# Result Checking in Global Computing Systems

Cécile Germain

*Laboratoire de Recherche en Informatique*
*Laboratoire de l'Accélérateur Linéaire*
*CNRS - Université Paris-Sud*
*cecile.germain@lri.fr*

## Abstract

*Global Computing is a particular modality of Grid Computing targeting massive parallelism, Internet computing and cycle-stealing. This new computing infrastructure has been shown to be exposed to a new type of attacks, where authentication is not relevant, network security techniques are not sufficient, and result-checking algorithms may be unavailable. The behavior of a Global Computing System, which is likely to be bimodal, nevertheless offers an opportunity for a probabilistic verification process that is efficient in the most frequent cases, and degrades gracefully as the problem becomes more difficult. For the two cases of a system based on anonymous volunteers, and a better controlled system, we propose probabilistic tests which self-adapt to the behavior of the computing entities.*

## 1 Introduction

Global Computing is a particular modality of Grid Computing targeting massive parallelism, Internet computing and cycle-stealing. The distinctive feature of Global Computing Systems (GCS) is to harvest the idle time of Internet connected computers, which may be widely distributed across the world, to run very large and distributed applications [1]. All the computing power is provided by volunteer computers, the *workers*, which offer some of their idle time to execute a piece of the application. The actual design of GCS may vary widely, from a fully master-slave architecture like SETI@home [2], to the decentralized style of peer-to-peer systems, which are mostly exemplified in the field of data storage [3, 4, 5].

An active research area is the interoperability between GCS and classical Grid systems in the Globus style [6]. Protocols like JXTA or OGSA [7] offer some opportunities to merge both approaches, through the concept of soft-state registration [8], to allow for impermanence of resources. Classical Grid systems and GCS are opposite in another aspect, which is the subject of this paper: the reliability of the computations. GCS use uncontrolled computing resources, and are thus exposed to *sabotage*: some of the workers may tamper with the computation process, so that they report erroneous results. Classical Grid systems have to protect only their network transactions, for which well-known techniques do exist, because the computers are assumed to be reliable. GCS cannot make this assumption, and this is not only a problem of theoretical interest: some SETI volunteers actually faked their results, by the use of a code different from the original one [9]. While the SETI@home project could afford to replay each computation many times, more efficient techniques can be considered for computations that are to some extent fault-tolerant.

Two strategies can be considered against sabotage: make it difficult, through code encryption, or check the results. In this paper, we consider the second solution, in the framework of probabilistic testing. Our goal is to define a probabilistic testing process that 1) ensures that a set of results is indeed correct, 2) do not unduly eliminate results which are actually correct, 3) keep the test cost low. The first requirement is obvious from the users point of view. The second one is important from the point of view of the GCS efficiency, and has been overlooked in some previous work (see related work). As there is a tradeoff between confidence and cost, the user and the GCS operator must be able to tune the test parameters to meet their quality requirements.

In this paper, we show that *sequential analysis* [10, 11], a technique widely used in statistical testing, is especially appropriate to realistic models of GCS. The main advantage of sequential analysis is that it is adaptive: it implements a quantitative assessment of the concept of *system reliability*, which appears in various GCS frameworks like JXTA or Bayanihan [12], by adapting the cost to the behavior of the system.

The rest of the paper is organized as follows. The second part discusses the features of GCS related to

error tolerance, and defines the entities which will be subject to testing. The next section defines the test model, and the features of GCS related to testing. The fourth section discusses sequential analysis. In section 5 this analysis is applied to the worst case of GCS, where workers cannot be identified, while the opportunities offered by blacklisting saboteurs are discussed in section 6, along with the motivations to consider only a weak form of blacklisting. The next section discusses related work, from the active area of program and property testing, and from spot-checking in GCS research, and we finally propose some conclusions.

# 2 Global computing and error tolerance

## 2.1 Computations

A computation is a very large set of computationally independent *jobs*: jobs do not communicate, nor have data dependencies (eg through files [13]). A GC service, the *Dispatcher*, commits jobs to execution following the availability of workers, marshals their execution, and gets back the results of each job. The Dispatch service is also responsible for assessing the correctness of the results, rerunning a job if its result is considered false, and eventually delivering the result to the client.

In the framework of statistical testing, we define a *batch* as the unit for testing purposes. All jobs of a batch have to wait for the quality assessment of the batch before being delivered (if the outcome is positive) to the client. The quality assessment test is a probabilistic procedure which takes as input a batch, and issues a binary answer, either ACCEPT, or REJECT, with high probability of being correct with respect to the quality criteria.

In some cases, despite this formal independency, some collection of jobs may make sense at the application level, because they contribute to the computation of a result; for instance in a Monte-Carlo simulation, to the computation of statistical quantities. To deliver useful results, batches must respect this organization. For other applications, the computation has no particular structure besides its basic units, the jobs, and the boundaries of batches are arbitrary.

