# Implementation of a Perfectly Secure Distributed Computing System

Rishi Kacker and Matt Pauker

Stanford University
{rkacker,mpauker}@cs.stanford.edu

**Abstract.** The increased interest in financially-driven distributed computations has necessitated greater scrutiny of the security and optimality of these distributions. We examine the real-world benefits of one proposed security solution by developing a simulation of a distributed computation that implements an algorithm to redundantly compute individual tasks, an algorithm to choose participants for task computation, and an algorithm to pay participants for successfully completing their work. We analyze the results of the simulations and describe the advantages the framework offers to commercial distributed computations.

**Keywords:** Distributed computing.

## 1 Introduction

The need to solve computationally intensive problems has resulted in significant research and investment in the field of distributed computing. The increased presence of PC's in homes and offices, coupled with an explosive growth in computing power, offers an untapped resource for these computations. And with early successes in the field, such as the Search for Extra-Terrestrial Intelligence project (SETI@home) [xxx], commercial interest in distributed computing has grown. While SETI@home and other initial computations could only offer participants personal satisfaction, projects with financially valuable computations have increasingly seen the opportunity to attract new users through monetary rewards and incentives. This new financial interest has created an industry for corporations who can facilitate the distribution of the computation and the proper payment of participants. However, the involvement of money in distributed computations brings with it an increased incentive for participant cheating. Thus, an important goal for these entities is ensuring security while maximizing computation and minimizing costs.

In [xxx], Golle and Stubblebine suggested two major security schemes for distributed computations that both ensured that non-risk-seeking participants would never cheat and provided increased efficiency for the overall computation. The first defined a dynamic method of distributing tasks dependent on trust, and the second developed an algorithm for assigning these tasks. Left unknown by the paper were three questions: the performance gains of these schemes, the value of a particular variable (elasticity) needed for the second scheme, and the real-world applicability of this secure framework. To examine these questions, we developed a simulation of a distributed computation, attempting to model the secure framework proposed by Golle in a realistic environment.

The rest of the paper is organized as follows. In the rest of this section, we provide a description of the security framework and needed definitions. Sections 2 concerns the simulation, detailing design decisions and implementation. In Section 3, we offer collected data, results, and observations. In Section 4, we provide a summary of the results, significance, and future work.

### 1.1 Review of Golle-Stubblebine Security Framework

We begin with the basic definitions of the framework, which we have extended from the original definitions. There are four main entities: the supervisor, the client, the registration, and the task.

The client represents a single identity (e.g. an individual or a corporation) which controls a number of registrations. Each registration represents a single computer or processor unit. The client's "personality" controls the behavior of his registrations, such as the propensity to withdraw his earnings. The original paper did not make a clear distinction between the client and the registration, using instead the term participant; the simulation, however, required that such a distinction be made.

To participate in the computer, a client must establish registrations with the supervisor. The supervisor divides large computations into small tasks, or work units, each of which is distributed to one or more registrations. In addition, he determines the accuracy of returned results and accordingly provides payment. Note that the the supervisor cannot associate a registration with a client, and that his only leverage is withholding payment from a registration.

We use terms defined in the original paper to describe the simulation:

- **Trust.** A measure of the supervisor's confidence in the unwillingness of a registration to cheat on a given task; it is directly proportional to the amount of money that would be lost by being caught cheating.
- **Redundant Computation.** The process of distributing one task to multiple registrations for the purpose of deterring cheating and verifying correctness of results.
- **Computational Overhead.** The percentage of extra task assignments made using the redundant computation scheme. For example, if 100 tasks need to be computed, and a total of 150 registrations are asked to compute these tasks, the computational overhead is 50%.
- **Activity.** A measure of the confidence of the supervisor in the stated computational power of a registration. A registration with a small activity is believed to be able to only process a few tasks, while a registration with a large activity is thought to be able to process many tasks. Activity is dependent on a value known as **"elasticity"**, which determines how quickly a registration's activity can change over the course of the distribution.

– **Deposit.** The amount of money a registration would have earned for the free tasks it completed to initially enter a computation (as required by the framework – see below).

