Gossiptron

Efficient Sharing on the Grid without Central Coordination

Jaap Weel

Gossiptron

Gossiptron

Efficient Sharing on the Grid Without Central Coordination

Jaap Weel

Thesis for the degree of
MSc in Parallel and Distributed Computer Systems
Supervised by Dr. Guillaume Pierre
Co-supervised by Guido Urdaneta

Department of Computer Science
Faculty of Exact Sciences
Vrije Universiteit Amsterdam

Abstract

To succeed at increasing the value of computational resources by pooling them across participants, grid computing needs a way of distinguishing resource allocations that increase value from ones that do not. I present an algorithm called GOSSIPTRON that is a fully decentralized grid resource allocator and replacement for Oner's peer-to-peer grid scheduler that reduces opportunities for free riding and other abusive patterns of resource use to acceptable levels without getting in the way of effective scheduling. To do so, GOSSIPTRON employs gradual escalation in one-on-one barter relationships, such as in BitTorrent, but extends the concept to transitive barter. I have implemented a simulation of the algorithm, compared results among several of its variants, and analyzed the remaining vulnerability to abuse. GOSSIPTRON maintains effective scheduling and reduces opportunities for abuse to acceptable levels.

Contents

Chapter 1

Introduction

According to CERN's *GridCafé* web site, "the Grid is a service for sharing computer power and data storage capacity over the Internet" [1]—but the concept of "sharing" covers up as much as it evokes.[1] As a participant in the grid, a user may be much more eager to see share the computational resources of others than to share their own. If the grid is to accommodate a wide variety of participants, each with their own goals and requirements and resources, then we must make arrangements to keep the grid from becoming a pool that everyone can draw on but that nobody bothers to contribute to.

1.1 Goals

This thesis describes an attempt to construct and evaluate a resource allocation algorithm for grids that is (1) decentralized and (2) efficient, even when some nodes in the grid act strategically. In the remainder of this section, we will examine what these goals mean and why they are necessary conditions for building a grid that is truly a useful tool for sharing computational resources.

Decentralized allocation

It is one of the goals of this project that the algorithm for allocating grid resources must be decentralized, or "peer-to-peer."

A peer-to-peer system is different from a regular distributed system. A regular distributed system runs on a number of separate communicating machines, but typically there will still be a small subset of the nodes in the network that has a special coordinating role. For instance, on the Internet, the DNS root servers are a small subset of IP hosts that have a special role in keeping track of the IP address top-level DNS servers. Peer-to-peer systems, by contrast, dispense with centralized coordination altogether.

Peer-to-peer systems can be harder to design than traditional distributed systems, so it is worthwhile to consider precisely why we would want a grid resource allocation system to be peer-to-peer. There are three major advantages to building it in that way.

[1] Parts of this chapter have previously been used for a proposal submitted as part of the Research Proposal course at Vrije Universiteit.

Table 1.1: Symbols used in this thesis

σ	The number of (not necessarily distinct) neighbors each node has
κ	The number of those neighbors exchanged upon each gossip transaction
ϕ	Time between neighbor gossip episodes
γ	Time between schedule gossip episodes
G	Generosity parameter
E	Escalation parameter
D	Depreciation parameter
T	Maximum transitivity
N_a	Neighbor array (primary overlay) for node a
V_a	Neighbor set (secondary VORONET overlay) for node a
A_a	Initially empty agenda for node a
S_a	Initially empty node schedule for node a
$U_a[b]$	UOI balance of node b with node a
$I_a[b]$	IOU balance of node b with node a
$C_a[b]$	Credit rating of node b with node a
$\mathscr{L}_a[b]$	Credit limit of node b with node a
i_j	Globally unique job ID for job j
m_j	Size of required team for job j
d_j	Time required for job j
t_j	Time job will start for job j
c_j	The job's coordinator node for job j
T_j	Team for job j, including cache of team members' node schedules
$\mathscr{V}(a)$	Voronoi cell corresponding to node address a
h	Hash function to translate node addresses to points in $\mathbb{R}^2$ for spatial routing
ℓ	Hash function to determine additional long-range links for spatial routing
$\mathbb{R}$	The set of real numbers
$[0,1)$	The set of real numbers between 0 and 1, including 0 but not 1.

The first advantage of a peer-to-peer system is that it eliminates a potential single point of failure: the resource allocator. Because grids can be very large, having a single point of failure is not a good idea. By distributing resource allocation, we can build a fault-tolerant system that will not break down simply because a small number of nodes break down.

Aside from avoiding a single point of failure, a peer-to-peer system has a second advantage. It avoids a single *bottleneck*, or point of congestion. Even if no nodes ever fail outright, it is still a bad idea to have all the resource allocation computations involved in resource allocation be done by a single node, simply because resource allocation computations consume more and more computational resources as the grid grows, while the single resource allocation node still has a fixed amount of resources to support that computation. In a peer-to-peer resource allocation system, unlike in a centralized one, we add more resource allocation capacity automatically as we add more nodes, and in that way we avoid having a single point of congestion.

Thirdly and finally, in this thesis, we assume that all nodes, including any central resource allocation node, may act strategically in their own interest, rather than precisely according to the rules we set. In that situation, a decentralized system becomes not only a good idea, but indeed crucial. An allocator acting strategically for its own interest, rather than reliably according to a set algorithm that is designed to serve a common good, can completely defeat the purpose of having a grid in the first place, which is to serve the mutual benefit of participants, and not the individual benefit of an allocator node.

Efficient allocation

The second stated goal of this thesis project is to achieve an efficient allocation of resources in the presence of strategic nodes. By "efficiency" I mean wealth maximization. Wealth maximization is a property of allocations of goods. Specifically, an allocation is wealth maximizing (or Kaldor-Hicks efficient, to use an even less descriptive term) if there are no more possible reallocation of resources that could increase total wealth, that is, it would harm those that it harms less than it helps those that it helps, measured by how much those that it harms are willing to pay to avoid it, and by how much those that it helps are willing to pay to get it. This may appear cryptic now, but we will encounter the concept of efficiency in more detail in chapter 2.

Why efficiency is important

I focus on efficiency as a criterion in this thesis because it is the very *raison d'être* of a grid. When participants choose to join a grid, they do it because the transaction creates wealth. If that is case, then the grid will be used, and otherwise, it will not.

Free riding and efficiency

A common source of inefficient allocations is free riding. Free riding is the practice of drawing on a shared resource without contributing to it. In the grid, free riding is obviously an issue. In most grid arrangements, several sites participate and each site can both submit tasks to be performed using the common grid resources, and contribute resources for the execution of tasks submitted by others. If there are sites that consistently submit tasks for others to execute, but do not contribute any local resources to execute jobs submitted by others, then those sites are trying to take a free ride. In that case, it is quite possible that prohibiting them from participating until they make a contribution will portend an improvement (in the sense to be further specified in chapter 2). That means that the situation where free riding occurs is inefficient.

Free riding in centralized and peer-to-peer systems

In the case of a centralized allocation algorithm, it is possible to prevent free riding from within the algorithm by enforcing explicit norms of fairness. Each node can be prevented, for instance, from delegating more jobs to other nodes than it accepts itself, on average. This can even be accomplished by some mechanism external to the allocation algorithm itself—a grid accounting service maybe, or simply informal norms of courtesy haphazardly enforced by system administrators.

In a peer-to-peer system, there is no central allocator. We cannot produce an efficient outcome simply by calculating an efficient allocation and imposing it on all participants, but instead we must devise rules of local peer-to-peer interaction that can lead to efficiency in the aggregate. Fortunately, we know from economics that, under certain assumptions, such systems, called markets, can exist. For a market-like grid to work, we need to assume that each node has the option to refuse any request to contribute its resources, that is, it has right of ownership over its local computational resources. Once the mechanism of ownership is in place, we also need to provide policies for each node to decide how to exercise its right of ownership.

1.2 Prior work

Before considering in more detail the approach we will take to decentralized, efficient allocation, we must examine what relevant work has been done on this topic in the past. Distributed grid resource allocation and the economics of grid resource allocation have both been studied, but not in combination. The present project can be seen as an attempt to tie together the two strands of research, incorporating techniques inspired both by other (non-grid) peer-to-peer systems and by social science.

Work on decentralized grid scheduling

We will draw on the work of Caglar Oner on decentralized grid scheduling. [12] Oner has developed a distributed algorithm for a grid computing system made up of nodes connected by a network as well as a peer-to-peer overlay constructed using gossip based peer sampling to resemble a random graph. [15, 11] A job, which must be performed by some given number of nodes simultaneously, is initially scheduled by the coordinator node for that job, based on local information that each node maintains about the schedules of its direct neighbors. In our implementation, the coordinator will simply be the node that originates a job, and there will therefore be as many coordinators as there are nodes originating jobs. Once the initial schedule has been computed and communicated to all participating nodes, it is then gradually refined as each proposed participant in the job communicates with its neighbors to find ways of scheduling it earlier.

Work on grid economics

We will also draw on prior work on the topic of grid economics. Most work in this field attempts to take into account the price that participants are willing to pay and the price at which participants are willing to make resources available. Typically, a centralized resource allocator is used that either uses an auction protocol or tries to explicitly calculate a market equilibrium using the methods of textbook microeconomics. (See e.g. [16].) Much of this research would meet the goal of ours but for the presence of the central allocator. Another line of research, sometimes referred to as the *catallactic* approach, attempts to remove the central allocator, but it appears to rely on the presence of a currency with which nodes in the system can trade among themselves. (See e.g. [8].) Such a system is either not robust against the simple attack of forging the currency, or else relies on a central "bank" (such as GridBank [4]) to execute all monetary transactions.

1.3 Approach

The mechanisms and policies used to accomplish decentralized, efficient research allocation are the topic of chapters 4–6, but before we lose ourselves in discussing more background knowledge, let me briefly outline the approach taken in this thesis.

Focus on allocation

We are concerned only with achieving an efficient allocation of resources in the presence of nodes that strategically try to manipulate the allocation mechanism. We shall not be concerned with the equally interesting but distinct problem of nodes that play

along with the allocation mechanism and then proceed to sabotage the tasks they have committed to performing, for example by returning the wrong results for a computation. The latter issue is an interesting and important concern, as well, but it is outside the scope of this thesis, which is about resource allocation, not grid security.

Resource ownership

In order to achieve efficient allocation, we must introduce the notion of ownership of computational resources. In Oner's original algorithm, it was assumed that nodes would only refuse jobs in case of a scheduling conflict. This is unrealistic: nodes that act strategically may pretend at times that there is a scheduling conflict when there is none. At the same time it is undesirable, because without the ability to refuse jobs for reasons other than scheduling conflicts we cannot prevent inefficient transactions from occurring. Instead of making any assumptions about which jobs will be accepted, I have designed my system on the assumption that any node may or may not want to reject any job—for its own reasons.

Blind trust

Once the basic mechanism of local ownership of local resources has been thus established, we need to look into what particular policy the nodes should employ to exercise their newly found discretion. The very simplest case is for nodes to accept any and all incoming requests, except in case of a scheduling conflict. This corresponds exactly to Oner's original algorithm. It is obviously vulnerable to a single participant in the grid submitting an enormous amount of work to the grid and not doing any.

Direct reciprocity

We need cleverer policies than that, and the first one of these, which we will encounter in section 4.2, is for nodes to accept jobs on the basis of direct reciprocity. This is similar to the Tit-for-Tat approach taken in Axelrod's classic study of the evolution of cooperation [2] and in the popular file sharing protocol BitTorrent [7]. Accepting jobs based on direct reciprocity means that a node A will only do work for a different node B if in the past, B has done work for A. In other words, requests for resources are evaluated by asking of the requesting party, "what have done for me lately?" Obviously, if this is the only principle we have got, cooperation will never get started. In order to jump-start cooperation, we need to modify every node's strategy to grant a small amount of blind generosity to get the process of escalating reciprocity going. (In BitTorrent, this is called "optimistic unchoking.") All this is described in section 4.2.

Local reciprocity

Unfortunately, the simple reciprocity algorithm, if it has to involve offers of blind generosity to any and all potential collaboration partners, is vulnerable to exploits. As the grid grows, so does the number of nodes that one can be ripped off by. Direct reciprocity just does not interact very well with Oner's scheduling algorithm. The number of relationships a node must maintain is bounded only by the size of the grid, and each relationships requires initial, potentially unreciprocated, favors to be bestowed on a stranger. These favors can quickly add up to a considerable free ride for the strangers.

In order to solve this free rider problem, a different, more complex approach is proposed in section 4.4. In that chapter, we will constrain the set of nodes that a node can directly interact with to include only its immediate neighbors. This is similar to the spatial models used by Axelrod [2], where participants were also placed in a spatial structure and allowed to trade only with neighbors.

Transitive reciprocity

Imposing spatial structure on the grid pretty much solves the problem of blind generosity being abused, but it still does not interact well with Oner's allocation algorithm. In order to find the first available team for a job, the allocation algorithm must sample the node schedules of a large number of nodes in the network. Very often, this makes it necessary for nodes to interact even though they do not happen to be neighbors. Restricting communication to neighbors may reduce vulnerability to exploits, but at the same time it impairs the ability of the resource allocation algorithm to find good allocations.

There is a way around this. If a node C—not being one of A's neighbors—needs A to do something, it may well be the case that A owes a favor to its neighbor B, and B owes a favor to its neighbor C. What we want to have is a way to discharge simultaneously A's obligation to B and B's obligation to C by having A do a job for C. This would considerably extend the number of partners that A can *effectively* deal with, even if it only *directly* deals with its neighbors. That way, we can circumvent the problem introduced by restricting transactions to neighbors only, without at all increasing the number of nodes that need to be eligible for exploitable blind generosity.

Money or routing

At this point we should note that the most common real world solution to both the problem of delayed reciprocity and the problem of transitive reciprocity does not involve finding explicit chains of obligations between neighbors, but rather the use of *money*. With money, rather than repaying past favors done to us or done to others

who have in turn done favors to us, we simply replace the mutual bestowing of favors by a series of immediate one-on-one exchanges of labor for money.

As it happens, scarce money, however naturally it comes to humans in nearly all cultures, is hard to implement in a computer system, where all we have to work with are bits. Bits are just too easy to copy. There exist approaches to try to approximate the unique properties of money using complex distributed algorithms, but they are typically quite complex indeed. In this thesis we shall not seek to implement a monetary system. Instead, we will employ a routing algorithm to explicitly find paths along which transitive trade is possible.

1.4 The rest of this document

In the remainder of this thesis, we shall first consider, on an informal level, the economics of public goods, in chapter 2. Next, we shall consider in chapter 3 Oner's algorithm for decentralized grid scheduling. That algorithm is not designed with strategic agents in mind, but it will form the basis for our further explorations. Once we have acquired a good understanding of the original, naïve version of Oner's algorithm, we proceed in section 4.1 to add a notion of resource ownership and in section 4.2 an exceedingly simple strategy for exercising those rights based on direct reciprocity. We will modify this strategy in section 4.3 to limit the number of neighbors that each node needs to grant initial blind generosity to, and we will discover that while solving the free rider problem, this change also limits the quality of schedules. Finally, we will attempt to remedy that problem of schedule quality, without bringing back the free rider problem, by implementing transitive reciprocity through explicit routing in section 4.4. Once we understand all these strategies, we will proceed to have a look at how they perform in simulation in chapter 5. Finally, in chapter 6, we discuss what significance the results and insights described in this thesis have for resource allocation on grids, and what further research might be inspired by them.

Chapter 2

A little economics

In this chapter, we will encounter an informal summary of some economic concepts that will illuminate the problem of decentralized, efficient resource allocation, on the grid and otherwise, and that will inspire the solutions we will encounter in later chapters.

2.1 Exchange

Let us start at the very beginning, with a story about Adam, a customer, and Eve, a greengrocer. Adam has a coin, but he does not care much for coins. He would much rather have an apple than a coin. Eve, the greengrocer, is holding an apple, but she has plenty of other apples, and frankly she would much rather have the coin than the apple.

Adam gives Eve a coin, and Eve hands Adam an apple. Adam says "thank you," and Eve in turn says "thank *you*." Adam is holding an object that is more valuable (to him) than the one he held before—and Eve is holding an object that is more valuable (to her) than the one that she held before.

In the instant of exchange, a small act of economic creation has occurred. *We cannot escape the conclusion that both Adam and Eve are now wealthier than they were before, and there is nobody around who has become poorer. Therefore, wealth must have been created.*

This phenomenon is sufficiently miraculous on the surface to warrant closer scrutiny. After all, nothing physical has come into existence that did not exist before, and yet we want to claim that something has been created.

From a physical point of view, all that happened was a shuffling around of pre-existing stuff: one apple, and one coin, to be precise. The reason that wealth has nonetheless been created is that wealth is in the eye of the beholder. Wealth does not consist in things, but in the value that persons attach to things. Adam and Eve disagree about the value of the objects. For Adam, the apple was worth more than the coin, and for Eve, the coin was worth more than the apple. It is precisely that disagreement that made it possible for them to engage in a wealth producing act of voluntary exchange.

Put differently, exchange creates wealth by allocating goods to those who value them most, as measured by what they are willing to give up to get them. If we put it like that, we need only two additional pieces of information to estimate how much value has been created in our economic parable: the most amount of money that Adam would have been willing to pay for an apple, and the least amount of that money Eve would have been prepared to sell it for. These amounts of money are called, respectively, Adam's reservation price and Eve's reservation price. The difference between the reservation prices is called surplus, and it measures the amount of wealth created in the transaction.

2.2 Property

In concluding that the exchange between Adam and Eve was mutually beneficial, we have so far made one very crucial hidden assumption: the exchange was voluntary.

To illustrate what happens when exchange is not voluntary, consider a variant of the story where Adam takes the apple from Eve's tree without asking and drops a coin in her mailbox as an unsolicited payment. Maybe this is a mutually beneficial exchange just like the previous one. Maybe it is not. But there is no way of knowing. We know Adam ostensibly liked the apple better than the coin, but from the mere fact of the exchange we can draw no conclusions about whether Eve preferred the coin to the apple.

This brings us to the first important wrinkle on the story we encountered in the previous section: when we observe a change in allocation, we can only be sure that it improved wealth if it was a voluntary exchange, that is, an exchange respecting property rights.

2.3 Wealth maximization

We have seen that voluntary exchange is necessarily mutually beneficial, or it would not happen, and therefore any voluntary exchange is wealth increasing. We do not yet know, though, whether by simply allowing voluntary exchange, we can create a wealth *maximizing* allocation.

Actually, before we try to find out more about how to achieve wealth maximization, let us first examine David Friedman's definition of this important concept, which he also sometimes likes to call "Marshall efficiency."

> We consider a change (the abolition of tariffs, a new tax, rent control, ...) that affects many people, making some worse off and others better off. In principle we could price all of the gains and losses. We could ask each person who was against the change how much money he would have to be

> given so that on net the money plus the (undesirable) effect of the change would leave him exactly as well off as before. Similarly we could ask each gainer what would be the largest amount he would pay to get that gain, if he had to. We could, assuming everyone was telling us the truth, sum all of the gains and losses, reduced in this way to a common measure. If the sum was a net gain, we would say that the change was a Marshall improvement. If we had a situation where no further (Marshall) improvement was possible, we would describe it as efficient. [9]

There are many criteria of besides wealth maximization that one might use to evaluate an allocation, but there is one major reason why wealth maximization is a an attractive one: it can often be achieved in a decentralized way. This may seem like reasoning backwards, but what good is a criterion, for our purposes, if it can be achieved only by centralized control? Not only is central allocation something we set out explicitly not to design, but also it raises the tremendous problem of enforcing the mandates of the centralized allocator.

It is not generally true of just any property of allocations that we could dream up that it can be achieved, or even approximated, in a decentralized way. If the criterion were to count, for instance, benefits to orphans double and veterans triple, but otherwise maximize wealth, it would be hard to imagine a mechanism to bring this about reliably that does not involve giving up control of allocation to a central allocator.

