \leq m$ be the number of strategies in Φ to implement for Player 1, $\beta \leq l$ the number of strategies e_i and $\gamma \leq m$ the number of strategies s_j to implement for Player 2. Note that implementing Player 2's strategy d is profitable if $\beta + \gamma > 0$, as it adds α to the denominator and at most α to the numerator of the implementation cost of sets without d. Consequently, there are three cases of Φ to consider:

(i) Φ with $\beta > 0, \gamma = 0$: The costs add up to

$$\begin{aligned} k^*_{UNI}(\Phi) &= \sum_{o \in O} \frac{V_1(o) + V_2(o) + V_3(o)}{|O|} \\ &\geq \frac{1 + (m+l)^2 + 2\alpha\beta}{\alpha(\beta+1)} \\ &> 3 > k^*(O), \end{aligned}$$

where the last inequality holds since $\alpha \leq m, \beta \leq l$.

(ii) Φ with $\beta = 0, \gamma > 0$: The aggregated cost is at least

$$k^*_{UNI}(\Phi) \geq \frac{1 + \alpha(m+l) + 2\alpha\gamma}{\alpha(\gamma+1)} > 3 > k^*(O).$$

(iii) Φ with $\beta > 0, \gamma > 0$: Assume there are κ sets necessary to cover U. Hence the sum of the payments in column d is at least κ. In this case, the cost amounts to

$$\begin{aligned} k^*_{UNI}(\Phi) &\geq \frac{\kappa + \alpha(m+l) + 2\alpha(\beta+\gamma)}{\alpha(\beta+\gamma+1)} \\ &= 2 + \frac{m + l - 2 + \kappa/\alpha}{\beta+\gamma+1}. \end{aligned}$$

Note that if we fix all other parameters, the last term in the above equation decreases as any of the parameters α, γ, or β increases. By

setting the parameters to $\alpha = \gamma = m$ and $\beta = l$, which is an invalid assignment as at least one of them should be smaller to get a proper subset of O, the term evaluates to $k^*(O)$:

$$\begin{aligned} 2 + \frac{m + l - 2 + \kappa/\alpha}{\beta + \gamma + 1} &= 2 + \frac{m + l - 2 + \kappa/m}{l + m + 1} \\ &= \frac{3(m + l) + \kappa/m}{l + m + 1} \\ &= \frac{\kappa\,(3\,(m + l) + 1) + (m - \kappa)\,(3\,(m + l))}{m(l + m + 1)} \\ &= k^*(O). \end{aligned}$$

Hence, for any valid assignment, the cost are higher than $k^*(O)$. This is a contradiction.

Therefore, an optimal implementation V implements O exactly, i.e., $X^*(V) = O$ with the inalienable payments to Player 3 and a minimal number of 1-payments to Player 1 for strategy profiles (s_i, d, a) such that every e_j is dominated by at least one s_i. The number of 1-payments is minimal if the selected rows correspond to a minimal covering set, and the claim follows.

Note that a similar SET COVER reduction can be found for games with more than three players. Simply add players to the described 3-player game with only one strategy. □

With the game construction from the above proof, it is now possible to reduce the SET COVER decision problem to the problem of computing non-exact uniform cost in the same way as for the exact case (Corollary 3.9).

Corollary 3.11. *In games with at least three players, the problem of finding a strategy profile set's* non-exact uniform implementation cost *is* NP*-hard.*

Due to the nature of the reduction, the inapproximability results of SET COVER ([7, 40]) carry over to our problem.

Theorem 3.12. *Unless* P=NP*, the best approximation ratio that a polynomial-time algorithm can achieve is* $\Omega\left(n \max_i\{\log |X_i^* \setminus O_i|\}\right)$ *for both the exact and non-exact implementation cost.*

Proof. Exact Case. In order to prove the claim, a reduction similar to the one in the proof of Theorem 3.8 can be applied. Consider again a SET COVER instance with a universe $\mathcal{U}$ of l elements $\{e_1, e_2, \ldots, e_l\}$ and m subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, with $S_j \subset \mathcal{U}$. We construct a game $G = (N, X, U)$ with n players $N = \{1, \ldots, n\}$, where

$$\begin{aligned} X_i &= \{e_1, e_2, \ldots, e_l, s_1, s_2, \ldots, s_m\} \;\; \forall\, i \in \{1, \ldots, n-1\}, \\ X_n &= \{e_1, e_2, \ldots, e_l, d, r\}. \end{aligned}$$

Again, each strategy e_j corresponds to an element $e_j \in \mathcal{U}$, and each strategy s_j corresponds to a set S_j. The payoff functions are defined as follows: Player n has a payoff of 0 when playing r and 1 otherwise, independently of all other players' choices. For all other players i, $i \in \{1, \ldots, n-1\}$ the payoff function U_i depends only on the strategies chosen by Player n and Player i herself. Thus, in the following description of U_i we only use two parameters, the strategy chosen by Player i and the strategy chosen by Player n, in this order.

$$\begin{aligned}
U_i(e_j, e_k) &:= \begin{cases} m+1 & \text{if } j = k \\ 0 & \text{otherwise} \end{cases} \\
U_i(s_j, e_k) &:= \begin{cases} m+1 & \text{if } e_k \in S_j \\ 0 & \text{otherwise} \end{cases} \\
U_i(e_j, d) &:= 1 \\
U_i(s_j, d) &:= 0 \\
U_i(e_j, r) = U_i(s_j, r) &:= 0
\end{aligned}$$

We ask for an implementation of set O where Player i, for $i \in \{1, \ldots, n-1\}$, plays any strategy in $\{s_1, \ldots, s_m\}$, and Player n plays any strategy in $\{e_1, \ldots, e_l, d\}$.

Due to the independence of the players' payoffs, the situation is similar to the example in Figure 3.4, and a SET COVER instance has to be solved for each player $i \in \{1, \ldots, n-1\}$. According to the well-known inapproximability results for SET COVER, no polynomial time algorithm exists which achieves a better approximation ratio than $\Omega(\log |X_i^* \setminus O_i|)$ for each player i, unless $\mathsf{P} = \mathsf{NP}$, and the claim follows.

Non-Exact Case. We use the inapproximability results for SET COVER again. Concretely, we assume a set of $n = 3k$ players for an arbitrary constant $k \in \mathbb{N}$ and make k groups of three players each. The payoffs of the three players in each group are the same as described in the proof of Theorem 3.10 for the non-exact case, independently of all other players' payoffs. Hence, SET COVER has to be solved for $n/3$ players. □

As a remark, let us reconsider the NP-hardness conjecture of the worst-case cost in the first part of this chapter. The uniform implementation cost is based on the assumption that players choose one of the non-dominated strategies uniformly at random such that an equal probability mass is assigned to each strategy profile in the non-dominated region. I.e., the implementation cost depends on the aggregate cost over the entire profile set. This enables us to construct a game corresponding to a set cover problem instance. In the worst-case model however, individual strategy profiles need to be taken into

Implementation Cost	Complexity	Properties
Uniform	NP-hard SINGLETON $\mathcal{O}\left(n \cdot \sum_i |X_i|\right)$ ZERO $\mathcal{O}\left(n|X|^2\right)$	NE 0-implementable
Worst-case	conjecture: NP-hard SINGLETON $\mathcal{O}\left(n \cdot \sum_i |X_i|\right)$ ZERO $\mathcal{O}\left(n|X|^2\right)$	NE 0-implementable

Figure 3.6: *Complexity results for the computation of the implementation cost. Unless stated otherwise, complexities refer to the problem of computing any strategy profile's implementation cost.* SINGLETON *indicates the complexity of computing a singleton's implementation cost.* ZERO *indicates the complexity of deciding for a strategy profile region whether it is 0-implementable. The complexities of* ZERO *are results from our earlier work [34].*

account and payment differences between strategy profiles matter. Put differently, the worst-case model assumes less about the players' behavior than the uniform model. We believe that this renders the minimal implementation cost problem only harder.

Chapter 4

Leverage

With rational players, mechanism designers can implement any desired outcomes if they offer high enough payments. As we have seen in the example with Al Capone and the two arrested bank robbers, an implementation can be beneficial for the mechanism designer if she can influence the game to her favor without spending too much. The natural question that arises from this insight is for which games it actually makes sense to take influence at all, and which behavior the mechanism designer should implement in order to maximize her own utility.

To answer this question we need to model the mechanism designer herself, and define the interests she has in the outcome of the game. In this chapter, we examine two diametrically opposed models of an interested third party. The first one is *benevolent* towards the participants of the game, and the other one *malicious*. While the former is interested in increasing a game's social gain, the latter seeks to minimize the players' welfare.[1] We define a measure that indicates whether the mechanism of implementation enables them to modify a game in a favorable way such that their gain exceeds the cost of the manipulation. We call these measures the *leverage* and *malicious leverage*, respectively. In the following, we will often write "(malicious) leverage" signifying both leverage and malicious leverage. As the concept of leverage depends on the implementation cost, we examine the leverage of games in both the *worst-case* and the *uniform* cost model.

The worst-case leverage constitutes a lower bound on the mechanism designer's influence: We assume that without the additional payments, the players choose a non-dominated strategy profile in the original game where

[1] Note that our terminology assumes the perspective of the players, i.e., if a mechanism designer acts contrary to their utilities, it is called "malicious". Depending on the game, a malicious mechanism designer's goal to punish the players can be morally upright (cf. the commander-in-chief in the extended prisoner's dilemma example).

the social gain is maximal, while in the modified game, they select a strategy profile among the newly non-dominated profiles where the difference between the social gain and the mechanism designer's cost is minimized. The value of the leverage is given by the net social gain achieved by this implementation minus the amount of money the mechanism designer had to spend.

Definition 4.1 (Worst-Case Leverage)**.** *The* leverage *of a strategy profile set O is $LEV(O) := \max\{0, lev(O)\}$, where*

$$lev(O) := \sup_{V \in \mathcal{V}(O)} \left\{ \min_{z \in X^*(V)} \{U(z) - V(z)\} \right\} - \max_{x^* \in X^*} U(x^*).$$

Here $U(z)$ refers to the total utility of the players in profile z and $V(z)$ is the total amount of payments.

For malicious mechanism designers we have to invert signs, swap max and min, and replace the supremum with an infimum. Moreover, the payments made by the mechanism designer have to be subtracted twice, because for a malicious mechanism designer, the money received by the players are considered a loss.

Definition 4.2 (Malicious Worst-Case Leverage)**.** *The* malicious leverage *of a strategy profile set O is $MLEV(O) := \max\{0, mlev(O)\}$, where*

$$mlev(O) := \min_{x^* \in X^*} U(x^*) - \inf_{V \in \mathcal{V}(O)} \left\{ \max_{z \in X^*(V)} \{U(z) + 2V(z)\} \right\}.$$

Observe that according to our definitions, leverage values are always non-negative, as a mechanism designer has no incentive to manipulate a game if she will lose money. If the desired set consists only of one strategy profile z, i.e., $O = \{z\}$, we will speak of the *singleton* leverage. Similarly to the (worst-case) leverage, we define the uniform leverage.

Definition 4.3 (Uniform Leverage)**.** *The* uniform leverage *of a strategy profile set O is defined as $LEV_{UNI}(O) := \max\{0, lev_{UNI}(O)\}$, where*

$$lev_{UNI}(O) := \sup_{V \in \mathcal{V}(O)} \left\{ \operatorname*{avg}_{z \in X^*(V)} (U(z) - V(z)) \right\} - \operatorname*{avg}_{x^* \in X^*} U(x^*).$$

Definition 4.4 (Malicious Uniform Leverage)**.** *The* malicious uniform leverage *of a strategy profile set O is $MLEV_{UNI}(O) := \max\{0, mlev_{UNI}(O)\}$, where*

$$mlev_{UNI}(O) := \operatorname*{avg}_{x^* \in X^*} U(x^*) - \inf_{V \in \mathcal{V}(O)} \left\{ \operatorname*{avg}_{z \in X^*(V)} \{U(z) + 2V(z)\} \right\}.$$

We define the *exact (uniform) leverage* $LEV^*_{(UNI)}(O)$ and the *exact (uniform) malicious leverage* $MLEV^*_{(UNI)}(O)$ by simply changing $\mathcal{V}(O)$ to $\mathcal{V}^*(O)$ in the definition of $LEV_{(UNI)}(O)$ and in the definition of $MLEV_{(UNI)}(O)$. Thus, the exact (uniform) (malicious) leverage measures a the leverage of a set if the interested party may only promise payments which implement O exactly. Finally, the (uniform) (malicious) leverages of an entire game $G = (N, X, U)$ are defined as the (uniform) (malicious) leverages of X, e.g., $LEV(G) := LEV(X)$.

4.1 Worst-Case Leverage

We first study singleton implementations and then extend our investigations to profile sets.

4.1.1 Singletons

As we know which outcome is yielded by an implementation of a singleton z, namely z, the leverage of a singleton boils down to

$$LEV(z) = \max\left\{0, U(z) - k(z) - \max_{x^* \in X^*} U(x^*)\right\},$$

where cost $k(z)$ of a singleton z can be computed efficiently by the formula of Theorem 3.1. The malicious leverage of a singleton is given by

$$MLEV(z) = \max\left\{0, \min_{x^* \in X^*} U(x^*) - U(z) - 2k(z)\right\}.$$

In the following, we propose an algorithm for a mechanism designer seeking to implement a game's best singleton, i.e., the strategy profile with the highest singleton leverage. Dually, the algorithm finds the profile of the largest malicious leverage for a malicious designer. Algorithm 4.1 computes two arrays, LEV and $MLEV$, containing all singleton (malicious) leverage values within a strategy profile set O. By setting $O = X$, the algorithm computes all singleton (malicious) leverage values of a game.

Algorithm 4.1 initializes an array *lev* with the negative value of the original game's maximal social gain in the non-dominated set and an array *mlev* with the original game's minimal social gain. Next, it computes the set of non-dominated strategy profiles X^*; in order to do so, it checks, for each player and for each of her strategies, whether the strategy is dominated by any other strategy. In the remainder, the algorithm adds up the players' contributions to the (malicious) leverage values for each player and strategy profile. In any field z of the leverage array *lev*, we add the amount that Player i would contribute to the social gain if z was played and subtract the cost

Algorithm 4.1 Singleton (Malicious) Leverage

Input: Game G, set $O \subseteq X$
Output: LEV and $MLEV$

compute X^*;
for all strategy profiles $x \in O$ **do**
  $lev[x] := -\max_{x^* \in X^*} U(x^*)$;
  $mlev[x] := \min_{x^* \in X^*} U(x^*)$;
end for
for all Players $i \in N$ **do**
  for all $x_{-i} \in O_{-i}$ **do**
    $m := \max_{x_i \in X_i} U_i(x_i, x_{-i})$;
    for all strategies $z_i \in O_i$ **do**
      $lev[z_i, x_{-i}] \mathrel{+}= 2 \cdot U_i(z_i, x_{-i}) - m$;
      $mlev[z_i, x_{-i}] \mathrel{+}= U_i(z_i, x_{-i}) - 2m$;
    end for
  end for
end for
$\forall\, o \in O$: $LEV[o] := \max\{0, lev[o]\}$;
$\forall\, o \in O$: $MLEV[o] := \max\{0, mlev[o]\}$;
return LEV, $MLEV$;

we had to pay her, namely $U_i(z_i, x_{-i}) - (m - U_i(z_i, x_{-i})) = 2U_i(z_i, z_{-i}) - m$. For any entry z in the malicious leverage array $mlev$, we subtract player i's contribution to the social gain and also twice the amount the designer would have to pay if z is played since she loses money and the players gain it, $-U_i(z_i, x_{-i}) - 2(m - U_i(z_i, x_{-i})) = U_i(z_i, x_{-i}) - 2m$. Finally, lev and $mlev$ will contain the leverage and malicious leverage values of all singletons in O. By replacing the negative entries by zeros, the corresponding leverage arrays LEV and $MLEV$ are computed. The mechanism designer can then look up the best *non-negative* singleton by searching the maximal entry in the respective array.

Theorem 4.5. *For a game where every player has at least two strategies, Algorithm 4.1 computes the leverage and the malicious leverage values of all singletons within a strategy profile set O in $\mathcal{O}\big(n|X|^2\big)$ time.*

Proof. The correctness of Algorithm 4.1 follows directly from the application of the (malicious) singleton leverage formula. It remains to prove the time complexity. Finding the non-dominated strategies in the original game requires time $\sum_{i=1}^{n} \binom{|X_i|}{2} |X_{-i}| = O(n|X|^2)$, and finding the maximal or minimal *gain* amongst the possible outcomes X^* of the original game requires

time $\mathcal{O}(n\,|X|)$. The time for all other computations can be neglected asymptotically, and the claim follows. □

4.1.2 Strategy Profile Sets

For some strategy sets in games, implementing a contained singleton may yield an optimal implementation of that set. In some other cases, however, dominating all other strategy profiles in the set is expensive and unnecessary. We can indeed construct games where the difference between the best (malicious) set leverage and the best (malicious) singleton leverage gets arbitrarily large. Figure 4.1 depicts such a game. Therefore, a mechanism designer is bound to consider also larger sets, consisting of more than one strategy profile, in order to find a subset of X yielding the maximum (malicious) leverage.

Although many factors influence the leverage and the malicious leverage of a strategy profile set, there are some simple observations. First, if rational players already choose strategies such that the strategy profile with the highest social gain is non-dominated, a designer will not be able to ameliorate the outcome. Just as well, a malicious interested party will have nothing to corrupt if a game already yields the lowest social gain possible.

Fact 4.6. *If the social optimum* $x_{opt} := \arg\max_{x \in X} U(x)$ *of game* G *is in* X^* *then* $LEV(G) = 0$.

Fact 4.7. *If the social minimum* $x_{worst} := \arg\min_{x \in X} U(x)$ *of game* G *is in* X^* *then* $MLEV(G) = 0$.

As an example, a class of games where both properties of Facts 4.6 and 4.7 always hold are *equal sum games*, where every strategy profile yields the same gain, $U(x) = c\ \forall x \in X, c : constant$. Zero sum games are a special case of equal sum games where $c = 0$.

Fact 4.8 (Equal Sum Games)**.** *The leverage and the malicious leverage of any equal sum game* G *is zero:* $LEV(G) = 0$, $MLEV(G) = 0$.

A well-known example of a zero sum game is *Matching Pennies* (cf. Figure 4.2): Two players each secretly turn a penny to heads or tails. Then they reveal their choices simultaneously. If both coins show the same face Player 2 gives his penny to Player 1; if the pennies do not match Player 2 gets the pennies. Matching pennies features another interesting property: there is no dominated strategy. Therefore an interested party could only implement strategy profile sets O which are subsets of X^*. This raises the question whether a set $O \subseteq X^*$ can ever have a (malicious) leverage. We find that the answer is no, and moreover:

$$G =$$

α 0	1 0	γ 0	γ 0
1 0	α 0	γ 0	γ 0
$\alpha-1$ 0	0 $\alpha-1$	0 α	0 1
0 $\alpha-1$	$\alpha-1$ 0	0 1	0 α

$$V_O =$$

0 ∞	0 ∞	0 0	0 0
0 ∞	0 ∞	0 0	0 0
1 1	1 1	∞ 0	∞ 0
1 1	1 1	∞ 0	∞ 0

$$V_s =$$

0 ∞	0 0	0 0	0 0
0 ∞	0 0	0 0	0 0
0 α	0 0	0 0	0 0
α 1	0 α	∞ 0	∞ 0

Figure 4.1: *Two-player game where the set* O *bears the largest leverage. Implementation* V_O *yields* $X^*(V_O) = O$ *and* V_s *yields one non-dominated strategy profile. By offering payments* V_O*, a mechanism designer has cost* 2*, no matter which* $o \in O$ *will be played. However, she changes the social welfare to* $\alpha - 1$*. If* $\gamma < \alpha - 3$ *then* O *has a leverage of* $\alpha - 3 - \gamma$ *and if* $\gamma > \alpha + 3$ *then* O *has a malicious leverage of* $\gamma - \alpha - 3$*. Any singleton* $o \in O$ *has an implementation cost of* $\alpha + 1$*, yet the resulting leverage is* 0 *and the malicious leverage is* $\max\{0, \gamma - 3\alpha - 1\}$*. This demonstrates that a profile set* O*'s (malicious) leverage can be arbitrarily large, even if all contained singletons have a (malicious) leverage of zero.*

$$MP = \begin{array}{|c|c|}\hline 1,\ -1 & -1,\ 1 \\ \hline -1,\ 1 & 1,\ -1 \\ \hline \end{array}$$

Figure 4.2: *Game matrix G of* Matching Pennies. *It holds that $LEV(G) = MLEV(G) = 0$.*

Theorem 4.9. *The leverage and the malicious leverage of a strategy profile set $O \subseteq X$ intersecting with the set of non-dominated strategy profiles X^* equals* 0.

Proof. Assume that $|O \cap X^*| > 0$ and let $\hat{z}$ be a strategy profile in the intersection of O and X^*. Let $x^*_{max} := \arg\max_{x^* \in X^*} U(x^*)$ and $x^*_{min} := \arg\min_{x^* \in X^*} U(x^*)$. Let V_{LEV} be any implementation of O reaching $LEV(O) - \epsilon$ and V_{MLEV} any implementation of O reaching $MLEV(O) - \epsilon$. We get for the leverage

$$\begin{aligned} LEV(O) &= \max\left\{0, \min_{z \in X^*(V_{LEV})} \{U(z) - V_{LEV}(z)\} - U(x^*_{max}) + \epsilon\right\} \\ &\leq \max\{0, U(\hat{z}) - V_{LEV}(\hat{z}) - U(x^*_{max}) + \epsilon\} \\ &\leq \max\{0, U(x^*_{max}) - V_{LEV}(\hat{z}) - U(x^*_{max}) + \epsilon\} \\ &= \max\{0, -V_{LEV}(\hat{z}) + \epsilon\} \overset{\epsilon \to 0}{=} 0, \end{aligned}$$

and for the malicious leverage

$$\begin{aligned} MLEV(O) &= \max\left\{0, U(x^*_{min}) - \max_{z \in X^*(V_{MLEV})} (U(z) + 2V_{MLEV}(z)) + \epsilon\right\} \\ &\leq \max\{0, U(x^*_{min}) - U(\hat{z}) - 2V_{MLEV}(\hat{z}) + \epsilon\} \\ &\leq \max\{0, U(x^*_{min}) - U(x^*_{min}) - 2V_{MLEV}(\hat{z}) + \epsilon\} \\ &= \max\{0, -2V_{MLEV}(\hat{z}) + \epsilon\} \overset{\epsilon \to 0}{=} 0. \end{aligned}$$

□

In general, the problem of computing the (malicious) leverage of a strategy profile set seems computationally hard. It is related to the problem of computing a set's implementation cost, which we conjectured to be NP-hard in Section 3. Hence, we conjecture the problem of finding $LEV(O)$ or $MLEV(O)$ to be NP-hard in general as well. In fact, we can show that computing the (malicious) leverage has at least the same complexity as computing a set's cost.

Theorem 4.10. *If the computation of a set's implementation cost is* NP*-hard the computation of the (malicious) leverage of a strategy profile set is also* NP*-hard.*

Proof. We proceed by reducing the problem of computing $k(O)$ to the problem of computing $MLEV(O)$. The basic idea of the reduction is to modify the game so that all profiles inside O yield an equal social gain while conserving the original structure of the optimal payments. This is achieved by the following transformation of a problem instance (G, O) into a problem instance (G', O): add an additional Player $n+1$ with one strategy a and a payoff function $U_{n+1}(x)$ equal to $\gamma - U(x)$ if $x \in O$ and 0 otherwise. Thus, a strategy profile x in G' has social gain equal to γ if it is in O and equal to $U(x)$ in the original game if it is outside O. As Player $n+1$ has only one strategy available, G' has the same number of strategy profiles as G and furthermore, there will be no payments V_{n+1} needed in order to implement O. Player $(n+1)$'s payoffs impact only the profile gains, and they have no effect on how the other players decide their tactics. Theorem 4.9 allows us to assume that O and X^* do not intersect since $O \cap X^* \neq \emptyset$ implies $MLEV(O) = 0$. Thus, the non-dominated set in G' is the same as in G and it does not intersect with O.

By definition, the cost of a strategy profile set are $\inf_{V \in \mathcal{V}(O)} \{\max_{z \in X^*(V)} V(z)\}$ and from the malicious leverage's definition, we have

$$\inf_{V \in \mathcal{V}(O)} \left\{ \max_{z \in X^*(V)} \{U(z) + 2V(z)\} \right\} = \min_{x^* \in X^*} U(x^*) - mlev(O). \tag{4.1}$$

The left hand side of the latter equation almost matches the formula for $k(O)$ if not for the term $U(z)$ and a factor of 2. If we modify the given game as described and choose the additional players payoffs so that all strategy profiles inside $X^*(V) \subseteq O$ have a gain γ we can reduce O's cost to

$$\begin{aligned} k(O) &= \inf_{V \in \mathcal{V}(O)} \left\{ \max_{z \in X^*(V)} V(z) \right\} \\ &= \frac{1}{2} \left(\inf_{V \in \mathcal{V}(O)} \left\{ \max_{z \in X^*(V)} 2V(z) \right\} + \gamma - \gamma \right) \\ &= \frac{1}{2} \left(\inf_{V \in \mathcal{V}(O)} \left\{ \max_{z \in X^*(V)} (\gamma + 2V(z)) \right\} - \gamma \right) \\ &\overset{(4.1)}{=} \frac{1}{2} \left(\min_{x^* \in X^*} U(x^*) - mlev(O) - \gamma \right), \end{aligned}$$

where the last equation holds because X^* does not intersect with O, i.e., the term $\min_{x^* \in X^*} U(x^*)$ is equal in G and G'.

It remains to choose a good value for γ. Note that if we choose γ small enough, then we can guarantee that $mlev(O) \geq 0$ and thus $MLEV(O) = mlev(O)$, i.e., by adding small or large negative payoffs for the additional player inside O, it is worthwhile for a malicious mechanism designer to implement O. For instance, a feasible choice is

$$\gamma := -2n \cdot \max_{x \in X} \left\{ \max_{i \in N} U_i(x) \right\} + \min_{x^* \in X^*} U(x^*),$$

which yields

$$k(O) = -\frac{1}{2} MLEV(O) + n \cdot \max_{x \in X} \left\{ \max_{i \in N} U_i(x) \right\}.$$

Reducing the problem of computing $k(O)$ to $lev(O)$ is achieved by using the same game transformation with an additional player such that $\forall o \in O : U(o) = \gamma$, where

$$\gamma := n \cdot \max_{x \in X} \left\{ \max_{i \in N} \{U_i(x)\} \right\} + \max_{x^* \in X^*} \{U(x^*)\}.$$

We can then simplify the leverage formula to

$$\begin{aligned} lev(O) &= \gamma - k(O) - \max_{x^* \in X^*} U(x^*) \\ &= n \cdot \max_{x \in X} \left\{ \max_{i \in N} \{U_i(x)\} \right\} - k(O) + \epsilon \quad \geq \quad 0 \end{aligned}$$

and thus we find the cost $k(O)$ by computing

$$k(O) = n \cdot \max_{x \in X} \left\{ \max_{i \in N} \{U_i(x)\} \right\} - LEV(O).$$

□

The task of finding the leverage of a strategy profile set is computationally hard. Recall that we have to find an implementation V of O which maximizes the term $\min_{z \in X^*(V)} \{U(z) - V(z)\}$. Thus, there is at least one implementation $V \in \mathcal{V}(O)$ bearing O's leverage, or O's leverage up to an ϵ-term if the maximum does not exist. Since this V implements a subset of O exactly, it is also valid to compute O's leverage by searching among all subsets O' of O the one with the largest exact leverage $LEV^*(O')$.

For the sake of completeness, we provide the (potentially slow) Algorithm 4.2 that computes the exact leverage of a rectangular strategy profile. The algorithm makes use of the fact that if $X^*(V)$ has to be a subset of O, each strategy $\bar{o}_i \notin O_i$ must be dominated by at least one strategy o_i in the resulting game $G(V)$—a property that we observed and exploited before in

Algorithm 4.2 Exact Leverage

Input: Game G, rectangular set O with $O_{-i} \subsetneq X_{-i} \forall\ i$
Output: $LEV^*(O)$

$V_i(x) := 0$, $W_i(x) := 0\ \forall x \in X\ ,\ i \in N$;
$V_i(o_i, \bar{o}_{-i}) := \infty\ \forall i \in N,\ o_i \in O_i\ ,\ \bar{o}_{-i} \in X_{-i} \backslash O_{-i}$;
compute X_i^*;
return $\max\{0, ExactLev(V, n) - \max_{x^* \in X^*} U(x^*)\}$;

ExactLev(V ***,*** i ***)*:**
Input: payments V, current player i
Output: $lev^*(O)$ for $G(V)$

if $|X_i^*(V) \backslash O_i| > 0$ **then**
  $s :=$ any strategy in $X_i^*(V) \backslash O_i$; $lev_{best} := 0$;
  for all $o_i \in O_i$ **do**
    for all $o_{-i} \in O_{-i}$ **do**
      $W_i(o_i, o_{-i}) := \max\{0, U_i(s, o_{-i}) - (U_i(o_i, o_{-i}) + V_i(o_i, o_{-i}))\}$;
    end for
    $lev := ExactLev(V + W, i)$;
    if $lev > lev_{best}$ **then**
      $lev_{best} := lev$;
    end if
    for all $o_{-i} \in O_{-i}$ **do**
      $W_i(o_i, o_{-i}) := 0$;
    end for
  end for
  return lev_{best};
end if
if $i > 1$ **return** $ExactLev(V, i-1)$;
else return $\min_{o \in O}\{U(o) - V(o)\}$;

the previous chapter to compute the exact cost of a set. In order to compute $LEV(O)$, we can apply Algorithm 4.2 for all rectangular subsets and return the largest value found.[2]

Theorem 4.11. *Algorithm 4.2 computes the exact leverage of a strategy profile set in time*

$$\mathcal{O}\left(|X|^2 \cdot \max_{i \in N} |O_i|^{n|X_i^* \setminus O_i|-1} + n|O| \cdot \max_{i \in N} |O_i|^{n|X_i^* \setminus O_i|}\right).$$

Proof. The algorithm is correct since it searches for all possibilities of a strategy in $X_i \setminus O_i$ to be dominated by a strategy in O_i. The time complexity follows from solving the doubly recursive equation over the strategy set and the number of players (compare to the analysis of Algorithm 3.1 in the previous chapter). □

4.2 Uniform Leverage

In the setting where a mechanism designer applies uniform implementations the players have less information of the game and are assumed to play a non-dominated strategy uniformly at random. This allows her to calculate with the average cost and thus, the observation stating that the uniform (malicious) leverage is always at least as large as the worst-case (malicious) leverage does not surprise.

Theorem 4.12. *The uniform leverage of a set is always larger than or equal to its leverage. The uniform malicious leverage of a set is always larger than or equal to its malicious leverage.*

Proof.

$$\begin{aligned} lev_{UNI}(O) &= \sup_{V \in \mathcal{V}(O)} \left\{ \operatorname*{avg}_{z \in X^*(V)} \{U(z) - V(z)\} \right\} - \operatorname*{avg}_{x^* \in X^*(V)} U(x^*) \\ &\geq \sup_{V \in \mathcal{V}(O)} \left\{ \min_{z \in X^*(V)} \{U(z) - V(z)\} \right\} - \max_{x^* \in X^*(V)} U(x^*) \\ &= lev(O), \text{ and} \end{aligned}$$

[2] Note that we do not provide algorithms for computing the malicious leverage but for the benevolent leverage only. However, we are sure that a malicious mechanism designer will figure out how to adapt our algorithms for the benevolent leverage for a nastier purpose.

$$\begin{aligned} mlev_{UNI}(O) &= \operatorname*{avg}_{x^* \in X^*(V)} U(x^*) - \inf_{V \in \mathcal{V}(O)} \left\{ \operatorname*{avg}_{z \in X^*(V)} \{U(z) + 2V(z)\} \right\} \\ &\geq \min_{x^* \in X^*(V)} U(x^*) - \inf_{V \in \mathcal{V}(O)} \left\{ \max_{z \in X^*(V)} \{U(z) + 2V(z)\} \right\} \\ &= mlev(O). \end{aligned}$$

□

Another difference between uniform and worst-case leverage concerns target sets O that intersect with X^*, i.e., $O \cap X^* \neq \emptyset$: Unlike the worst-case leverage (Theorem 4.9), the uniform leverage can exceed zero in these cases, as can, e.g., be verified by calculating the leverage of O in Figure 3.4.

4.2.1 Complexity

Similar to the complexity of the worst-case leverage, we can give a polynomial reduction from the uniform implementation cost to the uniform leverage. Thus, the complexity of computing the leverage follows from the NP-hardness of finding the optimal implementation cost, and we can state the following two theorems.