The framework defines four protocols that describe the process of distributing computations under perfect security. The first establishes a method for bringing new registrations into the computation, requiring new registrations to process a fixed number of tasks for free to build their initial trust. The second offers algorithms that determine the assignment of tasks to registrations. The third enumerates an algorithm for payment to registrations for successfully completing tasks. The fourth provides a means for registrations to leave the computation. Of these, we are primarily interested in the algorithms of the second and the third protocols.

The second protocol defines two main algorithms. The first, given a task $T$, determines the number of registrations which will redundantly compute $T$. Two major methods for this process are given. The first assigns $T$ to a fixed number of registrations, and its computational overhead is an easily determined known value. The second method involves a dynamic distribution scheme, in which a task $T$, assigned to $n$ registrations, is assigned to an $n+1$th registration dependent on the supervisor's trust of the $n$ registrations. The computational overhead of this scheme has not been computed absolutely, but our simulation attempts to provide an accurate real-world approximation.

The protocol's second algorithm is responsible for selecting a single registration from the registration pool. Again, there are two main methods for choosing a registration. The basic method simply chooses a registration at random, and is not concerned with optimizing the distribution. The more involved method picks registrations probabilistically based on their activity. The benefits of this method were not previously determined, and our simulation accurately evaluates its validity.

For the third protocol, the original paper defined the payment algorithm quite simply: each registration was paid a fixed amount for accurately computing a task. We have extended this system to implement dynamic payment, dependent on the trust of the registration. Our simulation demonstrates the usefulness of this dynamic scheme.

In our implementation section, we provide details of all three of these algorithms, including modifications that were made to the originals.

## 2 Implementation

### 2.1 Basic Structure

The simulation was developed in Java and consisted of the four parts described above (supervisor, client, registration, and task), along with a simulator object responsible for generating tasks to provide to the supervisor for distribution, maintaining statistics of the simulation, and generally orchestrating the process. The simulator divides the overall simulation into individual time periods, which could be thought of as representing real-world days. The following flow-chart illustrates the sequence of events for each object in one time period:

In each time period, the following sequence of events is executed:

1. (Simulator) Registration of new registrants
2. (Simulator) Generation of Tasks
3. (Simulator) Delivery of Tasks to Supervisor
   (a) (Supervisor) Assignment of Tasks
4. (Simulator) Running of Supervisor
   (a) (Supervisor) Execution of Each Registration
      i. (Registration) Execution of All Tasks Assigned, Dependent on Client Parameters
   (b) (Supervisor) Retrieval and Analysis of Task Results
   (c) (Supervisor) Payment of Registrations
5. (Simulator) Retrieval of Results from Supervisor
6. (Simulator) Generation of Daily Statistics
7. (Client) Withdrawal of Money

In one time period, the simulator generates a fixed number of tasks, which are distributed by the supervisor to individual registrations, each of which are "owned" by a client. Each client will then process as many tasks as possible and return its results to the supervisor at the end of the time period. The supervisor examines the results and distributes payment to those who successfully computed their tasks. The client, based on his personality, will probabilistically withdraw money from the accounts of his registrations.

### 2.2 Parameters

A number of variables determine the behavior and state of registrations:

- **Speed.** The number of tasks the registration can process in one time period.
- **Money Owed.** The amount paid to the registration for successfully processing tasks that has yet to be withdrawn from the account.
- **Withdrawal rate.** The average number of time periods between withdrawals from the registration's account.
- **Activity.** The registration's activity, as defined in section 1.1.

All registrations "owned" by a client will share the same speed and withdrawal rate (though registrations withdraw probabilistically based on that rate and therefore will not all withdraw on the same time period). These values are predetermined before the registrations are entered into the simulation. Activity and money owed are values that continually change throughout the simulation and will vary from registration to registration.

### 2.3 Design Decisions

In creating the simulation, a number of design decisions were made, though in all cases the attempt to was model a real-world computation as closely as possible.