The reason that the criterion of wealth maximization is more suitable for distributed implementation than any other known criterion is that it relies on the values of things by how much agents are willing to give up to acquire (or avoid) them. Given property rights as a "boundary condition", agents have incentives to act precisely as though they were trying to maximize value in that sense, but there is no readily imaginable equivalent to property rights that gives people incentives to maximize value in the "orphans double, veterans triple" sense.

2.4 Obstacles to wealth maximization

We can expect all mutually beneficial exchanges to happen if all involved have *full information* about each other's preferences, and if they face *no transaction costs*, especially in cases where beneficial exchange requires coordination among many people.

We do not necessarily get wealth maximization if information is incomplete, because in that case exchanges can be prevented from happening that would have been mutually beneficial. If Adam is a devoted connoisseur of antique teapots, and Eve has an antique teapot she uses but does not particularly care for, then a possible act of wealth creation could be prevented if Eve did not know of Adam's love of teapots.

Similarly, we do not necessarily get wealth maximization when there exist potential transactions that have costs associated with them. Such costs, called transac-

tion costs, can be anything from sales taxes or the risk of highway robbery to the cost of coordinating agreements needed for complex transactions among large groups of people and enforcing those agreements. Take the example of 10,000 town residents who would have to agree on contributing to construction of a road that all of them—including the miser who wouldn't pay—will be able to use once it is built. In that case, transaction costs may be prohibitive or even infinite, and the wealth-increasing creation of the road may not happen.[1]

2.5 Markets and money

Full information seems like a fairly outrageous assumption to make. Not even Adam and Eve are likely to have full information about each other's preferences, and the billions of participants in modern markets definitely have no such knowledge.

But they have money as a common unit of account. Once we reduce all transactions to immediate exchanges of money for goods, suddenly we drastically reduce the amount of information that participants need to have in order to discover potential beneficial exchanges. Rather than having to know how every other person values every good we have in terms of every good they have, we only need to know how much money they are willing to give for each good we have. Moreover, we do not need to know this information for every buyer separately, because in a market situation, prices will tend toward an equilibrium: once buyers know the going price for a good, they will typically not buy it for more even if they value it higher than the going price, just because they do not need to, and it is nicer to be able to pocket the difference. Similarly, once sellers know the going price for a good, they will typically not sell it for less even if they value it lower than the going price, just because they do not need to, and it is nicer for them, too, to be able to pocket the difference.

In the presence of a market, the information that needs to be known in order for agents to be able to discover all beneficial transactions is reduced to a single number—the market price—for each good. Finding that one number may still involve any number of auctions, assessors, inspectors, market researchers, and consumer reports booklets, but the fact that in principle the going price can summarize all that is worth knowing about the market in a given good goes a long way toward explaining how it can be that efficiency is ostensibly approached quite closely in the real world.

Unless... there are significant transaction costs. Obviously, even if a market price is observable, there still may exist potential beneficial transactions that are costly for

[1] This is one of the reasons why roads are often built by governments (the other being that it enriches politicians). Governments do not have to operate within the confines of property rights, but can instead just tax all residents, including the miser who wouldn't pay, and spend it on a road. Such mechanisms as government have their uses and their dangers, but both are outside of the scope of this thesis, since they do not meet the requirement that all mechanisms be decentralized.

other reasons. For example, we still have the case of the town of 10,000 where everybody would be happy to help buy out a polluter *if everyone contributed*, but somehow they cannot get their act together. Even if active markets exist in polluting factories and in environmental cleanliness and it is obvious that there exists a price to be paid that would make both the factory owner and the town residents better off, the transaction costs are still there.

Nonetheless, money can help here, too. One particular source of transaction costs is the problem of delayed reciprocity. Sometimes, there may be no instantaneous wealth increasing transactions, but there is a possibility for one agent to perform a service or donate a good to another agent in the expectation of a reciprocating "gift" later on. Such transactions can have great potential, but they do require that promises for future trade be enforceable. This is usually thought of as a transaction cost: promises for future trade can be made, but enforcing them has a cost. One way to deal with that is to replace a pair of time-separated gifts with an instantaneous transfer of goods for money now, and another instantaneous transfer of money for goods later. This way, money can help reduce transaction costs as well as provide a means of spreading information.

2.6 Back to grid resource allocation

Now that we are armed with a bit of background economic knowledge about efficiency, markets, information, and transaction costs, we can look at our grid allocation problem through an economic lens and discover that it breaks apart in two pieces.

2.7 Surplus on the grid

For our grid computing project, we will assume that all participants have some computational resources to share. This should make us wonder why anybody would prefer to use the grid rather than their own computational resources. If an organization or an individual needs some amount of computing power, why can't they just buy the computers needed to supply it? Sometimes, the answer is that indeed that is what they do, and they will not use a cooperative grid of the type we are discussing in this thesis. (That need not prevent them from calling whatever type of computer cluster they use a "grid," but for our purposes, a grid is only interesting if it involves several independent participants sharing resources.)

There are several reasons why individuals and organizations may nonetheless want to pool computational resources. The most important of these is probably temporary and irregular demand. Purchasing and maintaining a cluster that can entirely serve your own needs may be a good idea if you have a steady predictable workload that

will last for the next 5 years at least, but in many cases, people need computational resources for short-term projects, or at irregular intervals, or only during the day, or only at night.

Take, for instance, two research groups that want to run simulations. One is in Amsterdam and the other is in Sidney. For both of them, the demand for computational resources is higher during the day than at night, that is, the amount they are willing to pay per CPU-hour is more during the day than at night. It so happens that when it is day in Amsterdam, it is night in Sidney, and vice versa. This means that there is surplus to be had: during the Dutch day, the Sidney group has a resource that the Amsterdam group values more highly than the Sidney group does; during the Australian day, the Amsterdam group has a resource that the Sidney group values more highly than the Amsterdam group does. By exchanging some of their resources, they can make each other better off. It's Adam and Eve all over again: the very disagreement over the value of a CPU-hour of computational power *at a particular time of day* is what gives rise to the possibility of mutually beneficial economic interaction. This is one of the major ways in which a grid can be a wealth producing mechanism.

2.8 Property on the grid

In the real world, a common criticism of market mechanisms is that they are not as decentralized as they seem, because somebody has to enforce the property rights. On the grid, we are in luck: computer networks already tend to be set up with security measures that make it (nearly) impossible for the unauthorized to use any given machine. This means that we do not need to worry, at least in this document, about maintaining property rights. All we need to do, and will do in chapter 4, is to let nodes assert the property rights that in a sense they already had.

2.9 Money on the grid

Given the praise of money that we sung earlier in this chapter, we might expect to be able to use the concept of money to great effect on the grid. But there are problems.

Money works because it is a common unit of account that everybody can trust to be valuable. This requires that it is somehow guaranteed to be scarce, and indeed, we observe most cultures using for money either valuable scarce natural resources such as precious metals, salt, and shells, or else documents that are guaranteed by reputable banks, or governments, or others that monopolize and restrict the supply of such documents. When a natural resource becomes no longer as scarce as it used to be, or the institution behind an artificially scarce resource ceases to effectively restrict its supply, the resource quickly becomes worthless.

On computer systems, money is exceedingly hard to implement correctly. It can be done in a centralized way, but to do money in a distributed way leads us straight back into the issues of trust and reciprocity we were trying to avoid by introducing money. Unlike with property rights in computational resources, which are readily enforced in a distributed way, there is no obvious way to enforce in a distributed manner the rule that money shall not be counterfeited.

Nonetheless, one alternative route not pursued here to reaching our goal might be to build on top of prior attempts, most notably a project called KARMA, to introduce a type of scarce resource to peer-to-peer systems that is monitored for counterfeiting in a distributed way. [14, 10]

2.10 Transaction costs on the grid

Based on our understanding of economics, we can identify two potential obstacles on the road to efficient distributed resource allocation on the grid. Both of these obstacles can be interpreted as transaction costs, although it may be more customary to call the first a "transaction cost" and the second an "information cost".

Trust problem Instantaneous mutually beneficial transactions do not usually exist on the grid. Rather, what we have is a sort of delayed mutual backscratching, providing some resources now in the hope of receiving some later on. Such transactions might be transformed by the use of money into instantaneous transaction, but we just decided that money in a digital context is hard to implement, and that we therefore want to try to get along without introducing money. There is a positive transaction cost associated with enforcing contracts for future delivery. This is just a complicated way of saying that agents don't necessarily keep their promises.

Information problem As on all markets, the problem of collecting information about potential beneficial transactions is a real one for market agents. On the grid, in the absence of any centralized facility to keep track of market prices, we will have to pay attention to spreading information about supply and demand around in an efficient manner.

We will refer back to these two problems repeatedly when we try to solve them in chapter 4. First, though, let us have a look at how to do decentralized grid allocation in the comparatively simple situation where all nodes are assumed to be cooperative, and not strategic.

Chapter 3

Decentralized resource allocation

In this chapter, I will describe Oner's decentralized resource allocation algorithm, which was first described in Caglar Oner's masters thesis. [12] It provides an elegant solution to the problem of decentralized resource allocation in the absence of strategic behavior, and it will make for a good basis for further exploration of the same problem when strategic behavior gets introduced in chapter 4.

Oner's algorithm runs on a set of nodes called a grid. Each node can demand that a job be run continuously for some number t of time units on some number m of nodes simultaneously. Each node can also execute such jobs, one at a time.

The goal of the algorithm is to assign each job to a set of machines in such a way that it can start as soon as possible, but of course without conflicts, and without the need for any one node to have complete knowledge of the global schedule. In Oner's algorithm, there is no one node that functions as the scheduler—rather, each job is scheduled through a sequence of localized interactions between individual nodes.

3.1 Overview of the algorithm

We use a gossip algorithm to maintain an *overlay* among the nodes. [15, 11] The overlay is a graph "overlaid" on the graph of the underlying communication network. In the overlay graph, each node is responsible for keeping track of the set of nodes it is connected to, called the neighbor set. In the gossip algorithm, each node periodically initiates a transaction known as "gossip" with one neighbor in its neighbor set. In a gossip transaction, the two participating nodes swap parts of their neighbor sets. The transactions in the gossip algorithm serve to preserve the connectedness of the overlay graph. Without the periodical exchange of parts of the neighbor sets, the graph structure could easily become partitioned into several mutually unconnected subgraphs.

Now that we understand how the overlay is maintained, let us examine the algorithm used to allocate resources to a job. Each node maintains a *node schedule* that records what job it is to execute at each future time period.

When a node originates a job that needs to be run on m machines for t minutes, it acts as the scheduler, or *coordinator*, of its own job. The coordinator picks a ran-

dom initial team of m nodes from its own neighbor set. (If its own neighbor set is not big enough, the coordinator can initiate a recursive search for more nodes through its neighbor set.) When the initial team has been selected, the coordinator asks each team member for its own node schedule. With these node schedules in hand, the coordinator can now find the earliest possible time at which all the initial nodes can run the job. When that time has been found, a tentative *job schedule* can be constructed which consists of the node schedules of all the nodes involved in the new job, each updated to include the new job. The job schedule is then communicated to all nodes in the initial team, which update their respective node schedules to reflect it.

The algorithm as described so far is a fully functional scheduler. Each job that is submitted to it will get scheduled and—eventually—executed. It is simply allocated to a set of nodes that happen to be close to the coordinator in the overlay, and it will be executed at those nodes' earliest common convenience. This is not, however, a very good algorithm.

The most remarkable feature of Oner's algorithm, and the feature that makes it a good algorithm after all, is schedule gossip. This is a mechanism by which the initial schedule can be improved. To perform schedule gossip, each node sends its node schedule to a randomly selected neighbor. When such a node schedule arrives, the recipient will attempt to use it to improve the job schedules of all the jobs that it coordinates. For each such job, it will run through all team members, and check if the job can be expedited by firing that team member and hiring the node whose node schedule just arrived. If this is the case, the job schedule in the coordinator's agenda is updated, and all team members are notified of the changes.

Locking and simultaneous updates

Something has been left out of the description of the algorithm so far. If many jobs are initiated at around the same time, there is a chance that two coordinators will request the node schedules of a node, notice that it is free at a given time, and each schedule a job in the free time slot. To avoid this double booking, we will have to use two-phase commit.

In particular, what we will do is to split the process of entering a new job into the node schedules of team members in two phases. First, each node in the prospective team is asked to pencil the job into its schedule, and it reports back whether it was indeed able to do so. If any one node reports that it was unable to pencil in the job, presumably because of a scheduling conflict, then a second message goes out to each of the team members asking to erase the job from its node schedule.

When an update to an existing node schedule occurs, a similar procedure is followed. First, each node affected by the change, that is, all members of the old team as well as all members of the new team, is asked to pencil in the new schedule, but not to discard the old schedule yet. It reports back whether it succeeded. If every node

Table 3.1: Parameters for Oner's algorithm

σ	The number of (not necessarily distinct) neighbors each node has
κ	The number of those neighbors exchanged upon each gossip transaction
ϕ	Time between neighbor gossip episodes
γ	Time between schedule gossip episodes

succeeded, then a second message goes out to all nodes to ask them to discard the old job schedule from their node schedules. If any one node did not succeed, then a second message goes out to all nodes to ask them to discard the new job schedule and keep the old. The implementation does, of course, have to take account of the possibility of the old and the new schedule overlapping.

A note on the role of the coordinator

In Oner's original algorithm, the coordinator is not identified with the node that originates a job. In fact, the demand need not originate with one of the nodes at all, but could very well come from a client machine that does not itself participate in the grid. When a job demand arises, the role of coordinator is assigned to a random node within the grid.

This is a feasible strategy in the case where Oner's algorithm is used as a drop-in replacement for an existing, centralized scheduling algorithm, but it does not make sense in the case of a fully decentralized grid based on reciprocity rather than on blind execution of all submitted jobs.

3.2 Specification of the algorithm

The description of Oner's algorithm so far has been written to be comparatively easy to understand on first reading, but it omits many of the details involved in implementing the algorithm. In this section, we will go through the algorithm again, but this time in more detail. The casual reader will want to skim this section, but those who want to implement Oner's algorithm themselves are advised to pay close attention.

Oner's algorithm is parameterized by the parameters described in table 3.1. It runs on a set of nodes, each of which has an address a and maintains a state consisting of three parts:

1. A neighbor array N_a, which is an array of σ node addresses that records what the node's neighbors are. The initial neighbor arrays represent the initial *topology* of

Table 3.2: A job schedule for job j used by Oner's algorithm is a record consisting of 7 fields, the last of which is itself a complex data structure that in turn contains job schedules

Field	Meaning
i_j	Globally unique job ID
m_j	Size of required team
d_j	Time required
t_j	Time job will start
c_j	The job's coordinator node
T_j	The team, as a finite mapping from node addresses to node schedules (which are themselves mappings from times to job schedules.)

Table 3.3: Node state for the Oner's algorithm

N_a	Neighbor array (primary overlay)
A_a	Initially empty agenda
S_a	Initially empty node schedule

the overlay, and effectively forms another parameter to the algorithm.

2. An initially empty agenda A_a, which is a finite mapping from job IDs to job schedules that records what jobs the node coordinates.[1] Job schedules are records described in table 3.2 that record information about a job.

3. An initially empty node schedule S_a, which is a map from times to job schedules that records what jobs the node will run in at each time in the future.[2]

There are three ways in which activity can be initiated:

[1] A finite mapping can be implemented, for instance, as a hash table or a balanced tree. Some of the algorithms are easier to implement if we choose a functional finite map, that is, a finite map that can be non-destructively updated, such as a splay tree.

[2] In practice, because each job will run for a contiguous period, it is possible and useful to implement S_a more compactly, but the algorithm is easier to explain if we assume S_a simply has an entry for each unit time period, where units could for instance be seconds or minutes measured from the UNIX epoch.

Neighbor gossip Every ϕ time units, each node a will initiate a gossip transaction with another node $b = N_a[x]$, where x is selected at random from $\{1,\ldots,\sigma\}$. It sends a message GOSSIP($\langle k_1,\ldots,k_\kappa\rangle, \langle N_a[k_1],\ldots,N_a[k_\kappa]\rangle$) to b, where the k_i are drawn independently at random from $\{1,\ldots,\sigma\}$. Upon receipt of this GOSSIP message, b returns to a a message PISSOG($\langle k_1,\ldots,k_\kappa\rangle, \langle N_b[l_1],\ldots,N_b[l_\kappa]\rangle$), where the l_i are drawn independently at random from $\{1,\ldots,\sigma\}$, and only then it performs its updates $N_b[l_1] := N_a[k_1],\ldots,N_b[l_\kappa] := N_a[k_\kappa]$. Upon receipt of the PISSOG message, a performs its own updates $N_a[k_1] := N_b[l_1],\ldots,N_a[k_\kappa] := N_b[l_\kappa]$.[3]

New job At undetermined times, a node a may initiate a new job with a globally unique job ID i for which it requires the use of m nodes for d time units. First, it must assemble an initial team $T = n_1,\ldots,n_m$ by picking m distinct random neighbors from N_a, and send them each an ASK(i) message. Upon receipt of an ASK message, each node responds with an ANSWER(i, S_b), where S_b is b's node schedule.[4] Upon receipt of an ANSWER, a checks whether all the ANSWERs for a given job are in, it finds the earliest time t at which all nodes in T are available for d subsequent time units, and it constructs an initial job schedule j such that $i_j = i$, $m_j = m$, $d_j = d$, $c_j = a$, $t_j = t$, and $T_j = T$. It then sends a message ENTER($\bullet, j$) to each $b \in T$, where $\bullet$ is a dummy job schedule for which $t(\bullet) = 0$ and $d(\bullet) = 0$. [5] Note that the nodes must have synchronized clocks in order to actually execute a given job all at the same time, so that the various nodes participating in the job can communicate with each other. The details of actually executing the jobs are outside the scope of this thesis.

Schedule gossip Every γ time units, each node a will send a message SUGGEST(S_a) containing its node schedule to another node $b = N_a[x]$, where x is selected at random from $\{1,\ldots,\sigma\}$. Upon receipt of a SUGGEST message, b will examine each of the jobs schedules j in its agenda for which it is the coordinator, and for each job j it will consider each of the nodes n in its team T_j, and find out

[3] Note that the value of $\langle k_i,\ldots,k_\kappa\rangle$ does not need to be kept track of by a because it is passed back and forth in the messages. The specification is ambiguous as to what happens when there are duplicates among the k_i, l_i. In my implementation, the updates are made in order of ascending i. It may be tempting to require that the indexes be distinct, but not much is gained by this in practice because the elements of the neighbor arrays need not be distinct. Lastly, we observe that it is possible for a single node to be involved in multiple gossip transactions, and it is admittedly not specified in which order the updates are to be intertwined, but any order will do as long as each set of updates $N_b[l_1] := N_a[k_1],\ldots,N_b[l_\kappa] := N_a[k_\kappa]$ and each set of updates $N_a[k_1] := N_b[l_1],\ldots,N_a[k_\kappa] := N_b[l_\kappa]$ is performed atomically.

[4] The job ID is sent back and forth in this message once again for no other reason than to maintain state at the coordinator between the receipt of two messages.

[5] In Oner's initial description, it was possible to have jobs that require more than σ nodes. For the purpose of researching efficient distributed allocation, this is not relevant, so it will be left out for clarity.

whether $T'_j = T_j - \{n\} \cup \{a\}$ can execute the job j earlier than T_j can.[6] When an improvement is found, a new job schedule k is constructed with the new team and starting time, and a message ENTER(j,k) is sent to all members of $T_j \cup T'_j$.

Once the ENTER messages have been sent to the nodes $n_1,\ldots,n_z$, the two-phase commit process for a new job and the two-phase commit process for an improvement are identical, so we will describe the two together.

Upon receipt of an ENTER(j,k) message, a node n will attempt to enter the job k into its node schedule. First, it removes all entries $S_n[t_j],\ldots,S_n[t_j+d_k-1]$. Then, if k conflicts with an already scheduled job (other than j, which has just been removed),[7] it adds the old job j back in, $S_n[t_j] := j,\ldots,S_n[t_j+d_k-1] := j$ and returns a message ENTERED(F,j,k), where F means "false," because the attempt to enter the job was unsuccessful. If there is no such conflict, it returns a message ENTERED(T,j,k), it adds the old job j back in, $S_n[t_j] := j,\ldots,S_n[t_j+d_k-1] := j$, and then it also adds k, $S_n[t_k] := k,\ldots,S_n[t_k+d_k-1] := k$. If j and k overlap, this means that part of the entries for j will be overwritten, but the old job must go back in the schedule in order to keep its slots occupied until the two-phase commit is finished.