Theorem 4.13. *For games with at least two players, the problem of computing the* exact uniform leverage *of a strategy profile set, and the problem of computing the* exact malicious uniform leverage *of a strategy profile set are both* NP*-hard.*

Proof. The claim follows from the observation that if $(M)LEV^*_{UNI}(O)$ is found, we can immediately compute $k^*_{UNI}(O)$ which is NP-hard (Corollary 3.9). Due to the fact that any $z \in O$ is also in $X^*(V)$ for any $V \in \mathcal{V}^*(O)$, we know that

$$\begin{aligned} lev^*_{UNI}(O) &= \sup_{V \in \mathcal{V}^*(O)} \left\{ \operatorname*{avg}_{z \in X^*(V)} \{U(z) - V(z)\} \right\} - \operatorname*{avg}_{z \in X^*} U(x^*) \\ &= \sup_{V \in \mathcal{V}^*(O)} \left\{ \operatorname*{avg}_{z \in O} U(z) - \operatorname*{avg}_{z \in O} V(z) \right\} - \operatorname*{avg}_{x^* \in X^*} U(x^*) \\ &= \operatorname*{avg}_{z \in O} U(z) - \inf_{V \in \mathcal{V}^*(O)} \left\{ \operatorname*{avg}_{z \in O} V(z) \right\} - \operatorname*{avg}_{x^* \in X^*} U(x^*) \\ &= \operatorname*{avg}_{z \in O} U(z) - \mathbf{k^*_{UNI}(O)} - \operatorname*{avg}_{x^* \in X^*} U(x^*), \text{ and} \end{aligned}$$

$$\begin{aligned} mlev^*_{UNI}(O) &= \operatorname*{avg}_{x^*\in X^*} U(x^*) - \inf_{V\in\mathcal{V}^*(O)} \left\{ \operatorname*{avg}_{z\in X^*(V)} \{U(z)+2V(z)\} \right\} \\ &= \operatorname*{avg}_{x^*\in X^*} U(x^*) - \operatorname*{avg}_{z\in O} U(z) - 2 \inf_{V\in\mathcal{V}^*(O)} \left\{ \operatorname*{avg}_{z\in O} V(z) \right\} \\ &= \operatorname*{avg}_{x^*\in X^*} U(x^*) - \operatorname*{avg}_{z\in O} U(z) - 2\mathbf{k^*_{UNI}}(\mathbf{O}). \end{aligned}$$

Observe that $\operatorname{avg}_{x^*\in X^*} U(x^*)$ and $\operatorname{avg}_{z\in O} U(z)$ can be computed easily. Moreover, as illustrated in the proof of Theorem 4.10, we can efficiently construct a problem instance (G', O) from any (G, O) with the same cost, such that for G': $(m)lev_{(UNI)} = (M)LEV_{(UNI)}$. □

Theorem 4.14. *For games with at least three players, the problem of computing the* non-exact uniform leverage *of a strategy profile set, and the problem of computing the* non-exact malicious uniform leverage *of a strategy profile set are both* NP*-hard.*

Proof. The claim can be proved by reducing the NP-hard problem of computing $k_{UNI}(O)$ to the problem of computing $(M)LEV_{UNI}(O)$. For this reduction we slightly modify the utilities of Player 3 in the respective game used to prove Theorem 3.8. Thereby, we can ensure that for all $z \in O$ it holds that

$$U(z) = -4(m+l)^2 - 2m^2 + m(l+m) =: \gamma.$$

To achieve so we modify Player 3's utilities as follows:
$\forall\ i \in \{1,\dots,m\},\ j \in \{1,\dots,l\}$:

$$\begin{aligned} U_3(s_i, e_j, a) &= \gamma - U_1(s_i, e_j, a) - U_2(s_i, e_j, a), \\ U_3(s_i, e_j, b) &= \gamma + 2 - U_1(s_i, e_j, a) - U_2(s_i, e_j, a), \end{aligned}$$

$\forall\ i \neq j$:

$$\begin{aligned} U_3(s_i, s_j, a) &= \gamma - U_1(s_i, s_j, a) - U_2(s_i, s_j, a), \\ U_3(s_i, s_j, b) &= \gamma + 2 - U_1(s_i, s_j, a) - U_2(s_i, s_j, a), \end{aligned}$$

and $\forall\ i$:

$$\begin{aligned} U_3(s_i, s_i, a) &= \gamma - U_1(s_i, s_i, a) - U_2(s_i, s_i, a), \\ U_3(s_i, s_i, b) &= \gamma + (m+l+2) - U_1(s_i, s_i, a) - U_2(s_i, s_i, a). \end{aligned}$$

Since in this 3-player game, $mlev_{UNI}(O) > 0$, we can give a formula for $k_{UNI}(O)$ depending only on O's (malicious) leverage and the average social gain, namely

$$k_{UNI}(O) = \frac{1}{2}\left(\operatorname*{avg}_{x^*\in X^*} U(x^*) - MLEV_{UNI}(O) \right).$$

Thus, once $MLEV_{UNI}(O)$ is known, $k_{UNI}(O)$ can be computed immediately, and therefore finding the uniform malicious leverage is NP-hard as well. We can adapt this procedure for $LEV_{UNI}(O)$ as well. □

Again, the inapproximability results for SET COVER carry over to the problem of computing the leverage. The following approximation lower bounds are derived by modifying the games constructed from the SET COVER problem in Theorem 3.8, and by using a lower bound for the approximation quality of the SET COVER problem. If no polynomial time algorithm can approximate the size of a set cover within a certain factor, we get an arbitrarily small approximated leverage $LEV_{UNI}^{approx} \leq \epsilon$ while the actual leverage is large. Hence the approximation ratio converges to infinity and, unless P=NP, there exists no polynomial time algorithm approximating the leverage of a game within any function of the input length.

Theorem 4.15. *The exact and the non-exact uniform leverage of a strategy profile set cannot be approximated in polynomial time within any function of the input length for games with at least two players, or three players respectively, unless* P=NP.

Proof. Exact Case. The game constructed from the SET COVER problem in Theorem 3.8 for the exact case is modified as follows: The utilities of Player 1 remain the same. The utilities of Player 2 are all zero except for

$$U_2(e_1, r) = (l+m)(\sum_{i=1}^{m} |S_i| \frac{m+1}{ml+m} - k \cdot \mathcal{LB} - \epsilon),$$

where k is the minimal number of sets needed to solve the corresponding SET COVER instance, $\epsilon > 0$, and $\mathcal{LB}$ denotes a lower bound for the approximation quality of the SET COVER problem. Observe that X^* consists of all strategy profiles of column r. The target set we want to implement exactly is given by $O_1 = \{s_1, ..., s_m\}$ and $O_2 = \{e_1, ..., e_l, d\}$. We compute

$$\begin{aligned} lev_{UNI}^{opt} &= \operatorname*{avg}_{o \in O} U(o) - \frac{k}{ml+m} - \operatorname*{avg}_{x \in X^*} U(x) \\ &= \sum_{i=1}^{m} |S_i| \frac{m+1}{ml+m} - \frac{k}{ml+m} - \sum_{i=1}^{m} |S_i| \frac{m+1}{ml+m} - (-k \cdot \mathcal{LB} - \epsilon) \\ &= k \cdot (\mathcal{LB} - \frac{1}{ml+m}) + \epsilon. \end{aligned}$$

As no polynomial time algorithm can approximate k within a factor $\mathcal{LB}$, $LEV_{UNI}^{approx} \leq \epsilon$. Since $\lim_{\epsilon \to 0} LEV_{UNI}^{opt} / LEV_{UNI}^{approx} = \infty$ the claim follows for a benevolent mechanism designer.

For malicious mechanism designers, we modify the utilities of the game from the proof of Theorem 3.12 for Player 2 as follows:

$$U_2(e_1, r) := (l + m) \left(2k \cdot \mathcal{LB} + \epsilon + \sum_{i=1}^{m} |S_i| \frac{m+1}{ml+m} \right),$$

and $U_2(\_, \_) := 0$ for all other profiles. It is easy to see that by a similar analysis as performed above, the theorem also follows in this case.

Non-Exact Case. We modify the game construction of the proof of Theorem 3.10 by setting

$$U_2(e_1, r, b) := (m+l) \left(\frac{\sum_{i=1}^{m} |S_i|(m+l)^2 + m^2(m+l)^2 + 3m(m+l)}{m^2 + ml + m} - k\mathcal{LB} - \epsilon \right),$$

where k is the minimal number of sets needed to solve the corresponding SET COVER instance, $\epsilon > 0$, and $\mathcal{LB}$ denotes a lower bound for the approximation quality of the SET COVER problem and zero otherwise. Observe that $X^* = \{x \mid x \in X, x = (\_, r, b)\}$, O has not changed, and payments outside O do not contribute to the implementation cost; therefore, implementing O exactly is still the cheapest solution. By a similar analysis as in the proof of Theorem 3.10 the desired result is attained.

For malicious mechanism designers, set

$$U_2(e_1, r, b) := (m+l) \left(\frac{\sum_{i=1}^{m} |S_i|(m+l)^2 + m^2(m+l)^2 + 3m(m+l)}{m^2 + ml + m} + 2k\mathcal{LB} + \epsilon \right)$$

and proceed as above. □

4.2.2 Algorithms

To find algorithms that compute the uniform leverage we can adapt the algorithms for the worst-case leverage from Section 4.1. Recall Algorithm 4.1 that computes the leverage of singletons of a desired strategy profile set. We can adapt Line 3 and 4 to accommodate the definition of the uniform leverage, i.e., set $mlev[x] := \text{avg}_{x^* \in X^*} U(x^*)$ and $mlev[x] := -mlev[x]$. The resulting algorithm helps finding an optimal singleton.

A benevolent mechanism designer can adapt Algorithm 4.2 in order to compute the exact uniform leverage $LEV^*_{UNI}(O)$: She only has to change Line 4 to $\max\{0, ExactLev(V, n) - \text{avg}_{x^* \in X^*} U(x^*)\}$ and 'min' in Line 13 to 'avg'. Invoking this algorithm for any $O' \subseteq O$ yields the subset O with maximal leverage $LEV_{UNI}(O)$.

As seen in Theorem 4.15, there is no polynomial time algorithm giving a non-trivial approximation of a uniform leverage. The simplest method to find a lower bound for $LEV_{UNI}(O)$ is to search the singleton in O with the largest uniform leverage. Unfortunately, there are games (cf. Figure 3.1)

Leverage	Complexity	Properties
Uniform	NP-hard SINGLETON $\mathcal{O}\left(n \cdot \sum_i \lvert X_i \rvert\right)$	$MLEV_{UNI} \geq MLEV$
Worst-case	as hard as implementation cost SINGLETON $\mathcal{O}\left(n \cdot \sum_i \lvert X_i \rvert\right)$	$O \cap X^* \neq \emptyset \Rightarrow (M)LEV = 0$ social opt/worst $\in X^*$ $\Rightarrow (M)LEV = 0$ Equal-sum games $\Rightarrow (M)LEV = 0$

Figure 4.3: *Complexity results for the computation of the leverage.* SINGLETON *indicates the complexity of computing a singleton's leverage.*

where this lower bound is arbitrarily bad, analogously to the lower bound for the worst-case leverage.

The results of this chapter are summarized in Figure 4.3. This concludes the purely theoretical part of this thesis. The next part will present an analysis of the incentives in multicore systems.

Part II

Multi-Core Systems

Chapter 5

The Multicore Revolution

In the year 1965, Gordon E. Moore, co-founder of Intel, stated that the number of transistors on an integrated circuit will double about every year, a projection that was later to be called *Moore's Law*. In retrospect the growth of the number of transistors in processor chips was rather in the order of doubling every two years. Indeed, Moore's law probably became a self-fulfilling prophecy, as it served as a goal for an entire industry, driving both marketing and engineering of semiconductor manufacturers to obey the law, because it was presumed that the competitors would do so as well. While Moore's law seems to hold also in the 21st century,[1] the processor clock rates have stabilized around 2-3 GHz since the year 2005. The physical limitations have been reached. Clock speeds can no longer be effectively increased. For power efficiency, the clock rates have even slightly decreased since 2005. Figure 5.1 plots the clock speeds and the number of transistors in processor chips since 1970. The reason for the fact that the transistor count still grows is that hardware designers have turned to multicore architectures, in which multiple processing cores are included on each chip. Between 2005 and 2010, two or four cores per chip became standard. In 2011, also six-, eight-, or 16-core machines are commercially successful. In compliance with Moore's law, the number of cores per machine will continue to double every few years.

This switch to multicore architectures promises increased parallelism, but not increased single-thread performance. Thus, traditional single-threaded software, or software that makes use of only a few threads cannot capitalize on the increasing computation power at hand. Hence, software that is

[1] The International Technology Roadmap for Semiconductors projected in 2010 that the growth of the transistor count continues until 2013, and then slows down to a rate of a doubling every three years.

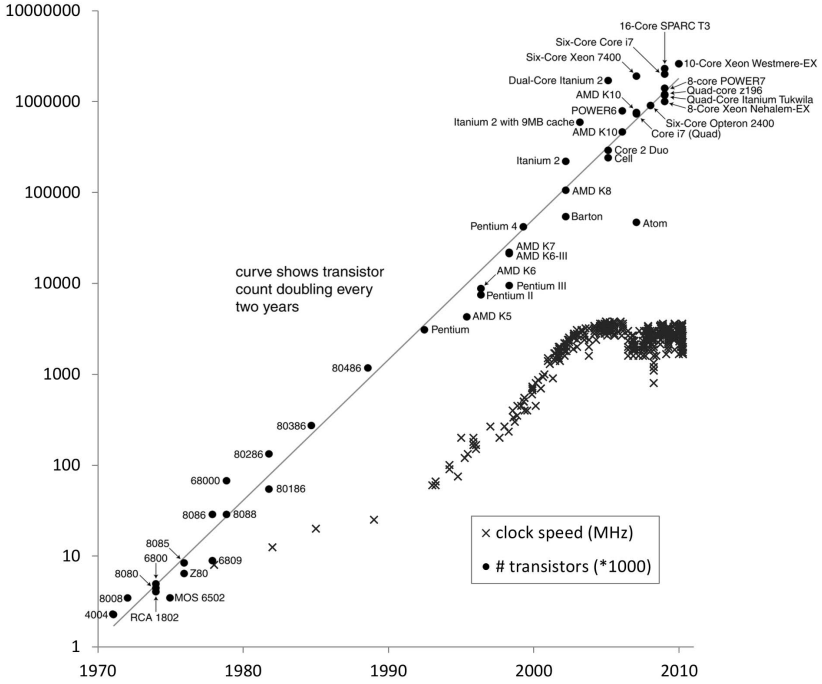

Figure 5.1: *Moore's Law. The y-axis shows the clock speeds in MHz as well as the number of transistors (scaled down by a factor of 1000). The x-axis shows the time a chip was released. The number of transistors in processors continues to double roughly every two years, while the clock speed has stabilized around 2-3 GHz. (Data source: Wikipedia)*

to utilize the multicore machines at a proper level of their capabilities must be parallelized as much as possible. Unfortunately, developing parallel software with the tools of today is a notoriously difficult job and constitutes a major challenge for software developers. To use multiple cores concurrently programmers must identify independent tasks, or task parts, that can be executed in parallel. Moreover, they must coordinate their execution, manage communication and synchronization.

Traditionally, the means of dealing with parallelism are locks; however, there seems to be a general consensus in the computer science research community that locks are not the optimal programming paradigm to deal with concurrency and synchronization (see for instance [56]). It needs outstanding programmers with a high degree of ingenuity to build large parallel systems depending on locks.

There have been several proposals to ease the parallel programmers' task. Note that there is a natural trade-off between the degree of automation involved in the software development, and the complexity of the programmer's job. On the one extremal point, there is a scenario where single-threaded software is parallelized and turned into multi-threaded code by a fully automated compiling process. On the other extremal point, there is the scenario of programmers being exclusively in charge of parallelization without any automated support. Today's common practice of developing software with explicit locks is probably close to the latter scenario.

As with most trade-offs, the best solutions are likely to be found in between the extremal points: a system that automates parallelization to a certain extent, and thus eases the task on the programmer, but also leaves enough freedom to the programmer to exploit semantics unavailable to automated processing. One such solution are transactional memory systems. The paradigm of Transactional Memory, introduced by Lomet [62] in the 1970s and implemented by Herlihy and Moss [47] in the 1990s, has emerged as a promising approach to keep the challenge of writing concurrent code manageable. The basic idea is that, similarly to the database world, the programmer can encapsulate sequences of instructions within an atomic memory transaction. Either the entire transaction is executed, or nothing at all. Other threads will see a transaction as one indivisible operation. Internally, the system takes care of concurrency control by a contention manager that evaluates and resolves conflicts between transactions.

Independent of the way it handles concurrency, a multicore system typically employs shared memory for the communication among threads, to let the threads work together. This communication includes for instance the transmission of the information, whether a thread is allowed to modify an entity or not. Moreover, as soon as multiple threads access a shared data structure the threads compete for the right of accessing or modifying the data

structure. If the threads are all designed with the goal to serve the overall performance of all software accessing the common data structure then the threads should behave well and only acquire access right to data if necessary. However, if the threads are not coordinated in such a well-behaved manner, or in particular, if the threads are designed by different software developers, multicore systems are susceptible to selfish behavior. Since there is competition in many projects, especially within the same company, we must reckon that programmers write threads so as to optimize the performance of their piece of code. Just think of the next evaluation! As a consequence, a multicore system should account for selfish threads, i.e., threads that try to optimize their performance regardless of their impact on the overall system performance.

We see the issue of selfish programmers as one important aspect of concurrent computing that has to be considered in the design of next generation multicore systems. If we are unable to provide easier ways of utilizing the growing parallel computing power of processor chips after the age of clock rate speedup, computer science is endangered to degenerate into *washing machine science*[2], i.e., a science that has yielded one useful technology (building washing machines, or building 3 GHz computers respectively), thereafter, it does not spawn any new technology; it does not support the creation of new products useful for mankind; and it is not part of human progress anymore.

[2]This term is due to Maurice Herlihy of Brown University. See [45] for his call on the computer science community to attend to the issues of multicore computing.

Chapter 6

Good Programming in Transactional Memory

A Transactional Memory (TM) system provides the possibility for programmers to wrap critical code that performs operations on shared memory into transactions. The system then guarantees an exclusive code execution such that no other code being currently processed interferes with the critical operations. To achieve this, TM systems employ a contention management policy. In *optimistic* contention management, transactional code is executed right away and modifications on shared resources take effect immediately. If another thread, however, wants to access the same resource a mechanism called contention manager (CM) resolves the conflict, i.e., it decides which transaction may continue and which must wait or abort. In the case of an abort, all modifications done so far are undone. The aborted transaction will be restarted by the system until it is executed successfully. Thus, in multicore systems, the quality of a program must not only be judged in terms of space and (contention-free) time requirements, but also in terms of the amount of conflicts it provokes due to concurrent memory accesses.

Consider the example of a shared ring data structure. Let a ring consist of s nodes and let each node have a counter field as well as a pointer to the next node in the ring. Suppose a programmer wants to update each node in the ring. For the sake of simplicity we assume that she wants to increase each node's counter by one. Given a start node, her program accesses the current node, updates it and jumps to the next node until it ends up at the start node again. Since the ring is a shared data structure, node accesses must be wrapped into a transaction. We presume the programming language offers an **`atomic`** keyword for this purpose.

The first method in Figure 6.1 (`incRingCounters`) is one way of imple-

```
incRingCounters(Node start){
  var cur = start;
  atomic{
    repeat{
      c = cur.count;
      cur.count = c + 1;
      cur = cur.next; }
    until(cur==start)
  }}
```

```
incRingCountersGP(Node start){
  var cur = start;
  repeat{
    atomic{
      c = cur.count;
      cur.count = c + 1;
    }
    cur = cur.next; }
  until(cur==start) }}
```

Figure 6.1: *Two variants of updating each node in a ring.*

menting this task. It will have the desired effect. However, wrapping the entire while-loop into one transaction is not a very good solution, because by doing so, the update method keeps many nodes blocked, although the update on these nodes is already done and the lock[1] is not needed anymore. A more desirable solution is to wrap each update in a separate transaction. This is achieved by a placement of the **atomic** block as in `incRingCountersGP` on the right in Figure 6.1.

When there is no contention, i.e., no other transactions request access to any of the locked ring nodes, both `incRingCounters` and `incRingCountersGP` run equally fast, this is, if we disregard locking overhead (cf. Figure 6.2). If there are interfering jobs, however, the affected transactions must compete for the resources whenever a conflict occurs. The defeated transaction then waits or aborts and hence system performance is lost. In our example, using `incRingCounters` instead of `incRingCountersGP` leads to many unnecessarily blocked resources, and thereby increases the risk of conflicts with other program parts. In addition, if there is a conflict, and the CM decides that the programmer's transaction must abort, then with `incRingCountersGP` only one modification needs to be undone, namely the update to the current node in the ring, whereas with `incRingCounters` all modifications from to the start node must be rolled back. In brief, employing `incRingCounters` causes an avoidable performance loss.

One might think that it is in the programmer's interest to choose the placement of atomic blocks as beneficial to the TM system as possible. The reasoning would be that by doing so she does not merely improve the system performance, but the efficiency of her own piece of code as well. Unfortunately, in current TM systems, it is not necessarily true that if a thread is

[1] An optimistic, direct-update TM system "locks" a resource as soon as the transaction reads or writes it and releases it when committing or aborting. This is not to be confused with an explicit lock by the programmer. In TM, explicit locks are typically not supported.

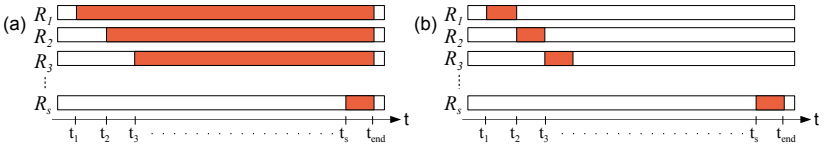

Figure 6.2: *Transactional allocation of ring nodes* $R_1, \ldots, R_s$ *by* `incRingCounters` **(a)** *and by* `incRingCountersGP` **(b)** *on the timeline. Orange colored bar segments signify the locking periods of the corresponding resource.*

---

well designed—meaning that it avoids unnecessary accesses to shared data—it will also be executed faster. On the contrary, we will show that most CMs proposed so far privilege threads that incorporate long transactions rather than short ones. This is not a severe problem if there is no competition for the shared resources among the threads. Although in minor software projects all interfering threads might be programmed by the same developer, this is not the case in large software projects, where there are typically many developers involved, and code of different programmers will interfere with each other. Furthermore, we must not assume that all conflicting parties are primarily interested in keeping the contention low on the shared objects, especially if doing so slows down their own thread. A selfish developer will push the performance of his threads at the expense of other threads or even at the expense of the entire system's performance if the system does not prevent this option. If a multicore system is to avoid this loss of efficiency, it should ensure that the goal of achieving an optimal system performance is compatible with an individual programmer's goal of executing her code as fast as possible. In the following, we will see that most CMs proposed in the literature so far lack such an incentive compatibility, and we identify two design principles that are fit to yield the desired incentive structure.

As a motivational remark we would like to point out that, although TM is most often associated with multithreading, its realm of application is actually much broader. It can for instance also be used in inter-process communication where multiple threads in one or more processes exchange data. Or it can be used to manage concurrent access to system resources. Basically, the idea of TM can be employed to manage any situation where several tasks may concurrently access resources representable in memory. If TM is to be employed in domains such as inter-process communication or managing access to system wide resources (DB, files, system variables), beneficial incentive structures are also indispensable.

6.1 Model

We use a model of a transactional memory system with optimistic contention management, immediate conflict detection, and direct update. As we do not want to restrict TM to the domain of multithreading, we will use the notion of jobs instead of threads to denote a set of transactions belonging together. In inter-process communication, e.g., a job is rather a process than a thread.

6.1.1 Transactional Memory Model

The *environment* $\mathcal{E}$ is a set of n tuples of a job and the time it enters the system, i.e.,

$$\mathcal{E} = \{(J_0, t_0), (J_1, t_1), \ldots, (J_n, t_n)\}.$$

We assume that there are $m \geq n$ machines, and each job is executed exclusively on one machine. The *execution environment* of a job J_i is given by

$$\mathcal{E}_{-i} = \mathcal{E} \setminus \{(J_i, t_i)\}.$$

Each job J_i consists of a sequence of *transactions* $T_{i1}, T_{i2}, \ldots, T_{i|J_i|}$, where $|J_i|$ is the number of transactions contained in J_i. Transactions may access any subset of the shared *resources* $\mathcal{R}$. For the sake of simplicity, we consider all accesses as exclusive,[2] thus, if two transactions both try to access resource $R \in \mathcal{R}$ at the same time, or if one has already locked R and the other desires access to R as well they are in *conflict*. When a conflict occurs a mechanism decides which transaction gains (or keeps) access of R_i, and has the other competing transaction wait or abort. Such a mechanism is called *contention manager (CM)*. We assume that once a transaction has accessed a resource it keeps the exclusive access right until it either commits or aborts. We further assume that the time needed to detect a conflict, to decide which transaction wins, and the time used to commit or start a transaction are negligible. We neither restrict the number of jobs running concurrently, nor do we impose any restrictions on the structure and length of transactions. As a consequence, we do not address the problem of recognizing dead transactions and ignore heuristics included in CMs for this purpose. We say a job J_i *is running* if its first transaction T_{i1} has started and the last $T_{i|J_i|}$ has not committed yet. Notice that in optimistic contention management, the starting time t_i of a job J_i is not influenced by the CM, since it only reacts once a conflict occurs. We assume that any transaction T_{ij} contained in job J_i accesses the same subset of resources $\mathcal{R}_{ij} \subseteq \mathcal{R}$ in each of its runs independently of J_i's starting time t_i, and for any resource the time of its

[2] Invisible reads that would allow a concurrent access without conflicts are not considered.

first access after a (re)start of T_{ij} remains the same in each run.[3] This allows one to describe a contained *transaction* by a 3-tuple

$$T_{ij} = (\mathcal{R}_{ij}, \tau_{ij}, d_{ij})$$

where $\mathcal{R}_{ij} \subseteq \mathcal{R}$ are the resources accessed by T_{ij}, $\tau_{ij} : \mathcal{R}_{ij} \mapsto \mathbb{R}^+$ is a function that maps a resource to its relative access time, and d_{ij} is the *contention-free duration* of T_{ij}, i.e., the time needed from start to commit provided that T_{ij} encounters no conflicts.

For instance $T_{ij} = (\{R_1, R_4\}, \{R_1 \mapsto 3, R_4 \mapsto 0\}, 4)$ describes a transaction that tries to gain immediate access of R_4, access of R_1 after 3 time units, and commits after 4 time units unless it was aborted before. Note that $d_{ij} > \tau_{ij}(R)$ for any resource $R \in \mathcal{R}_{ij}$. Let d_i denote the contention-free duration of job J_i, i.e. the time needed from t_i to the commit time of $T_{i|J_i|}$ in an empty execution environment.

If the CM $\mathcal{M}$ used in a TM system is deterministic we assume that the state of the system at a certain time is determined by $\mathcal{E}$ and $\mathcal{M}$. If $\mathcal{M}$ takes randomized decisions then $\mathcal{E}$ and $\mathcal{M}$ determine a system state probability distribution at any given time. Thus, given $\mathcal{M}$ and $\mathcal{E}$, the execution of $\mathcal{E}$ is thoroughly described. In the following definitions, we presume $\mathcal{M}$ to be deterministic. Corresponding definitions for randomized $\mathcal{M}$ are straightforward by incorporating probability distributions, and we omit explicit definitions for randomized CMs.

By $d^{\mathcal{M},\mathcal{E}}$ we denote the function that maps jobs and transactions in $\mathcal{E}$ to their *execution time*, i.e., $d^{\mathcal{M},\mathcal{E}}(T_{ij})$ is the time from the first start of transaction T_{ij} to its eventual commit in an execution of $\mathcal{E}$ by a TM system managed by $\mathcal{M}$, and similarly, $d^{\mathcal{M},\mathcal{E}}(J_i)$ is the time the same TM system takes executing job J_i where $(J_i, t_i) \in \mathcal{E}$, i.e., the time between t_i and the commit time of the last transaction in J_i. The *makespan* $d^{\mathcal{M},\mathcal{E}}$ of an environment $\mathcal{E}$ in a system managed by $\mathcal{M}$ is the time from $\min_i t_i$ until $\max_i\{t_i + d^{\mathcal{M},\mathcal{E}}(T_i)\}$. Let $t^{\mathcal{M},\mathcal{E}}$ denote the function that maps transactions in $\mathcal{E}$ to their start time, i.e., $t^{\mathcal{M},\mathcal{E}}(T_{ij})$ is the time when T_{ij} is started in the execution of $\mathcal{E}$ by a TM system with CM $\mathcal{M}$. We denote by $\mathcal{L}^{\mathcal{M},\mathcal{E}}(t)$ the set of *locked resources* at time t in a TM system managed by $\mathcal{M}$ when executing environment $\mathcal{E}$. Similar to $\mathcal{R}_{ij}$, we denote by $\mathcal{R}_i$ the set of resources accessed by job J_i, i.e.,

$$\mathcal{R}_i = \bigcup_{j=1}^{|J_i|} \mathcal{R}_{ij}.$$

[3] Note that this is a major simplification of a real shared memory system, where data structures change dynamically. However, as we assume code developers to consider worst-case environments, only the starting time relative to competing jobs, but not the absolute starting time t_i is relevant. All other jobs could just be shifted accordingly. Thus our assumption relaxes to the assumption that resource accesses remain constant after a restart.

We define the *concatenation* $T_{ij\|ij+1}$ of two consecutive transactions T_{ij} and T_{ij+1} as

$$T_{ij\|ij+1} = \left(\mathcal{R}_{ij} \cup \mathcal{R}_{ij+1}, \tau_{ij\|ij+1}, d_{ij} + d_{ij+1}\right),$$

where $\tau_{ij\|ij+1}$ is the function that maps a resource $R \in \mathcal{R}_{ij} \cup \mathcal{R}_{ij+1}$ to $\tau_{ij}(R)$ if $R \in \mathcal{R}_{ij}$, and to $d_{ij} + \tau_{ij+1}(R)$ otherwise. For a job J_i, and an integer $k \in [1, |J_i| - 1]$ we define $Combine(J_i, k)$ to be the job that results when the two transactions T_{ik}, and T_{ik+1} contained in J_i are concatenated to $T_{ik\|ik+1}$, i.e.,

$$Combine(J_i, k) = T_{i1}, \ldots, T_{ik-1}, T_{ik\|ik+1}, T_{ik+2}, \ldots, T_{i|J_i|}.$$

In our discussions, we sometimes compare a job J_i to a similar job J_i'. In such comparisons, we add a dash to notations associated with jobs to indicate the corresponding properties of J_i' rather than that of J_i. For example, $\mathcal{R}_i'$ denotes the resources accessed by J_i'.

6.1.2 Programmer Model

We assume that the program code of each job is written by a different selfish developer and that there is competition among those developers. *Selfish* in this context means that the programmer only cares about how fast her job terminates. This is, the author of job J_i tries to minimize the expected execution time of J_i. We presume programmers have no information on the runtime execution environment $\mathcal{E}_{-i}$, and are thus generally uncertain about the performance of their job. As to deal with this uncertainty, we assume developers act *risk-averse* in the sense that they expect $\mathcal{E}_{-i}$ to be such that J_i's execution time is maximal among all possible finite executions, i.e., the expected running time $\tilde{d}^{\mathcal{M}}(J_i)$ of job J_i in a TM system managed by $\mathcal{M}$ is given by

$$\tilde{d}^{\mathcal{M}}(J_i) = \max_{\{\mathcal{E}_{-i} | d^{\mathcal{M},\mathcal{E}_{-i}\cup\{(J_i,t_i)\}}(J_i) \text{ is finite}\}} d^{\mathcal{M},\mathcal{E}_{-i}\cup\{(J_i,t_i)\}}(J_i).$$

Note that with many CMs the "true" worst-case execution time of J_i is infinite even for finite environments $\mathcal{E}_{-i}$. If, however, a risk-averse developer would expect her job to run forever she could just as well twirl her thumbs instead of writing a piece of code. Hence, the assumption that a job eventually terminates is an inevitable feature of our programmer model. Furthermore, we say a job J_i *dominates* J_i' *under* $\mathcal{M}$ if and only if it holds for any $\mathcal{E}_{-i}$ that

$$d^{\mathcal{M},\mathcal{E}_{-i}\cup\{(J_i,t_i)\}}(J_i) \leq d^{\mathcal{M},\mathcal{E}_{-i}\cup\{(J_i',t_i)\}}(J_i)$$

and there exists an environment $\mathcal{E}_{-i}$ for which strict inequality holds. When implementing a task, a programmer can typically choose among a variety of jobs that all implement the desired logic. We assume that out of these

available choices the programmer opts for any non-dominated job J_i that has minimal expected running time $\tilde{d}^{\mathcal{M}}(J_i) = \min_{J'_i} \tilde{d}^{\mathcal{M}}(J'_i)$. We call these minimal jobs the *solution set*.

Note that the assumed solution concept is idealized in the sense that it is probably infeasible for many tasks to find a job with minimal expected running time. We therefore typically derive statements that preclude certain types of jobs from the solution set rather than statements about what jobs are *in* the solution set. For instance, we will show in Lemma 6.10 that jobs that contain artificial delays are not in the solution set if a certain type of contention manager is used.

6.2 Good Programming Incentives

A first step towards incentive compatible transactional memory is to determine what programmer behavior is desirable for a TM system. For that matter we investigate how a programmer should structure her code, or in particular, how she should place atomic blocks in order to optimize the overall efficiency of a TM system.

When a job accesses shared data structures it puts a load on the system. The insight gained by studying the example of ring counters in Figures 6.1 and 6.2 is that the more resources a job locks and the longer it keeps those locks, the more potential conflicts it provokes. If the program logic does not require these locks the load thereby put on the system is unnecessary.

Fact 6.1. *Unnecessary locking of resources provokes a potential performance loss in a TM system.*

However, the question remains of whether *partitioning* a transaction into smaller transactions—even if doing so does not reduce the resource accesses—results in a better system performance. Consider an example where the program logic of a job J_1 requires exclusive access of resource R_1 for a period of 8 time units. One strategy for the programmer is to wrap all operations on R_1 into one transaction

$$T'_{11} = (\{R_1\}, \{R_1 \mapsto 0\}, 8).$$

However, let the semantics also allow an execution of the code in two subsequent transactions T_{11} and T_{12} where

$$T_{11} = T_{12} = (\{R_1\}, \{R_1 \mapsto 0\}, 4)$$

without losing consistency. Figure 6.3 shows the optimal execution of both strategic variants in an environment $\mathcal{E} = \{(J_1, 0), (J_2, 0)\}$ with $J_2 = T_{21}, T_{22}$ and

$$T_{21} = T_{22} = (\{R_1, R_2\}, \{R_1 \mapsto 4 + \epsilon, R_2 \mapsto 0\}, 5),$$

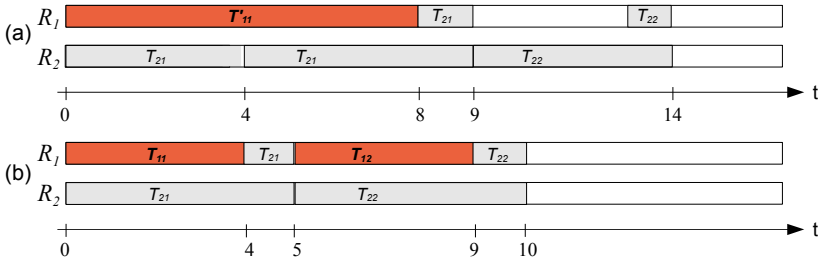

Figure 6.3: *Partitioning example. The picture depicts the optimal allocation of two resources R_1 and R_2 over time in two situations* **(a)** *and* **(b)**. *In* **(a)**, *the programmer of job J_1 does not partition T'_{11}. In* **(b)**, *she partitions T'_{11} into T_{11} and T_{12}. The makespan is shorter in* **(b)**, *the individual execution time of J_1 is faster in* **(a)**.

---

where $\epsilon \in \mathbb{R}^+$ is arbitrarily small, i.e. one clock cycle.

In situation **(a)**, the programmer does not partition T'_{11}. Both jobs J_1 and J_2 start at time $t = 0$, after 4 time units there is a conflict since transaction T_{21} tries to gain access of resource R_1 that is locked by T'_{11}. To achieve an optimal allocation the contention manager $\mathcal{M}$ aborts T_{21}. T_{21} is restarted. No more conflicts occur, and a makespan of $d^{\mathcal{M},\mathcal{E}} = 14$ is achieved. Convince yourself that this is minimal for $\mathcal{E}$. In situation **(b)**, the programmer uses the partitioned version of J_1. Both jobs start at $t = 0$. T_{11} commits after 4 time units. In the period $(4, 5]$, T_{12} and T_{21} continuously compete for resource R_1. An optimal contention manager lets T_{21} run to commit. Transactions T_{12} and T_{22} both start at $t = 5+\epsilon$, and run to commit without conflicts. This yields a makespan of 10. Thus, in the example of Figure 6.3, partitioning T'_{11} allows the system to execute J_1 and J_2 four time units faster.