A project requiring distributed computation generally produces tasks in two ways. Some projects, like Seti@HOME, have a fixed amount of data arriving each time period, and thus distribute the same number of tasks each day, regardless of the number of registrations. Other projects, such as GIMPS, which used distributed computing to find prime numbers, can immediately generate tasks for the entire computation; these types of computations can thus distribute a variable number of tasks each time period, dependent on the number of registrations. As most current distributed computations fall into the first category, we chose to distribute a fixed $T$ tasks each time period, as though only a certain amount of data were available to process. Each simulation was projected to run over $N$ time periods, and thus there would be $T * N$ total tasks for computation. Note that due to redundant computation and the occasional over-assignment of tasks to clients (too many tasks to process), simulations would normally run slightly longer than $N$ time periods.

In keeping with the original paper, we made the assumption that all registrations are risk-neutral or risk-averse, and thus will not cheat when the conditions of perfect security (as defined by Golle) are met. We do not attempt to model an environment in which clients arbitrarily cheat for non-financial motivations. However, to illustrate the importance of choosing an appropriate value of elasticity, we do address the situation in which a client attempts to maliciously disrupt a computation through controlling a large percentage of the overall registrations (see Section 3.2). Further, we assume that there will not be computer computation or network errors resulting in incorrect results from well-meaning clients. Such problems would likely occur infrequently or would be known conditions.

Although perhaps an oversimplification, we assume that clients will always respond to task processing requests and will attempt to process as many tasks as their computing power allows. This excludes problems resulting from computers crashing or participants temporarily suspending their participation in the project. We do not believe that such behavior would dramatically affect the quality of the results.

## 3 Results and Analysis

We present the results of simulations using each of the three schemes, describing for each the assumptions made and the environment used.

### 3.1 Dynamic Distribution Scheme

To examine the dynamic distribution scheme, we assume participants with constant computing power (consistent with the paper) and use a constant payment mode. Because the primary variable in the dynamic distribution scheme is a registration's trust, which is directly dependent on withdrawal rate, we created a registration population which varied over its average withdrawal varied, while keeping all other properties in ceteris paribus. Each registration was owned by an individual client, and thus no single client had control over a large percentage of the population. Withdrawal rate was varied from 1 time period (clients that would withdraw every day) to 365 time periods (clients that would only withdraw once a year).

Two other variables impact the performance of the dynamic distribution scheme. First is the ratio of tasks distributed per time period to the number of registrations (the "task ratio"), and second is the incentive to cheat, or the $\frac{e}{d}$ ratio, which Golle defined as the ratio of the amount a registration would be given to cheat on a given task to the deposit. A result set is therefore defined as the computational overheads calculated by running the simulation over all 365 populations, for a set $\frac{e}{d}$ and task ratio. We provide a graph of two such result sets in Figure 1.

