HARVARD UNIVERSITY
Graduate School of Arts and Sciences

DISSERTATION ACCEPTANCE CERTIFICATE

The undersigned, appointed by the

School of Engineering and Applied Sciences

have examined a dissertation entitled

"Rationally Motivated Failure in Distributed Systems"

presented by Jeffrey Allen Shneidman

candidate for the degree of Doctor of Philosophy and hereby certify that it is worthy of acceptance.

Signature [illegible]
Typed name: Prof. M. Seltzer

Signature [illegible]
Typed name: Prof. D. Parkes

Signature [illegible]
Typed name: Dr. J. Waldo

Date: July 28, 2008

Rationally Motivated Failure in Distributed Systems

A dissertation presented
by

Jeffrey Allen Shneidman

to
The School of Engineering and Applied Sciences
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the subject of

Computer Science

Harvard University
Cambridge, Massachusetts
November 2008

Dissertation Advisors
Professor Margo Ilene Seltzer
Professor David Christopher Parkes

Author
Jeffrey Allen Shneidman

Rationally Motivated Failure in Distributed Systems

Abstract

Modern distributed systems are under threat from a surprising source: the willful failure by their own users. These failures are not due to chance, nor are they maliciously motivated. Rather, these failures are *rationally* motivated.

Despite evidence of rationally motivated failure in real systems, there has been surprisingly little work on rational behavior in the context of system fault tolerance. This thesis describes a form of defensive design used to build distributed systems that are robust to rationally motivated failure. Our approach proactively prevents rationally motivated failure by addressing the underlying failure cause. This approach differs markedly from traditional distributed system techniques of dealing with failure, which reactively seek to recover from an expressed failure.

This thesis makes four main contributions toward understanding and designing for rationally motivated failure. This thesis...

- ...formalizes *faithfulness* as the metric by which to judge an algorithm's tolerance to rationally motivated failure. A proof of specification faithfulness is a certification that rational nodes will choose to follow the algorithm specified by the system designer.
- ...reveals *information revelation failure* as an important type of failure expression, and shows that the feasibility and cost of addressing information revelation failure is different from other types of failure.
- ...proposes a practical three-part methodology for building systems that can prevent rationally motivated failure expression.
- ...applies this methodology to two problems: First, we use the rationally motivated failure remedy methodology to extend an existing interdomain routing protocol and prove the extension to be faithful under the *ex post Nash* solution concept. We implement the extended algorithm in a simulator and compare the message complexity with the original unfaithful algorithm. Second, we apply the methodology to the problem of distributed consensus to solve a scenario called the *Rational Byzantine Generals*

problem. We design an algorithm that is faithful under the *1-partial ex post Nash* solution concept. We implement the algorithm over a network and evaluate the fault tolerance, message complexity, and scalability by comparing the new algorithm with appropriate existing protocols.

Contents

Citations to Previously Published Work

Earlier versions of content in Chapters 4 and 5 have appeared in the following paper:

> "Specification Faithfulness in Networks with Rational Nodes", J. Shneidman and D. C. Parkes, Proc. 23rd ACM Symp. on Principles of Distributed Computing (PODC'04), St. John's, Canada, 2004.

Ideas related to the *partition principle* appear in the following paper:

> "Distributed Implementations of Vickrey-Clarke-Groves Mechanisms", D. C. Parkes and J. Shneidman, Proc. 3rd Int. Joint Conf. on Autonomous Agents and Multi Agent Systems (AAMAS'04), New York, NY, 2004.

Acknowledgments

One of the greatest gifts that a researcher can make to a graduate student is the sentence that begins, "We assume..." The origins of this thesis are found in such a statement, and in a basic misunderstanding that I had in 1999 while attending Harvard as a visiting student. During that year, I took my first distributed systems class from Jim Waldo and came across the problem of *distributed agreement with faults.* This canonical problem concerns how to establish a consistent view of a piece of data across a distributed group of participants. When I first heard the problem, I mistook the problem to be the establishment of the *validity* of the data content itself, and not simply the establishment of a common view on the data content. It seemed to me impossible to check the correctness of a piece of revealed information, and indeed this was not the intention of the algorithm.

Fast forward to 2002. During my first year as a Ph.D. student, I had taken Nancy Lynch and Robert Morris's classes in distributed systems at M.I.T. and had settled into distributed systems as my intended focus. A new professor, David Parkes, was offering a class at Harvard in "Computational Mechanism Design" with a few lectures that looked relevant to distributed computing. I knew nothing else about the topic. What I found in David's course was a new field of computer science (the earliest papers dating from the late 90s) that provided a way to guarantee the correctness of a piece of revealed information under certain conditions.

It seemed exciting and obvious to me that the lessons and techniques learned in that class should be merged with ideas from distributed computing. My final project in that class paved the way to a qualifying exam topic that challenged an underlying assumption in Feigenbaum et al.'s interdomain routing protocol and finally to this thesis.

I am grateful to a number of people for helping me on my path to a Ph.D. First, I would like to thank my committee, consisting of Margo Seltzer, David Parkes, and Jim Waldo. Each provided a very different perspective on my thesis, and their feedback taken together dramatically increased the quality of this final document. I was grateful at times for the feeling that my advisors had not forgotten the challenges of writing their own theses. When the writing process was frustratingly slow, David in particular was fantastic, offering insightful feedback on whatever prose I *had* produced that week, which often lifted my mood and served as strong encouragement to keep going. Apart from their thesis advising duties, I feel especially lucky to have had Margo and David as advisors during my graduate school career. They demonstrate that it is possible to be both brilliant and kind in academia. I want to thank Margo, who is a wonderful vetter of ideas, even though she hates it when I make up words like vetter. Margo allowed this thesis to happen by encouraging me to do interesting research and supporting me even as that research led away from traditional

work in systems.

I would like to thank Susan Wieczorek who has been a supportive non-faculty resource over the last seven years. Throughout the process, ups and downs were shared with officemates Jonathan Ledlie, Chaki Ng, Lex Stein, and Uri Braun, and with members of the ICE team, including Benjamin Lubin, Adam Juda, Hassan Sultan, Ruggiero Cavallo, and Sebastien Lahaie, and Professor Peter Pietzuch and the rest of the EconCS and Syrah project groups. I would also like to thank Professors Michael Mitzenmacher, Joe Hellerstein, Rahul Sami, Joan Feigenbaum, and the systems and networking team at Microsoft Research U.K. including Peter Key, Laurent Massoulié, Miguel Castro and Ant Rowstron. Finally, I would like to thank my loving and supportive family, Mom and Dad, Laura and Ed, and last but most, my wife Danni.

Portions of this work and my other graduate research were supported in part by NSF grants IIS-0238147 and ACI-0330244.

For the folks.
And for Ed,
who first pointed me eastward.

Chapter 1

Introduction

Modern distributed systems are under threat from a surprising source: the willful failure by their own users. From the Internet Transmission Control Protocol (TCP) [SCWA99] to the protocols that enable the the search for extra-terrestrial life [Kah01], failures occur when self-interested participants *choose*, for selfish reasons, not to follow the system-specified algorithm. In such cases, the system suffers collateral damage from the deliberate failure of its own participants. The side effects of a participant's decision to fail may drive the system into an undesirable state.

These failures are not due to chance, nor are they maliciously motivated. Rather, these failures are *rationally motivated.* This thesis describes a form of defensive design that should be used to build distributed systems that are robust to rationally motivated failure. Our approach proactively *prevents* rationally motivated failure expression by understanding and addressing the underlying failure cause. This approach differs markedly from traditional distributed systems techniques of dealing with failure, which reactively seek to *recover* from an expressed failure.

1.1 Rationally Motivated Failure

Rationally motivated failure is the subset of intentional behavior that occurs when a participant aims to better its outcome in a distributed algorithm, resulting in a deviation from the correct system algorithm. For now, we will satisfy ourselves with this intuitive definition, but we formalize this concept later in Chapter 3. Rationally motivated failure is a concern when it is damaging to other participants or to an overall system goal. In this thesis, we show that the key difference between failures due to rational behavior and failures due to other causes is that rationally motivated failures can be avoided. Rational behavior is not arbitrary; rationally motivated failure is a consequence of system design that conflicts with a user's goal. Rational behavior is entirely *predictable*, assuming that one can correctly model rational participants, and *avoidable*, assuming that one is willing and able to change the system design.

1.1.1 Examples: Rationally Motivated Failure in Systems

One can readily find examples of failure due to rational behavior. In this section, we describe four such instances of rationally motivated failure. We describe each system, identify the rational behavior and failure opportunity, and examine the impact of the failure.

TCP Hacking

The Transmission Control Protocol (TCP) is a core Internet protocol used by higher level protocols to provide reliable and ordered message delivery. In TCP, a receiver provides an acknowledgment message to the sender and the contents of this message determine the sender's transmission rate.

Rationally Motivated Failure. Because a TCP sender always trusts the contents of the receiver's acknowledgment message, a performance-hungry receiver can force a sender to send data far faster than is fair in a multi-user environment. Savage et al. describe manipulation opportunities that arise from a specification oversight in TCP's congestion control mechanism [SCWA99]. For example, in one of Savage's manipulations the receiver preemptively acknowledges packets that have not yet been sent.

Cost and Effect. Savage et al. demonstrate how their attacks slashed HTTP download times [SCWA99] for the manipulating user and report how an early experiment [Jan00] saturated their university network with a single download! They speculate that if widely deployed these manipulations would lead to Internet congestion collapse. The initial cost of these changes was that a programmer modified 11-45 lines of source code for each manipulation. Presumably, these manipulations have not caught on, or else the Internet would have ground to a halt. However, a new cottage industry of vendors provides "high-performance" replacement TCP drivers aimed at gamers, file sharing aficionados, and Voice-over-IP users [cFo06]. So far, these drivers seem to stick to less system-harmful ideas like traffic shaping and ACK-promoting, but commercial pressures could provide an incentive to enable more inventive and system-damaging behavior.

Seti@Home Errors

Seti@Home is a distributed computation system where participants run clients that download raw data from a server, perform computation, and upload finished "work units" to the server. There is a scoring system that rewards users with (otherwise meaningless) points tied to their level of work submission.

Rationally Motivated Failure. While the majority of users contribute their computer time to further a research goal or to feel useful, about ten percent of users report that the competition of getting a high score on the leader boards is their primary motivation for participating in the system [BOI06]. Moreover, another set of users reports the leader board as a secondary motivation for participation, even justifying the purchase of new faster hardware to increase their score [Use06].

Early versions of the Seti@home client were hacked to upload junk, labeled as finished work, in order to drive up their leader board score. These manipulations were successful because early leader board algorithms trusted clients in their work reporting.

Cost and Effect. The benefit to some users of increased leader board rankings (and related feelings of accomplishment) outweighed the cost of manipulation and of any feelings for damaging the scientific experiment. Moreover, once this cost was incurred by one individual, the hacked software was widely distributed through websites, chat sessions and e-mail [Kah01]. The system cost of this rational manipulation is correctness, and later, lost efficiency though increased need to verify user calculations.

Participation Problems in Database Systems

Distributed database systems allow users to execute queries across data that is distributed around a network. The execution of a computationally expensive operation (such as a database *join*) can take place far from the user that originated the query.

Rationally Motivated Failure. When a rational participant feels that its cost of participating in a system outweighs its benefit, the participant can simply stop participating. Stonebraker and Hellerstein blame the lack of widespread distributed database adoption in the 1990s partly for this rationally motivated failure of participation. In these database systems, when "the optimizer decides that a join will happen on a particular machine, the local system administrator is powerless to forbid it, other than by refusing to participate in the distributed system at all" [SH03]. Database administrators could not express their costs and had little incentive to remain in the system. Denying remote queries in favor of local users removed much of the supposed benefit of the system.

Cost and Effect. Participants chose not to participate in the system because of real imposed costs and inconvenience. This behavior was particularly damaging when the opted-out participant had private data that was not shared by other users. The system cost was the risk of failure when too many users chose not to participate.

File Sharing Manipulations

Peer to peer file sharing systems enable users to store, locate, and retrieve files from other users. These systems' success relies on the network effect generated when many users choose to participate in the system.

Rationally Motivated Failure. Peer-to-peer file sharing systems showcase a host of rationally motivated manipulations. Most simply, if participants do not anticipate a benefit from their participation, they simply opt not to participate. File sharing systems such as MojoNation [McC01], Kazaa [Net07], and BitTorrent [Coh03] have attempted to address the partial participation *free riding* problem by adding incentives for participation.

The Gnutella [AH00] file sharing network is an early example where users' *partial participation* damaged the overall quality of the system. In Gnutella, participants had private information (files) that they were unwilling to share, because doing so only increased consumed bandwidth, risk of legal action, etc. These users simply consumed resources offered by less selfish participants, resulting in a host of network problems [AH00, HCW05].

Kazaa, a successive file sharing program, addressed this free riding problem by including a "participation level" score that allowed other peers to verify that a participant was fully participating in file exchanges. As a result, participants with high participation scores were more likely to get good service on the Kazaa network. However, this score was self-reported by peers and subject to manipulation; within two years, unofficial user-modified clients were released that allowed a user to lie and claim a high participation score [Kat]. This modified client eventually dominated the regular client and free-riding behaviors began to dominate the system [Net07]. More elaborate manipulative strategies were later published and the resulting system degradation played a role in the attrition of millions of users from the system [Net07] as they looked for better file sharing alternatives.

BitTorrent is a peer-to-peer file sharing system that works by coordinating users to exchange file pieces with each other, while allowing users to download random pieces from other users that have complete copies of the file. BitTorrent was designed specifically to address the types of rational manipulations that had stymied earlier file sharing systems. But even in BitTorrent numerous manipulations have been found. Our previous work was the first to report such manipulations [SPM04]. Since that work was published, other teams have built and published actual BitTorrent compatible software clients that rely on false identity and selfish protocol manipulations to increase their download speed (e.g., BitThief [LMSW06]). A more recent version of the BitTorrent protocol has been modified in a further attempt to combat free riding behavior. This revision relies on a centralized component called the *tracker* to act as a policeman to monitor participation. In this extension, the client software employs a Kazaa-like self-reported participation metric [Net07]. As might be expected, manipulative clients have now been released that allow users to tweak their self-reported scores, such as GreedyTorrent [Tor07]. BitTyrant [PIA$^+$07] is an another example of a system that manipulates partner selection, going against the designer's protocol fairness intentions. Researchers have also pointed out how modified clients are not needed since some of the design features meant to induce cooperation actually induce free riding [JA05].

Cost and Effect. The above discussion describes a slew of manipulations that have varying implementation costs, ranging from the simple changing of application parameters to editing of application code. These manipulations have varying levels of effect, ranging from free riding through partial participation to aggressive damage of the overall network.

1.1.2 Rationally Motivated Fault Tolerance: An Evolutionary Response

Despite evidence of rationally motivated failure in real systems, there has been surprisingly little work on rational behavior in the context of system fault tolerance. With the exception of the recent *autonomous nodes* and *BAR fault tolerance* papers [MT02, AAC$^+$05] discussed in the next chapter, we are not aware of work that attempts to bridge studies of rational behavior with prior work in fault tolerant distributed algorithms. We hypothesize that rationally motivated failure is only now receiving attention from the systems community for the following reasons:

- The Internet has given rise to historically unprecedented "open" distributed systems. As Christos Papadimitriou writes [Pap01], "the Internet is unique among all computer systems in that it is built, operated, and used by a multitude of diverse economic interests, in varying relationships of collaboration and competition with each other." Distributed systems are no longer closed systems monitored closely by a group of common-interest researchers, and along with an increase in the number of diverse participants, both the number of distributed algorithms and the number of algorithm implementations have also increased. As participants' needs and designers' goals conflict, one might expect rational behavior to play a more prominent role.

- Some of the tools for addressing rationally motivated failure have not been well-known to system designers. For example, this thesis combines systems techniques with ideas from economics to build better algorithms. A relevant area of economics known as *mechanism design* has historically been overlooked by systems researchers, perhaps being labeled as too theoretical. Systems researchers have been limited by working in a *fault-exclusion* mindset and by relying on existing Byzantine Fault Tolerance techniques [CL99] as a band-aid to rationally motivated deviations. In contrast, designing for rational participants requires a designer to work with *fault-prevention* techniques.

1.2 Thesis Contributions

This thesis makes four main contributions toward understanding and designing for rationally motivated failure.

1.2.1 Information Revelation Failure Expression

This thesis points out one important type of failure expression that is well-known within economics but has been overlooked or mis-classified by the distributed systems community: *information revelation failure.* Information revelation failure occurs when an algorithm participant is instructed by the algorithm specification to reveal a piece of private information, but the participant does not reveal this information correctly. Private information is information that is not known to other algorithm participants and that cannot be established or verified by other algorithm participants. This thesis shows that the feasibility and cost of addressing information revelation failure can be different from other types of failure and suggests incentives and design restrictions as two ways to prevent this type of failure.

1.2.2 Faithfulness

In the spirit of systems properties like *safety* and *liveness* [Wei92], we formalize *faithfulness* as the metric by which to judge an algorithm's tolerance to rationally motivated failure. A proof of specification faithfulness is a certification that rational nodes will choose to follow the algorithm specified by the system designer. This certification is given subject to assumptions about the types of nodes and failure that may be present in the system. When one can prove that a specification will be faithfully followed by rational nodes in a distributed network, one can certify the system to be *incentive-*, *communication-*, and *algorithm-compatible* (IC, CC and AC). Such a system is provably robust against rational manipulation.

Through our work on faithfulness, we also introduce the classic economic ideas of *solution concept* and *equilibrium behavior* into traditional distributed systems failure analysis. While our work is not the first work to explore distributed mechanism design (see Feigenbaum [FS02]), to our knowledge, this thesis is the first work to expand the notion of equilibrium in distributed systems to include communication and computation actions.

Our ultimate goal is show how incentives can be used to change a participant's behavior from *choosing to exhibit failure* to *choosing to follow the suggested algorithm.*

1.2.3 Methodology to Address Rationally Motivated Failure

A main contribution of this thesis is to present a practical methodology for building systems that can prevent rationally motivated failure expression. The methodology consists of three parts, which we preview below.

1. **Model the Environment.** In this step, the designer specifies the *target environment assumptions*, which dictate the node model, knowledge model, and network model. This bundle of assumptions specifies the conditions underlying any claim of rationally motivated fault tolerance.

2. **Design, Prove, and Build Specification.** The designer then constructs a new system specification that is designed to prevent rationally motivated failure. Specifically, the system designer should:

   - Identify a subset of rational compatible behavior. The designer's goal is to provide incentives for correct behavior to participants for those intentional failures that might be caused by this rational compatible subset.
   - Specify the *mechanism.* The mechanism maps between node strategies and the system outcomes. The mechanism is defined by a *strategy space* and an *outcome rule* that specifies the actual mapping.
   - Propose a *suggested node strategy* for each *participant type.*
   - Guarantee that the suggested strategy is a utility maximizing strategy for all participant types. In other words, the designer guarantees that a rational compatible node will exhibit *rationally motivated correctness* rather than *rationally motivated failure.* This guarantee is a rigorous proof that assumes a particular *solution concept* and holds in the chosen *environment.*

The designer then realizes the specification with an implementation.

3. **Evaluate Effectiveness, Impact, and Cost.** Designers evaluate the trade-offs required to implement a system that is robust to rationally motivated failure and measure the cost (in messages, etc.) of the system.

1.2.4 Application of Methodology

We apply the rationally motivated failure remedy methodology to two problems. First, we build an algorithm for the interdomain routing problem based on work by Feigenbaum et al. [FPSS02] (FPSS). We show the enhanced algorithm to be faithful under the *ex post Nash* solution concept. We implement the algorithm in simulation, evaluating and comparing the new algorithm's message complexity with FPSS. We show that a node's message cost when running the faithful algorithm depends on its *degree* (number of connections it has to other nodes) and that on a real Internet topology a node may incur a 2x-100x message traffic increase over the unfaithful version of the algorithm. We show how this overhead can be reduced to 2x-10x without serious connectivity consequences when high-degree nodes impose a cap on their number of neighbors.

Second, we apply the rationally motivated failure remedy methodology to the problem of distributed consensus to solve a version of what we call the *Rational Byzantine Generals* problem. We design an algorithm that is faithful under the *1-partial ex post Nash* equilibrium solution concept. We implement the algorithm over a network and compare the fault tolerance, message complexity, and scalability to MDPOP [PFP06] and a Byzantine-fault-tolerant version of the Paxos [Lam01] protocol. Our analysis is not intended to show the "best" algorithm; such an interpretation is not appropriate because of the unbridgeable difference in fault models assumed by each algorithm. Rather, these comparisons convey the relative trade-offs facing the system designer in supporting different fault models and algorithms.

1.3 Thesis Outline

The remainder of this thesis is organized as follows:

- **Chapter 2** introduces the *Rational Byzantine Generals* problem to help demonstrate the multiple challenges associated with rationally motivated failure. The Rational Byzantine Generals problem helps illustrate the problem of *information revelation failure.* The chapter then examines how previous research in computer science approaches this problem and how previous research provides ideas or tools that address rationally motivated failure. The chapter closes by distinguishing avoidable *rational compatible* failure from other types of failures that cannot be avoided.

- **Chapter 3** formalizes the notion of *rationally motivated failure.* The chapter introduces important concepts used later in the thesis, such as *utility*, *type*, *actions*, and *strategy.* The chapter then introduces the *mechanism* as the remedy for rationally motivated failure, comparing the centralized approach to mechanism design with the distributed approach used in this thesis. The chapter then introduces the mechanism *equilibrium* that holds in a given *solution concept.*

- **Chapter 4** has two parts: first, the chapter defines the methodology used to address rationally motivated failure. The second part of the chapter explicates the process that a designer uses to show specification *faithfulness.* Faithfulness is as important as the other systems correctness properties of *safety* and *liveness* in settings that contain rational nodes. A proof of specification faithfulness is a certification that rational nodes will choose to follow the algorithm specified by the system designer.

- **Chapter 5** applies the rationally motivated failure remedy methodology to an inter-domain routing problem based on work by Feigenbaum et al. [FPSS02] (FPSS). The enhanced algorithm is shown to be faithful under the *ex post Nash* solution concept. We implement the algorithm in simulation and compare the message complexity of the resulting algorithm to FPSS.

- **Chapter 6** applies the rationally motivated failure remedy methodology to the problem of distributed consensus. The chapter combines ideas from the MDPOP [PFP06] and Byzantine Generals [LF82] algorithms to form a new algorithm that is faithful under the *1-partial ex post Nash* equilibrium solution concept. The algorithm is robust to both rational and traditional failures. We implement the algorithm in a network and compare the fault tolerance, message complexity, and scalability of the new algorithm with MDPOP and a Byzantine-fault-tolerant version of the Paxos [Lam01] protocol.

- **Chapter 7** concludes and suggests future work.

Cross-Field Connection: This thesis necessarily uses concepts found in traditional *distributed systems* and in *mechanism design* analysis. We will use these "Cross-Field Connection" boxes to highlight overlapping ideas and language found in the two fields.

Chapter 2

Approaches to Rationally Motivated Failure

This chapter is divided into three parts: First, we introduce the *Rational Byzantine Generals* problem that we use throughout the thesis to study rationally motivated failure. Second, we describe related work on rationally motivated failure and explore how other research would approach problems with characteristics similar to the Rational Byzantine Generals problem. Third, we distinguish avoidable *rational compatible* failure from other types of failures that cannot be avoided.

2.1 A New Problem: The Rational Byzantine Generals

We open this chapter by building a specific problem that succinctly demonstrates the types of rationally motivated failures that this thesis must address. Our Rational Byzantine Generals problem is based on Lamport et al.'s Byzantine Generals [LSP82] problem, which is presented as follows:

> We imagine that [three] divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement. The generals must have an algorithm to guarantee that: (A) All loyal generals decide upon the same plan of action, and (B) A small number of traitors cannot cause the loyal generals to adopt a bad plan.

The generals agree to limit their communication to be their single-word vote on whether to "Attack" or to "Retreat". The generals agree to use Lamport et al.'s agreement algorithm with oral messages to establish a consistent view of battle plan votes across correct generals. In this algorithm, each general first directly sends its vote to all other generals and then relays what it hears from each general to all other generals. The generals agree to follow the majority-vote as their consensus *outcome function.* This outcome function means that the battle plan receiving the majority of votes is chosen by the generals. By agreeing on majority-vote, the generals meet Lamport et al.'s requirement that the same robust outcome function be used by all generals.

Among other results, Lamport et al.'s work shows upper bounds on the number of traitorous generals ($\lfloor \frac{n-1}{3} \rfloor$ when using unsigned messages) that can be present in an agreement algorithm while still being able to meet the correctness conditions. Having studied computer science while at military college, the generals know that their agreement algorithm cannot tolerate any failures because only three generals are participating in the algorithm. However, the generals trust each other not to fail.

We now add one seemingly innocent (but important) twist to this story: The generals' monarch, their queen, specifies to the three generals that after observing the city each general should vote to "Attack" if and only if they predict the city will be razed in a combined attack, and to vote "Retreat" otherwise.

Remark 2.1. *In Lamport et al.'s original problem, there is no instruction on how a general is supposed to reveal his vote. A general is correct in revealing his vote as long as it is consistent in unicasting the same vote to all other generals; his public vote declaration is not tied to his private prediction of success. The content of the vote is not a correctness concern to Lamport. It is, however, a correctness concern in the queen's problem.*

This setup describes a consensus problem where the goal is to select a *choice* from two candidates, "Attack" or "Retreat". The algorithm solution should satisfy the traditional correctness conditions of *liveness* and *safety.*[1] For traditional consensus the liveness and safety conditions are:

- **Liveness Conditions**

  L1. Some proposed choice is eventually selected.

  L2. Once a choice is selected, all non-failing participants eventually learn a choice.

- **Safety Conditions**

  S1. Only a single choice is selected.

[1]An introduction to specifications and their properties can be found in Weihl [Wei92].

S2. The selected choice must have been proposed by a participant.

S3. Only a selected choice may be learned by a non-failing participant.

Or, translated into the language of warriors:

L1. The generals eventually pick a choice from the proposed battle plans.

L2. Once a choice is made, all non-faulty generals eventually learn a choice.

S1. There is exactly one selected battle plan.

S2. The final selection of battle plan was proposed by at least one of the generals.

S3. Only the selected battle plan can be the choice learned by non-faulty generals.

The queen's last instruction, combined with the majority outcome rule, adds two additional safety conditions:

S4. "Attack" is selected only if the majority of non-faulty generals predicts that the city will be razed in a combined attack.

S5. "Retreat" is selected only if the majority of non-faulty generals do not predict that the city will be razed in a combined attack.

To continue our story, the three generals approach the enemy city via different routes and observe the enemy as the queen had instructed. Here is what the three generals privately think:

- **General 1** predicts that the city will be razed in a combined attack.
- **General 2** predicts that the city will not be razed in a combined attack.
- **General 3** predicts that the city will be razed in a combined attack, but that his beloved horse will surely be killed in the onslaught.

If General 3 values the life on his horse more than his loyalty to the queen, one of the following two scenarios may occur:

Scenario 1: A general lies about his prediction

Generals 1 and 2 follow orders and vote "Attack" and "Retreat" respectively. General 3 predicts that the city will be razed in a combined attack and should therefore vote "Attack". However, his love for his horse compels him to vote "Retreat". Generals 1, 2, and 3 otherwise execute the consensus algorithm correctly. The three generals select "Retreat". This is a violation of safety condition [S5].

Scenario 2: A general lies about what he hears

Generals 1 and 2 follow orders and vote "Attack" and "Retreat" respectively. General 3 predicts that the city will be razed in a combined attack and should therefore vote "Attack". General 3 knows that by telling the truth about his own prediction but incorrectly reporting what he hears from other generals, he can immediately violate the $\lfloor \frac{n-1}{3} \rfloor$ upper bound on faulty participants allowed by Lamport et al.'s agreement algorithm. General 3 fails by telling General 1 that General 2 voted "Attack", and by telling General 2 that General 1 voted "Retreat".[2] Both of the remaining non-faulty generals can detect that a failure has occurred, but neither knows which general has failed. The generals are unable to reach consensus. This is a violation of liveness condition [L1].

Analysis of Scenarios 1 and 2

These scenarios both highlight General 3's *rationally motivated failures* when General 3 acted in his own self-interest and not according to the system specification.

- The first scenario is interesting because had the queen not imposed safety condition [S5], the behavior of General 3 would have been correct. In Chapter 1, we briefly introduced *private information* as information not known to other algorithm participants and that cannot be established or verified by other algorithm participants. The Byzantine Generals problem and Byzantine Fault Tolerant (BFT) algorithms originally considered by Lamport et al. do not consider the correct *private information revelation* to be part of the algorithm specification. Furthermore, in the original problem, there is no queen to define what "correct" revelation even means. Yet, specifications can dictate such private information revelation actions and self-interested behavior can lead to "errors" in this information revelation.

- The second scenario is interesting because it outwardly appears to be a case of traditional failure, and would have been treated as a Byzantine fault in Lamport et al.'s original problem formulation. General 3 intentionally chose to fail *not* in information revelation, but in message passing. General 3's incentives were poorly aligned with the algorithm design. This misalignment of incentives led to an intentional failure, which led to the collapse of consensus. General 3 might not have chosen to fail had a different algorithm been used that, for instance, recognized and compensated the general for the loss of the horse.

[2]Lamport et al. [LSP82] pictorially demonstrate this example in which a failed consensus is detected by a non-failed participant, but not in a way that establishes who failed.

The Rational Byzantine Generals problem is a boiled-down "challenge problem" that we solve in this thesis. Our task is to present a system design that all generals will choose to follow, even if they have differing incentives. Borrowing the hardest aspects of the examples in the previous chapter, this problem has the following three characteristics:

- The problem demands that participants provide private information, which is information not known to other algorithm participants and that cannot be established or verified by other algorithm participants.
- The problem occurs in a setting where traditional system failures may occur in addition to any added rationally motivated failure. Translating the Generals problem back into computing, hardware may fail or bugs may arise, and a robust protocol should be able to deal with these faults.
- The problem is distributed. In problems we wish to solve, participants can be responsible both for simple message-passing tasks and for performing portions of a distributed computation.

We will return to the Generals throughout this thesis, proposing a protocol to solve their problem in Chapter 6. For now, though, we return to the world of computing. How might a system designer approach problems with these characteristics? What tools and research can one use to address rationally motivated failure? We now examine related work that helps us in answering these questions.

2.2 Relevant Prior Work

This thesis is a bridge between traditional research in fault tolerant distributed systems and algorithmic mechanism design, and there are several ways that we could categorize and discuss related work. We have chosen to present related work as follows: In this section, we examine specific research projects that are particularly relevant to this thesis. For example, we are aware of only one paper that provably addresses rational manipulation in distributed settings with private information, while also surviving traditional system failure.

Later in the chapter, we discuss additional related work on rationally motivated failure arranged by research area. We observe that work in historically separated research areas tends have similar strengths and weaknesses vis a vis each's ability to address problems with the three Rational Byzantine Generals characteristics, and we begin each section with those observations. Additional related work will be discussed in the following chapters' bibliographic notes.

2.2.1 Autonomous Nodes

The closest work to the types of problems considered in this thesis is Mitchell and Teague's work on *autonomous nodes* in distributed mechanisms [MT02]. Mitchell and Teague use the phrase "autonomous node" to designate a participant in a distributed algorithm that has full control over the hardware and software that it uses to interact with the rest of the distributed algorithm. In their words, these autonomous nodes are "strategic agents [that] control the computation at each local node that implements part of a distributed algorithmic mechanism."

The Mitchell and Teague work uses the example of the marginal cost mechanism for multicast cost sharing described by Feigenbaum et al. [FKSS01]. Mitchell and Teague start with a protocol that is susceptible to rational manipulation and propose two protocol variants to addresses this susceptibility. Their first protocol, like our work in Chapter 5, assumes a mixture of rational and obedient participants. Their second protocol, like our work in Chapter 6, assumes a mixture of rational, obedient, and malicious participants. Both of the Mitchell and Teague protocols use cryptographic signing with a central auditing mechanism utilizing a bank to punish deviant behavior. These techniques are relevant to designers who wish to build systems that are tolerant to rationally motivated failure.

The Mitchell and Teague work is more theoretically oriented than this thesis as it does not describe how to build an operational system. One major scalability hurdle is the work's reliance on an obedient central participant to act as a central checker mechanism that audits every other agent. (In comparison, this thesis pushes distributed checking back into the network of rational participants.) It is not obvious how one would incorporate distributed checking functionality into their solution so that a rational participant would still choose to participate correctly in the protocol. Nevertheless, Mitchell and Teague's work provides an important example of a system that aims to prevent rationally motivated failure in a problem that involves information revelation, algorithmic computation, and message passing.

2.2.2 BAR Fault Tolerance

In earlier work on *faithfulness* we asked how one would build systems that tolerate both rationally motivated and non-rational failure [SPM04, SP04]. One proposal can be found in Aiyer et al.'s work on BAR fault tolerance [AAC$^+$05]. Aiyer et al. present a state machine architecture based on Castro and Liskov's Practical Byzantine Fault Tolerant state machine [CL99] for building applications that run within networks of rational, altruistic (obedient), and non-rational but faulty nodes. Aiyer et al. demonstrate their work to power a peer-to-peer file backup system, similar to the system described in Castro and Liskov [CL99]. Their tiered model initially seems quite powerful since it includes a generic state machine that is meant to run a wide range of cooperative services. However, three important limitations of the BAR work prevent one from applying the BAR framework to problems like the Rational Byzantine Generals problem:

First, the BAR work provides incentives only for correct behavior in redundant computation and message passing actions. BAR does not address the *information revelation* manipulations that we described earlier in the Rational Byzantine Generals problem.[3] The BAR work prevents failures by making failure in computation and message passing actions an irrational choice for a rational participant. However, these failures would have been caught and corrected anyway by Castro and Liskov's underlying Practical Byzantine Fault Tolerant algorithm.

Second, the BAR work is able to provide incentives to rational participants for correct behavior only when rational participants assume that Byzantine failures occur in a worst-case manner.[4] This assumption breaks down in practice if a rational participant observes non-worst-case failures where the rational participant's own deviation is profitable.

Third, BAR assumes that rational nodes participate correctly in penance and penalty mechanisms. This is a major assumption, since these mechanisms are critical to proving an equilibrium, and compliance in these protocols by rational nodes is not guaranteed. In the example and language of their state machine protocol, non-leader nodes must be willing to pay the cost of checking and acting on received malformed penance messages. Their model dictates that rational participants minimize costs, and costs are defined as "computation cycles, storage, network bandwidth, overhead associated with sending and receiving messages, [etc.]." This means that participants may ignore any extra penalty mechanism work since this increases a participant's cost. Worse yet, there can actually be an incentive for one participant to ignore another's deviant behavior.[5]

[3]To readers familiar with the BAR work but confused by this claim, consider this example in the language of the BAR work: BAR allows a rotating leader to propose commands to be executed by a replicated state machine. While BAR prevents other rational nodes from changing this command (by using terminating reliable broadcast instead of consensus), nothing stops a rational leader from selfishly choosing a particular command to execute.

[4]More precisely, using concepts to be defined in Chapter 3: The BAR work establishes a rational participant's equilibrium behavior only if it is assumed that Byzantine nodes fail in a manner that maximally damages a rational participant's expected utility.

[5]For readers familiar with the BAR work, an example is that reported deviating nodes are kicked out of the BAR file backup system. But since there is a cost of replicating data, a reporting node may tolerate a

These three limitations stand in the way of a designer wishing to use the BAR state machine system as a basis for a rationally motivated fault tolerant system. The work that we describe in Chapter 6, while not as general as a state machine, does address these limitations.

2.3 Related Research: Distributed System Fault Tolerance

We now discuss related work by research area, starting with traditional work in distributed system fault tolerance. This area of research studies how to build systems that are robust to different types of failures in computation and message passing. Research in this area historically does not address private information revelation.

There are two important characteristics of conventional approaches to distributed system fault tolerance. The first characteristic is that conventional approaches are built around *fault expression*, rather than *fault cause.* Designers may classify a node as correct or faulty without giving thought as to why the node is correct or faulty. The second characteristic is that conventional approaches are *reactive* rather than *preventative.* The usual goal in building fault-tolerant systems is to identify faulty nodes and recover from failures they introduce, typically by excluding faulty nodes from future participation.

Failure Expression

Table 2.1 shows a distributed systems failure model taxonomy collected by Schneider [Sch93] (original citations omitted). We note that this is a taxonomy of failure expression, i.e., this is a listing of how faults are exhibited. Schneider argues that a failure expression deserves an entry in this failure taxonomy if the feasibility (what classes of problems can be solved?) and cost (how complex must the solution be?) of addressing the failure are distinct from other members on the taxonomy. For example, we reflect that the list omits *insertion failures*, defined as "A processor fails by transmitting a message not dictated by a specification," presumably because the feasibility and cost of treating the failure was thought by Schneider to be equivalent to one of the other failure types. A failure can always be labeled as Byzantine in that Byzantine failure subsumes any other failure that appears on the list, or that we might add to this list. For this reason, any observed distributed systems failure can be classified by this taxonomy. For now, we observe that the information revelation failures that we saw earlier in the Byzantine Rational Generals problem might be classified as a Byzantine failure under Schneider's original taxonomy.

mildly deviant node when the deviant node also acts as the reporting node's storage location.

Failstop	A processor fails by halting. Once it halts, the processor remains in that state. The fact that a processor has failed is detectable by other processors.
Crash	A processor fails by halting. Once it halts, the processor remains in that state. The fact that a processor has failed may not be detectable by other processors.
Receive Omission	A processor fails by receiving only a subset of the messages that have been sent to it or by halting and remaining halted.
Send Omission	A processor fails by transmitting only a subset of the messages that it actually attempts to send or by halting and remaining halted.
General Omission	A processor fails by receiving only a subset of the messages that have been sent to it, by transmitting only a subset of the messages that it actually attempts to send, and/or by halting and remaining halted.
Byzantine Failure	A processor fails by exhibiting arbitrary behavior.