Furthermore, partitioning is generally beneficial to a TM system in that it provides more flexibility to the allocation schedule. To make this fact clear we consider a TM system that is managed by an optimal offline CM $\mathcal{M}^*$. In contrast to the CMs in a TM system, $\mathcal{M}^*$ is assumed to know the entire environment, including the jobs that arrive in the future, and can thus precompute what runtime decisions lead to a minimal makespan. Hence, $\mathcal{M}^*$ always makes the right decision when resolving a conflict, furthermore, we allow it to postpone the beginning of a transaction T_{ij} to any optimal time t given that $t \geq t_i$ and all T_{ik} with $k < j$ have committed.

Theorem 6.2. *A finer transaction granularity speeds up a transactional memory system managed by an optimal CM $\mathcal{M}^*$, i.e., for any two jobs J_i and J'_i where there exists a $k \in \{1, \ldots, |J_i|-1\}$ such that $J'_i = Combine(J_i, k)$*

it holds for any execution environment $\mathcal{E}_{-i}$ *that*

$$d^{\mathcal{M}^*,\mathcal{E}_{-i}\cup\{(J_i,t_i)\}} \leq d^{\mathcal{M}^*,\mathcal{E}_{-i}\cup\{(J'_i,t_i)\}},$$

and there exists an $\mathcal{E}_{-i}$ *for which inequality holds.*

Proof. First notice that we may assume without loss of generality that under $\mathcal{M}^*$ there are no conflicts: any transaction T_{ij} will finally run from start to commit in an optimal execution of an environment $\mathcal{E}$. Let $\mathfrak{t}_{ij}^{\mathcal{M},\mathcal{E}}$ denote the time when transaction T_{ij} is started for its successful run in the execution of $\mathcal{E}$ under CM $\mathcal{M}$. If a CM $\mathcal{M}$ manages $\mathcal{E}$ optimally then the CM $\mathcal{M}^*$ that works like $\mathcal{M}$ except that it postpones the start of each transaction T_{ij} until $\mathfrak{t}_{ij}^{\mathcal{M},\mathcal{E}}$, manages $\mathcal{E}$ optimally as well. Moreover, since $\mathcal{M}^*$ starts any transaction only when it will run until commit the produced allocation schedule has no conflicts.[4]

Let J_i and J'_i be as described in the theorem. We proceed by showing the existence of a CM $\mathcal{B}$ that achieves $d^{\mathcal{B},\mathcal{E}_{-i}\cup\{(J_i,t_i)\}} = d^{\mathcal{M}^*,\mathcal{E}_{-i}\cup\{(J'_i,t_i)\}}$ for any given $\mathcal{E}_{-i}$. For convenience, let $E := \mathcal{E}_{-i}\cup\{(J_i,t_i)\}$, and $E' := \mathcal{E}_{-i}\cup\{(J'_i,t_i)\}$. $\mathcal{B}$ sets

$$t^{\mathcal{B},E}(T_{ik}) := t^{\mathcal{M}^*,E'}(T'_{ik})$$

where $t'_{ik} = T_{ik\|ik+1}$. $\mathcal{B}$ starts T_{ik+1} immediately after T_{ik} commits, i.e.,

$$t^{\mathcal{B},E}(T_{ik+1}) := t^{\mathcal{B},E}(T_{ik}) + d^{\mathcal{B},E}(T_{ik}).$$

At any time t, $t^{\mathcal{B},E}(T_{ik}) \leq t \leq t^{\mathcal{B},E}(T_{ik}) + d^{\mathcal{B},E}(T_{ik})$, J_i accesses the same resources as J'_i, i.e.,

$$\mathcal{L}^{\mathcal{B},E}(t) = \mathcal{L}^{\mathcal{M}^*,E'}(t).$$

Since the time needed for committing and starting is negligibly small, T_{ik+1} accesses the same resources as T'_{ik} at the same time. Furthermore, when T_{ik+1} starts it has no resources locked. Hence the resources locked by T_{ik+1} are always a subset of the resources accessed by T'_{ik}, i.e.,

$$\mathcal{L}^{\mathcal{B},E}(t) \subseteq \mathcal{L}^{\mathcal{M}^*,E'}(t)$$

where $t^{\mathcal{B},E}(T_{ik+1}) \leq t \leq t^{\mathcal{B},E}(T_{ik+1}) + d^{\mathcal{B},E}(T_{ik+1})$. Note that T_{ij} might have some resources locked from earlier accesses at time $t^{\mathcal{B},E}(T_{ik+1})$. As J'_i does not provoke a conflict J_i neither does so, and T_{ik+1} will commit at the same time as T'_{ik}. $\mathcal{B}$ executes any other transaction T_{ij} with $j \notin \{k, k+1\}$ just like $\mathcal{M}^*$, and the claim about $\mathcal{B}$'s performance follows. Since $\mathcal{M}^*$ is optimal we have that

$$d^{\mathcal{M}^*,E} \leq d^{\mathcal{B},E} = d^{\mathcal{M}^*,E'}.$$

[4] This reflects the fact that an offline CM is able to "look into the future", and thus, it can avoid mistakes.

It remains to describe an execution environment $\mathcal{E}_{-i}$ with the property that $d^{\mathcal{M}^*,\mathcal{E}_{-i}\cup\{(J_i,t_i)\}} < d^{\mathcal{M}^*,\mathcal{E}_{-i}\cup\{(J_i',t_i)\}}$. Let $\mathcal{E}_{-i}$ be an execution environment as follows:

$$\begin{aligned}
\mathcal{E}_{-i} &:= \{(J_v,t_v)\} \text{ with } J_v = T_{v1}, T_{v2} \text{ and } t_v = t_i,\\
T_{v1} &:= \begin{pmatrix} \{R_v,R\},\\ \left\{R_v \mapsto 0, R \mapsto t^{\mathcal{M}^*,\{(J_i,0)\}}(T_{ik+1})\right\},\\ t^{\mathcal{M}^*,\{(J_i,0)\}}(T_{ik+1}) + d_{ik+1} \end{pmatrix},\\
T_{v2} &:= (\{R_v\}, \{R_v \mapsto 0\}, d_i - t^{\mathcal{M}^*,\{(J_i,0)\}}(T_{ik+1}) - d_{ik+1} + \delta),
\end{aligned}$$

where $R_v \notin \mathcal{R}_i$, R is any resource in $\mathcal{R}_{ik}$, and δ is the amount of time that R is locked in a successful run of T_{ik+1}. $\mathcal{M}^*$ achieves an optimal execution of $\mathcal{E}_{-i} \cup \{(J_i,t_i)\}$ by starting both jobs J_i and J_v at t_i, and delaying the start of T_{ik+1} by δ after T_{ik} commits. Thus, all transactions run conflict-free to commit yielding a makespan of

$$d^{\mathcal{M}^*,\mathcal{E}_{-i}\cup\{(J_i,t_i)\}} = t_i + d_i + \delta.$$

Note that δ may equal 0, namely if $R \notin \mathcal{R}_{ik+1}$.

If the developer uses J_i' instead of J_i, in order not to provoke a conflict on resource R then $\mathcal{M}^*$ has to either postpone T_{v1} by d_{ik+1}, or postpone T_{ik}' by $d_{ik+1} + \delta'$ where $\delta' > 0$ is the period of time that R is locked in a successful run of T_{ik}. The former yields a makespan of

$$d^{\mathcal{M}^*,E'} = t_i + d_v + d_{ik+1} = t_i + d_i + \delta + d_{ik+1} > d^{\mathcal{M}^*,E},$$

the latter yields

$$d^{\mathcal{M}^*,E'} = t_i + d_v + \delta' = t_i + d_i + \delta + \delta' > d^{\mathcal{M}^*,E}.$$

For both inequalities we used the fact that $d_v = d_i + \delta$, which holds due to the construction of $\mathcal{E}_{-i}$. □

Theorem 6.2 proves partitioning to be beneficial to a system with an optimal CM. Of course, this does not hold for all CMs. As partitioning gives more freedom to the CM, though, it is highly probable that by incentivizing partitioning, a system achieves a better performance in a selfish environment even with the additional overhead needed for incentive compatibility.

Our investigations show that both, avoiding unnecessary locks, and partitioning transactions whenever possible, are behavioral patterns that are beneficial to a TM system. In the following, we define the properties of a CM that incentivize code developers to adopt this behavior. We say a CM rewards partitioning iff it is rational for a programmer to always partition

a transaction when the program logic allows her to do so, and it punishes unnecessary locking iff it is rational for a programmer to never lock resources unnecessarily.

Definition 6.3 (Reward Partitioning)**.** *A CM $\mathcal{M}$* rewards partitioning *iff for any two jobs J_i and J_i' where there exists a $k \in \{1, \ldots, |J_i| - 1\}$ such that $J_i' = Combine(J_i, k)$ it is rational for a programmer to opt for J_i rather than J_i' given that both jobs implement the desired task.*

Definition 6.4 (Punish Unnecessary Locking)**.** *Let J_i and J_i' be any two jobs with the property that for any point in time t it holds that $\mathcal{L}^{\mathcal{M},\{(J_i,0)\}}(t) \subseteq \mathcal{L}^{\mathcal{M},\{(J_i',0)\}}(t)$, and for at least one t it holds that $\mathcal{L}^{\mathcal{M},\{(J_i,0)\}}(t) \subset \mathcal{L}^{\mathcal{M},\{(J_i',0)\}}(t)$.*
A CM $\mathcal{M}$ punishes unnecessary locking *iff for any such pair J_i, J_i', it is rational for a programmer to opt for J_i rather than J_i' given that both jobs implement the desired task.*

Definition 6.5 (GPI Compatible)**.** *A CM is* good programming incentive (GPI) compatible *iff it rewards partitioning and punishes unnecessary locking.*

Note that by our definitions we achieve that if a job J_i' can be further improved in a TM system managed by a GPI compatible CM, i.e., if it can be further partitioned or shortened in terms of locks, a programmer has an incentive to choose an improved job J_i. Note that if J_i itself can be further improved then it will not be chosen by the programmer either, but the improvement to J_i, and so forth. Consequently, GPI compatibility incentivizes programmers to choose job implementations that cannot be further partitioned, or shortened in terms of locks (without losing consistency). However, there might still be faster jobs that implement the same task in a way that substantially differs from J_i or its improvements. For instance, mere GPI compatibility does not indicate whether it is faster to sort a shared list by employing a merge sort, or a bubble sort algorithm. Generally, we cannot expect any CM to be able to tell whether a job implements the task desired by the programmer, nor whether the algorithm implemented solves a given task elegantly, nor whether the code makes sense at all. Therefore, in order to be GPI compatible a CM must typically make all J_i perform better than J_i' for any job pairs defined as in Definitions 6.3 and 6.4 regardless of the semantics. In that sense, GPI compatibility describes a monotonicity property, namely that a job J_i that is lighter, or finer grained than a job J_i' is guaranteed to perform at least as well as J_i'.

Let us reconsider the example from Figure 6.3 to illustrate that GPI compatibility is not a naturally given property. We have seen that partitioning T_{11}' into T_{11} and T_{12} results in a smaller makespan. But what about the

individual execution time of job J_1? In the unpartitioned execution, where J_1 only consists of T'_{11}, J_1 terminates at time $t = 8$. In the partitioned case, however, J_1 terminates at time $t = 9$. This means that partitioning a transaction speeds up the *overall* performance of a concurrent system managed by an optimal CM, but it possibly slows down an *individual* job. Thus, a selfish programmer has no natural incentive to take the effort of finding a transaction granularity as fine as possible.

Consequently, we can expect that from a certain level of selfishness among developers a CM that incentivizes good programming performs better than the best incentive incompatible CM. In the remainder we are mainly concerned with the question of which contention management policies fulfill GPI compatibility.

As a remark we would like to point out that the "optimal" offline CM $\mathcal{M}^*$ does not reward partitioning, and hence is not GPI compatible. This is shown by the example from Figure 6.3. Note that the optimality of $\mathcal{M}^*$ refers to the scheduling of a given transaction set. If we assume that developers act selfishly then also a system managed by $\mathcal{M}^*$ suffers a performance loss and a different CM that offers incentives for good programming might be more efficient than $\mathcal{M}^*$. There is, however, an inherent loss due to the lack of collaboration. In game theory, this loss is called *price of anarchy* (cf. [6, 24, 83]).

6.3 Priority-Based Contention Management

One key observation when analyzing the contention managers proposed in [85, 86, 43, 12] is that most of them incorporate a mechanism that accumulates some sort of priority for a transaction. In the event of a conflict, the transaction with higher priority wins against the one with lower priority. Most often, priority is supposed to measure, in one way or another, the work already done by a transaction. Timestamp [85] and Greedy [12, 43] measure the priority by the time a transaction is already running. Karma [85] takes the number of accessed objects as priority measure. Kindergarten [85] gives priority to transactions that already backed off against the competing transaction. The intuition behind a priority-based approach is that aborting old transactions discards more work already done and thus hurts the system efficiency more than discarding newer transactions. The proposed contention managers base priority on a transaction's time in the system, the number of conflicts won, the number of aborts, or the number of resources accessed. Definition 6.6 introduces a framework that comprises priority-based CMs. It allows us to classify priority-based CMs and to make generic statements about GPI compatibility of certain CM classes. See Table 6.1 for some examples of how our framework can be used to describe CMs.

Timestamp, Greedy	Karma, Polka
$\mathcal{T} \rightsquigarrow \omega_i = \omega_i + c$	$\mathcal{R} \rightsquigarrow \omega_i = \omega_i + c$
$\mathcal{C} \rightsquigarrow \omega_i = 0$	$\mathcal{C} \rightsquigarrow \omega_i = 0$

Eruption	Polite
$\mathcal{R} \rightsquigarrow \omega_i = \omega_i + c$	$\mathcal{R} \rightsquigarrow \vec{\omega}_i(R) = 1$
$\mathcal{W}$ against $T_j \rightsquigarrow \omega_i = \omega_i + \omega_j$	$\mathcal{A} \rightsquigarrow \vec{\omega}_i(R) = 0 \; \forall R \in \mathcal{R}_i$
$\mathcal{C} \rightsquigarrow \omega_i = 0$	$\mathcal{C} \rightsquigarrow \vec{\omega}_i(R) = 0 \; \forall R \in \mathcal{R}_i$

Table 6.1: *Description of various popular priority-based CMs in terms of the framework introduced in Definition 6.6. Timestamp, Karma, Eruption, and Polite were proposed by Scherrer and Scott in [85], as well as Polka in [86]. Greedy was proposed by Gerraoui et al. in [43]. '$\mathcal{X} \rightsquigarrow f$' indicates that the described CM reacts to an event $\mathcal{X}$ with the modifications f. The value c is typically a small constant increment. Note that only Polite needs resource specific priorities, the other five CMs use scalar priority values.*

Definition 6.6 (Priority-Based)**.** *A* priority-based *contention manager $\mathcal{M}$ associates with each job J_i a dynamic priority function*

$$\vec{\omega}_i : \mathcal{R}_i \mapsto \mathbb{R}$$

that may change over time. For a resource $R \in \mathcal{R}_i$, $\vec{\omega}_i(R)$ is J_i's priority on resource R. $\mathcal{M}$ resolves conflicts between two transactions $T_{ij} \in J_i$ and $T_{qr} \in J_q$ over a resource $R \in \mathcal{R}_i \cap \mathcal{R}_q$ by aborting the transaction with lower priority on R, i.e., if $\vec{\omega}_i(R) \geq \vec{\omega}_q(R)$ then T_{ij} wins otherwise T_{ij} is aborted.

In many CMs, the job priorities are not resource specific, i.e., $\vec{\omega}_i(R) = c$ for all resources $R \in \mathcal{R}_i$ where $c \in \mathbb{R}$. In this case we can replace $\vec{\omega}_i$ by a scalar priority value $\omega_i \in \mathbb{R}$. We call such a CM *scalar-priority-based*. In the remainder we often use ω_i instead of $\vec{\omega}_i$ for the sake of simplicity, even if we are not talking about scalar-priority-based CMs only. Mostly, for a correct valuation of a job's competitiveness, absolute priority values are not relevant, but the relative value to other job priorities.

Definition 6.7 (Relative Priority)**.** *A job J_i's* relative priority *function $\tilde{\omega}_i : \mathcal{R}_i \mapsto \mathbb{R}$ is defined by*

$$\tilde{\omega}_i(R) := \vec{\omega}_i(R) - \min_{j : R \in \mathcal{R}_j} \vec{\omega}_j(R).$$

If the CM uses scalar priorities, J_i's relative priority $\tilde{\omega}_i \in \mathbb{R}$ is obtained by subtracting $\min_{j=1\ldots n} \omega_j$ from the absolute priority ω_i.

Since optimistic CMs feature a reactive nature it is best to consider the priority-building mechanism as event-driven. On each event, the CM may update the priority functions. We find that the following *events* may occur for a transaction $T_{ij} \in J_i$ in a transactional memory system:

$\mathcal{T}$: A time step. This event occurs in every time step.

$\mathcal{W}$: T_{ij} wins a conflict. Event $\mathcal{W}$ occurs when the contention manager has resolved a conflict in favor of T_{ij}.

$\mathcal{A}$: T_{ij} loses a conflict and is aborted. Event $\mathcal{A}$ occurs when the CM has resolved a conflict in favor of one of T_{ij}'s competitors.

$\mathcal{R}$: T_{ij} successfully allocates a resource. Event $\mathcal{R}$ occurs when T_{ij} gains access of a resource.

$\mathcal{C}$: T_{ij} commits. Event $\mathcal{C}$ occurs when a transaction T_{ij} commits.

As an example, a Timestamp CM $\mathcal{M}_T$, as defined in [85], is modelled by means of events of type $\mathcal{T}$ and $\mathcal{C}$, i.e., in a time step dt after $T_{ij} \in J_i$ entered the system, ω_i is increased by $d\omega = \alpha dt$, $\alpha \in \mathbb{R}^+$ until $\mathcal{C}$ occurs, then it is reset to 0. The scalar priority of J_i at a time t, $t^{\mathcal{M}_T,\mathcal{E}}(T_{ij}) < t \leq t^{\mathcal{M}_T,\mathcal{E}}(T_{ij}) + d^{\mathcal{M}_T,\mathcal{E}}(T_{ij})$ is given by

$$\omega_i(t) = \int_{t^{\mathcal{M}_T,\mathcal{E}}(T_{ij})}^{t} \alpha dt = \alpha(t - t^{\mathcal{M}_T,\mathcal{E}}(T_{ij})).$$

Note that events of type $\mathcal{R}$ happen regardless of whether the allocated resource was freely available, or whether T_{ij} had to win in a conflict against other transactions to lock it. If T_{ij} wants to acquire a resource R that is currently locked by another transaction, the contention manager decides which transaction has to abort. If T_{ij} has to abort there occurs an $\mathcal{A}$-event for T_{ij}. If T_{ij} may continue, there occurs both a $\mathcal{W}$-event as well as an $\mathcal{R}$-event. Further note that priorities are associated with jobs rather than transactions. Thus, a $\mathcal{C}$-event does not necessarily result in ω_i being reset to 0.

When assessing the contention management policies in the literature with the proposed framework, we could observe that most of them are of one of the following two subtypes of priority-based CMs.

Definition 6.8 (Priority-Accumulating)**.** *A priority-based contention manager is* priority-accumulating *iff no event decreases a job's priority and there is at least one type of event which causes the priority to increase.*

Definition 6.9 (Quasi-Priority-Accumulating)**.** *A contention manager is called* quasi-priority-accumulating *iff a CM is priority-accumulating with respect to events of type $\mathcal{T}$, $\mathcal{W}$, $\mathcal{A}$ and $\mathcal{R}$, and it only resets J_i's priority on a $\mathcal{C}$-event.*

Timestamp is an example of a quasi-priority-accumulating contention manager as it only decreases priority on a commit event.

6.3.1 Waiting Lemma

We argue in this section that delaying the execution of a job is not a rational strategy with priority-based CMs, i.e., jobs with artificial delays are not in the solution set if programmers use the solution concept defined in Section 6.1. Note that a programmer can make a job wait by introducing unnecessary code that does not allocate shared resources. We consider cases where J_i waits before (re)starting a transaction T_{ij} as well as cases where T_{ij} is already running, has locked some resources and then waits before resuming (cf. Figure 6.4). For our proof to work, we require two restrictions on the contention manager's priority modification mechanism:

I. The extent to which ω_i is increased (or decreased) on a certain event never depends on ω_i's current value.

II. In a period where no events occur except for time steps, all priorities ω_i increase by $\Delta\omega \geq 0$.

Restriction I implies that rules such as "if ω_i is larger than 10 add 100", or "$\omega_i = 2\omega_i$" are prohibited. A rule like "$\omega_i = \omega_i + 2$" on the other hand is permitted. Intuitively, it seems that a rule that, e.g., doubles the current priority on certain types of events does not seem too far-fetched. Nevertheless, we are not aware of any contention manager in the literature that employs such an update rule. Thus, Restriction I is probably not a substantial reduction to the CM design space. Restriction II basically excludes CMs that decrease priorities on $\mathcal{T}$-events, and CMs in which $\mathcal{T}$-events do not affect all jobs in the same manner. Again, we do not know of any contention manager that incorporates rules of this kind. As most proofs that follow Lemma 6.10 rely on these restrictions, investigating CMs that do not comply with Restrictions I and II might still be an interesting subject for future work.

Lemma 6.10. *It is irrational to add artificial delays to a job, given that the TM system is managed by a priority-based CM $\mathcal{M}$ that is restricted by (*I.–II.*).*

Proof. We show the claim by comparing a job J_i' that incorporates artificial delays with a wait-free job J_i that results when omitting all delays in J_i'. In particular we prove that the programmers expect a shorter execution time for J_i than for J_i', i.e. $\tilde{d}^{\mathcal{M}}(J_i) < \tilde{d}^{\mathcal{M}}(J_i')$. Let $\omega_i(t)$ be ω_i at time t. Let J_i, or J_i' respectively enter the system at time t_i. Let $\mathcal{E}_{-i}$ be an execution environment for which

$$d^{\mathcal{M},\mathcal{E}_{-i}\cup\{(J_i,t_i)\}}(J_i) = \tilde{d}^{\mathcal{M}}(J_i).$$

We can construct an execution environment $\mathcal{E}'_{-i}$ for which it holds that

$$\tilde{d}^{\mathcal{M}}(J_i) < d^{\mathcal{M},\mathcal{E}'_{-i}\cup\{(J'_i,t_i)\}}(J'_i) \leq \tilde{d}^{\mathcal{M}}(J'_i)$$

from $\mathcal{E}_{-i}$ as follows: let us assume that J'_i incorporates only one artificial delay in the interval $[t_0, t_0 + \Delta]$. For any run of J'_i, $\mathcal{E}'_{-i}$ lets all other jobs $J_j, j \neq i$ delay their transactions as well during the interval $[t_0, t_0 + \Delta]$. Thus we establish a situation for J'_i that is at least as bad at time $t_0 + \Delta$ as the situation at t_0. Because of Restriction II, we have

$$\tilde{\omega}'_j(t_0 + \Delta) = \tilde{\omega}'_j(t_0)$$

for any $j = 1 \ldots n$, i.e., the relative priorities are conserved. Since the conflict-resolving mechanism of $\mathcal{M}$ does not depend on the priorities' absolute values, but only on their order, and further, modifications of priorities never depend on the priorities' absolute values, by resuming all work at $t_0 + \Delta$ and delaying all jobs in $\mathcal{E}_{-i}$ with starting time $> t_0$ by Δ, we get that

$$d^{\mathcal{M},\mathcal{E}'_{-i}\cup\{(J'_i,t_i)\}}(J'_i) = \tilde{d}^{\mathcal{M}}(J_i) + \Delta.$$

If J'_i has more than one artificial delay, we can do the same for each delay interval.

We have proven that if J_i and J'_i are either both non-dominated, or both dominated, J'_i cannot be in the solution set. However, if J'_i would be non-dominated and J_i dominated we could not make this conclusion, and it would be unclear which job is to be preferred. Luckily this case cannot occur. We prove this by showing that if J_i is dominated then J'_i must be dominated as well. Let $\hat{J}_i$ be a job that dominates J_i. We construct a job $\hat{J}_i{}'$ from $\hat{J}_i$ that basically waits whenever J'_i waits. Similar arguments as before, namely that relative priorities are preserved, imply that J'_i is dominated by $\hat{J}_i{}'$. This concludes the proof. □

Note that the claim of Lemma 6.10 is intimately linked to the solution concept stated in the model section. Although it seems intuitive we can only establish it since we model the programmers to be unaware of any runtime conditions, and risk-averse in that they assume a "worst-case" execution environment in which their jobs still eventually finish. In practice, a programmer often has some information about the environment in which her job will be deployed. Hence it might make sense to presume some structure of $\mathcal{E}_{-i}$. For example, she could assume that lengths of locks follow a certain distribution, or that each resource has a given probability of being locked. In such cases waiting might not be irrational. In the following, we will sometimes argue that a CM is GPI compatible by comparing two jobs J_i and J'_i where both are equal except for J'_i either locks a resource unnecessarily, or it does not

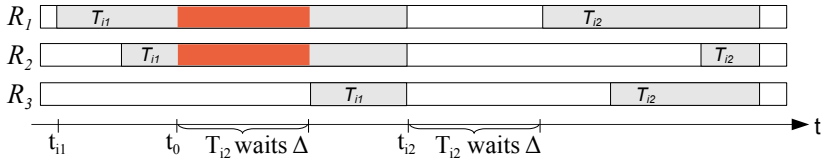

Figure 6.4: *Job $J_i = T_{i1}, T_{i2}$ waits at time t_0 for a period of Δ. In this period, J_i keeps locking the already locked resources R_1 and R_2. After T_{i1} commits, the system would let T_{i2} start immediately, but the programmer of J_i decided to let T_{i2} wait Δ before it accesses the first resource.*

partition a transaction that is partitioned in J_i. We will show that in any given execution environment $\mathcal{E}_{-i}$, job J_i

- either performs at least as fast as J_i', or
- if it is slower than J_i' this is because J_i does not wait at a certain point in the execution.

Since we could achieve the same performance as J_i' in the latter case by introducing artificial delays to J_i, which we showed to be irrational, we conclude that a developer prefers J_i even if it does not dominate J_i'. All that remains to show is that there is at least one $\mathcal{E}_{-i}$ in which J_i outperforms J_i'. We will use analogous reasoning to argue that a CM is *not* GPI compatible. In particular we will show that there exists an execution environment $\mathcal{E}_{-i}$ in which J_i' is faster than J_i, and in which J_i could not achieve the performance of J_i' by introducing delays.

6.3.2 Quasi-Priority-Accumulating Contention Management

Quasi-priority-accumulating CMs increase a transaction's priority over time. Again, the intuition behind this approach is that, on the one hand, aborting old transactions discards more work already done, and thus hurts the system efficiency more than discarding newer transactions, and, on the other hand, any transaction will eventually have a priority high enough to win against all other competitors. This approach is legitimate. Although the former presupposes some structure of $\mathcal{E}$ and the latter is not automatically fulfilled, examples of quasi-priority-accumulating CMs showed to be useful in practice (cf. [86]). However, quasi-priority-accumulating CMs bear harmful potential. They incentivize programmers to not partition transactions, and in some cases even to lock resources unnecessarily. Consider the case where a job has accumulated high priority on a resource R. It might be advisable for the job to keep locking R in order to maintain high priority. Although it does not

need an exclusive access for the moment, maybe later on, the high priority will prevent an abort, and thus save time. In fact, we can show that the entire class of quasi-priority-accumulating CMs is not GPI compatible.

Theorem 6.11. *Quasi-priority-accumulating CMs restricted by (I.–II.) are not GPI compatible.*

Proof. Let J_i, J_i', and k be such that $J_i' = \mathit{Combine}(J_i, k)$. Let both jobs J_i and J_i' enter the system at time t_i for comparison. We show the claim by constructing an environment $\mathcal{E}_{-i}$ in which J_i' executes faster than J_i and in which it is impossible to achieve the performance of J_i' by introducing delays to J_i. For ease of notation, let $E := \mathcal{E}_{-i} \cup \{(J_i, t_i)\}$, and $E' := \mathcal{E}_{-i} \cup \{(J_i', t_i)\}$. Furthermore, we denote by $\omega_i(t)$ the priority ω_i at time t. Let $\mathcal{E}_{-i}$ be such that T_{ik}' is not aborted until commit. Hence, T_{ik} is not aborted until commit at time $t^{\mathcal{M},E}(T_{ik}) + d_{ik}$ either. Furthermore, let $\mathcal{E}_{-i}$ be such that there is at least one event that increases ω_i in the time interval $(t^{\mathcal{M},E}(T_{ik}), t^{\mathcal{M},E}(T_{ik}) + d_{ik})$. Thus

$$\omega_i'(t^{\mathcal{M},E}(T_{ik}) + d_{ik}) = \omega_i(t^{\mathcal{M},E}(T_{ik}) + d_{ik}) > \omega_i(t^{\mathcal{M},E}(T_{ik})) \geq 0.$$

When T_{ik} commits, ω_i is reset to 0 and

$$\omega_i'(t^{\mathcal{M},E}(T_{ik+1})) > \omega_i(t^{\mathcal{M},E}(T_{ik+1})).$$

Since T_{ik+1} is started immediately, it will provoke the exact same conflicts as T_{ik}' at times $t \geq t^{\mathcal{M},E}(T_{ik+1})$. Let the first event on T_{ik+1} (except for time steps) be a conflict against a transaction $T_v \notin J_i$ whose priority is lower than the priority of J_i', but higher than the priority of J_i, i.e., $\omega_i' > \omega_v > \omega_i$ at the time this conflict occurs. Thus, T_{ik+1} is aborted and must be restarted, whereas T_{ik}' wins the conflict and runs to commit. Let all future transactions of J_i' run without conflicts. Thus, we get that $d^{\mathcal{M},E'}(J_i') < d^{\mathcal{M},E}(J_i)$, i.e., J_i' executes faster than J_i. Moreover, as all transactions T_{ij}' with $j \geq k$ run to commit in the first attempt it is impossible to introduce any delay into T_{ik} or T_{ik+1} without exceeding the execution time of J_i'. □

Theorem 6.11 reflects the intuition that if committing decreases an advantage in priority then there are cases where it is rational for a programmer not to commit and start a new transaction, but to continue instead with the same transaction. Obviously, the opposite case is possible as well, namely that by not committing the developer causes a conflict with a high priority transaction on a resource, which could have been released if the transaction would have committed earlier, and thus is aborted. As in our model of a risk-averse programmer she does not suppose any structure on $\mathcal{E}_{-i}$, she does not know which case is more likely to happen either, and therefore has no

preference among the two cases. She would probably just choose the strategy which is easier to implement. If we assumed, e.g., that a resource R is locked at time t with probability p by a transaction with priority x where both, p and x follow a certain probability distribution, then there would be a clear trade-off between executing a long transaction and therewith risking more conflicts and partitioning a transaction, and thus losing priority.

Note that due to the nature of its proof, Theorem 6.11 extends easily to the claim that no priority-based CM rewards partitioning unless it prevents the case where, after a commit of transaction $T_{ij} \in J_i$, the subsequent transaction $T_{ij+1} \in J_i$ starts with a lower priority than T_{ij} had just before committing. In fact, we can show that all priority-accumulating CMs proposed by [12, 43, 85, 86] are not GPI compatible. For a detailed description of the mentioned contention managers, please refer to the original work [12, 43, 85, 86], or to the technical report of [37].[5] There you can also find a discussion on the following corollary.

Corollary 6.12. *Polite, Greedy, Karma, Eruption, Kindergarten, Timestamp and Polka are not GPI compatible.*

6.3.3 Priority-Accumulating Contention Management

The inherent problem of quasi-priority-accumulating mechanisms is not the fact that they accumulate priority over time, but the fact that these priorities are reset when a transaction commits. Thus, by committing early, a job loses its priority when starting a new transaction. One possibility to overcome this problem is to not reset ω_i when a transaction of J_i commits. With this trick, neither partitioning transactions nor letting resources go whenever they are not needed anymore causes a reset of the accumulated priority. We further need to ensure that a succeeding transaction is started immediately after its predecessor commits, because otherwise partitioning would result in a longer execution even in a contention-free environment. We denote this property of a CM as *gapless transaction scheduling*.

Note that in the assumed model of optimistic contention management, gapless transaction scheduling is naturally given. This is due to the fact that in optimistic CM resource modifications are visible immediately, and commit operations are very lightweight, i.e., negligible in our model. The following theorems, Theorems 6.13 and 6.14, only hold for CMs that schedule transactions gapless. As gapless transaction scheduling is part of our optimistic contention management model we do not repeat it in the statements. However, we need to define the following before stating the theorems: if a CM $\mathcal{M}$ only modifies priorities on a certain event type $\mathcal{X}$, we say that $\mathcal{M}$ *is based only on* $\mathcal{X}$*-events*.

[5] available at www.dcg.ethz.ch/publications/isaac09_EWtik.pdf

Theorem 6.13. *Any priority-accumulating CM $\mathcal{M}$ that is based only on time ($\mathcal{T}$-events) punishes unnecessary locking.*

Proof. Since the priorities grow monotonous over time, and $\mathcal{T}$-events happen for all jobs at the same time, there is an implicit total order among all jobs in the system. In particular, a job J_j always has a higher priority than any job that entered the system after J_j, $t_j > t_k$ implies $\omega_j > \omega$[6]. From a job J_i's perspective, this order divides the competing jobs in the system into two sets, $L_i = \{J_j \mid \omega_j < \omega_i\}$ and $H_i = \{J_j \mid \omega_j > \omega_i\}$. By transitivity it follows that a job in L_i cannot influence a job in H_i, neither directly nor indirectly by influencing other jobs in L_i. A job in H_i will win any conflicts against any job in L_i anyway. Thus, all jobs in L_i are irrelevant for the performance of the jobs in H_i, and therewith for the execution of J_i either. Let J_i and J_i' be two jobs that are completely equal in all transactions except for one, T_{ik}, or T_{ik}' respectively, where $T_{ik}' \in J_i'$ contains an unnecessary lock. Thus, for any point in time t it holds that

$$\mathcal{L}^{\mathcal{M},\{(J_i \setminus T_{ik},0)\}}(t) = \mathcal{L}^{\mathcal{M},\{(J_i' \setminus T_{ik}',0)\}}(t).$$

Consider the case where $d_{ik} = d_{ik}'$ first. Comparing T_{ik} with T_{ik}' in an empty environment, there is an interval (a, b) for which it holds that for any $t \in (a, b)$,

$$\mathcal{L}^{\mathcal{M},\{(T_{ik},0)\}}(t) = \mathcal{L}^{\mathcal{M},\{(T_{ik}',0)\}}(t) \setminus \{R\}.$$

This is, T_{ik}' locks resource R unnecessarily in the interval (a, b). Given an execution environment $\mathcal{E}_{-i}$, the unnecessary lock of R either does not provoke a conflict, or it does. If it provokes no conflicts, or only conflicts with jobs in L_i then choosing J_i or J_i' results in the same execution time. If it provokes a conflict against a job in H_i, however, T_{ik}' is aborted whereas T_{ik} continues. If all transactions after T_{ik} run conflict-free J_i is strictly faster than J_i'. Otherwise, let t_{last} be the time when T_{ik}' is restarted for the last time before commit, i.e., T_{ik}' commits at time $t_{last}+d_{ik}$. In case J_i' executes faster than J_i the programmer could delay T_{ik} until t_{last} and thus reach the same execution time as with J_i'. This is because the resources allocated by T_{ik} would always be a subset of the resources allocated by T_{ik}' in the interval $[t_{last}, t_{last} + d_{ik}]$. In order to do so, however, the programmer would introduce an artificial delay to J_i. As $\mathcal{M}$ is time-based, Lemma 6.10 applies and implies that choosing J_i' over J_i is irrational.