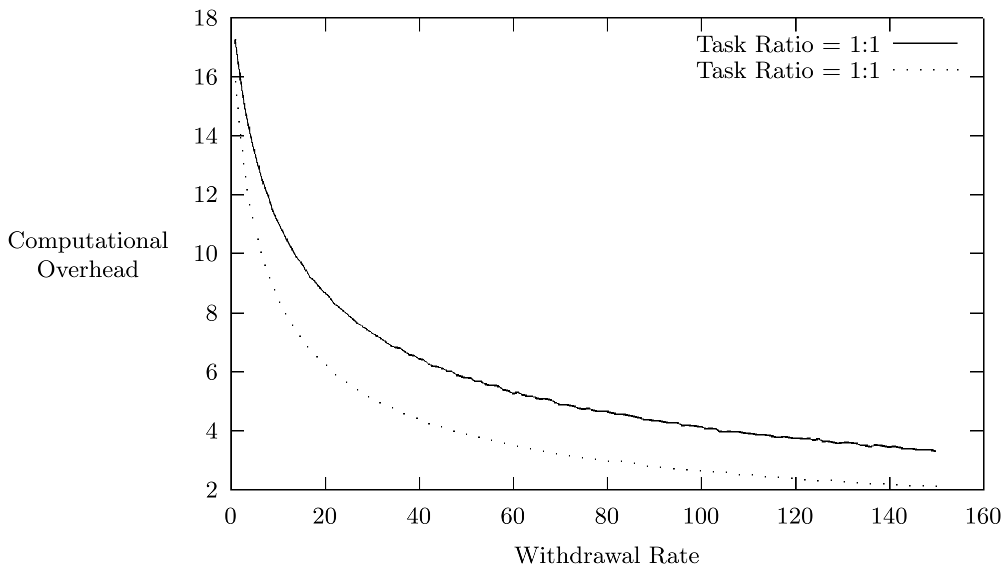


**Fig. 1.** Effect of withdrawal rate on computational overhead with constant $\frac{e}{d}$ ratio.

The graph illustrates the inverse relationship between withdrawal rate and computational overhead. In populations where clients rarely withdraw from their accounts, and thus have a great deal to lose from cheating, their trust level is much greater. The need to redundantly assign tasks is diminished, and we therefore see a reduction in computational overhead. Note that in the two shown results sets, $\frac{e}{d}$ is set, while the task ratio varies. With a greater task ratio, more tasks are available to each registration, who can thus more quickly accumulate money in their accounts and more rapidly develop trust. With the higher task ratio, we therefore see an absolute shift in the curve towards lower overhead at each point.

Figure 2 presents the original two result sets and two new ones which vary over the $\frac{e}{d}$ ratio. Note that the X-axis of the graph was shortened for clarity.

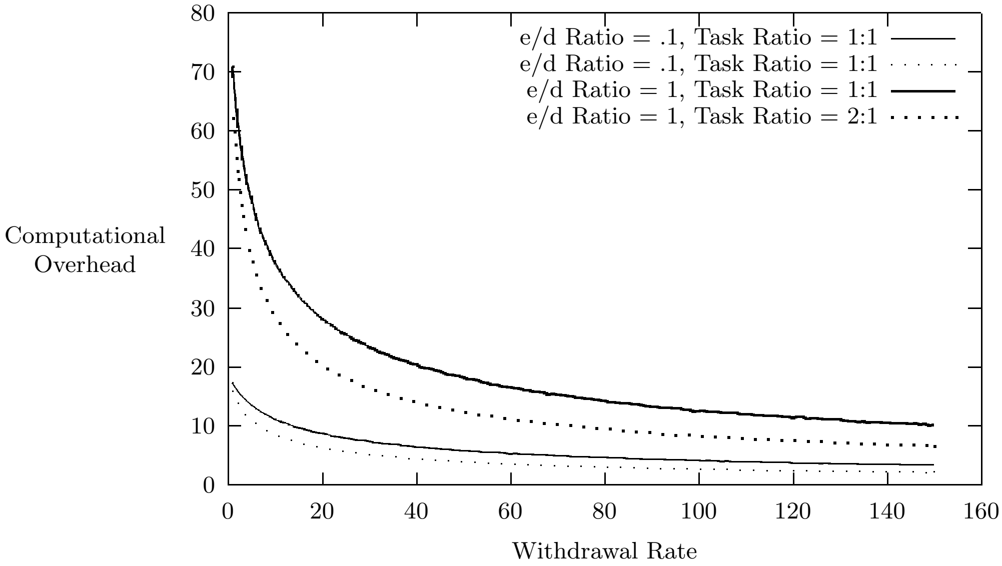


**Fig. 2.** Effect of withdrawal rate on computation overhead, varying $\frac{e}{d}$ ratio.

The graph demonstrates that the $\frac{e}{d}$ ratio, a partly exogenous factor, can significantly affect the performance of the dynamic distribution scheme. While the supervisor has control over the value of $d$, the importance of a specific project may increase the value of $e$, over which the supervisor has no leverage. To maintain perfect security with a larger $\frac{e}{d}$ ratio requires a much higher level of redundancy. Thus, in a real-world setting, the owner of a distributed computation using a dynamic distribution scheme would benefit from doing three things: first, paying out rewards only weekly or monthly, since decreased overhead results in both decreased costs and decreased time to complete the project; second, increasing the number of tasks available per registration (up to the point of saturation); and third, ensuring that the deposit sufficiently outweighs a potential cheater's gain.