Upon receipt of the last of the messages ENTERED(β_1,j,k),…, ENTERED(β_z,j,k) for a given job, a will send a second round of messages COMMIT(β,j,k) to $n_1,\ldots,n_z$ where $\beta = \beta_1 \wedge \cdots \wedge \beta_z$.

Upon receipt of a commit request COMMIT(T,j,k), a node n removes $S_n[t_j],\ldots,S_n[t_j+d_j-1]$ and only then it sets $S_n[t_k] := k,\ldots,S_n[t_k+d_k-1] := k$ and returns an commit acknowledgment COMMITTED(T,j,k). When a node n receives an abort request COMMIT(F,j,k), on the other hand, it removes $S_n[t_k],\ldots,S_n[t_k+d_k-1]$ and then it sets $S_n[\cdot] = k$ from S_n and it sets $S_n[t_j] := j,\ldots,S_n[t_j+d-1] := j$, and it returns an abort acknowledgment COMMITTED(F,j,k). Also, for each job j in S_n, n sends a message UPDATE(i_j,S_n) to c_j.

Upon receipt of an UPDATE(i,S_n) message, a coordinator node a updates $T(A_a[i])[n] := S_n$ to reflect it.

Upon receipt of the last of the COMMITTED messages for a given job, a checks if all these messages have contained T. If so, it can update its agenda such that $A_a[i_k] := k$ and report that the job has been successfully scheduled, or improved, as the case may be. If any one of them reports F, it must report that the job has not been successfully scheduled (or improved, as the case may be.)

To summarize, all the types of messages that nodes may send are listed in table 3.4.

[6] This nested loop is computationally intensive, and profiling has shown that it is worth optimizing both this loop and the data structure used to store node schedules, which it uses intensively.

[7] This is where jobs are refused because of scheduling conflicts. Later on, we will use this to refuse jobs for other reasons!

Message name	Signature
GOSSIP	list<index> × list<address>
PISSOG	list<index> × list<address>
SUGGEST	map<time, job_schedule>
ASK	job_id
ANSWER	job_id × map<time, job_schedule>
UPDATE	job_id × map<time, job_schedule>
ENTER	job_schedule × job_schedule
ENTERED	bool × job_schedule × job_schedule
COMMIT	bool × job_schedule × job_schedule
COMMITTED	bool × job_schedule × job_schedule

Table 3.4: Messages used by Oner's algorithm with their signatures, where list<...> is a polymorphic vector, and map<...,...> is a polymorphic finite mapping.

Chapter 4

Efficient resource allocation

In this chapter, we will begin by making some very simple changes to Oner's scheduler in order to turn it into a resource allocator that can deal with the presence of strategic nodes. In the process, we will see that the algorithm's ability to come up with good schedules is reduced, and that the mechanism presented is still vulnerable to certain strategic manipulations. This will set the stage for a series of improvements that remedy these problems.

4.1 Resource ownership

In Oner's algorithm, nodes are never asked whether they want to participate in executing a job. They are simply drafted into the team by the coordinator. The first step we will take in ensuring reciprocity is to give each node *ownership* of its own resources, which means it must have the option of refusing to participate in executing a job. As it happens, the solution to the scheduling conflict problem discussed in the previous chapter already provides the facility needed for nodes to refuse jobs. Nodes can refuse jobs for strategic reasons in the same way that they might refuse a job on account of a scheduling conflict. Therefore, adding resource ownership to the algorithm does not actually require any modifications.

4.2 A reciprocity-enabled algorithm

Now that nodes can refuse jobs for reasons other than scheduling conflicts, the next question is when they should do so. In Oner's original algorithm, the answer was: "never." But Oner's nodes did not have to deal with strategic competitors. Ours do, so they must be cleverer.

We are now moving from the realm of rules into the realm of strategy (in the language of game theory), or from mechanism into policy (in the jargon of computer systems design). We could easily imagine a variety of nodes with a variety of strategies coexisting and evolving, as is the case in practice with the BitTorrent protocol. [13] Just as Cohen did for BitTorrent, though, we will also come up with a default refusal

policy, and the first one we try will in fact be analogous to Cohen's "unchoking" policy for BitTorrent. [7]

In particular, I propose that a node n should maintain two numbers about every node m it interacts with. The first I will call the IOU ("I owe you") account $I_n[m]$, the second, admittedly ungrammatically, the UOI ("you owe I") account $U_n[m]$. Whenever a worker w performs a job for a coordinator c, the worker w will add an amount corresponding to the duration of the job to the UOI account it keeps for c ($U_w[c]$) and c will add the same amount to the IOU account it keeps for w ($I_c[w]$). Whenever a node gets removed from a team, the credits need to be reversed.

With this basic bookkeeping in place, the nodes have some information to base the decision on of whether to accept a given job. A simple policy would be for w to accept a job request from c whenever $I_w[c] > U_w[c]$. That way, at all times w ensures that it has gotten more out of its relationship with c than it has put in.

Obviously, this policy is too conservative. The nature of exchange in the grid, in the absence of money payments, is that each individual transaction is a one-sided transfer from one node to another, offset by some transfer in the other direction in the past or the future. It is this delayed nature of grid exchange that makes it impossible to maintain efficiency on a transaction-by-transaction basis. There has to be a mechanism by which nodes may come to trust each other enough to become willing to render services in exchange for the expectation of reciprocation later on.

We can implement this idea by insisting not that all accounts clear all the time, but rather that there can be some credit as long as it is below a certain limit. That is, instead of requiring that $I_w[c] > U_w[c]$, we require only that $I_w[c] > U_w[c] - \mathscr{L}_w[c]$, where $\mathscr{L}_w[c]$ is the credit limit. Obviously, there is a lot of leeway we have in choosing an expression for $\mathscr{L}_w[c]$.

We must start out the credit limit $\mathscr{L}_w[c]$ that w extends to c at a certain level so that interaction can get started. We will call that level the "generosity" parameter G. Furthermore, in order to increase the potential gains from trade, we may want to increase the credit limit over time as trust builds up—but we should be careful that a high credit limit once built up does not last forever, lest a node that suddenly turns from cooperative to exploitative be able to continue its exploits forever. Therefore, we increase the credit limit in proportion to contributions c has made to w, but we let the influence of those contributions decay exponentially with time. In summary, we set

$$\mathscr{L}_w[c] = G + E \times \left(\sum_{k \in \mathscr{C}_w[c]} v_k (1-D)^{t-t_k} \right),$$

where G is the "generosity" parameter, E is the "escalation" parameter, D is the "depreciation" parameter, and the v_k and t_k are the values and ages respectively of past contributions $k \in \mathscr{C}_w[c]$ that c has made to w.

Table 4.1: Parameters for reciprocity algorithm

G	Generosity parameter
E	Escalation parameter
D	Depreciation parameter

Table 4.2: Node state for the reciprocity-enabled algorithm

N_a	Neighbor array (primary overlay)
A_a	Initially empty agenda
S_a	Initially empty node schedule
U_a	UOI accounts
I_a	IOU accounts
C_a	Credit scores

This approach, contrary to appearance, does not actually require keeping track of the set $\mathscr{C}_w[c]$ of all prior contributions by c to w. We need only keep track of a credit score $C_w[c]$ that is initialized to 0, multiplied by $1-D$ at each time period, and incremented whenever $I_w[c]$ is incremented. Once we have that, we observe that

$$\mathscr{L}_w[c] = G + E \times C_w[c].$$

This is precisely how bank savings accounts work: the bank does not compute compound interest separately for each past contribution to the account, but instead performs the entirely equivalent operation of paying interest over a running balance.

The generosity parameter allows for trust to be initially established, and the escalation parameter allows for trust to increase between long established partners. The depreciation parameter limits the process of escalation, so that when trust is broken, it does not linger forever.

We will see in chapter 5 how direct reciprocity performs. For now, we will just rely on intuition and say what are likely to be the obvious problems with the direct reciprocity scheme:

The initial trust problem Each node stands to lose $G \times n$ CPU-ticks of computational resources in entirely unreciprocated gifts to strangers, where n is the number of other nodes interacted with.

The trust escalation problem Each node stands to lose additional CPU ticks to partners that have reciprocated, somewhat, but will in the future fail to reciprocate as much as they ought to. The dual mechanisms of the escalation parameter and the depreciation parameter present a crude TIT-FOR-TAT like way of dealing with this: the amount of credit we are willing to extend to a long-term partner is proportional to how much they have done for us lately, in a specified sense of "lately": the weight of contributions decays exponentially with age.

The information problem By reducing the number of transactions that will actually happen as compared to Oner's original algorithm, the reciprocity-enabled algorithm reduces the ability of the resource allocator to make effective use of dispersed information about supply and demand in the network.

And a scalability problem... Scalability relates more to the first goal of this thesis, which is that the system should be decentralized. The reciprocity-enabled algorithm requires each node to keep track of a potentially very large set of accounts; in fact, the total amount of account information kept in the grid scales quadratically with the number of nodes in the grid. This means that a grid running the reciprocity-enabled algorithm is not a scalable distributed system.

4.3 A locality-enabled algorithm

Of the problems we have met, we will first address the initial trust problem, in the process solving the newly created scalability problem, but making the information problem even worse. The way in which we limit the potential problematic consequences of generosity is by not extending it to all nodes, but only to those in a local neighborhood. Although we already have a structure—the primary overlay—in which nodes have neighbors, it is not suitable for this purpose, because that overlay is designed on purpose to be very fluid and change all the time, which would defeat the entire purpose of trying to restrict the number of nodes that can receive unreciprocated gifts. What we need is a new, secondary overlay, independent of the primary, that is rather static. Any graph with a bounded average out-degree will do, and there are several peer-to-peer protocols that can be used to maintain such a graph.

In the next section we will encounter the very important requirement that the graph must have certain particular routing properties, and design our overlay graph with that requirement in mind. Therefore, let us delay the discussion of the particular kind of graph to be used to establish locality, and describe the pros and cons of locality:

The initial trust problem In a static graph with a maximum average out-degree of d, the maximum average loss per node to unreciprocated initial generosity is $G \times d$. Since G and d are both fixed, and typically small, constant parameters,

Table 4.3: Node state for the locality-enabled algorithm

N_a	Neighbor array (primary overlay)
A_a	Initially empty agenda
S_a	Initially empty node schedule
U_a	UOI accounts
I_a	IOU accounts
C_a	Credit scores
V_a	Neighbor set (secondary overlay)

this is a rather acceptable result. We are unlikely to ever do much better on the initial trust problem, except possibly if we can transform all time-separated interaction into instantaneous monetary transactions, so we may as well declare the initial trust problem solved for now.

The credit escalation problem The trust escalation problem is also reduced, by reducing the number of nodes that a given node interacts with. While it is still possible to escalate a credit limit and then have your trust betrayed, there are fewer opportunities to suffer such a loss. There is some anecdotal evidence, from Axelrod's work for instance, and from Bittorrent, that a gradual trust escalation scheme as described here works well in practice, but it is very hard to come up with hard and fast theoretical results about it. The reason for this is simply that the number of possible strategies that other nodes may be following to mislead us into falsely trusting them is unbounded, and it is hard to generalize across all of them.

The information problem By severely reducing the set of nodes with which exchange can be engaged in, the spread of dispersed information about supply and demand is severely hampered. The locality-enabled algorithm will find it impossible to schedule any job on more than one machine unless all members of the initial team happen to be immediate neighbors in the secondary overlay of the coordinator node.

4.4 A transitivity-enabled algorithm: GOSSIPTRON

A chain of credit is a sequence of nodes such that each would normally accept a job if it were submitted by the previous one. If such a chain of credit exists between a worker

node and a coordinator, the job should be able to go through, because intermediate node in the chain can simply enter it into its accounts as one job performed for it by the next node, and one job that it performs for the previous node.

When a worker node gets a request from a coordinator, and it cannot grant that request directly because the coordinator is not a neighbor or it has insufficient IOUs, the worker node will have to find such a chain. It may try to do so by sending out a message to some of its neighbors in the secondary overlay. If one of these neighbors happens to owe something to the coordinator, and the worker owes something to that neighbor, then a two-hop chain has been established, and the intermediary informs the worker of the existence of the chain. In order to activate the chain, a two-phase commit can be used, and the balances adjusted as though the intermediary were doing a job for the coordinator and the worker were doing a job for the intermediary.

We now want to extend this mechanism to provide for chains of credit with more than two hops. If we are to find such multi-hop chains of credit effectively, we cannot rely on simply flooding messages out from the worker in all directions until they reach the coordinator. We need some kind of *routing* mechanism that will efficiently find the shortest multi-hop chain from one node to another. Moreover, the routing mechanism needs to be such that if a particular hop between two nodes turns out to be unusable, it is possible to find a second-best alternative route, and a third-best if that one fails, and so on.

A well known routing mechanism that makes it particularly easy to find not only some short route from one node to another, but also to find second-best routes if the shortest route turns out to be unavailable, is spatial routing. With spatial routing, each node is assigned a point in a space in some deterministic way, and connections are made (mostly) between nodes that are spatially adjacent according to the metric of the space. That way, we can easily route a message to a target by sending it at each step toward a node that is closer—in terms of the space in which the nodes are now embedded—to the target, and hence is separated from the target by fewer hops. This type of routing is not only able to find routes, but it can also find alternative routes if a particular hop is blocked for lack of credit, simply by passing a message from the node that it got to before the blockage occurred not to the neighbor that is closest to the target, but to the neighbor that is second-closest. When all possible directions have been exhausted in this way, it is even possible to backtrack one hop and do the same thing: excluding the neighbor that turned out to be a dead end, find the second nearest neighbor to the target.

Spatial routing in more detail

The spatial routing protocol that we will use employs a Delaunay triangulation as its overlay (which we shall call the secondary overlay to distinguish it from the primary overlay that is used for gossip.) Each node n in the set of nodes $\mathcal{N} \subset \mathbb{N}$ is assigned a

Table 4.4: Node state for the locality-enabled algorithm

N_a	Neighbor array (primary overlay)
A_a	Initially empty agenda
S_a	Initially empty node schedule
U_a	UOI accounts
I_a	IOU accounts
C_a	Credit limits
V_a	Neighbor set (secondary VORONET overlay)

point $h(n)$ in the Euclidean plane $\mathbb{R}^2$ using a hash function $h : \mathcal{N} \to [0,1) \times [0,1)$. Given the resulting set of points $\{h(n) | n \in \mathcal{N}\}$, we divide the Euclidean plane into a Voronoi tessellation of cells $\mathcal{V}(n)$, one for each node $n \in \mathcal{N}$, such that the cell $\mathcal{V}(n)$ that belongs to node n consists of all points that are closer to n than to any other $n' \in \mathcal{N}$, that is,

$$\mathcal{V}(n) = \{x \in \mathbb{R}^2 \mid \forall n' \in \mathcal{N}.\, d(x, h(n)) \leq d(x, h(n'))\},$$

where $d : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ is the Euclidean distance $d((x,y),(x',y')) = \sqrt{(x'-x)^2 + (y'-y)^2}$. Using this Voronoi tessellation, we then define a Delaunay triangulation, which is an undirected graph with the nodes $n \in \mathcal{N}$ for its vertices, and which has an edge between two nodes whenever their Voronoi cells are adjacent, that is, the set of neighbors in the Delaunay triangulation of a node n is

$$D_n = \{n' \in \mathcal{N} \mid \mathcal{V}(n) \cap \mathcal{V}(n') \neq \emptyset\}.$$

To improve the routing properties of the Delaunay triangulation overlay, we will find it useful to add to it a few long range connections $\{n \leftrightarrow n' \mid n' = \ell(n) \vee n = \ell(n')\}$ defined by some hash function $\ell : \mathcal{N} \to \mathcal{N}$, and thus the neighbors of n in the entire overlay graph are

$$V_n = \{n' \in \mathcal{N} \mid \mathcal{V}(n) \cap \mathcal{V}(n') \neq \emptyset \vee\ n' = \ell(n) \vee\ n = \ell(n')\}.$$

As it turns out, a peer-to-peer algorithm called VORONET exists to create and maintain an overlay graph very much like this graph V, with the additional complication of choosing long-range neighbors in a more sophisticated way, and adding a third kind of neighbors on top of the Delaunay and long range neighbors. These complications are needed in VORONET to maintain some desirable routing properties in cases that are exceedingly unlikely when coordinates are generated by a good hash function, as they are in GOSSIPTRON. [5] In the simulations in chapter 5, we will take the existence of the distributed VORONET algorithm on faith and rather

just compute the Delaunay triangulations centrally using the popular *qhull* program whenever nodes enter or leave the network. [3] This program computes the Delaunay triangulation of a set of points $P \subset \mathbb{R}^2$ by first computing the convex hull of $\{(x,y,z) \in \mathbb{R}^3 | (x,y) \in P \wedge z = x^2 + y^2\}$ and projecting that down onto the xy plane.

GOSSIPTRON in more detail

Once we have the secondary overlay in place, we can assume that each node n has a set of secondary neighbors V_n and that it is able to provide a good estimate of which one of these provides the shortest route through the secondary overlay to a given target. Now we can start worrying about using the secondary overlay to establish chains of credit.

The part of Oner's algorithm that we are going to modify is, again, the first phase of the two-phase commit process, where a request reaches a worker node asking it either to enter a new job into the node schedule, or to change the scheduling for a job that already is in the node schedule. At this point, the worker node can reject the job, or it can accept the job, at least preliminarily, and "pencil it into" the node schedule.

This is the point in the algorithm where Oner's algorithm checks for scheduling conflicts, and it is the point where the reciprocity-enabled algorithm checks the status of the reciprocity accounts, and it is also the point in the algorithm where the transitivity-enabled algorithm will attempt to establish a chain of credit.

More precisely, whenever a node n receives an ENTER request to enter a job into its node schedule, it then sends itself a BROKER request. When it receives a BROKER request, whether from itself or elsewhere, it checks whether it is the coordinator for that job, and if so, it adjusts the accounts, enters the job, and report success. If it is not itself the coordinator, it will select from its secondary overlay neighbors V_n the one that is nearest to the coordinator, adjust the accounts, and send a BROKER request onward to that node, where the process repeats (but carefully avoiding any circular chains), until the coordinator is reached, or the length of the chain exceeds the maximum chain length parameter T. When adjusting the accounts is impossible, because the credit limit is reached (and here we use the exact same conditions as were used in the reciprocity-enabled algorithm), failure is reported to the previous node, which will then try again with the next-nearest neighbor. If all possibilities are exhausted in this way, without ever yielding a suitable chain of credit, the whole enterprise is finally abandoned and the job rejected, just as it would be rejected in the case of a scheduling conflict in Oner's algorithm.

For more detail on the precise way in which backtracking is performed, and the way in which chains of credits are recorded to be later torn down if a node is removed from a team because a job schedule is improved, the extremely interested reader should refer to the appendix, in which extensively commented source code for the simulation is available. Everybody else will want to skip on to the next chapter, because these

details as uninteresting as they are crucial for correct implementation.

Chapter 5

Evaluation

In order to compare various approaches to grid scheduling, we are going to look at a simulation of the protocol, the source code for which is included in its entirety in the appendix.

5.1 Simulation techniques

There are two common ways to simulate peer-to-peer protocols. As we will see later on, I have used a third, but let us take a look at the two common techniques first.

Realistic simulation

The most obvious way to simulate a peer-to-peer protocol is what I shall call the *realistic* method. When you simulate a peer-to-peer protocol using the realistic method, you closely follow the way it would run on a real network by representing nodes as concurrent processes that pass messages. You implement those processes using *fork()* and sockets, or similar primitives provided by a thread library or programming language run-time. A cleverly written realistic simulation can easily share code with an actual implementation of the same protocol.