## 2.2 Sabotage

Sabotage can have two purposes. The first one is to modify the result of the computation, whatever may be the origin of this practice (e.g. actually changing the result, or getting a best rank in a hall of fame, or simply an erroneous manipulation). The second one, which has less been considered, is a kind of denial of service: the attacker has no particular interest in the computation, but wants to make suspicious all workers it can reach, so as to slowdown the whole GC system.

Sabotage can be modeled in two ways. In the first one, workers are permanent entities, which can be honest or not (saboteur). Then, the objective of the test should be to detect saboteurs, and to decide some action against them. Possible actions are to remove the saboteur from the worker pool, or to try to correct its behavior. However, workers cannot always been categorized so simply: remotely identifying a computer directly connected to the internet is possible even with dynamic IP addresses by the use of network card identifiers; workers that connect trough ISP cannot give such meaningful identifiers, and anonymity is currently quite easy to achieve (for a more detailed discussion of this issue, see [12]). Hence, it is necessary to design tests that can also deal only with sets of jobs.

## 2.3 Error-tolerance

To allow non perfectly guaranteed results, the application must be error-tolerant to some extent. More precisely, the application can accept a fraction $p_a$ of erroneous results. If the number of jobs is large, the distribution of defects can be modelled as a Bernoulli distribution (0 = correct, 1 = defect), with unknown probability $p$ of being defective.

# 3 Testing for global computing

## 3.1 Test definition

Our goal is to define a test $\mathcal{T}$ which decides if a batch has an error rate $p$ less than $p_0$, or is "far" from this property, that is $p > p_1$ with $p_1 > p_0$. More precisely:

*Given the thresholds $p_0$ and $p_1$, and the confidence parameters $\alpha$ and $\beta$*

*if $p \leq p_0$, $\mathcal{T}$ accepts with probability greater than $1-\alpha$;*

*if $p \geq p_1$, $\mathcal{T}$ rejects with probability greater than $1-\beta$;*

*$\mathcal{T}$ is given access to a job checker (an oracle), which states if the result of this particular job is exact or defective.*

*The* **cost** *of $\mathcal{T}$ is the number or queries to the oracle.*

Such a tester provides only partial information: when the actual error rate $p$ is between $p_0$ and $p_1$, there is no guarantee on the behavior of the tester. However, from the definition of the test process, the probability of accepting a batch under $p$ may be computed, and in some cases be useful.

In the terminology of statistical testing, $\alpha$ is the risk of the first kind(false alarm): the probability of rejecting a batch when the error rate is less than $p_0$ ; $\beta$ is the

risk of the second kind (false negative): the probability of accepting a batch when the error rate is greater than $p_1$.

## 3.2 Checking Jobs

The cost of checking an individual job (an oracle query) raises two issues: the checking algorithm, and the supporting machines. The choice for the checking algorithm is application-dependent. The most costly one, but always applicable, is to replay the job; for many applications, program-checking algorithms [14] provide a convenient alternative. In this case, the checking process is itself probabilistic, so that the outcome of the check can also be erroneous (with low probability). Uncertainty can also appear even with job replay: except if both environments are exactly identical, the results can disagree, because of different hardware and software. The second issue is where to run the checking program. In this paper, we assume that reliable machines will be used for this purpose. We will show below how reliable machines can, in some cases, be discovered in the worker pool itself.

## 3.3 Adaptation

Global Computing exhibits some distinctive features that must exploited to design an efficient test.

- We expect the vast majority of workers to be honest. Thus, it is of primary importance that the test exhibits a low cost for honest workers, or for batches with a low defect rate.
- The second most represented category of workers (after honest workers) will be naive saboteurs, which always fake the results. Thus, the test should also be fast for this class of saboteurs.
- Errors must no lead to complete collapse of the system. Thus, the test must not reject moderately erroneous workers.

The two first points show the need for an adaptive strategy: the test should self-adapt to the information previously got on workers or batches. Results coming from suspicious workers (or batches) should be tested more carefully than those computed by more reliable workers (or batches).

The third point requires a more careful analysis. Two scenarios can lead to a moderate and persistent error rate. The first one is probabilistic job checking, or job replay in a different environment, leading to a non-null rate of job rejection while the job is actually correct. The second one is the denial of service kind of attack. Assume that an adversary of the GC system is able to contribute by a significant number of workers.