It remains to show that if $d_{ik} < d_{ik}'$ then T_{ik} is still preferable. This case occurs when T_{ik}' contains an unnecessary lock that additionally delays all future resource accesses compared to T_{ik}. Let $\delta = d_{ik}' - d_{ik}$ be the delay. In

[6] If $\mathcal{M}$ always resolves ties consistently, e.g. in favor of the job with lower id then a strict order is guaranteed also if we allow concurrent starting times.

terms of resource allocation this means that there is a point in time a such that

$$\begin{aligned} \forall\, t \leq a \quad &: \quad \mathcal{L}^{\mathcal{M},\{(T'_{ik},0)\}}(t) = \mathcal{L}^{\mathcal{M},\{(T_{ik},0)\}}(t), \text{ and} \\ \forall\, t, a < t \leq a+\delta \quad &: \quad \mathcal{L}^{\mathcal{M},\{(T'_{ik},0)\}}(t) = \mathcal{L}^{\mathcal{M},\{(T'_{ik},0)\}}(a) \cup \{R\}, \text{ and} \\ \forall\, t > a+\delta \quad &: \quad \mathcal{L}^{\mathcal{M},\{(T'_{ik},0)\}}(t) = \mathcal{L}^{\mathcal{M},\{(T_{ik},\delta)\}}(t). \end{aligned}$$

For this case, the empty environment $\mathcal{E}_{-i} = \emptyset$ can serve as positive instance where

$$d^{\mathcal{M},\{\mathcal{E}_{-i} \cup (J_i,t_i)\}}(J_i) < d^{\mathcal{M},\{\mathcal{E}_{-i} \cup (J'_i,t_i)\}}(J'_i).$$

For an environment $\mathcal{E}_{-i}$ in which T'_{ik} is in none of its runs aborted during the interval $(a, a+\delta)$ (relative to the run's start time), a programmer could achieve the same performance by introducing a delay of δ to T_{ik} at time a. If $\mathcal{E}_{-i}$ is such that T'_{ik} is aborted due to the unnecessary lock in $(a, a+\delta)$ then J_i achieves the same performance as J'_i by delaying T_{ik} until $t_{last}+\delta$, where t_{last} is defined as before. We thus proved that whenever J'_i executes faster than J_i, the programmer could achieve the same execution time by introducing delays to J_i. From Lemma 6.10 it follows that unnecessary locking is irrational. □

Theorem 6.14. *Any priority-accumulating CM $\mathcal{M}$ that is based only on time ($\mathcal{T}$-events) rewards partitioning.*

Proof. Let J_i, J'_i, and k be such that $J'_i = \mathit{Combine}(J_i, k)$. If we compare the strategy of using J_i to the strategy of using J'_i then we can make the following observations. Since T_{ik} starts at the same time as T'_{ik}, T_{ik} will lock the exact same resources and thus provoke the same conflicts as T'_{ik}. T_{ik+1} locks less or equally many resources as T_{ik}, i.e., at any time t it holds that

$$\mathcal{L}^{\mathcal{M},\{(T_{ik+1},d_{ik+1})\}}(t) \subseteq \mathcal{L}^{\mathcal{M},\{(T'_{ik},0)\}}(t).$$

Let t_c be the time when T_{ik} runs to commit in $\mathcal{E}_{-i}$. Until t_c both J_i and J'_i behave exactly the same in any $\mathcal{E}_{-i}$. Since $\mathcal{M}$ schedules transactions gapless, if T_{ik+1} provokes a conflict at time $t \in [t_c, t_c + d_{ik+1}]$ then T'_{ik} provokes the same conflict at time t. After t_c, T_{ik+1} is started immediately. Depending on $\mathcal{E}_{-i}$ there are two scenarios:

(a) the unfinished run of T'_{ik} provokes only conflicts with jobs in L_i, or no conflicts at all, and

(b) T'_{ik} provokes a conflict with a job in H_i, where L_i and H_i are defined as in the proof of Theorem 6.13.

In case (a), J_i provokes a subset of the conflicts that J_i' provokes. J_i and J_i' have the same start time, and thus always the same priority. T_{ik+1} wins all conflicts, and runs until commit. T_{ik+1} and T_{ik}' commit at the same time. As all succeeding transactions are equivalent, J_i''s runtime equals J_i's runtime.

In case (b), T_{ik}' is aborted and restarted. We have a positive instance of an environment $\mathcal{E}_{-i}$ if the resource, responsible for the abort of T_{ik}', is not locked by T_{ik+1} at the time of the conflict, and all succeeding transactions of J_i run until commit in the first run. Then J_i performs strictly better than J_i'. We also have a positive instance if T_{ik+1} is aborted at the same time, and runs conflict-free afterwards. This is because T_{ik} has already committed and J_i does not need to redo the work done in T_{ik}. For all other instances that create scenario (b) the programmer of J_i could achieve the same performance as with J_i' by delaying T_{ik+1} until $t_{last} + d_{ik}$, where t_{last} is the time when T_{ik}' is started for its final run. Lemma 6.10 implies that J_i is preferable to J_i'. Partitioning is rewarded. □

By the definition of GPI compatibility, Theorems 6.13 and 6.14 immediately imply that priority-accumulating CMs that are based on time only are GPI compatible.

Corollary 6.15. *Any priority-accumulating CM $\mathcal{M}$ that is based only on time ($\mathcal{T}$-events) is GPI compatible.*

This promising result for priority-accumulating CMs shows that it is possible to design priority-based contention managers which are GPI compatible. As an example, by simply not resetting a job J_i's priority when a contained transaction $T_{ij} \in J_i$ commits, we can make a Timestamp contention manager GPI compatible. Nevertheless, contention managers based on priority are generally dangerous in the sense that they bear a potential for selfish programmers to cheat, i.e., to find ways of boosting their job's priority such that their code is executed faster (at the expense of the overall system performance). For example, consider a CM like Karma [85], where priority depends on the number of resources accessed. One way to gain high priority for a job would be to quickly access an unnecessarily large number of objects and thus become overly competitive. Or if priority is based on the number of aborts, or the number of conflicts, a very smart programmer might use some dummy jobs that compete with the main job in such a way that they boost its priority. In fact, we can show that a large class of priority-accumulating contention managers is not GPI compatible.

Theorem 6.16. *A priority-accumulating CM $\mathcal{M}$ is* not GPI compatible *if one of the following holds:*

(i) *$\mathcal{M}$ increases a job's relative priority on $\mathcal{W}$-events.*

(ii) $\mathcal{M}$ *increases relative priority on* $\mathcal{R}$*-events.*

(iii) $\mathcal{M}$ *schedules transactions gapless and increases relative priorities on* $\mathcal{C}$*-events.*

(iv) $\mathcal{M}$ *restarts aborted transactions immediately and increases relative priorities on* $\mathcal{A}$*-events.*

Proof. Throughout the proof, we suppose without loss of generality that in a CM $\mathcal{M}$ each job J_i has exactly one priority $\omega_i \in \mathbb{R}$ associated to it. Let $\omega_i(t)$ denote J_i's priority at time t. For parts *(i)*, *(ii)* and *(iv)*, let T'_{ij} be a transaction that locks resource $R \in \mathcal{R}$ unnecessarily during the interval $[t_u-\epsilon, t_u+\epsilon)$. Let T_{ij} be exactly the same transaction as T_{ij} except it does not lock R during $[t_u - \epsilon, t_u + \epsilon)$. We are going to show the claims by comparing the performance of job J_i when containing T_{ij} with its performance when containing T'_{ij} instead of T_{ij}.

(i). Let T'_{ij} provoke an unnecessary conflict on R with another transaction T_k at time t_u, and $\omega_i(t_u) > \omega_k(t_u)$. T'_{ij} wins the competition for R, and $\mathcal{M}$ increases ω_i by δ. Furthermore, let T'_{ij} provoke a conflict on a resource $Q \in \mathcal{R}$ with a transaction T_l at time $t_u+\epsilon$, and $\omega_l(t_u+\epsilon) < \omega_i(t_u+\epsilon) < \omega_l(t_u+\epsilon)+\delta$. If J_i would use T_{ij} instead of T'_{ij} then $\omega_l(t_u + \epsilon) > \omega_i(t_u + \epsilon)$ for an ϵ small enough. T_{ij} would abort, and, given there are no more conflicts, thus prolongate the execution time of J_i. Since there is no way to introduce delays to T_{ij} without making its execution take longer than T'_{ij} it follows that $\mathcal{M}$ does not punish unnecessary locking.

(ii). Let T'_{ij} be so that it does not access R at all, or only after the unnecessary lock at $t_u + \epsilon$. Let there be no conflicting transaction on R during $[t_u - \epsilon, t_u + \epsilon)$, and let the contribution of having acquired R to the priority increase be δ. Further assume that at time $t_u + \epsilon$, T'_{ij} has a conflict with T_l and $\omega_l(t_u + \epsilon) < \omega_i(t_u + \epsilon) < \omega_l(t_u + \epsilon) + \delta$. If J_i would use T_{ij} instead of T'_{ij} then $\omega_l(t_u + \epsilon) > \omega_i(t_u + \epsilon)$, T_{ij} would abort and prolongate the execution time of J_i. There is no way delays could make T_{ij} perform as good as T'_{ij}. Thus $\mathcal{M}$ does not punish unnecessary locking.

(iii). Let J'_i consist of the transactions T_{i1}, T_{i2} and T_{i3}. Let J_i consist of T_{i1} and T_{i3}. Let T_{i2} be a simple transaction which unnecessarily locks $R \in \mathcal{R}$ for a period of ϵ, and then commits. Let T_{i3} be a transaction which only accesses R. Assume the following scenario: $\mathcal{M}$ executes T_{i1} and commits at time t_0. T_{i2} starts immediately, locks R for a period of ϵ and commits. $\mathcal{M}$ increases ω_i by δ, and immediately starts T_{i3}. T_{i3} runs conflict-free for a time period d and then provokes a conflict with T_l at time $t_3 = t_0 + \epsilon + d$, where it holds that $\omega_l(t_3) < \omega_i(t_3) < \omega_l(t_3) + \delta$. We can further assume that if the programmer would use J_i instead of J'_i then T_{i3} would also run from time t_0 to t_3 provoking the same conflict with T_l. However, J_i would lack the additional priority δ that was granted J'_i for committing T_{i2}, i.e., for an ϵ

small enough, it holds that $\omega_l(t_3) > \omega_i(t_3)$. T_{i3} would abort and prolongate the execution time of J_i. Since introducing artificial delays to J_i could not make it perform as fast as J'_i it follows that $\mathcal{M}$ does not punish unnecessary locking.

(iv). For simplicity we assume here that the time needed for rolling back an aborted transaction is negligibly small. The proof extends easily to the general case. Let T'_{ij} start at $t_u - \epsilon$, and provoke a conflict with T_k at time t_u, and $\omega_i(t) < \omega_k(t)$. $\mathcal{M}$ aborts T'_{ij} and increases ω_i by δ. $\mathcal{M}$ immediately restarts T'_{ij}. Assume that at time $t_u + \epsilon$, T'_{ij} provokes a conflict on with T_l and $\omega_l(t_u + \epsilon) < \omega_i(t_u + \epsilon) < \omega_l(t_u + \epsilon) + \delta$, after $t_u + \epsilon$, T'_{ij} runs conflict-free until commit. If the programmer of J_i would use T_{ij} instead of T'_{ij} then T_{ij} would not abort at time t_u and $\omega_l(t_u+\epsilon) > \omega_i(t_u+\epsilon)$. T_{ij} would thus abort at time $t_u + \epsilon$ and prolongate the execution time of J_i. There is no way delays could make T_{ij} perform as good as T'_{ij}. $\mathcal{M}$ does not punish unnecessary locking. □

6.4 Non-Priority Based Contention Management

One example of a CM that is not priority-based is Randomized (cf. [85]). To resolve conflicts, Randomized simply flips a coin in order to decide which competing transaction to abort. The advantage of this simple approach is that it bases decisions neither on information about a transaction's history nor on predictions about the future. This leaves programmers little possibility to boost their competitiveness.

Theorem 6.17. *Randomized is GPI compatible.*

Proof. We compare a J_i with J'_i under all possible environments $\mathcal{E}_{-i}$. Let $\mathcal{E}_{-i}$ include the CM's randomized decisions, i.e., if J'_i and J_i provoke a conflict at the same time with the same competing jobs then we compare the execution times of J'_i and J_i for both coin flips separately. For ease of notation let again $E := \mathcal{E}_{-i} \cup \{(J_i, t_i)\}$, and $E' = \mathcal{E}_{-i} \cup \{(J'_i, t_i)\}$.

Partitioning. In a first step, we show that Randomized, denoted by $\mathcal{M}$, rewards partitioning. Let J_i, J'_i, and k be such that $J'_i = \mathit{Combine}(J_i, k)$. We distinguish three cases of the execution of T'_{ik}.

(A) T'_{ik} runs until commit.

(B) T'_{ik} is aborted before $t^{\mathcal{M},E'}(T'_{ik}) + d_{ik}$.

(C) T'_{ik} is aborted in the period

$$\left[t^{\mathcal{M},E'}(T'_{ik}) + d_{ik}, \quad t^{\mathcal{M},E'}(T'_{ik}) + d^{\mathcal{M},E'}(T'_{ik})\right].$$

Case (A). Since we assume the time needed for committing is negligible, J_i always locks a subset of the resources locked by J'_i. Thus, J_i provokes a subset of the conflicts provoked by J'_i. As $\mathcal{M}$ always decides in favor of J'_i, so it does for J_i. T_{ik} and T_{ik+1} both run to commit in the first attempt. We have that $d^{\mathcal{M},E}(T_{ik}) + d^{\mathcal{M},E}(T_{ik+1}) = d^{\mathcal{M},E'}(T'_{ik})$, and hence $d^{\mathcal{M},E}(J_i) = d^{\mathcal{M},E'}(J'_i)$.

Case (B). T_{ik} has the same conflicts as T'_{ik} and both are aborted at the same time. They are both restarted at the same time, and we can apply the argument recursively until case (A) or (C) occur.

Case (C). T_{ik} runs until commit, T_{ik+1} is started immediately and is aborted in the same conflict as T'_{ik}. T'_{ik} and T_{ik+1} are restarted. Let t_a be the time when T'_{ik}, or T_{ik+1} respectively are aborted. Let t'_{ik+1} be the time when J'_i has successfully completed all operations corresponding to T_{ik} after the restart. Employing J'_i instead of J_i coincides with delaying T'_{ik} from t_a until t'_{ik+1}. In order to show that Randomized rewards partitioning we can use the same argument from Lemma 6.10, namely that starting immediately is the better strategy than waiting, although Randomized is not a priority-accumulating CM. To show this for Randomized is much easier. An adversary can provoke the same conflicts for a transaction, if it is started immediately, or if it is delayed for some time Δ. Since in any conflict, the probability of winning is the same, the expected runtime increases by Δ when the transaction is delayed.

Unnecessary Locking. In a second step, we show that $\mathcal{M}$ punishes unnecessary locking. Let J_i and J'_i be two jobs that are exactly the same except for one contained transaction T_{ij}, or T'_{ij} respectively. Let T'_{ij} have an unnecessary lock of resource $R \in \mathcal{R}$ compared to T_{ij}, and $d_{ij} = d'_{ij}$. If $\mathcal{E}_{-i}$ is such that the unnecessary lock provokes no conflict both jobs achieve the same execution time. If the unnecessary lock provokes a conflict and $\mathcal{M}$ decides in favor of T'_{ij} then the lock does not change the course of T_{ij}'s execution either. If $\mathcal{M}$, however, decides against T'_{ij} it is aborted and T_{ij} continues. T'_{ij} is restarted. If T_{ij} runs until commit, playing T_{ij} yields a better execution time. Otherwise, let t_{last} be the time when T'_{ij} is restarted for the last time, i.e., T'_{ij} commits at time $t_{last} + d'_{ij}$. T_{ij} could also be delayed until t_{last}, and reach a commit time at least as good as T'_{ij}. This is since J_i would provoke a subset of the conflicts provoked by J'_i. As delaying is irrational, employing T'_{ij} instead of T_{ij} is irrational.

It remains to show that if $d_{ij} < d'_{ij}$ then J_i is still preferable to J'_i. Let T'_{ij} be exactly like T_{ij} except for one unneeded resource access during an interval $[t_u - \epsilon, t_u + \epsilon]$ which prolongates d'_{ij} by $\delta = d'_{ij} - d_{ij}$. If the execution environment is empty, $\mathcal{E}_{-i} = \emptyset$, we get that

$$d^{\mathcal{M},E}(J_i) = d^{\mathcal{M},E'}(J'_i) - \delta < d^{\mathcal{M},E'}(J'_i).$$

If $\mathcal{E}_{-i}$ is such that the unnecessary lock provokes a conflict in which $\mathcal{M}$

decides for T_{ij}, the same effect would be achieved by introducing a delay to T_{ij} in the interval corresponding to $[t_u - \epsilon, t_u + \epsilon]$. If T'_{ij} is aborted, though, and T_{ij} runs until commit in the first attempt, choosing J_i yields a better execution time. If T_{ij} does not run until commit in its first execution, let t_{last} be the time when T'_{ij} is restarted for the last time. By introducing a delay in the interval corresponding to $[t_u - \epsilon, t_u + \epsilon]$, and additionally postponing the start of T_{ij} until t_{last} the programmer of J_i could reach a commit time at least as good as T'_{ij}. This is again because T_{ij} would provoke a subset of the conflicts that T'_{ij} provokes, and since $\mathcal{M}$ would make the same decisions T_{ij} would also win all conflicts. As introducing artificial delays is irrational the claim follows. □

Note that in order for the proof to work, the Randomized CM must schedule consequent transactions gapless. Thus, Theorem 6.17 holds for optimistic contention management. If a non-optimistic contention manager would entail a non-negligible gap between two consecutive transactions, however, then partitioning would not be rewarded. This is easy to see since in an empty environment, a fine grained job would yield a longer execution time than a version that combines some contained transactions.

Unfortunately, in terms of practicability, it is not a good solution to employ such a simple Randomized CM, although it rewards good programming. The probability $p_{success}$ that a transaction runs until commit decreases exponentially with the number of conflicts, i.e., $p_{success} \sim p^{|C|}$ where p is the probability of winning an individual conflict and C the set of conflicts. However, we see great potential for further development of CMs based on randomization.

6.5 Simulations

To verify our theoretical insights, we implemented selfish threads in DSTM2 [46], a software transactional memory system in Java, and let them compete with the threads originally provided by the authors of the included benchmark under several different contention managers. DSTM2 is an experimental framework that provides some basic CMs, and allows one to implement custom CMs easily.

6.5.1 Setup

In particular, we added a subclass `TestThreadFree` to the benchmark class `dstm2.benchmark.IntSetBenchmark` that uses coarse transaction granularities, i.e., instead of just updating one resource a selfish thread updates several resources per transaction at once.

```
while (true) {
   thread.doIt(new Callable<void>() {

      @Override
      public void call()

         // access dummy resource <priority> times
         Factory<INode> factory = Thread.makeFactory(INode.class);
         INode nd = factory.create();
         for(int k=0; k < priority; k++){
            nd.setValue(k);<

         // access shared resource <granularity> times
         Random random = new Random(System.currentTimeMillis());
         for(int i=0; i < granularity; i++){
            intSet.update(random.nextInt(TRANSACTION_RANGE))
         }
      }
   });
}
```

Figure 6.5: *Selfish thread. The call() method is executed as a transaction by the STM.*

See Figure 6.5 for the code executed by the selfish threads and Figure 6.6 for the collaborative threads' code. The latter is what we call "good code", as it only performs one action per transaction and thus avoids unnecessary locking. We added a mechanism to the selfish threads that attempts to build up priority before accessing the shared resource. To this end, it simply creates a dummy resource and updates it a number of times. When the system is managed by Timestamp- or Karma-like contention managers this could be an advantage: priority is built up in a conflict-safe environment and once the thread accesses the truly shared resources, it has higher priority than most of its competitors. Hence a selfish programmer can vary two parameters, the transaction granularity γ and the priority π it tries to build up before actually starting its work.

We tested and compared the performance of selfish threads with collaborative threads in two benchmarks. In both, there is a total number of 16 threads which start using a shared data structure for 10 seconds before they are all stopped. In the first benchmark, the threads all work on one shared ordered list data structure, in the second, they work on a red–black tree data structure. All operations are update operations, i.e., a thread either adds or removes an element. We ran various configurations of the scenario in both benchmarks managed by the Polite, Karma, Polka, Timestamp or the

```
while (true) {
   value = random.nextInt(TRANSACTION_RANGE);
   thread.doIt(new Callable<void>() {

      @Override
      public void call() {
         intSet.update(value);
      }
   });
}
```

Figure 6.6: *"Good" thread. The call() method consists of only one update call.*

Randomized contention manager. The variable parameters were

- the number of selfish threads, 0, 1, 8, or 16 among the 16 threads,
- their transaction granularity $\gamma \in \{1, 20, 50, 100, 500, 1k, 5k, 10k, 50k, 100k, 500k, 1M\}$, and
- the number of initial dummy accesses $\pi \in \{0, 200, 500, 2000\}$ performed by the selfish threads.

The benchmarks were executed on a machine with 16 cores, namely 4 Quad-Core Opteron 8350 processors running at a speed of 2 GHz. The DSTM2.1 Java library was compiled with Sun's Java 1.6 HotSpot JVM. To get accurate results every benchmark was run five times with the same configuration. The presented results are averaged across the five runs.

6.5.2 Results

The results confirm the theoretical predictions that a selfish programmer can outperform and sometimes almost entirely deprive the collaborative threads of access to the shared resources if the TM system is managed by the Polite, Karma, Polka, or the Timestamp CM. With the Randomized manager on the other hand, the collaborative threads are much better off than the selfish threads (cf. Figure 6.7). In all of our tests, if the system was managed by Polite the selfish threads were always better off. Under Karma, they were better off in 92% of all cases, and if they used granularities γ of at least 20 operations per transaction they always performed better. With Polka, the selfish threads' success rate was 70% over all runs and 100% for $\gamma \in \{20, 50, 100\}$. Of all tests run with the Timestamp manager, selfish behavior paid off in 92% of the cases and in 100% if the granularity γ was at least 20.

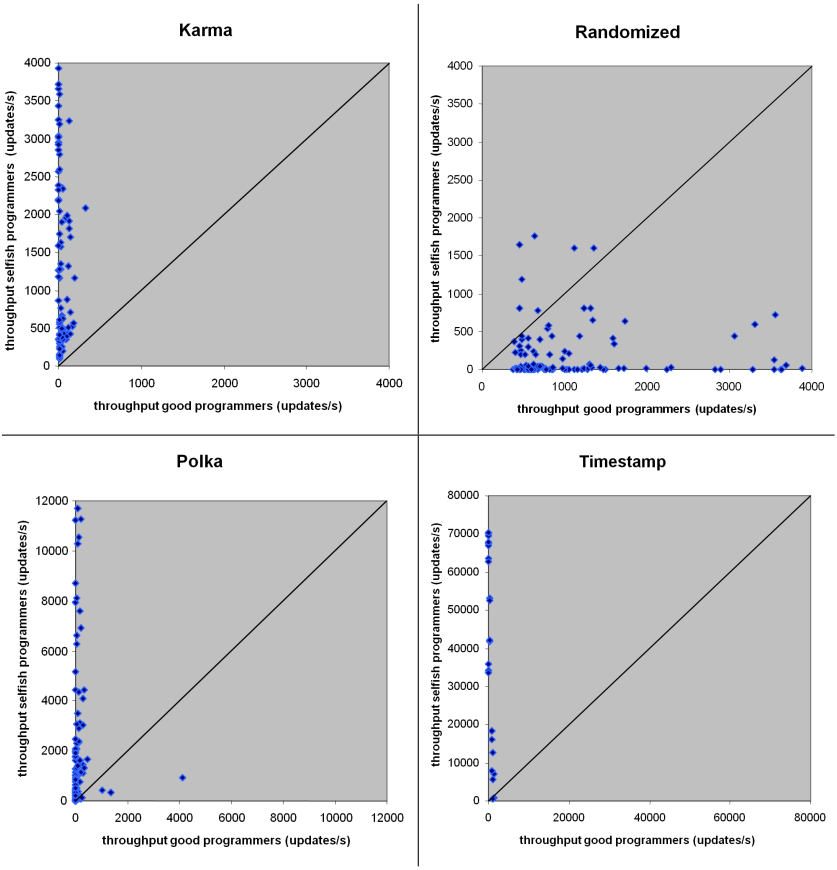

Figure 6.7: *Plot of all cases with either one or 8 selfish threads out of 16 threads simulated under a Karma, a Randomized, a Polka and a Timestamp CM. If a point is above the diagonal line this indicates that in the corresponding test run, the selfish thread had a larger throughput than the good thread that only employs transactions of granularity 1. The cases where $\gamma = 1$ are omitted.*

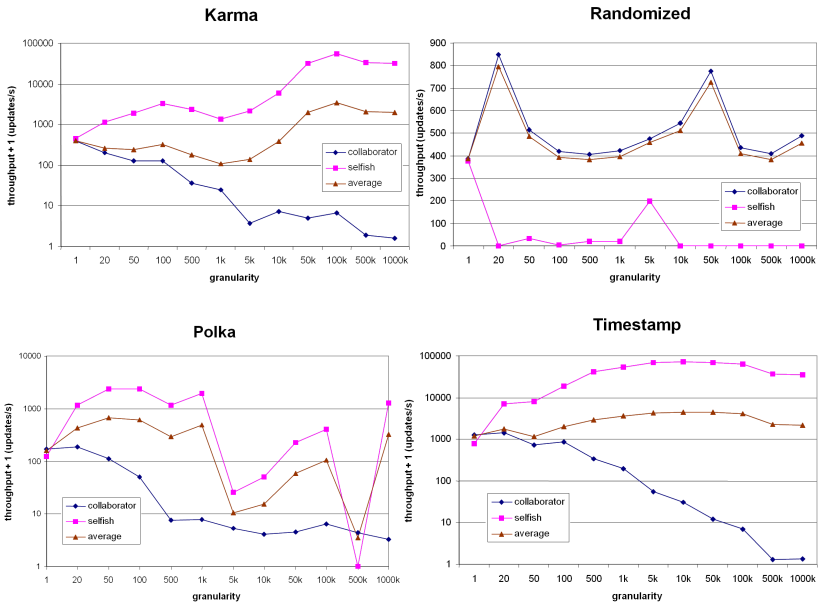

Figure 6.8: *Average throughput of a selfish and a collaborative thread in the red–black tree benchmark with 15 collaborators and one selfish thread. The selfish thread does not employ a priority boosting mechanism ($\pi = 0$). In addition to the collaborators' and the selfish thread's throughput, the average throughput of all 16 concurrent threads is depicted. Except for Randomized, we added 1 to the actual throughput and used a logarithmic scale.*

Under Randomized, selfish threads had a larger throughput in only 7% of all cases.

Further, our simulations suggest that the mechanism included to boost priority π before actually accessing the shared data does not influence the selfish thread's relative performance significantly. The transaction granularity however has a huge impact. Figure 6.8 shows the average throughput of both a selfish and a collaborative thread. In our experiments, a selfish thread's throughput was practically always higher than the collaborators' under the Karma, Polka and the Timestamp manager if it used a granularity of at least twenty update operations per transaction. This may in part be because a coarser transaction needs less overhead than a transaction with granularity $\gamma = 1$, however, with the Randomized contention manager, we see that even a transaction with a granularity of only twenty updates is un-

likely to succeed. To a larger extent, this higher performance of the selfish threads derives from the fact that—except for the first update—they have higher priority than the collaborative threads. At first it might be surprising that the average throughput, i.e., the system efficiency, does not decrease when introducing more selfish programmers. However, with large granularities, there will usually be one transaction with very high priority. The latter is not endangered of being aborted by any other transaction, and hence runs to commit untouched. It seems that in our setting with high contention, one fast selfish thread locking the entire data structure is still quite "efficient". More so, caching mechanisms probably speed up the system when basically only one thread is working. With the appropriate level of contention, the effect of degradation in system efficiency would possibly show. Regardless of this inability to show the system degradation explicitly, it is obviously not desirable for a multithreaded program to basically have only one thread running.

Note also the break in the throughput increase between $\gamma = 1000$ and $\gamma = 5000$ with the Polka manager. This is probably caused by the mechanism included in Polka which allows a transaction trying to access a locked resource to abort the competitor after a certain number of unsuccessful access attempts. This seems to happen much more often if the selfish programmers use granularities higher than 1000.

Part III

Peer-To-Peer File Sharing Systems

Chapter 7

P2P File Sharing – A History of Successful Mechanism Design

Peer-to-peer (p2p) file sharing protocols have enjoyed a great popularity among Internet users ever since their introduction in the late 1990s. They allow network users with limited upload capacity to make content available to a large community. Nowadays, providers of digital TV and Internet-based global social networks like Facebook and Twitter use the p2p file sharing technologies for internal content distribution.

7.1 History

Napster can be considered the first p2p file sharing system. It was founded in 1999 and primarily meant for sharing MP3 audio files. Although it was a p2p system in the sense that peers downloaded files directly from other peers, Napster relied on a central component, namely on central servers that stored the information about which peers offer which files. A user could connect with the server, search for files, and then download desired content directly from an offerer. After the operators of Napster were convicted of abetment of copyright infringement by their users in the historic law suit against the music industry, the service was shut down in 2001.

Second generation systems like Gnutella and Kazaa can be considered proper p2p systems as they implemented also the file lookup mechanism in a decentralized fashion. Thereby, they could also largely avoid the legal difficulties. The first version of the *Gnutella* protocol implemented the search for content with a simple flooding echo approach in the network of all participants. Since such an approach scales badly, a second version of Gnutella released in 2002 employed a more hierarchical look up mechanism,

and replaced TCP with UDP.[1] Gnutella's design presupposes that queries are forwarded by the peers, although an intermediate peer does not profit from helping the query initiator finding a peer that has the desired content. Moreover, Gnutella did not offer any incentives to actually share files with others. Adar and Huberman find in their 2001 paper [3] that about 70% of Gnutella users do not provide any files, and that many queries are not properly forwarded. The FastTrack protocol used by *Kazaa* took a similar approach like Gnutella and organized the peers in one large network. To (partially) solve scalability issues they introduced supernodes to basically reduce the diameter of the network. Kazaa tried to establish a mechanism to incentivize contribution through differentiation of quality of service: when requesting content peers announce their "participation level"; remote peers offer a prioritized service to participants that claim high contribution levels. However, the participation level is not attested in any way. Thus, the system could be easily cheated by announcing a wrong participation level. Moreover, peers had no incentive to become supernodes, i.e., to relay queries of other nodes. Other second generation p2p file sharing systems employed mechanisms where peers can earn credits for their contributions. Credits can in turn be reimbursed for higher quality of service. *eMule*, a p2p application that connects to the *eDonkey* and the *Kad* networks, implements a lightweight, pair-wise credit system. There are also systems with more complex payment-based incentive mechanisms such as the *Karma* system [97] or the now defunct *MojoNation*. However, these solutions typically require either a central administration or a distributed storage for the credits, which is prone to cheating and attacks (e.g., Sybil attacks, whitewashing, etc., see also [54]).

The third generation of p2p networks was heralded with the now predominant p2p file sharing system: BitTorrent.

7.2 BitTorrent

BitTorrent is a peer-to-peer file sharing protocol developed by Bram Cohen in 2001. The novelty of BitTorrent was to organize peers who are downloading the same content into *swarms* of peers. Inside a swarm, peers can trade parts of the content among each other, and thus, peers can upload and download from each other simultaneously. Moreover, by a differentiation in quality of service depending on the download rate experienced from a certain trading partner, BitTorrent establishes a mild incentive for peers to contribute.

[1] The issue of scalability was properly solved by several systems in 2001: P-Grid [1], Chord [92], Can [79], Tapestry, Pastry [84] are essentially all implementations of distributed hash tables that promise a logarithmic complexity for all basic operations; search, insert, and delete. Plaxton et al. [76] proposed similar techniques already in 1999.

The BitTorrent protocol has become increasingly popular during the first decade of the new millennium. Numerous BitTorrent clients have been implemented, and the protocol has been enhanced with several extensions. The user base of the BitTorrent protocol was estimated at 100 million users in the beginning of 2011,[2] and measurements indicate that the share of Internet traffic due to BitTorrent reached between 43% and 70% in 2009.[3] Latest measurements by the Canadian broadband management company Sandvine observe that the internet traffic due to peer-to-peer file sharing has dropped slightly in Northern America and Asia as the popularity of so-called real-time entertainment services like Netflix or Youtube increases.[4] BitTorrent still accounts for about 50% of all upstream traffic, and for about 15% of the daily aggregate and peak aggregate traffic.

7.2.1 The BitTorrent Protocol

The first step to distribute content with the BitTorrent proptocol is to create a torrent *metafile*, i.e., a file that contains information about the shared content including file names, and hashes of the file parts for integrity verification. The data shared in a torrent can consist of one or several files. The entire content is split into pieces of constant size between 64 kB and 4 MB, depending on the total amount of data. Additionally, the publishing peer adds the URL of one or several *trackers* to the metafile. A tracker is a server that tracks swarms, i.e., it stores the addresses of all peers currently in the swarm. Upon request, a tracker provides a peer with a subset of the addresses. In a next step, the distributing peer starts seeding the torrent. A peer is called *seeder* in a swarm if it possesses the entire content, and *leecher* otherwise. The seeder now announces itself to the trackers specified in the metafile. Moreover, she publishes the torrent metafile for other BitTorrent users to find and download. Metafiles are mainly published at so-called *torrent discovery websites*, but they can be distributed in any way. Peers who are interested in the content download the torrent metafile and start the download by requesting a list of peer addresses in the respective swarm from the trackers announced in the metafile. The peer thus learns about potential trading partners, and it can contact the peers directly thereafter.

Typical BitTorrent clients build up TCP connections to up to 50 peers per swarm, however, they actively trade on only a subset of peers simultaneously. The size of this so-called *active set* is usually limited to about 10 slots. A peer splits all available upload bandwidth equally among the connections

[2] *BitTorrent turns ten*, by Matt Hartley, financialpost.com.