Lockstep simulation

Another way to simulate peer-to-peer processes is the more efficient *lockstep* method. In this method, you divide the algorithm that is run by each node into steps. During each step, an arbitrary amount of computation may occur, and arbitrary messages can be sent, but only messages can be received that had already been sent at the beginning of the step. Any messages that are sent during the step are buffered to be made available to their recipients only at the beginning of the following step. Steps are synchronized globally, and each step lasts until all nodes are done with it.

Rewriting a peer-to-peer algorithm in this way makes simulation faster and more reliable. Because messages are the only way in which the concurrent processes can influence each other's state, and messages are delayed until the end of the step, it

does not matter at all whether the processes within a step are actually executed concurrently or sequentially. This makes it possible to dispense with all the overhead of threads and communication primitives provided by the operating system, and to implement everything in terms of loops, tables and queues.

One disadvantage of the lockstep method is that simulation is entirely deterministic, whereas the actual implementation is not. This overspecification makes it very hard to tell whether results obtained in lockstep simulation are a good prediction of what might happen in implementation. Another disadvantage of the lockstep method is that you need to rewrite the peer-to-peer algorithm in a very particular way that often increases the amount of internal bookkeeping that needs to be done.

Hybrid simulation

I have used neither the realistic nor the lockstep method of simulation. Rather, I have used a variant of the realistic method, inspired by the lockstep method. My nodes are represented by Glasgow Haskell *Control.Concurrent* threads running on a single machine, and they communicate using the Glasgow Haskell *Control.Concurrent.Chan* mechanism, which allows simple buffered message passing. So far, this looks like a typical realistic simulation.

The chief problem of the realistic simulation technique, though, is that it relies on computations and events happening at the same speed, or at least in the same order, as they would in an actual implementation. If no special care is taken to make that happen, then all job submissions and gossip events are initiated immediately when the simulation starts up, and the first round of the allocation algorithm will have barely enough time to finish before the last job gets submitted. If we try to remedy this by pacing the job submissions with a timer to correspond to a realistic schedule taken from an actual grid, such as DAS-2, it will take much longer than needed to do a simple simulation—after all, we are only interested in scheduling, and do not care much about the actual computations that the scheduled jobs may represent, so we should be able to model an entire grid on a simple PC with considerable speedup. A possible compromise is to try to use a wall clock timer to submit jobs and pace events, but apply a scale factor, such that for instance each second in the simulation corresponds to an hour on DAS-2. It is very hard to get such a scale factor just right. The results of such a simulation will be sensitive to what machine you use to run it and how heavily that machine is loaded at the time.

In order to avoid this problem, I have chosen to make a slight modification to Oner's algorithm for simulation purposes, not as radical as I would have to do for a typical lockstep simulation, but just enough to be able to simulate comfortably. It so happens that in Oner's algorithm, and the variants on it that we explore, when there is no activity in the system, activity can be generated only by the submission of a new job or by a node initiating a gossip transaction. In the absence of one of those two events,

if there is nothing going on, nothing can start.

This makes it possible to implement the simulation of Oner's algorithm in such a way that both gossip transactions and job submissions are only begun in response to the ticking of a global clock. Peer gossip happens once every ϕ ticks, schedule gossip once every γ ticks. New jobs are initiated upon the ticking of the clock according to a set workload that describes the behavior of the actual human users of the grid as a function $W : \text{time} \rightarrow \text{list}(\text{address} \times \mathbb{N} \times \mathbb{N})$ that generates upon each clock tick a list of tuples, each corresponding to a given node ordering a job for a given duration and team size.

After a clock tick, an avalanche of messages is generated. There is gossiping, there is scheduling, and there is two-phase-committing. At some point after the clock tick happens, this avalanche dies down, and there will not be any more messages before the next clock tick. I have modified the algorithm so that it keeps track of when this point of tranquility is reached and follows it immediately by another clock tick. This way the clock need tick no slower than necessary, but we know for sure that it will not tick so fast as to make it impossible for the system to react to the events triggered by one clock tick before the next one happens.

5.2 Evaluation

I stated in the introduction that the goal of this project is to arrive at a system that is decentralized and efficient.

The decentralized nature of the GOSSIPTRON algorithm is built right into the design, with the one caveat that the reciprocity-enabled algorithm, which requires an amount of storage space for the accounts that scales quadratically with the size of the grid, is not exactly scalable, as we have already seen in the previous chapter.

Efficiency, on the other hand, is a more elusive goal. The criterion of wealth maximization lends itself well to theorizing, but not to measurement, because it is impossible to tell by observation whether all wealth-increasing transactions have been exhausted. Preferences remain *per se* unobservable, so it becomes very hard to translate the abstract criterion of efficiency into some sort of a baseline maximally efficient outcome that our algorithm can be expected to approach. Moreover, it is impossible to test the algorithm in the face of every possible set of strategic behaviors on the part of malicious nodes.

In order to proceed from here, we will have to combine some analytical tests with some intuition about the problem. First, we must recognize that there are two major requirements for efficiency. One is for the algorithm to solve the information problem and find all potential interactions that would lead to aggregate gains of wealth. The second is for the algorithm to address the trust problem and cut off proposed interactions that would lead to aggregate loss of wealth. If we evaluate these two components

of efficiency separately, we can get a sense of how well the GOSSIPTRON algorithm is doing.

5.3 The information problem

The first claim we will want to evaluate is that the algorithm does no worse at solving the information problem than Oner's algorithm. It is definitely the case that Oner's algorithm does not always find the best possible schedule, and that in cases where it does not, GOSSIPTRON won't find it either, but that is not what this thesis is about. We will want to find out in this section whether GOSSIPTRON can do as well as Oner's algorithm in a fair comparison.

In order to get such a fair comparison, we focus on the case where the nodes make demands on the grid that, averaged over as short as possible a timescale, are equal in size. In that case, we cannot reasonably blame any failures of GOSSIPTRON to schedule jobs as fast as Oner's algorithm on a legitimate effort to prevent free riding, so the information problem is cleanly separated from the trust problem.

Before we do the comparison, let us explore in yet more detail than before why exactly the comparison is a fair one. We know that it is arithmetically impossible *on average, in the long run* for users to be using more resources than they contribute. From this, in combination with the fact that the grid does apparently get used, we must conclude that nodes, on average, get more value out of using grid resources than they give up by putting in the same amount. This may be the case, as we discussed in chapter 2, because their demand for resources fluctuates whereas their supply is steady—but for all we know, it might as well be because other people's CPU cycles are tastier than your own—the fact follows from *revealed preference*, not any particular assumption about "why" resources are valued the way they are. Furthermore, we will assume that nodes value jobs executed soon after submission higher than jobs executed much later.[1] This means that in the case where demands on the grid, averaged over as short as possible a timescale, are equal in size, there is strictly no gain in efficiency to be had from restricting the scheduling performed by Oner's original algorithm, and any additional delay is a loss.

In order to operationalize this criterion, we must choose a particular workload. We will use an artificial workload function W_{heavy} where in each of the first 5 time units, 20 jobs of 5 machine, 5 time each are submitted, as well as a load W_{lite} where 10 such jobs are submitted in each of the first 10 time units, and a load W_{wacky} which is like W_{lite} except that during the first 5 time units, jobs are originated by even-numbered nodes, and during the next 5, by odd-numbered nodes. The jobs produced by a workload

[1] This is a standard assumption in economics. Occasionally, an economist will try to use it to analyze a futures market in perishable goods and feign surprise at the fact that it does not apply, but usually it is a sound assumption, because if you prefer to consume the good later, you can always save it.

are submitted sequentially by all nodes (with the twist of the odd and even nodes in the "wacky" case), the grid has 100 nodes in it, and the simulation is run for 100 time units before being cut off, or 200 time units in the case of the GOSSIPTRON algorithm with $T = 10$ and $T = 99$. The primary overlay is initialized to a random graph so as to remove the need to run the gossip algorithm for a considerable time before the graph takes on desirable properties. The gossip frequencies are set to $\phi = \gamma = 1$, the primary overlay neighborhood size to $\sigma = 10$, and the primary overlay gossip size to $\kappa = 2$. We also choose to set the parameters for the reciprocity part of the algorithm to $G = 1, E = 0.6, D = 0.05$ for no particular reason other than that they seem intuitively reasonable and they seem to work.

Oner's original algorithm

In figure 5.1 we can see how Oner's original algorithm performs on the various loads. Not all jobs are scheduled for immediate runs because the load is fairly heavy compared to the size of the grid.

Reciprocity-enabled algorithm

In figure 5.1, we can see that introducing a direct reprocity requirement hampers the ability of the algorithm to find good schedules somewhat, especially with a heavier load, but not very much. This is partially due to the particularly smooth distribution of job origins in the workload we used, and in the case of the "wacky" workload, which is designed to put more strain on the generosity mechanism, we see that efficiency is reduced.

Locality-enabled algorithm

In figure 5.1, we can see that introducing locality into the reciprocity-enabled algorithm completely destroys the algorithm's ability to get anything done. Dealing only with secondary overlay neighbors just doesn't work if you need to get 5 worker teams together. It turns out that for smaller teams and smaller network sizes, it sometimes happens, as though by accident, that the initial team is composed of nodes that are all the workers' neighbors not only in the primary but also in the secondary overlay, but this is exceedingly rare. Obviously, the locality-enabled algorithm is no good except as a basis for the transitivity-enabled algorithm.

Transitivity-enabled algorithm

The transitivity-enabled algorithm (figure 5.2) for a low value of the transitivity limit T performs as poorly as the locality-enabled algorithm, but as T goes up, its perfor-

Figure 5.1: Results from Oner's algorithm, the reciprocity-enabled algorithm, and the locality-enabled algorithm. The graphs are histograms of scheduling delays, with delay on the x-axis and frequency on the y-axis, for a 100 node network run for 100 time steps with $\sigma = 10, \kappa = 2, \phi = \gamma = 1$.

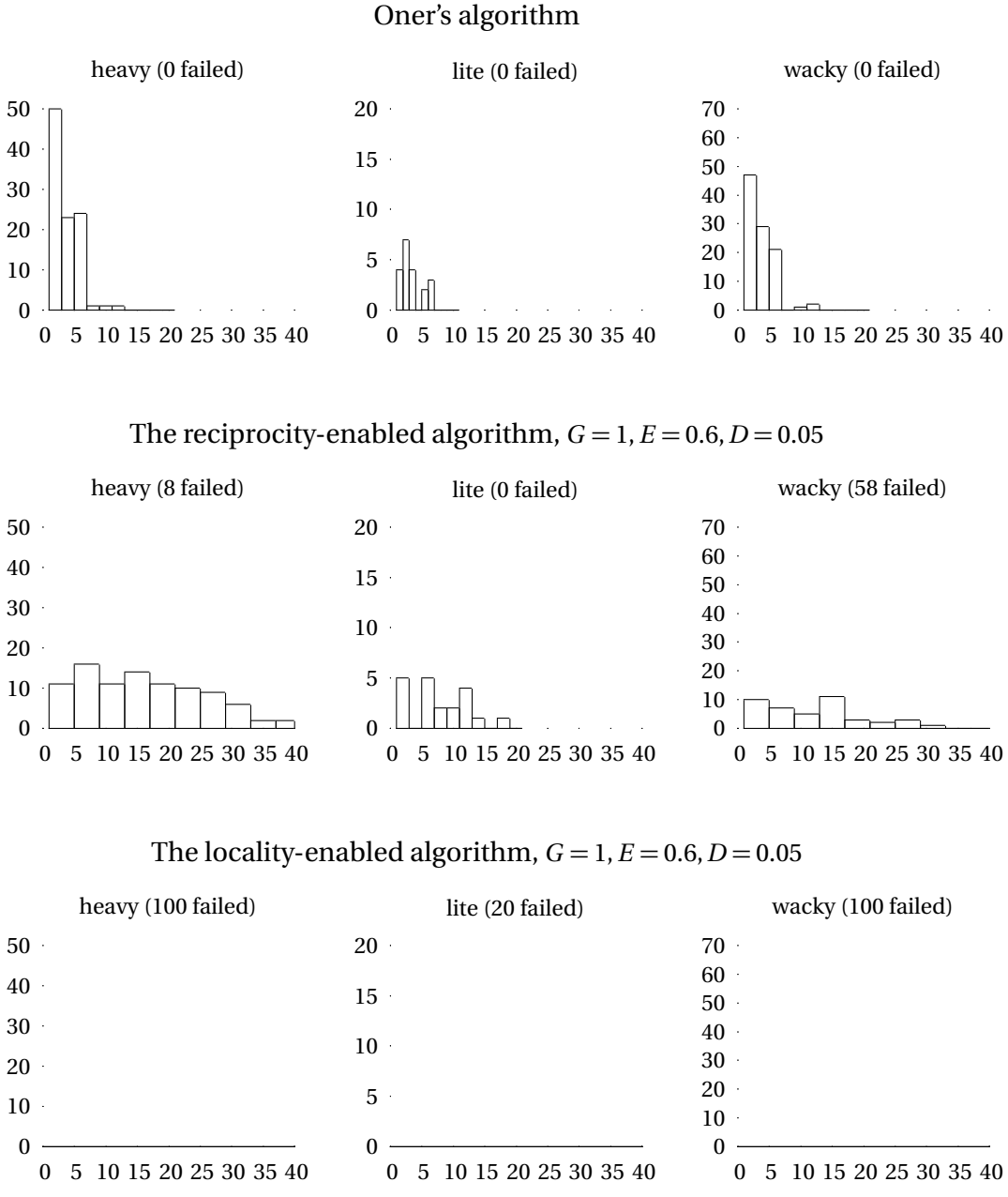
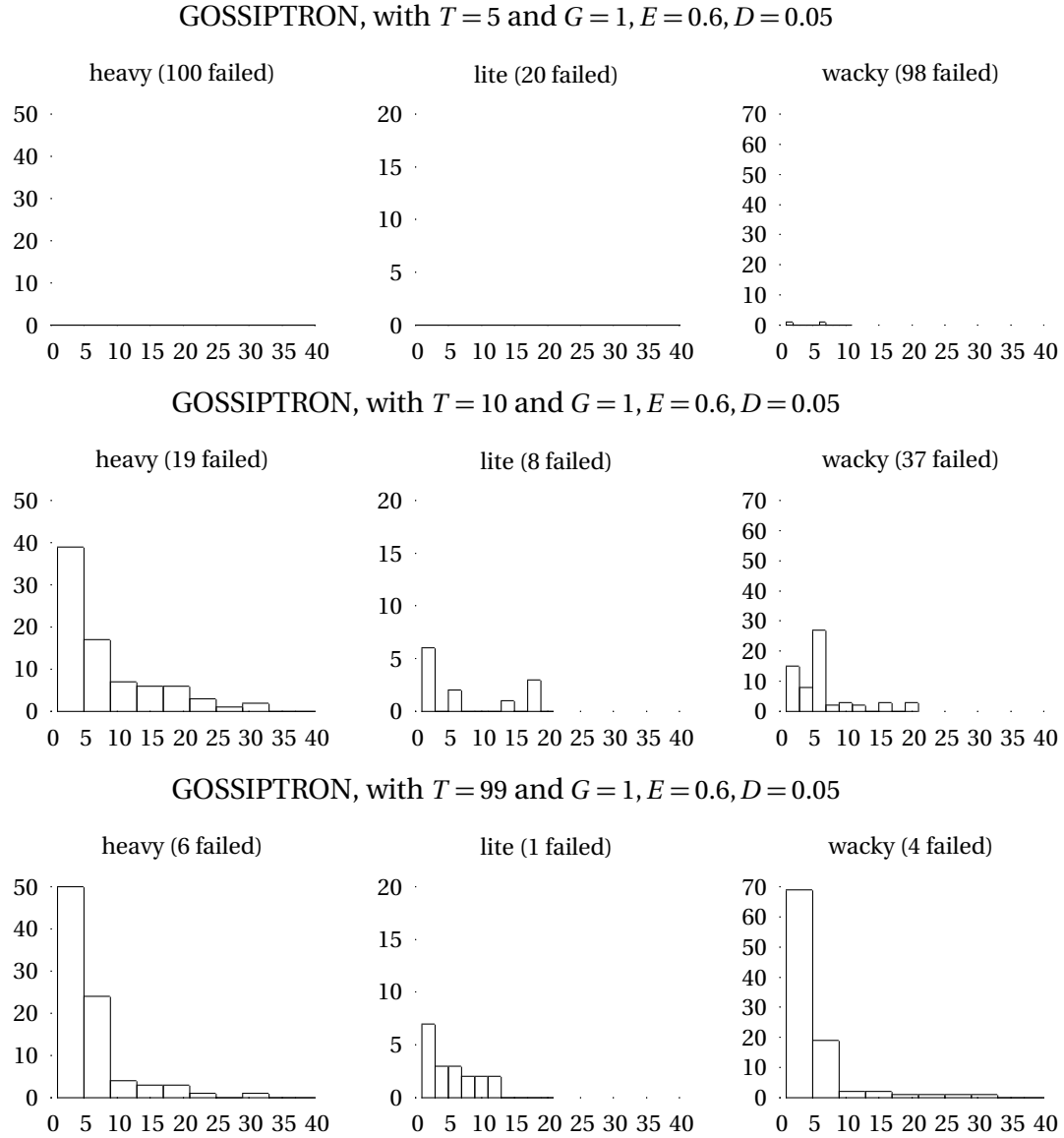


Figure 5.2: Results from GOSSIPTRON. The graphs are histograms of scheduling delays, with delay on the x-axis and frequency on the y-axis, for a 100 node network run for 100 time steps in $T = 5$ case and 200 time steps in the $T = 10$ and $T = 99$ cases, with $\sigma = 10, \kappa = 2, \phi = \gamma = 1$.



mance improves to approach that of the original algorithm. Most jobs are still scheduled within 0–5 time units, but the right-hand tail of jobs that take longer is longer and fatter than it was in the previous algorithms.

5.4 The trust problem

The trust problem does not lend itself to the same kind of direct empirical testing as the information problem, because it is not possible to test a given strategy against all possible other strategies. Instead, we need to inform any testing we do with some more analysis.

The only bounds we can give on the unbalance of any given node w for Oner's algorithm are

$$-\infty < \Sigma U_w - \Sigma I_w < \infty.$$

That means that the difference between the amount a node can get cheated for is unbounded, as is the amount to be gained in resources in excess of contributions by acting strategically.[2] These characteristics of Oner's algorithm clearly leave something to be desired.