### 3.2 Activity Scheme

For our examination of the activity scheme, the simulation was set to use a constant computational overhead of 17%, which Golle shows allows for perfect security given an $\frac{e}{d}$ ratio of .1 and a maximum coalition size of 10% (i.e. one client may control up to 10% of the total number of registrations). Rather than using the task ratio as defined earlier, we instead define a "task power ratio", which relates the number of tasks available per time period to the total computing power available for one time period. For our activity simulations, a task power ratio of 2:1 was used for all data. We immediately discovered that Golle's original equation for elasticity failed, as it allowed for negative values of elasticity (which is impermissible, as elasticity values are used as a probability distribution). Instead, a new elasticity equation was developed:

$$A_{t+1}(J) = A_t(J) * (\frac{\tilde{n}_t + 1}{n_t(J) + 1})^e$$

Where $A_t(J)$ is the activity of registration $J$ at time $t$, $n_t(J)$ is the number of tasks left uncompleted by $J$ at time $t$, $\tilde{n}_t$ is the average over all registrations of $n_t(J)$, and $e$ is the value of elasticity. Activity values are normalized across all registrations, such that $\sum A_t(J) = 1$. The major property of Golle's original equation holds true with ours: as elasticity increases, activity is more sensitive to change, thus allowing registrations to achieve their maximum activity more rapidly;

Selecting an appropriate value of elasticity is a game of balancing the ability of a well-meaning client to achieve their maximum computational potential against the ability of a malicious client to achieve their maximum potential and disrupt a computation. Thus, to examine the effects of elasticity, we created a population where 90% of the computational power was held by all but one of the clients, with a single client (the potential hijacker) maintaining 10% of the power (the maximum coalition size). We used two different measurements to determine the effectiveness of a given elasticity: the number of time periods until all tasks were completed and the number of time periods required for the hijacker to achieve maximum potential. The simulation was then run over a large range of elasticity values.

Although Golle had postulated that effective values of elasticity might range between 0.0 and 1.0, we discovered that, for our simulation, the range of acceptable elasticity values fell in a much more narrow range of 0.0 - 0.04. Beyond this range, the hijacker was able to achieve his maximum potential within only a few time periods, effectively nullifying any advantages of the activity scheme.

Figures 3 and 4 show the graphs of elasticity against the two recorded measurements.

Figure 3, showing time to completion of all tasks against elasticity, demonstrates the effect of higher elasticities on overall productivity. At small elasticities, registrations do not achieve maximum potential very quickly, and thus reduce the total efficiency of the system. However, beyond a certain elasticity, the effective decrease in time to completion asymptotically approaches zero. The