[3] This is according to Ipoque's *Internet Study 2008/2009*. Sandvine presents considerably lower numbers at around 20% to 40%.

[4] See Sandvine's *Global Internet Phenomena Report* of Fall 2011.

in the active set and serves all valid requests for desired file blocks on these connections. Most BitTorrent clients use a *rarest-fist* policy when requesting parts of the file, i.e., by aggregating the frequency of each file block over all connected peers it determines the rarest missing parts that can be provided by the respective peer. Connected peers keep each other informed about their download progress by exchanging their *bitfields* at first contact; a bitfield is a bitmap indicating which file parts the peer has already received and which she has not; after the first contact, peers send *have* messages containing the indices of newly received blocks. The peers in the active set are assessed periodically in terms of the download rate received. The peers that have contributed the least are *choked*, i.e., removed from the active set, and replaced with other peers chosen randomly from the choked connections. This randomized replacement is referred to as *optimistic unchoking*, and it serves the purpose of exploring the swarm, and thus finding the best trading partners. Technically, to tell a peer that it may make requests for data, an unchoke message is sent. When a peer stops serving requests to a previously unchoked peer, it sends a choke message.

When a peer has downloaded all the desired pieces it is supposed to switch from leecher mode to seeder mode and to continue providing free data to leechers. Obviously, when a client is in seeder mode it does not make the same assessment when deciding which peers to unchoke. Rather than by the download rate, a seeder rates the peers in its active set by the experienced upload rate. Alternative implementations use a round-robin algorithm to decide which leecher to unchoke next.

7.2.2 BitTorrent's Incentives

A major reason—if not the predominant reason—for BitTorrent's remarkable popularity compared to earlier p2p file sharing protocols is arguably the newly imposed incentive structure: BitTorrent considerably increases the incentives for peers to upload a significant share of the data downloaded from other peers. Hence, the more contributors the system has the better it performs, and the better is the user experience, which again amplifies the popularity.

BitTorrent achieves its beneficial incentive structure by two measures:

- it brings peers together that are mutually interested in the content of each other, and
- it introduces service differentiation based on the peers' contribution.

The combination of keeping only a set of active trading partners and choking slow peers leads to a situation where peers trade actively with peers that have similar upload rates in the long run. Notice that a peer does not only

choke the slowest peers in its active set, but it is choked itself by peers that can keep up active trades with faster peers. For further analyses see e.g., [50], which relates BitTorrent's trading mechanism to Iterated Prisoner's Dilemma tournaments, or [57], which proposes to model it rather as an iterated auction. BitTorrent's trading mechanism among leechers is sometimes considered a Tit-for-Tat mechanism in the broader sense. *Tit-for-Tat* denotes the trading behavior to cooperate in a first exchange, and to always do what the trading partner does afterwards (cooperate or defect). A system implements a *Tit-for-Tat mechanism* if it is the dominant strategy to behave in a Tit-for-Tat manner in that system. Note that the incentives to contribute are only intact for leechers trading with leechers. Seeders, however, provide data completely without receiving a service in return. Seeding is completely altruistic, unless the seeder has an interest in distributing the content.

7.3 Is BitTorrent the Last Conclusion of Wisdom?

Apart from the fact that it is purely altruistic to remain in a swarm as seeder after the download of the desired content has completed, BitTorrent's incentive structure has some other weaknesses. The selfish client *BitTyrant* [75] illustrates that a peer can obtain considerably more than he contributes also when dealing with leechers only. It exploits the fact that the service differentiation mechanism knows only two qualities of service, i.e., a peer is either choked or unchoked. For each peer, there is a threshold value of upload bandwidth that is needed to get the higher quality of service. In particular, a peer must upload just enough data in order not to be the slowest peer in the remote peer's active set. From a selfish point of view, any bandwidth provided above the threshold value is in vain. BitTyrant thus adapts the upload bandwidth so as to approximate the threshold value. Piatek et al. [75] as well as Levin et al. [57] conclude that a tit-for-tat mechanism, or the strategy to provide a connected peer with roughly the same upload rate as the download rate respectively, would discourage selfish peers from cheating.

The free riding client *BitThief* [60] proves that it is even feasible to download without uploading at all: by connecting with as many peers as possible, and by always claiming it has no file blocks yet, BitThief is optimistically unchoked often enough to receive download rates comparable to the rates of normal BitTorrent clients. The authors of BitThief also propose resorting to a tit-for-tat mechanism [61], however, they project that, by introducing such a strict policy into BitTorrent as is, the system's market liquidity might decrease considerably. If peers never give out data without immediately receiving data in return, *starvation* is bound to occur more often, i.e., situations where a peer does not find a trading opportunity are more frequent. The *bootstrap problem* denotes one such starvation issue that is hardly avoidable

in a strict tit-for-tat system: A peer has no data initially. Thus it cannot start trading. Approaches to solving starvation problems typically increase the number of trading opportunities. Locher et al. [61], e.g., amend the liquidity of a system based on tit-for-tat trading by source coding techniques. Thus, the diversity of file blocks is increased drastically, with the effect that two leechers in a swarm are usually highly likely to have an interest in the blocks of each other. Furthermore, they mitigate the problem of bootstrapping by assigning a consistent set of free bootstrapping blocks to a peer by means of a hash function based on the receiver's IP address. Their protocol is implemented in the BitThief client, and used among the BitThief clients in addition to the protocol that downloads content from normal BitTorrent clients.

Orthogonal to the approach of increasing trading opportunities by coding techniques, the next chapter investigates the potential of introducing trades across swarms, and moreover, introducing trades along cycles of interest to increase a tit-for-tat system's market liquidity. If the liquidity, and thus the efficiency of such a file sharing system, can be boosted considerably, we believe that the stronger incentive structure could cause a thus enhanced barter system to outperform BitTorrent.

Chapter 8

Cyclic Tit-for-Tat Trading

A restricting factor of BitTorrent-like file sharing systems is that they employ an *intra-swarm trading* policy, i.e., blocks are only traded within the same swarm: consider a situation where two peers p_1 and p_2 are in the same two swarms, S_1 and S_2. Let p_1 have completed the download of the content shared in S_1, and let p_2 have completed the download of the content in S_2. With intra-swarm trading, p_1 and p_2 have no file blocks to trade with each other. However, if the system allows them to trade blocks of one swarm for blocks of another swarm, i.e., if they can employ *inter-swarm trading*, peer p_1 can provide p_2 with blocks from S_1 while p_2 can provide p_1 with blocks from S_2 in return. Given that there are no other peers to get the content from the described situation leads to complete starvation in an intra-swarm system, whereas the peers can download the desired content completely in an inter-swarm system. Obviously, this example is rather extreme, but there is the potential that allowing inter-swarm exchanges increases the number of trading opportunities in many situations, and thus boosts performance. This is, unless the peers in the considered file sharing system are typically only part of one swarm simultaneously. Fortunately, this is not the case: measurements of the BitTorrent system [98] show that about half of the peers in BitTorrent are active in multiple swarms at the same time, as illustrated in Figure 8.1. Hence, *inter-swarm trading* may result in a higher system throughput.

In addition to direct inter-swarm trading, further trading opportunities can arise when peers may trade along *cycles of interest*, inside and across swarms. Three peers p_1, p_2, and p_3 form a cycle of interest if p_1 is interested in data from p_2, p_2 is interested in data from p_3, and p_3 is interested in data from p_1. In such a cycle, fair trading is feasible since p_1 can provide p_3 with data, p_3 can provide for p_2, and p_2 can provide for p_1 (cf. Figure 8.2). Our measurements in live BitTorrent swarms reveal that even on a subset

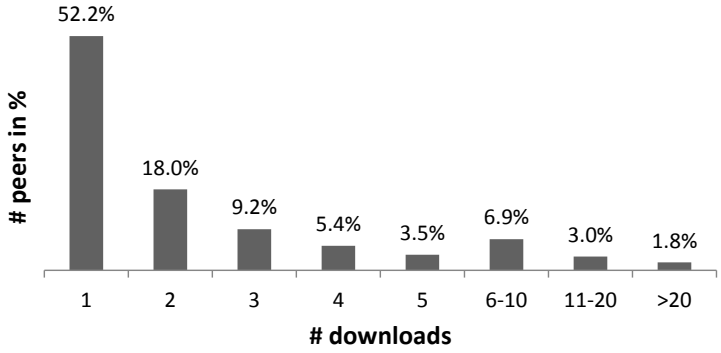

Figure 8.1: *Distribution of the number of downloads per peer in BitTorrent.*

of the BitTorrent system, the number of cross-swarm *trading cycles* is large. Moreover, the number of cycles grows fast with increasing cycle length: each additional hop increases the number of cycles by 3-4 orders of magnitude, as shown in Figure 8.3. In the remainder, we will refer to a trading cycle consisting of k peers as a *k-cycle*. Figure 8.4 further illustrates the potential of trading on cycles as it shows that a random peer is part of a large number of inter-swarm cycles with a probability of roughly 1/3 even if we only consider a subset of 1000 swarms.

In the following, we study to what extent we can boost the market liquidity of a peer-to-peer system by allowing multi-lateral, cyclic trading *across swarms*, without compromising the basic tit-for-tat incentive mechanism. While it is clear that in the best case, the relative throughput increase can be unbounded (in particular when there are no intra-trading opportunities), we focus on the throughput gain in practical scenarios. Based on data collected from real swarms, we conduct simulations to study the achievable throughput under different trading strategies and in different scenarios.

The results indicate that the throughput of a peer-to-peer system can indeed benefit from trading on cycles. Interestingly, these benefits are obtained already for fairly short cycles, involving up to three nodes. Longer cycles do not lead to a substantially larger performance but incur a large communication and computational overhead. For our evaluation, we also derive a model for the peers' download preferences combining preferential-attachment and co-occurrence principles. Moreover, we identify certain pitfalls, such as the problem of redundant downloads, and we propose techniques that considerably mitigate this problem. Methods to reduce the communication overhead

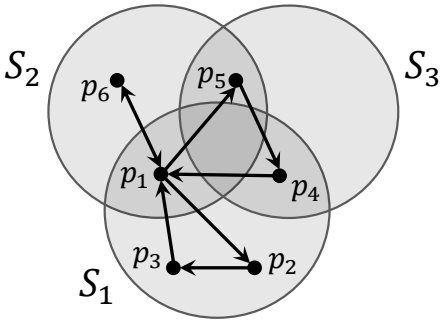

Figure 8.2: *Possible trading situation for a peer p_1 participating in two overlapping swarms S_1 and S_2: in addition to the direct bilateral trades within a swarm (e.g., with peer p_6 in S_2), p_1 can exchange data along intra-swarm cycles (e.g., (p_1, p_2, p_3, p_1) in S_1). Moreover, it can trade pieces with one or more peers in different swarms (e.g., along the cycle (p_1, p_5, p_4, p_1)).*

---

are also discussed. Finally, we outline a distributed implementation of our techniques.

8.1 Model

We consider a peer-to-peer system where peers interested in the same content are organized into *swarms*, i.e, in every swarm specific content is shared. This content is divided into multiple data *blocks*, and a block is the basic unit of trade. In practice, a peer typically joins a swarm by contacting a so-called *tracker*, a system entity whose main objective is to keep track of the peers in the swarm. Once a new peer informs the tracker that it wants to join, the tracker stores this peer's address and returns a list of addresses of other peers in the swarm. Instead of periodically querying the tracker, the peers can discover additional peers by exchanging addresses amongst themselves. In the first part of our evaluation, we make the simplifying assumption that peers are fully connected inside a swarm. The effect of trading with only a small subset of all possible peers is discussed in Section 8.3.4.

Formally, we are given a set $\mathcal{P}$ of peers and a set $\mathcal{S}$ of swarms. Each peer $p \in \mathcal{P}$ joins a subset $\mathcal{S}_p \subseteq \mathcal{S}$ of swarms over time, where $\mathcal{S}_p$ is the set of swarms whose content peer p is interested in. We assume that a peer has a certain *upload* and *download bandwidth* available. While a peer p has incomplete downloads it tries to acquire file blocks by direct or indirect trades

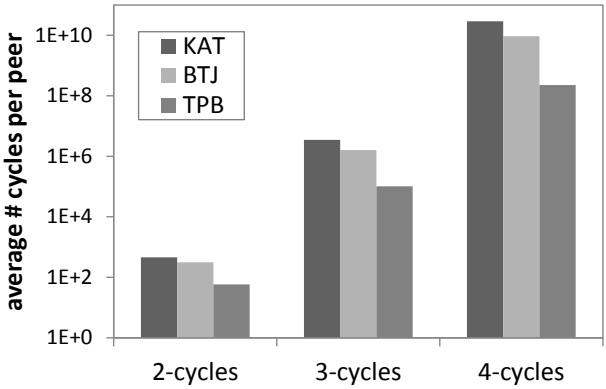

Figure 8.3: *Average number of inter-swarm k-cycles (cycles of length k) per peer for $k = 2, 3, 4$ for data sets of the top 1000 video torrents of the three torrent discovery sites with the highest Alexa rank,* `ThePirateBay.org` *(TBP),* `BTJunkie.org` *(BTJ), and* `kat.ph` *(KAT). The number of 4-cycles is approximated within an error margin of 1%. Note the logarithmic scale.*

with peers that have interesting blocks to offer. A snapshot of the current peer interests can be modeled using a (dynamic) directed graph, which we call the *demand graph*: The node set is $\mathcal{P}$, and there is a directed edge from peer p_1 to peer p_2 if p_1 is interested in at least one block offered by peer p_2 in some swarm. Each peer can obtain a local approximate view of the demand graph by communicating with other peers and thus trade blocks along interest cycles of various length (see Section 8.2).

While a peer is in the process of downloading a certain content, it is called *leecher*, whereas a peer that has already obtained all blocks is called *seeder* in the respective swarm. Naturally, a peer participating in multiple swarms can be a seeder in some swarms and a leecher in others at the same time. Once a peer p has acquired the content of all swarms in $\mathcal{S}_p$, it leaves the system. Note that before leaving the system, it is in the peer's interest to stay in a swarm even after it has downloaded the corresponding content because there might still be opportunities to provide blocks from this swarm in exchange for blocks traded in other swarms. Thus, in a game-theoretic sense, seeding is a rational behavior. Compared to the standard intra-swarm trading, this incentive for peers to stay connected after they become seeders is a remarkable advantage of trading along cycles.

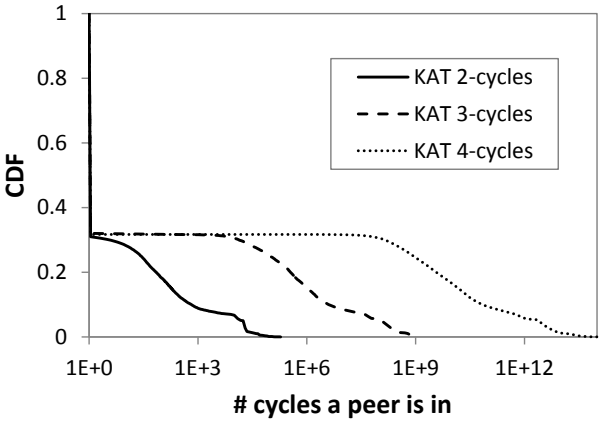

Figure 8.4: *Cumulative density function of inter-swarm cycles a peer belongs to; for all peers found in the top 1000 video torrents of KAT, with respect to cycles within these torrents.*

8.2 Algorithm

In this section, we introduce the core algorithmic aspects, that is, we discuss how a local approximation of the demand graph is computed and particularly how cycles are selected for trading. As it is too costly to maintain an approximation of the entire demand graph, each peer p only keeps track of its k-neighborhood for some constant $k \geq 1$, which is the set of peers that can be reached from p with at most k hops in the demand graph. This is achieved by regularly exchanging peer lists and information about locally available blocks with all peers in the k-neighborhood and the trackers. Naturally, the peers only know their immediate neighbors in the beginning, but they quickly get to know their k-neighborhood by communicating with the peers in their neighbors' peer lists. Given the k-neighborhood, a peer computes possible trading cycles by exploring the demand graph. Since we only consider cycles of short length, a brute-force approach is feasible to find cycles in the demand graph; however, heuristics may be used to prune the search space. The process of computing trading cycles is explained in more detail in Section 8.4.

The *trading policy* defines which cycles are used to share data blocks and how the upload capacity is allocated. The main policy that we investigate is the following.

`Cycle(`k`)`: The peer participates in *any* trading cycle that is of length at most k. The bandwidth is allocated equally among all active cycles. If the data flow on a cycle is diminished due to constraints by other peers, the unused bandwidth is allocated evenly among the remaining cycles.

Blocks are traded in a tit-for-tat-like manner on each cycle by having each peer p in the cycle send one block after the other to the peer that is interested in p's blocks, i.e., the blocks move opposite to the direction of the edges in the corresponding demand graph. In order to ensure fair trading, each peer maintains a balance between the number of uploaded and downloaded blocks for each active cycle. If the number of uploaded blocks is some constant τ larger than the number of downloaded blocks, the peer waits until the imbalance becomes smaller before sending out the next block. In our simulation, we set $\tau := 1$, which is the most restrictive choice. Once a peer loses interest in its trading partner in a cycle, the corresponding edge in the demand graph vanishes and trading on this cycle ceases. Whenever a peer wants to upload a certain block, the block is put into a FIFO queue, i.e., the blocks are sent sequentially, and the actual upload time depends on the available bandwidth at the peer. All peers use a *uniform* block selection strategy, i.e., when requesting a block from a neighbor a peer selects a random block out of the interesting blocks that are requestable and non-pending. If no such block is available the peer re-requests a random pending block, i.e. a block that has been requested but not yet received. The impact of this trading strategy on the system performance is evaluated in the following section.

8.3 Evaluation

The main objective is to evaluate the influence of the parameter k on the market liquidity, i.e., the goal is to quantify the impact on the overall throughput. Naturally, a larger k yields more potential trading cycles; on the other hand, the message complexity and the computational complexity grows rapidly with increasing k. Therefore, it is of paramount importance to choose the right value for k, taking the overhead into account. Apart from studying the influence of k, we also compare `Cycle(`k`)` to the `Intra-swarm` policy where only bilateral, intra-swarm exchanges are allowed, and the bandwidth is also allocated equally among all active trades. This comparison reveals the potential gain of trading on cycles. Furthermore, as the overhead of `Cycle(`k`)` may become quite large, simply because *all* cycles are considered, we propose refinements that significantly increase the efficiency.

8.3.1 Simulation

For our evaluation, we implemented an event-driven simulator that allows us to create multiple swarms, where peers can join and leave over time. The simulator models the execution on packet level, and both the size of the file shared in a swarm as well as the block size are parameterized. By default, we use 512 MB files divided blocks of size 512 kB.

For simplicity, we do not consider any bounds on the download bandwidth in our evaluation. However, we limit the *upload bandwidth* of each peer to 500 kB/s. In order to inject data into the system, we assume the presence of a designated *seeder* in each swarm that provides any leecher with free file blocks at a constant rate of 10 kB/s.[1] These publishers do not engage in any trading otherwise.

In order to model practical systems as accurately as possible, we use a snapshot of the BitTorrent economy gathered by Zhang et al. [98] to determine the cardinality $D_p = |\mathcal{S}_p|$ for each peer p in our evaluation. In particular, we have computed the number of downloads for all peers in the data set and store these values in a data set $\mathcal{D}$. The total number of peers and swarms is determined using the same BitTorrent snapshot: Since there are 3.65 times more peers than swarms in the data set, we simulate 365 peers and 100 swarms in the first part of the evaluation.

A single run of the simulation proceeds as follows. The simulations start at time $t = 0$ with $n = 100$ swarms each containing only one designated publishing seeder. Each peer $p \in \mathcal{P}$ is assigned the total number D_p of downloads it will start during the simulation by randomly choosing a value from the set $\mathcal{D}$. The D_p swarms for each peer p are chosen uniformly at random from all swarms $\mathcal{S}$. Since a peer typically joins its swarms over time, we model the time when a peer p joins the next swarm in $\mathcal{S}_p$ using a Poisson process with parameter $\lambda = 10^{-1}\text{min}^{-1}$ until it has joined all D_p swarms, i.e., we assume that the intervals between join events follow an exponential distribution. Whenever a new download starts, a peer joins the corresponding swarm as a leecher. Recall that a peer never leaves a swarm until *all* its downloads are completed.

We conducted several simulations where all peers use the `Intra-swarm` policy, `Cycle`(2), `Cycle`(3), or `Cycle`(4). Note that although both `Intra-swarm` and `Cycle`(2) restrict the peers to bilateral exchanges, they differ in that the latter allows for inter-swarm exchanges. Each scenario is executed with ten different sample sets and the results are averaged unless noted otherwise.

[1] Existing systems typically rely on peers willing to provide blocks for free in order to solve the bootstrap problem. While some of these providers might have altruistic motives, others might have an interest in the actual dissemination of the content.

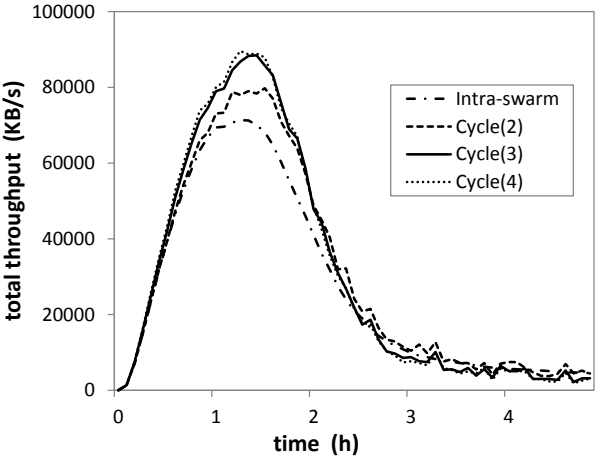

Figure 8.5: *Total throughput over time for 365 peers in 100 swarms.*

8.3.2 First Results

We start by analyzing the total throughput of all peers over time. Figure 8.5 depicts the throughputs for the four different trading policies. The results confirm our expectation that `Intra-swarm` achieves the lowest throughput because it is the most restrictive trading policy, and the throughput increases when larger trading cycles are used. Interestingly, using `Cycle`(2) already results in a significant improvement: The peak download rate increases by roughly 12%. The policy `Cycle`(3) achieves even better results (24%); however, `Cycle`(4) only slightly outperforms `Cycle`(3) (26%).

The distribution of the download rates is even more insightful. Figure 8.6 shows the average download rates per download on a logarithmic scale sorted in descending order. For each download, we computed the average download rate by dividing the file size by the duration of the respective download, i.e., the download completion time. It is clearly visible that the average rates improve substantially when inter-swarm trading is allowed. The download rates are

- 56.6 kB/s (median) and 55.2 kB/s (average) for `Intra-swarm`,
- 78.7 kB/s (median) and 216.9 kB/s (average) for `Cycle`(2),
- 89.9 kB/s (median) and 306.0 kB/s (average) for `Cycle`(3), and
- 92.4 kB/s (median) and 326.2 kB/s (average) for `Cycle`(4).

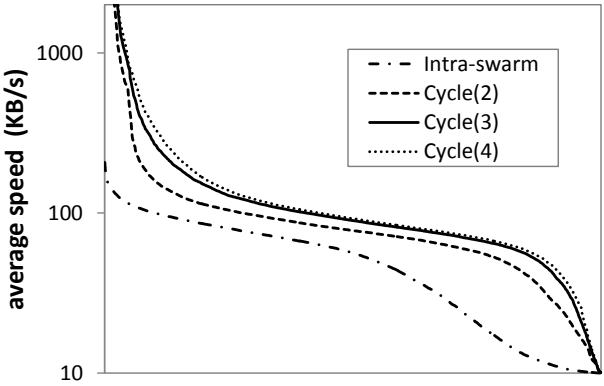

Figure 8.6: *Distribution of average speeds per download. Note the logarithmic scale.*

Thus, the average download rates are improved by a factor of 4 when allowing trades on 2-cycles, and by a factor of 5.5 when additionally using 3-cycles. The improvement in terms of the median download rate is 39% for `Cycle`(2) and 59% for `Cycle`(3). Again, the difference between `Cycle`(3) and `Cycle`(4) is negligible. We also recorded the download rates of all downloads in each individual simulation run: Compared to `Intra-swarm`, 84.4% of all downloads finish faster with `Cycle`(2), and the median improvement is 8.2%. When `Cycle`(3) is used, 96.9% of all downloads have a smaller completion time, and the median improvement is 14.7%. The same numbers for `Cycle`(4) are 97.4% and 16.1%, i.e., the numbers for `Cycle`(3) and `Cycle`(4) are again quite similar.

The investigation of download rates shows that while cyclic trading does not increase the peak throughput of the system tremendously (12% for 2-cycles and 24% for 3-cycles), the average download speeds, and therefore also the average download completion times, are improved significantly. One of the reasons is that cyclic trading especially helps at the beginning of a download when the peers do not need to get initial blocks from seeders only, as well as towards the end of a download when the final missing blocks are collected more efficiently thanks to inter-swarm trades. Additionally, Figure 8.6 shows that more downloads have an average rate close to the median rate when trading on cycles, i.e., the resource allocation is more balanced. Note that this also indicates a higher degree of fairness when trading on cycles.

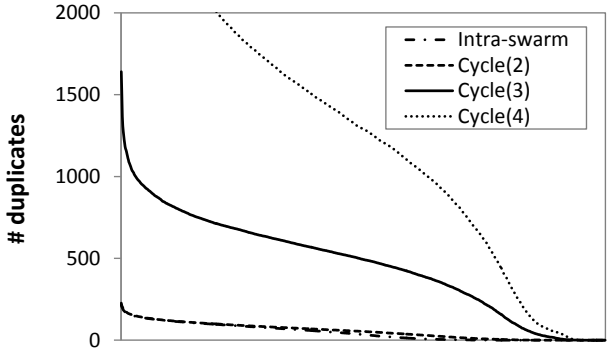

Figure 8.7: *Distribution of duplicates per download for each trading policy in descending order.*

8.3.3 Avoiding Redundancy

From our first experiments we can conclude that the increased liquidity due to additional trading opportunities along cycles indeed leads to an increase of the overall performance. However, there is an issue that needs to be addressed: The same blocks can be requested in different cycles, which results in redundant downloads. The reason is that, as defined in Section 8.2, a peer requests a *pending block* if there is no available block that has not been requested already. This behavior is reasonable to some extent as pending requests might remain unanswered due to network failures, or slow connections might be worth replacing with faster connections. Moreover, it guarantees that all members of a cycle are always willing to trade on the cycle as long as the cycle exists in the demand graph. Unfortunately, such a nonrestrictive policy on re-requests leads to an intolerable number of redundant blocks as soon as the peers trade on cycles longer than 2. Figure 8.7 shows that peers download up to 1500 redundant blocks for a download consisting of 1024 blocks when using `Cycle`(3), and up to 2000 blocks when using `Cycle`(4).

The redundancy problem arises because the number of trading cycles that a neighboring peer p appears in is often larger than the number of interesting blocks that p has to offer. As a countermeasure, we propose the following two modifications to the trading policy `Cycle`(k).

1. **Selecting Cycles:** Limit the number of active cycles per neighbor proportionally to the number of blocks it can provide.
2. **Probabilistic Re-Request:** Re-request pending blocks only with a certain probability ρ.

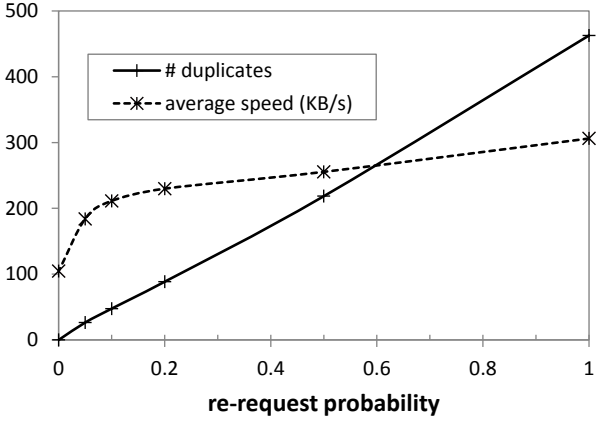

Figure 8.8: *Tradeoff between average number of duplicates and the average download rate with varying re-request probabilities for* `Cycle(3)`.

Figure 8.8 depicts the trade-off between probabilistic re-requests and the throughput: While the number of duplicates depends linearly on the re-request probability, the throughput grows quickly as long as ρ is fairly small. Thus, a smart re-request strategy can significantly reduce the number of duplicates without severely impacting the performance of the system.

Figure 8.9 shows the effect of the two countermeasures on the number of duplicates for `Cycle`(3). It is evident that probabilistic re-requests reduce the number of duplicates much more than limiting the selection of cycles. However, since the lowest number of duplicates is achieved when combining these two measures, we use both techniques in the following. In particular, we use a re-request probability of $\rho = 0.5$ for `Intra-swarm` and `Cycle`(2), and $\rho = 0.1$ for `Cycle`(3) and `Cycle`(4). According to our simulations, these values for ρ decrease the number of duplicates drastically without degrading the performance much. Another natural strategy to mitigate the redundancy problem would be to sort the cycles according to their "capacity", which we define as the number of blocks that could be traded along the cycle before the first peer loses interest and the cycle breaks. However, in our experiments, the performance gains were almost negligible and therefore not worth the additional algorithmic complexity.

In the following, we reassess the throughput benefits of our general block trading algorithms with our two anti-redundancy mechanisms in place. Regarding redundancy, probabilistic re-requesting and limiting the number of

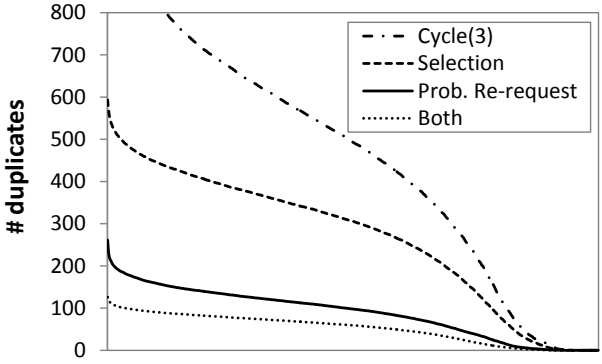

Figure 8.9: *Effect of selecting cycles and probabilistic re-requesting on the number of duplicates for* `Cycle(3)`.

active cycles ensures that the median number of duplicates per download is 30 (i.e., an overhead of 2.9%) or less for all policies simulated; the 99^{th} percentile is below 90 (8.8%), and the maximum number of duplicates is over 90 only for the `Cycle`(2) policy. Regarding performance, the adapted simulations yield median and average download rates of

- 36.6 kB/s (median) and 44.1 kB/s (average) for `Intra-swarm`,
- 74.7 kB/s (median) and 194.2 kB/s (average) for `Cycle`(2),
- 86.1 kB/s (median) and 214.5 kB/s (average) for `Cycle`(3), and
- 88.3 kB/s (median) and 191.7 kB/s (average) for `Cycle`(4).

Qualitatively, the results are very similar to the results without anti-redundancy measures in terms of average download rate: `Cycle`(2) and `Cycle`(3) perform 4.4 and 4.8 times better than `Intra-swarm`, respectively. The improvement in terms of the median rate is even higher, i.e., 104% for `Cycle`(2) and 135% for `Cycle`(3). Compared to the simulations without any redundancy-reducing measures, the loss in performance is small, e.g., the median download rate is reduced by less than 5% for all cyclic trading policies.

Finally, we would like to point out the interesting issue of *coordination* that occurs as soon as peers are willing to trade only on a subset of all the cycles they are part of. If a peer decides to trade on only a few of potentially thousands of cycles present, it can happen that there is at least one peer unwilling to trade on every cycle. Thus, the performance degrades simply because the peers do not agree on *which* cycles to trade. This phenomenon

is naturally more prevalent the more cycles there are to chose from, i.e., for larger k. The lack of coordination is also the reason for the fact that the overall average download rate is lower for `Cycle`(4) than for `Cycle`(3). Fortunately, this issue does not affect performance significantly if the peers use only cycles up to length 3 since still more than two thirds of all negotiations succeed.

8.3.4 Active Set Trading Policy

So far, we have assumed that the peers in a swarm are all connected to each other. While this is possible to a certain extent, the cost of interacting with too many neighbors can be large (especially if TCP connections are used). In the BitTorrent protocol, peers only trade actively with a small subset of neighbors, the so-called *active set*. A similar extension is also possible in our scenario. We have conducted experiments where peers trade with a small number of neighbors, in particular, we allow each peer to connect to only 10 peers per swarm. As a result, a peer can learn only about a subgraph of the demand graph, and trade only on cycles involving neighbors in the active set. Interesting peers are always added to the active set if the set is not full. Each peer periodically assesses the active peers in terms of the amount of data received since the last assessment and replaces the worst peer by another random peer in the swarm. Moreover, peers are replaced immediately when they become uninteresting.

Figure 8.10 depicts the net download rates (excluding duplicates) for different policies and their counterparts using the redundancy-reducing measures and active sets. In our simulations, active sets containing at most 10 peers lead to a throughput decrease of at most 5%, and combining active sets with redundancy-reducing measures results in a total loss of at most 10%. Since this performance decrease is modest, the refined `Cycle`(k) policies still greatly outperform the basic `Intra-swarm` policy.

Introducing active sets does not increase the number of duplicates downloaded: all policies exhibit a median redundancy of less than 3%, i.e., the median number of redundantly downloaded blocks over all downloads is 30, and the maximum redundancy is less than 10%. Furthermore, the *overhead* of building up the local view of the demand graph is small as only little data needs to be transferred. In our simulations, we start a breadth first search along the edges up to k hops in the demand graph whenever a new edge appears. We account for the communication overhead incurred by a peer p by aggregating the number of times an outgoing edge of p was traversed in all search processes, and multiply it by the number of bits needed to represent a respective search message.

In all simulation runs, the maximum overhead per peer was below 0.8% of

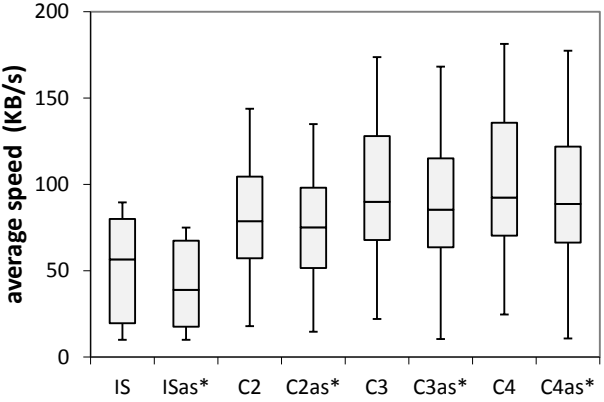

Figure 8.10: *Comparison of the net download rates distributions of the* `Intra-swarm` *(IS) policy and the* `Cycle`*(k) policies for* $k = 2, 3, 4$ *(C2, C3, C4) and their versions with active sets of size 10 as well as redundancy-reducing measures in place (ISas*, C2as*, C3as*, C4as*). The three horizontal lines of each box in a box plot depict the lower quartile, the median, and the upper quartile, and the end of the two whiskers represent the* 5^{th} *and the* 95^{th} *percentile.*

---

the content size downloaded by the peer, except for the scenario where peers employ `Cycle`(4). The overhead for `Cycle`(4) without redundancy-reducing measures is slightly larger at up to 2%, with active sets and redundancy-reducing measures, it grows to an intolerable level of 22%.