Table 2.1: Failure expression taxonomy due to Schneider [Sch93] (original citations omitted).

Failure Cause

One can use the taxonomy in Table 2.1 to answer the question, "How did the participant fail?" The taxonomy cannot be used to answer the question, "Why did the participant fail?" To answer the "why" question, computer scientists must examine the failure cause. There are two categories of failure cause, defined as failure due to:

- **unintentional** erroneous behavior. A failure expression is due to chance, nature (cosmic rays), bugs in a specification, bugs in an implementation, etc. A failure in this class is one where the failing participant, if given the choice, would either choose to avoid the failure or has no concept of "choosing" failure vs. no failure.
- **intentional** behavior. A failure expression is the result of a participant choosing to fail.

Computer scientists sometimes define *intentional failure* to be a synonym for *malicious failure.* For example, the security research literature is particularly concerned with this type of intentional failure. Designers are accustomed to defending their systems against a "worst-case" adversary who is imagined to exhibit a particularly heinous failure pattern [Sch95, Nee93]. The modus operandi for dealing with Byzantine failure, regardless of cause, is to build a Byzantine fault tolerant algorithm [LF82]. These algorithms typically rely on redundancy and/or cryptography and provide hard guarantees about algorithm correctness when the number of faults is bounded [CL99].

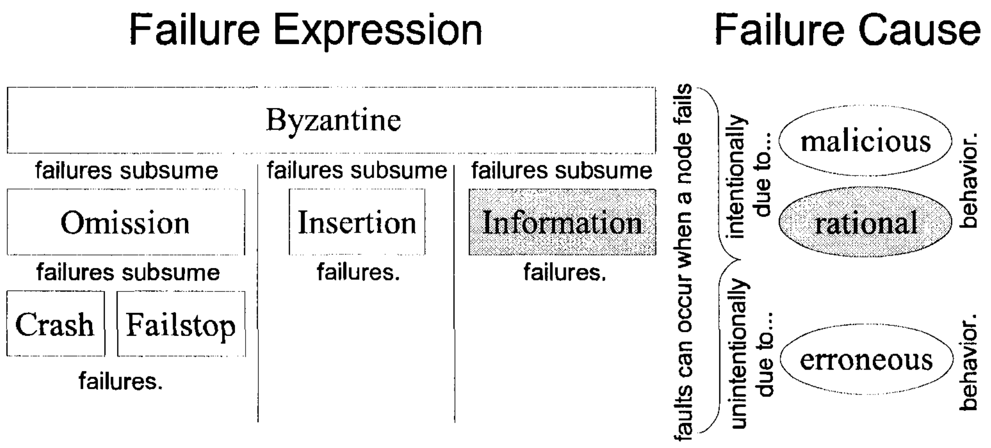


Figure 2.1: Failure expression and failure's underlying cause. Information failure and faults due to rational behavior are not typically studied in distributed systems but are the subject of this thesis.

Rational = Byzantine?

Are existing Byzantine fault tolerant methods sufficient to address rationally motivated failure? Byzantine failure equates to *arbitrary* failure, where *arbitrary* is interpreted to mean *any*. By casting such a wide net, we might expect that practical tools and techniques for building Byzantine fault tolerant (BFT) distributed systems, such as those proposed by Castro and Liskov [CL99], are the most appropriate ways to deal with rationally motivated failure. However, this is not the case. Even in building Byzantine fault tolerant algorithms, there are advantages in specifically addressing rationally motivated failure:

- **The set of Byzantine fault tolerant algorithms can be expanded.** Traditional BFT techniques are reactive and rely on redundancy, cryptography, and verification techniques to tolerate failure. However, none of these tools can be used to address *information revelation failures* that we demonstrate in the Byzantine Rational Generals problem. This means that existing traditional so-called Byzantine Fault Tolerant algorithms are not tolerant to information revelation failure. There is no contradiction here, since all of these BFT algorithms either do not direct participants to reveal private information, or if private information is elicited (as in the original Lamport et al. Byzantine Generals problem), then there is no guidance as to the actual content of the revealed information.

- **The robustness of Byzantine fault tolerant algorithms can be increased.** Claims of Byzantine fault tolerance are bounded by the number of simultaneous failure expressions. The techniques introduced in this thesis can reduce the number of simultaneous faults that occur due to rational behavior. The result is that the BFT algorithm has more "wiggle-room" to tolerate faults from other causes.

- **The cost of Byzantine fault tolerant algorithms can be decreased.** Reducing the number of failures can also make the actual execution of a BFT algorithm more efficient. Several BFT algorithms treat correct behavior as the common case, running faster and with less overhead when there are fewer faults (e.g., Kursawe's Optimistic Byzantine Agreement [Kur02] and FaB Paxos [MA05]).

Example Application: Seti@Home Errors

When a protocol makes no information revelation demands on participants, conventional BFT approaches *can* be used to address rationally motivated failure, albeit at some increased cost. The Seti@Home "fake work packets" problem described in Chapter 1 is an example of a computation manipulation. To remind ourselves of that example, Seti@Home is a distributed computation system where participants run clients that download raw data from a server, perform computation, and upload finished "work units" to the server. There is a scoring system that rewards users with (otherwise meaningless) points tied to their level of work submission. Some participants used a hacked client to upload junk, labeled as finished work, to obtain a high score on the "packets submitted" leader-board. [Kah01]

In this system, any user's work packet can be verified by any other participant in the system. To address the problem of a participant falsely claiming completed work to drive up their leader-board score, the Seti@Home team changed their software to replicate work packets across multiple clients. The revised client software verifies work with other participants and updates a central leader board only when these packets agree. Three out of four participants must now agree before credit is awarded.

This approach had several drawbacks: the throughput of the system was reduced by more than 75% as participants were forced to triple-check each others' results [Use05]. Moreover, in the revised system participants keep track of the number of successfully verified work packets, which equates to their submitted leader-board score. As of this writing, one can download newly hacked clients that submit inflated scores [Kah01]. These client manipulations have anecdotally reduced the incentive for some honest participants to participate [Use07].

We see several lessons in this example. One lesson is that in a system with no private information, it is *possible* to rely on traditional Byzantine Fault Tolerance techniques such as replication to address a rationally motivated failure. However, it may not be *optimal*: the robustness of Byzantine fault tolerant algorithms can be increased and the computational cost of some Byzantine fault tolerant algorithms can be decreased by specifically addressing rationally motivated failure. In the Seti@Home case, a flawed leader-board implementation created a perverse incentive to damage the overall system. An alternate solution that simply removed the leader-board might actually increase throughput, assuming that leader-board manipulations are the sole cause of bad packets in the system.

2.4 Related Research: Failure in Mechanism Design

Mechanism design (MD)[6] is an area of economics that studies how to build systems that exhibit "good" behavior (for some designer-defined notion of "good") when self-interested nodes pursue self-interested strategies in *decision problems*: within systems, decision problems include leader election [Lam98], scheduling [LL73], and bandwidth or CPU resource allocation [SNP$^+$05].

MD research focuses on eliciting truthful information from algorithm participants. Work in mechanism design historically has not studied the implementation problem of turning a mechanism into a protocol, let alone any distribution of a mechanism, or operation in the face of traditional systems failure. These areas have started to receive attention as computer scientists have become more interested in rational behavior. As part of his dissertation, Parkes [Par01] reviews research from both economics and computer science that considers how bounded computation, communication, and a participant's ability to calculate a rational strategy can affect mechanism design. This research is related to work on *algorithmic mechanism design*, which studies how to build mechanisms that retain useful game-theoretic and computational properties.

Algorithmic mechanism design (AMD) [NR99, NR00] places a special emphasis on a mechanism's computational tractability. As in MD work, AMD research typically assumes a obedient and failure-free networking infrastructure. AMD focuses on the study of centralized mechanisms, where participants report their complete private information to a special trusted participant (a *center*) that runs a decision algorithm. In concluding their seminal work on AMD, Nisan and Ronen noted the "set of problems" that come in implementing a decision problem with rational nodes in a real network [NR99], suggesting that distributed computation and message encryption could be useful tools, but leaving

[6]Concepts and terminology from mechanism design that are useful for our thesis are described in Chapters 3 and 4. The reader can find alternate introductions to mechanism design in Jackson [Jac00] or Parkes [Par01].

such problems open to future work.

Contemporary with early AMD developments, Monderer and Tennenholtz [MT99] present the earliest work dealing with truthful private information revelation *and* rationally motivated failure in message passing. This paper challenges the MD/AMD assumption of obedient network communication. Monderer and Tennenholtz recognize that messages from a participant to a decision-making center may be carried by other rational participants who have a vested interest in the decision. These rational participants may inspect, drop, or change messages that they are supposed to route to the single center. Monderer and Tennenholtz present a simple encryption scheme that makes it weakly harmful for participants to drop or change such messages, assuming certain restrictive beliefs are shared among the participants.

Distributed algorithmic mechanism design (DAMD) [FPSS02, FS02] considers the MD/AMD problem in a network setting with no single center, where computation is distributed across the self-interested participants who also control the message paths. The extension from AMD to DAMD was first explored by Feigenbaum et al. [FKSS01, FPSS02, FPSS02].[7]

Feigenbaum, Papadimitriou, Shenker, and Sami [FPSS02] (FPSS) proposed a routing protocol to prevent rational participants from lying about their transit costs in the problem to find *efficient*, i.e., lowest cost, routes between nodes. Their work proposes a payment scheme that can be used to pay *transit nodes* for relaying traffic along lowest cost paths. The protocol incorporates a Vickrey-Clarke-Groves mechanism [Vic61, Cla71, Gro73] to guarantee truthful private cost declaration from rational participants. Their resulting protocol makes certain assumptions, including the impractical requirement that participants otherwise follow the distributed algorithm without deviation. We save further discussion of this work for Chapter 5 where it serves as the basis for a new protocol that addresses these limitations.

Feigenbaum, Ramachandran and Schapira [FRS06] recently re-visited the interdomain routing problem [FPSS02] to propose a solution to a modified problem based on "next-hop" policy routing. Instead of ensuring that packets are routed along lowest-cost paths, each participant decides among available routes solely based on the routes' next hops. With this change, the authors show that their algorithm is immune to the types of rational manipulation that were not addressed in FPSS.

[7]Feigenbaum et al. [FSS07] observe that a number of early networking papers that invoke game theory are related to DAMD. This makes sense, as networking problems are inherently distributed and must have reasonable complexity. More importantly, networking is a natural area in computer science where limited resources are under contention from autonomous self-interested participants. Even so, the minority of game-theoretic networking papers concern distributed decision problems, and even fewer address failure.

2.5 Related Research: Failure in Networking

Networking researchers have long been interested in designing "fair" systems that work with rational participants and in evaluating existing systems in the presence of selfish behavior. Most of the work in game-theoretic networking focuses on optimizing systems (e.g., Gibbens and Kelly's research on resource pricing [GK99]) or analyzing performance (e.g., Roughgarden and Tardos's work on the cost of selfish routing [RET02]) and is not directly related to this thesis. We *are* interested in work that addresses failure. Within networking, these failures equate to avoiding the responsibility to route or forward others' packets or finding creative ways to make one's own packets cheaper or faster.

2.5.1 Game-Theoretic Reasoning

Feldman et al. [FCSS05] examine the problem of "hidden-action" in packet forwarding.[8] They observe that endpoints in a network rely on intermediate nodes for packet forwarding, and that rational intermediate nodes may choose to disrupt the packet flow. The authors construct a model of the problem to show how incentive-laden all-or-nothing contract offers can be used to overcome the hidden-action problem. Feldman et al. apply their work to packet forwarding in the FPSS [FPSS02] interdomain routing protocol to demonstrate correct packet forwarding as a Nash equilibrium in the absence of external monitoring. They also show that correct packet forwarding is a dominant strategy equilibrium in the presence of monitoring. Both of these proofs – in the presence and absence of external monitoring – require the strong assumption that participants otherwise follow the algorithm, which is an assumption that we cannot make in this thesis.

Zhong et al. [ZCY03] propose Sprite, a virtual currency-based system made for mobile/ad-hoc networks to stimulate cooperation in message forwarding tasks among selfish participants. Unlike previous reputation-based approaches (e.g., routing research by Marti et al. [MGLB00], or the CONFIDANT protocol by Buchegger and Boudec [BB02]) or solutions that use secure communications hardware (e.g., the Terminodes/Nuglets work by Buttyan and Hubaux [BH03]), Sprite uses a centralized trusted node that acts as an accountant and behavior enforcer. Zhong et al. define a game that includes the actions of forwarding messages and reporting received messages, and show that in their game, an optimal participant strategy is to play the game truthfully. The authors implement their system and provide some evaluation to document the overhead of their approach.

Mahajan et al. [MRWZ04] report on challenges encountered in directly applying

[8]Using the language of Chapter 4 and defined in our earlier work [SP04], to solve a hidden-action problem is to show that a system is *communication compatible*.

results from traditional game theory to problems in multi-hop routing and packet forwarding: They report on the asymmetry problem that arises when using a barter system or equilibrium forwarding argument, where nodes at the edge of a network cannot provide the same packet forwarding or routing value that they demand. They also consider problems with virtual currency, such as balancing and control and the larger problem that a closed virtual currency cannot express real-world costs and benefits.[9]

The authors also bring up concerns about the types of mechanisms used in Feigenbaum et al. [FS02, FPSS02] and later in this thesis. Namely, Mahajan et al. point out that publication of private information can be a competitive concern for future interactions when information is used later to hurt the revealing participant. For example, by revealing true costs for routing packets, a participant in a routing algorithm may be revealing cost-of-entry information to competitors. Finally, Mahajan et al. pose a set of theory-meets-systems questions concerning the (im)possibility of global identities, the instability of systems affecting game theoretic results, and the costs of implementing the system (run-time overhead, required infrastructure, etc.) vs. the potential benefits. While these are all valid concerns, the authors' primary disappointment is that game theory offered few off-the-shelf silver bullet for their particular routing problems. We believe, however, that elements of game theory can be applied in ways to further the design of robust distributed systems, and this thesis is predicated on that belief.

2.5.2 Overlay and Peer-to-Peer Networks

Some of the most interesting attempts to address rationally motivated failure have been developed in successive generations of overlay and peer-to-peer networking programs. A large class of this work deals with free-riding *participation problems*. A lack of participation is an *individual rationality failure*, which is the economics term that describes a user's unwillingness to participate in a mechanism based on its expected reward. The failure is similar to the failstop/crash studied in traditional systems, except that the failure may occur even before a system starts or the participant set is known since potential users simply choose not to participate in an algorithm. In relation to this thesis, we look to overlay and peer-to-peer networking work for ideas on how to ensure correct participant behavior even when the initial system design seems to discourage correct participation by rational participants. Researchers have proposed probabilistic audits (either by trusted third parties [NWD03] or by algorithm participants [CN03]), forced symmetric storage relation-

[9]These same problems seem to plague every virtual currency system that we have examined, from Spawn [WHH$^+$92], Mariposa [SAL$^+$96], Mojonation [McC01], to more recent work like Sprite [ZCY03], Bellagio [ACSV04], and Tycoon [LRA$^+$05].

ships [ABC$^+$02, CMN02], or token payment [McC01, IIKP02] to encourage participation.

Most of these attempts include simple game theoretic ideas: Kazaa [Net07] encouraged global tit-for-tat relationships with its participation metric. MojoNation [McC01] provided a virtual economy to encourage participation; one gained currency by sharing files, disk space, and bandwidth. Unfortunately, many users found that the currency was not expressive or well-managed. The incentives were not effective in encouraging continued participation as many users reportedly did not understand how to value the artificial currency [McC01]. BitTorrent [Coh03] simplified the tit-for-tat into a trading game with centralized trusted control (via the tracker component), wherein participants receive pieces of file faster when they send out other correct pieces of the same file. While its creator has argued that BitTorrent's incentive mechanism is responsible for its success [Coh03], researchers have shown manipulation opportunities in the protocol [LMSW06, Tor07, PIA$^+$07, LNKZ06] and have argued that the protocol's built-in altruism is the reason for its success, and that its incentives actually encourage manipulation [JA05].

Peer-to-peer research has promoted the threat of the *Sybil attack*, which occurs when a system relies on identity but has no trusted identity verification mechanism.[10] Douceur [Dou02] first coined this term to describe a participant who spoofs multiple identities in order to extract better service from a distributed algorithm. In peer-to-peer systems, the basic Sybil problem is often related to bootstrapping: a new participant may have nothing to offer the system but must have an incentive to join the system. Participants can create many identities that *free ride* any initial incentive. Douceur poses some conjectures about the impossibility of fully overcoming Sybil attacks while Cheng and Friedman [CF05] attempt to formalize the ideal property of "sybilproofness."

An interesting approach to encourage participation and counter Sybil attacks in peer-to-peer networks is to introduce a *distributed reputation system.* A reputation system provides a participation score for each participant based on each's dealings with other parties. Reputation systems do not remove underlying opportunities for manipulation, but the goal is to make it difficult for a "bad" participant to receive good service. Representative research includes Resnick et al. [RKZF00], who describe specific manipulations that can occur in reputation systems. Kamvar et al. [KSGM03] present EigenTrust, which uses notions of transitive trust to reduce the effect of false reputation claims from untrusted participants. Friedman et al. [FRS07] study manipulation-resistant reputation systems and present a system that induces honest reporting of feedback in certain settings, and additionally study a

[10]Problems of identity verification have confounded researchers for at least 1900 years before being considered by computer systems designers; one ancient example is the Ship of Theseus paradox [Plu75] found in the work of the Greek philosopher Plutarch.

number of practical implementation issues. Cheng and Friedman [CF05] show under what conditions reputation systems may be expected to recover from Sybil attacks.

Computational game theorists have also turned their attention to participation problems in peer to peer systems. The models used in these works tend to be too abstract to be used directly by systems practitioners. Systems designers should look this work as a source of interesting ideas for combating participation failure, rather than an off-the-shelf solution. One reason for caution is that these works tend to prove Nash equilibria, which is probably too strong of a knowledge assumption as we show in Chapter 4. Representative papers include:

Feldman et al. [FLSC04] study free riding in peer-to-peer systems with collusion, zero-cost identities, and traitors. They model such systems as a prisoner's dilemma problem [FT91] and propose a simple decision function that serves as the basis for a tit-for-tat incentive scheme. The authors show through simulation that their techniques can drive a system of strategic users to nearly optimal levels of cooperation.

Friedman et al. [FHK06] and Kash et al. [KFH07] study systems that use virtual currency (scrip) to reduce free riding by encouraging good Nash equilibrium behavior. With respect to failure, they examine the effect of a "monetary crash," where virtual currency is sufficiently devalued so that no agent is willing to perform a service. They further examine the effects of altruists and money hoarders on the performance of the system, and how these behaviors affect monetary crashes.

Aperjis and Johari [AJ06] solve participation failure by formulating peer-to-peer file sharing as a barter economy. In their work, a price is associated with each file and participants exchange files only when each user can provide a file that is desired by the other party. Their formulation solves the free-riding problem, since uploading files is a necessary condition for being able to download. They characterize the Nash equilibria that occurs in this game and show that the Nash equilibrium rates become approximately Pareto efficient as the number of users increases in the system.

TCP Hacking

Not all rationally motivated networking failures are overcome with game theory. The faults observed by Savage et al. [SCWA99] described in Chapter 1 occur because of a manipulation of *private information* in the system. To remind ourselves of the example, performance-hungry receivers in the TCP protocol can manipulate their partner's sending rate at the expense of other network users. A receiver provides an acknowledgment message (ACK) to the sender, and the receipt of this ACK determines the sender's outgoing data rate. Because a TCP sender always trusts the receiver's ACK, a manipulating receiver can force a sender to send data more quickly than is fair in a multi-user environment.

The private information in TCP is the packet acknowledgment message: while the sender can form an expectation of this private information (e.g., "If I didn't send a packet, you could not have ACK'd the packet"), the manipulation studied by Savage occurs in the post-transmission moment when the receiver is the only party that knows if its TCP stack has received and processed a message.

Savage et al. propose fixing TCP's design by removing the reliance on private information. In their modified TCP, receivers must echo back a random number provided by the sender in each packet. Effectively, this eliminates the dependence on private information in the algorithm. Without labeling the initial failure as a rationally motivated fault, Savage et al. effectively remove the opportunity for manipulation. This is a fine way to address rationally motivated failure: simply get rid of the opportunity for failure! However, this approach only goes so far: in some algorithms (such as the Rational Byzantine Generals) there is an inherent dependence on the private information of each general's opinion that cannot be removed without changing the problem.

2.6 Related Research: Security

A growing body of work studies secure private algorithms through the use of cryptographic secret sharing [Sch95]. Part of the motivation for secret sharing is to overcome partial failure: a participant splits and shares a numerical secret s to n other participants, so that any $m < n$ of them may reconstruct the secret. Even if $n - m$ participants are "bad" and do not cooperate, m "good" participants may cooperate to learn s. This work is similar to our thesis in that secret sharing requires private information revelation, but differs from our thesis in that secret sharing-based systems generally assume participants wish to minimize revealed private information but have an interest in the correct system outcome. In contrast, we are concerned with participants that reveal untruthful private information to steer the system outcome to an incorrect but selfishly beneficial outcome.

McGrew et al. [MPS03] generalize a non-cooperative computation framework by Shoham and Tennenholtz [ST03] where every participant has the mutually-exclusive goal of wishing to be the only participant to compute a function of participants' inputs, but with the caveats that its own input remains private and that the participant learns the inputs of other agents. Addressing the same problem, Halpern and Teague [HT04] present randomized mechanisms using secret sharing for rational participants that wish to compute a multiparty function. Participants' utility is directly linked to how much information they are able to learn about a secret relative to the amount of information discerned by other participants. Gordon and Katz [GK06] present an alternate technique for secret sharing among n rational players that removes the involvement of tbe third party dealer present in earlier work. All of these works rely on secret sharing for some degree of fault tolerance.

In the context of secret sharing and multiparty computation [GMW87], Abraham et al. [ADGH06] study joint strategies where no member of a coalition of size up to k can do better than an equilibrium strategy, even if the whole coalition defects from equilibrium play. The authors show that such *k-resilient Nash equilibria* exist for secret sharing and multiparty computation, provided that players prefer to learn information over not learning information.

Izmalkov et al. [IML05] address the problem of rational behavior in secure multiparty computation. Their goal is to create a rational secure computation protocol that is secure even when all players pursue selfish interests. They show that any mediated game with incomplete information can be rationally and securely simulated with a particular extensive form game known as a *ballot-box game*, assuming that communication channels can be restricted.[11] Lysyanskaya and Triandopoulos [LT06] take an alternate approach to solving the same problem. In contrast to Izmalkov et al., Lysyanskaya and Triandopoulos use standard communication channels, allow for the existence of covert channels and make assumptions that participants have an incentive to participate in the protocol and to reveal private information truthfully.

Secret sharing has recently been used to enable private auctions. Brandt and Sandholm [BS04] studied the problem of building distributed auctions so that bidder's valuations remain private, while still retaining the property that the auction awards an item to the highest bidder. Their distributed algorithms tolerate failure by using the recovery properties of secret sharing.

2.7 Reflections on Related Work: What is Rational?

Whether one is interpreting related work or designing a new system, it is important to clarify what one classifies as *rational behavior*. It is meaningless to claim that a system "addresses rational behavior" or "overcomes rationally motivated failure" without defining a set of behaviors as "rational" and the environment in which such a claim holds. In the following two chapters, we formalize rationally motivated failure and what it means to define

[11]The communication assumptions in Izmalkov et al. are similar to the assumptions we make in Chapter 6 to prevent collusion with exogenous payments via hidden channels.

the environment where such a claim can hold. In this last section, we distinguish failures that can be addressed as *rational compatible* failures.

2.7.1 Rational Compatible Behavior

In the introduction to this thesis, we wrote that rational behavior is entirely *predictable* assuming that one can correctly model rational participants. On one hand, this predictability is powerful: a designer simply "designs for the participants", making sure that participants' individual objectives are satisfied when they correctly follow the system specification. On the other hand, the task of modeling rational participants is equivalent to predicting the future — that is, to anticipate the demographics of users who will participate in a distributed system.

Moreover, even if one can predict future rational behavior, the behavior may be *incompatible* with the design goal. To use a real life example, it is well known that the Recording Industry Association of America (RIAA) opposes illegally shared music. If a system designer wants to build a file sharing system that facilitates on-line piracy and expects the RIAA to participate, it is not hard to predict the RIAA's rational behavior will be an attempt to destroy the system. This extreme example highlights the two types of rationally motivated failure behavior as seen by the system designer:

- **compatible** rational behavior can be addressed through incentives and the system design techniques discussed in this thesis, so that participants do not exhibit rationally motivated failure *and* the system operates as the designer intends.
- **incompatible** rational behavior is behavior that cannot be addressed without also sacrificing the intended system goals.

We illustrate this difference by considering the traditional distributed systems world view, as shown on the left-hand side of Figure 2.2. In the traditional view, failure expressions are attributed to either unintentional or malicious behavior. A more precise world view is shown on the right-hand side. This new breakdown allows for *rational incompatible behavior*, as well as the possibility of *irrational* behavior that cannot be explained by some view of rationality. Finally, there is the category of *rational compatible behavior.* This last category of rational behavior is the only failure cause that designers can hope to address. In theory, all failure expressions due to rational compatible behavior can be prevented. However, in systems practice, only a subset of these behaviors are addressed due to limitations in modeling, or in the ability or willingness of the system designer to accommodate a range of rational participants.

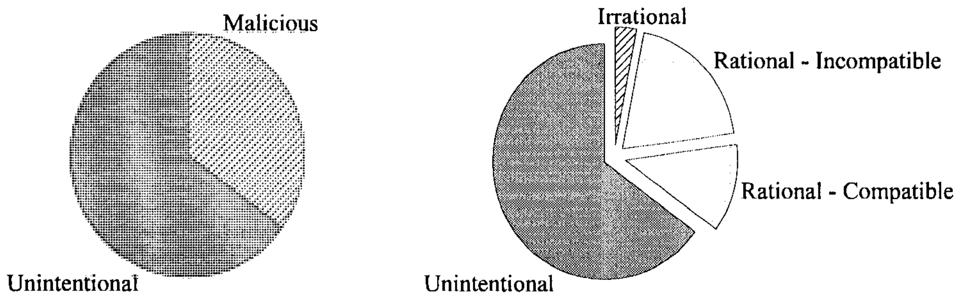


Figure 2.2: **The failure cause pie. On the left** is the old systems world view, where failure expressions are attributed to unintentional behavior or to malicious behavior (where *malicious* is viewed as synonymous to *intentional*). **On the right** is the world view proposed by this thesis. All failure expressions due to compatible rational behavior can be prevented. In practice, due to limitations in modeling and the will of the designer, only a subset of these motivated behaviors are addressed in a system. The relative sizes of the pie slices will depend on the system goals, design, and participants.

Remark 2.2. *We are extremely hesitant to ascribe failure to irrational behavior; in the real world, even most suicide bombers believe that they are acting rationally and so their behavior should be classified as rational incompatible behavior. However, we cannot preclude the possibility of a participant who fails intentionally, but who knowingly is not seeking to better his/her outcome in a distributed algorithm. While our definitions allow for this possibility, our feeling is that in practice most malicious behavior should be classified as rational incompatible behavior. We maintain the distinction only to observe that rational incompatible behavior could be addressed by sacrificing the intended system goals, whereas irrational behavior can never be addressed. The distinction does not matter from a system fault-response perspective as we address* irrational *and* rational incompatible *behavior with the same traditional (Byzantine) fault tolerance techniques.*

Remark 2.3. *For the remainder of this thesis, the ambiguous terms* rational *and* rational behavior *will strictly refer to rational compatible behavior. This thesis does not add insights on how to address rational incompatible behavior, irrational behavior, unintentional behavior or rational compatible behavior that is not addressed by the system designer — failures due to these causes are left to be treated with traditional Byzantine Fault Tolerance (BFT) techniques.*

2.7.2 Limitations of Rational Behavior

There are a few types of behavior that we do not study in this thesis, and (unless noted) are not studied by the aforementioned previous work. This thesis will treat these behaviors as rational incompatible behavior.

Collusion

Collusion is the coordination of two or more parties to act as one participant. The examples in this thesis will treat collusion between participants as a form of rational incompatible behavior. We note that the relative anonymity of the Internet, combined with partnering and communication restrictions such as what we build in Chapter 6, may help reduce collusion opportunities. Similarly, we do not contribute to the problem of addressing Sybil attacks [Dou02], where a single participant claims to be multiple participants that collude with each other. All of the examples presented in this paper assume that each participant is acting in its own independent self-interest.

Repeated Games

The examples in this thesis do not consider the effects of repeated games on participant behavior. A repeated game is a series of decision problems, in which a user can observe and learn from one decision problem to reason about strategy in a successive iteration of the decision problem. For example, Afergan [Afe03] studies repeated games in networks and has shown that how the equilibrium strategy presented in Feigenbaum et al. [FPSS02] does not hold in a repeated setting. This is discussed further in Chapter 5.

From the set of rational compatible behaviors, we further exclude participants who seek to damage other users as their primary goal. These participants may exist in a repeated game, where a participant may be willing to hurt itself in the short-term in order to drive a competitor away in the future. Brandt [BW01] labels these participants as *anti-social*, since their happiness with an outcome is based on how damaging the outcome is for other participants.

Chapter 3

Rationally Motivated Failure

Up until this chapter, we have been working with an informal definition of rational behavior as the subset of intentional behavior that occurs when a participant aims to better its outcome in a distributed algorithm. This chapter formalizes the notion of *rationally motivated failure*. We then move into the concepts needed to describe the remedy to such failure, such as the *mechanism*. The language and definitions introduced in this chapter will help us in the remainder of the thesis as we make claims of *faithfulness* as the specification property that guarantees that rational compatible participants will follow a correct implementation of the specification.

3.1 Preliminaries

A distributed system is composed of a set of *participants* that interact with the system through their computational proxies known as *nodes*.

Cross-Field Connection: *For those readers familiar with distributed systems:* We stress our break from the traditional view that a node is a self-contained device that is supposed to implement a central designer's specification. In our world, motivated participants (people) actively direct their nodes to act according to the participant's wishes.
For those readers familiar with mechanism design: The participant is the rational *agent* while the node assumes the duties of an agent's *proxy* in an indirect mechanism. The node may also implement part of a distributed mechanism.

In this thesis, we assume that the set of participants and nodes is known at the start of the system; this thesis is not concerned with *online mechanism design* where participants can enter and leave the system during normal execution. We also assume that the participant-node relationship is one-to-one, meaning that each participant controls exactly one node. If there are n participants in the system, there are n nodes in the system indexed $i = 1, ..., n$. These nodes communicate with each other via a network whose details will be defined for a particular problem instance. The nodes are modeled using *state machines* [Wei92]. A node's *state* captures all relevant information about its status in a system. It may include received messages, partial computations, private information about its participant, and derived or estimated knowledge about other nodes, participants, and the system. We will refer to the *system outcome* as the consistent snapshot of node states at some designer-specified *snapshot point*. We will denote a particular system outcome, chosen from the world of possible system outcomes as: $o_1 \in O$.

3.2 Utility and Type

Participants can have preferences over the range of system outcomes. These participants have a *type*, which is the smallest set of information that completely determines the participant's *utility function*. The utility function rationalizes preferences for different system outcomes. The *utility* of a particular outcome is defined as the relative rating of that outcome as calculated by the participant using its utility function.[1]

Example. A participant's type in an auction includes the participant's private estimate of its value for a particular item, which dictates the utility on the outcomes of winning or losing an item at various closing prices.

Example. In the Rational Byzantine Generals problem from Chapter 2, a general's type includes all information that will affect his preference for the consensus outcomes of "Attack" or "Retreat". For example, General 3's type includes his value for the horse and his private prediction of the chance of a successful attack.

Remark 3.1. *The concept of participants'* type *is problem-specific. For a silly example in the Rational Byzantine Generals world, the information that determines a general's utility for various battle plan consensus outcomes is (probably) different from the information that determines the general's utility for various foods in the problem of choosing what the generals should eat for dinner. This example illustrates that the sets of information defined by a participant's* type *and its* private information *are not equal. The concept of type will be*

[1]We assume that participants are able to calculate their utility for particular system outcomes, perhaps by using an economic analysis or a *felicific calculus* [Ben23]. We believe that the concepts introduced and used in this thesis apply even when participants are bounded in their rational decision making (cf. bounded rationality [FT91]) but we do not study those limitations in this thesis.

important later in this chapter when we consider incentive compatibility, *which will be defined in terms of truthful revelation of type.*

Remark 3.2. *Note that there is no way for another participant to be certain of another participant's* type *or* private information. *Private information never becomes non-private information; even if a participant announces its private information, there is no way for another person to ensure that this information is truthful. (As a silly example, if a participant claims that her favorite color is blue, there is no way to know if this is the truth.)*

We denote type using the symbol θ. It is convention that the type of participant i is often denoted θ_i, but there is not necessarily a unique type for every agent. Since type determines a participant's utility function, and since a utility function determines a participant's utility for a particular outcome, full utility statements are written in terms of type: $u_i(o_1;\theta_i)$ (read: the utility of participant i for outcome o_1, given that participant i has type θ_i). This writing is precise, because it allows for utility statements such as $u_j(o_1;\theta_i)$, $u_k(o_1;\theta_i)$ where all three participants i, j, and k would have the same type i, and thus the same utility value for o_1 or any other system outcome. In this thesis, we assume that a participant's type is well-defined and does not change over the course of a system.

Preference orderings are expressed via a *preference relation* (denoted $\succ$ or $\prec$), and are rationalized with a participant's utility function. Thus, we say that participant i prefers outcome o_1 to outcome o_2, written $o_1 \succ o_2$, if and only if $u_i(o_1;\theta_i) > u_i(o_2;\theta_i)$.

Utility can be expressed in terms of *cardinal utility* and *ordinal utility*. Most of the examples in this thesis will assume cardinal utility; work using ordinal preferences is discussed in this chapter's bibliographic notes. When using ordinal utility, a participant can state the relative preference ranking of one outcome vs. another outcome, but cannot make a statement expressing the relative strength of the preference. In contrast, when using *cardinal utility*, a participant assigns a number to each outcome, which allows statements about the relative strengths of preferences for outcomes. While there is no well-established unit of utility, it is more convenient to pick a unit that is both transferable among participants and is meaningful outside of the distributed system. For the rest of this thesis, we assume that cardinal utility is measured in a monetary currency (e.g., *Euros*), and that all participants are able to express their utilities in this monetary currency. This assumption is backed by the assumption of participants' *quasi-linear utility* functions. The assumption of quasi-linear utility functions means that utility can be transfered between agents via monetary transfers by using a *payment function* $p(\cdot)$.[2]

[2]We denote a function x with notation $x(\cdot)$ when it might be confused, such as in this case where we want to distinguish payment function $p(\cdot)$ from some scalar payment p or price p.

Definition 3.1 (Quasi-linear utility function)**.** A **quasi-linear utility function** for agent i is a mapping of each outcome $o \in O$ to a utility value expressed of the form:

$$u_i(o; \theta_i) = v_i(o; \theta_i) - p_i(o)$$

where $v(\cdot)$ is a participant's *valuation function* dependent on outcome and type, and $p(\cdot)$ is the *payment function* dependent on outcome. The valuation function, like the utility function, is a mapping from an outcome to a preference for the outcome. The payment function is a monetary transfer, either positive (a payment from the participant) or negative (a payment to the participant), that depends on the system outcome.

Example. In the Rational Byzantine Generals problem from Chapter 2, assume that General 3 has value 10 for his horse. The utility of outcome o_1 "Attack" where he loses his horse can be represented as: $u_3 = v_3(o_1; \theta_3) - p_3(o_1)$ or $u_3 = -10 - p_3(o_1)$. General 3's utility for outcome "Attack" is positive only if $p_3(o_1) < -10$, which means that General 3 receives at least 10 to compensate the loss of his horse.

Remark 3.3. *We observe the need for some support to enable and enforce currency and payment transfers. The structure utilized later in this thesis is a trusted bank. We believe that the reliance on this trusted component is justified in the same way that the reliance on a trusted public key infrastructure is used to tolerate Byzantine faults [Rei95, CL99]: the trusted component is not ideal, but it is a practical solution to a systems problem. In Chapter 6 we give an example of a bank operating in a real system.*

3.3 Actions

Traditional distributed systems researchers use state machines [Wei92, Lyn96] to model node behavior. The state machine model is also useful in our work where nodes are proxies for rational participants. We will use state machines to model participant *strategies*, which are expressed through their nodes. A node's state machine *SM* consists of:

1. A set T of *states*, where one state is designated as the *initial state.*

2. A set $A = \{EA, IA\}$ of actions (whose availability may differ at each state), of which set IA are *internal actions* and set EA are *external actions.*

3. A set L of state transitions of the form (t, a, t') where t and t' are states in T and a is an action in A.

A node's *state* captures all relevant information about its status in a system. It may include received messages, partial computations, private information about its participant, and estimated or derived knowledge about other nodes, participants, and the system.

External actions cause externally visible side effects. The most common side effect is a *message*, which is sent to one or more other nodes as part of a state transition (t, a, t'). Somewhat paradoxically, the absence of a message is also an externally visible effect and is interpreted by other participants as a delay. External actions are the only verifiable actions of a participant, where by *verifiable* we mean that other participants can confirm that these actions have occurred. This verification is possible by witnessing messages (or delays) generated by these actions. As such, only external actions are considered in traditional specification implementation correctness proofs.

Internal actions cause internal side effects. These actions are private and not visible to other nodes. We model these actions as instantaneous actions. If in fact an internal action is not instantaneous, we model the action as two actions – an internal action and an external delay action.

It is useful to divide external actions into three categories: information revelation, computation, and message passing. The reason for this separation is to aid the designer in proving that actions will be correctly implemented by rational participants. The effective approach to ensure proper behavior from a rational participant will differ for each category of action, as we will see later in this thesis. These externally visible actions are the result of a *function* that maps private state information to a *delay* or a *message* that is delivered to at least one other node.