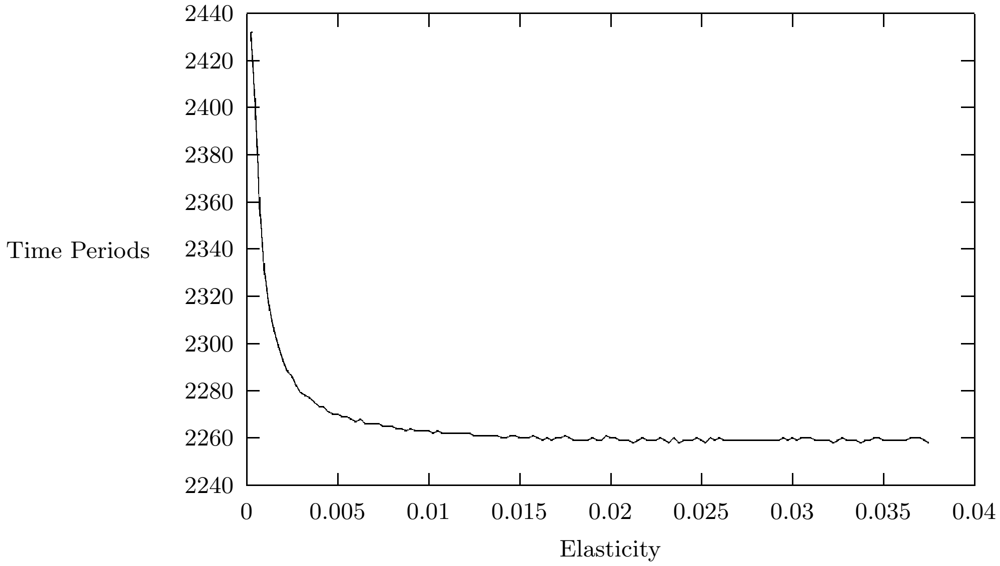


**Fig. 3.** Effect of elasticity on total time needed to complete all tasks.

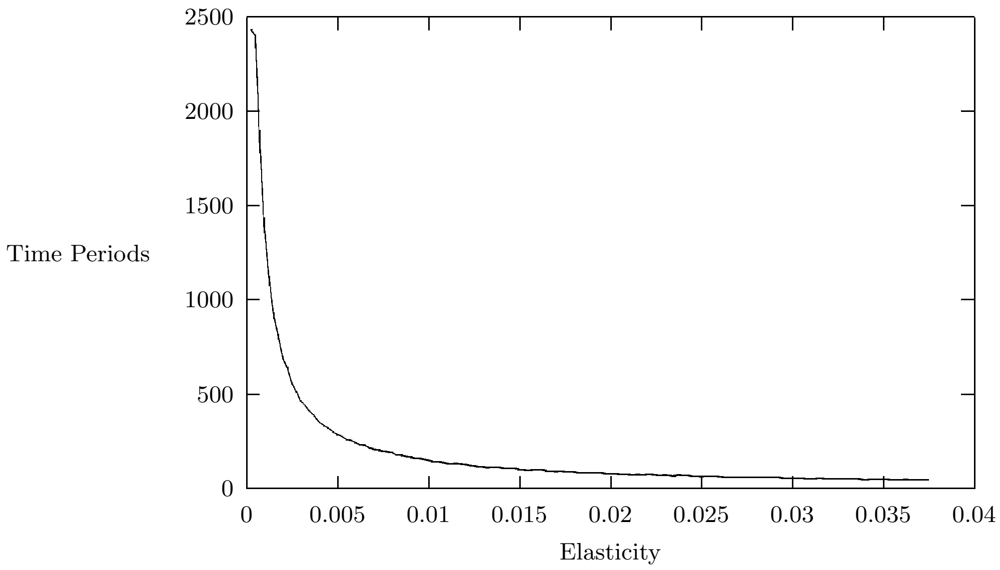


**Fig. 4.** Effect of elasticity on time needed for hijacker to reach full potential.

time difference observed between low and high elasticities is approximately 10%; while not a great savings, decreasing the overall time might be of great benefit to computation owners under time pressures.

Figure 4 shows the effect of elasticity on the ability of the hijacker to achieve maximum potential and receive 10% of the available tasks. Small values of elasticity prevent the hijacker's activity from changing rapidly, and thus limit his [ability] to corrupt a computation.

Elasticity is inversely proportional to both of the described measures. In a real-world application of the framework, the owner of the computation must weigh the benefit of saving time against the danger of potentially allowing a hijacker to disrupt a project. There is not necessarily an optimal value of elasticity; it is dependent on the needs of a given project. Computations needing a high level of security would likely choose a lower elasticity, while urgent projects would benefit from the increased speed of a distribution using a higher elasticity.

### 3.3 Dynamic Payment Scheme

As the dynamic payment scheme was not detailed in the original paper, we first provide the equation used to implement the system. Let $a(J)$ be the amount paid to registration $J$ for successfully computing a task and $a_0$ be a constant representing the maximum payment. We write $d'$ for the initial deposit given by $J$ and $d$ for the amount of money owed to $J$. $a(J)$ then is defined as:

$$a(J) = a_0 * [1 - (\frac{d'}{d + 1.25d'})^3]$$

Thus, registrants with more money in their accounts will receive more money for completing tasks than those who have recently withdrawn their earnings. Figure 5 illustrates the effects of withdrawal rate on the average amount paid per task over the entire distribution. We assume $a_0 = 1$ and $d' = 10$, and use a task ratio of 2:1, an $\frac{e}{d}$ ratio of .1, and a constant computational overhead of 17%.