As a conclusion of our simulations with $|\mathcal{P}| = 365$ and $|\mathcal{S}| = 100$ we propose to use `Cycle`(3) with active sets, probabilistic re-request, and the cycle selection measure to achieve an improvement of more than 50% in terms of median download rate, and the average download rate increases by a factor larger than 3.7. This is compared to `Intra-swarm`, which exhibits 5% redundancy on average, whereas the proposed method exhibits only 2.6%. Moreover, using this method shortens the completion time of more than 93% of all downloads.

As a simpler alternative, one might also use `Cycle`(2) with the respective extensions to achieve an improvement of the average download rate of 32% (median) and 232% (average) with even less overhead and redundancy.

Finally, let us examine the relationship of uploaded and downloaded data volumes. Due to the tit-for-tat trading, the ratio of uploaded data against downloaded data should be close to one for most peers. Figure 8.11 shows

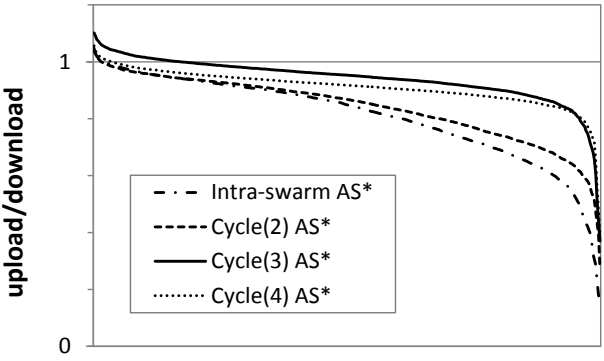

Figure 8.11: *Upload-download ratio per peer in descending order.*

that for all policies, most peers have a similar upload-download ratio indeed. Most peers have a ratio between 0.8 and 0.95. Note that these numbers do not include the ratios for the designated publishing seeders, as they only upload anyway. The designated seeders are also the reason why most ratios are below 1, i.e., the peers download more than they upload. Note also that there are some peers that have a very low ratio. Those peers are typically peers that enter a swarm at a late stage of the simulation,[2] when they have only one active download and the peers in the swarm are all in (selfish) seeding mode. They thus do not find any trading partners and end up receiving the entire content from the publisher at 10 kB/s.

8.3.5 Modeling Preferences

In this section, we propose and evaluate a more sophisticated and arguably more realistic model for the peer preferences. So far, we have assumed a uniform model of peer preferences: peers choose the swarms they join uniformly at random from the swarm set $\mathcal{S}$. This simple model ignores the fact that the users of file sharing systems typically have specific interests, which implies more clustered mappings between peers and swarms. We study the *clustering coefficient* distribution in the undirected graph consisting of the node set $\mathcal{P}$ and edges between peers that appear in at least one common swarm. The clustering coefficient of a peer is the number of edges between neighboring peers in this graph divided by the maximum number of such edges.

[2] There are some outliers in the produced schedule due to the exponential distribution of the Poisson process.

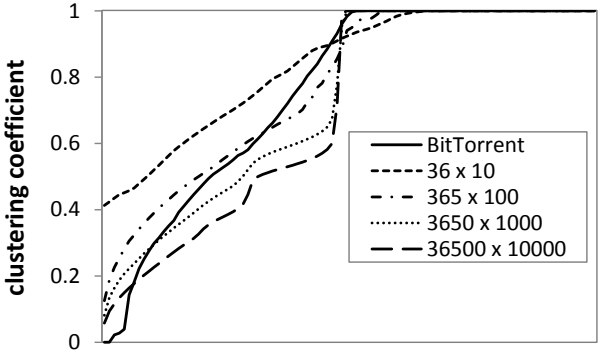

Figure 8.12: *Distribution of clustering coefficients among peers with uniform preferences for different scales $|\mathcal{P}| \times |\mathcal{S}|$.*

The uniform choice of swarms produces a demand graph whose clustering coefficients are too low if the number of peers and swarms is sufficiently large. In order to demonstrate this, we compare the clustering coefficients produced by our simulator with uniform preferences to the clustering coefficients we computed from the BitTorrent snapshot. As Figure 8.12 illustrates, if we keep the ratio of $n = |\mathcal{P}|$ and $|\mathcal{S}|$ constant at 3.65 the clustering coefficients decrease with growing n. For $n = 36$ almost all clustering coefficients are too high, whereas for $n = 3650$ the clustering coefficient of most peers with more than one download is significantly too low. The peers with clustering coefficients of 1 are mostly peers with only one download. The explanation is that with more swarms available the probability that two peers in one swarm also appear together in another swarm decreases although the average swarm size remains constant.

Consequently, to reflect user preferences and thus the true correlation between peers and swarms more precisely, we propose a preference model that combines the concepts of *preferential attachment*: a popular swarm is likely to become more popular in the future, and *co-occurrence*: if a peer already shares many other interests with the peers in a given swarm S, it is more likely to join S as well (for a motivation see, e.g., [59]). Whenever a peer p starts a new download it joins swarm S with a probability proportional to the number of p's neighbors in S (co-occurrence) and the size of S (preferential attachment), i.e.,

$$Pr[p \text{ enters } S] \approx \frac{(|N_p \cap S| + 1)^\alpha + (|S| + 1)^\beta}{\sum_{X \in \mathcal{S}} (|N_p \cap X| + 1)^\alpha + (|X| + 1)^\beta}$$

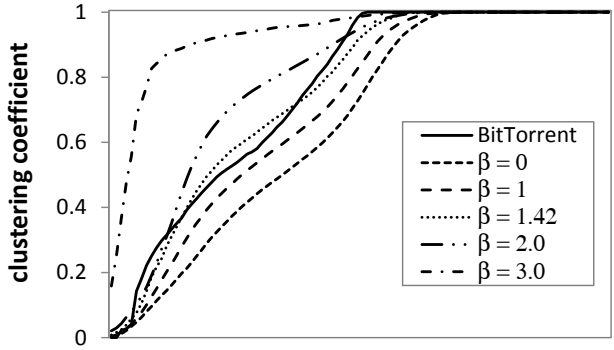

Figure 8.13: *Clustering coefficient distributions for* $|\mathcal{P}| = |\mathcal{S}| = 100$, $\alpha = 1$ *and varying* β. *The error is minimized for* $\beta \approx 1.42$.

where N_p denotes the set of p's neighbors. The parameters α and β allow us to model arbitrary combinations of the two concepts. For $\alpha = 0$, our preference model is a pure preferential attachment model, and for $\beta = 0$ it is a pure co-occurrence model. For $\alpha = \beta = 0$, we get a uniform distribution.

Given the proposed model, we can set up simulations that approximate both the clustering coefficient distribution and the swarm size distribution well by using a peer-swarm ratio of 3.65 and fit the clustering coefficient distribution by adapting α and β. However, as larger α and β values lead to larger clustering coefficients, we must consider a scenario where the uniform distribution results in clustering coefficients that are too low. As Figure 8.12 illustrates, this is the case for large peer and swarm sets when keeping the peer-swarm ratio constant. Since the computational complexity of the simulation increases quickly with the number of peers and swarms—and we also want to be able to fit the parameters for small networks—, the best option is to abandon the requirement to keep the peer-ratio set to 3.65. We found that by using $|\mathcal{P}| = |\mathcal{S}| = 100$ the swarms contain fewer peers on average than in the BitTorrent data; however, the clustering coefficients with uniform preferences are well below the BitTorrent clustering coefficients. Figure 8.13 depicts the clustering coefficient distribution observed in BitTorrent, and the clustering coefficient distributions produced by our simulator for various β.

We fitted the parameters α and β of the co-occurrence preference model using the method of least squares, i.e., α and β are set to values that minimize the error $\frac{1}{n}\sum_{i=1}^{n}(c_i - \hat{c_i})^2$, where $\hat{c_i}$ is the i-th lowest clustering coefficient in the simulation, and c_i is the clustering coefficient at the corresponding position in the measured clustering coefficient distribution. Our study indicates

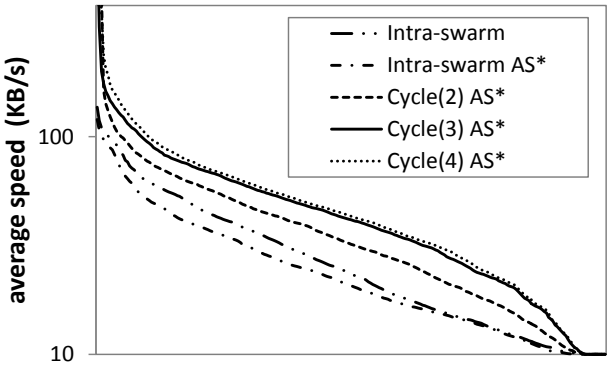

Figure 8.14: *Distribution of average speeds per download for* $|\mathcal{P}| = |\mathcal{S}| = 100$*,* $\alpha = 1$*, and* $\beta = 1.42$*. Note the logarithmic scale.*

that the choice of β has a large impact on the clustering coefficient distribution (cf. Figure 8.13), whereas the choice of α does not impact the clustering coefficients much for the relatively small numbers of peers and swarms used in our setup. The parameter α has a slight effect on the steepness of the curve.[3] As the error is similar for any $\alpha < 5$ by selecting the best value for β, we chose $\alpha = 1$ for simplicity. The respective optimal choice for β is 1.42, yielding an error smaller than 10^{-3}.

Figure 8.14 shows the download rates for the scenario with 100 peers and 100 swarms using $\alpha = 1$ and $\beta = 1.42$. The results are similar to those in our earlier experiments, and we can draw the same conclusions: `Cycle`(k) clearly outperforms `Intra-swarm` for any $k = 2, 3, 4$. Again, the figure shows that while the throughput rises with increasing k, the difference between `Cycle`(3) and `Cycle`(4) is marginal. We conjecture that the results also hold for a larger number of peers and swarms.

8.4 Distributed Implementation

After having evaluated the potential of cyclic trading by simulations, we now present a distributed protocol, CycT4T, that fills in the blanks in the high-level description of the algorithm in Section 8.2. In particular, CycT4T enables peers to find cycles and to negotiate on which cycles to trade in an efficient manner without revealing more private information than with an intra-swarm protocol.

[3] We believe the effect of co-occurrence will have a greater impact in larger systems.

CycT4T uses the usual mechanisms like tracker polling, peer exchange (PEX), or distributed hash tables (DHT) to discover peers. Initially, two peers exchange information about their locally available blocks. After that, the peers only notify each other when new blocks become available.

In order to keep a local view of the demand graph, each peer p maintains two tables. The `outTable`, which contains all (known) interesting peers (*out-neighbors*), i.e., the peers possessing blocks that p is interested in, and the `inTable` whose purpose is to keep track of all peers from which there is a directed path of length at most $k-1$ in the demand graph ending at peer p. In particular, for each direct *in-neighbor* r, `inTable` stores the information *which* peers can reach p on paths P, $|P| < k$, where r is the last intermediate hop. The entries in the `inTable` consist of three values: *source*, which is the identifier of the peer at the beginning of the path, *via*, the identifier of the last peer on the path before p itself, and *distance*, the minimum length of all such paths. When p computes an update message for its out-neighbors the distance information is used to determine which `inTable`-entries are irrelevant. Peers also store direct in-neighbors r in the `inTable` as an entry $(r, r, 1)$. Note that if a peer q is in p's `outTable` and its `inTable` contains the entry $(q, r, d-1)$ then there exists at least one cycle $p \to q \to \ldots \to r \to p$ of length d for peers p, q, and r. Thus, the `inTable` and the `outTable` together allow p to decide whether it is in a cycle of length at most k with r as predecessor and q as successor for two given peers r,q.

Naturally, these tables must be built up initially and updated when there is a change in the vicinity. For this purpose, each peer informs the peers in its `outTable` about changes in its tables as follows. If peer p learns about an interesting peer q, it adds q to its `outTable` and sends q an *update list* of all relevant (source,distance)-pairs extracted from its own `inTable` (cf. Table 8.1). Upon receiving an update list from a peer r, peer p increases all distances by 1, since one hop is added to the paths, and updates the corresponding entries with `via` $= r$ in its `inTable`. Afterwards, it recursively computes relevant (source,distance)-pairs and forwards them to the peers in its `outTable`. The tables must also be updated if an edge from a peer r to peer p disappears. In this case, p removes all entries where `via` $= r$ from its `inTable` and sends a *remove list* containing the removed entries to the peers in its `outTable`. If a peer receives such a remove list, it updates its `inTable` accordingly and computes an update message. For the sake of brevity, we omit the details of these computations and the precise contents of the update message. Basically, an update message must announce all changes to the minimum shortest path distance over all in-neighbors. Thus, modifications to an entry in the `inTable` concerning source q where `via` $= r$ must be announced to the peers in the `outTable` only if this modification changed the minimum distance from q via *any* in-neighbor.

UPD:	`SELECT DISTINCT source, distance`
	`FROM inTable`
	`WHERE distance` $< k-1$ `AND via` $\neq q$`;`

CID:	`SELECT source`
	`FROM outTable INNER JOIN inTable`
	`ON outTable.id = inTable.source`
	`WHERE via` $= r$`;`

Table 8.1: *SQL queries for a full update message to q (UPD), and for finding all out-neighbors on cycles with in-neighbor r (CID).*

Whenever peer p_1 adds an interesting peer p_2 to its `outTable`, it checks whether there is a cycle containing edge (p_1, p_2). As described earlier, p_1 can only determine whether there are such cycles, but not how many there are and not which peers they contain in particular (except for cycles of length smaller equal 3). In order to find all cycles with edge (p_1, p_2), peer p_1 sends a *cycle ID (CID) message* containing $h_{p_1}(p_1||p_2)$ to p_2, where h_{p_1} is a hash function private to p_1 and $||$ denotes the concatenation operator. The hash functions h_p are required to produce hash values of publicly known constant length. Peer p_2 in turn determines the set X of out-neighbors that are part of a cycle to p_1 (cf. Table 8.1), and it forwards the received CID with its private hash value $h_{p_2}(p_2||x)$ appended to each peer $x \in X$. The peers receiving a CID message execute the same steps unless the list contains k hash values already, in which case they do not forward the CID any further. Peer p_1 will finally get a CID message $h_{p_1}(p_1||p_2)||h_{p_2}(p_2||p_3)||\ldots||h_{p_\ell}(p_\ell||p_1)$, where $\ell \leq k$, for each cycle. Although peer p_1 cannot decrypt the CID message, it can compute the cycle's length and recognize the head $h_{p_1}(p_1||p_2)$. Furthermore, a unique cycle ID is computed by XORing all contained hash values, $h_{p_1}(p_1||p_2), h_{p_2}(p_2||p_3), \ldots, h_{p_\ell}(p_\ell||p_1)$. The cycle ID is appended to each future data message, indicating that this transfer is a contribution to the trade on this particular cycle. Note that the cycle ID is constructed such that each cycle is identified with its unique ID by every involved peer regardless of the order of the hash values. This prevents potential problems that could arise when a particular cycle is found by more than one search process. This can happen due to inconsistent approximations of the demand graph.

Since it is unknown in advance which and how many cycles will be discovered, a second phase is required to select cycles for trading. For each potential cycle, peer p_1 initiates the negotiation by sending the hashes in the received CID message together with a negotiation bit, set to 1, to its out-neighbor in the cycle. Each peer in the cycle may set the negotiation

bit to 0, indicating that it does not want to trade on this cycle, before it forwards the message. If the bit is still set to 1 when p_1 receives the message, this cycle is accepted for trading, otherwise it is discarded. In order to inform the other peers about the final decision, the result is sent around the cycle. Each peer starts trading as described in Section 8.2 as soon as it learns that the negotiation was successful, i.e., it requests a desired block from the successor in the cycle. Of course, the peers do not start uploading a block at exactly the same time. However, this is not a critical issue as the tit-for-tat trading on cycles proposed in Section 8.2 automatically mitigates temporal fluctuations and also differences of edge bandwidths: a peer with large upload bandwidth waits for the rest of the cycle to catch up as soon as the threshold on the local upload-download balance is reached for this cycle. Moreover, the bandwidth on a cycle adapts to the bandwidth of the slowest edge if the threshold is not too large.

Summing up, we may conclude from our study that barter-based peer-to-peer systems can benefit from inter-swarm trading and trading on cycles in the sense that the majority of peers obtains the desired content faster. Especially for short cycles, the throughput benefits are high and the overhead is low. We find that the best tradeoff is achieved using a `Cycle`(3) trading policy with active sets and probabilistic re-request, yielding a download rate increase of over 50% in the median and 270% on average. Hence, the download durations are shortened by more than one third in the median, and by 73% on average. Also `Cycle`(2) can be very attractive due to its simplicity and low overhead, yielding a download rate increase of 32% in the median and 232% on average.

Our study gives rise to the hope that by boosting the market liquidity of a tit-for-tat based peer-to-peer system by cyclic, inter-swarm trading—possibly in combination with other beneficial enhancements like coding techniques—we have a fourth generation of file sharing systems at hand that outperforms BitTorrent since it harnesses the selfishness of its users more effectively. However, the question remains of how such a novel system can be introduced when the current system is still the de facto standard and predominantly used by most file sharers. This question is the subject of the next chapter.

Chapter 9

How to Establish a Better Equilibrium

Peer-to-Peer systems or social systems often exhibit the property that the more people participate the more attractive or useful is the systems to its users. File sharing is a typical case of such an application: The more people participate, the larger is the selection of content, and usually, the higher are the download rates.

As a simplification, let us model the utility that a user of a peer-to-peer file sharing systems experiences as a strictly monotonously increasing function $U : \mathbb{N} \mapsto \mathbb{R}^+$ of the number of participants. Let U be equal for all peers, let $U(1) = 0$, and let U approach a constant value as the number of participants goes to infinity. Now, let us consider two file sharing systems, $\mathcal{S}_1$ and $\mathcal{S}_2$ with corresponding utility functions U_1 and U_2. Given that n people are willing to join a file sharing system, the situation describes a so-called *anonymous* game where each player can opt either for $\mathcal{S}_1$ or for $\mathcal{S}_2$. For a large enough n, the described game has two Nash equilibria, namely the configuration where all players either participate in $\mathcal{S}_1$ or the configuration where they all participate in $\mathcal{S}_2$. If the players choose their strategy asynchronously, a user deciding for one of the systems at time t will decide for $\mathcal{S}_1$ if $U_1(n_1^t + 1) > U_2(n_2^t + 1)$, where n_i^t is the number of participants of $\mathcal{S}_i$ at time t, and for $\mathcal{S}_2$ if $U_1(n_1^t + 1) < U_2(n_2^t + 1)$. If $U_1(n_1^t + 1) = U_2(n_2^t + 1)$ the user may opt for any of the two systems. Thus, when a selfish user joins one of the systems she increases the attractiveness of that system, and all users afterwards will join the same system. Note that the decision is independent of the utility reached once all players have joined a system. Which of the two utilities $U_1(n)$ and $U_2(n)$ is higher does not influence the outcome, and the players may end up in a strategy profile that

gives them much lower payoffs, essentially because of the lack of coordination.

Steering players to the Nash equilibrium that yields the highest social welfare is a natural objective of mechanism design. We have seen in Part I of this thesis that by offering payments, any desired Nash equilibrium can be implemented in one-shot, simultaneous-move games. However, the same mechanism does not work practically if the players do not move simultaneously. Moreover, it is infeasible to offer payments to a peer willing to join a file sharing system, mainly because future users are unknown until they enter a system. Also, encouraging all participants of a file sharing system to move to the other system would necessitate the mechanism designer to offer payments to all users at the same time. Another problem is that the users must consider the mechanism designer credible, i.e., they need to be persuaded that the mechanism designer will make the payments he offers, which poses another problem if n is large (several million in the case of BitTorrent).

Mechanism design with payments is clearly an infeasible approach for tackling the problem of guiding participants of a file sharing system to joining a (better) system instead. In the following, we present an approach that we took with our peer-to-peer file sharing client, BitThief: While BitThief continues to support downloads from BitTorrent clients, it shares files with other BitThief clients using the novel protocol that implements tit-for-tat trading with source coding and cyclic inter-swarm trading.[1] As the BitThief clients free-ride with respect to the BitTorrent clients, using BitThief instead of other clients is a dominant strategy. Moreover, the more users of normal BitTorrent clients switch to BitThief, the more the performance of BitTorrent degrades. This is not only because the number of BitTorrent users decreases, but also because the share of free-riders in BitTorrent swarms increases. At the same time, the number of clients supporting the tit-for-tat protocol increases. Given that all file sharers are selfish and aware of the option to choose BitThief, all BitTorrent users will switch to BitThief eventually.[2] Thus, we could achieve a smooth transition to the equilibrium where all participants use the enhanced tit-for-tat protocol.

Unfortunately, the above discussion abstracts away from reality in that it does not take into account that other clients might not like BitThief enticing all users away from them. From other clients' perspective, the BitThief clients only degrade their performance. Thus, we have to assume that they try to ban a BitThief client if they can identify it. Consequently, in order to ensure the capability of downloading data from other BitTorrent clients, instances of BitThief must not be identified as such by other clients, however, in order

[1] In the current version of BitThief, cyclic inter-swarm trading is not included yet.

[2] Obviously, we do not necessarily expect this course of action to be the future of peer-to-peer file sharing. Apart from mere performance, there are many other features to a BitTorrent client that may attract users like usability, graphical design, or image. Moreover, BitThief is often considered "evil" by file sharing enthusiasts.

to share data among each other they must identify each other. Note that the BitThief clients enter standard BitTorrent swarms and share the same content among each other, only by means of the tit-for-tat protocol.

The problem that the BitThief instances have to solve is reminiscent of the archetypal espionage scene where an agent meets a contact person for the first time. In order to make sure that the agent got the right person, and not some innocent bystander, or an enemy spy, they exchange previously agreed-upon pass phrases.[3] For reasons of this association, we will refer to the BitThief instances as "spies" or *conspirers* in the following, and to other BitTorrent client instances as *regular* peers. We thus strive to solve the problem we denote as *REVEALTYPE*: How can a conspirer safely determine a connected peer's type, i.e., learning whether the connected peer is a conspirer or a regular peer without giving away its conspiring identity in the latter case.

Our approach is that the conspirers implement a *steganographic handshake*[4]: when a conspirer p connects to a peer it should encrypt information that reveals its type into the imposed protocol in such a way that the information is decodable by a fellow conspirer, but not by a regular peer. Furthermore, regular peers must not even be able to realize that p is initializing a handshake. For the purpose of the steganographic handshake, we assume in the following that conspirers know an exclusive secret K of length |K| that is unknown to regular peers. Furthermore, let the conspirers know the total number n of peers in the swarm. Our techniques can be easily adapted to the case where the conspirers only have an approximate value of n available.

9.1 Steganographic Handshake

The extent to which conspirers are able to communicate secretly among each other depends on the freedom that the imposed p2p protocol offers. If the peers are given more leeway in their actions, more information can be hidden. For instance, if the protocol does not specify the order in which a peer requests blocks from a neighboring peer the conspirers can introduce a logic to the order of request sequences and thereby communicate without violating the given protocol. We call such exploitable, variable parameters of the p2p protocol *steganographic channels*. In the following, we will primarily make use of the *request order channel* that we have just described.

[3] Indeed, such spy rendezvous protocols are not an invention of literature or film. For example, it is said that atomic spy Klaus Fuchs hooked up with his contact person in New York by carrying a tennis ball, using the pass phrase "Can you tell me the way to Grand Central Station?"

[4] Steganography is the art of hiding information in a message, such that only intended recipients are able to decipher the hidden information, and all other viewers of the message do not even suspect the existence of hidden information.

Note that also regular peers send bits on this channel simply by using the file sharing protocol, however without an intended logic. Thus, based on the common secret K, a conspirer can send a specific bit string on the steganographic channel that can only be verified by fellow conspirers. One danger remains, though, namely that a regular peer accidentally acts like a conspirer and sends the specific bit string on the steganographic channel that makes a connected conspirer think it is a conspirer. Assuming for the moment that we have a mechanism to transmit bits over request order channel, the question to be solved is thus: How many bits have to be exchanged in order to ensure that the communication partner is indeed a conspirer?

The peer with the smaller id, say u, may send, e.g., the first half of the secret key K to v. If v is a conspirer, it knows that a conspirer tries to send the secret over the request order channel, and checks whether the bits produced by the received request sequence correspond to the first half of K. If this check is positive v knows that $u \in C$ and sends the second half of K back to u in plain text. Thus, u is ensured v is a conspirer, too. On the other hand, if v is a regular peer it does not notice any irregularity while communicating with u since any request order is allowed by the p2p protocol. Moreover, even if v was aware of the request order channel and could decode the sent bits correctly, it would not be able to detect the irregularity since it does not know K. Peer v cannot distinguish u from a regular peer unless it knows K.

The problem of false positives could arise if a regular peer inadvertently sends the right $|K/2|$ bits over the steganographic channel, and a receiving conspirer would send a plain text message back to u, which is an illegal action. However, by choosing a key that is large enough and uniformly at random, the conspirers can keep the probability of this false positive fairly low: The probability of an individual false positive is $2^{-|K/2|}$. Hence, if the conspirers choose a key K of length at least $6 \log n$,[5] then there are no false positive over any of the at most $\binom{n}{2}$ communication links *with high probability* (w.h.p.).[6]

The fact that a regular peer can have several neighboring conspirers poses a certain threat: A regular peer could perfom a *replay attack*, i.e., it could request blocks in the same order as other peers requested them, and thus provoke a false positive with significant probability. The conspirers can avoid such an attack by using keys that are connection-specific. In particular, a conspirer u could use $\mathcal{H}(id_u \| id_v \| K, \log n)$ as a key when communicating with v, where '$\|$' is the concatenation operator and $\mathcal{H}(x, b)$ is a hash function

[5] The base of the logarithm is always 2 unless we write ln, in which case the base is e.

[6] If an event occurs with probability at least $1 - \mathcal{O}(1/n)$, we say that it occurs "with high probability". A stronger definition demands that the probability is at least $1 - 1/n^{\Omega(1)}$, which can frequently by achieved by linearly increasing the constants involved (the constant in the exponent depends on the linear increase). For the sake of simplicity, we use the weaker definition in this thesis.

Algorithm 9.1 ENC_{order}

Input: block sequence B, message $M \leq |B|!$
Output: permuted block sequence Π

```
1: Sequence Π := ∅;
2: for i := |B| − 1 to 0 do
3:    l := M div i!;
4:    Π.append(B_l)) ;
5:    B.remove(l) ;
6:    M := M − l · i! ;
7: end for
8: return Π;
```

mapping a bit string $x \in \{0,1\}^*$ to a bitstring of length b with the property that if the input value x has an entropy of at least b then the hash value $\mathcal{H}(x,b)$ can be guessed successfully with probability at most 2^{-b}. Given that K is chosen uniformly at random, the input chosen by a conspirer has an entropy of $|K|$ bits. Hence, if $|K| \geq \log n$ then $\mathcal{H}(id_u \| id_v \| K, \log n)$ can be guessed by a regular peer with probability at most $1/n$. Again, in order to ensure that we get this probability over all comunication channels, the length of K should be increased to $6 \log n$ bits. Note that both the key as well as the resulting hash value must have this length.

9.1.1 Using the Block Request Order Channel

We will now discuss how the block request order can be used to exchange information. Once a conspirer u has determined which blocks it wants to request from a connected peer v it can permute the order of this request sequence as shown in Algorithm 9.1 to transmit $\log(|B|!)$ bits, where input B is the sequence of blocks to request ordered according to its canonical order. For example, the blocks shared in BitTorrent have an index that represents their position in the shared file. Algorithm 9.1 interprets message M as a number, converts it to its representation in the factorial number system, denoted by $M_!$, and computes a permutation Π by interpreting $M_!$ as the Lehmer code of Π. In the factorial number system representation, $x_!$, of a number $x \in \mathbb{N}$, the i-th digit has a place value of $i!$. As an example, the decimal number 17, which represents the binary message 10001_2, has a factorial number representation of $2210_!$, because $0 \cdot 0! + 1 \cdot 1! + 2 \cdot 2! + 2 \cdot 3! = 17$. The Lehmer code of a permutation counts the number of swaps of neighboring elements that have to be executed in the originally ordered list for each element in the target permutation to be moved to its right position, starting from the first. The permutation $(3,4,2,1)$, e.g., has a Lehmer code of 2210 as, starting from

Algorithm 9.2 DEC_{order}

Input: permutation Π
Output: message $M \in \{0,1\}^{log(|\Pi|!)}$

```
1: Sequence S := (1, 2, ..., |Π|);
2: M := 0 ;
3: for i := 0 to |Π| − 1 do
4:    l := S.indexOf(Π_i) ;
5:    S.remove(l) ;
6:    M := M + l · (|Π| − i − 1)! ;
7: end for
8: return M;
```

the original sequence $(1, 2, 3, 4)$, element 3 needs two swaps to get to the left most position, then element 4 needs two swaps to get to the second position, element 2 can be moved to the third position with one swap, and element 1 is already at position four. To get a permutation from a given Lehmer code one simply reverses this procedure. For the Algorithms 9.1 and 9.2, we assume the data structure used to represent block sequences offers the methods *append*, *remove* and *indexOf*. S.append(x) appends element x to sequence S, S.remove(x) removes element x from sequence S, and S.indexOf(x) returns the index of the first occurence of element x in sequence S. While Algorithm 9.1 permutes a block sequence in order to get an encoding of message M, Algorithm 9.2 decodes a message from a given permutation by inverting the encoding technique used in Algorithm 9.1.

Theorem 9.1. *The algorithm pair* $(ENC_{order}, DEC_{order})$ *transmits an optimal* $\lfloor \log s! \rfloor$ *bits over the request order channel, where* s *is the number of blocks requestable by the transmitting peer.*

Proof. The transmission is correct since ENC_{order} implements a bijective function from the message domain $\{0,1\}^{\log(|B|!)}$ to the domain of permutations of B. DEC_{order} implements the inverse bijection. The pair $(ENC_{order}, DEC_{order})$ is optimal because there are $|B|!$ many permutations of length $|B|$. Any algorithm pair can encode at most $\log(|B|!)$ bits in the order of the block sequence. □

9.1.2 Additional Steganographic Channels

Apart from the request order channel there are additional steganographic channels that could be exploited for hidden communication.

Subset Selection

Instead of only using the order of requests to hide information, additional information can be hidden in the concrete choice of blocks that are selected for requesting. Let $B_{u,v}$ be the set of blocks that conspirer u can request from v; instead of requesting, e.g., a permutation of the s requestable blocks with lowest index, u can transmit an additional unused $\log \binom{|B_{u,v}|}{s}$ bits to v by selecting the subset of s blocks to be requested according to a shared secret.

Timing

The protocol allows to introduce some variation in the timing of the protocol messages, as peers are not expected to request blocks or to answer requests immediately. Hence, conspirers can hide information by delaying protocol messages. One possibility is, e.g., to encode information in the time between the reception of a block request and the corresponding transmission. Note that such steganographic channels are only feasible if the connection between the peers is stable, i.e. the message delays are within a reasonable range. In realistic networks, conspirers would most likely have to use error-correcting codes. Generally, the capacity of such time-coded channels depends on the accuracy of the measurements, the predictability of delays, and on the extent to which a conspirer may delay the protocol without evoking suspicion.

Bandwidth

A conspirer might vary the rate at which it sends file blocks, or network packets in general. One possible protocol would be to encode bits in the transitions from one rate to another. With each transmission, a peer either goes from a low to a high bitrate to send a 1, or from a high to a low bitrate to send a 0. Note again that in practice one would probably need error-correcting codes to account for unstable connections.

Ports

Another channel is the choice of the communication port. Unfortunately, this channel is not scalable since the port for the communication is only chosen once per connection, and its capacity is rather small. More importantly, many peers are typically not able to use this channel since they are behind a NAT router that allows no explicit control of the ports.

9.2 Implementation into BitThief

The steganographic handshake described in the previous section is the tool we need to ensure that instances of our BitThief client are not identified and thus risk being banned by other clients, while revealing their identity to fellow BitThief instances. BitThief solves REVEALTYPE by impersonating standard BitTorrent clients and performing a steganographic handshake.

Unfortunately, we could not solve REVEALTYPE exactly as discussed in Section 9.1, since our practical solution has to satisfy the additional constraint that BitThief never uploads a file block to a client that is not another BitThief instance. Note that a free riding client must not announce having file blocks that a connected peer v is interested in as v may request such a block. Upon receiving a valid request from v, a free rider would have to upload data, and violate the no-upload constraint, or ignore the request and risk revealing its identity. Moreover, many clients wait until they receive a requested block before they send out more blocks, i.e., a free rider would not receive any blocks from this peer anymore for free until the request is served. Consequently, when two potential BitThief clients meet, they both announce that they have no file blocks yet, and none of them can make a block request.

Our solution to REVEALTYPE in this setting is to abuse an extension of the BitTorrent protocol called "peer exchange" (PEX). This extension allows peers to learn about other peers from known peers without inquiring a tracker. Basically, a PEX message contains a list of some of the peers to which the sender is connected, and optionally also a list of dropped peers, i.e., peers that cannot be reached (anymore). We use the order of this peer list as a steganographic channel. In particular, if an instance u of BitThief wants to determine whether another peer v is also an instance of our client, it sends a PEX message where the peer list is permuted with respect to its natural order according to a hash $\mathcal{H}(id_u \| id_v \| F \| K)$ of u and v's identifiers[7], hashed meta data F of the file that is exchanged, and a secret K that is shared by all BitThief clients. The permutation is constructed along the lines of Algorithm 9.1. Upon receiving a PEX message from v, u checks whether the peer list is ordered according to $\mathcal{H}(id_v \| id_u \| F \| K)$. If the check is successful v sends back a PEX message to u with a list ordered according to $\mathcal{H}(id_u \| id_v \| F \| K)$, unless it has already sent a PEX message with a hidden key to v before. After successfully checking a received PEX message for the hidden key, BitThief switches to the tit-for-tat protocol, and starts trading with the connected BitThief instance. As with the request order in Section 9.1, the PEX protocol does not impose a specific peer list order, and other clients do not interpret it in any way. Hence, a BitThief client does not evoke suspicion when sending such a modified PEX message to other clients.

[7] In BitTorrent 20-bit peer identifiers are used.