A Note on Delay Actions

We identify two settings where delay actions may be of particular strategic interest, but both of these are outside of the assumptions made by this thesis: In the first setting, participants' nodes may strategically enter or leave the system over the course of the running of a system. In the second setting, a participant changes its type over the running of a system, possibly due to information that it learns from other participants.

Example. As of this writing, the eBay auction system [eBa08] is an example of a real system that exhibits both of these delay properties: First, participants enter and leave the system at different times. Second, participants tend to refine their values for items based on the closing prices of similar auctions [RO01] or based on the level of activity for a particular item. In other words, participants rely on other participants' activity to help refine their type. On the eBay system, it behooves a participant to *snipe* an auction; that is, a participant should delay until the last possible second before placing their bid. The delay strategies are useful because early participation sends a signal to competitors that may ultimately increase the price of an item, thereby decreasing the winning participant's utility.

Some delay actions may not be based on private information. In this case, we choose to treat the actions as computation actions, described below. However, other delay actions may be taken because of private information: Consider a hypothetical protocol where participants' nodes have synchronized clocks and communicate with each other by waiting some number of seconds before sending a one-bit "ping" message, where the delay between messages is significant only in revealing a participant's type. In this example, the absence of the "ping" message is actually informative. Alternatively, consider an auction for an item with a publicized minimum bid of x. If a rational participant does not bid on the item before the close of the auction, one can infer that this participant's value for the item is less than x. In these cases, because the delay is informative, we treat the delay as a "hidden" message and a form of information-revelation action, described next.

3.3.1 Information Revelation Action

Definition 3.2 (information revelation action). A node's **information revelation action** is a delay or message-generating function whose inputs include private information from the participant.

A subset of *information revelation actions* are *type-restricted information revelation actions*:

Definition 3.3 (type-restricted information revelation action). A node's **type-restricted information revelation action** is a delay or message-generating function where the message is a (perhaps partial) claim about the participant's type.

The difference between these two types of actions is that a *type-restricted information revelation action* by definition only provides another participant with information about this participant's utility function, whereas the more general *information revelation action* is not so restricted. (Examples are given below.) This thesis will only address rational manipulation in protocols whose information revelation actions are type-restricted. The usefulness of carving off the subset of type-restricted information revelation actions will become clear later in the chapter when we define *incentive compatibility*.

This definition uses *claim* to mean that a node may make an untruthful statement about the participant type (e.g., declaring θ_i when the participant has type θ_j), or may make a a partial statement (e.g., declaring the participant to have either θ_i, θ_j, or θ_k) or a statement that is inconsistent with previous claims (declaring θ_j when it earlier sent a message claiming type θ_i).

Example. In the Rational Byzantine Generals problem from Chapter 2, each general is asked to send a message containing its vote for an attack plan. General 3 takes the action $f(\theta_3) = \textit{Retreat}$. This message revelation is an example of type-restricted information revelation since the action is an utility claim that can be made only by General 3.

Example. A group of employees wants to calculate their average salary without any single employee learning the salary of the other participants. They follow this simple algorithm [Sch95]: The participants form a ring. The first participant adds a huge private negative number M to his salary s_1 and whispers the result $M + s_1$ to the second participant, who adds his salary and whispers the result $M + s_1 + s_2$ to the third participant, and so on, until the n^{th} participant whispers $M + s_1 + s_2 + \ldots + s_{n-1} + s_n$ to the first participant, who subtracts M, divides this total by n, and publicly announces the result. These actions are information revelation actions. None of these actions are type-restricted as they do not reveal any information about the employee's type: while salary information is private (since we assume that verifying a claim with the boss is taboo), it is not part of the type in that a participant's individual salary does not affect its valuation for successfully learning the average salary.

Example. Participants in a distributed algorithm are given a list of numbers and a proposed algorithm outcome and are told to take the mean of the values. Each participant is instructed to report "yes" if their private utility for the proposed outcome is greater than the mean, and "no" otherwise. This yes/no message action is a type-restricted information action since it provides a partial claim (in the form of a range) of a participant's type. We note an interesting subtlety in this example: the internal mean calculation cannot be directly checked unless it were made into an external computation action (defined below), but an indirect check is possible if there are incentives for a rational node to perform the subsequent comparison correctly. In other words, if there is an incentive to say yes/no truthfully, then the same incentive encourages the node to perform correctly the mean calculation and comparison.

3.3.2 Computation Action

Definition 3.4 (computation action). A node's **computation action** is a delay or message-generating non-identity[3] function whose inputs exclude a participant's private information.

In contrast to an *information revelation action*, all *computation actions* can be replicated. In other words, any node in the system can perform the same computation action, and correctly generated messages resulting from these actions will be identical.

Example. In the salary example, the final step where the first employee announces the result of subtracting the reported total by M and dividing by n is an *information revelation action* and not a *computation action*, because M is private information.

Example. At the end of the Rational Byzantine Generals problem from Chapter 2, each general is asked to run an *outcome function* where the inputs are a consistent view of reported votes. The consistent view of votes, resulting from Lamport's agreement algorithm, is public information. If the outcome function generated a message for the queen, we would classify this outcome function as a *computation action.*

We restrict computation actions in that they cannot be identity functions, meaning that the generated message cannot be exactly equal to the input of the function for all possible inputs. When the identify function is used, we instead label the action as a *message-passing action.*

3.3.3 Message-Passing Action

Definition 3.5 (message-passing action). A node's **message-passing action** is message-generating identity function whose input (and thus, output) is limited to a previously received message.

Example. Part of the Rational Byzantine Generals problem from Chapter 2 requires each general to simply relay other generals' votes to the remaining generals. For instance, General 3 passes General 2's vote to General 1. This relay action is classified as a *message-passing action.*

An alternate definition of *computation* and *message-passing* actions could broaden computation actions to subsume message-passing actions. However, this distinction is valuable because it allows us to model the network communication layer as a set of nodes that only execute message-passing actions.

[3]An identity function is defined as $f : x \rightarrow x$; the message output is always equal to the message input.

3.4 Strategy

Earlier in this chapter, we wrote that a participant actively controls and chooses the behavior of its node. This concept of participant choice is known as *strategy.* A participant's strategy is the complete plan or decision rules that define the actions that a participant's node will take in every state of the system. We use the language of state machines to define strategy:

Definition 3.6 (strategy). A participant's **strategy** is the complete state machine implemented by that participant's node.

We denote the particular state machine selected by participant i as s_i, selected from the *participant strategy space* Σ_i defining the world of strategies available to participant i. The world of strategy spaces for all participants is denoted Σ. This thesis uses the concept of strategy *equivalence*:

Definition 3.7 (strategy equivalence). Two strategies s_1 and s_2 are **equivalent**, denoted $s_1 \equiv s_2$ as long as the two strategies produce the same sequence of external actions when provided the same external inputs, even if the strategies differ in internal actions.

Cross-Field Connection: In traditional direct mechanism design, a participant's strategy is conditioned on its knowledge of the mechanism and its type. Given a mechanism, $s_i(\theta_i)$ defines a participant's (type-restricted information revelation) actions in all states of the world. This definition is incomplete in distributed mechanism design. For example, computation and message passing actions are restricted based on the participant's network location and from any computational limitations of the node. As an example, consider three participants that have the same type, but whose nodes are networked in a line: $i - j - k$, so that i's node can talk directly to j's node, but not directly to participant k's node. Participant i cannot choose any strategies with actions that involve direct communication to participant k, and participant j has strategies that are enabled in its role as intermediary of messages sent from participant i to k.

3.4.1 Suggested Strategy

Out of the strategies composing the *strategy space* Σ_i, the distributed system specification provided by the system designer defines a particular *correct* state machine to be implemented by a participant's node. This state machine specification of correct behavior has a special name: it is the *suggested strategy* $s_i^m \in \Sigma_i$ for participant i. More

Σ	strategy space potentially available to any node.
Σ_i	strategy space available to node i. $\Sigma_i \in \Sigma$.
Σ^m	strategy space defined in mechanism specification (i.e., defined in $g(\cdot)$). $\Sigma^m \in \Sigma$.
Σ_i^m	subset of strategy space available to node i defined in mechanism. $\Sigma_i^m \in \Sigma^m$, $\Sigma_i^m \in \Sigma_i$.
s^m	suggested individual strategies for every node in the mechanism. $s^m \in \Sigma^m$.
s_i^m	suggested individual strategy for node i. $s_i^m \in s^m, s_i^m \in \Sigma_i^m$.
s_i^*	equilibrium strategy for node i.
s_i	the particular strategy (state machine) picked by node i. $s_i \in \Sigma_i$.

Table 3.1: A summary of the notation used in describing strategies and strategy space. Note that there is no automatic relation between s_i, s_i^m, and s_i^*, but that it will be a goal of the mechanism designer to make $s_i \equiv s_i^m \equiv s_i^*$.

generally, the specification may provide a *suggested strategy profile* containing the suggested strategies s^m for *every* participant in the network. A participant i that directs its node to follow strategy $s_i' \not\equiv s_i^m$ is exhibiting failure. Note that again we use equivalence and not equality because failure defined as $s_i' \neq s_i^m$ is too strong of a statement: a node may use the strategy $s_i' \equiv s_i^m$ that produces the same external actions but differs in internal actions. In this case, s_i' would not exhibit failure.

A Notation on Notation:
It is important to understand the various strategies and strategy spaces that are used in this thesis. Table 3.1 summarizes the notation that is defined in this chapter, while later in this chapter Figure 3.1 gives a pictorial example of how these concepts interrelate.

3.4.2 Equilibrium Strategy

One important class of strategy is known as a participant's *equilibrium strategy*, denoted $s_i^* \in \Sigma_i$ for every participant i in the network. Informally, s_i^* is the strategy that participant i will choose to maximize its utility from the system outcome, given its understanding about the environment, including what it believes about other nodes. We write more on the concept of *equilibrium* later in this chapter. The goal of the system designer is to produce a system with a "good" equilibrium strategy s_i^* for rational participants, where the publicized suggested strategy $s_i^m \equiv s_i^*$, so that rational nodes choose to implement s_i^m correctly in order to maximize their utility.

A Notation on Notation: Readers unfamiliar with economics literature face a short learning curve to parse the notation commonly used when discussing utility functions and types. It is common to encounter a term like:

$$u_i(f(s_i(\theta_i), s_{-i}(\theta_{-i})); \theta_i)$$

In English this equation measures "the utility that i receives when i truthfully reveals its true type, and everyone else reveals their true type." From left to right, the equation is read: the utility to node i for the outcome (calculated by outcome function f) when node i plays the strategy that would be chosen by a node with type θ_i, given that agents other than node i play strategies corresponding to their types θ_{-i} and that agent i has type θ_i. To a systems practitioner, this notation may seem confusing – and this feeling is justified. Part of the confusing aspect of the notation is that θ_{-i} here means "the types of the participants other than node i", which is particularly confusing because one of those types could certainly be θ_i. (If this confuses you, imagine that there could be a limited number of types and an infinite number of nodes. Some nodes share the same type.) Another confusing aspect is that u_i refers to node i's utility, s_i refers to node i's strategy, while θ_i in most contexts refers to a type labeled i that may not have to do with node i. To give an example:

$$u_i(f(s_i(\theta_j), s_{-i}(\theta_{-i})); \theta_k)$$

In English, this means "the utility that node i receives (who has type k) when it pretends to have type j, given that everyone else reveals their true type. What makes the notation additionally confusing are the layers of superscripts and subscripts that change the meaning of the equation. For example:

In context, $s'_i(\theta_i)$ refers to a strategy other than some $s_i(\theta_i)$.
$s^m_i(\theta_j)$ refers to node i following the suggested strategy for a node with type j.
$s^*_i(\theta_i)$ refers to node i following an equilibrium strategy for a node with type i.
$s_i(\theta_i)$ generally means node i acting as its true type i, while $s_i(\hat{\theta}_i)$ generally means node i acting (lying) as if it were a different (and unspecified) type.

Strategies can be written without their predicate: s'_i means $s'_i(\theta_i)$ and s^*_i means $s^*_i(\theta_i)$.

3.5 Rationally Motivated Failure

With these preliminaries defined, we are ready to define rationally motivated failure. This definition is in terms of a utility statement that compares the results of following the suggested strategy s_i^m to some alternative strategy, $s_i^{'} \not\equiv s_i^m$.

Definition 3.8 (rationally motivated failure). Participant i's node exhibits **rationally motivated failure** when participant i intentionally causes its node to execute strategy $s_i^{'} \not\equiv s_i^m$ because of the participant's expectation that

$$u_i(g(s_i^{'}(\theta_i), s_{-i}(\theta_{-i})); \theta_i) > u_i(g(s_i^m(\theta_i), s_{-i}(\theta_{-i})); \theta_i)$$

where s_{-i} denotes the strategies executed by other nodes and where $g(\cdot)$ is a function that maps strategies to the system outcome.

This definition defines rationally motivated failure as a participant's deviation from the suggested strategy in order to improve its expected utility by trying to force an alternate system outcome. We do not worry about the *cost* of deviation in this definition, but solely focus on the payoffs from outcomes. If the designer is successful in providing $s_i^m \equiv s_i^*$ for all rational participants in the system, we can actually prove that nodes will not exhibit *rationally motivated failure* and will instead show *rationally motivated correctness.*

3.6 The Mechanism as the Remedy

We have reached a turning point in the preliminaries of this thesis; this chapter has so far been concerned with the build-up of the problem of rationally motivated failure. We now define the concepts necessary to describe the solution.

In the last section we defined participant i's rationally motivated failure in terms of a strategy picked from a strategy space $s_i \in \Sigma_i$ and a mapping function $g(\cdot)$. The *mechanism* defines the valid strategy space and outcome mapping function.

Definition 3.9 (mechanism). A **mechanism** is comprised of two components: a *restricted strategy space* Σ^m that defines the set of valid node strategies, and an *outcome rule* $g(\cdot)$ that maps the combination of each node's selected strategy $(s_i \ldots s_n \in \Sigma^m)$ to a system outcome.

The restricted strategy space Σ^m is not necessarily equal to the entire strategy space Σ. We imagine that a mechanism is run in the wider world where all sorts of crazy actions are possible. The mechanism designer limits Σ^m to make the construction of a distributed mechanism manageable by restricting the domain of $g(\cdot)$. In practice, the designer may attempt to limit Σ to exactly match Σ^m with design tools that will be discussed below and in Chapter 4.[4]

Example. A sealed-bid second-price auction is an example of a simple mechanism. A trusted auctioneer defines participants' Σ^m to be those strategies that result in the revelation of a number, known as the buyer's *bid.* Σ is more general than Σ^m, and includes strategies where the participant attempts to bribe the auctioneer, or attempts to prevent another participant from bidding, etc. The auctioneer defines $g(\cdot)$ as a mapping from submitted bids to two instructions. The first instruction tells the seller to award the item to the single participant who submitted the highest bid. The second instruction tells the winning buyer to pay the seller the second-highest bid that was submitted in the auction. A designer may enforce Σ^m by removing the opportunities for bribery or manipulation, etc., to ensure *mechanism enforcement* where the seller correctly transfers the item and the buyer makes the corresponding payment. In human interaction, the design tools are laws, police, courts, a free investigative media, etc. In computational settings, we'll rely on *abstract dependencies* introduced in the next chapter, which include obedience, hardware restrictions, and incentives.

This thesis frames the discussion of a "remedy" for rationally motivated failure in terms of the design, proof, and construction of a specific mechanism. Our approach to building a system that can prevent rationally motivated failure is based in the ideas found in mechanism design, which has been loosely described as inverse game theory [Pap01]. Whereas game theory studies how participants will interact in a game, mechanism design studies how to build games so that rational participants will exhibit "good" behavior, for some designer-defined notion of "good." This idea of encouraging good behavior can be made concrete by introducing a designer-specified *social choice function.*

Definition 3.10 (Social choice function). The **social choice function** $f(\cdot)$ is a mapping from participant types to a particular social choice outcome:

$$f(\theta_1, \theta_2, \dots \theta_{n-1}, \theta_n) \rightarrow o^m$$

The goal of mechanism design is to construct a *mechanism* so that participants acting in their own self-interest actually promote the social choice outcome o^m intended by

[4]In traditional centralized mechanism design any restriction is easily performed by the center, which can ignore any unsupported strategies.

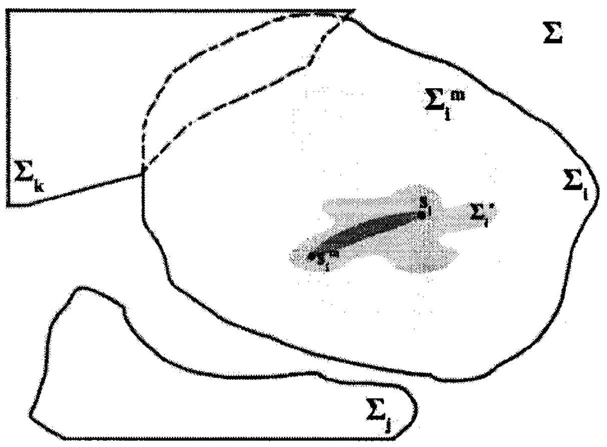


Figure 3.1: Visualizing node strategy. This three-node example focuses on node i. Node i has picked particular strategy s_i, which produces the same sequence of external actions, and thus is equivalent, to a range of strategies denoted by the darkest band. In particular, $s_i \equiv s_i^m$, where s_i^m is the suggested strategy for participant i. In this example these equivalent strategies have been crafted to land in node i's equilibrium strategy space Σ_i^* and so s_i is also s_i^*. Σ_i^* is a subset of Σ_i^m, which is the strategy space defined by the mechanism (i.e., the strategy space mapped by $g(\cdot)$), which in turn a subset of the set of strategies Σ_i available to i.

the mechanism designer.

Whereas $f(\cdot)$ maps types to an outcome, $g(\cdot)$ maps strategies to an outcome. In the ideal world, all participants would be truthful about their types in following strategies and this truthfulness would lead to o^m as the outcome of the mechanism. In practice, participants may not desire outcome o^m and can instead choose a strategy in an attempt to drive the system toward an alternate outcome.

Definition 3.11 (mechanism design problem). The **mechanism design problem** is to build a mechanism so that nodes with types $(\theta_1, \theta_2, \ldots \theta_{n-1}, \theta_n) \in \Theta$ can express and will choose to follow their chosen strategies selected from strategy space Σ^m so that the mechanism's outcome function $g(\cdot)$ implements social choice function $f(\cdot)$.

3.6.1 Approaches to Mechanism Design

Centralized Mechanism Design

Traditional direct mechanism design effectively confines the strategy space Σ^m to strategies involving only *type-restricted information revelation actions* so that nodes may only (perhaps falsely) report information about their type. This is because traditional direct mechanism design simply does not model situations where self-interested participants' nodes have a role in computing, communicating, and enforcing $g(\cdot)$. In traditional direct mechanism design the enforcement of a restricted Σ^m and the implementation of $g(\cdot)$ are performed by a trusted *center* that is exogenous to the participants. In traditional mechanism design, function $g(\cdot)$ is a mapping from the strategies that (mis)report type information to an algorithm outcome. In contrast to direct mechanism design, *indirect* mechanism design allows for more general information revelation actions but the enforcement of a restricted Σ^m and the implementation of $g(\cdot)$ are still performed by an exogenous trusted *center.*

Example. The sealed-bid second-price auction (cf. Vickrey [Vic61]) is a simple example of a direct traditional centralized mechanism.

Distributed Mechanism Design

In the distributed mechanism design problem, the participant strategy space includes the three strategy components of *information-revelation, message-passing,* and *computational strategy.* The suggested strategy s_i^m for each participant i decomposes into $s_i^m = (r_i^m, p_i^m, c_i^m)$, with information-revelation strategy r_i^m, message-passing strategy p_i^m, and computational strategy c_i^m.

In distributed mechanism design the actual implementation of $g(\cdot)$ is performed by the very participants who can exhibit failure or who may prefer system outcomes other than designer-intended o^m. Stated another way, $g(\cdot)$ is itself implemented as part of the participants' node strategy and Σ^m specifies both how a participant "plays a game" *and also* what game the participant will implement! If the designer fails to construct s^m such that nodes implement $g(\cdot)$ correctly, participants may follow strategies *outside* of Σ^m, which may result in a system outcome not even defined in the mapping from valid strategies to outcomes $g(\cdot)$. This is because Σ_i^m can be ignored by node i unless this restricted strategy space is enforced outside of the mechanism. Shown pictorially in Figure 3.1, Σ_i^m is not the actual strategy restriction unless some outside tools are used to constrict Σ_i down to Σ_i^m.

Remark 3.4. *Given that node strategy is effectively unbounded in distributed mechanism design, is a definition of Σ^m or $g(\cdot)$ useful? Our answer is that in practice, external tools are used to restrict the set of valid strategies to Σ^m, of which a particular s^m will exist in which nodes implement $g(\cdot)$ correctly. This strategy s^m needs to be a system equilibrium that will hold because other participants are also playing that strategy. Another way to visualize this using Figure 3.1 is that in practice, the designer will make Σ_i snap down to Σ_i^m, and will make Σ_i^* snap down to a singular point s_i^m, for each node i. In the next chapter, and in the examples in this thesis, we will see how such a restriction can occur.*

Example. In the Rational Byzantine Generals problem from Chapter 2, the queen plays the role of the mechanism designer. She specifies to the three generals that each general should vote to "Attack" if and only if they predict the city will be razed in a combined attack and to vote "Retreat" otherwise. In this problem, social choice function $f(\cdot)$ maps the majority of truthful razing opinions, which is part of each general's type, to a battle plan. The queen imposed a $g(\cdot)$ and Σ^m that corresponded to a direct implementation of $f(\cdot)$. But this mechanism was broken in the sense that it did not meet Definition 3.11: $g(\cdot)$ did not implement $f(\cdot)$ for these participants using the intended strategy space Σ^m. The queen's suggested strategy s^m was that every node implement $g(\cdot)$ correctly and provide a one-word input to $g(\cdot)$ based on an honest assessment of victory. However, $g(\cdot)$ was neither sufficient to capture General 3's type (as Σ^m was not expressive enough), nor did it provide incentives for General 3 to act according to the queen's wishes. Since General 3 had no way to coordinate with other generals to choose a different mechanism, he did what he could to change the outcome computed by $g(\cdot)$. Two examples of General 3's response were described in Chapter 2: in the first example General 3 implemented $g(\cdot)$ correctly and kept his strategies within the intended Σ_3^m but failed to provide truthful input, and in the second example General 3 provided truthful input but failed in implementing $g(\cdot)$ by using a strategy outside of Σ_3^m.

In the previous queen example, we wrote that "$g(\cdot)$ did not implement $f(\cdot)$ *for these participants.*" For example, had the generals been clones of the queen, we would have expected s^m to have been a success: $g(\cdot)$ should have been implemented correctly and votes for a battle outcome would have been based on an honest assessment of victory. As a forecast of what is to come in this thesis, one cannot solve the mechanism design problem listed in Definition 3.11 without stating the solution in terms of environment assumptions (discussed as the first step in the methodology in Section 4.1) that support a particular *solution concept* (discussed later in this Chapter in Section 3.6.3).

3.6.2 Mechanism Equilibrium

The equilibrium of a system dictates how rational participants will select their strategies, and in most *solution concepts* (defined below) it is through strategy selection that rational participants create the system equilibrium. This cycle is the key that allows system designers to make provable statements that a particular system prevents (compatible) rationally motivated failure.

Definition 3.12 (equilibrium). The **equilibrium** in a mechanism is the situation when no participant can force the selection of a selfishly better system outcome by changing only its strategy.

The suggested strategy s^m may not correspond to a system equilibrium, or may correspond to many equilibria. Both of these situations are problematic for a system designer. In the case where the suggested strategy is not a mechanism equilibrium there is nothing to stop a rational participant i from choosing to follow strategy $s_i' \not\equiv s_i^m$. When a distributed mechanism has multiple equilibria there is a *coordination problem* in that the system designer must somehow convince players to pick strategies associated the same mechanism equilibrium. In systems with multiple equilibria the suggested strategy s_i^m for each agent acts as a *coordination device* to ensure that participants focus on the same system equilibrium.

Example. It is easy to explain the coordination problem with a real life example [Par04]: two people wandering separately around New York want to meet up on a street corner in Manhattan. There are many equilibrium strategies (corresponding to each of the street corners) that lead to outcomes with the same utility. However, there still needs to be some coordination to ensure that the two people pick the same equilibrium. One third party coordination device might be an aerial skywriting plane that happens to spell out the address of a particular street corner, effectively providing s^m to each person.

Example. A default client can act as a coordination device in a software system where there may be multiple equilibria. For example, the official BitTorrent client is described in our earlier work [SPM04] as a coordination device in the BitTorrent protocol.

3.6.3 Mechanism Solution Concepts

A claim of a mechanism equilibrium is not made in a vacuum. Rather, the designer formally states the strategies that are expected from rational nodes by proving an equilibrium statement in terms of a *solution concept* that dictates the equilibria identifying the strategies followed by rational nodes.

Definition 3.13 (solution concept). A **solution concept** is a mapping from a mechanism to a set of equilibrium strategy profiles.

Each equilibrium strategy profile specifies the exact equilibrium strategy of every participant. If the solution concept maps the mechanism to a single strategy profile, we say that there is a unique equilibrium in that mechanism using that solution concept.

This section provides a brief tour of several important solution concepts that originate in the game theory literature (cf. Fudenberg and Tirole [FT91]) and are particularly applicable to our work in system design. Additionally, we define a new solution concept that permits reasoning in the presence of rational and traditional faults.

Dominant Strategy

A standard solution concept is the *dominant strategy* equilibrium [Jac00]. This solution concept identifies an equilibrium profile where the only assumption made by participants is that other participants can play any strategy in Σ_{-i}.

Definition 3.14 (dominant strategy equilibrium). A strategy profile s^m is a **dominant strategy equilibrium** in distributed mechanism specification $m = (g, \Sigma^m, s^m)$, if s_i^m satisfies

$$u_i(g(s_i^m(\theta_i), s_{-i}(\theta_{-i})); \theta_i) \geq u_i(g(s_i'(\theta_i), s_{-i}(\theta_{-i})); \theta_i)$$

for all rational compatible participants i with θ_i, for all participants $-i$ with θ_{-i}, for all $s_i' \neq s_i^m$.

When a system designer can suggest a dominant node strategy to some participant, the suggested strategy is utility maximizing for that participant, regardless of other participants' node strategies selected from Σ_{-i}. Unfortunately, we will rarely encounter a distributed mechanism with a dominant strategy solution concept. The main problem is that the strategy space Σ_i^m is probably unenforceable as a boundary on Σ_i when participants $-i$ are not following strategies in Σ_{-i}^m. In the centralized setting, Σ_{-i} could be reasonably restricted to simple information revelation actions. In the distributed setting, participants' nodes implement the rules of a mechanism and it is possible that other nodes can change the rules of the game in the middle of the game as part of their strategy! It seems unlikely that a participant can pick a utility-maximizing strategy *regardless of the mechanism*, with the exception of strange corner cases such as when the participant's utility is the same for all system outcomes, for all games. We view the loss of the dominant strategy solution concept as a necessary cost of moving from traditional to fully distributed mechanism design.

Example. A small silly example of the failure of a dominant strategy mechanism in the distributed setting is as follows: two people agree to implement a distributed Vickrey-Clarke-Groves (VCG) mechanism [Vic61, Cla71, Gro73] to bid for and allocate a new car.

Following an established distributed VCG protocol, Player One submits her truthful bid for the car to Player Two. Player Two changes her portion of the decision function to force the outcome "Player Two wins the car" and terminates the distributed algorithm.

Nash

On the other end of standard solution concepts, as ranked by knowledge requirement, is the *Nash equilibrium* [Nas50]. This famous solution concept requires that a participant understand the types and strategies of all other participants in the system. Given this requirement, in equilibrium every participant will select a utility-maximizing strategy given the strategy of every other agent:

Definition 3.15 (Nash equilibrium). A strategy profile s^m is a **Nash equilibrium** in distributed mechanism specification $m = (g, \Sigma^m, s^m)$ if s_i^m satisfies

$$u_i(g(s_i^m(\theta_i), s_{-i}^m(\theta_{-i})); \theta_i) \geq u_i(g(s_i'(\theta_i), s_{-i}^m(\theta_{-i})); \theta_i)$$

for all rational compatible participants i with θ_i, for all $s_i' \neq s_i^m$.

Whereas dominant strategy could be defined without reference to the equilibrium strategies of other nodes, the Nash equilibrium solution concept (and the remainder of the solution concepts defined in this section) depend (at least partly) on other nodes' equilibrium behavior.

The Nash equilibrium has received attention in recent literature on network games [Rou01] but adopting this notion for practical systems is not usually appropriate; a Nash equilibrium requires a participant to have knowledge of other participants' type, which is usually unrealistic.[5]

Example. The knowledge assumptions underlying a Nash equilibrium solution to the Rational Byzantine Generals problem from Chapter 2 would require that each general knows exactly the type and selected strategy of every other general. This requirement is akin to each general being a mind-reader with perfect predictive ability.

Ex post Nash

Ex post[6] Nash is a refinement on Nash; its definition is exactly the same as Nash

[5]On the other hand, it could be argued that a variant on Nash equilibrium *is* appropriate when it is well known that nodes are implementing a default obedient strategy, perhaps provided as part of a default client software installation.

[6]Ex post is Latin for "after the fact". In models where there is uncertainty that is resolved by the

except that it relies on a different participant knowledge model. The assumption in ex post Nash is that the rationality of participants is common knowledge amongst participants.

Definition 3.16 (ex post Nash equilibrium). A strategy profile s^m is an **ex post Nash equilibrium** in distributed mechanism specification $m = (g, \Sigma^m, s^m)$ if s_i^m satisfies

$$u_i(g(s_i^m(\theta_i), s_{-i}^m(\theta_{-i})); \theta_i) \geq u_i(g(s_i'(\theta_i), s_{-i}^m(\theta_{-i})); \theta_i)$$

for all rational compatible participants i with θ_i, for all rational compatible participants $-i$ with θ_{-i}, for all $s_i' \neq s_i^m$.

The knowledge assumption in *ex post* Nash is much weaker than that required in the more standard Nash equilibrium solution concept, but still stronger than dominant strategy. Unlike Nash, a participant does not need to know other participants' exact types. This solution concept is appropriate when a system contains only rational compatible behavior. Our work on the FPSS protocol in Chapter 5 uses this solution concept.

k-partial ex post Nash

This thesis introduces the *k-partial ex post Nash* solution concept for systems where failures may still occur despite the designer's use of incentives. This refinement on the ex post Nash concept informally states that each rational compatible participant i should direct its node to follow the suggested strategy s^m regardless of the identity and actions of "faulty" nodes, as long as at least k other nodes also adhere to the specification. This concept is attractive because it says that in practice a participant just needs to know that compatible rationality (or obedience) of at least k other nodes is common knowledge amongst nodes.

Definition 3.17 (k-partial ex post Nash equilibrium). A strategy profile s^m is a **k-partial ex post Nash equilibrium** in a distributed specification $m = (g, \Sigma^m, s^m)$ if s_i^m satisfies:

$$u_i(g(s_i^m(\theta_i), s_1^m(\theta_1)...s_k^m(\theta_k), s_{-(i,1...k)}^?(\theta_?)); \theta_i) \geq u_i(g(s_i'(\theta_i), s_1^m(\theta_1)...s_k^m(\theta_k), s_{-(i,1...k)}^?(\theta_?)); \theta_i)$$

for all rational compatible participants i with θ_i, for all rational compatible participants $1...k$ with $\theta_1...\theta_k$, for all $s_i' \neq s_i^m$, and where the remaining nodes have an arbitrary type and follow an arbitrary strategy.

execution of the mechanism, the ex post calculation of utility from the system outcome is the calculation that occurs after the uncertainty has been resolved. Here, ex post Nash refers to the situation where the common rationality of participants is assumed and then the equilibrium holds when this assumption is borne out in practice.

This solution concept is technically a family of solution concepts whose members are defined by values of k; our work on the RaBC protocol in Chapter 6 uses a *1-partial ex post Nash* solution concept, meaning that the solution concept holds as long as at least one other node in the algorithm is rational compatible (or obedient).

3.6.4 Choosing the Appropriate Solution Concept

In traditional mechanism design, the knowledge model is the sole factor in determining the appropriate solution concept. Distributed mechanism designers must not make the mistake of picking a solution concept in the same fashion.

In the next chapter, we will explore how the participant, network, and dependency models affect the decision to pick an appropriate solution concept in a distributed environment. In general, the system designer should pick a solution concept that is appropriate to these assumptions. For example, traditional solution concepts are not appropriate when faced with unrestricted failure. Rational nodes cannot depend on failing nodes to behave rationally, and the implementation of $g(\cdot)$ may change because of node failure. Solution concepts such as the k-partial ex post Nash family are appropriate when some nodes fail. In other more restricted settings, other appropriate solution concepts such as *trembling hand Nash equilibria* [Sel86] or *k-fault tolerant Nash* [Eli02] (both described in this chapter's bibliographic notes) may be appropriate.

Remark 3.5. *Each of the solution concepts as presented above assumes that nodes, although self-interested, are also benevolent in the sense that a node will implement the suggested strategy as long as it does not strictly prefer some other strategy. In the previous solution concepts, a strong equilibrium would hold if the utility comparisons were stated in terms of strict inequality and not weak inequality.*

3.7 Bibliographic Notes

Utility and Type

All of the examples in this thesis assume cardinal utility. Other work, such as Halpern and Teague [HT04] and McGrew, Porter, and Shoham [MPS03] work with ordinal and not cardinal preferences. Cardinal utility is appropriate in this thesis, where we will rely on monetary transfers to provide incentives for correct participation.

Brandt [BSS07] studies participants whose utility, contrary to the common assumption of self-interest, is instead based on how damaging a system outcome is for other participants. These participants might be described as possessing an inherent spitefulness. We would label this spitefulness as a form of rational incompatible behavior. Alternatively, these participants might expect competitive scenarios where the loss of a competitor will likely result in future gains, external to the current system.

One assumption that we make in this thesis is that participants are capable of calculating their own utility for various system outcomes. Economics also studies *bounded rational* participants [GS02], who do not have this capability. For example, a participant's computational faculties may not be up to predicting one's utility for an outcome. Alternatively, a participant may only be able to provide a range of possible utilities for an outcome. In our model, we could classify erroneous behavior due to such bounded rational reasoning as *irrational behavior.*

Actions

The particular kind of state machine model used in this chapter is based on that described by Lynch [Lyn96]. The notion of using a state machine to describe a system specification as well as an implementation of this specification is described by Weihl [Wei92].

Work by Monderer and Tennenholtz [MT99] recognizes that participants may perform communication actions as part of the translation from centralized mechanism design to an actual network implementation. Furthermore, the general problem of participants' strategies extending beyond information revelation actions has been identified by Nisan and Ronen [NR99] and by Feigenbaum et al. [FS02]. The breakdown of strategy into components of information revelation, computation, and message passing is described in Shneidman and Parkes [SP04].

Mechanisms, Strategy, and Equilibria

Brafman and Tennenholtz [BT04] also reflect on the problem of directing agents to choose among multiple equilibria. They believe, as we do, that computational systems make the coordination problem easier. The notion of the default client software as the basis for a suggested strategy appears in Shneidman et al. [SPM04].

Solution Concepts

Besides the k-partial ex post Nash solution concept presented here, there are other interesting related solution concepts. In his paper on *k-fault tolerant implementation,* Eliaz [Eli02] investigates the implementation problem arising when some of the players are faulty in the sense that they fail to act optimally. The designer and the nonfaulty players only know that there can be at most k faulty players in the population. Eliaz's work defines a solution concept that requires a player to optimally respond to the nonfaulty players regardless of the identity and actions of the faulty players. The main differences between the solution concept proposed by Eliaz and our work is that Eliaz assumes a Nash equilibrium and is limited to information revelation actions.

Also related to Eliaz's work is the concept of the *trembling hand perfect equilibrium.* A trembling hand perfect equilibrium, described by Selten [Sel86], is an equilibrium that takes into account a small chance that a participant may choose an unintended strategy. It is so-named because a human participant is imagined to suffer from a "slip of the hand" in selecting his strategy. This "failure" creates the possibility of off-equilibrium play.

Abraham et al. [ADGH06] develop *k-resilient Nash equilibria*, which are joint strategies where no member of a coalition of size up to k can do better, even if the whole coalition defects. Abraham et al. show how this concept can be used in secret sharing and multiparty computation problems.

Feigenbaum et al. [FRS06] introduce the *collusion-proof ex-post Nash equilibrium* solution concept. In a collusion-proof ex-post Nash equilibrium, no deviation by a group of agents can strictly improve the outcome of a single agent in that group without strictly harming another. It may be possible to extend this concept into a *k-partial collusion-proof ex-post Nash equilibrium* by using the traditional systems failure approaches highlighted in this thesis.

Chapter 4

Methodology to Address Rationally Motivated Failure

This chapter defines the methodology used to address rationally motivated failure and explicates the step where one proves that a specification prevents rationally motivated failure. We first step through the methodology:

1. **Model the Environment.** In this step, the designer specifies the *environment assumptions*, which define the participant model (including knowledge model), network model, and dependency model. These models describe "expected reality" of a network environment and serve as the basis for reasoning about rationally motivated failure and fault tolerance.

2. **Design, Prove, and Implement Specification.** The designer constructs a new system specification that is designed to prevent rationally motivated failure. The specification consists of a *mechanism*, as defined in the last chapter, and a *suggested strategy* for each type of participant. Specification correctness and failure robustness are proved with respect to a particular *solution concept* that depends on the environment assumptions established in the modeling step. The system specification is then realized with an implementation.

3. **Evaluate Effectiveness, Impact, and Cost.** Designers evaluate the trade-offs required to implement a system that is robust to rationally motivated failure, and measure the cost (in messages, etc.) of the system.

4.1 Model the Environment

The *environment assumptions* lay out the ground rules for a failure analysis. These assumptions capture all of the relevant features and limitations that are present in a system, and that will be relevant to the construction of a distributed algorithm that is robust to rationally motivated failure. The assumptions are stated in terms of a set of *participant*, *network* and *dependency* models.