For the reciprocity-enabled algorithm it is

$$|\mathcal{N}| \times \left(G + E \times \sum_{k \in \mathcal{U}_w} v_k (1-D)^{t-t_k} \right) < \Sigma U_w - \Sigma I_w < |\mathcal{N}| \times \left(G + E \times \sum_{k \in \mathcal{I}_w} v_k (1-D)^{t-t_k} \right)$$

where $\mathcal{I}_w$ is the set of contributions k made to w expressed as amounts v_k and times t_k (but we have always set $v = 1$ in the simulation), $\mathcal{U}_w$ is the set of contributions made *by* w, and $|\mathcal{N}|$ is the number of nodes in the grid. For both the locality-enabled algorithm and the ultimate transitivity-enabled algorithm GOSSIPTRON, the bound is

$$\delta_w \times \left(G + E \times \sum_{k \in \mathcal{U}_w} v_k (1-D)^{t-t_k} \right) < \Sigma U_w - \Sigma I_w < \delta_w \times \left(G + E \times \sum_{k \in \mathcal{I}_w} v_k (1-D)^{t-t_k} \right)$$

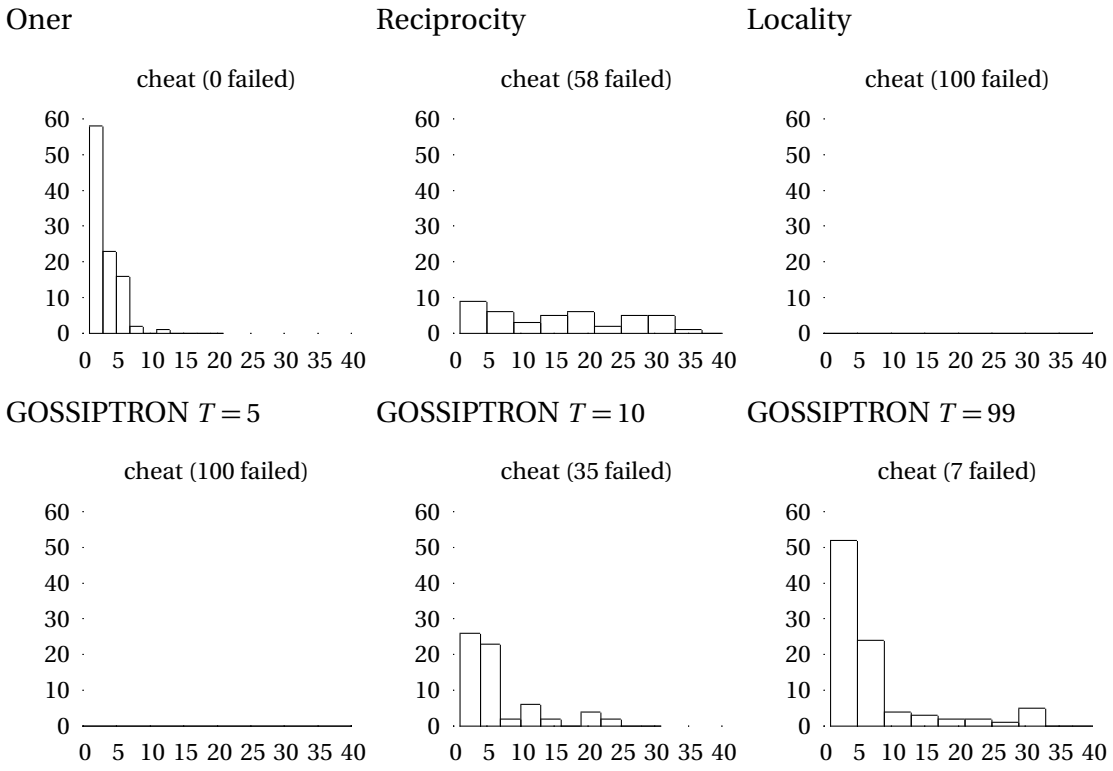
δ_w is the secondary overlay out-degree of w, which is at most 7 on average [6, corollary 9.5.3][3], so we can estimate the bound to be typically

$$7 \times \left(G + E \times \sum_{k \in \mathcal{U}_w} v_k (1-D)^{t-t_k} \right) < \Sigma U_w - \Sigma I_w < 7 \times \left(G + E \times \sum_{k \in \mathcal{I}_w} v_k (1-D)^{t-t_k} \right)$$

[2] It is of course possible to compute bounds on the amount of resources that can be gained within some amount of time based on the size of the grid, but that is beside the point here.

[3] The theorem referred to states that for a simple, finite planar graph, the average degree of nodes is at most 5. Our graph is such a graph with additional long-range links bringing that average up to 7.

Figure 5.3: Results from the various algorithms with 10% cheaters. The graphs are histograms of scheduling delays, with delay on the x-axis and frequency on the y-axis, for a 100 node network run for 100 time steps in $T = 5$ case and 200 time steps in the $T = 10$ and $T = 99$ cases, with $\sigma = 10, \kappa = 2, \phi = \gamma = 1$.



If we assume that $D = 0$, which is really the worst case in terms of risk, then

$$\lim_{I_w \to \infty} \Sigma \mathscr{L}_w / \Sigma I_w = 8E.$$

This means that, if $D = 0$, in the limit a node running GOSSIPTRON may still have to provide 8 times as many resources as it has gained from participating in the grid. We know from previous analysis that the reason nodes participate in the grid has to do with the fact that they value the resources they get higher than the resources they contribute, but how likely is it to be 4.2 times more? (That's what $7E$ is in the examples of the previous section, where $E = 0.6$.)

Actually, such situations could easily occur. There are many situations in which people who occasionally need lots of resources really need them only a small fraction of the time, and providing 4.2 nighttime CPU-hours for every daytime CPU-hour consumed may be a fair deal for many casual uses. Also, future research could focus on fine-tuning the parameters to see by how much E can be brought down, or whether the number of long-range links can be reduced, or any number of other expedients.

But it is not only the amount of damage that can be done to any given node that becomes bounded, but also the amount of gains any given node can make from doing damage. The hard bound on damages of $\lim_{I_w \to \infty} \Sigma \mathscr{L}_w / \Sigma I_w = 8E$ is not what GOSSIPTRON is built on. Trust is extended slowly toward those who have contributed, and reduced again slowly when they no longer contribute. Moreover, if we assume that node addresses are a hash of a node's actual IP address, so that it is not possible to execute Sybil attacks (an assumption that we can make by noting that identity fraud is outside of the scope of this thesis), the bound on how much there is to be *gained* from cheating becomes a disincentive for anybody who actually wants to use the grid.

Chapter 6

Conclusion

In the introduction, I wrote that the concept of "sharing" as it is often applied to grid computing covers up a lot of complexity. In this thesis, we have dug through some of that complexity. We have seen that through the careful application of known principles such as escalating reciprocal trust and spatial routing, the grid can become not just a place for friends to share resources in ways determined by a common understanding of goals, but also a place where the multitude of conflicting preferences of a group of not otherwise connected users can co-exist. Moreover, that co-existence is achieved without central management in a way that creates value and can draw users to the grid.

We moved from a grid that was a common pool that everyone could draw on without contributing to one where such abuse is prevented by the allocation mechanism itself. As we made that move, we also moved from the language of sharing, commonly applied to small homogeneous groups with commonly understood goals, to notions of property and exchange, commonly applied to larger groups of people with possibly conflicting goals. It is no different with computers than with people—in small groups, it may suffice to determine how to divide tasks to reach a certain goal, and rely on common understanding for coordination, but in large groups, abstract principles are needed. Without such abstract, "civic" principles, the blind application of small group principles to large groups leads inevitably to overloading of communication channels and the tragedy of the commons.

The abstract principles used to solve the public goods problem of the grid resource pool involve property rights, which are very well understood by economists, and transitive barter, which is usually overlooked because it is so often made unnecessary by the introduction of money. We have also looked into the issues of trust and delayed reciprocity, issues that are relevant to Bittorrent, and to consumer credit, and to friendship, but that we do not fully understand. We are perfectly capable, though, of taking the solutions that appear to work in the one-on-one case and extending them to a more general case by way of chains of one-on-one relationships, and this is what we have done.

There are other ways in which this type of research could have been approached. We should not write off the possibility of creating a decentralized monetary system of

sorts and then organizing an efficient grid on top of that as a set of straightforward monetary transactions. It is to be expected, though, that such an approach would raise much the same problems that we have encountered, albeit on the monetary system level rather than on the resource allocation level.

In order for my research to become practically significant on the grid, the algorithm needs to be tested under a larger variety of circumstances. It needs to be determined how tolerant it is to changes in any of its parameters, and how it holds up to particular attacks. Furthermore, it needs to be investigated how well the algorithm deals with churn in the grid and with failures of various sorts, and how well the transitivity algorithm scales as grids get bigger. To ensure scalability, a limitation on backtracking may well be needed in addition to the already established limitation T on chain length.

What we have established is not a ready made grid resource allocator to be deployed on a production grid tomorrow, but the understanding of broad economic principles in the specific context of grid computing that is required to build one. Unlike the previous literature, we have not passed off a centralized application of the textbook models used by economists to describe decentralized markets as a true decentralized resource allocation system, and neither have we relied on hidden centralized facilities such as grid banks. GOSSIPTRON is a truly decentralized grid allocator as much as Oner's original algorithm was a truly decentralized grid allocator, but unlike Oner's original algorithm, it can be shown to yield acceptable levels of economic efficiency under a variety of conditions where nodes are allowed to act strategically.

Appendix A

The Gossiptron code

The GOSSIPTRON algorithm and its constituent algorithms—the gossip algorithm and Oner's scheduler—are subtle distributed algorithms. It is easy to overlook details in implementing any of them. In order to make absolutely explicit what the algorithms are that I have used, I have written the simulation code as literate Haskell, so that it can be included entirely in this document. Where there is a discrepancy between the algorithms as specified in the text of this thesis and in the following Haskell code, either one (or both) could be wrong, but it is certain that the Haskell code is what was used for all measurements. I have attempted to accompany the code with a commentary written in such a way as to make this appendix useful even to readers who are not familiar with the Haskell programming language.

A.1 Module header

```
import Control.Arrow
import Control.Concurrent
import Control.Exception (assert)
import Control.Monad
import Data.Array
import Data.Array.IO
import Data.Bits
import Data.Char
import Data.Graph.Inductive.Graph hiding(out)
import Data.Graph.Inductive.Graphviz
import Data.Graph.Inductive.Tree
import Data.IORef
import Data.List
import Data.Maybe
import Data.Ord
import Data.Word
import System
import System.Directory
import System.IO
import System.Process
import System.Random
import Text.Printf
```

```
import qualified Data.IntMap as IntMap
import qualified Data.Map as Map
import qualified Data.Set as Set
import Compat
```

A.2 The simulation framework

The simulation framework operates on an object of type `Simulation m t`, which consists of a list of node addresses along with protocol objects that describe the behaviors of the nodes. Typically, these protocol objects will be identical or at least very similar, but this is not necessary. The only requirement is that all nodes communicate with each other using the same type `m` of messages, and that they all generate the same type `t` of log messages.

```
type Simulation m t  = [(Addr, Proto m t)]
type Addr            = Int
type Proto m t       = Net m t -> IO ()
```

Protocol objects are arbitrary actions in the IO monad. If you do not know Haskell, just remember that they represent algorithms that can do anything, including both computation and I/O. Each of these arbitrary actions is provided with a special `Net` object to play with, which serves as a handle that needs to be passed to the various primitives provided by the simulation framework library, which we will now discuss.

First of all, nodes can send messages of type `m` to each other. In order to do so, they must know the (integer) address of another node.

```
send :: Net m t -> Addr -> m -> IO ()
send net addr msg =
    do  node <- catch (IntMap.lookup addr (_us net)) $ \e ->
          printf "send: node %d not found" addr >> ioError e
        writeChan (_mbox node) (_addr (_me net), msg)
```

Messages can also be received, and they are delivered along with the address of the node that sent the message.

```
recv :: Net m t -> IO (Addr, m)
recv net =
    readChan (_mbox (_me net))
```

It is also possible to receive a message in a non-blocking way. If there is no message to be received, this procedure returns `Nothing`, and otherwise `Just (addr,msg)`, where `addr` is the address the message came from and `msg` is the message.

```
peek :: Net m t -> IO (Maybe (Addr, m))
peek net =
    do isEmpty <- isEmptyChan (_mbox (_me net))
       (if isEmpty then
            return Nothing
        else
            do msg <- readChan (_mbox (_me net))
               return (Just msg))
```

Aside from comunicating with each other, nodes can also generate output to be displayed on the user's terminal. Every time they do this, they must also supply a string called the `tag`. (This may sound annoying, but it is a very convenient feature, because it allows us later to filter the sometimes voluminous output of a simulation by tag!)

```
out :: Show u => Net m t -> String -> u -> IO ()
out net tag msg = writeChan (_out net) (_addr (_me net),tag, show msg)
```

Outputting data to the terminal is convenient for informal debugging, but in order to collect statistics from the simulation process, you should use the `report` procedure. Whereas `out` accepts anything that is fit to print (in Haskell-speak, anything that implements the `Show` typeclass), the `report` procedure takes only objects of type `t`. Remember that `t` can be anything—it just needs to be the same all the time and for all nodes. After the last call to `report` by any of the nodes, `eot` should be called once.

```
report :: Net m t -> t -> IO ()
report net msg = writeChan (_report net) (Just msg)

eot :: Net m t -> IO ()
eot net = writeChan (_report net) Nothing
```

Once we have assembled an object of type `Simulation m t` that describes the simulation we want to run, it is time to feed it to the `run` procedure. The simulation will then be run until either it receives a character of input from `stdin` (the terminal), or one of the nodes calls `off`. While the simulation is running, any invokations of `out` will result in terminal output, but `run` takes an argument `supress` which must be a list of tags, and when `out` is called with a tag included in that list, no output will result.

When the simulation is over, the `run` procedure returns a list of all log messages of type `t` that have been output by any of the nodes during the simulation. It does not keep track of which node sent which log message, and log messages do not necessarily come out in order!

```
run :: Show t => Bool -> [String] -> Simulation m t -> IO [t]
run interactive suppress nodes =
    do (switch, report,children) <- sim suppress nodes
```

```
       -- The interruptor waits for the user to interrupt
       interruptor <- forkIO $
                      if interactive then
                        hGetChar stdin >> flick switch
                      else
                        return ()
       -- The user can flick the switch, and so can the process...
       wait switch
       -- But either way, when the switch is thrown, kill them all!
       forM_ (interruptor:children) killThread
       -- Return log messages for bookkeeping
       reportOut <- getChanContents report
       return $ map fromJust (takeWhile isJust reportOut)

off Net {_off=switch}   = flick switch
isOff Net {_off=switch} = isEmptyMVar switch >>= return . not
flick switch            = tryPutMVar switch () >> return ()
wait switch             = readMVar switch >> return ()

sim :: [String] -> Simulation m t -> IO (MVar (), Chan (Maybe t),[ThreadId])
sim suppress nodes =
    do bus     <- foldM addToBus IntMap.empty nodes
       outC    <- newChan
       reportC <- newChan
       switch  <- newEmptyMVar
       cs      <- forM (IntMap.assocs bus) $ \(_,node) ->
                  forkIO $ (_proto node) (Net bus node outC reportC switch)
       c       <- forkIO $ forM_ [1..] $ \_ ->
                  do o@(_,tag,_) <- readChan outC
                     when (not (tag `elem` suppress)) (displayOutput o)
       return (switch, reportC, c:cs)
    where
      addToBus bus (addr, proto) =
          do mbox <- newChan
             return $ IntMap.insert addr (NetNode proto addr mbox) bus

      displayOutput (sender, tag, msg) =
          printf "[%4d] %8s %s" sender (map toUpper tag) (fmt 17 msg)
      fmt indent s   = drop indent (unlines (fmt' indent s))
      fmt' indent "" = []
      fmt' indent s  = let (x,xs) = splitAt (78-indent) s in
                       [replicate indent ' ' ++ x] ++ (fmt' indent xs)

data Net m t =
    Net { _us :: IntMap.IntMap (NetNode m t),     -- All nodes, by address
          _me :: NetNode m t,              -- The present node
          _out :: Chan (Addr,String,String),  -- An output channel
          _report :: Chan (Maybe t),    -- A report channel
          _off :: MVar ()               -- An "off switch"
        }
```

```
data NetNode m t =
    NetNode { _proto :: Proto m t,    -- This node's protocol
              _addr :: Addr,          -- This node's address
              _mbox :: Chan (Addr, m) -- This node's communication channel
            }
```

A.3 Basic data structures

In the GOSSIPTRON algorithm, nodes communicate by sending objects of type `Msg` to each other. The meaning of all these messages will become clear as we discuss how they are used.

```
data Msg = Tick Time
         | Tock
         | Wake
         | Sleep
         | HoldOn
         | Gossip [(Int,Addr)]
         | Pissog [(Int,Addr)]
         | Apply NodeSch
         | Run Work
         | Ask JobId
         | Answer JobId NodeSch
         | Update JobId NodeSch
         | Enter Job Job
         | Broker Job Job [Addr] (Maybe Addr)
         | Brokered Bool Job Job [Addr] [Addr]
         | Entered Bool Job Job [Addr]
         | Commit Bool Job Job [Addr]
         | Committed Bool Job Job [Addr]
         | Cancel Job Job [Addr]
         | Unbroker Job Job [Addr]
         | Canceled Job Job
         | Ran Bool Job Job
         | Finish
         | Finished
           deriving(Show)
type JobId  = Int
type Time   = Int
data Work   = Work { _wStart:: Time,  -- Time to start
                     _wId :: JobId,   -- Job ID
                     _wLength :: Int, -- Duration of job
                     _wSize :: Int,   -- Number of nodes in team
                     _wNode :: Addr   -- Address of team boss
                   } deriving Show
```

Besides sending messages to each other, the nodes can also report diagnostics to a special logging facility as messages of type `Report`.

```
data Report = ReportDelay Int
            | ReportFail Int
            | ReportOvl1 Addr [Addr]
            | ReportOvl2 Addr [Addr]
            | ReportConf Conf
              deriving(Show)
```

The GOSSIPTRON algorithm is parameterized by a number of parameters which are collected into an object of type `Conf`. The meaning of these parameters has been discussed in the main text of the thesis, and will become clear as they are used.

```
data Conf = Conf { _n :: Int,     -- number of nodes
                   _u :: Int,     -- duration of run in ticks
                   _s :: Int,     -- nr of neighbors
                   _c :: Int,     -- nr of neighbors exchanged in gossip
                   _f :: Int,     -- neighbor gossip frequency (0=never)
                   _g :: Int,     -- schedule gossip frequency  (0=never)
                   _G :: Double,  -- generosity (<0: naive)
                   _E :: Double,  -- escalation
                   _D :: Double,  -- depreciation
                   _T :: Int,     -- maximum transitivity
                   _cheaters :: [Addr],  -- cheaters
                   _ovl1     :: Ovl1,    -- initial primary overlay
                   _ovl2     :: Ovl2,    -- initial secondary overlay
                   _workload :: Workload -- workload function
                 }
data Ovl1     = Ovl1     String (Conf -> IO (Array Addr [Addr]))
data Ovl2     = Ovl2     String (Conf -> IO (Array Addr [Addr]))
data Workload = Workload String (Conf -> Time -> [Work])
```

The followng piece of code overrides the standard Haskell printer for the purpose of converting objects of type `Conf` to human-readable strings. It is needed because the standard Haskell printer loops on recursive data structures.

```
instance Show Conf where
  show Conf {_n=n,_u=u,_s=s,_c=c,_f=f,_g=g,
             _E=ee,_G=gg,_D=dd,_T=tt,
            _ovl1=ovl1, _ovl2=ovl2, _workload=workload }=
    "n="++show n ++" "++ "u="++show u ++" "++ "s="++show s ++" "++
    "c="++show c ++" "++ "f="++show f ++" "++ "g="++show g ++" "++
    "E="++show ee++" "++ "G="++show gg++" "++ "D="++show dd ++" "++
    "T="++show tt++" "++
    show ovl1++" "++show ovl2++" "++show workload
instance Show Ovl1     where show (Ovl1 s _)     = s
instance Show Ovl2     where show (Ovl2 s _)     = s
instance Show Workload where show (Workload s _) = s
```

A.4 Specifying the simulation

In the following description of the GOSSIPTRON protocol itself, I will be making use of a number of data structures and procedures that have not yet been defined, but if you've read this far, you now know enough to be able to appreciate the structure of the algorithm and fill in those details later.

The protocol is described by a procedure `gossiptron`, which accepts a configuration object of type `Conf` and returns (in the `IO` monad, but if you don't know haskell, don't worry about that) an object of type `Simulation`, ready to be interpreted by the simulaton framework described in `module Simulator`. The returned object will contain one entry for each of the worker nodes (the actual nodes in the grid), as well as one entry each for the clock node and the user node, both of which are contraptions required for purposes of the simulation, but not part of the specification of the algorithm proper.

```
gossiptron :: Conf -> IO (Simulation Msg Report)
gossiptron conf@Conf{ _ovl1 = Ovl1 _ ovl1',
                      _ovl2 = Ovl2 _ ovl2' } =
  do ovl1 <- ovl1' conf
     ovl2 <- ovl2' conf
     return $ [(worker, workerP conf (ovl1!worker) (ovl2!worker) worker)
                 | worker <- workers conf ] ++
              [(clock, clockP conf)] ++
              [(user, userP conf)]
```

A.5 The worker nodes

Each worker node runs nearly the same protocol, but it is parameterized by the node's initial neighbors in the primary and secondary overlays and the node's own address. To that end, the `workerP` procedure takes three additional parameters, which are supplied by `gossiptron`.