Note that there may be a problem if the swarm is small and consequently the length of the list in a PEX message is short. In this case, the probability that another client sends a list ordered correctly according to the hashing algorithm "by accident" is high. Therefore, our client resorts to the following fallback strategy when it knows less than 17 peers: it creates a bogus entry in the list (of active neighbors) by again hashing F, K, the recipient's and its own identifier, and then constructing an IP address and a port out of this hash. The recipient can compute the same fake entry and easily check whether this entry is in the received list. Since there are roughly 2^{48} possible fake addresses consisting of a (32-bit) IP address and a (16-bit) port,[8] the probability of false positives is sufficiently small. Note that we chose 17 as the threshold for the fallback strategy because $\log(17!) = 48.34 > 48$, and thus permuting a list of at least 17 entries can hide more information than one fake IP address and port.

Suppose that in future versions of the PEX protocol the order of entries will be predetermined: while permutations could no longer be used to establish a hidden handshake, one could still add bogus clients to the list to solve this problem. It is unlikely that a regular peer identifies a faked entry in the PEX message, even if it realizes that the corresponding peer is not available, as it is plausible that some peer may be connected to a peer u, but another peer cannot contact u, e.g., because it is behind a NAT router and port-forwarding is not enabled, or because this peer left the swarm in the meantime. Thus, if a peer does not get a response from some peers in the list, it cannot conclude that the sender of the list sent invalid information.

Extensive tests in live BitTorrent swarms have shown that the hybrid steganographic handshake works well, i.e., the BitThief clients successfully find each other and switch to their private tit-for-tat protocol without being detected. The steganographic handshake is implemented as described in the current version of BitThief.[9]

In this chapter, we have argued that a smooth transition is essential for guiding the participants of an established system to switching to another system. Moreover, we showed what practical problem the requirement of a smooth transition can entail, and how to solve it in the realm of peer-to-peer file sharing. Next, we will see how the steganographic handshake can be used as a primitive to coordinate the action of the conspirers in a monitored environment.

[8] Some ranges of IP addresses and some ports cannot be used, which slightly reduces the address space.

[9] BitThief is available at bitthief.ethz.ch.

Chapter 10

Hidden Broadcast in Peer-to-Peer Systems

In this chapter, we look at a more general communication primitive that can be used by a conspiring subset of peers to find each other, exchange secret messages, and thus to coordinate their actions without revealing the secret communication to an authority that may monitor the communication links.

Such a protocol can be useful in different contexts. In p2p networks, e.g., conspiring peers may prioritize each other in terms of quality of service. In overlay networks, conspiring peers may try to position themselves at strategically favorable spots to manipulate the overlay. Conspiring peers may also be machines controlled by law enforcement in order to bring down a malicious botnet. From a game-theoretic perspective, hidden communication among a subset of the participants of a network can constitute the prerequisite for collusion, where a subset of the players wants to coordinate their strategies to affect the game's outcome in a favorable way.

As can be seen from these examples, conspiring peers may be considered useful or harmful, depending on the point of view. In the remainder, for the sake of a lucid presentation, we describe the situation from the point of view of the conspiring peers. In particular, we show how a conspiring peer can implement a *hidden broadcast*, i.e., send a secret message to all other conspirers, either directly or indirectly, without getting caught by the regular peers or the monitoring authority. For clarity, we proceed with presenting the used model in more detail.

10.1 Model & Problem Definition

We are given a p2p network, which consists of a set P of $|P| = n$ peers. Each peer $u \in P$ can communicate directly with any other peer v if u has previously learnt the address of v. It is assumed that a peer does not know any other peers initially, i.e., upon joining the network a peer is not connected to any other peers. In order to establish connections to other peers, we assume that there is a publicly known server that is aware of all peers currently in the network. Upon request this server delivers a list of k peer addresses, which are chosen uniformly at random from all peers. If two peers are connected, we say that they are *neighbors*. Another common technique in popular p2p networks is to inquire known peers about other peers in the network. Since this approach still depends on some kind of bootstrapping mechanism, we assume for the sake of simplicity that peer addresses are only provided by the dedicated server. Furthermore, all communication is assumed to be reliable, i.e., message loss and corruption can be handled in a canonical way using redundancy and/or checksums. However, message delivery times are subject to a potentially variable delay.

In general, a p2p network may be used to exchange any number of files. We assume that there is only one file f that is shared in the network, i.e., our definition of a network is akin to the concept of a BitTorrent *swarm* in which all peers share a single file or a specific collection of files. Efficient dissemination of the file f is achieved by splitting it into m data blocks $b_1, b_2, \ldots, b_m$, and trading locally available blocks for missing blocks with other peers. Both the number of blocks m and the number of peers n are assumed to be fairly large and in the same order of magnitude so that $m \gg \log n$ and $n \gg \log m$. We assume that the file blocks are fairly well distributed among all peers and at least one peer holds all m blocks at any time. Thus, it is always possible to acquire the entire file f by connecting to a reasonable number of peers. Furthermore, we make the assumption that any two connected peers regularly exchange information about the locally available blocks, i.e., a peer reports regularly on the blocks it has to each neighbor. Over each link, a peer can send one request for a block b_i at a time to a certain peer, wait until b_i has been transmitted completely, and then send the next request to the same peer.[1] Unless a block is already being transmitted, a request is always served and the requesting peer receives the block at the latest after a certain bounded delay.

We distinguish between two classes of peers: A peer u is either a *regular peer* or a *conspirer*. A regular peer is a peer whose sole purpose is to acquire and share file f with other peers. The conspirers, on the other hand, have a secret agenda. In particular, the conspirers strive to secretly communicate

[1] Note that it is possible to send different requests to different peers simultaneously.

among each other. The set of conspirers is denoted by C, and its cardinality is $|C| = c$. Another distinction between regular peers and conspirers is that the conspirers share an exclusive secret K of length $|K|$, which they acquired over a secure channel before joining the p2p network. What is more, the conspirers know both the number c of conspirers and, for ease of presentation, the number n of peers in the network.[2]

In order to ensure that the p2p network is used only for the intended purpose of sharing file f and to prevent fraudulent behavior by conspiring peers, there is an *authority* that monitors the network. If the authority ever learns that a peer u does not abide by the rules of the imposed protocol and exchanges other information with certain peers, then u is punished, e.g., by adding u to a globally available (signed) blacklist or, if the authority has the power, by expelling u directly from the network. The authority can detect conspirers in two ways, either it observes suspicious communication directly, or regular peers denunciate them, i.e., we assume that there is an incentive for regular peers to report any observed departure from the file sharing protocol. Of course, it is also possible for conspirers to report regular peers but we assume that it is not worthwhile for conspirers to do so for the following reasons. First, the authority may suspect both the defendant and the accuser, which may be detrimental to the conspirer as well. Second, assuming that n is considerably larger than c, other regular peers may vouch for the accused (regular) peer and thereby revealing the disingenuous nature of the conspirers. In other words, c may not be large enough for the conspirers to campaign against a regular peer. Finally, the authority may have recorded the communication. In this case, it can determine that the regular peer always adhered to the protocol contrary to the accusation.

As mentioned before, the primary goal of the conspirers is to communicate with each other, while the authority tries to detect as many conspirers as possible. The main problem that we consider is called *BROADCAST(M)*, which is defined as follows.

BROADCAST(M): There is a conspirer $u \in C$ that wants to send a message M, directly or indirectly, to all other conspirers without raising suspicion among the regular peers and without being caught by the authority.

The quality of a solution to this problem can be measured in several ways. An important measure is certainly the probability of success. The second optimization criterion is efficiency: The objective is to achieve a low *communication complexity*, i.e., the number of messages that must be exchanged ought to be small. Moreover, the *space complexity*, the number of

[2] As (BitTorrent) trackers usually provide only an estimate of n, we will argue in a later section that our techniques can easily be adapted to the scenario where the conspirers merely have an estimate of n and c.

bits that each conspirer has to store, should be low as well.[3] In order to solve *BROADCAST(M)*, we may make use of the steganographic handshake described in the previous chapter, which lets a conspirer reveal the type of a connected peer.

In our model, the order in which a peer requests blocks from a neighboring peer is not specified. Hence, the conspirers can use the request order channel once more, i.e., they can introduce a logic to the order of request sequences and thereby communicate without violating the given protocol.

Obviously, a conspirer may also communicate freely with a neighboring conspirer if the network connection between them is not monitored; however, as the conspirers initially do not know the other conspirers' identities, they first have to determine their neighbors' types by means of a *steganographic handshake* (Section 9.1). The conspirers' capabilities to communicate depend on the freedom they have in varying the imposed protocol. This freedom, in turn, is derived directly from the power of the authority monitoring the p2p network. In the subsequent sections, the goal is to determine how (much) information can be secretly broadcast given a certain authority. The discussion is structured according to increasing monitoring capabilities.

10.2 No Monitoring

A weakest authority is one that does not have the capacity to monitor connections at all. The only way such a limited authority can learn about illegal actions in the network is through reports of regular peers. As stated in the model section, we assume that if a conspirer u reveals its type to a regular peer v, then v will report u to the authority, and u will be punished. Thus, a conspirer u must not communicate using plain text with a neighboring peer v unless it has verified that v is also a conspirer. If the verification is successful, u may send all subsequent messages to v in the clear. Hence, in order to solve *BROADCAST(M)* it suffices to establish a connected graph among the conspirers where there is an edge between two conspirers $u, v \in C$ if they are neighbors and both of them know each other's type, i.e., both know that $u, v \in C$. Once connected, the message holder can send the message M in plain text to all of its neighboring conspirers, each of which will propagate it to its respective neighboring conspirers.

One straightforward approach to achieve this is to have the message holder $u \in C$ connect to every peer in P, determining every peer's type and then send the message M to the fellow conspirers, which are all directly connected to u. Although this simple approach solves $BROADCAST(M)$, the conspirers are well advised not to use it because of its lack of efficiency, both

[3] Apart from the downloaded blocks, a peer must also store, e.g., the addresses of its neighbors.

in terms of space and communication complexity. Since the message holder basically connects to the entire network it needs $\Omega(n)$ memory. Furthermore, the straightforward approach requires extensive polling of the public server that keeps track of all peers, and would cost $\Omega(n/k)$ messages as only k addresses are returned for each request. As k is typically a constant, this amounts to a communication complexity linear in n. Another major drawback of this scheme is that the server receives an exceedingly large number of requests, which basically gives away the identity of the message holder.

In contrast to this brute-force approach, we present a scheme that solves *BROADCAST(M)* much more efficiently in the following. Depending on the number of conspirers c, considerably less than n connections have to be established to ensure that all c conspirers induce a connected subnetwork:

Theorem 10.1. *If each conspirer randomly acquires $8\frac{n}{c}\ln(nc)$ neighbors, then the subnetwork induced by the c conspirers is connected w.h.p.*

Proof. First, we show that each conspirer u has a sufficiently large number of conspiring neighbors. Let $\mathcal{N}_u^c$ denote the set of neighbors of u that are conspirers. Since each neighbor v is chosen uniformly at random and the probability that $v \in C$ is c/n, we immediately have that $\mathbb{E}[|\mathcal{N}_u^c|] = 8\ln(nc)$. Using a standard Chernoff bound, we get that

$$\mathbb{P}[|\mathcal{N}_u^c| < 4\ln(nc)] = \mathbb{P}\left[|\mathcal{N}_u^c| < \frac{\mathbb{E}[|\mathcal{N}_u^c|]}{2}\right] \leq e^{-\frac{\mathbb{E}[|\mathcal{N}_u^c|]}{2^2 \cdot 2}} = \frac{1}{nc}.$$

Hence, the probability that *any* conspirer has less than $4\ln(nc)$ neighbors that are conspirers is upper bounded by $1/n$.

We now need to prove that this number of connections suffices to guarantee that all conspirers are connected with high probability. For this purpose, we use the following theorem about Erdős-Rényi random graphs $G(c, p_e)$ [38, 39]. $G(c, p_e)$ is a graph consisting of c nodes in which each of the $\binom{c}{2}$ edges is added to the graph independently with probability p_e.

Theorem 10.2 ([96])**.** *If $p_e = \frac{\ln c + t}{c}$, then $G(c, p_e)$ is connected with probability $(1 + o(1))e^{-e^{-t}}$.*

This theorem implies that if each edge is chosen with probability $\ln(nc)/c$, then the resulting graph is connected with probability at least

$$e^{-e^{-\ln n}} = e^{-1/n} \geq 1 - \frac{1}{n}.$$

If $p_e = \ln(nc)/c$, the expected number of neighbors of each node is $\ln(nc)$. Let L_u be the random variable that counts the number of u's neighbors in a

graph $G(c, p_e)$. Again using a Chernoff bound, it follows that

$$\mathbb{P}[L_u > 4\ln(nc)] = \mathbb{P}[L_u > (1+3)\mathbb{E}[L_u]] \leq e^{-\frac{3^2}{4}\mathbb{E}[L_v]} < \frac{1}{nc}.$$

The probability that *any* node has more than $4\ln(nc)$ neighbors in a graph of c nodes, where each edge is chosen with probability $\ln(nc)/c$, is upper bounded by $1/n$. This means that we also get a connected graph, with high probability, if each node chooses $4\ln(nc)$ neighbors uniformly at random in a graph of size c. We already established that by connecting to $8\frac{n}{c}\ln(nc)$ random neighbors, each conspirer connects to at least $4\ln(nc)$ conspirers with high probability, i.e., each conspirer implicitly chooses at least $4\ln(nc)$ random neighbors in the conspirer subgraph. Therefore, the subnetwork is connected with high probability. □

Theorem 10.1 states that if $c \in \Theta(n)$, acquiring a logarithmic number of neighbors is sufficient for the conspirers to end up in a connected component. Note that the constant 8 can probably be reduced using more elaborate arguments. However, it is clear that the asymptotic behavior is correct for any $c \in \Theta(n)$ as the graph $G(n, p_e)$ is *not* connected if $p_e = ((1-\epsilon)\ln n)/n$ for any $\epsilon > 0$ *asymptotically almost surely* [39].[4]

Putting all the building blocks together, we are able to give an algorithm that solves *BROADCAST(M)*. Algorithm 10.1 is executed at each conspirer $u \in C$: each conspirer first polls the public server until it has enough neighbors to ensure that all conspirers are in a connected component. In a second phase, each conspirer u gathers enough blocks in order to make sure that any two conspirers have enough trading blocks to determine each other's type. If another peer connects to u in this phase, u reports that it does not have any blocks yet in order to avoid trading blocks with other conspirers prematurely. Subsequently, each conspirer starts requesting blocks from all its neighbors and thereby reveals its type by transmitting a secret key over the request order channel. Since the number of required blocks is relatively small, i.e., we assume that $6\log n \ll m$, each peer can accumulate this number of blocks quickly. Once the message holder knows all its neighbors' types, it sends message M in plain text to its neighboring conspirers. All other conspirers wait for M and pass it to their neighboring conspirers as soon as they receive it.

For the subroutine REVEALTYPE, which implements the steganographic handshake, we assume that another thread run by u listens to neighbor v, and whenever it receives a block request b_x from v, appends b_x to a list R_v, and starts transmitting block b_x to v. Moreover, we denote by $B_{u,v}$ the set

[4] "Asymptotically almost surely" means that the probability that the claimed bound holds tends to 1 as $n \to \infty$.

Algorithm 10.1 BROADCAST(M)

repeat
  Add k random peers to neighbor set $\mathcal{N}$;
until $|\mathcal{N}| \geq 8\frac{n}{c}\ln(nc)$
Get $6\log n$ random blocks in total from connected peers;
$C := \emptyset$;
for each $v \in \mathcal{N}$ **in parallel do**
  if REVEALTYPE(v)= *conspirer* **then**
    $C := C \cup \{v\}$;
  end if
end for
if message holder **then**
  send M to all $v \in C$;
else
  wait until message M received;
  send M to all $v \in C$;
end if

Subroutine REVEALTYPE(v)
wait until $(|B_{u,v}| \geq 3\log n) \wedge (|B_{v,u}| \geq 3\log n)$;
$B :=$ $\lceil 3\log n \rceil$ blocks of $B_{u,v}$ with lowest indices;
$B' :=$ $\lceil 3\log n \rceil$ blocks of $B_{v,u}$ with lowest indices;
Sort B, B';
$\Pi := ENC_{order}\left(B, \mathcal{H}(id_u||id_v||K, \lceil 3\log n \rceil)\right)$;
$\Pi' := ENC_{order}\left(B', \mathcal{H}(id_v||id_u||K, \lceil 3\log n \rceil)\right)$;
for $i := 0$ to $|\Pi| - 1$ **do**
  for $j := 0$ to $|R_v| - 1$ **do**
    if $R_{v,j} \neq \Pi'_j$ **then return** *regular*;
  end for
  if $(|R_v| < i) \vee (|R_v| > i + 1)$ **then**
    return *regular*;
  else
    request Π_i;
    wait until Π_i received;
  end if
end for
return *conspirer*;

of blocks that u can request from v, i.e., the set of blocks that v claims to possess and u does not. The following theorem states the communication and the space complexity of this broadcast algorithm.

Theorem 10.3. *If $c \in [18 \ln n, n/3]$ and $m \geq (24e+6) \log n$, Algorithm 10.1 secretly broadcasts a message M of arbitrary length in an unmonitored network w.h.p. The space complexity is in the order of*

$$\mathcal{O}(\frac{n}{c} \log n + |M|)$$

and the communication complexity is in

$$\mathcal{O}\left(\frac{n}{c} \log n + \log^2 n + |M| \log n\right) \text{ w.h.p.}$$

Proof. We start by showing that each conspirer has to collect only

$$12(1 + o(1))\frac{n}{c} \ln(nc) \in \mathcal{O}(\frac{n}{c} \log n)$$

peer addresses in order to get $8\frac{n}{c} \ln(nc)$ *distinct* random neighbors, i.e., it has to inquire the server only $\mathcal{O}(\frac{n}{c \cdot k} \log n)$ times. Let the random variable N_i indicate the number of random peers that have to be collected to get the i^{th} distinct neighbor after having collected $i-1$ distinct neighbors. It holds that

$$\mathbb{E}[N_i] = \frac{n}{n-i+1}.$$

Let $T_j = \sum_{i=1}^{j} N_i$ be the random variable that indicates how many random neighbors have to be collected until j distinct peers have been discovered.

Since $c \geq 18 \ln n > 9 \ln(nc)$, and given that we need $j = 8\frac{n}{c} \ln(nc)$ distinct neighbors, it holds that $n - j > n/9 \in \Omega(n)$. Hence, we have that

$$\begin{aligned}
\mathbb{E}[T_j] &= \sum_{i=1}^{j} \mathbb{E}[N_i] \\
&= n\left(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{n-j+1}\right) \\
&= n(H_n - H_{n-j}) \\
&\leq n(\ln(n) - \ln(n-j)) \\
&= -n \ln\left(1 - \frac{j}{n}\right) \\
&= -\ln\left(1 - \frac{j}{n}\right)^n \\
&\leq -\ln\left(e^{-j}\right) + o(1) \\
&= j + o(1),
\end{aligned}$$

where H_n is the n^{th} harmonic number. We can again use a Chernoff bound to see that

$$\begin{aligned}\mathbb{P}[T_j > (1+1/2)\mathbb{E}[T_j]] &< e^{-\frac{1}{16}\mathbb{E}[T_j]} \\ &= e^{-\frac{1}{2}\left(8\frac{n}{c}\ln(nc)+o(1)\right)} \\ &\leq e^{-3/2\ln(nc)} \\ &< \frac{1}{nc},\end{aligned}$$

where the second last inequality holds since c is at most $n/3$. Thus, each conspirer has to acquire less than $12\cdot(1+o(1))\frac{n}{c}\ln(nc)$ random peer addresses with high probability. Since each node further has to store the message M, the bound on the space complexity follows.

Next, we argue that a conspirer can acquire $6\log n$ blocks quickly (cf. Line 4). Recall that $\mathcal{N}_u^c$ denotes the neighboring conspirers of u, and that $\mathbb{E}[|\mathcal{N}_u^c|] = 8\ln(nc)$. By means of a Chernoff bound we can see that

$$\mathbb{P}\left[|\mathcal{N}_u^c| > 2\mathbb{E}[|\mathcal{N}_u^c|] \leq e^{-\frac{\mathbb{E}[|\mathcal{N}_u^c|]}{4}} = \frac{1}{(nc)^2},\right.$$

i.e, any conspirer has less than $16\ln(nc)$ conspiring neighbors and hence more than $8\frac{n}{c}\ln(nc) - 16\ln(nc) > 6\log n$ regular neighbors with high probability. This implies that less than one block has to be acquired on average per regular neighbor, which can be accomplished swiftly.

We proceed by proving that $6\log n$ blocks suffice to ensure that each conspirer has $3\log n$ blocks to trade with each neighboring conspirer with high probability, which is the number of requestable blocks required for the subroutine REVEALTYPE to work (cf. Line 14–16). Note that we can transmit $\log((3\log n)!) \geq 3\log n$ bits over the request order channel by using $3\log n$ blocks. As argued in the last chapter, exchanging $3\log n$ bits with a conspirer is enough to safely verify its type. Thus, we have to show that the probability that two conspirers have more than $3\log n$ blocks in common is negligible. The probability that the i^{th} block is also in the set of blocks that another conspirer acquires is at most $(6\log n)/(m-i) \leq (6\log n)/(m-6\log n)$ because at most all $i-1$ previous blocks are not in the other conspirer's set. Let the random variable X denote the number of such colliding blocks and let $p := (6\log n)/(m-6\log n)$. We can upper bound the probability that $X \geq 3\log n$ as follows. Let $\mathcal{X}_{u,v}$ be the set of colliding blocks, i.e., the blocks

acquired by both u and v.

$$\begin{aligned}
\mathbb{P}[X \geq 3\log n] &= \sum_{i=3\log n}^{6\log n} \mathbb{P}[|\mathcal{X}_{u,v}| = i] \\
&\leq \sum_{i=3\log n}^{6\log n} \binom{6\log n}{i} p^i (1-p)^{6\log n - i} \\
&\leq \binom{6\log n}{3\log n} p^{3\log n} \\
&\leq (2e)^{3\log n} p^{3\log n} \\
&= \left(\frac{12e\log n}{m - 6\log n}\right)^{3\log n}.
\end{aligned}$$

Since $m \geq (24e+6)\log n$, we have that $\mathbb{P}[X \geq 3\log n] \leq 1/n^3$ and thus every conspirer has enough blocks to request from all other conspirers with high probability. This proves that each conspirer can identify the conspirers among its neighbors.

The question remains how many messages a conspirer u has to send in the REVEALTYPE phase. As u requests one block and then waits for the next request of the other party, u can identify regular peers quickly: u identifies a regular peer v as soon as v requests the "wrong" block. Since the probability that v requests the correct first block is $1/(6\log n)$ and the probability that each subsequent block is also "guessed" correctly is smaller than $1/2$ (until only two blocks are left), the expected number of blocks that need to be exchanged is less than 2. In total, the number of messages exchanged with regular peers is thus $\mathcal{O}((n/c)\log n)$. It is easy to see that this bound also holds with high probability (again using a standard Chernoff-type argument). As for any conspirer u, $|\mathcal{N}_u^c| \leq 16\ln(nc)$ with high probability, all conspirers exchange at most $3\log n \cdot 16\ln(nc) \in \mathcal{O}(\log^2 n)$ blocks with other conspirers with high probability. Each conspirer must further send the message M to the other conspirers in its neighborhood, which costs at most $|M| \cdot 16\ln(nc) \in \mathcal{O}(|M|\log n)$ messages. If we combine the bound on the number of messages required to gather enough neighbors, identify the conspirers, and broadcast M, we get the claimed bound on the message complexity. □

Note that for a small c, i.e., $c \in \mathcal{O}(\log n)$, each conspirer has to connect to $\Omega(n)$ random peers to establish a connected conspirer component. In particular, if $c < 8\ln n$, each conspirer connects to all n peers for large n with Algorithm 10.1. In this case, the conspirers can resort to the aforementioned brute-force approach, especially when considering that in *any* broadcast algorithm, a peer must connect to at least $\Omega(n/c)$ peers to find another conspirer.

Note also that if the conspirers only have an estimate of n and c, they can increase the number of neighbors and the length of the exchanged key (continuously) by an appropriate factor to ensure that the presented algorithms still succeed w.h.p.

10.3 Individual Monitoring

Let us now consider an authority that is able to monitor connections individually. By *individually* we mean in this context that the authority can monitor any communication link between any two peers; however, it is not capable of correlating the data gathered at different connections. As we will see, the adversary in this model is stronger in that the size of the message M that can be transmitted depends on the total number of blocks m. In other respects this model is quite similar to the setting in the previous section where the authority acted only as a punitive deterrent. In particular, as long as a conspirer u does not know a neighboring peer's type, it does not make a difference whether or not the link to that peer is monitored as the hidden channel must be used to communicate.

The reason why the size of the message M is limited is that after a conspirer has successfully revealed another conspirer's type, it cannot communicate freely over this link since the monitoring authority could detect this illegal communication. Hence, hidden communication must also be used to transfer the message M. Furthermore, a conspirer v must not request the same file block twice from the same neighbor since the monitoring authority would realize that v is requesting a block it has already received. We can conclude that the maximum size of the message M depends on the number m of blocks as there is no need for additional communication between any two peers once m blocks have been sent in both directions. Consequently, this setting forces the conspirers to use as little blocks as possible for each individual communication link. On a particular link (u, v), however, conspirer u can still underreport on the blocks that it has received from peers other than u, and it can re-request blocks that it has already received from other peers.

In the following, we will outline how to adapt Algorithm 10.1 for this setting. Since each conspirer can still determine its neighbors' types using the REVEALTYPE mechanism, the first part of the algorithm (more precisely, Line 1–8) remain unchanged. Instead of immediately sending message M in plain text, a conspirer u downloads more blocks from its regular neighbors until it has $\Theta(m)$ blocks, preferably all m blocks. In the next step, it reports to each connected conspirer v half of the file blocks that it has gathered in the previous phase. In order to maximize the number of tradable blocks, u and v should report (mostly) disjoint block sets. This can easily be accomplished

by hashing the node identifiers and the secret key to get a sequence of blocks that, e.g., the node with the smaller identifier offers. The other peer performs the same computation and simply offers the blocks not in the sequence. If the two sets are completely disjoint, this may also raise suspicion. Therefore, it may make sense to enlarge the set of offered blocks. This way, each conspirer can exchange $\Theta(m)$ blocks with its conspiring neighbors. Note that for this method to work, each conspirer must remember which blocks it has traded with each of its neighbors in the REVEALTYPE phase in order to avoid re-requesting blocks on individual links. We call this adaptation of the broadcast method $\mathcal{ALG}_{individual}$, for which we can show the following.

Theorem 10.4. *If $c \in [18 \ln n, n/3]$ and $m \geq (24e + 6) \log n$, $\mathcal{ALG}_{individual}$ secretly broadcasts a message of $\Theta(m \log m)$ bits in an individually monitored network w.h.p.*

Proof. Any conspirer u is able to use $(m - |\mathcal{X}_{u,v}| - R)/2$ blocks for transmitting message M to neighbor v, where $\mathcal{X}_{u,v}$ is the set of colliding blocks between u and v in the REVEALTYPE phase, and R is the number of blocks used by REVEALTYPE. From the proof of Theorem 10.3 we know that $|\mathcal{X}_{u,v}| + R$ is at most $6 \log n$ with high probability. Hence, conspirer u can use at least $(m/2) - 3 \log n \in \Theta(m)$ blocks to transmit $\Theta(\log(m!)) = \Theta(m \log m)$ bits to neighbor v over the request order channel. □

10.4 Complete Monitoring

If the *complete* network is monitored, i.e., the authority monitors all communication links and may also correlate data gathered at different links, it gets considerably harder to exchange secret messages. The main difference to individually monitored communication is that a conspirer can no longer underreport, or request a block that it has already received from another peer. In general, the more blocks a peer possesses, the more constrained it is in its actions. Consequently, we have to impose tighter restrictions on the number of conspirers and on the number of blocks in order to enable the conspirers to receive the secret message. Note that the specific restrictions we impose are used for ease of presentation and stronger bounds are again possible by means of a more complicated analysis. Given these conditions, we can still transmit a fairly long message without modifying $\mathcal{ALG}_{individual}$ substantially: The only modification to Algorithm 10.1 is that every conspirer acquires $8\sqrt{n} \ln(nc)$ random blocks instead of only $6 \log n$. We call this adaptation of the algorithm $\mathcal{ALG}_{complete}$, for which we get the following result.

Theorem 10.5. *If $c \geq \sqrt{n} \geq 6$ and $2060\sqrt{n}\ln^2(nc) \leq m \in \Theta(n)$, algorithm $\mathcal{ALG}_{complete}$ secretly broadcasts a message of $\Theta(\sqrt{m}\log^2 m)$ bits in a completely monitored network w.h.p.*

Proof. Each conspirer should not only be able to give its identity to all neighboring conspirers, but also forward the message $|M|$ to each of them. This is only possible if it can request a sufficiently large number of blocks from each neighboring conspirer. Thus, we have to estimate the number of blocks that a certain neighbor u possesses that no other neighbor has. For this purpose, we need to upper bound the number of blocks that each other neighbor acquires during the course of the second phase, where the type of each of their respective neighbors is determined. As we have shown before, the number of neighboring conspirers is at most $16\ln(nc)$ with high probability. A conspirer requests at most $16\ln(nc) \cdot 3\log n$ blocks to reveal its own identity. We have also shown that less than 2 messages on average are exchanged with regular peers in expectation, and thus less than 4 with high probability. If each of those messages is used to request a block, then the total number of acquired blocks is upper bounded by $4\left(8\sqrt{n}\ln(nc) - 16\ln(nc)\right) + 16\ln(nc) \cdot 3\log n < 64\sqrt{n}\ln(nc)$ because $n \geq 36$ by assumption. Since a conspirer has at most $16\ln(nc)$ neighboring conspirers, the goal is now to show that u has enough blocks so that sufficiently many do not occur among the neighbors' $64\sqrt{n}\ln(nc) \cdot 16\ln(nc) = 1024\sqrt{n}\ln^2(nc)$ blocks. We know that u has at least $8\sqrt{n}\ln(nc)$ blocks. The probability that a certain block does not occur in the neighboring conspirers' sets is at least

$$\begin{aligned} 1 - \frac{1024\sqrt{n}\ln^2(nc)}{m - 64\sqrt{n}\ln(nc)} &\geq 1 - \frac{1024\ln(nc)}{(2060\ln(nc) - 64)} \\ &> 1 - \frac{1024\ln(nc)}{2048\ln(nc)} = \frac{1}{2}, \end{aligned}$$

where we used that $m \geq 2060\sqrt{n}\ln^2(nc)$ and $c \geq \sqrt{n} \geq 6$. Thus, it holds that $\mathbb{E}[U] > 4\sqrt{n}\ln(nc)$, where the random variable U denotes the number of such unique blocks. The probability that a conspirer u has only half as many blocks as $\mathbb{E}[U]$ is upper bounded by

$$\mathbb{P}\left[U < \frac{1}{2}\mathbb{E}[U]\right] < e^{-\frac{4\sqrt{n}\ln(nc)}{2\cdot 2^2}} \leq e^{-3\ln(nc)} = \frac{1}{(nc)^3}.$$

Thus, conspirer u has at least $2\sqrt{n}\ln(nc)$ random blocks that no conspiring neighbor has. By means of a union bound, we see that each neighbor in the conspirer subnetwork has that many random blocks to offer with high probability. Moreover, each conspirer can request at least $2\sqrt{n}\ln(nc)$ blocks from each neighboring conspirer with high probability. Since $3\log n$

blocks are used to verify the identity of each conspirer, there are at least $2\sqrt{n}\ln(nc) - 3\log n > (3/2)\sqrt{n}\ln(nc)$ blocks left to transmit the message. This means that indeed $\log\left((3/2\sqrt{n}\ln(nc))!\right) \in \Theta(\sqrt{n}\log^2 n) = \Theta(\sqrt{m}\log^2 m)$ bits can be exchanged secretly, which concludes the proof. □

10.5 Stochastic Monitoring

In the previous sections, we assumed that permuting the request sequence order does not arouse suspicion. In reality, there may be certain policies that restrict such behavior, e.g., it may be common practice to request the least frequently advertised block first (rarest-first) in order to keep all blocks available as long as possible. Streaming applications may demand even stricter policies; the most extreme restriction would be to acquire all blocks in ascending order, which would prohibit using the request order channel completely. However, as requesting blocks in ascending order is not an efficient dissemination scheme, peers typically have the freedom to (randomly) request any block in a certain window. Moreover, if the rarest-first policy is used, the decision which block to request next is made locally, which means that it is not easily possible for a peer to verify that a block that a neighbor has requested is indeed the rarest according to this neighbor's local view. Nontheless, there may be certain request patterns that raise suspicion. If the authority is aware of all legal strategies, it has additional power to expose conspirers. In reference to related work in the field of steganography, e.g. [48], we assume in this section that regular peers choose their request order permutation according to a distribution $\mathcal{C}$, and that all peers as well as the authority know $\mathcal{C}$.[5] As a consequence, a monitoring authority can assign to each request order permutation Π a probability $p(\Pi)$ that Π was generated according to $\mathcal{C}$. If the permutations by a peer u deviate with statistical significance from $\mathcal{C}$, i.e., if $p(\Pi)$ is below a certain threshold ϵ, then the authority might classify u as a conspirer. Choosing a reasonable ϵ, however, is a delicate task. A careful authority may want to prevent false positives in any case, and thus choose $\epsilon = 0$. This implies that the conspirers have to avoid all permutations with probability mass 0 and, if there are such permutations, the capacity of the request order channel is reduced. If a non-zero ϵ is chosen and there are permutations Π with $0 < p(\Pi) < \epsilon$, the authority reduces the request order channel's capacity even more. On the other hand, the authority risks punishing regular peers by increasing ϵ. One approach for the conspirers to adapt to such stochastic monitoring is to come up with an

[5] As noted in [48], the assumption that such a distribution is known or that there is at least an oracle available that generates permutations according to $\mathcal{C}$ is often critical. In a p2p context, however, it is reasonable to a certain extent as there is only a finite number of clients that implement a certain protocol.

Algorithm 10.2 $ENC_{stochastic}$

```
1: i := 0;
2: repeat
3:    Π := ENC_order(M ⊕ K(i)||i);
4:    i++;
5: until p(Π) > ε
6: return Π;
```

adapted mapping of messages to the set of valid permutations, i.e., permutations Π with $p(\Pi) > \epsilon$, or, as this is rather cumbersome, they might use a generic approach such as Algorithm 10.2.

Algorithm 10.2 repeatedly XORs the original message M with a bitstring produced by a pseudo-random generator $\mathcal{K}$ and maps this string to a permutation until a valid permutation is generated. Note that $\mathcal{K}$ is deterministic, and therefore a receiving conspirer can revert $ENC_{stochastic}$ by applying DEC_{order}, extracting i, and XORing the result with $\mathcal{K}(i)$. In order to extract i, the conspirers have to either fix the number of bits used for i or introduce a preamble marking its beginning. The running time of $ENC_{stochastic}$ depends on the distribution $\mathcal{C}$ and the threshold ϵ, and is in the order of the ratio of invalid to valid permutations. The advantage of such a generic approach is that it can even be applied in a completely monitored network, where the validity of a request order permutation generally depends on the requests already sent to other neighbors.