4.1.1 Participant Model

The **participant model** describes the makeup of participants in the system. It may describe the identity, exact number, percentage, or bound of participants that are obedient, capable of expressing rationally motivated failure, or capable of non-rational failure. In practice it is possible to construct fairly rich and meaningful participant models: This can be done experimentally, such as in Seuken et al. [SPP08], where a model of participants is reverse-engineered from observed user behavior in a real protocol, or simply by fiat, such as in traditional Byzantine distributed systems work like Castro and Liskov [CL99], which simply defines a bound on the amount of certain types of behavior so that their failure analysis will hold.

The participant model includes a *knowledge model* about what rational participants *believe* about other participants in the system – these beliefs may differ from reality but nevertheless serve to explain a rational participant's actions.[1] The knowledge model can be specific and dictate what every participant knows and believes about others. More realistically, the knowledge model can describe beliefs in general terms. For example, a general model can state that participant beliefs equate to:

- **no knowledge:** In this model, nothing is assumed by participants about other participants.[2]

It is useful to separate what a participant believes about other rational participants from what a participant believes about other types of participants. Classes of beliefs about other rational nodes include:

[1]The knowledge model might be better described as a *belief model*, but we use the terminology already established from mechanism design.

[2]For those readers familiar with mechanism design: the "no knowledge" assumption is actually stronger than what is assumed by centralized *dominant strategy*. For example, true "no knowledge" does not assume anything about possible collusion between nodes, whereas claims about *dominant strategy mechanisms* only hold if no collusion is assumed.

- **full ex post rationality:** The rationality of participants is common knowledge amongst participants. The scope of rationality (e.g., anti-social, bounded rational, etc.) must be defined.
- **full knowledge:** In this model, participants have complete type knowledge about other participants in the system.

Participants can believe that only a subset of participants is rational, and that the remaining participants are faulty:

- **k-fault:** In this model, participants are aware that up to k nodes may simultaneously exhibit faults in the distributed algorithm. This model is appropriate to deal with failure in traditional distributed systems, with all existing Byzantine failure work limiting k to be $k < \frac{n}{3}$ (E.g., Lamport [LF82], Castro [CL99]), where n is equal to total number of nodes in the distributed system.

Knowledge models about rational compatible participants and other participants can be combined when appropriate; e.g., a *k-fault, full knowledge* model would describe a setting where a participant believes there are at most k simultaneous faults and where a participant has complete knowledge of all compatible participants.

Cross-Field Connection: For those readers familiar with mechanism design, the knowledge model is recognizable as the standard support of a solution concept. But why do we describe the knowledge model as a part of the participant model? In a system with failure that is not rationally motivated, it is important to capture the *actual* system makeup in addition to each node's *belief* about the system makeup in order to prove meaningful statements about system correctness. The beliefs of a rational node are used to show equilibrium behavior, which in turn is used to show satisfaction of system correctness conditions.

4.1.2 Network Model

The network model includes connectivity (topology), bandwidth, latency information and message limitations. Synchronicity information is included as it affects the proofs and proof techniques used to show distributed algorithm correctness (cf. Lynch [Lyn96]).

4.1.3 Dependency Model

The dependency model makes explicit any "hidden" system features or limitations that may exist in a system. Examples of information contained in the dependency model might be the details of a public key infrastructure (PKI) or the description of the software running on nodes, insofar as these details pertain to rationally motivated failure.

4.2 Design, Prove, and Implement Specification

In this step, the designer constructs a new system specification that is designed to prevent rationally motivated failure. We give an overview of the sub-steps before discussing them in more detail in this section. Specifically, the system designer should:

- Identify a subset of rational compatible behavior (failure) that the designer wishes to address. The system designer chooses to provide incentives to correct a specific subset of bad behavior.
- Specify a *mechanism*. As defined in the last chapter, the mechanism maps between node strategies and the system outcomes. The mechanism is defined by a *strategy space* Σ^m and an *outcome rule* $g(\cdot)$ that specifies the actual mapping.
- Specify a *suggested strategy* s_i^m for each participant type. As described in the last chapter, the suggested strategy is a utility maximizing strategy, for all types $i \in \Theta$.
- Build a rigorous proof of specification faithfulness that holds for a particular *solution concept* that holds in a particular *environment*. Informally, the designer guarantees that a rational compatible node will exhibit *rationally motivated correctness* rather than *rationally motivated failure.*
- Implement the specification, being careful not to violate specification faithfulness.

4.2.1 Identify a Subset of Rational Compatible Behavior.

In Chapter 2.7, we described the difference between *rational compatible* behavior and *incompatible* behavior:

- **compatible** rational behavior can be addressed through incentives and the system design techniques discussed in this thesis, so that participants do not exhibit rationally motivated failure *and* the system operates as the designer intends.
- **incompatible** rational behavior is behavior that cannot be addressed without also sacrificing the intended system goals.

In the perfect world, all rational compatible behavior would be addressed. In practice, preventing some types of rationally motivated failure may be seen as too expensive or difficult, even if addressing such behavior would not sacrifice the original intended system goal. Within the set of rational compatible behavior, the designer should identify the subset that he or she wishes to address. In effect, the system goal is refined so that certain types of rational compatible behavior are treated as incompatible.

Example. In the Rational Generals problem, we might imagine a General who will agree to follow orders if and only if he can become King. While incentives could be created to ensure this General's cooperation, the cost of placating this would-be Caesar is rather high. The queen may choose to treat this type of behavior as incompatible.

Example. In earlier work we identified rationally motivated manipulation opportunities in the BitTorrent file distribution protocol [Coh03]. That work considers two classes of users: those concerned with maximizing distribution speed and those concerned with minimizing network bandwidth. A designer may decide that the negative impact of network bandwidth-minimizing participants is expected to be low and the cost to address these behaviors is relatively high. A revised protocol might address only the rational compatible manipulations that originate from nodes that seek to maximize distribution speed.

The selected rational compatible behavior(s) become the target audience of a properly constructed *mechanism.* The designer's goal is to provide incentives for correct behavior to this set of participants.

4.2.2 Specify a Mechanism and Suggested Strategy.

The concepts of the suggested strategy and mechanism design were introduced in Sections 3.4 and 3.6. The specification embeds a social goal and likely provides incentives for correct behavior from rational compatible participants. There are several tools that can be used to specify the mechanism: a system designer familiar with state-based specifications [Lyn96] might lay out the strategies and outcomes in terms of a finite state machine and then identify the suggested strategy as a particular path through the machine. However, a full definition of the mechanism may be prohibitively complex; the strategy space could be infinite. There are two combinable approaches to specifying a mechanism in the face of a large strategy space:

The first approach is to restrict the allowed strategy space Σ^m, either weakly by fiat or strongly by system "features" that restrict participants from performing certain actions. For example, a designer could use trusted networking hardware (as in Perrig et al. [PSST01]) to force participants to use a limited vocabulary when sending messages.

The second approach is to produce a full specification that favors a few strategies and maps other strategies into a "catch-all" outcome. For example, the designer may propose a suggested strategy for all participants that is shown to be utility maximizing for all participants. The designer would then map all other strategies to an outcome where participants receive strictly less utility.

4.2.3 Prove Specification Faithfulness

A proof of specification *faithfulness* is a certification that rational nodes will choose to follow the algorithm specified by the system designer. Faithfulness is as important as other systems correctness properties of *safety* and *liveness* [Lyn96] in settings that contain rational nodes. This certification is given for a solution concept and a particular environment. To achieve faithfulness, a system designer must create a mechanism where the suggested strategy s^m is equivalent to an equilibrium profile behavior s^* chosen by rational nodes.

- To show weak faithfulness, one proves that any combination of deviations by a single player from s^m will not increase that rational participant's utility.
- To show strong faithfulness, one proves that any combination of deviations by a single player from s^m will strictly decrease a node's utility.

An interesting aspect of real system design is that in many cases, there is a default software program or official protocol specification that is supplied to clients. A system designer can use this software to convey the suggested behavior that it wishes a participant's node to follow. In effect, a default client *is* s^m. This observation allows the mechanism designer to suggest an equilibrium strategy or to coordinate an equilibrium if there are many potential equilibria in a mechanism.

Definition 4.1 (faithfulness). An implementation is a **faithful implementation** of the specification in a particular solution concept and environment when the suggested strategy s^m is equivalent to an equilibrium profile strategy s^* chosen by all compatible rational nodes in a mechanism implementation.

4.3 Evaluate Effectiveness, Impact, and Cost.

It makes sense for a system designer, throughout the design process, to weigh the trade-offs of building a system that is tolerant to rationally motivated failure. In this step, the designer evaluates the system implementation to construct a cost-benefit analysis of a robust system, as compared to a non-robust system. A robust system probably has a higher cost when compared with a non-robust system. For example, participants that follow a faithful specification may need to expend more computation, utilize more bandwidth, etc. The details of such an analysis are problem specific, and we give two examples in Chapters 5 and 6.

4.4 Proving Specification Faithfulness

Having stepped through the methodology for addressing rationally motivated failure, we spend the remainder of this chapter explicating the middle step, where one must *prove* that a specification prevents rationally motivated failure.

4.4.1 Useful Properties: CC, AC, and IC

We introduce *communication- and algorithm compatibility* as properties that are necessary when showing that a mechanism avoids failure from rational compatible participants. We also translate the idea of *incentive compatibility*, found in the traditional mechanism design literature, into a definition suitable for a distributed environment. Recall that earlier in the chapter we decomposed the suggested strategy for each participant i, s_i^m, into into $s_i^m = (r_i^m, p_i^m, c_i^m)$, with information-revelation strategy r_i^m, message-passing strategy p_i^m, and computational strategy c_i^m

Definition 4.2. A distributed mechanism specification $m = (g, \Sigma^m, s^m)$ is **incentive compatible** (IC) when there exists a particular equilibrium in which participant i cannot receive higher utility by instructing its node to deviate from the suggested type-restricted information-revelation strategy components of $r_i^m(\theta_i)$, for all nodes i and all types θ_i.

The concept of incentive compatibility in a centralized mechanism with a *dominant strategy* solution concept is known as a *strategyproof* mechanism. In this mechanism, each participant i's strategy $s_i(\theta_i)$ is simply to report its type θ_i, and $g(\cdot)$ is equal to the social choice function $f(\cdot)$.

Definition 4.3 (strategyproof). A centralized mechanism $m = (f, \Theta)$ is **strategyproof** if $u_i(f(\theta_i, \theta_{-i}); \theta_i) \geq u_i(f(\theta_i', \theta_{-i}); \theta_i)$ for all θ_i and θ_{-i}, all $\theta_i' \neq \theta_i$, all $\theta_i, \theta_{-i}, \theta_i' \in \Theta$.

We note that a distributed notion of a *strategyproof* mechanism is not useful for the same reasoning given earlier as to why a distributed mechanism with a *dominant strategy* solution concept is not useful. Namely, it will almost never be the case that one strategy of information revelation will be the utility-maximizing strategy, regardless of the game implemented by other participants. However, as we will see later, it *will* be useful to prove specification faithfulness by starting with a strategyproof centralized mechanism, which is then distributed and thus loses any claim of dominant strategy equilibrium.

Cross-Field Connection: Readers familiar with mechanism design will recognize *incentive compatibility* from traditional mechanism design. In traditional mechanism design, an incentive-compatible mechanism is one that has an equilibrium strategy profile where $s_i(\theta_i) = \theta_i$ for all agents $i \in n$. In other words, an incentive-compatible mechanism is one where every agent follows a truth-revealing strategy and simply reports its type information.

In our distributed mechanism design-friendly definition above, we have translated the intention of centralized *incentive compatibility* into the distributed environment by restricting the term to describe type-restricted information revelation actions. Our definition also extends the idea of incentive compatibility to allow for incremental type revelation, rather than an all-at-one declaration of type.

This thesis will not address systems with information revelation actions that are *not* type-restricted. It seems unlikely that a designer can provide incentives for a participant to reveal such information when that participant receives the same utility whether or not the participant follows the suggested (not type-restricted) information revelation strategy.

Definition 4.4. A distributed mechanism specification $m = (g, \Sigma^m, s^m)$ is **communication compatible** (CC) when there exists a particular equilibrium in which node i cannot receive higher utility by deviating from the suggested message-passing strategy $p_i^m(\theta_i)$, for all nodes i and all types θ_i.

CC means that a rational node will choose to participate in the suggested message-passing actions within the distributed-mechanism specification.

Definition 4.5. A distributed mechanism specification $m = (g, \Sigma^m, s^m)$ is **algorithm compatible** (AC) when there exists a particular equilibrium in which node i cannot receive higher utility by deviating from the suggested computational strategy $c_i^m(\theta_i)$, for all nodes i and all types θ_i.

AC means that a rational node will choose to participate in the suggested computational actions within the distributed-mechanism specification. We note that in each one of the definitions of AC, CC, and IC, we allow for deviations in other aspects of the strategy. When combined, properties IC, CC and AC are required for a faithful distributed implementation. Moreover, IC, CC and AC are *sufficient* for a faithful implementation:

Proposition 4.1. *A distributed mechanism specification $m = (g, \Sigma^m, s^m)$ in which suggested strategy $s^m = (r^m, p^m, c^m)$ is IC, CC and AC in the* **same** *equilibrium will yield a faithful implementation from all rational compatible participants, and will lead to the outcome $g(s^m(\theta_i)) \in O$ for all participating θ_i.*

Proof. IC, CC and AC provide for the existence of an equilibrium in which nodes will follow suggested information-revelation r_i^m, and similarly for message-passing p_i^m and computation c_i^m. To achieve faithfulness we simply need that there is an equilibrium that achieves each one of these simultaneously. □

4.4.2 Strong AC and Strong CC

If participants in a distributed mechanism can be limited in their information-revelation actions to follow type-restricted information revelation, then there is a useful proof technique that applies when the distributed algorithm has a corresponding strategyproof centralized mechanism. By a corresponding mechanism, we mean that there is a well-defined mechanism where $g(\cdot)$ can be implemented by a center, and where participants can only participate in information revelation to the center. The systems considered later in this thesis in Chapters 5 and 6 do in fact ensure a phase of type-restricted information revelation. In this phase, messages are interpreted as claims of type, and are restricted from being interpreted as more general information revelation actions. In both of the systems considered later in this thesis, it is not profitable for a node to execute other types of actions until a first type-restricted information revelation phase is certified by a trusted node.

For these and similar systems, we define *strong*-AC and *strong*-CC, and show that together with the strategyproofness of the corresponding centralized mechanism, *strong*-AC and *strong*-CC provide IC, and in turn a faithful implementation. In this setting, we can reduce the problem of proving faithfulness in a particular solution concept to that of:

1. Demonstrating that the corresponding centralized mechanism is strategyproof.

2. Strong-CC: a rational node should *always* follow its suggested message-passing strategy (whatever its information revelation and computational actions) in a particular solution concept.

3. Strong-AC: a rational node should *always* follow its suggested computational strategy (whatever its information revelation and message-passing actions) in a particular

solution concept.

Definition 4.6. A distributed mechanism specification $m = (g, \Sigma^m, s^m)$ is **strong-CC** if a node cannot receive higher utility by deviating from the suggested message-passing actions $\hat{c}_i$, **whatever** its computational and information-revelation actions in a particular solution concept.

Definition 4.7. A distributed mechanism specification $m = (g, \Sigma^m, s^m)$ is **strong-AC** if a node cannot receive higher utility by deviating from the suggested computational actions $\hat{p}_i$, **whatever** its message-passing and information-revelation actions in a particular solution concept.

Taken together, the strategyproofness of a centralized mechanism, strong-CC, and strong-AC rule out any useful joint deviations in which a participant directs its node away from s^m by a combination of actions.

Proposition 4.2. *A distributed mechanism specification $m = (g, \Sigma^m, s^m)$ elicits a **faithful** implementation of $g(s^m(\theta))$ from compatible rational participants when the corresponding centralized mechanism is strategyproof and when the specification is strong-CC and strong-AC.*

Proof. To prove that the specification is an equilibrium we first assume that the strategies used by other nodes follow the behavior assumed by a particular solution concept. For example, in ex-post Nash, we assume that every node except i is following suggested specification, s^m_{-i}. As another example, In 1-partial ex post Nash, we assume that at least one other node j besides node i is following s^m_j.

By strong-CC and strong-AC shown of the mechanism in this solution concept and environment, a rational node i will follow the suggested message-passing and computation actions. (Notice that we can rule out joint deviations of both message-passing and computation actions). To prove IC, we can now safely assume that other nodes follow the strategies dictated by the chosen solution concept. Let $f(\theta) = g(s^m(\theta))$ denote the outcome rule in the corresponding centralized mechanism. By definition of type-restricted information-revelation actions, the space of possible outcomes becomes $g(s^m_i(\hat{\theta}_i), s^m_{-i}(\hat{\theta}_{-i}))$, but $g(s^m_i(\hat{\theta}_i), s^m_{-i}(\hat{\theta}_{-i})) = f(\hat{\theta}_i, \hat{\theta}_{-i})$, and $u_i(f(\theta_i, \hat{\theta}_{-i}); \theta_i) \geq u_i(f(\hat{\theta}_i, \hat{\theta}_{-i}); \theta_i)$ for all $\hat{\theta}_{-i}$, all θ_i, and all $\hat{\theta}_i \neq \theta_i$ by strategyproofness of $g(s^m(\theta)) = f(\theta)$. □

Remark 4.1. *In applying Proposition 4.2 one must be careful to ensure that actions labeled as "type-restricted information revelation" within the suggested specification satisfy the technical requirement of consistent information-revelation which can require consistency* *checking. Consistent information revelation simply means that a node does not claim to be θ_i to one participant and conflicting type θ_j to another participant, and that some algorithmic consistency checking is required to catch this situation.*

4.4.3 Faithfulness Proof Tools

For faithful adherence to a specification, we need to demonstrate strong-CC and strong-AC as well as the consistency of information-revelation actions. The following approaches are useful to this end:

Break into Phases

A distributed mechanism can be decomposed into *disjoint phases* that are individually proved to be strong-CC and strong-AC without considering joint deviations that involve actions from other phases. Phases can be separated during runtime with *checkpoints* where some (perhaps trusted) node certifies a phase outcome and the start of a subsequent phase. One must be sensitive to the added computational and communication complexity in using checkpoints. This decomposition technique is powerful because it can allow an exponential reduction in the number of joint manipulation actions that must be checked in a faithfulness proof since each phase can be self-contained and checked for correctness before moving on to a subsequent phase.

Abstract Dependencies

A remedy may assume certain *external dependencies*. It is useful to describe these dependencies in terms of their logical features, rather than specifying the actual implementation. This abstraction is common in distributed systems, where *logical devices* such as "failure detectors" [CT96] are commonly included in specifications, but are not described in terms of their full *device realization*:

- The **logical devices** describe the ideal conceptual tools that are relied upon by a remedy. These devices are not implementations; they are ideas like "correct unicast" or "perfect identity service". Logical devices can further rely on other logical devices.
- The **device realizations** describe how one actually engineers a particular device. Realizations need not be "perfect"; a device realization that is "good enough" may be fine as long as it reasonably approximates the logical device specification. For example, a logical device may specify a "guaranteed encryption service." A realization that uses

a public key infrastructure may be acceptable even if the resulting device is not perfect and can be broken with a brute-force key attack.

The main advantage to the *logical device / device realization* abstraction is that one can prove a remedy in terms of logical devices without worrying about the details of the device realization. The proof of the remedy holds so long as the dependency realization is correct. Of course, if a remedy that relies on logical devices that cannot be realized or reasonably approximated is not terribly useful.

Design Restrictions

Design restrictions, also known in previous work as *problem partitioning* [PS04], move manipulation opportunities away from rational participants. For example, in the TCP hacking example introduced in Chapter 1, Savage et al. [SCWA99] propose that a message recipient must acknowledge a packet by echoing a specific number generated by the sender. In effect, the designer has restricted the set of manipulative strategies that can be effectively employed by a manipulative receiver. There is a design trade-off: an opportunity for manipulation is reduced but the sender must now keep track of what numbers correspond to what recently sent packets.

Another example of this technique is the distributed Vickrey Clarke Groves mechanism [PS04], where a distributed mechanism runs on a single set of participants, but the portion of the calculation performed by that participant never directly affects that participant's outcome. A more extreme way to achieve problem partitioning is to split a group of participants into two and run separate distributed mechanism computations, where one mechanism calculates the outcome for the other and *vice versa.*

Incentives

Incentives are rewards and punishments, deployed into or alongside a system, that elicit good behavior from participants. Incentives can be basic. In the aforementioned TCP hacking example, the incentive for echoing the sender's number (as opposed to echoing a random number, or not echoing a number at all) is that the receiver will be rewarded with the next packet in the sequence. A further incentive extension could deny service to receivers that attempt a "brute-force" echo of all possible "specific numbers." In-algorithm incentives work when participants will respond to such *internal* rewards and punishments. Incentives need not be internal; in the earlier Seti@Home example, the *external* leader-board score provides incentives for participation by tracking submitted work. External incentives must be carefully designed, since participants are asked to offset any participation cost with an outside benefit. Furthermore, participants may look for opportunities to achieve external rewards without correctly participating in the main system, as the earlier Seti@Home example demonstrates.

A good external incentive is a monetary payment. Currency payments are commonly assumed in mechanism design when building mechanisms that avoid information-revelation manipulation. Payments can also be used to provide incentives for nodes to perform faithful computation and message passing.

Redundancy

Redundancy in distributed systems is often used to detect faulty executions of a common calculation [Lyn96]. Redundancy in a network of rational nodes is a more powerful idea, since rational nodes that choose to act in a "faulty" fashion can be penalized with a negative payment large enough to ensure compliance with semi-private value revelation, a distributed calculation, or in message passing [MT99].

Explicit Obedience

It is sometimes realistic for designers to inject a small number of nodes running an unmodified correct specification implementation (e.g., Harkavy et al. [HTK98]). These nodes can be useful as equilibrium tie-breakers in settings with multiple equilibria. In some cases, the identity of these nodes need not be made public. [KSGM03].

Cryptographic Methods

Cryptography has been used to make deviations from a specified algorithm detectable [Bra02]. Furthermore, by signing or encrypting messages, it may not be rational to change a message. One problem with cryptography can be increased complexity: if a system relies heavily on this technique, computation and communication complexity can become prohibitive.

4.5 Bibliographic Notes

Faithfulness

Our earlier work on *faithfulness* [SP04] has been expanded into this thesis and of course attempts to provide tools and research to address rationally motivated failure. We are excited to report that other researchers have started to use our earlier work to build faithful systems. For example, Garg and Grosu [GG07] propose two protocols that are faithful implementations of the Shapley Value mechanism [Rot05], and actually build and evaluate these faithful protocols on Planetlab [PCAR02].

Models and Inferences

Much of the research in algorithmic mechanism design has focused on dominant strategy implementations, where participants assume that other participants are running the same algorithm, but assume nothing about the inputs that are fed into that algorithm.

Mu'alem [Mu'05] observes that the very nature of distributed environments allows algorithm participants to make inferences about the types of other agents. This is similar to our observation in this chapter that specification correctness and failure robustness must be proven with respect to a particular solution concept that depends on the environment assumptions. For instance, in a routing problem, the observed local connectivity graph can be used to make probabilistic assumptions about the types and strategies of other agents. Mu'alem argues that in such *partially informed environments*, solution concepts that are weaker than dominant strategy make sense since participants are partially informed. Mu'alem focuses on peer to peer file sharing, where a participant's observation of network traffic (e.g., Node A sent File X via me to Node C) is used to construct an equilibrium strategy in a trading game using an ex-post Nash solution concept. While Mu'alem's work acknowledges the threat of malicious players, the work only considers spiteful malicious players who exclusively seek to maximize the weighted difference of their utility and all other agents utility (so called *q-malicious*; cf. Brandt et al. [BSS07]).

Chapter 5

Rationally Motivated Failure in Interdomain Routing

In this chapter we apply the rationally motivated failure remedy methodology to an interdomain routing problem based on work by Feigenbaum et al. [FPSS02] (FPSS). The enhanced algorithm is shown to be faithful under the ex post Nash solution concept. We implement the algorithm in simulation and compare the message complexity of the resulting algorithm to FPSS. We show that a node's message cost when running the faithful algorithm depends on its degree (number of neighbors in the connectivity graph) and that on a real Internet topology a node may incur a 2x-100x message traffic increase. We show how this overhead can be reduced to 2x-10x (compared with the unfaithful algorithm) without serious connectivity consequences when high-degree nodes impose a cap on their number of neighbors.

5.1 Introduction

The Internet is composed of many *autonomous systems* (ASs), which are collections of machines and networks typically named for and under the control of one entity, such as Harvard University, U.C. Berkeley, Microsoft, etc. The general *interdomain routing problem* is defined as the problem of calculating routes between these autonomous systems. The solution to this routing problem on the Internet is embedded in the workings of the Border Gateway Protocol (BGP) [GR01]. BGP has evolved over the years, and many researchers have proposed variants and related protocols that address BGP shortcomings (e.g., security [vOWK07]) or advocate new functionality.

Inspired by this interdomain routing problem, Feigenbaum et al. (FPSS) [FPSS02] proposed a routing protocol that finds *efficient*, i.e., lowest cost, routes between nodes. Their work is loosely designed to fit within the simplified BGP model of Griffin and Wilfong [GW99]. The main contribution of their work is to advocate the inclusion of mechanism design ideas into routing protocols such as BGP to address rationally motivated failure occurring when participants lie about their transit costs. The FPSS work proposes a payment scheme that can be used to pay *transit nodes* for relaying traffic along lowest cost paths. Their protocol incorporates a Vickrey-Clarke-Groves mechanism [Vic61, Cla71, Gro73] to guarantee truthful cost declaration from rational compatible participants. Their resulting protocol is *incentive compatible* for routing cost declarations under certain assumptions, including the strong requirement that participants otherwise follow the distributed algorithm without deviation.

A Notation on Notation: Both in keeping with the language of FPSS and for the sake of brevity, we erase the distinction between human *participants* and their computational *nodes* for the rest of the chapter. We assume that every node is controlled by exactly one rational participant, and that each rational participant controls exactly one node, and that each participant communicates with other participants through its single node acting as a communication proxy.

The basic interdomain routing problem that we consider in this chapter is similar to the problem considered in FPSS. Like FPSS, we seek to build a distributed protocol for calculating *efficient* routes between nodes. However, our work differs substantially in the assumptions and guarantees of the protocol. We expand the set of behaviors that can be chosen by a node to include all aspects of computation and message passing. In addition to guaranteeing truthful routing cost declaration and correct lowest-cost path (LCP) computation by rational compatible participants, we make the following guarantees:

1. Correct LCPs will be used when it comes time to forward data messages. Transit nodes representing rational compatible participants correctly transit all message traffic when they are part of the true lowest cost path from source to destination.

2. Transit nodes are correctly paid for transiting message traffic.

There is still a reasonable set of rational behavior that we do not address and is relegated to the category of rational incompatible behavior. Our protocol cannot avoid rationally motivated failure due to *collusion* or a participant's willingness to take advantage of *off-equilibrium behavior* that occurs when other nodes exhibit unintentional failure. Like

FPSS, our mechanism makes the unrealistic assumption of a static environment where routing costs and paths never change. FPSS and our modified protocol cannot exist as part of a live system deployment because of these limitations. We view our protocol as a step in answering the open question posed by Feigenbaum et al. of how "to reconcile the strategic model with the computational model" in that it shares the same incentive compatibility properties of FPSS while adding additional algorithm- and communication-compatibility with an expanded model of participant behavior. We compare the message cost of our new protocol to that of FPSS by building both protocols in a custom network simulator that simulates the protocols on actual historical AS topology graphs. We show that a node's message cost when running the faithful algorithm depends on its degree and that on a real Internet topology a node may incur a 2x-100x message traffic increase. We show how this overhead can be reduced to 2x-10x (compared with the unfaithful algorithm) without serious connectivity consequences when high-degree nodes impose a cap on their number of neighbors.

This chapter is presented as follows: We first describe the interdomain routing problem and briefly review the results of FPSS. We then introduce and prove the faithfulness of FPSS+, which is our modification to FPSS that does not assume correct computation or message passing. Both FPSS and FPSS+ are protocols to *establish* lowest-cost paths. We then introduce and prove the faithfulness of FPSS++ to provide incentives to *use* the routing paths correctly after they are established. We close the chapter with an evaluation of the three protocols as measured in simulation.

5.2 Problem Description

The goal in this chapter is to maximize network efficiency by routing packets along true lowest-cost paths (LCPs) for various traffic source-destination pairs. Each node incurs a per-packet *transit cost* for transiting traffic on behalf of other nodes. The cost represents the additional load imposed by external traffic on the internals of an individual node. It costs nothing for a node to transit a packet originating or terminating at that node.

Example. Figure 5.1 shows a small network. The LCPs from Z to all other nodes are drawn with bold lines. Numbers are the per-packet transit node costs incurred by each node. Assuming that the numbers in this figure represent true transit costs, the total LCP cost of sending a packet from X to Z is 2; the cost of sending a packet from Z to D is 1. The cost of sending a packet from B to D is 0 since there are no transit nodes between B and D.

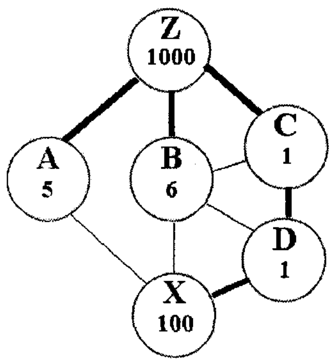


Figure 5.1: A small graph with routing costs for each node, and bold lines showing the lowest cost paths (LCPs) from Z.

To compensate a transit node for its routing services, each transit node is given a payment for carrying traffic. FPSS observes, however, that *"under many pricing schemes, a node could be better off lying about its costs; such lying would cause traffic to take non-optimal routes and thereby interfere with overall network efficiency."*

Example. In Figure 5.1, path X-D-C-Z is the lowest cost path between X and Z; if C declared a cost of 5, X-A-Z would become the LCP between X and Z. C can benefit from this manipulation, depending on the transit pricing scheme and expected traffic flows. Even if C no longer transits the X to Z traffic, it can make up the financial loss with higher payments received by transiting D to Z traffic. This damages overall efficiency - packets from X to Z are now being routed over a path whose true cost is higher.

The goal in the truthful lowest-cost routing problem is to promote correct LCP formation by designing a pricing and payment scheme where nodes receive the greatest utility when they declare their true transit costs.

5.3 FPSS Interdomain Routing

FPSS solves the problem of computing correct lowest-cost paths for a set of rational participants by using a distributed Vickrey-Clarke-Groves (VCG) mechanism [Vic61, Gro73, Cla71]. In the VCG mechanism, transit nodes are paid based on the utility that they bring to the routing system plus their declared cost. Their resulting protocol is *incentive compatible* for routing cost declarations under certain assumptions, including the strong requirement that participants otherwise follow the distributed algorithm without deviation.

5.3.1 Model

Network Model

A network is composed of *nodes* that are controlled by independent participants who share the goal of being able to send each other *data messages.* Two adjacent nodes, e.g. A and Z in Figure 5.1, can send data messages via their direct link. But when a *source* and a *destination* do not share the same link, e.g. A and B, these two nodes may still communicate by sending messages via one or more *transit* nodes. Transit nodes incur a per-message cost for transiting messages and can receive a payment from some other node(s) to cover this cost. The transit cost is the same regardless of message source, destination, or content. Data messages between two neighbors trivially do not use any transit nodes and do not incur any transit cost.

Neighboring nodes can also communicate with each other via special *control messages.* Control messages are used to set up and administer the routing algorithm. Control messages always terminate at direct neighbors and thereby never incur a transit cost.

FPSS makes these additional network assumptions: nodes are bi-connected and links are bi-directional so that there are at least two independent paths from any node to any other node. There are no networking faults or delays. The environment is static, meaning that nodes neither enter nor leave the system once the FPSS algorithm begins.

Participant Model

The FPSS algorithm assumes that every node can fully participate in the distributed algorithm. Functionally, this means that nodes have enough memory to hold the state information used in an abstract model of the Border Gateway Protocol (BGP) [GW99] and for the additional overhead required by FPSS. The additional state overhead in FPSS grows linearly with the number of nodes in the network. There is a set of assumptions that both restricts the participants and also defines the **knowledge model** that each participant in FPSS believes about other participants in the system:

- all nodes will perform all computation and message passing actions completely and correctly, and no actions outside of the specification will be performed by any node.
- all nodes' information-revelation actions will be type-restricted information actions.
- nodes are rational compatible individual participants and do not collude with other nodes.

Cross-Field Connection: A reader familiar with mechanism design will notice that the knowledge model in FPSS is similar to the knowledge model in a centralized strategy-proof mechanism, in that both models assume strategies limited only to type-restricted information revelation actions.

We define a rational compatible node to reflect the expected behavior of an AS: A node has the primary goal of establishing communication paths to other nodes, so that AS users can send data messages to any destination, and the secondary goal of maximizing:

((incoming payments for transiting data) - (actual cost for transiting data))
- (outgoing payments for originating data)

In FPSS, nodes can do nothing to reduce outgoing payments because of the requirement of correct computation and message passing actions. Nodes can control their incoming payments by changing their type-restricted information actions. Feigenbaum et al. show that truthfully revealing transit costs maximizes a node's utility given the two goals above.

Dependency Model

FPSS requires a set of "accounting and charging mechanisms [that] are used to enforce the pricing scheme." These logical devices are not detailed in the original paper.

5.3.2 FPSS Algorithm Overview

The structure of the FPSS algorithm is as follows:

- First, nodes publish a single routing cost that is supposed to indicate that node's true cost for routing data.
- Second, each node builds a local routing table based on its known destinations and on reachability declarations sent by neighbors. A node's local routing table contains the routes and routing cost from itself to every other node in the system. This routing table is published to neighbors and updated as reachability knowledge expands through claims made by immediate neighbors.
- Third, while routing table creation is occurring, a parallel local computation is run by every node i to calculate prices that node i owes transit nodes for transiting traffic that node i originates, for every possible destination. In other words, every node is responsible for calculating its payments due to transit nodes for all traffic that it originates. There is no payment due from node i for transiting traffic on behalf of

Phase	Message Sent by node X to neighbor node W
Phase 1: Cost Declaration	"I, neighbor node X, hear that node Y costs C"
Phase 2: LCP Table	"I, neighbor node X, declare my LCP to node Z to be through nodes $(Y_1, ...Y_{j-1})$"
Phase 3: Pricing Table	"I, neighbor node X: For some destination node Y: For each transit node K: X owes K payment p (FPSS Update)"
Phase 4: Execution	"I, neighbor node X, wish to *originate* packet w to destination node J".
Phase 4: Execution	"I, neighbor node X, wish to *transit* packet w to destination node J on behalf of source node I."

Table 5.1: Messages sent in the FPSS algorithm, organized by phase.

another node, and node i is actually due a payment for such transit work from the original sender of the data traffic.

- Once the routing and pricing tables are established, nodes can begin to send actual data traffic and are supposed to keep track of payments and periodically report these to an accounting and charging mechanism.

We find it useful to describe FPSS in terms of three **construction** phases. We call these phases "construction phases" because they are constructing the LCPs and pricing tables that will be used in a later **execution** phase. We note that FPSS (and our FPSS+ extension) only address rationally motivated failure that may occur in the construction phases. Later in the chapter, FPSS++ will address rationally motivated failure in execution.

The goals in the construction phases are to establish a routing network by building a transit node cost list, which is then combined with reachability announcements to establish routing and pricing tables. The three construction phases are treated as overlapping calculations in the original FPSS paper, but our alternate description will prove useful in building our later algorithm extensions. Furthermore, breaking FPSS into phases makes sense as each construction phase has a goal that must be accomplished before moving on to the next part of the algorithm. Table 5.1 details the messages that are sent in each of these phases. The phases run as follows: Nodes broadcast and relay transit-cost information during the **first construction phase**. In the first construction phase, nodes make a claim about their routing transit cost to their neighbors. These neighbors in turn relay the transit costs to their neighbors, until each node in the network has the same vector of transit cost data. Nodes compute LCP routing tables during the **second construction phase**. In the second construction phase, nodes make LCP routing claims to their neighbors. These claims inform neighbors of currently known lowest-cost routes to other destinations. This calculation is performed iteratively, reaching stability in a number of rounds that depends on the length of the longest path, a.k.a. diameter, or the network. Nodes compute pricing tables during the **third construction phase**. In the third construction phase, nodes perform pricing calculations that determine how much that node should pay each other nodes when originating message traffic. Like the LCP routes, this process is performed iteratively, reaching stability in a number of rounds that depends on the diameter of the network. Nodes originate, route, and deliver actual data messages during the **execution phase**. FPSS suggests that in the execution phase every node i should keep track of how much it owes transit nodes as node i originates data messages, and that transit nodes should send data messages along the LCP paths.

Within the construction phases, we classify the declaration of transit costs and connectivity information as type-restricted information-revelation actions. The relaying of other nodes' transit cost announcements are message passing actions. Updating and forwarding routing and pricing tables are computation actions. While we have described the messages sent in each phase and the data that is calculated in each phase, we have not described the local algorithm that each node performs to turn messages into this data. This calculation is straightforward for the transit cost list, the routing tables, and the payment tables. As for the pricing tables, for our purposes it is sufficient to know that FPSS embeds a distributed VCG calculation. We refer interested readers who are not familiar with FPSS to Feigenbaum et al.'s original paper [FPSS02] to see how the pricing scheme is constructed so that type-restricted information revelation is truthful given a correct implementation of the remainder of the mechanism.

5.4 FPSS+: An Extension to FPSS

Feigenbaum et al. close their paper with the open problem that:

> On the one hand, we acknowledge that [participants] may have incentives to lie about costs in order to gain financial advantage, and we provide a strategyproof mechanism that removes these incentives. On the other hand, it is these very [participants] that implement the distributed algorithm we have designed to compute this mechanism; even if the [participants] input their true costs, what is to stop them from running a different algorithm that computes prices more favorable to them?