When a worker node is started, it proceeds to initialize its state, and then it enters an infinite loop in which it receives messages and acts upon them.

```
workers conf = [0 .. _n conf-1]
workerP conf ovl1 ovl2 addr net = do
  report net $ ReportOvl1 addr ovl1
  report net $ ReportOvl2 addr ovl2
  ledger    <- newLedger
  neighbors <- newNeighborhood ovl1
  myNodeSch <- newNodeSch addr
  agenda    <- newAgenda
  brokerage <- newBrokerage
  time      <- newIORef (-1)
  forM_ [0..] $ \_ -> do
    now           <- readIORef time
```

```
(sender, msg) <- recv net
case msg of
```

A `Tick` message updates the internal clock. That's all it does.

```
Tick t ->
  do writeIORef time t
     depreciate conf ledger
     send net sender $ Tock
```

A `Wake` message instructs the worker node to initiate all pending gossip operations for the current time step. `Gossip` and `Pissog` messages, which are exchanged every f units of time, are used to exchange parts of neighbor lists to maintain the primary overlay.

```
Wake ->
  do nodeGarbage myNodeSch now
     currentJobs <- agendaNow agenda now
     forM_ currentJobs $ \job ->
       do report net $ ReportDelay (now - _submit job)
     when (_f conf /= 0 && now `mod` _f conf == 0) $
       do [(_,peer)] <- pickNeighbors 1 neighbors
          i_k <- pickNeighbors (_c conf) neighbors
          send net clock $ HoldOn              -- >A
          send net peer  $ Gossip i_k
     when (_g conf /= 0 && now `mod` _g conf == 0) $
       do ns <- getNodeSch myNodeSch
          [(_,boss)] <- pickNeighbors 1 neighbors
          send net clock $ HoldOn              -- >B
          send net boss  $ Apply ns
     send net clock Sleep
Gossip i_k ->
  do j_l <- pickNeighbors (_c conf) neighbors
     let ((i,k),(j,l)) = (unzip i_k, unzip j_l)
     let (i_l, j_k)    = (zip i l, zip j k)
     send net sender $ Pissog i_l
     forM_ j_k $ setNeighbor neighbors
Pissog i_l ->
  do forM_ i_l $ setNeighbor neighbors
     send net clock $ Sleep                    -- <A
```

`Apply` messages, which are sent out by each node every g units of time, are what drives the improvement phase in Oner's algorithm. They are "job applications" that come with the node schedule of the sender. When one is received, the receiver proceeds to find out if the sender might be of use as a worker in any of the teams for the jobs that it coordinates. When a worker is hired, it receives a `Ran` message in return.

```
Apply ns ->
  do ps  <- agendaImprove agenda now sender ns
     forM_ ps $ \(old,new) ->
       do send net clock $ HoldOn              -- >C
          let ns   = Map.keys (Map.union (_team new)(_team old))
          let new' = new { _user = addr }
          clear <- agendaJobIni agenda (_id old) (length ns)
          case clear of
            0 ->
              forM_ ns $ \n -> send net n $ Enter old new'
            _ ->
              send net addr $ Ran False old new'
     send net clock $ Sleep                    -- <B
Ran r old new ->
  do send net clock $ Sleep                    -- <C
     dumpAcc net r now old new
```

Run messages are requests for entirely new jobs to be scheduled. They are sent by the user node, which models the actions of the users of the grid system. When a Run message is received, a random initial team is assembled, and the job must be scheduled at the earliest common convenience of the team. The first step in that process is to ask all team members for their schedules. Further action is postponed until all the schedules are in. The sender will receive a Ran message telling it whether the request succeeded, either immediately if things fail early on, or otherwise later.

```
Run w@(Work { _wId=i, _wSize=m, _wLength=d }) ->
  do job  <- agendaNewJob agenda i m d sender addr now
     team <- pickDistinctNeighbors m neighbors
     ini  <- agendaJobIni agenda i m
     forM_ team $ \p -> send net p $ Ask i
     when (null team) $ send net sender $ Ran False job job
```

An Ask message is a request for a worker's current node schedule. The worker responds with its node schedule in an Answer message, which is duly recorded in the boss's agenda. When enough Answer messages have been collected, an attempt is made to schedule the job based on the collected node schedules, and all workers are notified of the new job schedule using Enter messages.

```
Ask jobId ->
  do nodeSch <- getNodeSch myNodeSch
     send net sender $ Answer jobId nodeSch
Answer jobId nodeSch ->
  do agendaJobAddNode agenda jobId sender nodeSch
     left <- agendaJobDown agenda jobId True
     when (left==0) $
       do rdy <- agendaJobLaunch agenda jobId (now + 1)
          job <- agendaJob       agenda jobId
```

```
            ini <- agendaJobIni    agenda jobId (_size job)
            let _ = assert (ini==0) ()
            let team = Map.keys (_team job)
            forM_ team $ \node ->
              send net node $ Enter (nulJob job) job
```

At this point, it is still possible for the job to be rejected, either because the node schedules that the boss used to schedule it were out of date, or because the boss has insufficient credit with the worker.

```
Enter old new  ->
  do nodeRelease myNodeSch old
     isFree <- nodeFree myNodeSch new addr
     nodeCommit myNodeSch old                  -- B
     (if isFree && not (addr `elem` _cheaters conf) then
        do nodeCommit myNodeSch new            -- B
           uoi ledger addr                     -- B -a
           send net addr $ Broker old new [] Nothing
      else
        do send net sender $ Entered False old new [])
```

A `Broker` message is a request to be rewarded for help offered. There are three possible cases. The first is that the current node is the boss, who requested the help in the first place. The boss will chalk up an IOU for the helper and declare the transaction successful. The second case is that the current node is not the boss. In that case, it must find out which of its neighbors lives closest to the boss and whether that neighbor is worthy of credit. It must also make sure not to exceed the limit on transitivity (parameter T). If the right conditions are met, it chalks up a "you owe me" to the next neighbor on the path and sends it another `Broker` request. If the conditions are not met, it has to report failure to the sender of the `Broker` message.

```
Broker old new path detour ->
  do let routes = route ovl2 (_boss new) \\ path
     liquidRoutes <- filterM (liquid conf ledger) routes
     let nextRoute =
           case (liquidRoutes, detour) of
             ([], _)        -> Nothing
             (n:_, Nothing) -> Just n
             (_, Just d)    -> case dropWhile (/= d) liquidRoutes of
                                 _:n:_ -> Just n
                                 _     -> Nothing
     case (addr == _boss new, nextRoute) of
       (True, _) ->
         do iou ledger sender                  -- +a / +b
            send net sender $ Brokered True old new path path
       (False, Just next) | length path < _T conf ->
         do iou ledger sender                  -- A +a / +b
```

```
            uoi ledger next                    -- A -b
            send net next $ Broker old new (addr:path) Nothing
        (False, _) ->
          do send net sender $ Brokered False old new path []
```

When `Broker` requests fail, there is still a possibility to backtrack and try it by some other route.

```
Brokered False old new (x:xs) path ->
  -- Too clever by half! After successor fails, cancel account
  -- modifications and emulate a message that will lead the
  -- existing above code to try it again with the next available
  -- successor. Get it?
  do uniou ledger sender                       -- undo A ^+a
     unuoi ledger sender                       -- undo A ^-b
     send net addr $ Broker old new xs (Just sender)
Brokered False old new [] path ->
  do nodeRelease myNodeSch new                 -- undo B
     nodeCommit myNodeSch old                  -- undo B
     unuoi ledger sender                       -- undo B ^-a
     send net (_boss new) $ Entered False old new path
Brokered True old new (x:xs) path ->
  do send net x $ Brokered True old new xs path
Brokered True old new [] path ->
  do send net (_boss new) $ Entered True old new path
```

At some point, the brokering is done, and the node that sent the `Enter` message, i.e. requested that a job be entered into a node's schedule, must be notified of whether this worked out. When all these replies have been gathered, it can then in turn decide whether they were all positive. If they were, it will proceed to notify the workers to make the reservation permanent. If they were not, it will tell them to roll back the transaction.

```
Entered ok old new@Job{ _id=j } path ->
  do left <- agendaJobDown agenda j ok
     when ok $ brokerageAdd brokerage j sender path
     when (left==0) $
       do let ns = Map.keys (Map.union (_team new) (_team old))
          acc <- agendaJobAccepted agenda j
          ini <- agendaJobIni      agenda j (length ns)
          x   <- agendaJobNak      agenda j
          let _ = assert (ini==0) ()
          let _ = assert (x==length ns) ()
          forM_ ns $ \n -> send net n $ Commit acc old new path
Commit True old new@Job{ _id=j } path ->
  do nodeRelease myNodeSch old
     nodeCommit myNodeSch new
     jobs    <- nodeAllJobs myNodeSch
```

```
      nodeSch <- getNodeSch myNodeSch
      forM_ jobs $ \job@Job { _id=i, _boss=boss } ->
        do send net clock $ HoldOn              -- >D
           send net boss $ Update i nodeSch
      send net sender $ Committed True old new path
Commit False old new path ->
  do nodeRelease myNodeSch new
     nodeCommit myNodeSch old
     send net sender $ Committed False old new path
Update jobId nodeSch ->
  do agendaJobUpdateNode agenda jobId sender nodeSch
     send net clock $ Sleep                     -- <D
Committed ok old new@Job{ _id=j,_user=c, _team=t } path ->
  do left  <- agendaJobDown agenda j ok
     acc   <- agendaJobAccepted agenda j
     case (left,acc) of
       (0, True) ->
         do agendaJobRelease agenda old
            agendaJobCommit agenda new
            send net c $ Ran True old new
       (0, False) ->
         do agendaJobRelease agenda new
            agendaJobCommit agenda old
            brokered <- brokerageGet brokerage j
            (if (Map.null brokered) then
               send net (_boss new) $ Canceled old new
             else
               forM_ (Map.assocs brokered) $ \(node,path) ->
                 send net node $ Cancel old new path)
       _ ->
           return ()
Cancel old new path ->
  do unuoi ledger addr                          -- ^-a
     send net addr $ Unbroker old new path
Unbroker old new going ->
  do case going of
       y:ys ->
         do uniou ledger sender                 -- ^+b / ^+a
            unuoi ledger y                      -- ^-b
            send net y $ Unbroker old new ys
       [] ->
         do uniou ledger sender                 -- ^+b
            send net (_boss new) $ Canceled old new
Canceled old new@Job {_id=j,_user=c}->
  do brokerageRemove brokerage j sender
     brokered <- brokerageGet brokerage j
     when (Map.null brokered) $
       do send net c $ Ran False old new
Finish ->
  do send net sender $ Finished
```

A.6 The clock pseudo-node

Peer-to-peer systems are typically simulated in either one of two ways. The first is a "realistic" setup with communicating threads, where delays are simulated as such. The second is a "lockstep" setup where the algorithm is rewritten as a series of actions and delays, and communication is only allowed once per action. Because actions and delays are so neatly separated, there is no need to actually delay anything when a delay happens; the lockstep simulation can just deterministically march onward to the next action.

My simulation uses a hybrid model. In this hybrid model, there are episodes of "realistic" simulation, in which arbitrary amounts of communication are allowed and the precise order of messages depends on details of the underlying message passing and scheduling implementation. Each such episode, however, is initiated by a special clock node, which will initiate the next episode precisely when it is sure that no more communication will result from the previous; hence the episodes can never overlap.

To accomplish this, there is a system of `Sleep`, `HoldOn` and `Wake` messages that are exchanged between nodes and a single, special "clock" node. The clock node sends a `Wake` message to each node, and then proceeds to wait for `HoldOn` and `Sleep` messages until the number of `Sleep` messages equals the total number of Wake messages sent plus the total number of `HoldOn` messages received. At that point, the clock "knows" that there will be no more messages until it initiates a new cycle, so it proceeds to lose no time and initiate a new cycle immediately.

```
clock = (-1)
clockP conf net = do
  out net "conf" $ conf
  report net $ ReportConf conf
  forM_ [0 .. _u conf] $ \t ->
    do
       interrupt <- isOff net
       when interrupt $ fail "interrupt"
       out net "t" $ t
       let agents = user:(workers conf)
       forM_ agents $ \i -> do send net i (Tick t) >> recv net
       forM_ agents $ \i -> send net i $ Wake
       sem <- newBucket (length agents)
       let loop = do (sender,msg) <- recv net
                     case msg of
                       Sleep  -> bucketOut sem
                       HoldOn -> bucketIn  sem
                     done <- bucketEmpty sem
                     when (not (done)) $ loop
       loop
  -- One last message to know that the user has finished
  (_, Sleep) <- recv net
```

```
  forM_ (workers conf) $ \w -> send net w Finish >> recv net
  eot net
  off net
```

A.7 The user pseudo-node

In addition to the clock node, there is another pseudo-node used for simulation purposes, called the "user" node. The user node instructs various peers in the peer-to-peer system to initiate new jobs. In this sense, the user node algorithm models the unpredictable actions of the actual human users of the grid who submit jobs when they need them done. It keeps track of the jobs that failed to be scheduled in a given episode and resubmits them in the next.

```
user = (-2)
userP conf net = do
  stale <- foldM buy [] [0 .. _u conf]
  forM_ stale $ \w -> report net $ ReportFail  (_wId w)
  send net clock Sleep          -- >E
  where
    buy stale _ =
      do (sender, Tick now) <- recv net
         send net sender Tock
         (_, Wake) <- recv net
         let Workload _ workload = _workload conf
         success <-
           forM (stale ++ workload conf now) $ \w ->
             -- do a <- randomRIO (0, _n conf -1)     -- RANDOM
             do send net (_wNode w) $ Run w           -- AS PLANNED
                (_, Ran r old new) <- recv net
                dumpAcc net r now old new
                return (r, w)
         send net clock $ Sleep -- <E
         return $ map snd $ filter (not . fst) success
dumpAcc net r t old new =
  out net (if r then "acc" else "!acc") $
        ("i",_id old,"t",t,"s",_submit old,"b",_boss old,
         "old",(fromMaybe (-1) $ _start old, Map.keys $ _team old),
         "new",(fromMaybe (-1) $ _start new, Map.keys $ _team new))
```

A.8 Primary overlay

The management of the primary overlay is done according to the gossip algorithm. Each node has a neighborhood stored as an array of node addresses and interpreted as a multiset of neighbors:

```
type Neighborhood = IOUArray Int Addr

newNeighborhood :: [Addr] -> IO Neighborhood
newNeighborhood iniNeighbors =
    newListArray (0, length iniNeighbors-1) iniNeighbors

setNeighbor :: Neighborhood -> (Int,Addr) -> IO ()
setNeighbor n (i,a) = writeArray n i a

getNeighbor :: Neighborhood -> Int -> IO Addr
getNeighbor = readArray

getNeighbors :: Neighborhood -> IO [Addr]
getNeighbors = getElems
```

In order to maintain the primary overlay in a state approximating a random graph, nodes will need to periodically pick random sub-multisets of their neighbor multisets.

```
pickNeighbors ::  Int -> Neighborhood -> IO [(Int,Addr)]
pickNeighbors n nb =
  replicateM n $
  do bounds <- getBounds nb
     i      <- randomRIO bounds
     x      <- getNeighbor nb i
     return (i,x)
```

For the purposes of the part of GOSSIPTRON that follows Oner's scheduling algorithm, you also need to be able to pick a given number of distinct neighbors. This algorithm is a little clumsy, but it turns out upon profiling not to be a bottleneck.

```
pickDistinctNeighbors :: Int -> Neighborhood -> IO [Addr]
pickDistinctNeighbors m set =   -- Don't optimize
  do nb <- getElems set
     let xs = nub (sort nb)
     pickDistinct m (listArray (0, length xs - 1) xs)
  where
    pickDistinct m xs =
      let (lb,ub) = bounds xs
          n = ub - lb in
      if m > ub - lb then
          return []
      else
        let p 0 acc = return acc
            p m acc = do i <- randomRIO (0, n - 1)
                         let x = xs ! i
                         (if x `Set.member` acc then
                            p m acc
                          else
                            p (m - 1) (Set.insert x acc)) in
```

```
        do r <- p m Set.empty
           return $ Set.toList r
```

A.9 Node schedules

In order to implement the part of GOSSIPTRON that follows Oner's scheduling algorithm, each node needs t omaintain a node schedule that records what jobs will be run at each future moment on that node. In order to save memory and improve processing speed of these schedules, I record them not as maps of times to jobs, but as "Intermaps" of time intervals to jobs.

```
data NodeSch = NodeSch Addr (Intermap Job)
instance Show NodeSch where
    show (NodeSch _ ns) = interShow (\j -> show (_id j,_ticks j)) ns

newNodeSch :: Addr -> IO (IORef NodeSch)
newNodeSch a = newIORef (NodeSch a Map.empty)

getNodeSch :: IORef NodeSch -> IO NodeSch
getNodeSch = readIORef

nodeGarbage :: IORef NodeSch -> Time -> IO ()
nodeGarbage nsRef now =
  do NodeSch a ns <- readIORef nsRef
     writeIORef nsRef (NodeSch a (interCleanup now ns))

nodeCommit :: IORef NodeSch -> Job -> IO ()
nodeCommit nsRef job =
  do ns@(NodeSch a _) <- readIORef nsRef
     when (Map.member a (_team job)) $
          writeIORef nsRef (nodeAddJob job ns)

nodeRelease :: IORef NodeSch -> Job -> IO ()
nodeRelease nsRef job =
  do ns <- readIORef nsRef
     writeIORef nsRef (nodeRemoveJob job ns)

nodeFree :: IORef NodeSch -> Job -> Addr -> IO Bool
nodeFree nsRef Job { _start=Nothing } _                  = return True
nodeFree nsRef job@Job { _start=Just s, _ticks=d } addr =
  if (not (Map.member addr (_team job))) then
    return True
  else
    do (NodeSch _ ns) <- readIORef nsRef
       return $ not $ interMember (interval s (s + d - 1)) ns

nodeAllJobs :: IORef NodeSch -> IO [Job]
nodeAllJobs nsRef =
```

```
  do NodeSch _ ns <- readIORef nsRef
     return $ interVals ns

nodeAddJob :: Job -> NodeSch -> NodeSch
nodeAddJob j@Job { _start=Just s, _ticks=d } (NodeSch a ns) =
  NodeSch a (interInsert (interval s (s + d - 1)) j ns)

nodeRemoveJob :: Job -> NodeSch -> NodeSch
nodeRemoveJob j@Job { _start=Nothing } n = n
nodeRemoveJob j@Job { _start=Just s, _ticks=d } (NodeSch a ns) =
  NodeSch a (interWipeWhen (== j) (interval s (s + d - 1)) ns)
```

A.10 Agendas

In order to implement the part of GOSSIPTRON that follows Oner's scheduling algorithm, each node needs to maintain an agenda that records data about the jobs it is "boss" for.

```
data Agenda = Agenda (Map.Map JobId Job) deriving Show
data Job =
  Job { _id :: JobId,                    -- job ID number
        _size :: Int,                    -- size of team
        _ticks :: Int,                   -- ticks required
        _team :: Map.Map Addr NodeSch,   -- node schedules
        _submit :: Int,                  -- time submitted
        _start :: Maybe Int,             -- start time
        _nak :: Int,                     -- msgs pending
        _nok :: Int,                     -- msgs pending or !OK
        _user :: Addr,                  -- user
        _boss :: Addr                    -- boss
      } deriving Show
```