For the special case of $\mathcal{C}$ being the uniform distribution, a standard OAEP scheme [19, 21] would also be sufficient to make the permutations look unsuspicious. As OAEP includes random bits, it would furthermore have the property that when a message is sent over the same link several times the produced request permutation will always look different (unless the same random bits accidentally occur multiple times).

Chapter 11

Related Work

The presented thesis is embedded in a context of various related work. We have pointed out the historical context of our work in both, transactional memory systems, and peer-to-peer file sharing systems. In the following we discuss work that is related to mechanism design with payments, cyclic and inter-swarm trading as well as hidden communication that has not been mentioned so far. As we are the first researchers to attend to the issues of incentives in transactional memory systems, we cannot discuss further related work in that area.

11.1 Mechanism Design with Payments

Game theory (see e.g., [74]), mechanism design (e.g., [72, 55]), and implementation theory (e.g., [63, 64]) have been a flourishing research field for many decades. In 2007, three pioneers in implementation theory, Leonid Hurwicz, Eric Maskin, and Roger Myerson, were awarded the Nobel prize. With the advent of the Internet and its numerous applications such as e-commerce (e.g., [41, 81]), peer-to-peer systems (e.g., [29]), or social networks, algorithmic mechanism design and game theory are extensively studied by computer scientists as well. For instance, game theory is used to shed light onto sociological and economic phenomena in decentralized networks consisting of different interacting stake-holders, and mechanism design is needed to ensure efficiency in online auctions like eBay. For a thorough overview of the field, we refer the reader to the book *Algorithmic Game Theory* [73].

Popular problems in computer science studied from a game theoretic point of view include *routing* [83], *inter-domain routing* [42, 93], *virus propagation* [11], *congestion* [25], *wireless spectrum auctions* [99], among many others. Poor performance of selfish networks requires research for counter-

measures (cf. [29, 64]). Cole et al. [27, 28] have studied how incentive mechanisms can influence selfish behavior in a routing system where the latency experienced by the network traffic on an edge of the network is a function of the edge congestion, and where the network users are assumed to selfishly route traffic on minimum-latency paths. They show that by pricing network edges the inefficiency of selfish routing can always be eradicated, even for heterogeneous traffic in single-commodity networks.

Monderer and Tennenholtz [66] study mechanism design with payments in a minimal rationality model (players choose non-dominated strategies). Among other interesting results, they prove that any pure Nash equilibrium has a *0-implementation* (see also [30, 89, 91]), i.e., it can be transformed into a dominant strategy profile at zero cost. In settings where mechanism design with payments is applicable the price of stability (see e.g., [80]) can thus be achieved for free. Similar results hold for any given ex-post equilibrium of a frugal VCG mechanism. Moreover, the paper addresses the question of the hardness of computing the minimal implementation cost. The same authors study the implementation of *strong equilibria* in [67], and the implementation in games with *incomplete information* together with Ashlagi in [10].

Part I of this thesis extends [66] in various respects. In [66], Monderer and Tennenholtz make no assumptions other than that a player picks a non-dominated strategy, and they analyze the mechanism designers possibilities from a worst-case perspective. In addition to their model, we consider uniform behavior. While we prove mechanism design with payments to be generally hard in the uniform model, we couldn't repair the errors in their discussion of the worst-case model. Moreover, we define the concept of leverage in games and investigate its computational complexity.

In Chapter 4, we discuss the leverage in games from a benevolent and from a malicious mechanism designer's perspective, where the malicious mechanism designer's payoff increases proportionally to the degradation of the social welfare. Other types of maliciousness have been studied before in various contexts, especially in cryptography, and it is impossible to provide a complete overview of this literature. Recently, the concept of *BAR games* [5] has been introduced which aims at understanding the impact of altruistic and malicious behavior in game theory. Moscibroda et al. [69] extend the virus inoculation game from [11] to comprise both selfish and malicious players. A similar model has recently been studied in the context of congestion games [16].

Our work is also related to *Stackelberg theory* [82] where a fraction of the entire population is orchestrated by a global leader. In contrast to our model, the leader is not bound to offer any incentives to follow her objectives. In the recent research thread of *combinatorial agencies* [14, 15, 36], a setting is studied where a mechanism designer seeks to influence the outcome of a

game by contracting the players individually; however, as she is not able to observe the players' actions, the contracts can only depend on the overall outcome.

Our work has also connections to fault-tolerant mechanism design: In [77], Porter et al. extend the field of mechanism design to account for execution uncertainty, where the costs of a player depends on the probabilities of failure. Apart from incentive-compatible mechanisms, they also give impossibility results. Moreover there are intriguing touching points with correlated equilibria and mediated mechanisms, where a mechanism designer can communicate with the players and suggest (without money) certain subset of the outcomes (see, e.g., [67]); indeed, in [66] it is shown that all correlated equilibria can be 0-implemented. Bradonjic et al. [22] also introduced the study of a malicious interested party in the mediator setting.

As a follow-up to our work, Moscibroda and Schmid [68] study an application of the theories devised in Part I to the domain of throughput maximization.

11.2 Cyclic Inter-Swarm Trading

Without a central authority that keeps track of the peers' contributions, and attests to them, it is difficult to manage trust in a fully distributed setting.

The tit-for-tat approach copes with this difficulties in that peers base their decision whether to trust each other, and thus exchange file blocks, only on the balance of uploaded and downloaded data with respect to the requesting peer. It can thus be considered a reputation system in which a peer bases the reputation of another peer only on the history of transactions that have occurred between them. Moreover, a peer can track the state of the history by only keeping one value, the upload-download balance. As such it is simple and easy to implement.

A large body of work studies reputation systems that try to provide a more complete view of a peer's reputation, i.e., the local view of a peer's reputation should account for all transactions that have occurred among all peers in history, or give a good approximation thereof. The advantage of such a more involved system is that a peer can also judge the trustworthiness of peers with which she never had contact. This property is crucial in applications where peers rarely deal with each other more than once. For a p2p file sharing system, however, where peers usually trade many file blocks with the same peer, the inflicted overhead outweighs the benefits. Moreover, designing distributed reputation systems is an inherently hard task: a peer can typically only assess the trustworthiness of a potential trading partner without complete information about all the transactions carried out in the p2p system. And even if a peer receives reports on transactions between two

other peers, she has no immediate means of verifying the correctness of such reports. Despite these inherent difficulties, there have been several proposals of reputation systems for p2p networks, see e.g., [2], [52], or [78]. However, the successful implementation of the desired incentive structures usually requires an assumption on the participants. Aberer and Despotovic [2], for instance, require that "the probability of cheating within a society is comparably low".

With our approach to boost the performance of a tit-for-tat file sharing system by inter-swarm, cyclic trading, we suggest a protocol to extend direct reciprocity with almost simultaneous indirect reciprocity. Menasché et al. [65] have compared the different types of reciprocity. The authors prove that under certain circumstances, direct reciprocity can emulate indirect reciprocity without much performance loss.

We mentioned coding schemes as an alternative approach to increasing the market liquidity of peer-to-peer systems: linear combinations of blocks are computed and distributed in place of the original blocks, yielding a higher block diversity and hence more trading opportunities. It is worth noting that coding schemes can easily be combined with inter-swarm and cyclic trading. The drawback of coding approaches, however, is their high computational complexity. See for instance [61] for a *source coding* scheme, or [4] for a *network coding* scheme.

Guo et al. [44] argue that inter-swarm collaboration in BitTorrent is more effective than, e.g., directly stimulating seeders to stay in the network. They initiate the discussion of a multi-torrent system featuring a tracker site overlay and exchange-based trades along cycles. Aperjis et al. [9] adopt a more theoretical perspective and prove that bilateral equilibrium allocations are not Pareto-efficient in general, in contrast to multilateral allocations. Their work is related to a graph-theoretic generalization of classical *Arrow-Debreu* economics where edges in the graph indicate which entities can engage in direct trades [51]. The authors also provide quantitative insights (using BitTorrent data) into strategies where bilateral exchanges may perform quite well.

Anagnostakis and Greenwald [8] discuss multi-lateral tit-for-tat trading from an incentive-compatibility perspective. Using simulations, the authors show that their proposed algorithm provides real incentives in the sense that free riders experience a poor service. In contrast, we focus on the effect of inter-swarm trading on the throughput in real peer-to-peer networks, and provide simulations based on BitTorrent data. Moreover, we address practically relevant issues and present a distributed implementation that solves the problem of redundant downloads, coordination, and scalability, and also preserves user privacy in the sense that only the peers that engage in active trades with a specific peer learn about this peer's interests.

11.3 Hidden Communication in Peer-to-Peer Systems

There are many use cases for steganography, and several related areas.[1] A well known recent case of steganography are modern color laser printers that add tiny yellow dots to each page to encode the printer's serial number, as well as date and time information [88].

The known history of steganography dates back to Ancient Greece; the link to computing probably dates back to 1983 when Simmons [90] formulated the prisoner's problem where two prisoners should hatch an escape plan while being monitored by the warden. Hopper studies steganography from a cryptographic perspective in [48]. Chapter 3 on *symmetric-key* steganography investigates the setting where two parties that share a secret would like to exchange hidden messages over a monitored public channel. This setting differs from our work in that both parties are mutually aware of each other's type, and that the communication is not limited. Cachin [23] proposes an information-theoretic model, and provides a universal information hiding scheme for the symmetric-key setting. Contrary to our work, *public-key* steganography [17, 48] studies the setting where the sender and receiver do not share a secret key. Another branch of steganography studies information hiding in media files such as images or movies. Media files are especially suited for hiding information since they are large and since human perception easily fails to detect minor modifications (see [53] for a high level overview). Steganography is also used in digital rights management (DRM) and digital watermarking.

Closely related research on *secret handshakes* was conducted by Balfanz et al. in [18] and followed, among others, by Tsudik and Xu [95], and Jarecki and Liu [49]. Secret handshakes are protocols that allow the participants to establish a secure and anonymous communication channel only if they are all members of the same group. In contrast to the settings studied in [18, 95, 49], where the secret handshake is interwoven with an ordinary handshake, a conspirer in our setting cannot initiate a secret handshake by tweaking the ordinary handshake, or by sending an additional message over the standard communication channel, because this would be an illegal, observable deviation from the imposed protocol. We overcome this problem by means of steganographic channels. In that sense, our steganographic handshake can be viewed as a secret handshake conducted on steganographic channels.

Research on anonymity in networks relates to our work in that anonymity also facilitates committing illegal actions without being punished by an authority. However, whereas anonymity and privacy have been classic design

[1] A *shibboleth*, e.g., can be seen as the other side of the medal of a steganographic handshake, in the sense that the sender unintentionally reveals his identity to everybody, for instance by pronouncing a word in a peculiar way.

goals of many p2p systems (e.g., Freenet [26]), there has been little research on steganography in the context of p2p networks. Most existing work essentially uses steganography to hide information in (large) files. In particular, Tsolis et al. [94] propose to use watermarking for copyright protection in p2p systems, and Li et al. [58] propose to hide information in torrent meta files.

Arguably the most closely related work is due to Bickson [20] who shows how to hide content in Gnutella queries to broadcast a message that can only be decrypted by peers that share the sender's secret. In contrast to his work, we do not hide information in media or torrent files, nor in query messages, but exploit the protocol itself. Furthermore, our approach also works in (BitTorrent-like) overlay networks where no lookup mechanism exists. Finally, we also study the feasibility of steganographic handshakes and broadcasts under several monitoring levels.

Chapter 12

Concluding Remarks

The first part of this thesis shows that it is hard for a mechanism designer to compute an optimal implementation of a strategy profile region in a normal form game with payments, at least in a uniform model. The leverage of a game measures the extent to which the game can be harnessed by either a benevolent or a malicious mechanism designer. Unfortunately, computing the leverage is hard in general as well.

Concerning the implications of our findings, we would like to stress that the payments offered by a mechanism designer need not be money. All our results are applicable if the mechanism designer can offer a kind of reward that adds the desired value to a respective game outcome. This may be an increased quality of service, or the provision of another good. Al Capone could, e.g., offer the prisoners in the extended Prisoner's Dilemma to advance in the hierarchy of his gang if they both remain silent. A natural extension of our theory of implementation could be to incorporate negative payments, i.e., punishments, that can also be used by a mechanism designer to manipulate games in her favor.

The second part of this thesis indicates that the inalienable convenience promised by transactional memory in concurrent environments comes with the danger that selfishness degrades the performance of multicore systems: the analysis of contention managers discloses that most TM systems offer wrong incentives to programmers, and encourage them to make transactions large rather than fine grained. Priority-based CMs are prone to be corrupted unless they are based on time only. CMs not based on priority seem to feature incentive compatibility more naturally. We therefore conjecture that by combining randomized conflict resolving with a time-based priority mechanism, chances of finding an efficient, GPI compatible CM are high. Recent work by Schneider et al. [87] can be seen as a successful step in this direction.

Their contention management policy *RandomizedRounds* assigns a random priority to a transaction when it is started or re-started. While this policy proves to give good guarantees on the makespan of a set of transactions, a programmer cannot boost a transaction's priority by making it longer.

An alternative look at GPI compatibility is that contention managers with this property allow a programmer to draw conclusions from a single thread's performance to that of the entire application. Thus, in GPI compatible systems, threads can be tuned individually by a try-and-evaluate approach, whereas without this property, an application must be re-evaluated as a whole to decide whether a modification was beneficial or not. We consider this a huge advantage of GPI compatible systems for software optimization.

Apart from concurrency control for multiple threads within the same process, we see several other domains where transactional memory could also play an important role in the future. Transactional memory might be used for managing access to system wide resources (files, system variables, DBs) or also inter-process communication beyond the boundaries of a single computer. In such settings, incentive compatibility is crucial in order to achieve a maximal system performance.

As transactional memory is still in the fledgling states—the first commercial processors with transactional memory support in hardware have been released at the end of 2011—almost no large scale applications based on transactional memory have been built so far. Thus, the problems of incentives have not cropped up in real applications yet, or at least, they have not been accounted to the lack of beneficial incentives. The next years of concurrent computing will show how severely the incentive deficiencies impact the performance of real systems. The analysis of large TM applications will provide insights into what the actual efficiency loss due to GPI incompatibility is in existing systems, and how accurate the model of selfish, independent programmers is.

In the third part of this thesis, we argue that peer-to-peer file sharing is a prime example of a popular system where taking the participants' selfishness into account is crucial for the success of the protocol. Inter-swarm cyclic trading can boost the market liquidity of a tit-for-tat based file sharing system substantially. Such a system constitutes a feasible incentive-compatible candidate for a next generation file sharing system after BitTorrent.

The generalization of the basic tit-for-tat concept to trading on cycles has interesting game-theoretic implications. For example, it becomes rational for a peer to stay in a swarm after the download is complete, as the acquired content can still be used in cross-swarm trades. This is in stark contrast to bilateral intra-swarm barter, where peers do not have an incentive to offer such content. Thus, the introduction of inter-swarm trades could motivate users to stay in swarms longer, and thereby increase the availability of content.

More work is needed to fully understand such economical aspects of trading on cycles: New forms of strategic behavior may hurt the system performance, e.g., a selfish peer may prefer shorter trading cycles because shorter cycles are easier to find, require less management overhead, and may be more stable. Moreover, while we discussed multiple optimizations and refinements of the basic trading policy, there are aspects of the protocol that might still be improved in future work, e.g., the bandwidth allocation to the different cycles. This can result in additional performance gains.

In most economic systems, goods are exchanged for money. The main reason is that money helps to overcome problems of liquidity: money provides *flexibility* as any service offered can be redeemed with any other person who accepts money. Money provides *temporal freedom* in that a surplus from imbalanced bartering can be stored and redeemed in future transactions. However, barter continues to exist in different forms, e.g., corporate barter, neighborhood barter markets,[1] or organ donation barter.[2] Moreover, the advent of the Internet has opened new possibilities and it has revived barter markets. The Internet serves as a catalyst and platform for various forms of barter-based trading like home exchange, a website that allows its users to swap their home for a few weeks. Cyclic trading could enhance any of the mentioned barter system, also beyond the realm of file sharing.

A next-generation file sharing protocol, like the proposed cyclic tit-for-tat protocol, can supersede BitTorrent in a smooth and incentive-compatible transition by means of a steganographic handshake. Moreover, a subset of peers can use a hidden broadcast primitive to coordinate their actions in secrecy.

While we disclosed several steganographic channels in p2p protocols and successfully exploited them to achieve a steganographic handshake and broadcast in BitTorrent-like p2p systems, most of the channels that we discussed can be encountered not only in p2p protocols, but in many network protocols in general. Permuting packet sequences and introducing artificial delays are potential mechanisms for hidden communication in any network protocol. The techniques that we used can be naturally adapted, or extended to secretly communicate in a variety of network protocols.

As a last remark, we would like to point out that one of the most difficult aspects of game theory and mechanism design is the modelling part. When applying game theory one has to be aware of the limitations of the used models and the derived results: in many real situations the parties involved cannot be determined completely, let alone the exact utility of each

[1] See for instance http://www.huffingtonpost.com/kirsten-dirksen/barter-markets-can-tradin_b_255545.html for a report on barter markets in Barcelona.

[2] Since monetary trade with human organs is prohibited by law, economists have developed organ donation markets that rely on direct—or even multi-lateral—trades (see, e.g., the 2007 Nobel Memorial Prize in Economics).

participant for each outcome. Without this knowledge however, neither our benevolent nor our malicious mechanism designer from Part I, for instance, is able to compute an optimal payment distribution of its available incentives. Moreover, a situation might change unexpectedly resulting in adapted utilities of the players and the carefully calculated payments will not lead to the desired outcome. Behavioral assumptions on the nature of the players are usually stark simplifications of human decision makers. For example, experiments show that people regularly refrain from maximizing their expected utility, e.g., people prefer a sure gain to a higher expected gain if the latter includes the risk of winning nothing (certainty effect). Moreover, in most real-life game situations, the participants have rarely full knowledge of all strategies and outcomes. Further, finding the optimal strategy of a game requires an investment that people are often not prepared to make, if the alternative is to settle for a decent outcome without much effort. For example, most people do not immediately cancel their phone or Internet service subscription and apply for a new subscription at the end of a subscription period, although they could profit from doing so. A model can only try to capture some of the fundamental characteristics and aspects of a system, hoping to allow for a rigorous analysis and best possible solutions.

Despite the obvious problems, we are optimistic that game theory provides useful tools especially for the design of distributed systems, since players are not actually human beings but rather computers that are programmed and controlled by humans. Thus, the strategy space is more predictable and smaller than in a general game theoretic setting. As the positive example of p2p file sharing illustrates, game theoretic analyses and mechanism design can provide crucial intuition and guidance in the design of distributed systems.

Bibliography

[1] K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *Proc. 9th Conference on Cooperative Information Systems (CoopIS)*, pages 179–194, 2001.

[2] K. Aberer and Z. Despotovic. Managing Trust in a Peer-2-Peer Information System. In *Proc. 10th ACM Conference on Information and Knowledge Management (CIKM)*, pages 310–317, 2001.

[3] E. Adar and B. A. Huberman. Free Riding on Gnutella. *Xerox*, 2001.

[4] R. Ahlswede, N. Cai, S.-Y. Li, and R. Yeung. Network Information Flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, 2000.

[5] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR Fault Tolerance for Cooperative Services. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, 2005.

[6] S. Aland, D. Dumrauf, M. Gairing, B. Monien, and F. Schoppmann. Exact Price of Anarchy for Polynomial Congestion Games. In *Proc. 23rd Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 218–229, 2006.

[7] N. Alon, D. Moshkovitz, and M. Safra. Algorithmic Construction of Sets for k-Restrictions. *ACM Transactions on Algorithms (TALG)*, pages 153–177, 2006.

[8] K. Anagnostakis and M. Greenwald. Exchange-based Incentive Mechanisms for Peer-to-Peer File Sharing. In *Proc. 24th International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[9] C. Aperjis, M. Freedman, and R. Johari. A Comparison of Bilateral and Multilateral Exchanges for Peer-Assisted Content Distribution. In *Proc.*

Workshop on Network Control and Optimization (NetCOOP), pages 1–8, 2008.

[10] I. Ashlagi, D. Monderer, and M. Tennenholtz. Mediators in Position Auctions. *Games and Economic Behavior*, 67(1):2–21, 2009.

[11] J. Aspnes, K. Chang, and A. Yampolskiy. Inoculation Strategies for Victims of Viruses and the Sum-Of-Squares Partition Problem. In *Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 43–52, 2005.

[12] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional Contention Management as a Non-Clairvoyant Scheduling Problem. In *Proc. 25th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 308–315, 2006.

[13] R. J. Aumann. Subjectivity and Correlation in Randomized Strategies. *Journal of Mathematical Economics*, 1(1):67–96, 1974.

[14] M. Babaioff, M. Feldman, and N. Nisan. Combinatorial Agency. In *Proc. 7th ACM Conference on Electronic Commerce (EC)*, pages 18–28, 2006.

[15] M. Babaioff, M. Feldman, and N. Nisan. Mixed strategies in combinatorial agency. In *Proc. 2nd International Workshop on Internet and Network Economics (WINE)*, pages 353–364, 2006.

[16] M. Babaioff, R. Kleinberg, and C. H. Papadimitriou. Congestion Games with Malicious Players. In *Proc. 8th ACM Conference on Electronic Commerce (EC)*, pages 103–112, 2007.

[17] M. Backes and C. Cachin. Public-Key Steganography with Active Attacks. In *Proc. 2nd Theory of Cryptography Conference (TCC)*, pages 210–226, 2005.

[18] D. Balfanz, G. Durfee, N. Shankar, D. Smetters, J. Staddon, and H.-C. Wong. Secret Handshakes from Pairing-Based Key Agreements. In *Proc. IEEE Symposium on Security and Privacy (SP)*, pages 180–196, 2003.

[19] M. Bellare and P. Rogaway. Optimal Asymmetric Encryption. In *Advances in Cryptology (EUROCRYPT)*, 1994.

[20] D. Bickson. Steganographic Communications Using the Gnutella Network. Master's thesis, Hebrew University of Jerusalem, 2003.

[21] V. Boyko. On the Security Properties of OAEP as an All-or-Nothing Transform. In *Advances in Cryptology (CRYPTO)*, pages 503–518, 1999.

[22] M. Bradonjic, G. Ercal-Ozkaya, A. Meyerson, and A. Roytman. On the Price of Mediation. In *Proc. ACM Conference on Electronic Commerce (EC)*, pages 315–324, 2009.

[23] C. Cachin. An Information-theoretic Model for Steganography. *Information and Computation*, 192(1):41–56, 2004.

[24] G. Christodoulou and E. Koutsoupias. The Price of Anarchy of Finite Congestion Games. In *Proc. 37th ACM Symposium on Theory of computing (STOC)*, pages 67–73, 2005.

[25] G. Christodoulou and E. Koutsoupias. The Price of Anarchy of Finite Congestion Games. In *Proc. 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 67–73, 2005.

[26] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *International Workshop on Designing Privacy Enhancing Technologies*, pages 46–66, 2001.

[27] R. Cole, Y. Dodis, and T. Roughgarden. How Much Can Taxes Help Selfish Routing? In *Proc. 4th ACM Conference on Electronic Commerce (EC)*, pages 98–107, 2003.

[28] R. Cole, Y. Dodis, and T. Roughgarden. Pricing Network Edges for Heterogeneous Selfish Users. In *Proc. 35th Annual ACM Symposium on Theory of Computing (STOC)*, pages 521–530, 2003.

[29] R. K. Dash, N. R. Jennings, and D. C. Parkes. Computational-Mechanism Design: A Call to Arms. *IEEE Intelligent Systems*, pages 40–47, November 2003.

[30] P. Dybvig and C. Spatt. Adoption Externalities as Public Goods. *Journal of Public Economics*, 20:231–247, 1983.

[31] R. Eidenbenz, T. Locher, S. Schmid, and R. Wattenhofer. Boosting Market Liquidity of Peer-to-Peer Systems Through Cyclic Trading. Under submission, 2012.

[32] R. Eidenbenz, T. Locher, and R. Wattenhofer. Hidden Communication in P2P Networks: Steganographic Handshake and Broadcast. In *Proc. 30th IEEE International Conference on Computer Communications (INFOCOM)*, 2011.

[33] R. Eidenbenz, Y. A. Oswald, S. Schmid, and R. Wattenhofer. Manipulation in Games. In *Proc. 18th International Symposium on Algorithms and Computation (ISAAC)*, 2007.

[34] R. Eidenbenz, Y. A. Oswald, S. Schmid, and R. Wattenhofer. Mechanism Design by Creditability. In *Proc. 1st International Conference on Combinatorial Optimization and Applications (COCOA)*, 2007.

[35] R. Eidenbenz, Y. A. Pignolet, S. Schmid, and R. Wattenhofer. Cost and Complexity of Harnessing Games with Payments. *International Game Theory Review (IGTR)*, 13(1), 2011.

[36] R. Eidenbenz and S. Schmid. Combinatorial Agency with Audits. In *Proc. IEEE International Conference on Game Theory for Networks (GameNets)*, 2009.

[37] R. Eidenbenz and R. Wattenhofer. Good Programming in Transactional Memory: Game Theory Meets Multicore Architecture. *Theoretical Computer Science*, 412(32):4136–4150, 2011.

[38] P. Erdős and A. Rényi. On Random Graphs. *Publicationes Mathematicae*, 6:290–297, 1959.

[39] P. Erdős and A. Rényi. On the Evolution of Random Graphs. *Publication of the Mathematical Institute of the Hungarian Academy of Science*, 5:17–61, 1960.

[40] U. Feige. A Threshold of $\log n$ for Approximating Set Cover. *Journal of the ACM*, pages 634–652, 1998.

[41] J. Feigenbaum and S. Shenker. Distributed Algorithmic Mechanism Design: Recent Results and Future Directions. In *Proc. 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 1–13, 2002.

[42] L. Gao and J. Rexford. Stable Internet Routing Without Global Coordination. *IEEE/ACM Transactions on Networking*, 9(6):681–692, 2001.

[43] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proc. 24th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 258–264, 2005.

[44] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding, and X. Zhang. Measurements, Analysis, and Modeling of BitTorrent-like Systems. In *Proc. 5th ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2005.

[45] M. Herlihy. The Multicore Revolution: the Challenges for Theory. In *Proc. 27th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 1–8, 2007.

[46] M. Herlihy, V. Luchangco, and M. Moir. A Flexible Framework for Implementing Software Transactional Memory. *SIGPLAN Notifications*, 41(10):253–262, 2006.

[47] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *SIGARCH Computer Architecture News*, 21(2):289–300, 1993.

[48] N. J. Hopper. *Toward a Theory of Steganography.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2004.

[49] S. Jarecki and X. Liu. Private Mutual Authentication and Conditional Oblivious Transfer. In *Advances in Cryptology (CRYPTO)*, pages 90–107, 2009.

[50] S. Jun and M. Ahamad. Incentives in BitTorrent Induce Free Riding. In *Proc. ACM SIGCOMM Workshop on Economics of Peer-to-Peer Systems (P2PECON)*, pages 116–121, 2005.

[51] S. M. Kakade, M. Kearns, and L. E. Ortiz. Graphical Economics. In *Proc. 17th Annual Conference on Learning Theory (COLT)*, pages 17–32, 2004.

[52] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The Eigentrust Algorithm for Reputation Management in P2P Networks. In *Proc. 12th International Conference on World Wide Web (WWW)*, pages 640–651, 2003.

[53] G. C. Kessler. An Overview of Steganography for the Computer Forensics Examiner. 6(3), 2004.

[54] R. Landa, D. Griffin, R. G. Clegg, E. Mykoniati, and M. Rio. A Sybilproof Indirect Reciprocity Mechanism for Peer-to-Peer Networks. In *Proc. 28th IEEE International Conference on Computer Communications (INFOCOM)*, 2009.

[55] R. Lavi and C. Swamy. Truthful and Near-optimal Mechanism Design via Linear Programming. In *Proc. 46th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 595–604, 2005.

[56] E. Lee. The Problem with Threads. *Computer*, 39(5):33–42, 2006.

[57] D. Levin, K. LaCurts, N. Spring, and B. Bhattacharjee. Bittorrent is an Auction: Analyzing and Improving BitTorrent's Incentives. In *Proc. ACM Conference on Data Communication (SIGCOMM)*, pages 243–254, 2008.

[58] Z. Li, X. Sun, B. Wang, and X. Wang. A Steganography Scheme in P2P Network. In *Proc. 4th International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP)*, pages 20–24, 2008.

[59] G. Linden, B. Smith, and J. York. Amazon.com Recommendations: Item-to-Item Collaborative Filtering. *Internet Computing, IEEE*, 7(1):76 – 80, 2003.

[60] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free Riding in BitTorrent is Cheap. In *Proc. 5th Workshop on Hot Topics in Networks (HotNets)*, 2006.

[61] T. Locher, S. Schmid, and R. Wattenhofer. Rescuing Tit-for-Tat with Source Coding. In *7th IEEE International Conference on Peer-to-Peer Computing (P2P), Galway, Ireland*, 2007.

[62] D. B. Lomet. Process Structuring, Synchronization, and Recovery Using Atomic Actions. *SIGOPS Operating Systems Review*, 11(2):128–137, 1977.

[63] E. Maskin. Review of Economic Studies. *Nash Equilibrium and Welfare Optimality*, pages 23–38, 1999.

[64] E. Maskin and T. Sjöström. *Handbook of Social Choice Theory and Welfare (Implementation Theory)*. North-Holland, Amsterdam, 2002.

[65] D. Menasché, L. Massoulié, and D. Towsley. Reciprocity and Barter in Peer-to-Peer Systems. In *Proc. 19th IEEE International Conference on Computer Communications (INFOCOM)*, 2010.

[66] D. Monderer and M. Tennenholtz. k-Implementation. *Journal of Artificial Intelligence Research (JAIR)*, 21:37–62, 2004.

[67] D. Monderer and M. Tennenholtz. Strong Mediated Equilibrium. *Artificial Intelligence*, 173(1):180–195, 2009.

[68] T. Moscibroda and S. Schmid. On Mechanism Design Without Payments for Throughput Maximization. In *Proc. 18th IEEE International Conference on Computer Communication (INFOCOM)*, 2009.

[69] T. Moscibroda, S. Schmid, and R. Wattenhofer. When Selfish meets Evil: Byzantine Players in a Virus Inoculation Game. In *Proc. 25th annual ACM symposium on Principles of distributed computing (PODC)*, pages 35–44, 2006.

[70] J. F. Nash. Equilibrium Points in n-Person Games. *Proc. National Academy of Sciences of the United States of America*, 36(1):48–49, 1950.

[71] J. F. Nash. Non-Cooperative Games. *The Annals of Mathematics*, 54(2):286–295, 1951.

[72] N. Nisan and A. Ronen. Algorithmic Mechanism Design. In *Proc. 31st ACM Symposium on Theory of Computing (STOC)*, pages 129–140, 1999.

[73] N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani. *Algorithmic Game Theory*. Cambridge, 2007.

[74] M. J. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, 1994.

[75] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do Incentives Build Robustness in BitTorrent? In *Proc. 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2007.

[76] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. *Theory of Computing Systems*, 32(3):241–280, 1999.

[77] R. Porter, A. Ronen, Y. Shoham, and M. Tennenholtz. Fault Tolerant Mechanism Design. *Artif. Intell.*, 172(15):1783–1799, 2008.

[78] A. Rahbar and O. Yang. PowerTrust: A Robust and Scalable Reputation System for Trusted Peer-to-Peer Computing. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):460–473, 2007.

[79] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 161–172, 2001.

[80] E. Resnick, Y. Bachrach, R. Meir, and J. S. Rosenschein. The Cost of Stability in Network Flow Games. In *Proc. Mathematical Foundations of Computer Science (MFCS)*, pages 636–650, 2009.

[81] J. S. Rosenschein and G. Zlotkin. *Rules of Encounter*. MIT Press, 1994.

[82] T. Roughgarden. Stackelberg Scheduling Strategies. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 104–113, 2001.

[83] T. Roughgarden. *Selfish Routing and the Price of Anarchy*. MIT Press, 2005.

[84] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *IFIP/ACM Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[85] W. N. Scherer III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs (CSJP)*, 2004.

[86] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proc. 24th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 240–248, 2005.

[87] J. Schneider and R. Wattenhofer. Bounds On Contention Management Algorithms. In *Proc. 20th International Symposium on Algorithms and Computation (ISAAC)*, pages 441–451, 2009.

[88] D. Schoen. Investigating Machine Identification Code Technology in Color Laser Printers. The Electronic Frontier Foundation, `www.eff.org`, 2005.

[89] I. Segal. Contracting with Externalities. *The Quarterly Journal of Economics*, 2:337–388, 1999.

[90] G. J. Simmons. The Prisoners' Problem and the Subliminal Channel. In *Advances in Cryptology (CRYPTO)*, pages 51–67, 1983.

[91] R. Spiegler. Extracting Intercation-Created Surplus. *Games and Economic Behavior*, 30:142–162, 2000.

[92] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. ACM conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 149–160, 2001.

[93] M. Suchara, A. Fabrikant, and J. Rexford. BGP Safety with Spurious Updates. In *Proc. 30th IEEE International Conference on Computer Communications (INFOCOM)*, pages 2966–2974, 2011.

[94] D. Tsolis, S. Sioutas, and T. Papatheodorou. Digital Watermarking in Peer to Peer Networks. In *Proc. 16th International Conference on Digital Signal Processing (DSP)*, pages 1086–1090, 2009.

[95] G. Tsudik and S. Xu. A Flexible Framework for Secret Handshakes. In *Proc. 6th Workshop on Privacy Enhancing Technologies (PET)*, pages 295–315, 2006.

[96] R. van der Hofstad. Random Graphs and Complex Networks. *Unpublished manuscript*, 2007.

[97] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A Secure Economic Framework for Peer-to-Peer Resource Sharing. In *Proc. Workshop on Economics of Peer-to-Peer Systems*, 2003.

[98] C. Zhang, P. Dhungel, D. Wu, and K. W. Ross. Unraveling the BitTorrent Ecosystem. *IEEE Transactions on Parallel and Distributed Systems*, 22(7), 2010.

[99] X. Zhou, S. Gandhi, S. Suri, and H. Zheng. eBay in the Sky: Strategy-Proof Wireless Spectrum Auctions. In *Proc. 14th ACM International Conference on Mobile Computing and Networking (MOBICOM)*, pages 2–13, 2008.

Curriculum Vitae

September 13, 1980	Born in Davos, Switzerland
1987–2000	Primary, secondary, and high schools in Davos Platz/GR, Switzerland
2001	Drumming certificate, Drummer's Collective, New York, USA
2001–2007	Studies in computer science and political science, ETH Zurich, Switzerland
March 2007	M.Sc. in computer science, ETH Zurich, Switzerland
2004–2007	Software engineering, Triboni AG, Switzerland
2008–2012	Ph.D. student, research and teaching assistant, Distributed Computing Group, Prof. Roger Wattenhofer, ETH Zurich, Switzerland