They go on to observe that:

> If [nodes] are required to sign all of the messages that they send and to verify all of the messages that they receive from their neighbors, then the [FPSS protocol]

can be modified so that all forms of cheating are detectable. Achieving this goal without having to add public-key infrastructure (or any other substantial new infrastructure or computational capability) to the BGP-based computational model is the subject of ongoing further work.

It is here that we pick up with our extensions to the FPSS protocol to bring correct computation and communication during the construction phases into equilibrium. This section presents a protocol called FPSS+ that relies on redundancy, incentives, and problem partitioning to ensure that participants *choose* to behave correctly. Neither message signing nor a public-key infrastructure are used to detect cheating. We still find that some signing infrastructure is useful if nodes do not have trusted communication to the referee/bank. Given certain network topologies, it may be possible to eliminate signing altogether.[1]

5.4.1 Model

The models that we assume in FPSS+ are refined from the model introduced for FPSS in the last section. Our goal in this refinement is to make the setting a bit more realistic. The FPSS+ environment is identical to that of FPSS with the following exceptions:

Participant Model

As in FPSS, we assume that every node can fully participate in the distributed algorithm. Our assumption of the goals of a rational compatible node is the same as before: a node has the primary goal of establishing communication paths to other nodes and the secondary goal of maximizing:

((incoming payments for transiting data) - (actual cost for transiting data))
- (outgoing payments for originating data)

The change comes in loosening the set of assumptions that both restrict the participants and also define the **knowledge model** that each participant in FPSS+ believes about other participants in the system:

- nodes are not restricted in their actions during construction phases.

[1]We can think of two examples where signed referee/bank messages can be safely eliminated. In the first example, the network of rational nodes runs as an overlay on top of an obedient underlay. This models how many peer to peer applications work today, where the application software is untrusted, but where the underlay is obedient. Any messages sent by nodes to the referee/bank follow the obedient underlay. In the second example, messages to the referee/bank travel over rational transit nodes that are that are guaranteed to have no interest in the message. A special case of this scenario is assumed by related work [Bra02] that requires a fully connected communication graph so that there are no intermediate nodes to worry about.

- nodes are selfish and do not collude with other nodes.
- nodes correctly use the LCPs and pricing tables in the execution phase.

The main advantage of FPSS+ over FPSS is that participants' computation and message passing actions are not assumed to be correct during construction phases. Rather, correct behavior will be part of a rational participant's utility-maximizing strategy in FPSS+.

Dependency Model

In FPSS+, we define the set of "accounting and charging mechanisms [that] are used to enforce the pricing scheme" [FPSS02] in FPSS as a requirement for a logical *referee service* and *bank service*. We must also guard against spoofing by relying on a *message authenticity service.*

The **logical referee service** fulfills the *enforcement* role of "whatever accounting and charging mechanisms [that] are used to enforce the pricing scheme." The referee in FPSS+ is a trusted and obedient entity that can perform simple comparisons and enforce penalties through bank transfers when it detects a problem. The details of this referee are given below as part of the FPSS+ algorithm. The referee is not as powerful as the traditional mechanism center: the referee does not actually perform the distributed mechanism computation and instead only compares results calculated by others. The details of deviation detection are phase specific. In the construction phases, the referee penalizes deviant nodes by preventing them from moving to the next phase of the algorithm.

The **logical bank service** fulfills the *bookkeeping* role of "whatever accounting and charging mechanisms [that] are used to enforce the pricing scheme." The bank is characterized by an interface that allows transfers of currency from one participant to another. We intend the referee service to be the only entity that directly communicates with the bank. In practice, the node that realizes (implements) the referee logical service may also realize the bank.

The **logical message authenticity service** ensures that a message directly received by node i marked as originating from node j did indeed originate at node j. This logical service is required by each node to calculate correct table information and is required by the logical referee and bank services. We note that the logical message authenticity service need *not* stop a lie: node j can send a message to node i that appears to be a relayed communique from node k. The only guarantee made by the logical message authenticity service to node i is that the message in question was last sent by node j. A stronger message authenticity service could require and be built around a *message signing service.* However, for simplicity we use the approach assumed in FPSS, where the message authenticity service is realized through direct trusted network connections, and that the network underlay plays a role in identifying and authenticating messages.

Finally, our proof makes use of hash table comparisons. We assume that a suitable hashing function is used so as to drive to zero the probability of a node being able to construct a manipulable hash table collision.

Network Model

We assume that every node has a correct connection path to the logical referee service, which in turn has a connection with a *logical bank service.* For simplicity, we assume that these bank and referee connections are trustworthy, which can be justified in practice by a secure communication channel.

5.4.2 Checker Nodes

In extending FPSS to FPSS+ we introduce the *checker role* for nodes already participating in the distributed algorithm. We stress that we are not adding new nodes, but rather giving additional tasks to existing nodes. The assignment of the checker nodes is very important: every neighbor of a node is assigned as a checker for that node. The node that is being checked is known as the *principal* P, to refer to its role in the core distributed algorithm. Every node in the network plays the role of both a principal node and a checker node for all of its neighbors. By the FPSS bi-connected topology assumption, every principal has at least two checker nodes.

Nodes in this checker role are supposed to mirror the principal's computation actions. A difference between the principal and the checker is that the checker does not send outputs of computations to neighbors, perform information revelation, or execute message passing actions. Rather, a hash-value of the output of the computation may be periodically sent to a special *referee* node whose role is to look at a set of supposedly equal hash-values and identify a discrepancy. The checkers allow the referee to catch and punish nodes that do not behave correctly in the distributed algorithm.

Each checker node is supposed to perform the internal computation of P based on copies of P's messages that it receives from P. We must establish that the checker will follow the checking behavior in equilibrium with a faithful P. This will be true in FPSS+ because of problem partitioning — a checker cannot individually benefit from allowing a deviation by P and may be worse off by allowing the deviation. Moreover, we must show

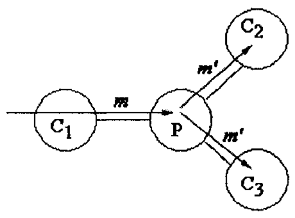


Figure 5.2: A computation or message passing deviation by principal node (P) can be detected by checker nodes.

that P cannot benefit from changing or dropping the messages sent to each checker. Perhaps surprisingly, we guarantee this property without the use of message signing or encryption: because of the network arrangement of the checkers, at least one of P's checker's calculations will differ from the other checkers and cause P to be punished if P misbehaves. It is in P's best interest to ensure that each checker receives a correct copy of every message that P receives from neighbors. This claim can be understood by examining the graph shown in Figure 5.2.

Example. Figure 5.2 illustrates how nodes C_1, C_2 and C_3 acting as checker nodes can monitor principal P. In that network, P is supposed to forward m to nodes C_2 and C_3 to allow them to replicate P's calculations, but let us say that P deviates and forwards the message as m'. This deviation will change the view that C_2 and C_3 have of P's calculation input, but as checker C_1 was on the incoming path of m, it still has the correct view of m. The resulting differences in the calculations performed by P, C_1, C_2 and C_3 are sufficient to catch a deviation, as we will see later when we discuss the referee actions.

5.4.3 FPSS+ Algorithm Overview

Mirroring our explanation of FPSS, FPSS+ contains three construction phases. We list the principal node external actions [PA#], checker node external actions [CA#], and referee actions [RA#] for each phase. Actions are labeled with shorthand for ease of reference in our faithfulness proof later in the chapter. Principal and checker actions are further categorized into their information revelation, computation, and message passing components. Referee and bank behavior are assumed to be correct and therefore is not checked.

We describe the workings of the algorithm in terms of the three construction phases and one execution phase in the following sub-sections.

First Construction Phase

Phase Goals: Establish uniform transit cost list. At the end of this phase, every node has an identical list of node-cost pairs.

[**PA0**] **On startup:** *Information Revelation:* Send truthful private transit cost to neighbors.

[**PA1**] **On receiving transit cost update from neighbor:** *Computation:* Check if update would cause a transit cost discrepancy, where two different transit costs have been received for the same node. If discrepancy exists, set discrepancy flag and wait to be polled by referee. If no discrepancy, update local transit cost list and continue. *Message Passing:* Forward complete transit cost information to all neighbors;

[**RA1**] **Periodically**, ask nodes requesting the hash of their transit cost list and a discrepancy flag. Once the referee detects that the node reports are consistent and quiescent, send a message to all nodes initiating next phase. If a discrepancy flag has been raised, restart the algorithm.

The first construction phase does not need to use checker nodes, since checker nodes are not used to check information revelation actions (i.e., [PA0]), and every node is capable of detecting transit cost discrepancies (i.e., [PA1]) in their own roles as principals.

Second Construction Phase

Phase Goals: Each node establishes its correct routing table. Each node's routing table specifies the route that a packet should follow for each destination from this node.

[**PA2**] **On receiving routing table update from neighbor:** *Message Passing:* Forward message to all checkers. *Computation:* Recompute LCPs based on new information; send recomputed LCPs as a routing table update to all neighbors.

[**CA2**] **When the principal forwards a routing update:** *Computation:* Verify that declared LCP is correct with local cost information. Re-run the LCP routing update.

[**RA2**] **Periodically**, ask all principals and checkers for hashes of their routing tables and check for a deviation. If there is a deviation in hashes, the referee restarts the algorithm from routing table calculation phase. Once the referee detects that the node reports are consistent and quiescent, send a message to all nodes initiating next phase.

Third Construction Phase

Phase Goals: Each node establishes its correct pricing table. Principal i's pricing table contains the amount of money that principal i should pay each transit node when principal i originates a packet, for any destination.

The biggest change in the third construction phase when moving from FPSS to FPSS+ is that each entry in the pricing table is augmented with an *identity tag.* This tag identifies the node that triggered the most recent pricing table update. (In the case of a pricing tie, this tag field contains the union of the nodes that propagated the same pricing entry.)

[PA3] On receiving pricing table update from neighbor: *Message Passing:* Forward message to all checkers. *Computation:* Recompute pricing tables based on new information; update tag information for every changed pricing entry to reflect source of change; send new pricing tables to all neighbors.

[CA3] When the principal forwards pricing tables: *Computation:* Ignore messages with identity tags that are not checker nodes of the principal; re-run the pricing table computation.

[RA3] Periodically, ask all principals and checkers for pricing table information and check for a deviation. If there is a deviation then restart the implementation from the pricing-table calculation phase. Once the referee detects that the node reports are consistent and quiescent, send a message to all nodes initiating the execution phase.

Execution Phase

As the third construction phase ends, LCP paths have been computed, and every node i knows the exact amount that it owes to every transit node j each time node i originates and sends a packet to some destination node k. The execution phase is so named because nodes now execute the main task of routing data packets along LCPs. The execution phase runs indefinitely. As in FPSS, we assume in FPSS+ that there is no failure in the execution phase. This assumption will be relaxed in FPSS++ discussed below.

5.4.4 FPSS+ Faithfulness Proof

A proof of FPSS+ specification *faithfulness* is a certification that rational compatible nodes will choose to follow the FPSS+ algorithm without modification. We now follow the methodology given in Chapter 4 to prove faithfulness. We defined rational compatible node behavior in in Section 5.4.1, and note that the participant model effectively defines the node strategy space Σ^m. In the last section, we stepped through the suggested strategy s^m in detail for nodes, breaking out expected node behavior into actions taken as a principal and actions taken as a checker. Simultaneously, we defined the intended mechanism by defining the system outcome g that occurs when nodes follow the suggested strategy – namely, that nodes will form correct LCPs and correct pricing tables.

Now we must now show that the FPSS+ suggested strategy s^m corresponds to the equilibrium strategy picked by rational compatible nodes, given an appropriate solution concept. The solution concept that we adopt is *ex-post* Nash, which is defined in Section 3.6.3. In an *ex post* equilibrium no node would like to deviate from its strategy even if it knows the private type information of the other nodes. Thus, as designers we can be agnostic as to whether or not nodes have any knowledge about the private type of other nodes. The main assumption when adopting ex post Nash is that the rationality of nodes is common knowledge amongst nodes. This corresponds with the knowledge model asserted in Section 5.4.1. The FPSS+ faithfulness proof will rely on tools introduced in Chapters 3 and 4 of this thesis, and specifically on Proposition 4.2, repeated here:

Proposition 4.2 (repeated). *A distributed mechanism specification* $m = (g, \Sigma^m, s^m)$ *elicits a* ***faithful*** *implementation of* $g(s^m(\theta))$ *from compatible rational participants when the corresponding centralized mechanism is strategyproof and when the specification is strong-CC and strong-AC.*

For each construction phase, we must show strong-AC, strong-CC, and consistent information revelation irrespective of a nodes behavior in other construction phases of the mechanism. Once this is shown for each construction phase, and with the assumption of strong-CC and strong-AC in the execution phase, we use Proposition 4.2 to show that the entire mechanism is faithful.

The key in proving proper behavior in these construction phases is the reliance on a rational compatible node's desire to reach the execution phase, and in the tension that naturally arises between nodes acting in the principal role and neighboring nodes that act as checkers. In proving faithfulness in any algorithm, one must worry about *joint deviations* where a node can selectively fail at multiple steps of the algorithm so that the combined failure benefits the node. We must worry about any joint deviations within a phase, but since the algorithm checks for correct behavior at the end of each phase, the opportunity for any such combined deviation is virtually eliminated.

First Construction Phase

We must show that [PA0] and [PA1] are correct. In demonstrating [PA1], we must show that a principal cannot benefit from dropping, changing, or spoofing another node's transit cost information [PA1, message passing] and that a principal cannot benefit from ignoring a detected transit cost discrepancy [PA1, computation].

Proposition 5.1. *A node cannot benefit by revealing inconsistent, non-truthful, or incomplete cost information about itself to its neighbors in equilibrium. [PA0]*

Proof. First, we observe that a node has no incentive to declare a private transit cost that is above or below its true transit cost. The proof of truthfulness of the distributed VCG mechanism is given in Feigenbaum et al. [FPSS02], but to summarize: Nodes that run a distributed VCG mechanism calculate their payment to some on-LCP node i as node i's declared cost plus the value that node i brings to the system for being on the LCP. Node i's raising or lowering of its transit cost is at best a zero-sum game: for every dollar that node i lowers its declared transit cost, node i brings a dollar more of value to the system. For every dollar that node i raises its declared transit cost, node i subtracts a dollar of value from the system, until at some point it is no longer on the LCP and receives no payment and transits no traffic. By the same logic, node i has no incentive to report its own transit cost inconsistently, since even if other nodes somehow used differing transit costs for node i, the payment due to node i is always the same.

Node i has no incentive to refuse to report its own transit cost to any subset of neighbors. Other nodes establish neighbor connectivity information from transit cost announcements. A neighbor will refuse routing service to any neighbor that has not declared a connection, since the neighbor can neither expect nor enforce payment from an initially hidden node. Moreover, by hiding during the cost announcement, node i cannot be a transit node on any LCPs and thus (weakly) cannot maximize its utility.

□

Importantly, there is no joint deviation possible between [PA0] and later phases: a node cannot increase its received payment for transiting traffic for *any* combination of its own computation or information revelation actions in *any* of the construction phases, beyond what it would receive by declaring its true transit cost. This is because other nodes pay no attention to pricing updates for some node i when calculating node i's transit payment. (For the curious, this form of problem partitioning can be seen in Section 6 of Feigenbaum et al. [FPSS02]. While a pricing update message has the potential to trigger a series of pricing table updates on various nodes, each of these nodes ignores the node that caused the update.)

Proposition 5.2. *A node cannot benefit by refusing to report a discrepancy to the referee in equilibrium. [PA1, computation]*

Proof. By the referee's use of hash tables to compare transit cost results [RA1], all nodes' reported transit cost hash tables must be equal before the first construction phase can end. By Proposition 5.1, some node i will never benefit from agreeing to a hash table built on misreporting node i's transit cost. Node i will rationally choose to report the discrepancy unless the misreporting does not change node i's transit involvement with all LCPs, in which case neither node i nor any other node either benefits or suffers from the change. □

Any rational compatible node wishes to be on as many LCPs as possible in order to maximize any incoming transit payment. In order to join as many LCPs as possible, node i might like to change routing cost announcements so that routes without node i seem more expensive, and routes with node i seem cheaper.

Proposition 5.3. *A node cannot benefit by changing, dropping, or spoofing other nodes' transit cost information in equilibrium. [PA1, message passing]*

Proof. Because the network topology is bi-connected, a single node cannot definitively drop or change another node's transit cost announcement in that such an announcement will always flow along at least one other path. Moreover, by Proposition 5.2, the node responsible for publishing the transit cost path will itself always be weakly better off by disallowing such a change or omission. Any discrepancy in transit cost tables is caught by the referee in action [RA1], who will restart the phase if a cost discrepancy is not resolved.

We note that at this stage, a node i can introduce phantom nodes with made-up transit costs. However, this action has no benefit now, nor can it lead to a profitable joint deviation with any other phase of the algorithm, since a principal is prevented from declaring LCPs with false connectivity in the second construction phase. □

Proposition 5.4. *By the above propositions, the first construction phase is strong-AC and strong-CC.*

Second Construction Phase

We must show that [PA2] and [CA2] are correct. In demonstrating [PA2], we must show that a principal cannot benefit from dropping, changing, or spoofing another node's

LCP routing table updates [PA2, message passing], and that a principal cannot benefit from mis-computing or mis-reporting its own LCP routing table update [PA1, computation].

Proposition 5.5. *A checker cannot benefit from deviating from adopting an invalid LCP path on behalf of a principal in equilibrium. [CA2]*

Proof. A checker node has an innate ability to verify messages based on internal information that it itself established when acting as a principal. If the principal is correct, then no checker will choose to deviate in its calculations on behalf of the principal, since deviation would cause the phase to be restarted. If the principal is incorrect, then the principal must be consistent in reporting the incorrect calculation to its checkers, or else [RA2] will trivially detect the discrepancy and restart the phase. So if the principal is incorrect and consistent, then the only way the principal can benefit is if every node along the incorrect LCP agrees to the incorrect LCP; otherwise, the principal's packets will be dropped by the first node on the incorrect LCP path to disagree. By induction, this requires every checker on the incorrect LCP path to agree with the incorrect calculation. However, the checker will not do so because such an adoption would lead to *its own originated packets being dropped*, when acting in the role of principal, unless every node along the incorrect LCP agrees to the incorrect LCP. To see why this will never be true, consider some node j that was on some correct LCP for some source-destination pair, but because of neighbor node k's incorrect LCP calculation, is the first node in the incorrect LCP path from source to destination that is no longer on the LCP. By definition, node j is a checker for node k, and rationally would never agree to be dropped from the LCP path, since it loses the income from transiting packets. □

Proposition 5.6. *A principal cannot benefit from dropping, changing, or spoofing another node's LCP routing table updates in equilibrium [PA2, message passing]. A principal cannot benefit from mis-computing or mis-reporting its own LCP routing table update in equilibrium [PA2, computation].*

Proof. For any principal-destination pair, one checker must be on the shortest path from principal to destination. That checker has a correct view of the cost of that path (in its other role as a principal in the network), because each node has a local transit cost table by the end of the first phase.

Assume the behavior specified in [CA2]. All checkers ignore LCP information that is not judged correct through their local transit cost table established in their own role as a principal. The result is that a principal has no way to successfully change the LCP information stored by every checker. Any routing table deviation shows up by comparing a hash of LCP tables between the principal and its multiple checkers.

We further observe that a principal that insists on using an incorrect LCP in a later stage will have its packets discarded by the first node not on the true LCP, since off-true-LCP nodes are not paid for transiting traffic. □

Proposition 5.7. *By the above propositions, the second construction phase is strong-AC and strong-CC.*

Third Construction Phase

We must show that [PA3] and [CA3] are correct. In demonstrating [PA3], we must show that a principal cannot benefit from dropping, changing, or spoofing another node's pricing table updates [PA3, message passing], and that a principal cannot benefit from mis-computing or mis-reporting its own pricing tables [PA3, computation].

As before, faithfulness will depend on the tension between principals and their checker nodes. However, catching manipulations in the pricing table information is more subtle, since in the third construction phase a checker node does not have an innate ability to verify messages based on internal information that it already has in its own role as a principal. We note that while distributed VCG problem partitioning ensures that a principal has no reason to modify its newly created outgoing pricing update messages (since its utility is not affected by changes caused by these messages), a principal might like to change the pricing table that it must use for its own originated traffic.

Proposition 5.8. *A principal cannot benefit from dropping, changing, or spoofing another node's pricing table updates in equilibrium [PA3, message passing].*

Proof. First, assume the checking behavior specified in [CA3] is correct. Now, consider a principal A and a pair of neighbors B and C. Let A receive a pricing table update from B. We first note that if A drops this pricing table message received from B (rather than forwarding the message to C), an inconsistency results between pricing tables held by B and C, and therefore [CA3] will cause a phase restart. The same argument holds for changing, rather than dropping, an incoming message. If a principal attempts to spoof another node's pricing table update message, this spoof will create an inconsistency in checkers' identity tag information stored along with the pricing table. This inconsistency will be caught by [RA3]. □

Proposition 5.9. *A principal cannot benefit from mis-computing or mis-reporting its own pricing tables in equilibrium [PA3, computation].*

Proof. For some principal i to successfully compute and use an incorrect pricing table, it must convince all of its checkers to also use the incorrect pricing table. But the pricing update that triggered node i's re-computation has already passed through one of its checkers, and so any deviation by principal i will be known to at least one checker. The principal can cause checkers to use an incorrect pricing table only by spoofing a pricing table update message. By the same logic as in the last proposition, if a principal attempts to spoof another node's pricing table update message, this spoof will create an inconsistency in checkers' identity tag information stored along with the pricing table. This inconsistency will be caught by [RA3]. □

Proposition 5.10. *No checker will benefit from deviating from checking actions in equilibrium [CA3].*

Proof. No checker will deviate in pricing table calculation if the principal is correct in its calculation because deviation would cause the phase to be restarted. If the principal is incorrect but consistent in its pricing table calculation, it seems possible for the principal to fail in such a way as to benefit a checker if the checker lets the calculation pass as correct. (This is not active collusion, but the principal could claim to pay more to the checker and less to everyone else on the LCP path than it should.) However, this error can be caught by [RA3] because of a symmetry in pricing tables: the price that node i should pay to node j for transiting traffic to distant node k is exactly the same price that node k should pay to node j for transiting traffic to distant node i. Because of our assumptions against collusion, node i and node k have no way to coordinate the selection of an invalid pricing table. In this situation, accuracy is a coordination device to avoid phase restart. □

Proposition 5.11. *By the above propositions, the third construction phase is strong-AC and strong-CC.*

Theorem 5.1. *The FPSS+ specification is a faithful implementation of the VCG-based shortest-path interdomain routing mechanism in an ex post Nash equilibrium.*

Proof. All phases of this specification are strong-AC and strong-CC in an ex post Nash equilibrium, and all phases have consistent information revelation, and the corresponding centralized mechanism is strategyproof. By Proposition 4.2, FPSS+ elicits a faithful implementation of the VCG-based shortest-path interdomain routing mechanism. □

5.5 FPSS++: An extension of FPSS+

FPSS and FPSS+ address rationally motivated failure in the construction phases of an interdomain routing system. But once correct LCPs and pricing tables have been computed, what assurance do we have that correct packet routes and payments will be used? Our last extension addresses rationally motivated failure in the execution phase of the FPSS/FPSS+ algorithms.

5.5.1 Model

The network and dependency models in FPSS++ are the same as in FPSS+. However, we make an important change in the participant knowledge model by not restricting the participant behavior.

Participant Model

The set of assumptions that both restricts the participants and defines the **knowledge model** that each participant in FPSS believes about other participants in the system is as follows:

- nodes are not restricted in their actions.
- nodes are selfish and do not collude with other nodes.

5.5.2 FPSS++ Algorithm Overview

The first three construction phases are the same as in FPSS+. The change in FPSS++ comes in the execution phase. We add additional state to the system called the *packet transit count list.* This list stores a count of the number of packets transited through the principal by origin. (A packet that originates or terminates at the principal is not considered a transit packet.) A principal does not keep a packet transit count for itself, as this state information is only held by nodes in their role as checker.

Execution Phase

Phase Goals: Data packets are sent along lowest-cost paths and corresponding correct payment reports are sent to the referee.

Unlike previous phases, the execution phase does not end. Rather, the referee enforces the faithful payment and usage logs by auditing components of the execution, applying a provably effective penalty to any detected deviation.

[PA4] **On originating new traffic:** *Computation:* Increment entries in local payment table by appropriate transit price amount. *Message Passing:* Label the source and destination of the packet correctly and route the packet to the correct next hop of the LCP for the destination.

[PA5] **On receiving transit traffic:** *Computation:* Verify that the LCP for this packet is being followed and drop packet if verification fails; otherwise: *Message Passing:* Forward the traffic to the correct next hop of the LCP for the destination.

[CA4] **When principal-originated traffic is seen:** *Computation:* Increment entries in local copy of the principal's payment list (of transit nodes) by the appropriate transit price amount.

[CA5] **When principal-transited traffic is seen:** *Computation:* Update counts in the transit table (of traffic transited by the principal on a per-origin basis) appropriately: A packet that passes through the principal and then passes through the checker decrements the appropriate origin count by one. A packet that passes through the checker and then through the principal increments the appropriate entry in this list by one.

Referee actions in this phase are:

[RA4] **Periodically**, request the payment list from all principals and checkers. If the principal disagrees with the payment entries of any checker, then penalize the disagreeing nodes. The penalty is set to be ϵ (a small positive value) higher than the sum of all discrepancies in total payments.

[RA5] **Periodically**, request the transit count list information from all checker nodes. Check for a mismatch between the total number of packets into and out from a node. At this stage, if a deviation is found then ask a checking node for the maximal single-node VCG payment, which is then multiplied by the length of the longest route in the pricing table and charged to the principal.

5.5.3 FPSS++ Faithfulness Proof

We must show that [PA4], [PA5], [CA4], and [CA5] are correct. In demonstrating [PA4], we must show that a principal cannot benefit by incorrectly updating its local payment table or by not updating the payment table at all [PA4, computation]. We must also show that a principal cannot benefit from incorrectly labeling the source or destination of the packet or by routing the packet differently than the correct next hop [PA4, message passing]. In demonstrating [PA5], we must show that a principal cannot benefit from forwarding a packet that it is not supposed to forward [PA5, computation] and that the principal cannot benefit by dropping, changing, or spoofing message traffic that is supposed to be delivered to the next hop [PA5, message passing].

Proposition 5.12. *A principal cannot benefit by incorrectly updating its local payment table or by not updating the payment table in equilibrium [PA4, computation].*

Proof. Assume [CA4] is performed correctly. Then, any deviation in the local payment table will be caught by [RA4]. The penalty imposed in [RA4] is constructed to be greater than any benefit that the principal may have received from a successful deviation. □

Proposition 5.13. *A principal cannot benefit from incorrectly labeling the source or destination of the packet, or by routing the packet differently than the correct next hop in equilibrium [PA4, message passing].*

Proof. Assume [PA5] is performed correctly by other nodes on the LCP path between the source and destination. Then, any modification to the source or destination of the packet that changes the LCP will trigger the packet being dumped by the transit node that detects the change, since the transit node can neither expect nor enforce payment for transiting the packet. The same argument applies for a principal that initially routes the packet to a node other than to the correct next-hop. A principal i might try to change the source tag of the packet, in an attempt to make it seem like node i is transiting and not originating the packet. But this behavior will be caught by [CA5] and [RA5] – namely, the transit count list will contain a mis-match. The penalty imposed in [RA5] are constructed to be greater than any benefit that the principal may have received from a successful deviation. □

Proposition 5.14. *A principal cannot benefit from forwarding a packet off the LCP path, rather than forwarding the packet along the LCP path in equilibrium [PA5, computation].*

Proof. In FPSS and its derivatives, the transit cost incured by a node is the same regardless of the destination of the message. Moreover, this behavior will be caught by [CA5] and [RA5] – namely, the transit count list will contain a mis-match and trigger a penalty from [RA5]. □

Proposition 5.15. *A principal cannot benefit by dropping, changing, or spoofing message traffic that is supposed to be delivered to the next hop in equilibrium [PA5, message passing].*

Proof. Changing message content is not beneficial. In FPSS and its derivatives, the transit cost incured by a node is the same regardless of the content of the message. A drop or spoof will be caught by [CA5] and [RA5] – namely, the transit count list will contain a mis-match and trigger a penalty from [RA5]. □

Proposition 5.16. *A checker cannot benefit by incorrectly incrementing entries in its local payment list by the appropriate transit price amount in equilibrium [CA4].*

Proof. Assume [PA4] is performed correctly. Then, any deviation in the local payment table will be caught by [RA4]. The penalty imposed in [RA4] are constructed to be greater than any benefit that the principal may have received from a successful deviation. □

Proposition 5.17. *A checker cannot benefit by incorrectly updating entries in the transit table in equilibrium [CA5].*

Proof. The transit table has no effect on the utility of a checker node. □

Proposition 5.18. *By the above propositions, the execution phase is strong-AC and strong-CC.*

Theorem 5.2. *The FPSS++ specification is a faithful implementation in an ex post Nash equilibrium of the VCG-based shortest-path interdomain routing mechanism and provides for faithful usage of the resulting LCP paths.*

Proof. All phases of this specification are strong-AC and strong-CC in an ex post Nash equilibrium, and all phases have consistent information revelation, and the corresponding centralized mechanism is strategyproof. By Proposition 4.2, FPSS++ elicits a faithful implementation of the VCG-based shortest-path interdomain routing mechanism, and provides for faithful usage of the resulting LCP paths. □

No Enforcement Against Changing Messages

Note that as described, this execution phase enforces the *act* of message forwarding but does not enforce the integrity of the message *content.* In other words, if node A sends a message to node B, the execution phase attempts to enforce that *a* message gets sent from A to B, and that the payments are recorded faithfully, but does not restrict a transit node from changing the content of a message.

We feel that this is a reasonable middle ground for a routing algorithm for several reasons: First, in FPSS++ there is no financial advantage for a transit node to change the content of a packet. A record exists that compels a transit node to forward a packet even before the transit node receives the packet, because of our use of checker nodes and the packet transit count list. Second, we assume that a routing node has no interest in changing the content of an application-level message, since there is no connection to the routing node's utility function. Third, we believe that if required, validation of data messages can be solved at user-level with cryptographic signing.

All this said, it is possible to extend FPSS++ further to enforce message non-malleability. One potentially onerous scheme effectively adds "license plates" to packets: just as checker nodes keep a packet transit count list, checker nodes could also cumulatively store the hash of every packet that passes through a principal. Just as the packet transit count list is used to ensure that the number of transit packets entering a node equals the number of transit packets exiting the node, so too could the incoming hash be compared with the outgoing hash. An advantage of this scheme is that it does not rely on cryptographic signing to ensure that message content is consistent. However, we see this addition as unnecessary for the three reasons stated above.

5.6 Complexity and Simulation

What is the state and message complexity of FPSS+ and FPSS++? As part of our design process, we built a network simulator that allows us to measure the relative state and message complexity of FPSS+ and FPSS++ as compared to Feigenbaum's original FPSS algorithm for all construction phases.

Our custom-built discrete network event simulator maintains the nodes' state including a per-node FIFO incoming message queue. When started with a topology file, each node's state is updated to learn about its direct links as well as its own transit cost. After loading this information, each node announces its transit cost to all neighbors, thus starting the first construction phase in the appropriate algorithm. Our simulator services nodes' queues in a round-robin fashion, processing the oldest incoming message on behalf of each node and updating that node's state accordingly. Since there is no node failure in our simulator, the referee is implemented to allow nodes to progress to subsequent phases when all message queues are empty in the current construction phase. For the following experiments our simulator uses the Michigan AS topology graph data [Mic08] that lists interconnected ASs according to the Oregon route-views project [Ore08] for data collected on May 15, 2002. We start with an AS graph that contains 13,233 nodes. From this graph, we selected the bi-connected subset of 8927 nodes. We assign a uniform transit cost to each of these nodes in order to run LCP calculations.

It is always problematic to select a representative graph to use in a network simulation. This is especially true in evaluating FPSS and its derivatives, for two reasons: The first reason is that transit cost must be artificially introduced into the topology graph. The second and more important reason is that the network topology formed before running FPSS is likely to be affected by the fact that nodes know that they are running FPSS. As Feigenbaum et al. observe in their paper, the network graph is likely to depend on "the incentives present when an AS decides whether or not to connect to another AS." As we discovered, our experiments stress this likelihood.

Through our simulation, we are able to measure the convergence time of FPSS+ and FPSS++ and verify the convergence time of FPSS, as reported in Feigenbaum et al. for an Internet topology. FPSS has a convergence time (measured in number of rounds of updates) of d', whereas BGPs convergence time for the LCP computation alone would be d; in the pathological worst case, $\frac{d'}{d}$ could be $\Omega(n)$, where n is the number of nodes of the system. This is a potentially serious problem, but fortunately does not appear in practice, at least when one examines highly connected Internet-like topologies. In their original work, Feigenbaum select a 5773-node bi-connected AS graph and compute $d = 8$ and $d' = 11$. While we do not have access to their exact topology graph, we calculate a similar result on the Michigan 8928-node bi-connected topology, where we find $d = 9$ and $d' = 13$. The convergence time (measured in rounds of updates) is the same for FPSS, FPSS+, and FPSS++.

In their original paper, Feigenbaum et al. analyze the state complexity of the FPSS algorithm with respect to Griffin and Wilfong's model of BGP [GW99] and show that the FPSS additions to the BGP algorithm add only a constant-factor increase to the size of the BGP routing state. Our additions to the FPSS principal state have been similarly small. Our change to the pricing tables is a tag that identifies the node that triggered the last pricing update. The majority of our added state complexity comes with the addition of the checker role, and the added complexity is linear in the degree of a node. In the checker role, each node keeps a copy of each neighbor's state and additionally maintains a transit packet count list for each neighbor. The additional list adds negligible state complexity since each entry in the list is a single integer (the packet count) and in the pathologic worst case (where a node is on the LCP from every source to every destination) the number of entries in the list is bounded by n.

At first glance, a linear increase in state seems reasonable. However, one must then ask about the typical degree of a node. In the Michigan AS topology data, 8413 nodes (out of 8928 nodes, or 94%) have a degree of ten or less. 8849 nodes (or 99%) have a degree

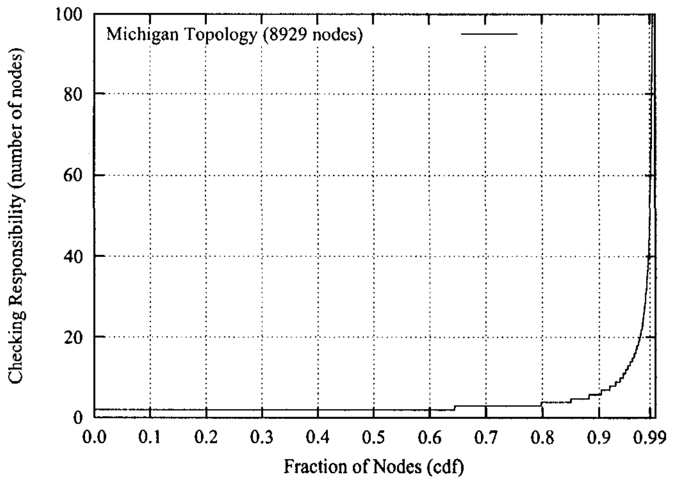


Figure 5.3: This cumulative distribution plot shows the onus of checking in the Michigan topology in FPSS+/FPSS++ simply by measuring the degree of each node. For every node checked, a checker must maintain one copy of the checked node's transit cost information, LCP routing tables, pricing tables, payment lists, as well as a transit count list. This graph omits the 30 nodes (0.3%) with the highest degrees, where the node with the highest degree has 2615 neighbors.

of fifty or less. However, there are some extremely well connected nodes in the AS topology as captured in the Michigan data; the most connected node has a degree of 2615. This unlucky node must act as a checker to 31% of the rest of the network! Figure 5.3 shows the checker burden for the bottom 99% of nodes.

We constructed our simulator to measure the amount of control message traffic that each node must process for its participation in the construction phases of both the FPSS and FPSS+/FPSS++ mechanisms. We show total message traffic size instead of message count to highlight the scalability issues of the algorithms in highly connected networks. The first and second data columns in Table 5.2 shows the amount of message traffic that a node must process, for nodes at specific percentiles over the first three construction phases in the Michigan topology. In this table, total message traffic is measured in megabytes (M). For any given node in the FPSS+/FPSS++ protocols, the amount of individual traffic a node receives will depend on: (1) its placement in the network, which affects algorithm convergence speed and number of updates, (2) the node's degree, which affects (a) the number of transit cost, LCP, and pricing updates it receives, and (b) the number of nodes sending messages to check, (3) the size of the network, which equates to the number of message destinations and size of update messages, (4) the diameter of the network, which affects the average number of hops to each message destination, which in turn affects the

Node Percentile	FPSS (Mich.)	FPSS+/FPSS++ (Mich.)	FPSS+/FPSS++ (Mod. Mich.)
Lowest Node	1.1M	2.2M	2.2M
5%	1.6M	3.2M	3.5M
25%	1.9M	5.7M	6.2M
50%	2.0M	9.5M	10.0M
75%	2.0M	15.3M	15.3M
95%	2.1M	22.7M	22.7M
Highest Node	48.6M	4949.5M	189.3M

Table 5.2: Total message traffic measured in megabytes (M) that a node receives in FPSS, and in its roles as principal and checkers in FPSS+/FPSS++, in the Michigan topology (data columns 1 & 2) and the modified Michigan topology (column 3).

size of the update messages, and (5) the physical encoding of the message, which affects the efficiency of the message.

For much of Table 5.2, the news is positive: while there is a notable increase in the amount of message traffic when we add checker nodes and actions into the system, 95% of nodes receive message traffic approximately within an order of magnitude of FPSS for the entire construction of the algorithm. The terrible news comes from the highest-connected node in the system, which as mentioned above is responsible for checking roughly a third of the rest of the network, translating into approximately 4.9 gigabytes of messages.