Note that at high withdrawal rates, registrations accumulate less money in their accounts and receive less money per task completed. At low withdrawal rates, registrations receive increasingly higher returns for computed tasks, and we see an increase in average cost.

The dynamic payment scheme therefore provides two significant benefits. First, it provides a psychological advantage: while clients believe they will be rewarded up to $a_0$ for completing tasks, they will only receive, on average, a fraction of $a_0$ for each task computed. If originally the owner of a computation were paying a constant \$1 for each task computed, the dynamic pricing scheme would allow him to maintain that average task payment while offering an $a_0 >$ \$1. This could provide an enticement to clients, who might see the new system as a better "value". The second major benefit of this scheme is the incentive it creates against withdrawal. A client clearly stands to lose a fixed amount by withdrawing money from his accounts, which provides the supervisor with added leverage over the relationship. Thus, the dynamic payment scheme

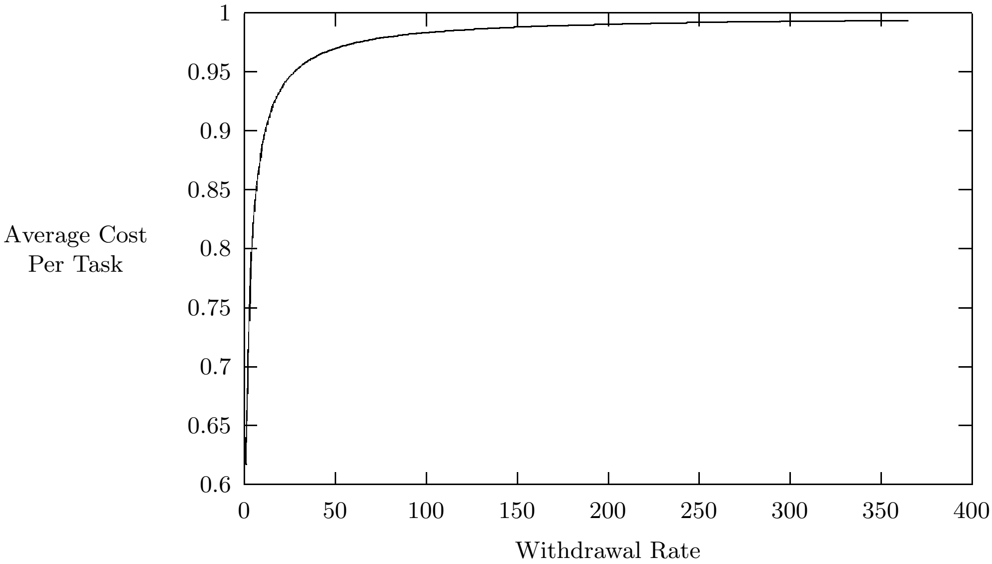


**Fig. 5.** Effect of withdrawal rate on average cost per task.

could be combined with a dynamic distribution system to significantly improve computational overhead. As noted earlier, the dynamic distribution scheme is heavily dependent on withdrawal rates; using dynamic payment could decrease those rates and in turn decrease computational overhead.

## 4 Conclusion and Further Work

We provide details of an implementation of a security framework for distributed computing and offer an improved algorithm for calculating activity and a function for determining dynamic prices. We present results for three of the framework's schemes, attempting to show the real-world benefits they offer.

More optimal functions for calculating activity and pricing may exist, and would probably depend heavily on the makeup of the registrant population and the nature of the computation. Most real-world distributed computations would likely benefit from formulas tailor-made to their situations.

The exact effects of the dynamic pricing scheme on the withdrawal rates of clients (and therefore on the dynamic distribution scheme) are not known, and would require a thorough economic analysis to determine.

## Acknowledgements

The authors would like to thank Philippe Golle and Ilya Mironov for their guidance and extensive support.