Two jobs with the same ID are considered equal.

```
instance Eq Job where a == b = (_id a == _id b)

newAgenda :: IO (IORef Agenda)
newAgenda = newIORef (Agenda Map.empty)

newJob :: JobId -> Int -> Int -> Addr -> Addr -> Int -> Job
newJob id size ticks user boss submit =
  Job { _id=id, _size=size, _ticks=ticks, _team=Map.empty,
           _submit=submit, _start=Nothing, _nak=0, _nok=0,
           _user=user, _boss=boss }

nulJob :: Job -> Job
nulJob job = job { _team=Map.empty, _start=Nothing }
```

```
agendaNow :: IORef Agenda -> Time -> IO [Job]
agendaNow ref t =
  do Agenda agenda <- readIORef ref
     return $ Map.elems (Map.filter ((== Just t) . _start) agenda)

agendaNewJob ::
  IORef Agenda -> JobId -> Int -> Int -> Addr -> Addr -> Int -> IO Job
agendaNewJob ref j size ticks user boss now =
  do Agenda agenda <- readIORef ref
     let new = newJob j size ticks user boss now
     writeIORef ref $ Agenda (Map.insert j new agenda)
     return new

agendaJobCommit :: IORef Agenda -> Job -> IO ()
agendaJobCommit ref job =
  do Agenda agenda <- readIORef ref
     writeIORef ref (Agenda (Map.insert (_id job) job agenda))

agendaJobRelease :: IORef Agenda -> Job -> IO ()
agendaJobRelease ref Job { _start=Nothing }  =
  return ()
agendaJobRelease ref job  =
  do Agenda agenda <- readIORef ref
     writeIORef ref (Agenda (Map.delete (_id job) agenda))

agendaJobIni :: IORef Agenda -> JobId -> Int -> IO Int
agendaJobIni ref j n =
  do Agenda agenda <- readIORef ref
     old <- Map.lookup j agenda
     case _nak old of
       0 ->
         do let new = old { _nak=n, _nok=n }
            writeIORef ref $ Agenda (Map.insert j new agenda)
            return 0
       nak ->
         do return nak

agendaJobDown :: IORef Agenda -> JobId -> Bool -> IO Int
agendaJobDown ref j accepted =
  do Agenda agenda <- readIORef ref
     old <- Map.lookup j agenda
     let new = old { _nak=_nak old - 1,
                     _nok=_nok old - (if accepted then 1 else 0) }
     writeIORef ref $ Agenda (Map.insert j new agenda)
     return (_nak new)

agendaJobNak ref j =
  do Agenda agenda <- readIORef ref
     old <- Map.lookup j agenda
     return (_nak old)
```

```
agendaJobAccepted :: IORef Agenda -> JobId -> IO Bool
agendaJobAccepted ref j  =
  do Agenda agenda <- readIORef ref
     old <- Map.lookup j agenda
     return (_nok old == 0)

agendaJobAddNode :: IORef Agenda -> JobId -> Addr -> NodeSch -> IO ()
agendaJobAddNode ref j node nodeSch =
  do Agenda agenda <- readIORef ref
     old <- Map.lookup j agenda
     let new = old { _team=Map.insert node nodeSch (_team old) }
     writeIORef ref $ Agenda (Map.insert j new agenda)

agendaJobUpdateNode :: IORef Agenda -> JobId -> Addr -> NodeSch -> IO ()
agendaJobUpdateNode ref j node nodeSch =
  do Agenda agenda <- readIORef ref
     case Map.lookup j agenda of
       Nothing ->
         return ()
       Just old@Job { _team=t } ->
         if Map.member node t then
           let new = old { _team=Map.insert node nodeSch t } in
           writeIORef ref $ Agenda (Map.insert j new agenda)
         else
           return ()

agendaJob :: IORef Agenda -> JobId -> IO Job
agendaJob ref j =
  do Agenda agenda <- readIORef ref
     Map.lookup j agenda
```

When a new team has been assembled for a new job, it is the task of the boss to find the first time at which all team members are available. It does this simply by trying out time periods until it finds one that fits for everyone. It might be possible to optimize this, but it turns out upon profiling that this procedure is not a bottleneck.

```
agendaJobLaunch :: IORef Agenda -> JobId -> Time -> IO ()
agendaJobLaunch ref j start =
  do Agenda agenda <- readIORef ref
     old <- Map.lookup j agenda
     writeIORef ref $
                 Agenda (Map.insert j (launch old start) agenda)
  where
    launch old start =
      let iv = interval start (start + _ticks old - 1) in
      if or [ interMember iv s | NodeSch _ s <- Map.elems (_team old) ] then
        launch old (start+1)
      else
```

```
        let newNodes = Map.map (nodeAddJob nu) (_team old)
            nu = old { _team=newNodes, _start = Just start } in nu
```

The big trick of Oner's algorithm is the ability to improve schedules by adding a new member to a team and simultaneousy removing another. When it is suggested that a given member could be added to a job's team, it is the task of the boss to determine whether it would be possible to schedule that job earlier than it is currently scheduled by using this new member. This is a costly operation and it happens often, so it may be worth trying to think about how to optimize it.

```
agendaImprove :: IORef Agenda -> Int -> Int -> NodeSch -> IO [(Job,Job)]
agendaImprove ref now hireAddr hire =
  do Agenda agenda <- readIORef ref
     return $ concatMap improve (Map.elems (Map.filter fresh agenda))
  where
    fresh Job { _start=Just t, _submit=s } | t>now &&  s<now = True
    fresh _ = False
    improve old =
      let teams =
            [ Map.map (nodeRemoveJob old) $
              Map.insert hireAddr hire $
              Map.delete fire (_team old)
                  | fire <- Map.keys (_team old),
                    not (Map.member hireAddr (_team old)) ]
          starts =
            [ now+1 .. (fromMaybe 0 (_start old)) - 1 ]
          options =
            [ (team, start)
                | start <- starts,
                  team <- teams,
                  let iv = interval start (start+_ticks old - 1),
                  let meToo (NodeSch _ me) us = interMember iv me || us,
                  not $ Map.fold meToo False team ] in
      case options of
        [] ->
          []
        ((team,start):_) ->
          let new = old { _team=Map.map (nodeAddJob new) team,
                          _start=Just start, _nok=0, _nak=0 } in
          [(old,new)]
```

A.11 Buckets

The bucket is a data structure used by the clock node.

```
type Bucket   = IORef Int
newBucket n   = newIORef n
```

```
bucketIn u    = do x <- readIORef u; writeIORef u (x + 1)
bucketOut u   = do x <- readIORef u; writeIORef u (x - 1)
bucketEmpty u = do x <- readIORef u; return (x == 0)
```

A.12 Accounting

Accounting is not a feature of the Oner's original scheduler. It is new in GOSSIPTRON. A `Ledger` represents what a node remembers about its history of interaction with other nodes. For each known node it records an `Account`. Each account has three entries: `_iou` (for "IOU" or "I Owe You") and `_uoi` (for "UOI" or "You Owe Me"), and `_limit`, which is used to compute credit limits.

```
type Ledger = IORef (Map.Map Addr Account)
data Account = Account { _iou, _uoi, _limit :: Double } deriving Show

newLedger :: IO Ledger
newLedger = newIORef Map.empty

depreciate :: Conf -> Ledger -> IO ()
depreciate Conf { _D=depreciation } ref =
  do old <- readIORef ref
     let dep a@Account{_limit=limit} = a { _limit = limit * (1-depreciation) }
     let new = Map.map dep old
     writeIORef ref new
```

Check whether they are worthy of a favor...

```
liquid :: Conf -> Ledger -> Addr -> IO Bool
liquid Conf { _E=escalation, _G=generosity } ref addr =
  if generosity < 0 then
    return True
  else
    do old <- readIORef ref
       let Account { _uoi=uoi,_iou=iou,_limit=limit} =
             Map.findWithDefault (Account 0 0 0) addr old
       return $ iou  + generosity + escalation * limit > uoi
```

Record that we owe them a favor...

```
iou :: Ledger -> Addr -> IO ()
iou ref addr =
  do old <- readIORef ref
     let acct = Map.findWithDefault (Account 0 0 0) addr old
     let new = Map.insert addr acct { _iou=_iou acct+1,
                                      _limit=_limit acct+1 } old
     writeIORef ref new
```

Record that we don't owe them a favor after all...

```
uniou :: Ledger -> Addr -> IO ()
uniou ref addr =
  do old <- readIORef ref
     let acct = Map.findWithDefault (Account 0 0 0) addr old
     let new = Map.insert addr acct { _iou=_iou acct-1 } old
     writeIORef ref new
```

Record that they owe us a favor...

```
uoi :: Ledger -> Addr -> IO ()
uoi ref addr =
  do old <- readIORef ref
     let acct = Map.findWithDefault (Account 0 0 0) addr old
     let new = Map.insert addr acct { _uoi=_uoi acct+1 } old
     writeIORef ref new
```

Record that they don't owe us a favor after all...

```
unuoi :: Ledger -> Addr -> IO ()
unuoi ref addr =
  do old <- readIORef ref
     let acct = Map.findWithDefault (Account 0 0 0) addr old
     let new = Map.insert addr acct { _uoi=_uoi acct-1 } old
     writeIORef ref new
```

A.13 Keeping track of chains of credit

Chains of credit need to be kept track of because they must sometimes be torn down again if a team changes. This is what the `Brokerage` object is for. The details of this are not described in the main text of the thesis, because they are not very interesting.

```
type Brokerage = Map.Map JobId (Map.Map Addr [Addr])

newBrokerage :: IO (IORef Brokerage)
newBrokerage = newIORef Map.empty

brokerageAdd :: IORef Brokerage -> JobId -> Addr -> [Addr] -> IO ()
brokerageAdd ref j node path =
  do b <- readIORef ref
     let new = Map.insert node path (Map.findWithDefault Map.empty j b)
     writeIORef ref $ (Map.insert j new b)

brokerageRemove :: IORef Brokerage -> JobId -> Addr -> IO ()
brokerageRemove ref j node =
  do b <- readIORef ref
```

```
    let new = Map.delete node (Map.findWithDefault Map.empty j b)
    writeIORef ref $ (Map.insert j new b)

brokerageGet :: IORef Brokerage -> JobId -> IO (Map.Map Addr [Addr])
brokerageGet ref j =
  do b <- readIORef ref
     return (Map.findWithDefault Map.empty j b)
```

In order for the secondary overlay to work, we have to impose a spatial structure on the nodes. That is what these three functions do.

```
-- Sort a list of addresses by proximity to a target address
route :: [Addr] -> Addr -> [Addr]
route partners target = sortBy (comparing (distance target)) partners

-- Map an integer to a point in the unit square with FNV hash
hashXY :: Int -> (Double,Double)
hashXY i =
  let scale x = fromIntegral (x .&. 0xffff) / fromIntegral (maxBound::Word16)
      fnv32 i = foldl (\h x -> (h * 16777619) `xor` x) 2166136261
                [ 0xff .&. (i`shiftR`(y*8)) | y <- [0..(bitSize i`div`8)-1]]
  in (scale (fnv32 i), scale (shiftR (fnv32 i) 16))

-- Distance measure using the hash
distance :: Addr -> Addr -> Double
distance p0 p1 =
  let (x0,y0) = hashXY p0; (x1,y1) = hashXY p1 in sqrt ((x1-x0)^2+(y1-y0)^2)
```

A.14 Some common parameters used in simulation

A circular primary overlay topology.

```
circular :: Ovl1
circular =
  Ovl1 "circular" $ \ Conf {_n=n,_s=s} ->
    do return $ listArray (0,n-1) [ [ x`mod`n | x<-[i+1..i+s] ]
                                    | i<-[0..n-1] ]
```

A randomized primary overlay topology

```
randomized :: Ovl1
randomized =
  Ovl1 "r" $ \ Conf {_n=n,_s=s} ->
    return $
    accumArray (flip (:)) [] (0,n-1) $
    zip (concat $ replicate s [0..n-1]) (randomRs (0,n-1) (mkStdGen seed))
  where
    seed = 31415
```

A secondary overlay topology where each node is connected to each other node.

```
dense :: Ovl2
dense =
  Ovl2 "d" $ \ Conf {_n=n} ->
    do return $ listArray (0,n-1) (repeat [0..n-1])
```

A secondary overlay topology where nodes are connected as a Delaunay graph according to coordinates determined from their addresses by the *hashXY* hash function, and another topology that is identical but with the addition of long range links, more or less the same way as in VORONET (but not quite.)

```
shell cmd inpS =
  do (inp,out,err,pid) <- runInteractiveCommand cmd
     hPutStr inp inpS
     hClose inp
     outS <- hGetContents out
     errS <- hGetContents err
     status <- waitForProcess pid
     return (outS,errS,status)

delone :: Ovl2
delone =
  Ovl2 "d" $ \ Conf {_n=n} ->
    do let input = unlines $ ["2",show n] ++
                      map (\i -> let (x,y) = hashXY i in
                                 show x ++ " " ++ show y) [0..n-1]
       (output,_,_) <- shell "qdelaunay i" input
       let readTriangle t =
             case map read t of
               [a,b] -> [(a,b),(b,a)]
               [a,b,c]-> [(a,b),(b,c),(c,a),(b,a),(c,b),(a,c)]
               [a,b,c,d]-> [(a,b),(b,c),(c,d),(d,a),(b,a),(c,b),(d,c),(a,d)]
               xs -> error (show xs)
       let delaunayEdges =
             nub $ sort $ concatMap readTriangle $ map words $ tail $ lines output
       let ovl = accumArray (flip (:)) [] (0,n-1) $ delaunayEdges
       return ovl

voronet :: Ovl2
voronet =
  Ovl2 "v" $ \ Conf {_n=n} ->
    do let input = unlines $ ["2",show n] ++
                      map (\i -> let (x,y) = hashXY i in
                                 show x ++ " " ++ show y) [0..n-1]
       (output,_,_) <- shell "qdelaunay i" input
       let readTriangle t =
             case map read t of
               [a,b] -> [(a,b),(b,a)]
```

```
              [a,b,c]-> [(a,b),(b,c),(c,a),(b,a),(c,b),(a,c)]
              [a,b,c,d]-> [(a,b),(b,c),(c,d),(d,a),(b,a),(c,b),(d,c),(a,d)]
              xs -> error (show xs)
       let delaunayEdges =
             nub $ sort $ concatMap readTriangle $ map words $ tail $ lines output
       let farEdges = [ (a,fnv32 a`mod`n) | a <- [0..n-1] ]
       let farEdges' = map (\(a,b)->(b,a)) farEdges
       let ovl = accumArray (flip (:)) [] (0,n-1) $ delaunayEdges ++ farEdges
       return ovl
```

A very common hash function.

```
fnv32 i = fromIntegral (f (fromIntegral i))
  where f :: Word32 -> Word32
        f i = foldl (\h x -> (h * 16777619) `xor` x) 2166136261
              [ 0xff .&. (i`shiftR`(y*8)) | y <- [0..(bitSize i`div`8)-1]]
```

A workload imported from a GWF file.

```
fromTrace :: Int -> FilePath -> IO Workload
fromTrace n fn =
  do gs <- readSimpleGwfFile n fn
     return $ Workload "fn" $ \conf t ->
       Map.findWithDefault [] t $ Map.fromListWith (++) $
       [ ( t`div`60,
           [ Work { _wStart=t`div`60,
                    _wId=i,
                    _wLength=(d`div`60)+1,
                    _wSize=m,
                    _wNode=a`div`_n conf }] )
         | (SimpleGwf i t m d, a) <- zip gs [0..] ]
```

Some other workloads.

```
steady :: Int -> Int -> Workload
steady intensity tMax =
  Workload "" $ \ conf t ->
    [ Work { _wStart=t,
             _wId=id,
             _wLength=5,
             _wSize=5,
             _wNode= id `mod`_n conf }
      | i <- [0..intensity-1], t < tMax, let id=t*intensity+i ]

switchy :: Int -> Int -> Workload
switchy intensity tMax =
  Workload "" $ \ conf t ->
    [ Work { _wStart=t,
             _wId=intensity*t+i,
```

```
            _wLength=5,
            _wSize=5,
            _wNode=
               if t < tMax`div`2 then
                 ev id `mod`_n conf
               else
                 od id `mod`_n conf
          }
      | i <- [0..intensity-1], t < tMax, let id=t*intensity+i ]
  where
    ev n = (n`div`2)*2
    od n = (n`div`2)*2+1
```

A.15 Run the simulation

These are a number of sample configurations for the simulation. You can design your own configurations by analogy to these.

```
t            = Conf { _n=100, _u=100, _s=10, _c=2, _f=1, _g=1,
                      _T=1, _G=1, _E=0.6, _D=0.01, _cheaters=[],
                      _ovl1=randomized, _ovl2=dense, _workload=steady 0 0 }

l            = t { _workload=steady 2 10 }
lite t f     = gossiptron t >>= run False ["!acc"] >>= tex f
lite_ "o"    = lite l  { _T=1,  _G=(-1),  _ovl2=dense   } "lite/o"
lite_ "r"    = lite l  { _T=1,  _G=1,     _ovl2=dense   } "lite/r"
lite_ "l"    = lite l  { _T=1,  _G=1,     _ovl2=voronet } "lite/l"
lite_ "t5"   = lite l  { _T=5,  _G=1,     _ovl2=voronet } "lite/t5"
lite_ "t10"  = lite l  { _T=10, _G=1,     _ovl2=voronet , _u=200} "lite/t10"
lite_ "t99"  = lite l  { _T=99, _G=1,     _ovl2=voronet , _u=200} "lite/t99"
h            = t { _workload=steady 10 10 }
heavy t f    = gossiptron t >>= run False ["!acc"] >>= tex f
heavy_ "o"   = heavy h  { _T=1,  _G=(-1), _ovl2=dense   } "heavy/o"
heavy_ "r"   = heavy h  { _T=1,  _G=1,    _ovl2=dense   } "heavy/r"
heavy_ "l"   = heavy h  { _T=1,  _G=1,    _ovl2=voronet } "heavy/l"
heavy_ "t5"  = heavy h  { _T=5,  _G=1,    _ovl2=voronet } "heavy/t5"
heavy_ "t10" = heavy h  { _T=10, _G=1,    _ovl2=voronet,_u=200 } "heavy/t10"
heavy_ "t99" = heavy h  { _T=99, _G=1,    _ovl2=voronet,_u=200 } "heavy/t99"
c            = t { _workload=steady 10 10, _cheaters=[1,10..99] }
cheat t f    = gossiptron t >>= run False ["!acc"] >>= tex f
cheat_"o"    = cheat c  { _T=1,  _G=(-1), _ovl2=dense   } "cheat/o"
cheat_ "r"   = cheat c  { _T=1,  _G=1,    _ovl2=dense   } "cheat/r"
cheat_ "l"   = cheat c  { _T=1,  _G=1,    _ovl2=voronet } "cheat/l"
cheat_ "t5"  = cheat c  { _T=5,  _G=1,    _ovl2=voronet } "cheat/t5"
cheat_ "t10" = cheat c  { _T=10, _G=1,    _ovl2=voronet,_u=200 } "cheat/t10"
cheat_ "t99" = cheat c  { _T=99, _G=1,    _ovl2=voronet,_u=200 } "cheat/t99"
s            = t { _workload=switchy 10 10 }
```

```
wacky t f    = gossiptron t >>= run False ["!acc"] >>= tex f
wacky_ "o"   = wacky s  { _T=1,  _G=(-1), _ovl2=dense   } "wacky/o"
wacky_ "r"   = wacky s  { _T=1,  _G=1,    _ovl2=dense   } "wacky/r"
wacky_ "l"   = wacky s  { _T=1,  _G=1,    _ovl2=voronet } "wacky/l"
wacky_ "t5"  = wacky s  { _T=5,  _G=1,    _ovl2=voronet } "wacky/t5"
wacky_ "t10" = wacky s  { _T=10, _G=1,    _ovl2=voronet,_u=200 } "wacky/t10"
wacky_ "t99" = wacky s  { _T=99, _G=1,    _ovl2=voronet,_u=200 } "wacky/t99"
```

A.16 Driver code

This is the bit that processes command line arguments.

```
main =
  do [exp,alg] <- System.getArgs
     case exp of
       "heavy" -> heavy_ alg
       "cheat" -> cheat_ alg
       "lite"  -> lite_ alg
       "wacky" -> wacky_ alg
     return ()
```

A.17 Interval maps

In this section, we will define the `Intermap`, a data structure that behaves very similarly to a standard Haskell `Map`, but that is specialized to deal with the case where the keys are integer ranges. This data structures is used to implement some of the schedules more efficiently.