Given this progressive burden, is there anything that we can do to reduce the amount of message traffic that each node must process? Modifying the communication network may be a tenable option. One interpretation of the Michigan graph data is that the peering algorithms in nodes running FPSS++ would likely change their pairing relationships to account for the increased cost in acting as a checker node. This is especially true once actual transit cost information is factored in, since higher degree does not necessarily equate to lowest cost path.

What would be the benefits and costs of re-forming the transit network so as to force the most well-connected 0.3% of nodes to adopt a lower degree? As an experiment, we re-processed the Michigan AS topology graph to delete edges, artificially restricting the degree in the network to a maximum value of 100. Where possible, the new topology was formed by deleting edges that still left the entire graph bi-connected without the need to add a replacement edge.

The benefits of FPSS+/FPSS++ running in the reduced-degree topology is the reduced computation and message burden placed on large-degree checker nodes. The third data column in Table 5.2 shows the amount of message traffic that a node must process on the modified topology graph. Importantly, the highest node is able to reduce message traffic to 189.3 megabytes, or more than an order of magnitude decrease. Additionally, by favoring edges that when deleted still leave the entire graph bi-connected, the overall message traffic in the network goes down substantially. This effect is more pronounced in topologies where many nodes have degrees greater than two, such as in the Michigan topology, where gigabytes of checker messages have been removed from the system.

There are two drawbacks to imposing this artificial degree-capping limitation on the network topology: the first drawback is illustrated by Table 5.2: in some cases, a two-neighbor node in the Michigan topology was bi-connected via a deleted edge, and a new edge had to be added to keep the bi-connected property. This translates to a small increase in the message traffic at the less saturated end of the spectrum, as other nodes take on these "bi-connection orphans". The second drawback is that by deleting edges, one is effectively adding hops to certain source-destination LCP routes. This is easy to understand: whereas the highly connected node had a direct connection to 2614 neighbors in the original Michigan topology, in the revised topology messages to 2514 of these nodes involve at least one hop and a transit payment. The increase in LCP path length in the modified Michigan topology is small: focusing only on affected source-destination routes, we find that the majority of affected paths add just 1 transit node, with an average number of 1.4 transit nodes, and a longest path of 5 transit nodes.

Unfortunately, we see no way to maintain faithfulness and to reduce the checking burden without adding additional infrastructure or re-forming the communication graph. The straightforward idea to enlist fewer checkers would violate correctness since our proofs rely on update messages to a principal traversing a checker before reaching the principal, to ensure that the principal does not change the message before relaying to other checkers. (This property is used in proving Propositions 5.6 and 5.8, and illustrated in Figure 5.2.) Moreover, our proof of Proposition 5.5 requires that a node be a neighbor's checker if the node is on the LCP from its neighbor to any other destination. One idea for reducing the checking burden may lie in forming "checker subsets" that are responsible for checking calculations that affect portions of the LCP and pricing tables. However, we see no way to incorporate such a change into our existing algorithm extensions while maintaining correctness. Moreover, it seems that such a change would require the referee to take on more responsibility in knowing how to interpret the resulting partial checks. We leave such explorations for future work. We view the cost of checking as the price that we pay for a faithful specification that does not rely on cryptographic message signing or another incentive scheme. It is clear that the FPSS+/FPSS++ extensions are hindered by their reliance on checker nodes in highly-connected topologies. However, in situations where cryptographic primitives are not available or are particularly expensive, it may be possible to adopt checker-based strategies to avoid rationally motivated failure.

5.7 Bibliographic Notes

The starting point for this chapter was a paper by Feigenbaum, Papadimitriou, Shenker, and Sami [FPSS02]. Feigenbaum, Ramachandran and Schapira recently re-visited the interdomain routing problem [FRS06] to propose an incentive compatible routing solution to a modified problem based on "next-hop" policy routing. Instead of ensuring that packets are routed along lowest-cost paths, each participant decides among available routes to a destination solely based on the routes' next hops. They can show that their algorithm is immune to the types of rational manipulation that we address here for a LCP-based algorithm and that their algorithm is incentive-compatible in an ex-post Nash equilibrium.

Afergan [Afe03] studied repeated games in networks and shows how the dominant strategy equilibrium in Feigenbaum et al. (FPSS) is lost when run as a repeated game. Our revised protocol is sensitive to the same criticism. In other work, Afergan [Afe06] studies how repeated game analysis can be used to tweak routing protocol parameters to have a significant impact on the equilibrium.

Feldman et al. [FCSS05] in their evaluation of "hidden-action" packet forwarding show how incentive-laden all-or-nothing contract offers can be used to achieve communication compatibility in the FPSS [FPSS02] problem. Taking algorithm compatibility for granted, they create a revised protocol that achieves a Nash equilibrium in the absence of per-hop monitoring, and a dominant strategy equilibrium in the presence of an external obedient per-hop monitoring mechanism.

Levin et al. [LSZ06] posit that the Gao-Rexford model of BGP [GR01] under an ex-post Nash knowledge model is almost faithful[2] and can be made faithful with a cryptographic route verification procedure. To make this claim, their work assumes that the cost for transiting traffic is zero. Furthermore, a rational compatible user's valuation function depends solely on the existence of a stable unchanging set of routes, from itself to other destinations, at some point in future time. Finally, the authors must exclude any strategic behavior that would create a *dispute wheel* [GSW02]. In practice, a rational participant may choose to violate the Gao-Rexford assumptions in order to selfishly benefit from the result, and there is no disincentive for this misbehavior.

[2]Their work does not use the language of faithfulness, but instead implicitly extends the traditional interpretation of incentive compatibility to include actions beyond information revelation.

Chapter 6

Methodology Applied: Failure in Distributed Consensus

6.1 Introduction

This chapter returns to the Rational Byzantine Generals problem first introduced in Chapter 2. We consider the problem of rationally motivated failure in a form of *distributed consensus*. We solve this problem by providing an algorithm that solves consensus by maximizing overall participant utility. This algorithm runs on the same network of users that provide consensus inputs.

It is no accident that this thesis focuses partly on consensus: First, consensus is an important systems problem. There are many distributed systems problems involving consensus, such as scheduling, resource allocation, and leader election. Second, consensus relies on private inputs from each participant, making the algorithm particularly susceptible to rationally motivated failures. Third, both the traditional and our modified consensus problem are well-defined. This allows us to focus on the integration of incentives with other Byzantine failure techniques. This chapter makes the following contributions:

- It demonstrates how incentives can be combined with traditional Byzantine Fault Tolerance (BFT) tools to prevent rationally motivated failure in information revelation, computation, and message passing, while staying robust to other Byzantine failures in computation and message passing.
- It presents and evaluates a new distributed consensus problem and algorithm that is robust to the above forms of failure. This work composes a new consensus algorithm from two existing algorithms.

6.2 Problem Description

The goal in this problem is to construct a consensus algorithm that runs in a network with a mixture of obedient, rational, and non-rational faulty participants, and makes a *choice* decision that maximizes overall participant utility. This goal differs from previous consensus problems (and the consensus problem defined in Chapter 2) as the same participants that run the algorithm are allowed to express specific *values* for different choice candidates in a decision problem. However, the safety and liveness conditions that otherwise specify consensus remain the same. The safety and liveness conditions are:

- **Liveness Conditions**

  L1. Some proposed choice is eventually selected.

  L2. Once a choice is selected, all non-failing participants eventually learn a choice.

- **Safety Conditions**

  S1. Only a single choice is selected.

  S2. The selected choice must have been proposed by a participant.

  S3. Only a selected choice may be learned by a non-failing participant.

  We add two new safety conditions for value-based consensus decisions:

  S4. A choice that maximizes overall value is selected.

  S5. Only truthful values for choices are reported by non-failing participants.

We present the problem in terms of two consensus problems that help describe the manipulation issues. These problems are Byzantine Generals [LSP82] and consensus via MDPOP [PFP06]. The former problem addresses Byzantine consensus on "given" private information, and the second problem addresses rationally motivated failure in the actual private information revelation.

6.2.1 The Rational Byzantine Generals Revisited

When we last left our generals in Chapter 2, they were retreating from the enemy city after failing to reach consensus. To pick up the story where we left off, the three generals encounter each other on the path back to the queen and compare notes on their recent failed campaign. General 3 breaks down and confesses to the other generals that he did not follow the queen's algorithm specification. Knowing the temperament of the queen, and realizing that they have failed, each general realizes that returning to the queen in this state is a death sentence.

Being rational generals, the three decide to renounce their loyalty to the queen and elect to strike out on their own. The generals check their maps and agree to plunder a new city that lies nearby and split the gold between themselves from their campaign. As before, the generals agree to approach the city via three different paths and will communicate with one another only by messenger. After observing the city, they must decide upon a common plan of action. To avoid any problems this time (or so they think), the generals change their communication to be the estimate of their utilities, measured in gold pieces, for both an "Attack" and a "Retreat" battle plan. After establishing agreement on the battle plan utilities, the generals agree to pick the utility-maximizing decision as their *outcome function*. This outcome function means that the battle plan associated with the highest utility is chosen by the generals. In so selecting this outcome function, the generals meet Lamport et al.'s requirement that the same robust outcome function be used by all generals.

The generals begin to carry out their plan. Each general approaches the enemy city via a different route. After observing the city, here is what the three generals privately think:

- **General 1** can see a lot of gold from his vantage point and thinks that his utility for an "Attack" is 90 gold pieces. However, on his rush to the city, he noticed a small traveling caravan that could be privately pillaged for 25 gold pieces if his men were allowed to "Retreat".

- **General 2** can see a bit of gold from his vantage point and thinks that his utility for an "Attack" is 50 gold pieces. A "Retreat" costs nothing to him, but does not help him either and so General 2 assigns "Retreat" a utility of 0 gold pieces.

- **General 3** sees a bit of gold from his vantage point, but once again believes that his horse will surely be killed in the onslaught. He estimates that buying a new horse will cost about his share of the spoils, and so his utility for an attack is 0 gold pieces. Moreover, he also noticed a large traveling caravan that could be privately pillaged for 150 gold pieces if his men were allowed to "Retreat".

The utilities are summarized in the following table:

In an attempt to satisfy [S4] and [S5], the generals have agreed to use an outcome function that selects the "winning" choice after summing reported utilities. In this problem, the utility-maximizing outcome is "Retreat" (a sum of 175 vs. a sum of 140.) This new outcome function is a step in the right direction; notice that under the old *majority vote*

	Attack	Retreat
General 1	90	25
General 2	50	0
General 3	0	150

outcome function, Generals 1 and 2 would have voted to "Attack", and the consensus majority function would be to "Attack". Because the utility-maximizing outcome is "Retreat", the *majority vote* violates [S4].

However, this simple change of outcome function is also subject to the same sorts of manipulations that we discuss in Chapter 2:

- There is nothing to stop the generals from manipulating their declared utilities to drive their personally desired outcome. As a simple example, Generals 1 and 2 can overstate their "Attack" utility and General 3 can overstate his "Retreat" utility. The utility information quickly becomes useless as rational generals attempt to lie their way into a desired outcome, violating [S5].

- There is nothing to stop a participant from failing. Once again, recall that one failure in the generals' three-party agreement algorithm with unsigned messages will detectably destroy the attempted consensus, since this algorithm can only tolerate up to $\lfloor\frac{n-1}{3}\rfloor$ failed participants. We have not defined the utility for each participant when the algorithm cannot reach consensus (we would need to add a third column to the utility matrix above), and so we really do not know if General 3 has the option of exhibiting a rationally motivated failure. But we can speculate: if General 3 feels that a failed consensus (which saves his horse) is better than the decision to "Attack", then we should expect General 3 to fail once it learns the utilities of his compatriots.

This chapter proposes a modified consensus algorithm that is utility-based, but addresses these problems by providing incentives for participants not to fail. This solution is based on the MDPOP algorithm by Petcu et al. [PFP06], but unlike MDPOP, the algorithm that we suggest can both tolerate rationally motivated and non-rationally motivated failures.

6.2.2 Faulty MDPOP

MDPOP [PFP06] is a distributed constraint optimization algorithm in which self interested agents with private utility functions agree on values for a set of variables subject to side constraints. In this chapter, we describe a simpler version that can solve distributed consensus.[1] MDPOP provides a decentralized way to calculate a Vickrey-Clarke-Groves (VCG) mechanism. The algorithm is weakly faithful for an ex post Nash equilibrium solution concept, in that all participants are rational but will follow a protocol whenever there is no deviation that will make them strictly better off. To solve a single-variable decision problem, all participants must be able to communicate with each other without failure, and every participant has a trusted channel to an obedient *bank*. The bank is used to collect taxes and provide payments to participants at the conclusion of the algorithm. MDPOP, applied to an n-General decision problem, works as follows:

1. The generals form a communication path of length n, labeling the ends head and tail. This path is the *main economy path* ($\vec{P}$). The generals then form n *marginal economy paths* ($\vec{P}_{-1}...\vec{P}_{-n}$), each of length $n-1$, where each path omits a different General i.

2. Value information is passed and aggregated from tail to head in all paths. Each participant sums the payoff information and sends the sum towards the head.

3. The system-optimal battle plan is calculated by the head and passed from head to tail in all paths.

4. Whenever the battle plan computed in the main economy path and a marginal economy path that omitted General i is different, each participant j in the marginal economy calculates a tax for participant i. This tax is j's value for the choice selected in the marginal economy, subtracting j's value for the choice selected in the main economy. This value is then reported to a bank, who imposes this tax.

Example of MDPOP:

Step 1: The generals in the previous section use MDPOP to form a main economy path and marginal economies. Let us say that the selected main economy path (tail to head) is $1-2-3$, which forms marginal economies of $2-3$, $1-3$, $1-2$.

Step 2: The head in each economy receives the following values:

	Attack	Retreat
$\vec{P}$ (Main)	140	**175**
$\vec{P}_{-1}$	50	**150**
$\vec{P}_{-2}$	90	**175**
$\vec{P}_{-3}$	**140**	25

[1]MDPOP can be more general than the single-variable version described here. The additional complexity needed to support multiple variables is not relevant in explaining our problem. However, the techniques we use in our paper can be applied to the full MDPOP algorithm.

Step 3: The system-optimal battle plan is "Retreat" in $\vec{P}$, $\vec{P_{-1}}$, and $\vec{P_{-2}}$, and "Attack" in $\vec{P_{-3}}$.

Step 4: The two Generals in $\vec{P_{-3}}$ calculate a tax for General 3: $Tax_1(3) = (90 - 25) = 65$ and $Tax_2(3) = (50 - 0) = 50$, or 115 total tax, which is reported to the bank. The reasoning is that General 3's values caused "Retreat" to be selected and is the only General who should be taxed. General 3 was still much happier with "Retreat" than with the outcome where "Attack" had been chosen, even after subtracting the tax of 115 from the pre-tax payoff of 150 ($35 > 0$).

Note that if General i is taxed, the value of $Tax(i)$ does not depend on General i's report. This characteristic is key to a rational agent's truthfulness. If a general overstates or understates its value for a battle choice, either nothing will change, or the choice will switch, but the lie will always cause the general to be worse off after paying any applicable tax.

Petcu et. al. [PFP06] effectively demonstrate [L1-L2] and [S1-S5] in their paper since MDPOP uses a distributed VCG mechanism and their only participants are (in our language) rational compatible. With Byzantine failures, however, [L1-L2] and [S1-S5] are no longer assured, as the following example demonstrates.

Example: Byzantine Failure Destroys Equilibrium

MDPOP is shown to be weakly faithful in an ex post Nash equilibrium. This equilibrium requires an environment with a participant model where all participants are rational and a knowledge model where the notion of rationality is commonly understood by algorithm participants. When a Byzantine general enters the system, the ex post Nash equilibrium may be destroyed. A rational general can no longer assume that truthful information revelation, message passing, or computation is its utility maximizing behavior.

Consider the following scenario: General 2 in the previous setup is the middle general in the $1 - 2 - 3$ main economy path. General 2 can change General 1's "Attack" value to be 150, thus changing the *claimed* utility maximizing decision to be "Attack" (when the *true* utility maximizing decision was "Retreat"), and ensuring that General 1 is the only general to pay tax. The tax that General 1 pays is now greater than its benefit for an attack ($Tax_2(1) = (0 - 50) = -50$ and $Tax_3(1) = (150 - 0) = 150$, or 100 total tax), and General 1 would have been better off not participating in the algorithm ($90 < 100$). This is of course just one example of how a particular participant could fail; any participant that fails so as to disrupt the equilibrium will destroy the faithfulness guarantees made by MDPOP.

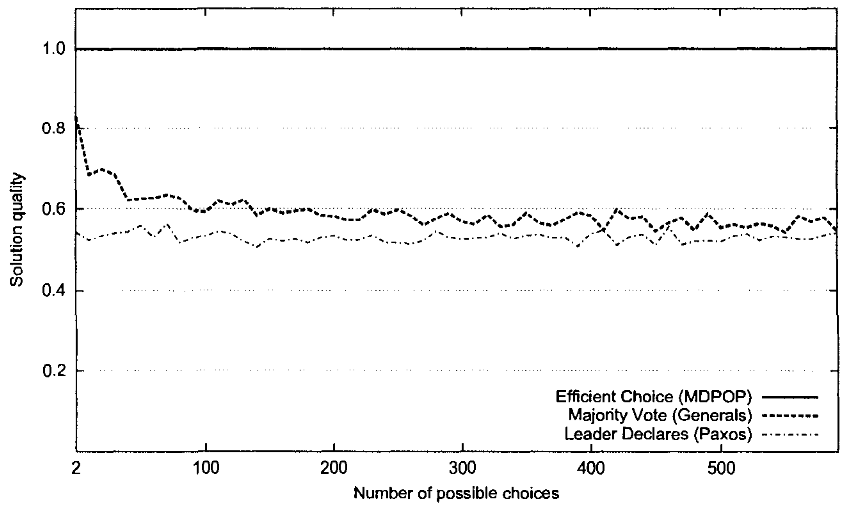


Figure 6.1: Experimental no-fault solution quality of Efficient (MDPOP), Leader's Choice (FaB Paxos), and Majority Vote (Byzantine Generals) as the number of possible choices is varied.

6.2.3 Problem Comments

The problem studied in this chapter is to construct a new consensus algorithm. The goal for the consensus algorithm is to make a decision choice that maximizes overall value, running on the same network of opinionated participants. For comparison, the experiment shown in Figure 6.1 displays the relative solution quality selected by different known algorithms when there are no failures. Solution quality measures how well an algorithm selects a choice that maximizes overall value. This metric ranges from 0.0 to 1.0, respectively indicating an approach that picks the worst or best possible choice. In this simulation 100 participants are given choice-value vectors with random integer values from a uniform distribution. They participate in each of the algorithms with the same set of values and the process is repeated and averaged over 100 runs. The average quality of the choice is then normalized. A score of 0.5 with this distribution indicates an algorithm that scores as well as picking a random choice since values are picked from a random distribution.

Algorithms that pick the efficient system utility maximizing choice (e.g., MDPOP) always score 1.0. Assuming that participants do not manipulate their inputs to the algorithm, a majority vote (as in Byzantine Generals) does better than random for small numbers of choices. An algorithm that allows a designated leader to pick its favorite choice is almost akin to picking a random choice; solution quality is slightly better because the selected choice is always the best for at least one participant. These effects are more or less pronounced depending on number of participants, number of choices, and the distribution function of used to set the participant's values.

Our goal, then, is to build an algorithm that has a solution quality and rationally motivated failure tolerance equal to that of MDPOP, but with the traditional fault tolerance found in a Byzantine agreement algorithm.

6.3 Model

6.3.1 Participant Model

We allow three types of participant behavior:

- An *obedient* participant instructs its node to follow the suggested specification exactly without incentives. There is no minimum number of obedient participants in the decision problem, although we will assume one infrastructure node as described below.
- A *rational* (compatible) participant is capable of directing its node to fail in order to achieve a preferred consensus outcome. We target a *k-partial ex post Nash* knowledge model as described in Chapter 3. This model means that that rational participants believe that a minimum number of participants is guaranteed not to exhibit non-rationally motivated faults. We will aim to make k as small as possible, and later in the chapter we will see the trade-off between a smaller k and the message complexity of our proposed solution. There is no limit to the number of rational participants in the system.
- *Byzantine* participants direct their nodes to express arbitrary failure. There is no limit on the number of Byzantine participants in the system, but we do assume a maximal number of simultaneous non-rationally motivated faults, as described below. A bound on simultaneous Byzantine faults is standard in other BFT algorithms [CL99, Rei95].

Remark 6.1. *We support an unlimited number of rational participants but do not need to bound the number of simultaneous rationally motivated faults. In equilibrium, the rational participants will not exhibit rationally motivated failure.*

Remark 6.2 (Crazy Millionaire Problem). *Value is in the eye of the beholder, and it is impossible to distinguish between rational participants and users that are consistent but foolish in their value reporting. For example, a participant can declare an arbitrarily high value for a certain choice. This node could be "stuck" and always declare* \$1,000,000 *for* *the first choice it sees, regardless of what choice appears first. This is indistinguishable from the rational participant that has value* $1,000,000 *for some choice, which just happens to appear first in the list of choices. Such participant errors in private information revelation are neither studied in traditional system failure models, nor are they a behavior model that is studied in our thesis.*

Somewhere in the system, we assume that there is a participant that is capable of correctly running some trusted components. One of these components is the role of the *referee*. The referee will not participate in the decision problem, and its tasks will be described later in this chapter. We require three other trusted components, described as logical devices: an *unforgeable signature device*, a *bank device*, and an *anti-collusion device* as described in the next section. It is unlikely that any of these trusted components can be distributed to non-trusted participants.

6.3.2 Network Model

Nodes are connected via an Internet-style network. The network may temporarily delay, duplicate, re-order, or fail to deliver messages. However, we assume that communication faults are eventually rectified via transport layer protocols. We require that all nodes be able to communicate correctly, though possibly indirectly, with the node running the trusted referee service. We require subsets of nodes, defined in the next section as *cliques*, to communicate directly with each other.

6.3.3 Dependency Model

We will build our remedy on top of logical dependencies, as described in Section 6.4.1, and as realized in Section 6.5.2. We assume that there is nothing exogenous to the system that would prevent new manipulation opportunities.

6.4 Remedy: The RaBC Algorithm

We now present the manipulation remedy, in the form of a distributed consensus algorithm called RaBC (for *Ra*tional *B*yzantine *C*onsensus). An overview of this algorithm is shown in Figure 6.2, and a graphical example follows later in the chapter. The RaBC consensus algorithm selects a *choice* for a *variable*. Each node[2] has a private value for

[2]We follow the convention from the last chapter and refer to the *node* in the algorithm, instead of belaboring the relationship between the human *participant* and his/her computational node that actually participates in the distributed algorithm.

each choice. Nodes assign negative values to choices that are undesirable and zero value to indifferent choices. The list of values for each choice is the node's *choice-value vector.* The goal of RaBC is to select the choice that is the most value-maximizing in total out of all choices and value declarations made by nodes. Formally, we wish to satisfy aforementioned liveness and safety conditions [L1-L2] and [S1-S5]. Consensus is calculated by the same set of nodes with an interest in the consensus decision. Rational nodes receive appropriate incentives and will choose to make truthful value declarations and obediently participate in computational aspects of the algorithm. Failures are divided into *faults* and *halts.* Nodes report *proof* of faults and *claims* of halts to a trusted node called the *referee.* A rational node is always better off reporting failure when it occurs. Fault proofs cannot be faked, and faking a halt claim yields no benefit when the halt does not exist. Nodes cannot spoof the proofs or claims of other nodes. Failed nodes are penalized and excluded by the referee, who then restarts the consensus.

By introducing the limited reliance on a trusted referee, RaBC trades off the elegance of a fully distributed algorithm for the pragmatic considerations in building a system provably robust to rationally motivated failure. The consensus is divided into smaller computation and agreement problems for scalability reasons. The algorithm trades off scalability for non-rational fault tolerance.

6.4.1 Logical Device Dependencies

Chapter 4.4.3 explained the utility of building a remedy that relies on logical devices. These devices are integral to the success of RaBC, but their implementation details are separate from the RaBC algorithm. This section describes the logical devices that are required by RaBC; in Section 6.5.2 we describe how these devices are realized in our implementation.

Unforgeable signature device

RaBC requires an an *unforgeable signature device* to prevent spoofing, replays, and message corruption. Our reliance on signing and consequential assumptions are similar to other practical systems that detect and recover from Byzantine manipulation [CL99, Rei95] where private information is not involved. In this thesis, we will denote a message that is signed by node i as $(message)_{\sigma_i}$. The realization of this device, through a public key infrastructure service, is described in Section 6.5.2.

Bank device

We require a non-faulty *bank service.* The bank is dormant during normal operation but coordinates currency transfers ordered by the referee. As in real life, the bank provides a trusted external way to enforce currency transfers. The bank is implemented by a trusted node. The realization of this device, through a reliance on an existing Internet banking mechanism called PayPal [Pay06], is described in Section 6.5.2.

Anti-collusion device

Our algorithm uses a Vickrey-Clarke-Groves (VCG) mechanism, which is susceptible to collusion problems. These collusion problems can be group-collusion through coordinated efforts of several nodes, or self-collusion where a single node performing a Sybil identity attack [Dou02] claims multiple identities. We require an *anti-collusion device* that can address group- and self-collusion. The realization of this device, through a Tor anonymizing network [DMS04] and an external identity check, is described in Section 6.5.2.

6.4.2 Algorithm Mechanics: The Suggested Strategy

Rather than stating a full (huge) strategy space Σ^m and corresponding outcome function $g(\cdot)$, we can provide a suggested strategy s^m in the form of an algorithm and show that this algorithm is equilibrium behavior for all rational compatible nodes, given a relevant solution concept in our target environment. For now, we focus on simply defining s^m; we show how s^m satisfies this requirement later in the chapter.

The suggested consensus algorithm operates by summing up node values for each possible choice for a variable and selecting the choice with the highest value, as in MDPOP. Like the MDPOP example in Section 6.2.2, the algorithm calculates the choice in a *main economy* and in a set of *marginal economies.* As before, each marginal economy omits a different node. Unlike MDPOP, the RaBC algorithm runs many small compute-and-agreement phases among *cliques* of nodes in the *main economy.* The cliques are ordered along a line with the tail clique connected via intermediate cliques to the head clique. After the choice is made in the *main economy*, the nodes locally compute the marginal economy choice. Taxes are collected by the referee when the main economy choice differs from any marginal economy choice. At a high level, nodes should follow the suggested algorithm. The overview of this algorithm is shown in Figure 6.2, and an illustrated example is shown in Figures 6.3 and 6.4.

Bootstrap

1. Each node learns of and declares interest in the choice variable to the referee.
2. The referee presents the participant vector to each node, defining cliques and economies, and activating the tail clique of the main economy.

Consensus in Main Economy

3. Each node in the active clique declares its input (initially its choice-value vector) to all nodes in its clique.
4. Each node in the active clique validates the received input, sums all vectors, and performs agreement on the summed choice-value vector.
5. If a node belongs to as-yet unactivated adjacent cliques, it takes the agreed sum as its new input and jumps to Step 3 in the adjacent clique, activating that clique. Steps 3-5 are repeated until the consensus decision is performed at the head clique.
6. The optimal choice is selected by the head clique and propagated to all nodes from head to tail clique.

Consensus in Marginal Economies

7. Every node i who has learned node j's choice-value vector computes the choice in the marginal economy excluding node j. If this choice and the main economy choice are different, the marginal economy choice is propagated to all nodes.

Tax

8. For each choice that is different in the main and a marginal economy excluding node j, each node i calculates a portion of j's tax bill. The bills are sent to the referee, who then imposes a tax on node j.

Failure Detection and Reporting

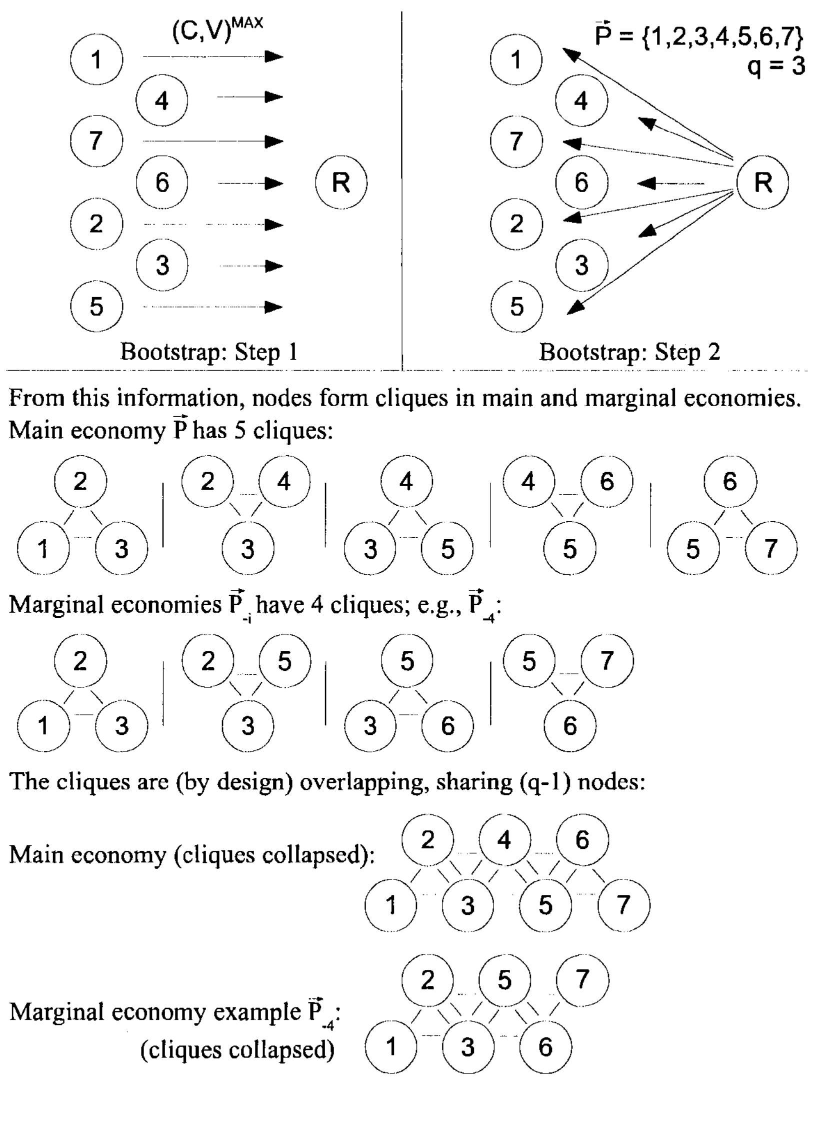
9. While the above phases execute, each node detects any failure by another node. In the case of a fault, nodes send a proof to the referee. In the case of a halt, nodes send a claim to the referee for further investigation. The differences between faults and halts are discussed in detail in Section 6.4.7.

Figure 6.2: Overview of the suggested algorithm.

6.4.3 Bootstrap

The bootstrapping process starts as nodes signal their intention to participate in the decision problem by proposing choices for the decision variable to the obedient *referee* node. At some point in time, the referee stops accepting new proposals and fixes the variable's choice domain. Nodes learn the final choice domain and calculate their value V for each possible choice C. A node i commits to participating in the consensus by sending the referee its highest and lowest valued entries in its choice-value vector. These two messages are $((C,V)^{max})_{\sigma_i}$ and $((C,V)^{min})_{\sigma_i}$, using the $(msg)_{\sigma_i}$ signing notation described in Section 6.4.1. These two entries are part of the full choice-value vector $(\vec{C,V})_i$. Once nodes have committed to the decision problem, the referee signals the start of consensus by sending to each node i the following information:

- $\vec{P}$, the participant vector, a random permutation of the n nodes.



From this information, nodes form cliques in main and marginal economies.
Main economy $\vec{P}$ has 5 cliques:

Marginal economies $\vec{P}_{-i}$ have 4 cliques; e.g., $\vec{P}_{-4}$:

The cliques are (by design) overlapping, sharing (q-1) nodes:

Main economy (cliques collapsed):

Marginal economy example $\vec{P}_{-4}$:
(cliques collapsed)

Figure 6.3: RaBC in action: bootstrap phase.

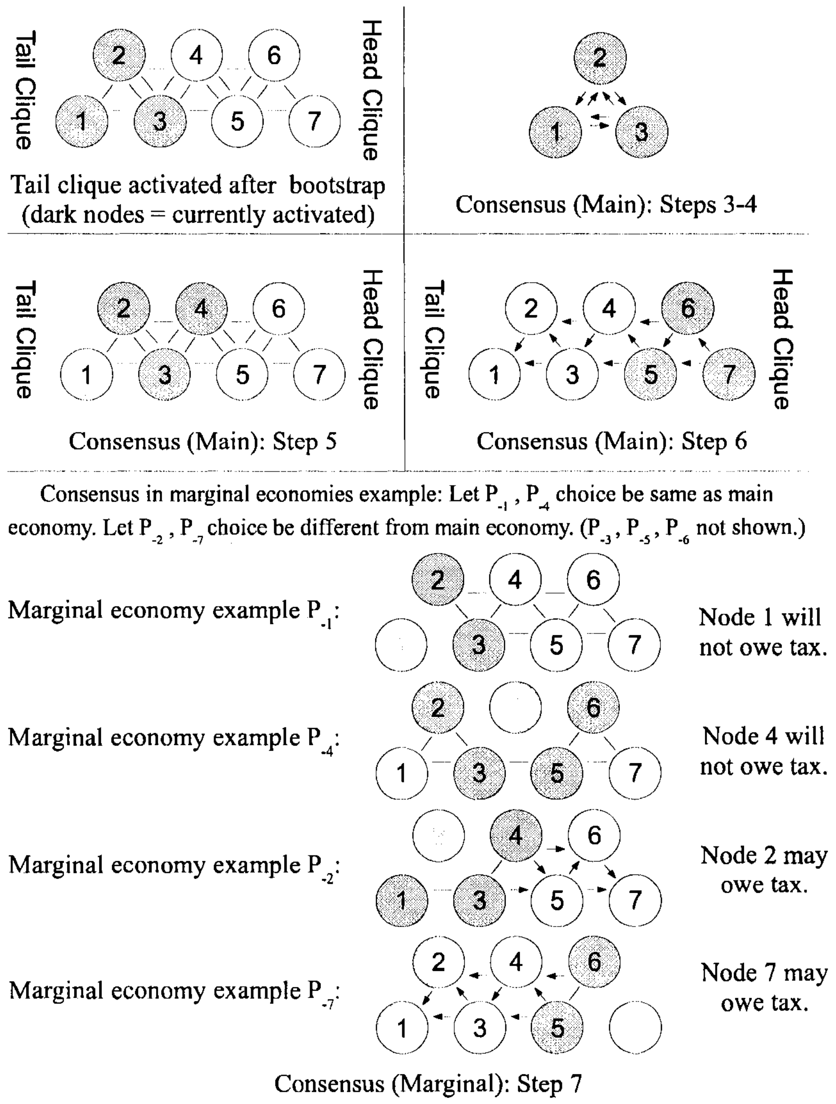

Tail Clique
Head Clique
Tail clique activated after bootstrap (dark nodes = currently activated)
Consensus (Main): Steps 3-4
Consensus (Main): Step 5
Consensus (Main): Step 6
Consensus in marginal economies example: Let P_{-1}, P_{-4} choice be same as main economy. Let P_{-2}, P_{-7} choice be different from main economy. (P_{-3}, P_{-5}, P_{-6} not shown.)
Marginal economy example P_{-1}:
Node 1 will not owe tax.
Marginal economy example P_{-4}:
Node 4 will not owe tax.
Marginal economy example P_{-2}:
Node 2 may owe tax.
Marginal economy example P_{-7}:
Node 7 may owe tax.
Consensus (Marginal): Step 7

Figure 6.4: RaBC in action: consensus phases.
]

- q, the size of each clique. q is tuned by the system designer based on the number of simultaneous non-rational faults f that must be tolerated. Raising the value of q raises the number of non-rationally motivated faults that can be tolerated. Yet, it is advantageous to keep q small for scalability reasons, since the number of messages in our algorithm grows with clique size. Section 6.5 discusses the trade-offs in picking an appropriate q, given f. We pick $q = f + 2$ and $f = 1$ to keep our examples simple in the first half of this chapter.

- $((C,V)^{max})_{\sigma j}$ and $((C,V)^{min})_{\sigma_j}$ for each node j in a clique where node i is also in that clique.

$\vec{P}$ is used to define the *main economy*, *marginal economies*, and *clique sets* in each economy as follows: The main economy is simply set to the value of $\vec{P}$. The n marginal economy paths are created by removing a single node from $\vec{P}$. For example, the marginal economy path[3] without node j is set to P with node j removed, and is denoted $\vec{P}_{-j}$. *Cliques* are formed as the set of q adjacent nodes in $\vec{P}$. There are $(n + 1 - q)$ such cliques. The clique with the first q adjacent nodes is called the *tail* clique, and the clique with the last q adjacent nodes is called the *head* clique.

6.4.4 Consensus in Main Economy

The nodes in the tail clique are the first to be active in the algorithm. Nodes *not* in the tail clique wait to hear from another node before running their portion of the algorithm.

The consensus process starts when each active node i sends its signed choice-value vector $(\vec{C,V})_{\sigma_i}$ to all other nodes in its clique. All nodes in the clique gain a consistent view of all choice-value vectors in this clique. These nodes now execute an agreement on the sum of the choice-value vectors. In effect, each node in a clique is computing the choice-value sum for all nodes up to an including those nodes in this clique.

With the sum in this clique agreed upon, the algorithm proceeds through the next cliques. Adjacent cliques share $q - 1$ common nodes, and these common nodes use the result of the last clique's agreed sum as their input for the next clique. Only the newly activated node in the new clique need declare its private signed choice-value vector. This process (declaration of inputs, agreement on sums) continues through the cliques until the messages reach the head clique.

[3]While the marginal economy vectors are helpful in explaining the algorithm, note that the bulk of RaBC only runs in the main economy. Unlike MDPOP, marginal economy calculations are local decisions.

When the process reaches the head clique, the head clique's nodes are able to select the optimal choice $C(\vec{P})$ which is the entry in the summed value-choice vector that has the highest value. Other nodes learn about the choice when the signed and agreed summed value-choice vector is passed back down toward the tail.

6.4.5 Consensus in Marginal Economies

After the main economy's final summed value-choice vector is passed down through the nodes, each node is responsible for calculating the marginal economy choice. Every node j's private value-choice vector is known by $q-1$ other nodes. It is the task for this set of nodes to calculate the marginal economy choice $C(\vec{P}_{-j})$ by subtracting $(\vec{V,C})_{\sigma_j}$ from the main economy's final summed value-choice vector and selecting the choice that has the highest value. If the choices in the $\vec{P}$ and $\vec{P}_{-j}$ are different, the choice of the $\vec{P}_{-j}$ economy is propagated both up and down the line.

6.4.6 Tax

Taxes are a normal part of algorithm operation, and are distinct from penalties imposed in case of failure. Similar to MDPOP, tax bills are calculated by nodes and enforced by the referee.

Tax is collected whenever the choice selected in the main and any marginal economy are different, and is an important mechanism for preventing rational manipulation. Whenever a node i learns of a different choice in the main and a marginal economy that omitted node j, node i calculates a tax $T_i(C(\vec{P}_{-j}))$ for node j. This tax is i's value for the choice selected in the marginal economy subtracting i's value for the choice selected in the main economy, or $V_i(C(\vec{P}_{-j})) - V_i(C(\vec{P}))$. These tax calculations are agreed upon by every node k who knows node i's private value. Once agreed upon, tax calculations are sent to the referee.

6.4.7 Failure Detection and Reporting

Failures are divided into *faults* and *halts*, regardless of the rationality underlying the failure.

Faults

A fault occurs whenever a faulting node provides "bad" data to another node. If a fault occurs, at least one node in the system has the ability to present a set of messages to the referee that together act as proof of faulty behavior. Because of the simplicity of RaBC, all faults are eventually represented in one of four types of error:

A **Signing Inconsistency** occurs when a message does not match its signature. These bad messages are simply ignored as noise.

An **Agreement Inconsistency** occurs when a set of messages, each supposedly for the same calculation, are inconsistent in their contents. This can occur when agreeing on the summed choice-value vector or when agreeing on the amount of tax to charge another node. If a fault occurs during agreement, a node asks other nodes in its clique for help in finding the Message Inconsistency (discussed next) that caused agreement problem, and presents this as a proof to the referee.

A **Message Inconsistency** occurs when a set of properly signed, supposedly identical messages are inconsistent in contents. For instance, some node i may share its value-choice vector but claim different values $(\vec{C,V})_{\sigma_i}$ and $(\vec{C,V'})_{\sigma_i}$ to different nodes. Because these messages are signed, the inconsistency is apparent to anyone who acquires and compares both messages.

An **Out of Bounds Input** occurs when a node attempts to declare a choice-value vector that violates or restates the initial minimum/maximum declaration to the referee. Catching this fault is important for rewards and penalties, as discussed in Section 6.6.3.

Halts

A halt occurs whenever a halting node stops participating in the algorithm. Halt detection by nodes is not required (but speeds up algorithm execution) as the referee will eventually detect halts on its own according to a preset timeout. Unlike a fault, there can be no signed proof offered by a node for a halt. A halt is suspected by nodes in a clique when a node is activated but then waits longer than the message delivery timeout. A suspected halt report to the referee contains the halting node identity and a description of the missing data. The referee investigates the report and either verifies the halt or collects and delivers the claimed missing data back to the reporting node.

6.4.8 The Referee

The referee is used to bootstrap the algorithm and to collect taxes after the consensus decision is made. It is intended to be as lightweight as possible, while still guaranteeing that our penalty enforcement mechanism functions correctly.

If there are no failures, the referee does not communicate with nodes while consensus is being decided. After a choice is selected, nodes report tax bills for other nodes to the referee, who in turn debits the bank account of the taxed node. In our scheme, taxes are kept by the referee, perhaps to offset any cost of running the consensus infrastructure. Other work explores schemes to redistribute collected taxes back to participants in systems with no failure [PFP06, Cav06].

Remark 6.3 (Paying Taxes). *We assume that nodes can and do pay their taxes. If non-payment of taxes is a concern, perhaps due to the "crazy millionaire" problem discussed in Section 6.3.1, the referee may require that nodes submit a monetary* bond *before starting the algorithm. Node i's maximum bond amount can be conservatively calculated as:*

$$\sum_{-i}(V^{max} - V^{min})$$

This amount, easily calculated by the referee before it starts the consensus, is the worst-case amount that node i will need to pay in taxes, which occurs when its preferred choice is selected, but its presence causes the choice to switch from the common "best" of every other node to the common "worst" of every other node.

If there are failures, the referee runs a recovery and penalty enforcement algorithm. During failure handling, the referee has the power to:

- certify a fault or halt condition, which may involve querying nodes to check for halt.
- choose an appropriate penalty, and notify the bank accordingly.
- restart the consensus algorithm to exclude fault or halt exhibiting nodes from the algorithm.

Looking at each of these in turn:

Certifying Faults and Halts

Faults, as opposed to halts, are presented to the referee as proofs. These proofs are signed inconsistent message bundles, as discussed in the previous section. Proof bundles cannot be faked, because their components are signed by the faulty node.

Halts involve more work on the part of the referee. A halt check can be triggered in one of two ways. The first way is that a node i can submit a halt claim to the referee, declaring the suspected node j and the type of information that node i is expecting to receive. The second way is that the referee can decide that too much time has passed with no consensus result from an economy, and instigate a halt check on its own. The referee is able to estimate when consensus should end by bounding the communication and computation time of any step in the consensus algorithm, and by multiplying these times by the number of messages and computations in a correct consensus. When this time expires, the referee executes a binary search along the participant vector to find the node who is not responding. Correct nodes reply to the referee with a message indicating their status in the algorithm. If node i claims to be waiting for information that is provided to the referee from some node j, the referee gives this missing information to node i and resets its timeout. Only if a truly halted node is identified will that node be penalized and excluded. In this case, any correct nodes that reported the true halt node are rewarded. The referee is thus able to identify a halting node or enforce progress when a node is delayed.

Rewards and Penalties

The referee rewards nodes that correctly detect problems, and penalizes nodes that do not report detectable problems reported by others, or falsely report problems that do not exist. Byzantine and obedient nodes' behavior are (by definition) not affected by the choice of penalty or reward. The incentives must be chosen so that it is always in a rational node's self-interest to report an actual fault, and never attempt to forge a fault.

How does the referee choose the incentives? Nodes actually tell the referee the minimum reward required to guarantee that any rational node will choose to report an observed failure. This reward R_i for node i is:

$$2(V_i^{max} - V_i^{min}) + \epsilon$$

where ϵ is a small positive value. To see this, let the true consensus choice be C but let node j cause a fault that forces the choice to be C'. We seek a reward R_i for node i that bests all possible benefits to node i of node j's manipulation. Node i would maximally benefit if $V_i(C') := V_i^{max}$ and $V_i(C) := V_i^{min}$, and any tax due from node i is reduced to zero. An upper bound on tax is $V_i^{max} - V^{min}$, which should be interpreted as the maximum amount that node i can hurt the selection of a different choice preferred by other nodes in the consensus. (In the most expensive tax case, node i declares a large positive V_i^{max} for its selected choice and large negative V_i^{min}s for other choices.) Setting node i's reward to be greater than the possible benefit equals $(V_i^{max} - V_i^{min}) + (V_i^{max} - V_i^{min}) + \epsilon$. Later, we'll see why V_i^{max} and V_i^{min} will be reported truthfully.

Node j, identified as faulty, is fined the reward and excluded from the consensus algorithm. Any other node k in i's clique was (by algorithm construction) also active and should have identified the fault. If node k does not do so, it in turn is considered faulty, fined the reward, and excluded from the consensus algorithm. Finally, if any node falsely claims a failure that is a bad proof or cannot be validated by the referee it is fined ϵ.

There is one subtlety to the reward structure. In equilibrium, the referee will reward a reporting node with a small ϵ, instead of the potentially large R_i calculated above! This will be explained in the proof of reliable deviation reporting in Section 6.6.3.

Restarting Consensus

At any point in the algorithm, the referee may send a restart interruption to all nodes. This restart signals that the referee has confirmed a failure and is excluding the failed node. All other nodes will restart.

Remark 6.4. *It is tempting to ask if the referee can be distributed across the same network of rational nodes that runs the consensus. We believe the answer is that there will always need to be* some *obedience in the protocol. Our reasoning is that part of the referee's job is to coordinate tax payments. With a distributed referee, there would need to exist a mechanism to check that rational nodes correctly implement the tax payment coordination. And then there would need to exist a mechanism that ensures that these checkers checked correctly. (The problem continues ad infinitum.) One option is to distribute the referee across rational nodes with no interest in the problem outcome. However, in this chapter, we opt for a centralized referee that has very little work in the (expected) common no-fault case.*

Remark 6.5. *The reader might wonder why, if we assume some trusted point, can we not let this trusted node perform the complete consensus? One reason for distribution is scale, as the distributed consensus breaks down the main and marginal economy calculations into tractable chunks that can be handled by many nodes in small pieces. Distribution is appropriate when there is a large amount of local information, such as in RaBC's tax calculation. Another reason may be privacy; with distribution, a center does not learn the details of nodes' private values. Greenstadt has evaluted the related DPOP algorithm and shows the advantages of distribution as measured by privacy loss [GGS07].*

6.5 Analysis, Implementation, and Evaluation

6.5.1 Experimental Setup

Nodes and the referee are implemented according to the default strategies described earlier in this chapter. Our implementation takes advantage of the following fault-recovery optimization: a referee restart interruption need not completely restart the consensus. Nodes can re-use previously known value-choice information across restarts, and all calculations, up to and not including the cliques involving the failed node. The actual RaBC clients are distributed as Python client scripts to make it easy for any user to change node behavior. Of course, a rational user should choose to run the suggested implementation of RaBC as-is.

Paxos Comparison

Lampson provides a short overview for those readers unfamiliar with Lamport's original Paxos protocol [Lam98, Lam01]. Paxos poses a consensus problem in terms of three classes of agents: *proposers*, *acceptors*, and *learners*. True to their names, the proposers propose values, the acceptors together work to select a single consensus value, and the learners are those nodes that learn the selected value from the acceptors. Participants in the Paxos protocol select a proposer as the leader of a decision problem, who in turn communicates with acceptors to produce a consistent decision.

For experiments involving Paxos, we chose to implement a "modern" version of the Paxos protocol in the form of FaB Paxos [MA05]. FaB Paxos [MA05] is a recent update of the Paxos protocol that guarantees safety even in the Byzantine fault model and has been optimized for the non-faulty case. We wrote our own implementation of the FaB Paxos protocol based on the pseudo-code listings and descriptions given in Martin and Alvisi's original paper.

In the Paxos-based experiments found in this chapter, one node becomes the leader and selects the choice that maximizes its value. We set the *proposal number* defined in Paxos to be the highest value that a node has for any choice, taking care that these values are unique across nodes. We follow the optimization heuristic given by Lamport [LSP82] where nodes abandon their attempt to become leader if they detect some other node with a higher proposal number. Paxos nodes execute in a random order, not tied to their proposal value.

MDPOP Comparison

For experiments involving MDPOP [PFP06], we wrote our implementation based on the pseudo-code listings and descriptions given in Petcu et al.'s original paper. Our MDPOP implementation follows the optimization suggested by Petcu et al. [PFP06] where cliques in marginal economies re-use any relevant work done in the main economy. An example of the basic algorithm was given in Section 6.2.2.

6.5.2 Logical Device Realizations

This section describes the implementation of the logical devices defined in Section 6.4.1.

Unforgeable signature device

Nodes use cryptographic public-key signatures [RSA83] to approximate an unforgeable signature device. Our algorithm therefore relies on a public key identity (PKI) layer implemented on top of Cryptlib [Cry07] that permits nodes to learn and verify each other's public keys. We assume that nodes are computationally bounded and (with very high probability) are unable to subvert the cryptographic primitives.

Bank device

Our bank is a "real" bank implemented using a commercial system called PayPal [Pay06]. Our referee implementation uses PayPal's SOAP programmatic interface to access node accounts. The referee, as the entity that activates payment transfers, makes direct calls to the PayPal bank.

Anti-collusion device

There are two techniques that we can use to mitigate collusion. Although we can do nothing to eliminate collusion between nodes before they enter the system, we can address collusion attempts that would occur after nodes enter the system and negotiate to form a collusive group. In order to mitigate this type of group collusion, we run our system on the Tor [DMS04] anonymizing infrastructure. Tor is an implementation of onion routing [Oni03], which enables users to communicate anonymously on the Internet. Tor provides a standard network interface, and our code runs on top as a normal network application. In order to provide true end-to-end anonymity and message filtering, we implemented a simple trusted Tor *relay service.* This relay service coordinates with the referee so that the relay can be an intermediate destination and source for inter-node messages. The relay service passes messages that conform to the limited vocabulary of the RaBC algorithm. Nodes send their messages to the relay service, which then strips sender information and relays the message to the true destination. Because of the strong anonymity properties of onion routing, the relay provides the only means for one node to contact another node. A collusive negotiation cannot occur out of band, since each RaBC participant only knows the true identity of a relay node and an outwardly meaningless coded identity at the other end of the conversation.

Section 6.4.1 also declared the need to rely on an identity check to prevent Sybil attacks. Requiring a node to have a unique bank account seems like a reasonable, though not foolproof, option. PayPal in the United States restricts accounts to one per social security number and has the outside machinery to enforce their policies.

6.5.3 Benchmarks

This section evaluates the effectiveness, impact, and cost of the RaBC remedy. We show RaBC's choice-optimality fault tolerance as calculated in the best/worst case and measured experimentally in the average case. We show RaBC's message requirements and message scalability, using Fast Byzantine Paxos (FaB Paxos) [MA05] and MDPOP [PFP06] as references in the no-fault case.

Our message analysis is **not** intended to show the "best" algorithm; such an interpretation is not appropriate because of the unbridgeable difference in fault models assumed by each algorithm. Rather, these comparisons convey the relative trade-offs facing the system designer in supporting different fault models and algorithms.

Choice-Optimality Fault Tolerance

How many faults can RaBC tolerate? RaBC's reliance on a referee as the final certifying authority assures traditional consensus correctness. However, aforementioned safety conditions [S4] and [S5] can be newly violated: we should understand the number of non-rationally motivated failures that will break these new conditions. We look at this measurement in a best, worst, and experimentally measured "average" case.

The level of tolerated failure is linked to the clique size q and the number of nodes in the system n. A pessimistic fault distribution would occur when all faults occur in the same clique; in this case, the algorithm can only tolerate $q-2$ faults to still support the *1-partial ex post Nash* solution concept that is used to prove faithfulness in Section 6.6. In the optimistic case, where faults are distributed uniformly to groups of nodes the size of a clique, the algorithm's fault tolerance is defined by:

$$f = \lfloor n/q \rfloor (q-2) + \begin{cases} n \bmod q & (n \bmod q) \leq (q-2) \\ q-2 & (n \bmod q) > (q-2) \end{cases}$$

The intuition behind the optimistic calculation is that the algorithm running over n nodes can support up to $q-2$ faulty nodes in every clique. There can be at most $\lfloor n/q \rfloor$ sequences of length q that are the maximal packing arrangement pattern. The right-hand

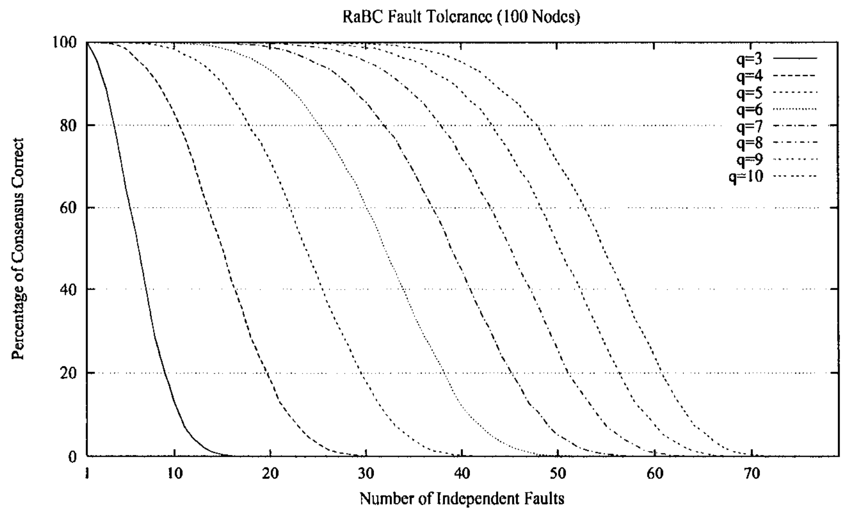


Figure 6.5: Experimental tolerance to faults as a function of clique size where failures are independent and randomly distributed.

term makes the bound precise in the case where the number of nodes in the system is not an integer multiple of the clique size.

Figure 6.5 reflects how the system's fault tolerance varies experimentally in the case where faults are independent and randomly distributed. From this experiment, we show the probability that a consensus decision is correct, subject to varying values of q, and f. For this experiment, the clique size q is varied. The number of total nodes n is 100. For each configuration, we perform 1000 complete independent RaBC consensus runs and plot the percentage of consensus runs that are executed correctly. This experiment is interesting in settings where a system is willing to choose the optimal consensus value *most* of the time.

For instance, when $q = 10$, the worst case fault tolerance is $q - 2 = 8$. However, there is a 99.9% probability that a correct consensus will occur succesfully with as many as $f = 26$ faults. If a probabilistic optimality is sufficient for the consensus algorithm, then q can be relatively small, which lowers the total RaBC message cost. The total message cost is shown in the next experiment.

Message Traffic

The message traffic benchmark shows the communication requirements of the MD-POP, FaB Paxos, and RaBC algorithms in the no-fault case. The results are shown in Figure 6.6. Tor underlay traffic is not included in our message traffic graphs.

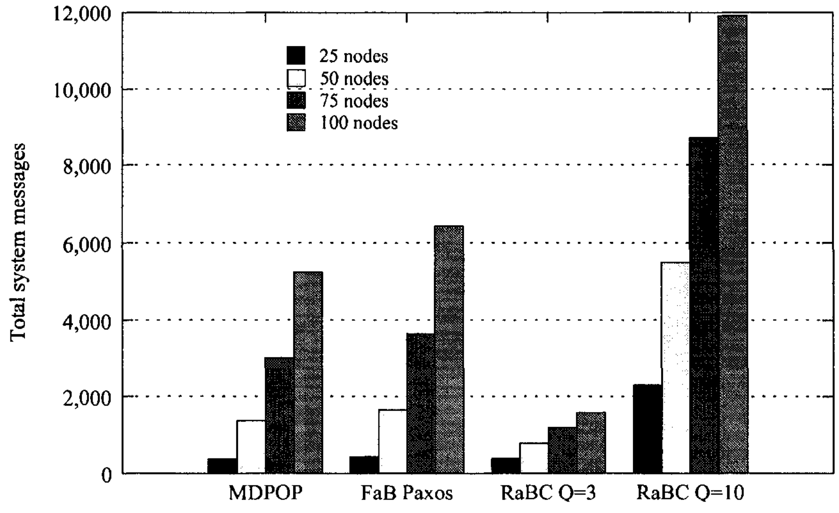


Figure 6.6: No-fault message complexity of MDPOP, FaB Paxos, and RaBC with Q=3 and 10 for various network sizes.

RaBC is run with clique sizes 3 and 10, respectively configured to support 1 and 8 (worst-case) non-rationally motivated failures. The figure shows how the cost of using RaBC increases when deployed for a larger number of possible non-rational faults for different network sizes n. RaBC generally requires many more messages than MDPOP or Paxos, but unlike MDPOP and Paxos, is always able to reach the system-optimal consensus even in the presence of rational and traditional faults. RaBC has less total message traffic than MDPOP (or Paxos) when the clique size is small (e.g., 3 in Figure 6.6). This traffic savings can be understood from Step 7 of Figure 6.2; in RaBC, consensus messages are generated in marginal economies only when the choice in the marginal economy is different from the choice in the main economy. Unlike MDPOP, which always must run the marginal economy calculation, message savings in RaBC is possible because of the incentive scheme that guarantees that a node's shared choice-value vector will be used correctly in the marginal economy choice calculation. The amount of FaB Paxos traffic does not depend on the level of desired fault tolerance; FaB Paxos can tolerate $\frac{n-1}{5}$ faults (between 4 and 19 failures with the choices of n shown). MDPOP would fail in the presence of non-rational faults.

RaBC Scalability

Figure 6.7 shows how RaBC's message traffic scales with clique size and the number of nodes in the system. RaBC is run with clique sizes 3 through 10, respectively configured to support 1 to 8 non-rationally motivated failures. The figure shows how the cost of

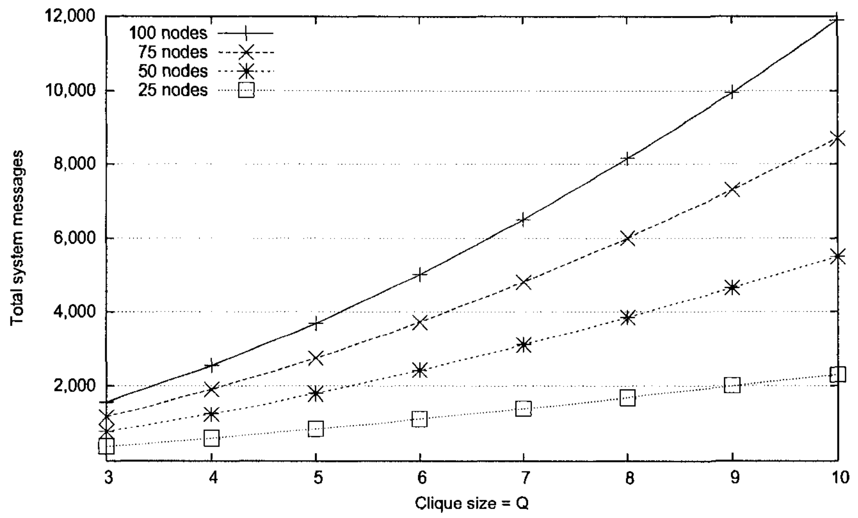


Figure 6.7: Message complexity as a function of clique size for various network sizes.

using RaBC increases when deployed for a larger number of possible non-rational faults for different network sizes n. While no faults were generated while running this experiment, the extra *fault message complexity* adds a total of $q + 2$ extra messages with the referee because of the *recovery optimization* discussed in Section 6.5.1.

6.6 Mechanism Proof

To show correctness, we must show safety and liveness. Our use of cryptography and reliance on the referee ensures that *traditional* consensus safety concerns [S1-S3] and liveness concerns [L1-L2] are met. However, the safety property we want to show is that the selected choice is the value-maximizing choice for all non-faulty nodes [S4]. A necessary part of this proof is to show that rational nodes choose to exhibit no faults [S5]. Thus we must show *faithfulness*, namely that rational nodes do not manipulate their information and choose to implement the consensus specification correctly.

This faithfulness proof is trickier than the proof of FPSS faithfulness in Chapter 5, because we now switch the solution concept from *ex post Nash* to *1-partial ex post Nash*. In other words, we now allow traditional failure. The faithfulness proof is broken up into the following steps:

- We start with a strategyproof centralized mechanism. In the centralized mechanism, consensus inputs are already reported truthfully by rational nodes *regardless* of faulty information revelation by other nodes.

- We then consider the distributed version of this centralized mechanism. Distributing the mechanism eliminates the claim of strategyproofness. Instead, in the distributed version of the mechanism, consensus inputs are reported truthfully by rational nodes assuming that the mechanism computation and message-passing is correctly implemented.

- To support claims of communication-compatibility (strong-CC) and strong algorithm-compatibility (strong-AC) for the 1-partial ex post Nash equilibrium, we must prove two mechanism properties:

    1. Any single fault deviation from the suggested algorithm can be identified by at least one other rational node in a clique.
    2. In equilibrium, a rational node will choose to report correctly at least one deviation that it observes and will not falsely report correct behavior as a deviation.

- We finally show strong-AC and strong-CC by proving that these sub-theorems ensure that a rational node should always follow its suggested message-passing and computation strategies.

The result is a distributed consensus algorithm that is faithful in the 1-partial ex post Nash equilibrium. In other words, faithfulness is assured as long as each clique contains at least two rational nodes.

6.6.1 Truthful Information Revelation

Proposition 6.1. *RaBC implements a strategyproof centralized mechanism in a distributed setting.*

Proof. The algorithm in Section 6.4 is a distributed implementation of the centralized Vickrey-Clarke-Groves (VCG) mechanism [Jac00]. The VCG mechanism provides a solution to resource allocation problems where the goal is to reach a decision that is globally value-efficient. Truthful reporting is a dominant strategy equilibrium. The distributed version of VCG that we use in RaBC is due to Petcu et al, and the proof of faithfulness in the distributed setting is given in Petcu et al. [PFP06]. Our addition of cliques does not affect this proof, as the cliques in RaBC simply add redundancy that is not used in the no-failure model assumed by Petcu et al. □

Petcu et al.'s proof relies on truthfulness properties of VCG, where no agent can affect its tax payment, and this tax payment is computed as the marginal impact of this node's presence on the rest of the nodes in the system. The effects of changing one's declared value can be to change the consensus choice or to change some other agent's tax payment. But changing the outcome leads to a correspondingly higher tax meaning that the deviation is not profitable.

6.6.2 Reliable Fault Detection

Before proving anything about correct message passing or algorithm execution, we first show that any deviations from the suggested mechanism can be detected by an observer node, assuming that at least one other node in a clique also chooses to follow the suggested mechanism. Our reliance on signing provides easy deviation detection. Section 6.4.7 listed the four types of faults that can occur in RaBC. The first of those, a *signing inconsistency*, is immediately verified when a node receives a message, and the received message acts as proof of the inconsistency. The remaining three faults are reliably detected. Figure 6.6.2 lists these faults, as if made by some node j, and detected by some node i. The table lists the reasoning node i can use to declare a fault. In the next section, we will show why a rational node would always want to report this proof to the referee. Because of cryptographic signing, these proofs are easy to generate and verify. In RaBC, a signed message is signed by other members of the clique before being processed. The unforgeable signature logical device used by RaBC means that the signing of a message cannot be faked.

Proposition 6.2. *An out-of-bounds input will be reliably detected by a rational node running the suggested RaBC algorithm in equilibrium.*

Proof. An out-of-bounds input implies that a node is attempting to declare a value V' that is either higher or lower than the initial declaration made to the referee in step 1 of the algorithm mechanics given in Section 6.4.2. This bounds information is sent by the referee to all other nodes in the faulty node j's clique. Then, it is easy to check if the values are out of bounds ($V' > V^{max}$ or $V' < V^{min}$), or if the high/low choices do not correspond to the high/low values. $((C' = C^{max} \;\&\; V' \neq V^{max})$ or $(C' = C^{min} \;\&\; V' \neq V^{min}))$. The observing node then uses node j's signed messages $(((C,V)^{max})_{\sigma_j}, ((C,V)^{min})_{\sigma_j}, (C',V')_{\sigma_j})$ to form a message proof-set that demonstrates the inconsistency. □

Proposition 6.3. *An agreement inconsistency will be reliably detected by a rational node running the suggested RaBC algorithm in equilibrium.*

Proof. An agreement inconsistency implies that some node j has performed a its portion of the value summation incorrectly. In this case, by step 4 of the algorithm mechanics

Deviation	Proof Set	Why Detected
Out of bounds input	$((C,V)^{max})_{\sigma_j}$ $((C,V)^{min})_{\sigma_j}$ $(C',V')_{\sigma_j}$	$V' > V^{max}$ $V' < V^{min}$ $(C' = C^{max}$ & $V' \neq V^{max})$ $(C' = C^{min}$ & $V' \neq V^{min})$
Agreement inconsistent	$(Sum)_{\sigma_i}$ $(Sum')_{\sigma_j}$ (+ Msgs)	Sum ≠ Sum'
Message inconsistent	$(Msg)_{\sigma_j}$ $((Msg')_{\sigma_j})_{\sigma_k}$ -or- $(Msg)_{\sigma_j}$ $(Msg')_{\sigma_j}$	Msg ≠ Msg'

Figure 6.8: Detectable deviations, why nodes can detect them, and the proofs they can offer to the referee.

given in Section 6.4.2, the observing node i uses received messages $(Sum')_{\sigma_j}$ and has its own computation $(Sum)_{\sigma_i}$, along with the signed component messages that together form a message proof-set that demonstrates the inconsistency. □

Proposition 6.4. *A message inconsistency will be reliably detected by a rational node running the suggested RaBC algorithm in equilibrium.*

Proof. A message is signed by other members of the clique before being processed by an observing node. The unforgeable signature logical device used by RaBC means that signing of a message cannot be faked. A message inconsistency implies that some node j has originated Msg to some observing node and Msg' to another node k. In this case, by step 4 of the algorithm mechanics given in Section 6.4.2, the observing node uses received messages $(Msg)_{\sigma_j}$ and $((Msg')_{\sigma_j})_{\sigma_k}$ to form a proof-set that demonstrates an inconsistency. □

Theorem 6.1. *Any single fault deviation from the suggested RaBC algorithm can be identified by at least one other node in a clique in equilibrium.*

Proof. This follows from the last three propositions and the fact that a message signing inconsistency can be detected. □

Note that this theorem only discusses fault deviations. Halt deviations, where a node prevents liveness by crashing or rationally deciding not to forward messages, do not give rational nodes enough information to form a proof. However, the algorithm in Section 6.4.2 specified that the referee can interrogate a node if a halt is suspected after a well-defined message timeout.

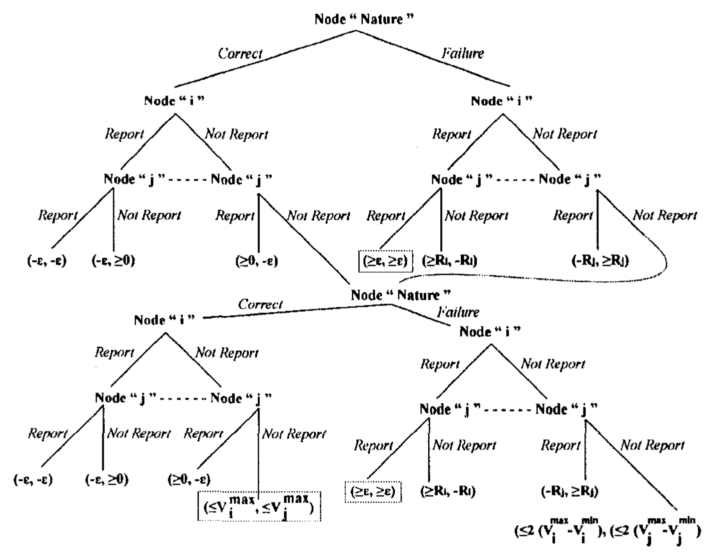


Figure 6.9: A portion of the extensive form representation of a reporting game facing two rational nodes and a third (potentially faulty) node called *Nature*. Equilibrium outcome nodes, found by eliminating non-credible threats, are boxed. This game is described in Section 6.6.3.

6.6.3 Reliable Deviation Reporting

Reliable deviation reporting is a *reporting game* played between two rational nodes in a clique, who respond to the actions of a third node labeled "Nature". By the 1-partial ex post Nash equilibrium concept, each clique must contain at least two nodes that are rational (or obedient).[4] An *extensive form* [FT91] representation of this reporting game is pictured in Figure 6.6.3. The reporting game is actually the last portion of the larger game consisting of that node's strategic plays in RaBC. Before the reporting game is started, nodes have revealed their information and have performed their portion of the RaBC algorithm. While we have not proven that a node's portion of the algorithm was executed correctly, we have shown in the last section that incorrect execution can be identified by rational nodes. Faulty behavior generated a proof, while halting behavior required intervention from the referee. The reporting game analysis shows that a rational node will choose to report a fault to the referee.

This reporting game runs in a sequential set of stages. Each stage is a complete instance relating to one of the four faults listed in Section 6.4.7. A node plays the reporting game with other rational nodes on the receipt of each "proof-set" of messages, which either

[4]There may be other rational nodes in the clique. In this case, multiple copies of this three-party game are played simultaneously, but no game affects the strategy of a node in any other game (since a node's reward and penalty are identical in every game).

is a proof-of-fault or a proof-of-correctness. The game is a sequential set of stages in that the same nodes have a series of reporting decisions; it is possible for a node to detect multiple failures and then decide to report only one of those failures. However, the result of *any* failure report is the same: the reporting node receives the same reward regardless of which fault it reports, and the algorithm restarts with the failed node is excluded.

The game has a final stage when all decisions are made for the set of two game players, and the node calculates an expected payoff. The payoff is expected (and not actual) because the payoff is only realized after all nodes in the system are finished playing the reporting game. So, for example, a node in the first clique must wait for all of the other nodes in other cliques to finish – and if those other nodes find problems in their own clique's computations, then the expected reward is canceled as the consensus restarts.

Two stages, representing two decisions, are illustrated in Figure 6.6.3. The second stage in Figure 6.6.3 is the final stage. Each stage starts by a move by a node labeled "Nature". This move either generates evidence of a fault, leading along the *Failure* state, or generates evidence of no fault, leading down the *Correct* path. Two game-playing nodes i and j then simultaneously react to the behavior of this third node. The actions of nodes i and j lead to *payoffs* for nodes i and j. The payoff for Nature is not shown; Nature represents the new existence of a proof-set that was already detected by the reasoning shown in the last section. Nodes i and j interact with the referee by their decision to *Report* or *Not Report* a claimed fault to the referee. On a *Report* action by any player, the referee imposes a reward or penalty and may exclude one or more nodes, and then continues the reporting game play.

Reward payoffs R_i and R_j are set as described in Section 6.4.8 to be

$$R_i = 2(V_i^{max} - V_i^{min}) + \epsilon$$

and

$$R_i = 2(V_j^{max} - V_j^{min}) + \epsilon$$

where ϵ is a small positive value, and $V_{i,j}^{max}, V_{i,j}^{min}$ are defined in Section 6.4.3 to be the maximum and minimum values that those nodes have for any choice. Given that the reward and penalty payoffs are set as shown in Figure 6.6.3, we show the following propositions.

Proposition 6.5. *A rational node has greater utility for reporting a deviation than from not reporting a deviation in equilibrium.*

Proof. Let the observer be node j. Node j must choose between *Report* and *Not Report.*

Choosing *Report* regardless of node i's decision leads to greater payoff ($\geq R_j$) vs. ($\leq 2(V_j^{max} - V_j^{min})$) and ($\geq \epsilon$) vs. ($-R_i$). Working backwards, given that node j chooses to *Report*, node i also chooses to *Report* to maximize its payoff.

□

Proposition 6.6. *A rational node's utility is the same regardless of the fault it elects to report in equilibrium.*

Proof. Let the observer be node j. Node j's reward is based only V_j^{max} and V_j^{min}, (or is the constant ϵ), which do not vary according to fault. □

Looking carefully at the game tree, one may think that some node j has an incentive to overstate V_j^{max} or understate V_j^{min}. Indeed, if a node believes that it will catch other nodes in a failure report, then it seems at first glance that the potential reward to node i is substantial and can be manipulated. However, in equilibrium this is not the case:

Proposition 6.7. *A rational node j has no incentive to misreport V_j^{max} or V_j^{min} in equilibrium.*

Proof. By the 1-partial ex post Nash equilibrium concept, at least one non-faulty node is in the game with node j. The equilibrium behavior is for both nodes to report the fault by Nature, in which case the reward to node i is always ϵ and does not depend on the reported value of V_i^{max} or V_i^{min}. □

Proposition 6.8. *A rational node receives lower expected utility for reporting correct behavior as a deviation in equilibrium.*

Proof. Let the observer be node j. Node j must choose between *Report* and *Not Report.* Choosing *Not Report* regardless of node i's decision leads to greater payoff (≥ 0) vs. ($-\epsilon$) Working backwards, given that node j chooses *Not Report*, node i also chooses *Not Report* to maximize its payoff. □

Proposition 6.9. *A rational node's expected utility is unchanged if a faulty node reports correct behavior as a deviation in equilibrium.*

Proof. While Nature's payoff is not shown in Figure 6.6.3, if Nature is correct and the referee is unable to validate a fault, it is the reporting node who is penalized and not Nature. □

6.6.4 Achieving strong-AC and strong-CC

Theorem 6.2. *A node will choose to report correctly at least one deviation that it observes and will not falsely report correct behavior as a deviation in equilibrium.*

Proof. By Theorem 6.1, any single deviation in RaBC can be identified by at least one other node in the clique. By the equilibrium condition of 1-partial ex post Nash, the identifying node must be either obedient or rational. If obedient, this theorem is trivially true by virtue of default behavior. If rational, by Proposition 6.5, the utility maximizing behavior is to report a deviation, and by Proposition 6.8, correct behavior will not be falsely reported.