An `Intermap` is simply a map from intervals to values, of some arbitrary type `t`. Intervals are pairs of integers representing the beginning and end of the interval. They are considered equal by the `Eq` typeclass only when their beginning and end points are exactly identical, but the `Ord` typeclass specifies a partial order where all overlapping intervals are "`EQ`", which turns out to be useful because we do not want to have overlapping entries in our `Intermaps`.

```
type Intermap t  = Map.Map Interval (Interval,t)
newtype Interval = Interval (Int, Int)

instance Show Interval where
    show (Interval (x0,x1)) = show x0++"-"++show x1

instance Eq Interval where
    Interval (x0,x1) == Interval (y0,y1) = x0 == y0 && x1 == y1

instance Ord Interval where
    compare (Interval (x0,x1)) (Interval (y0,y1)) | x1 < y0   = LT
```

```
                                                  | y1 < x0   = GT
                                                  | otherwise = EQ

interEmpty :: Intermap t
interEmpty = Map.empty

interShow :: (t -> String) -> Intermap t -> String
interShow showVal m =
    "["++ (concat $ intersperse ", " $
           map (\(k,v) -> show k++":"++showVal v) (Map.elems m)) ++ "]"

interInsert :: Interval -> t -> Intermap t -> Intermap t
interInsert k v m =
    foldr (uncurry interAdd) (interWipe k m) additions
    where
      interAdd k v m = Map.insert k (k,v) m
      additions = (k,v):[ (q,v_)
                        | (k_,v_) <- interLookup k m, q <- k_ .-. k ]
      Interval (x0,x1) .-. Interval (y0,y1) =
          msum [mInterval x0 (min x1 (y0-1)), mInterval (max x0 (y1+1)) x1]

interFindWhen :: (t -> Bool) -> Int -> Intermap t -> Maybe (Interval,t)
interFindWhen f t m =
    case Map.lookup (instant t) m of
      Just (k,v) | f v -> Just (k,v)
      _                -> Nothing

interFind :: Int -> Intermap t -> Maybe (Interval,t)
interFind t m = interFindWhen (\_ -> True) t m

interLookupWhen :: (t -> Bool) -> Interval -> Intermap t -> [(Interval,t)]
interLookupWhen f (Interval (k0,k1)) m =
    let (_,a,x) = Map.splitLookup (instant k0) m
        (y,b,_) = Map.splitLookup (instant k1) x in
    filter (f . snd) $ concatMap maybeToList [a,b] ++ Map.elems y

interLookup :: Interval -> Intermap t -> [(Interval,t)]
interLookup k m = interLookupWhen (\_ -> True) k m

interMemberWhen :: (t -> Bool) -> Interval -> Intermap t -> Bool
interMemberWhen f k m = not $ null $ interLookupWhen f k m

interMember :: Interval -> Intermap t -> Bool
interMember k m = Map.member k m

interWipeWhen :: (t -> Bool) -> Interval -> Intermap t -> Intermap t
interWipeWhen f k m = Map.filter (f . snd) m

interWipe :: Interval -> Intermap t -> Intermap t
interWipe k m = interWipeWhen (\_ -> True) k m
```

```
interCleanup :: Int -> Intermap t -> Intermap t
interCleanup t m =
    case Map.splitLookup (interval t t) m of
      (_,Nothing,x)    -> x
      (_,Just (k,v),x) -> Map.insert k (k,v) x

interVals :: Intermap t -> [t]
interVals m =
    map snd $ Map.elems m


instant :: Int -> Interval
instant x = Interval (x,x)

interval :: Int -> Int -> Interval
interval x y | x <= y    = Interval (x,y)
             | otherwise = error "bad interval"

mInterval :: Monad m => Int -> Int -> m Interval
mInterval x y | x <= y    = return $ Interval (x,y)
              | otherwise = fail "bad interval"
```

A.18 Statistics

These functions do some basic descriptive statistics.

```
-- Compute the mean of a list of numbers
mean :: Integral a => [a] -> Double
mean = mean' . map fromIntegral
  where mean' xs = sum xs / fromIntegral (length xs)

-- Compute the standard deviation of a list of numbers
sd :: Integral a => [a] -> Double
sd = sd' . map fromIntegral
  where sd' xs = sqrt (sum [ (x-x_)**2 | x <- xs, let x_ = mean' xs ] )
        mean' xs = sum xs / fromIntegral (length xs)

-- Compute the median of a list of numbers
median :: Integral a => [a] -> Double
median = median' . map fromIntegral
  where median' xs | odd (length xs) = xs !! (length xs `div` 2)
                   | otherwise = (xs !! (length xs `div` 2)+
                                  xs !! ((length xs `div` 2)+1))/2
```

A.19 Data analysis and visualization

```
-- Print the maximum delay to stdout
mxd report =
  do print $ maximum  [ n | ReportDelay n <- report ]
     return report

-- Print the log messages to stdout unmodified
raw report =
  do mapM_ print report
     return report

-- Show various bits of data on X11
gui level report =
  do when (level>=0) $ summarize "delays" delays
     when (level>=1) $ histogram "x11" "temp" title xlabel 10 delays
     when (level>=2) $ graphic "neato" ovl2 (>)
     when (level>=3) $ graphic "circo" ovl1 (/=)
     return report
  where
    delays = [ n | ReportDelay n <- report ]
    failed = [ n | ReportFail n <- report ]
    ovl1   = [ (x,xs) | ReportOvl1 x xs <- report ]
    ovl2   = [ (x,xs) | ReportOvl2 x xs <- report ]
    [conf] = [ p | ReportConf p <- report ]
    title  = show conf
    xlabel = "and " ++ show (length failed) ++ " failed"

-- Output a histogram of delays to a LaTeX file
tex name report =
  do histogram "tex" name "" xlabel 10 delays
     writeFile (name ++ "c.tex") title
     writeFile (name ++ ".err") (show (length failed))
     return report
  where
    delays = [ n | ReportDelay n <- report ]
    failed = [ n | ReportFail n <- report ]
    ovl1   = [ (x,xs) | ReportOvl1 x xs <- report ]
    ovl2   = [ (x,xs) | ReportOvl2 x xs <- report ]
    [conf] = [ p | ReportConf p <- report ]
    xlabel = "and " ++ show (length failed) ++ " failed"
    title  =
      let Conf { _n=n,_u=u,_s=s,_c=c,_f=f,_g=g,_E=ee,_G=gg,_D=dd,_T=tt,
                 _ovl1=ovl1, _ovl2=ovl2, _workload=workload } = conf in
      foo "n" n ++
      foo "u" u ++
      foo "\\sigma" s ++
      foo "\\kappa" c ++
      foo "\\phi" f ++
      foo "\\gamma" g ++
```

```
      foo "G" gg ++
      foo "E" ee ++
      foo "100\\times D" (100*dd) ++
      foo "T" tt
    foo x y = "$" ++ x ++ "=" ++ show y ++ "$ "

-- Print a summary of the data on stdout
summarize :: String -> [Int] -> IO ()
summarize name vals =
  do printf "\tSUMMARY\t(%s) mean=%.4f median=%.4f sd=%.4f N=%d\n"
            name (mean vals) (median vals) (sd vals) (length vals)

-- Plot a histogram using gnuplot
histogram :: String -> String -> String -> String -> Int -> [Int] -> IO ()
histogram terminal name title xlabel bins vals =
  do withFile fnDat WriteMode $ \fd ->
       forM_ frequencies $ \(delay, frequency) ->
         hPrintf fd "%f %f\n" delay frequency
     when (terminal == "x11") $
       do withFile fnPlt WriteMode $ \fd ->
            do hPrintf fd "unset title\n"
               hPrintf fd "unset border\n"
               hPrintf fd "set xlabel \"%s\"\n" xlabel
               hPrintf fd "set title \"%s\"\n" title
               hPrintf fd "plot \"%s\" notitle with boxes\n" fnDat
          runCommand ("gnuplot " ++ fnPlt) >>= waitForProcess >> return ()
     return ()
  where
    fnDat = name ++ ".dat"
    fnErr = name ++ ".err"
    fnPlt = name ++ ".plt"
    frequencies :: [(Double,Double)]
    frequencies = [ ((dbl n0 + dbl (n1 + 1))/2, dbl f)
                      | n0 <- take bins [min, min + bin ..],
                        let n1 = n0 + bin - 1,
                        let inBin n0 n1 n = n >= n0 && n <= n1,
                        let f = length (filter (inBin n0 n1) vals) ]
    dbl = fromInteger . toInteger
    min = if null vals then 0 else minimum vals
    max = if null vals then 40 else maximum vals
    bin = ((max - min) `div` bins) + 1

-- Draw a digraph using circo/neato
graphic :: String -> [(Int,[Int])] -> (Int -> Int -> Bool) -> IO ()
graphic tool graph filter =
  do withFile "dat/neighbors.dot" WriteMode $ \fd -> hPutStr fd viz
     runCommand cmd >>= waitForProcess
     return ()
  where
    nodes = [ (x,()) | (x,_) <- graph ]
```

```
    edges = concat [ [ (x,y,()) | y <- ys,filter x y] | (x,ys) <- graph ]
    viz   = graphviz (mkGraph nodes edges::UGr) "" (6,6) (1,1) Portrait
    cmd = case tool of
            "n" -> "neato -Earrowhead=none -Nfontname=georgia -Nfontsize=10 \
                    \-Nheight=0.1 -Nwidth=0.1 -Ncolor=white -Txlib neighbors.dot"
            "c" -> "circo -Nfixedsize=true -Earrowhead=none -Txlib neighbors.dot"
```

A.20 Parsing GWF files

```
-- This is the meaning of the GWF fields according to a DAS data file:
--
-- 1  JobID              counter
-- 2  SubmitTime         in seconds, starting from zero
-- 3  WaitTime           in seconds
-- 4  RunTime            runtime measured in wallclock seconds
-- 5  NProcs             number of allocated processors
-- 6  AverageCPUTimeUsed average of CPU time over all allocated processors
-- 7  Used Memory        average per processor in kilobytes
-- 8  ReqNProcs          requested number of processors
-- 9  ReqTime:           requested time measured in wallclock seconds
-- 10 ReqMemory          requested memory (average per processor)
-- 11 Status             job completed = 1, job failed = 0, job cancelled = 5
-- 12 UserID             string identifier for user
-- 13 GroupID            string identifier for group user belongs to
-- 14 ExecutableID       name of executable
-- 15 QueueID            string identifier for queue
-- 16 PartitionID        string identifier for partition
-- 17 OrigSiteID         string identifier for submission site
-- 18 LastRunSiteID      string identifier for execution site
-- 19 JobStructure       single job = UNITARY, composite job = BoT
-- 20 JobStructureParams if JobStructure = BoT, contains batch identifier
-- 21 UsedNetwork        used network resources in kilobytes/second
-- 22 UsedLocalDiskSpace in megabytes
-- 23 UsedResources      list of comma-separated generic resources
--                             (ResourceDescription:Consumption)
-- 24 ReqPlatform        CPUArchitecture,OS,OSVersion
-- 25 ReqNetwork         in kilobytes/second
-- 26 ReqLocalDiskSpace  in megabytes
-- 27 ReqResources       list of comma-separated generic resources
--                             (ResourceDescription:Consumption)
-- 28 VOID               identifier for Virtual Organization
-- 29 ProjectID          identifier for project

-- I have assumed that doubles are intended whenever the sample data
-- file had numbers with decimal points in them, and integers
-- elsewhere. XXX Check this in official docs, if there are any!

-- REAL GWF RECORDS ------------------------------------------------
```

```
data Gwf = Gwf { _gwfJobid :: Int,
                 _gwfSubmittime :: Int,
                 _gwfWaittime :: Int,
                 _gwfRuntime :: Int,
                 _gwfNprocs :: Int,
                 _gwfAveragecputimeused :: Double,
                 _gwfUsedmemory :: Double,
                 _gwfReqnprocs :: Int,
                 _gwfReqtime :: Double,
                 _gwfReqmemory :: Int,
                 _gwfStatus :: Int,
                 _gwfUserid :: String,
                 _gwfGroupid :: String,
                 _gwfExecutableid :: String,
                 _gwfQueueid :: String,
                 _gwfPartitionid :: Int,
                 _gwfOrigsiteid :: String,
                 _gwfLastrunsiteid :: String,
                 _gwfJobstructure :: String,
                 _gwfJobstructureparams :: Int,
                 _gwfUsednetwork :: Int,
                 _gwfUsedlocaldiskspace :: Int,
                 _gwfUsedresources :: String,
                 _gwfReqplatform :: Int,
                 _gwfReqnetwork :: Int,
                 _gwfReqlocaldiskspace :: Int,
                 _gwfReqresources :: Int,
                 _gwfVoid :: Int,
                 _gwfProjectid :: Int
               }
           deriving (Show)

-- READ REAL GWF RECORDS ------------------------------------------
readGwfFile :: FilePath -> IO [Gwf]
readGwfFile fn = readFile fn >>= return . readGwf

readGwf :: String -> [Gwf]
readGwf = map parse . map splitOnTab . uncomment . lines
    where
      parse [ jobid, submittime, waittime, runtime, nprocs,
              averagecputimeused, usedmemory, reqnprocs, reqtime,
              reqmemory, status, userid, groupid, executableid, queueid,
              partitionid, origsiteid, lastrunsiteid, jobstructure,
              jobstructureparams, usednetwork, usedlocaldiskspace,
              usedresources, reqplatform, reqnetwork, reqlocaldiskspace,
              reqresources, void, projectid ] =
          Gwf { _gwfJobid = read jobid :: Int,
                _gwfSubmittime = read submittime :: Int,
                _gwfWaittime = read waittime :: Int,
                _gwfRuntime = read runtime :: Int,
```

```
                _gwfNprocs = read nprocs :: Int,
                _gwfAveragecputimeused = read averagecputimeused :: Double,
                _gwfUsedmemory = read usedmemory :: Double,
                _gwfReqnprocs = read reqnprocs :: Int,
                _gwfReqtime = read reqtime :: Double,
                _gwfReqmemory = read reqmemory :: Int,
                _gwfStatus = read status :: Int,
                _gwfUserid = userid :: String,
                _gwfGroupid = groupid :: String,
                _gwfExecutableid = executableid :: String,
                _gwfQueueid = queueid :: String,
                _gwfPartitionid = read partitionid :: Int,
                _gwfOrigsiteid = origsiteid :: String,
                _gwfLastrunsiteid = lastrunsiteid :: String,
                _gwfJobstructure = jobstructure :: String,
                _gwfJobstructureparams = read jobstructureparams :: Int,
                _gwfUsednetwork = read usednetwork :: Int,
                _gwfUsedlocaldiskspace = read usedlocaldiskspace :: Int,
                _gwfUsedresources = usedresources :: String,
                _gwfReqplatform = read reqplatform :: Int,
                _gwfReqnetwork = read reqnetwork :: Int,
                _gwfReqlocaldiskspace = read reqlocaldiskspace :: Int,
                _gwfReqresources = read reqresources :: Int,
                _gwfVoid = read void :: Int,
                _gwfProjectid = read projectid :: Int
              }

uncomment  = filter (not . (all white)) .
             map (takeWhile (/= '#'))
splitOnTab = filter (not . (all white)) .
             groupBy (\x y -> not (white x || white y))
white      = (`elem` " \t\v\n\r")

-- SIMPLIFIED GWF RECORDS ------------------------------------------

-- In practice, we only use a few elements of the GWF records, and it
-- is much more convenient to throw the rest away right
-- away. Meanwhile, we add the option to read only the first i lines
-- of a file, we modify the records to start at t=0, and we clean up
-- records with invalid reqtimes, so that the result can be used for
-- simulation without further ado. For accurate retrieval of a GWF
-- file in all its glory, the whole shebang above is still necessary,
-- but for the purposes of gossiptron, the simplified version is
-- faster and better.

data SimpleGwf = SimpleGwf { _jobid, _submittime, _reqnprocs, _reqtime :: Int }
                 deriving (Show)

readSimpleGwfFile :: Int -> FilePath -> IO [SimpleGwf]
readSimpleGwfFile i fn = readFile fn >>= return . readSimpleGwf i
```

```
readSimpleGwf :: Int -> String -> [SimpleGwf]
readSimpleGwf i = normalize . map (defaultize . parse . splitOnTab) .
                  uncomment . take i . lines
    where
      parse [ jobid, submittime, _, _, _, _, _, reqnprocs, reqtime, _,
              _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _, _ ] =
          SimpleGwf { _jobid = read jobid,
                      _submittime = read submittime,
                      _reqnprocs = read reqnprocs,
                      _reqtime = round (read reqtime :: Double) }

      normalize [] =
          []
      normalize gs@(SimpleGwf { _submittime=t0 }:_) =
          map (\g@SimpleGwf { _submittime=s } -> g { _submittime=s-t0 }) gs

      defaultize g@SimpleGwf { _reqtime=d }
          | d < 0     = g { _reqtime=defaultReqtime }
          | otherwise = g

      defaultReqtime = 60                       -- XXX check this!
```

Bibliography

[1] ANONYMOUS. What is the grid? http://gridcafe.web.cern.ch/gridcafe/whatisgrid/whatis.html.

[2] AXELROD, R. *The Evolution of Cooperation*. Basic Books, 1984.

[3] BARBER, B. *Qhull manual*. The Geometry Center, University of Minnesota http://www.qhull.org.

[4] BARMOUTA, A., AND BUYYA, R. GridBank: a Grid Accounting Services Architecture (GASA) for distributed systems sharing and integration. *Parallel and Distributed Processing Symposium, 2003* (2003), 8.

[5] BEAUMONT, O., KERMARREC, A., MARCHAL, L., AND RIVIÈRE, E. VoroNet: A scalable object network based on Voronoi tessellations. *Writing* (2006), 02.

[6] BONDY, J., AND MURTY, U. *Graph Theory with Applications*. MacMillan. http://www.ecp6.jussieu.fr/pageperso/bondy/books/gtwa/gtwa.html, 1976.

[7] COHEN, B. Incentives Build Robustness in BitTorrent. *Workshop on Economics of Peer-to-Peer Systems 6* (2003).

[8] EYMANN, T., REINICKE, M., STREITBERGER, W., RANA, O., JOITA, L., NEUMANN, D., SCHNIZLER, B., VEIT, D., ARDAIZ, O., CHACIN, P., ET AL. Catallaxy-based Grid markets. *Multiagent and Grid Systems 1*, 4 (2005), 297–307.

[9] FRIEDMAN, D. *Price Theory: an Intermediate Text*. Cincinnati, OH: South-Western publishing Co. http://www.daviddfriedman.com/Academic/Price_Theory/PThy_ToC.html, 1990.

[10] GARCIA, F., AND HOEPMAN, J. Off-Line Karma: A Decentralized Currency for Peer-to-peer and Grid Applications. In *Applied Cryptography And Network Security: Third International Conference* (2005).

[11] JELASITY, M., VOULGARIS, S., GUERRAOUI, R., KERMARREC, A., AND VAN STEEN, M. Gossip-based peer sampling. *ACM Transactions on Computer Systems* (2007).

[12] ONER, C. Decentralized grid scheduler. Master's thesis, Vrije Universiteit, 2007.

[13] PIATEK, M., ISDAL, T., ANDERSON, T., KRISHNAMURTHY, A., AND VENKATARAMANI, A. Do incentives build robustness in BitTorrent? In *4th USENIX Symposium on Networked Systems Design & Implementation* (2007).

[14] VISHNUMURTHY, V., CHANDRAKUMAR, S., AND SIRER, E. KARMA: A secure economic framework for peer-to-peer resource sharing. In *Workshop on Economics of Peer-to-Peer Systems* (2003).

[15] VOULGARIS, S., GAVIDIA, D., AND VAN STEEN, M. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management 13*, 2 (2005), 197–217.

[16] WOLSKI, R., BREVIK, J., PLANK, J., AND BRYAN, T. Grid resource allocation and control using computational economies. In *Grid Computing: Making the Global Infrastructure a Reality* (2003).