To further validate the equilibria shown in Figure 6.6.3, we use a *subgame perfect Nash equilibrium analysis* to remove non-credible threats and reveal the only equilibrium outcomes. This analysis is performed by starting at the leaves of the tree, and for each decision subgame, asking how rational play would proceed. By induction, one locates the credible strategies and finds game equilibria. Following this process in Figure 6.6.3 reveals the game equilibria, which are shown boxed, assuming that Nodes i and j act rationally. (By the 1-partial ex post Nash equilibrium, it is guaranteed that at least one instance of this game will be played consisting of two rational nodes.) The equilibria are for Nodes i and j to both "not report" correct behavior, and to "report" faulty behavior. Because consensus will not occur if a node is reported as being faulty, consensus is reached in the single equilibrium with no failure. □

Corollary 6.1. *The suggested RaBC algorithm is strong-AC and strong-CC in a 1-partial ex post Nash equilibrium.*

Proof. Strong-AC and Strong-CC imply that a rational node should always follow its suggested message-passing and computation strategies. By Theorem 6.2, a rational node will report a deviation by any other node. Because of the assumption behind the 1-partial ex post Nash equilibrium that at least two rational nodes will be in a clique and able to observe each other's behavior, RaBC is strong-AC and strong-CC. □

Corollary 6.2. *RaBC has specification faithfulness.*

Proof. True because the corresponding centralized mechanism is strategyproof, and because RaBC is strong-AC and strong-CC. □

6.6.5 Effects of Collusion and Sybil Attacks

This proof assumes the logical dependencies listed in Section 6.4.1. While an *unforgeable signature device* and a *bank device* may seem obvious, in this section we comment on the need for an *anti-collusion device.*

A rational node would benefit with a Sybil attack [Dou02] if it could fake a second node with the same declared choice-value input. In effect, this behavior would ensure that node i had zero tax liability. We view this as a form of "self-collusion." In fact, any form of choice-value collusion can destroy the truthfulness of the VCG mechanism. This susceptibility is well known and previous work has suggested non-systems approaches to limiting the impact [ER91]. Unfortunately, there is no possible value-maximizing consensus algorithm where truthful value revelation is a dominant strategy that is immune to collusion. This strong negative result holds even if the size of a coalition is fixed [GJJ77]. In our device realization given in Section 6.5.2, we used systems techniques to reduce the chance of collusion. Specifically, an anonymity service is used to hide participants from each other. The goal is to remove the opportunity for collusion by keeping the identities of the participants hidden and by limiting their communication to known message formats.

6.7 Summary

This chapter used the Rational Byzantine Generals problem to demonstrate how incentives can be combined with traditional Byzantine Fault Tolerance (BFT) tools to prevent rationally motivated failure in information revelation, computation, and message passing, while staying robust to other Byzantine failures in computation and message passing.

6.8 Bibliographic Notes

Petcu and Faltings have produced a string of papers on distributed constraint optimization, such as DPOP [PF05]. Petcu has also released a single-machine simulator of the related DPOP algorithm [Pet06].

The algorithm in this chapter was inspired by the Bayou [PSTT96] distributed consistency research. That work is motivated with a meeting room scheduling application, but Bayou "is intended for use after a group of people have already decided" on the meeting details and "does not help [participants] determine a mutually agreeable place and time for a meeting." RaBC started as our method to solve this decision problem.

Chapter 7

Conclusions

7.1 Summary

This thesis formalized *faithfulness* as the metric by which to judge an algorithm's tolerance to rationally motivated failure. We identified *information revelation failure* as an important type of rationally motivated failure in distributed systems. This type of failure is well-known within economics but had been overlooked or mis-classified by the distributed systems community. We introduced the Rational Byzantine Generals problem in Chapter 2 and used this example to highlight the problems of rationally motivated failure in information revelation, computation, and message passing.

We examined how to prove faithfulness. A proof of faithfulness is a certification that rational nodes will choose to follow the algorithm specified by the system designer given certain assumptions and proved in an appropriate knowledge concept. In Chapter 3 we introduced the *k-partial ex post Nash* solution concept for systems where failures may still occur despite the designer's use of incentives. In Chapter 4 we demonstrated one way to prove that a distributed mechanism is faithful if the mechanism can be shown to be *strong communication-* and *strong algorithm compatible*, and if it has a corresponding centralized strategyproof mechanism. We gave a three-step methodology for showing these properties and for certifying mechanism faithfulness: First, the designer specifies the *target environment assumptions*, which dictate the node model, knowledge model, and network model. Second, the designer constructs a new system specification making sure to identify a subset of *rational compatible* behavior. The designer further specifies the *mechanism*, which is defined by a *strategy space* and an *outcome rule*, as well as a *suggested node strategy* for each *participant type*. Finally, the designer provides a rigorous faithfulness proof for a particular *solution concept* that holds in the chosen *environment* and evaluates the trade-offs required to implement a system that is robust to rationally motivated failure.

We applied the rationally motivated failure remedy methodology to two problems. In Chapter 5 we built an algorithm for the interdomain routing problem based on work by Feigenbaum et al. [FPSS02] (FPSS). We showed the enhanced algorithm to be faithful under the *ex post Nash* solution concept. We implemented the algorithm in simulation, showing that a node's message cost when running the faithful algorithm depends on its degree and equates to a 2x-100x message traffic increase on a real Internet topology. We showed how this overhead can be reduced to 2x-10x when high-degree nodes impose a cap on the number of neighbors.

In Chapter 6 we applied the rationally motivated failure remedy methodology to the Rational Byzantine Generals problem. We designed an algorithm called RaBC that is faithful under the *1-partial ex post Nash* equilibrium solution concept. In contrast to earlier rational-aware systems work (e.g., BAR [AAC$^+$05]), this work provides incentives for correct behavior in computation, message passing, *and* information revelation without placing limitations on types of Byzantine behavior or making assumptions about a node's willing participation in penance and penalty mechanisms. We implemented the algorithm over a network and compared the fault tolerance, message complexity, and scalability to MDPOP [PFP06] and a Byzantine-fault-tolerant version of the Paxos [Lam01] protocol. Our analysis conveys the relative trade-offs facing the system designer in supporting radically different fault models, showing for example that RaBC generally has higher message complexity but can actually beat the message complexity of MDPOP and Paxos while tolerating a small number of failures due to the redundancy of shared private information and the use of a certifying referee.

7.2 Future Work

7.2.1 Faithfulness of non-VCG based Systems

Both systems in Chapters 5 and 6 made use of the Vickrey-Clarke-Groves [Vic61, Cla71, Gro73] (VCG) mechanism, but the main results of this thesis hold even when decision problems use a different mechanism.

In many systems settings, the VCG mechanism is not ideal. It is well known that the VCG mechanism is not tolerant to collusion. Moreover the VCG mechanism requires infrastructure (a banking system) and assumptions (a participant's ability to value choices) that many systems designers find heavy and foreign. The advantage of the VCG mechanism is that it implements effective problem partitioning: The design restriction built into VCG is that a participant's outcome is not affected by the input that it feeds into the VCG algorithm.

Algorithms that do not require direct and truthful private information revelation have other mechanism options, and it is certainly possible to prove faithfulness under these looser constraints. In Chapter 4, we gave an example where a dependence on private information was eliminated: in the Savage et al. [SCWA99] TCP hacking example a message recipient must acknowledge a packet by echoing a specific number generated by the sender. In effect, the designer has restricted the set of manipulative strategies that can be effectively employed by a manipulative receiver making it easier to show faithfulness. One excellent candidate for future work is to apply the ideas of faithfulness to other protocols. Any system that faces rationally motivated failure is a fair candidate for a faithful protocol.

7.2.2 Revised Protocols: RaBC++?

There is also work to be done on upgrading existing protocols (such as RaBC) to remove remaining centralized components and to improve protocol efficiency. In particular, we are not completely satisfied with our reliance on a centralized bank. However, the problem of designing and implementing a decentralized bank running on a network of rational participants seems difficult. One also might seek to improve the efficiency of existing algorithms: it is possible to optimize RaBC to reduce its message requirements. One might seek a more efficient version of RaBC in the future, in the same way that FaB Paxos was developed to address performance concerns in the original Paxos.

7.2.3 Demographics of Rationality

It is anecdotally evident that rationally motivated failure can disrupt systems. Each of the systems examples in Chapter 1 demonstrates that selfish users exist and are selfishly willing to subvert an intended specification. But mere existence may not be too great of a threat; what a designer wants to know is how many users will exhibit rational behavior in a particular system. We suspect that the answer depends on the details of the system, such as the types and likelihood of rewards and penalties, the ease of manipulation, and the sophistication and demographics of users:

- What are the potential rewards for successful manipulation? What are the penalties if manipulation is detected? (Do I earn money or just extra CPU time? Do I get thrown out of the system or thrown in jail?)

- How difficult is the manipulation? As we observed in previous work [SPM04], "changing a command line parameter is easier than modifying a configuration file, which is easier than changing compiled opcodes in an executable file."
- How sophisticated are the users of a system? Are they able to recognize manipulation opportunities? Are they technically capable of carrying out a manipulation?

Default client software provides a suggested strategy to an algorithm participant. While there is a cost to modifying system-provided software, if the utility gained from such a modification is greater than the cost of modification, a rational participant will expend this cost. We expect that most algorithm participants will be obedient by default, and that the system as a whole can rely on this expected obedience to keep other participants in line. As incentives for rationally motivated failure increase, one may be able to plot rationality graphs (e.g., "reward vs. ease,") to demonstrate points at which previously obedient agents become rational agents.

As this thesis was being completed, the author posed this experiment to a group of Harvard EconCS group talk attendees: can one deploy a system where participants are given opportunities to manipulate the system, with varying manipulation costs and payoffs, to determine what percentage of participants is willing to hurt the system in order to achieve some selfish gain? Seuken et al. [SPP08] have run one version of this experiment, where two versions of a peer-to-peer file sharing system client named Triber [Del08] have been made available to the general public. One version of the client is programmed to act selfishly, while the other client is programmed to act altruistically. Participants are informed of this choice on initial client download, and are asked in an indirect and validated way if they are aware of the consequences of their choice. Initial results show significant user bases of both rational and altruistic participants in this file sharing setting.

7.2.4 Real Currency

As we wrote in Chapter 2, systems that rely on virtual currency often fail in their goal to convince rational users to play by the system rules. We blame these failures on economically closed virtual currencies that provide little value to participants. To pick some examples, the incentive schemes in the Mariposa and Mojonation examples of Chapter 1 failed for two reasons: The first problem was that participants did not know how to value their virtual currency. The second problem was a variation on "carrying coal to Newcastle" — some participants were flooded with virtual currency that could only be used to "pay" for something that they did not want.

Rather than use a meaningless currency, it is possible to use actual currency to ensure cooperation, as we have assumed in Chapters 5 and 6. Real currency is an interesting external incentive to address failures in systems where participants can incur real costs for following the official suggested behavior. For example, in both the Mariposa example and in the Rational Byzantine Generals problem, it is clear that participants incur a real cost for correctly following the system specification. An example of a closed, centralized system that uses currency is Google's Adwords sponsored search slot allocation problem [Goo08]. Some future distributed systems work, when tempted to use a virtual currency, should opt to use a real currency instead. We feel that there needs to be a fully flushed out and tested system that is able to evaluate the pros (cooperation) and cons (real world support structure, increased motivation to cheat) of a real currency in the distributed environment.

7.2.5 Mechanism Execution

One area of future work is in the area of *mechanism execution.* Just because a system designer deploys a faithful mechanism, there is often no reason why participants must stick to the results of the mechanism in what we call the *execution* phase on the algorithm. Consider the following three examples:

In an auction setting like eBay [eBa08], even after a buyer wins an item, the seller can refuse to sell the item to the winning bidder. In fact, a seller can disregard the results of the mechanism entirely, opting to sell the item to a bidder inside or outside the system.

In Chapter 5, we presented a system where participants should correctly compute and use transit packet payments. But there is no binding obligation that nodes will route packets along lowest-cost paths and choose to respect intended payments. Furthermore, there is no policing of the data messages: a message from Node A to Node B may consist of collusive messaging at the application level, and there is little that can done to prevent this application-level collusion at a systems level.

In Chapter 6, we described a system for picking a system-value maximizing choice, but there is no guarantee that participants will live by this choice after the mechanism is finished. In other words, if the system decision is that Node A should become the leader in a leader election problem, the owner of Node A can simply walk away and refuse to abide by the mechanism outcome.

These examples demonstrate the problems of enforcing mechanism execution: just because a designer sets up a mechanism, there is still no guarantee that participants at run-time will agree to abide by the results of the mechanism. This is a problem in real-life as well, but society uses *mechanism enforcement tools*, such as laws, a legal system and jail.

In computing, these enforcement tools are less clearly defined.

7.2.6 Additional Failure Expressions?

When we look back to Chapter 2, and specifically to the table of traditional distributed system fault expressions in Table 2.1, we notice that information revelation is not on the list, and for many years has not been viewed as an important type of failure expression. We wonder if other developments in the last twenty years would allow new types of failure expression to be added to that list, using the original criteria that the feasibility (what classes of problems can be solved?) and cost (how complex must the solution be?) of addressing the failure are distinct from other members on the taxonomy. Are there other participant behaviors besides information revelation that are newly important in system design?

7.2.7 Backtracing

The software executed by a particular participant's node is a representation of that participant's strategy. Each machine code instruction in a program can be thought of as representing a little bit of one or more components of a node's strategy. Each function call has the potential to emit an external action, and/or evaluate and change the perceived state of the world. A system is robust to rational manipulation, and therefore faithful, if the designer is able to give a node a piece of source code, a set of suggested run-time options, and state, "Feel free to modify this client as you see fit. One can prove that the best strategy you can follow is the strategy that I have given you."

In evaluating such a claim, one idea is to extend the technique of *backtracing program slices* [Wei81, Tip95] to help a designer find candidate *manipulation points* in actual source code. A program slice consists of the parts of a program that potentially affect the values computed at some point of interest. When it comes to rationally motivated failure, the points of interest are the goal states of the distributed algorithm. Traditional program slicing studies one user in a single program, whereas our use of backtracing focuses on at least two users interfacing with the same client software in a game-like setting. Our earlier work [SPM04] provided a proof of concept for backtracing, where we defined backtracing as follows:

```
For each type of rational participant in the system:
  For each communication pattern:
    For each goal state:
      Trace backwards through program logic.
```

```
Mark branches and node interactions as candidate manipulation points.
Examine and classify resulting points.
```

The result of this process is a set of backtracing graphs. The number of backtracing graphs depends on the complexity of the system. As such, backtracing is too labor intensive for large systems, since the depth of the backwards trace depends on the complexity of the program code. The graphs generated from a backtracing exercise contain candidate manipulation points that must be examined manually and to reveal the effects of a selfish manipulation. We observe that similar challenges also affected the viability of program slicing, but automated tools eventually helped program slicing become more mainstream and integrated into debugging software [Tip95]. No such backtracing tools exist for finding manipulation opportunities in source code, and we suggest this as an area of future work.

7.2.8 Repeated Games and Collusion-Proof Games

This thesis did not study either repeated games or collusion-proof games. A repeated game is a series of decision problems, in which a user can observe and learn from one decision problem to reason about strategy in a successive iteration of the decision problem. Collusion is the coordination of two or more parties to act as one participant.

These research areas are related: repeated games enable signaling between participants, which can lead to collusion. On the other hand, collusion in a single game can allow participants to learn about other users in ways that could have been discerned from watching a repeated game. We suspect that a whole separate thesis could be written to address faithfulness in repeated games that face collusion.

7.3 Final Words

Until this thesis, there had been surprisingly little work on rational behavior in the context of system fault tolerance. Earlier we speculated that rationally motivated failure had been hidden from designers until the relatively recent rise of the Internet. The significance of the Internet – as it applies to rationally motivated failure – is that is brings together a number of diverse participants running a diverse set of algorithms.

Rationally motivated failure will become more pronounced as participants' needs and designers' goals conflict. Our view is that systems researchers, who until recently have been able to work in a *fault-exclusion* mindset, will begin to adopt *fault-prevention* techniques to address rational behavior that violates system goals. Our expectation is that such rationally motivated failures will become a priority for designers as the stakes determined by distributed algorithms become more important and as algorithms are run on more diverse sets of participants. Running over rational participants is unavoidable in many algorithms and this thesis is one important step in helping designers cope with the associated challenges.

Bibliography

[AAC+05] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR Fault Tolerance for Cooperative Services. *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, 2005.

[ABC+02] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 1–14, New York, NY, USA, 2002. ACM.

[ACSV04] Alvin AuYoung, Brent N. Chun, Alex C. Snoeren, and Amin Vahdat. Resource allocation in federated distributed computing infrastructures. In *Proceedings of OASIS 2004*, 2004.

[ADGH06] Ittai Abraham, Danny Dolev, Rica Gonen, and Joe Halpern. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 53–62, New York, NY, USA, 2006. ACM.

[Afe03] Michael Afergan. Repeated game analysis of internet routing (extended abstract). In *Poster session at 19th ACM Symposium on Operating Systems Principles (SOSP '03).*, St. Johns, Newfoundland, Canada., October 2003.

[Afe06] Michael Afergan. Using repeated games to design incentive-based routing systems. In *INFOCOM*. IEEE, 2006.

[AH00] Eytan Adar and Bernardo Huberman. Free Riding on Gnutella. *First Monday*, 5(10), October 2000.

[AJ06] Christina Aperjis and Ramesh Johari. A peer-to-peer system as an exchange economy. In *GameNets '06: Proceeding from the 2006 workshop on Game theory for communications and networks*, page 10, New York, NY, USA, 2006. ACM.

[BB02] Sonja Buchegger and Jean-Yves Le Boudec. Performance analysis of the confidant protocol. In *MobiHoc '02: Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing*, pages 226–236, New York, NY, USA, 2002. ACM.

[Ben23] Jeremy Bentham. *An Introduction to the Principles of Morals and Legislation.* W. Pickering, 2rd edition, 1823.

[BH03] Levente Buttyán and Jean-Pierre Hubaux. Stimulating cooperation in self-organizing mobile ad hoc networks. *Mob. Netw. Appl.*, 8(5):579–592, 2003.

[BOI06] BOINCStats. Why do you participate in BOINC? `www.boincstats.com/page/poll_results.php?id=8`, 2006. [Online; accessed 07-April-2008].

[Bra02] Felix Brandt. A verifiable, bidder-resolved auction protocol. In *Proceedings of the 5th International Workshop on Deception, Fraud and Trust in Agent Societies (Special Track on Privacy and Protection with Multi-Agent Systems)*, pages 18–25, 2002.

[BS04] Felix Brandt and Tuomas Sandholm. (Im)possibility of unconditionally privacy-preserving auctions. In *Proc. of the 3rd AAMAS*, 2004.

[BSS07] Felix Brandt, Tuomas Sandholm, and Yoav Shoham. Spiteful bidding in sealed-bid auctions. In Manuela M. Veloso, editor, *IJCAI*, pages 1207–1214, 2007.

[BT04] Ronen I. Brafman and Moshe Tennenholtz. Efficient learning equilibrium. *Artificial Intelligence*, 159(1-2):27–47, 2004.

[BW01] Felix Brandt and Gerhard Weiß. Antisocial Agents and Vickrey Auctions. In *Pre-proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, pages 120–132, 2001.

[Cav06] Ruggiero Cavallo. Optimal decision-making with minimal waste: Strategyproof redistribution of VCG payments. In *Proc. of AAMAS*, Hakodate, Japan, May 2006.

[CF05] Alice Cheng and Eric Friedman. Sybilproof reputation mechanisms. In *P2PECON '05: Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*, pages 128–132, New York, NY, USA, 2005. ACM.

[cFo06] cFos. cFosSpeed TCP Driver, 2006. `www.cfos.de`.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of OSDI*, 1999.

[Cla71] Edward H. Clarke. Multipart pricing of public goods. *Public Choice*, 11:17–33, 1971.

[CMN02] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: making backup cheap and easy. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 285–298, New York, NY, USA, 2002. ACM.

[CN03] Landon P. Cox and Brian D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 120–132, New York, NY, USA, 2003. ACM.

[Coh03] Bram Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer to Peer Systems*, June 2003.

[Cry07] Cryptlib security toolkit v3.3.2. `http://www.cs.auckland.ac.nz/~pgut001/cryptlib/`, 2007. [Online; accessed 22-April-2008].

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. In *Journal of the ACM*, volume 43, pages 225–267, March 1996.

[Del08] Tribler home page, 2008. [Online; accessed 25-April-2008].

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proc. of the 13th USENIX Security Symposium*, August 2004.

[Dou02] John Douceur. The Sybil Attack. In *Proc. of IPTPS*, March 2002.

[eBa08] eBay. eBay Home Page, 2008. `www.ebay.com`.

[Eli02] Kfir Eliaz. Fault tolerant implementation. *Review of Economic Studies*, 69:589–610, 2002.

[ER91] E. Ephrati and J.S. Rosenschein. The Clarke tax as a consensus mechanism among automated agents. In *Proc. of AAAI*, 1991.

[FCSS05] Michal Feldman, John Chuang, Ion Stoica, and Scott Shenker. Hidden-action in multi-hop routing. In *EC '05: Proceedings of the 6th ACM conference on Electronic commerce*, pages 117–126, New York, NY, USA, 2005. ACM.

[FHK06] Eric J. Friedman, Joseph Y. Halpern, and Ian Kash. Efficiency and nash equilibria in a scrip system for p2p networks. In *EC '06: Proceedings of the 7th ACM conference on Electronic commerce*, pages 140–149, New York, NY, USA, 2006. ACM.

[FKSS01] Joan Feigenbaum, Arvind Krishnamurthy, Rahul Sami, and Scott Shenker. Approximation and collusion in multicast cost sharing. In *Proc. of the 3rd Conference on Electronic Commerce*, pages 253–255, 2001.

[FLSC04] Michal Feldman, Kevin Lai, Ion Stoica, and John Chuang. Robust incentive techniques for peer-to-peer networks. In *EC '04: Proceedings of the 5th ACM conference on Electronic commerce*, pages 102–111, New York, NY, USA, 2004. ACM.

[FPSS02] Joan Feigenbaum, Christos Papadimitriou, Rahul Sami, and Scott Shenker. A BGP-based mechanism for lowest-cost routing. In *Proceedings of the 2002 ACM Symposium on Principles of Distributed Computing*, pages 173–182, 2002.

[FRS06] Joan Feigenbaum, Vijay Ramachandran, and Michael Schapira. Incentive-compatible interdomain routing. In *EC '06: Proceedings of the 7th ACM conference on Electronic commerce*, pages 130–139, New York, NY, USA, 2006. ACM.

[FRS07] Eric Friedman, Paul Resnick, and Rahul Sami. Manipulation-resistant reputation systems. In Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay Vazirani, editors, *Algorithmic Game Theory*, chapter 27. Cambridge University Press, 2007.

[FS02] Joan Feigenbaum and Scott Shenker. Distributed Algorithmic Mechanism Design: Recent Results and Future Directions. In *Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 1–13, 2002.

[FSS07] Joan Feigenbaum, Michael Schapira, and Scott Shenker. Distributed algorithmic mechanism design. In Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay Vazirani, editors, *Algorithmic Game Theory*, chapter 14. Cambridge University Press, 2007.

[FT91] Drew Fudenberg and Jean Tirole. *Game Theory.* MIT Press, 1991.

[GG07] Nandan Garg and Daniel Grosu. Faithful distributed shapley mechanisms for sharing the cost of multicast transmissions. In *Proc. of the 12th IEEE Symposium on Computers and Communications (ISCC-07)*, pages 741–747, July 2007.

[GGS07] Rachel Greenstadt, Barbara Grosz, and Michael D. Smith. SSDPOP: improving the privacy of DCOP with secret sharing. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–3, New York, NY, USA, 2007. ACM.

[GJJ77] Jerry Green and Jean-JacquesLaffont. Characterization of satisfactory mechanisms for the revelation of preferences for public goods. *Econometrica*, 45:427–438, 1977.

[GK99] Richard J. Gibbens and Frank P. Kelly. Resource pricing and the evolution of congestion control. In *Automatica*, volume 35, pages 1969–1985, 1999.

[GK06] S. Dov Gordon and Jonathan Katz. Rational secret sharing, revisited. In Roberto De Prisco and Moti Yung, editors, *SCN*, volume 4116 of *Lecture Notes in Computer Science*, pages 229–241. Springer, 2006.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229, New York, NY, USA, 1987. ACM.

[Goo08] Google. Adwords home page, 2008. [Online; accessed 25-April-2008].

[GR01] Lixin Gao and Jennifer Rexford. Stable internet routing without global coordination. *IEEE/ACM Trans. Netw.*, 9(6):681–692, 2001.

[Gro73] Theodore Groves. Incentives in teams. *Econometrica*, 41:617–631, 1973.

[GS02] Gerd Gigerenzer and Reinhard Selten, editors. *Bounded Rationality: The Adaptive Toolbox.* The MIT Press, Cambridge, MA, USA, 2002.

[GSW02] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.*, 10(2):232–243, 2002.

[GW99] Timothy Griffin and Gordon T. Wilfong. An analysis of BGP convergence properties. In *SIGCOMM*, pages 277–288, 1999.

[HCW05] Daniel Hughes, Geoff Coulson, and James Walkerdine. Free riding on gnutella revisited: The bell tolls? *IEEE Distributed Systems Online*, 6(6), 2005.

[HT04] Joseph Halpern and Vanessa Teague. Rational secret sharing and multiparty computation: extended abstract. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 623–632, New York, NY, USA, 2004. ACM.

[HTK98] Michael Harkavy, J. D. Tygar, and Hiroaki Kikuchi. Electronic auctions with private bids. In *WOEC'98: Proc. of the 3rd conference on USENIX Workshop on Electronic Commerce*, 1998.

[IIKP02] John Ioannidis, Sotiris Ioannidis, Angelos D. Keromytis, and Vassilis Prevelakis. Fileteller: Paying and getting paid for file storage. In Matt Blaze, editor, *Financial Cryptography*, volume 2357 of *Lecture Notes in Computer Science*, pages 282–299. Springer, 2002.

[IML05] Sergei Izmalkov, Silvio Micali, and Matt Lepinski. Rational secure computation and ideal mechanism design. In *FOCS '05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 585–595, Washington, DC, USA, 2005. IEEE Computer Society.

[JA05] Seung Jun and Mustaque Ahamad. Incentives in bittorrent induce free riding. In *P2PECON '05: Proc. of the 2005 ACM SIGCOMM Workshop on Economics of P2P Systems*, pages 116–121, New York, NY, USA, 2005. ACM.

[Jac00] Matthew O. Jackson. Mechanism theory. In *The Encyclopedia of Life Support Systems*. EOLSS Publishers, 2000.

[Jan00] John Jannotti. Network Services in an Uncooperative Internet. `http://arstechnica.com/reviews/2q00/networking/networking-1.html`, 2000. [Online; accessed 07-April-2008].

[Kah01] Leander Kahney. Cheaters bow to peer pressure. `http://www.wired.com/science/discoveries/news/2001/02/41838`, 2001. [Online; accessed 29-May-2008].

[Kat] Glez Katz. K-hack: The ultimate kazaa hack. [Online; accessed 28-May-2008].

[KFH07] Ian A. Kash, Eric J. Friedman, and Joseph Y. Halpern. Optimizing scrip systems: efficiency, crashes, hoarders, and altruists. In *EC '07: Proceedings of the 8th ACM conference on Electronic commerce*, pages 305–315, New York, NY, USA, 2007. ACM.

[KSGM03] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the Twelfth International World Wide Web Conference (WWW)*, 2003.

[Kur02] Klaus Kursawe. Optimistic byzantine agreement. In *SRDS*, pages 262–267, 2002.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[Lam01] Butler Lampson. The ABCD's of Paxos. In *Proc. of PODC*, 2001.

[LF82] Leslie Lamport and Michael J. Fischer. Byzantine generals and transactions commit protocols. Technical Report Opus 62, SRI International, Menlo Park, California, 1982.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[LMSW06] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. In *5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA*, November 2006.

[LNKZ06] Nikitas Liogkas, Robert Nelson, Eddie Kohler, and Lixia Zhang. Exploiting bittorrent for fun (but not profit). In *Proc. of IPTPS*, 2006.

[LRA$^+$05] Kevin Lai, Lars Rasmusson, Eytan Adar, Li Zhang, and Bernardo A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 1(3):169–182, 2005.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[LSZ06] Hagay Levin, Michael Schapira, and Aviv Zohar. The strategic justification for BGP. Technical report, Leibniz Center for Research in Computer Science, 2006. [Online; accessed 28-October-2007].

[LT06] Anna Lysyanskaya and Nikos Triandopoulos. Rationality and adversarial behavior in multi-party computation. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 180–197. Springer, 2006.

[Lyn96] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[MA05] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. In *Proceedings of DSN*, June 2005.

[McC01] Jim McCoy. Mojo nation responds. `http://www.openp2p.com/pub/a/p2p/2001/01/11/mojo.html`, 2001. [Online; accessed 29-May-2008].

[MGLB00] Sergio Marti, T. J. Giuli, Kevin Lai, and Mary Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 255–265, New York, NY, USA, 2000. ACM.

[Mic08] As topology project. `topology.eecs.umich.edu/data.html`, 2008. [Online; accessed 07-April-2007].

[MPS03] Robert McGrew, Ryan Porter, and Yoav Shoham. Towards a general theory of non-cooperative computation. In *TARK '03: Proceedings of the 9th conference on Theoretical aspects of rationality and knowledge*, pages 59–71, New York, NY, USA, 2003. ACM.

[MRWZ04] Ratul Mahajan, Maya Rodrig, David Wetherall, and John Zahorjan. Experiences applying game theory to system design. In *PINS '04: Proceedings of the ACM SIGCOMM workshop on Practice and theory of incentives in networked systems*, pages 183–190, New York, NY, USA, 2004. ACM.

[MT99] Dov Monderer and Moshe Tennenholtz. Distributed games. *Games and Economic Behavior*, 28(1):55–72, 1999.

[MT02] John C. Mitchell and Vanessa Teague. Autonomous nodes and distributed mechanisms. In Mitsuhiro Okada, Benjamin C. Pierce, Andre Scedrov, Hideyuki Tokuda, and Akinori Yonezawa, editors, *ISSS*, volume 2609 of *Lecture Notes in Computer Science*, pages 58–83. Springer, 2002.

[Mu'05] Ahuva Mu'alem. On decentralized incentive compatible mechanisms. In *EC '05: Proceedings of the 6th ACM conference on Electronic commerce*, pages 240–248, New York, NY, USA, 2005. ACM.

[Nas50] John Nash. Equilibrium points in n-person games. In *Proceedings of the National Academy of Sciences*, volume 36, pages 48–49, 1950.

[Nee93] Roger M. Needham. Cryptography and secure channels. In *Distributed systems (2nd Ed.)*, pages 531–541. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[Net07] Sharman Networks. Kazaa. `http://www.kazaa.com`, 2007. [Online; accessed 07-April-2007].

[NR99] Noam Nisan and Amir Ronen. Algorithmic Mechanism Design. In *Proceedings of the 31st ACM Symposium on Theory of Computing*, pages 129–140, 1999.

[NR00] Noam Nisan and Amir Ronen. Computationally feasible VCG mechanisms. In *Proc. 2nd ACM Conf. on Electronic Commerce (EC-00)*, pages 242–252, 2000.

[NWD03] Tsuen-Wan Ngan, Dan S. Wallach, and Peter Druschel. Enforcing fair sharing of peer-to-peer resources. In M. Frans Kaashoek and Ion Stoica, editors, *IPTPS*, volume 2735 of *Lecture Notes in Computer Science*, pages 149–159. Springer, 2003.

[Oni03] Onion content delivery network. `http://onionnetworks.com/`, 2003.

[Ore08] Route views project web page. `http://www.routeviews.org/`, 2008. [Online; accessed 07-April-2007].

[Pap01] C H Papadimitriou. Algorithms, games and the Internet. In *Proc. 33rd Annual*

ACM Symp. on the Theory of Computing, pages 749–753, 2001.

[Par01] David C. Parkes. *Iterative Combinatorial Auctions: Achieving Economic and Computational Efficiency.* PhD thesis, University of Pennsylvania, May 2001.

[Par04] David C. Parkes. Harvard university computer science 286r: Computational Mechanism Design, 2004. http://www.eecs.harvard.edu/~parkes/cs286r/syllabus.html.

[Pay06] PayPal. PayPal API Documentation, 2006. `www.paypal.com`.

[PCAR02] Larry Peterson, David Culler, Tom Anderson, and Timothy Roscoe. A blueprint for introducing disruptive technology into the internet. citeseer.nj.nec.com/peterson02blueprint.html, 2002.

[Pet06] Adrian Petcu. FRODO: A FRamework for Open/Distributed constraint Optimization. Technical Report No. 2006/001, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, 2006. http://liawww.epfl.ch/frodo/.

[PF05] Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 266–271. Professional Book Center, 2005.

[PFP06] Adrian Petcu, Boi Faltings, and David C. Parkes. MDPOP: faithful distributed implementation of efficient social choice problems. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1397–1404, New York, NY, USA, 2006. ACM.

[PIA$^+$07] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do incentives build robustness in bittorrent? In *NSDI'07*, Cambridge, MA, April 2007.

[Plu75] Plutarch. Ship of theseus – the Internet Classics Archive. `http://classics.mit.edu/Plutarch/theseus.html`, 75. [Online; accessed 5-September-2008].

[PS04] David C. Parkes and Jeffrey Shneidman. Distributed implementations of vickrey-clarke-groves mechanisms. In *Proc. 3rd Int. Joint Conf. on Autonomous Agents and Multi Agent Systems*, pages 261–268, 2004.

[PSST01] Adrian Perrig, Sean W. Smith, Dawn Xiaodong Song, and J. D. Tygar. Sam: A flexible and secure auction architecture using trusted hardware. In *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 170, Washington, DC, USA, 2001. IEEE Computer Society.

[PSTT96] Karin Petersen, Mike Spreitzer, Douglas B. Terry, and Marvin Theimer. Bayou: replicated database services for world-wide applications. In Andrew Herbert and Andrew S. Tanenbaum, editors, *ACM SIGOPS European Workshop*, pages 275–280. ACM, 1996.

[Rei95] Michael K. Reiter. The rampart toolkit for building high-integrity services. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, pages 99–110, London, UK, 1995. Springer-Verlag.

[RET02] Tim Roughgarden and Éva Tardos. How bad is selfish routing? *J. ACM*, 49(2):236–259, 2002.

[RKZF00] Paul Resnick, Ko Kuwabara, Richard Zeckhauser, and Eric Friedman. Reputation systems. *Commun. ACM*, 43(12):45–48, 2000.

[RO01] Alvin E. Roth and Axel Ockenfels. Last-Minute bidding and the rules for ending second-price auctions: Evidence from eBay and Amazon auctions on the internet. *American Economic Review*, 2001. forthcoming.

[Rot05] Alvin E. Roth, editor. *The Shapley Value: Essays in Honor of Lloyd S. Shapley.* Cambridge University Press, 2005.

[Rou01] Tim Roughgarden. Designing networks for selfish users is hard. In *Proc. 42nd Ann. Symp. on Foundations of Computer Science*, pages 472–481, 2001.

[RSA83] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 26(1):96–99, 1983.

[SAL$^+$96] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A wide-area distributed database system. *VLDB Journal: Very Large Data Bases*, 5(1):48–63, 1996.

[Sch93] Fred B. Schneider. What good are models and what models are good. In Sape Mullender, editor, *Distributed Systems*, chapter 2. Addison Wesley, 1993.

[Sch95] Bruce Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[SCWA99] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP congestion control with a misbehaving receiver. *Computer Communication Review*, 29(5), 1999.

[Sel86] Reinhard Selten. Evolutionary stability in extensive two-person games correction and further development. Discussion Paper Serie A 70, University of Bonn, Germany, May 1986.

[SH03] Michael Stonebraker and Joseph M. Hellerstein, editors. *Readings in database systems.* Morgan Kaufmann, 3rd edition, 2003.

[SNP$^+$05] Jeffrey Shneidman, Chaki Ng, David C. Parkes, Alvin AuYoung, Alex C. Snoeren, Amin Vahdat, and Brent Chun. Why markets could (but don't currently) solve resource allocation problems in systems. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 7–7, Berkeley, CA, USA, 2005. USENIX Association.

[SP04] Jeffrey Shneidman and David C. Parkes. Specification Faithfulness in Networks with Rational Nodes. In *Twenty-Third Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2004)*, July 2004.

[SPM04] Jeffrey Shneidman, David C. Parkes, and Laurent Massoulie. Faithfulness in Internet Algorithms. In *Proceedings of 3rd SIGCOMM workshop on Practice* *and Theory of Incentives in Networked Systems (PINS).* ACM, 2004.

[SPP08] Sven Seuken, Johan Pouwelse, and David Parkes. Selfishness vs. altruism in p2p networks: A large-scale economics field experiment. In *In preparation for submission.*, 2008.

[ST03] Yoav Shoham and Moshe Tennenholtz. Non-cooperative evaluation of logical formulas: The propositional case., 2003. Unpublished Manuscript.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[Tor07] TorrentFreak. GreedyTorrent: The Deadly BitTorrent Sin. `http://torrentfreak.com/greedytorrent-the-deadly-bittorrent-sin/`, 2007. [Online; accessed 07-April-2007].

[Use05] MacNN Users. New altivec-enhanced seti worker in need of testing (message thread). `http://forums.macnn.com/72/team-macnn/266339/new-altivec-enhanced-seti-worker-need/`, 2005. [Online; accessed 29-May-2008].

[Use06] Rosetta@Home Users. Connection Error (message thread), 2006. `http://boinc.bakerlab.org/rosetta/forum_thread.php?id=883`.

[Use07] ArsTechnica Users. BOINC Optimized client - still an advantage? (message thread). `http://episteme.arstechnica.com/eve/forums/a/tpc/f/122097561/m/328001213831`, 2007. [Online; accessed 29-May-2008].

[Vic61] William Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16:8–37, 1961.

[vOWK07] P.C. van Oorschot, Tao Wan, and Evangelos Kranakis. On interdomain routing security and pretty secure bgp (psbgp). *ACM Trans. Inf. Syst. Secur.*, 10(3):11, 2007.

[Wei81] Mark Weiser. Program slicing. In *Proc. of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.

[Wei92] William E. Weihl. Specifications of concurrent and distributed systems. In Sape Mullender, editor, *Distributed Systems, 2nd Ed.*, chapter 3. Addison Wesley, 1992.

[WHH$^+$92] Carl A Waldspurger, Tad Hogg, Bernado Huberman, Jeffrey O Kephart, and W. Scott Stornetta. Spawn: A distributed computational economy. *IEEE Trans. on Software Engineering*, 18:103–117, 1992.

[ZCY03] Sheng Zhong, Jiang Chen, and Richard Yang. Sprite: A simple, cheat-proof, credit-based system for mobile ad-hoc networks. In *INFOCOM*, 2003.