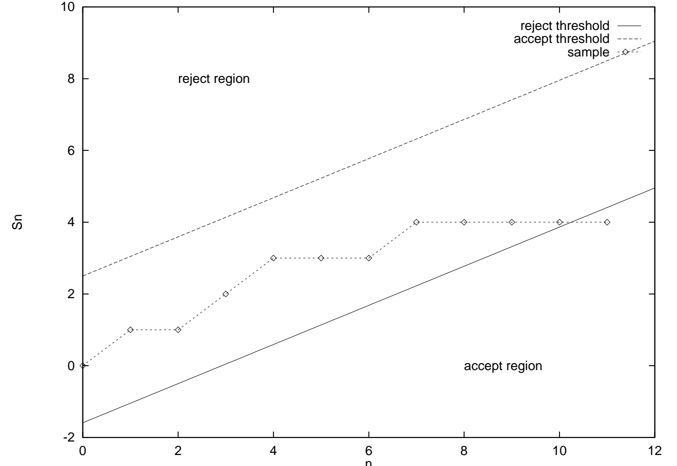


Figure 1: Sequential testing as a Random Walk

If these saboteurs are naive, they will be eliminated at once, and the system will continue to function with as many honest workers it is able to gather; thus, to be efficient, the clever adversary will try to remain in the system, which is possible only if it fakes only a few results. The GCS can thus use the results of the adversary, if the error-tolerance criteria are met, that is, the overall error rate remains low enough. However, working with clever adversaries has a cost, as the next sections will show; detecting and coping with such an attack is considered in section 7.

# 4 The Sequential Test Procedure

Let $p_0, p_1, \alpha, \beta$ be the parameters of the test. Classical tests do not provide any flexibility: the sample size is statically determined from the $(p_0, p_1, \alpha, \beta)$ parameters. On the contrary, sequential testing [10, 11] is precisely based on the intuitive idea that a partial test may be enough to decide, if the data strongly points to the acceptance or rejection decision. Formally, it works as follows: for a sample $x = \{(x_1, \ldots, x_n) \in \{0,1\}^n$, let the *likehood ratio* $l(x)$ be defined by $l(x) = P_{p_1}(x)/P_{p_0}(x)$ (as in classical tests). Two values $A$ and $B$ ( $B < A$) are chosen; at the $m$th step, if $l(x_1, \ldots, x_m) \leq B$, accept; if $l(x_1, \ldots, x_m) \geq A$, reject; otherwise ($B < l(x_1, \ldots, x_m) < A$), draw a new item $x_{m+1}$ and test on $(x_1, \ldots, x_{m+1})$. $A$ and $B$ can be respectively approximated by $(1-\beta)/\alpha$ and $\beta/(1-\alpha)$. In the Bernoulli case, the test can be conveniently expressed through $S_m$. From $P_p(x) = p^{S_m}(1-p)^{m-S_m}$, and the approximation for $A$ and $B$, the rejection re-

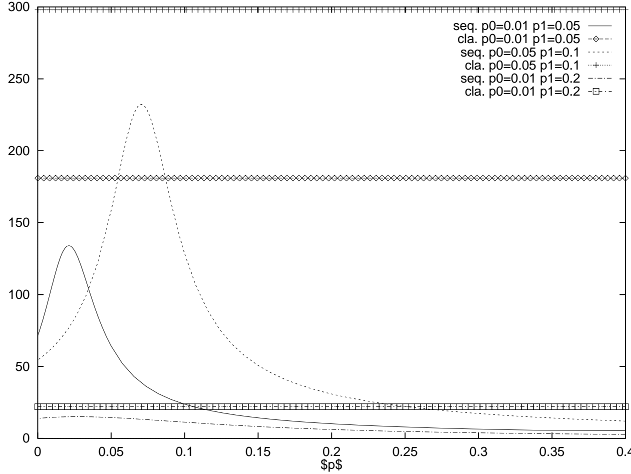


Figure 2: The average sample size

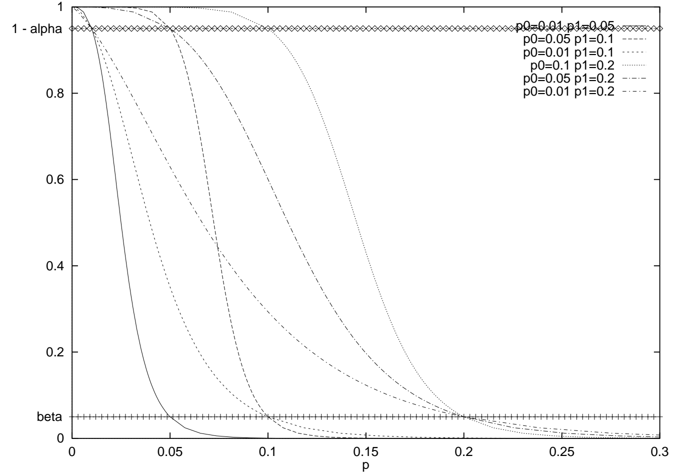


Figure 3: Probability of acceptance

gion is above the line

$$S_m = \frac{\log \frac{\beta}{1-\alpha} + m \log \frac{1-p_0}{1-p_1}}{\log \frac{p_1}{p_0} - \log \frac{1-p_1}{1-p_0}},$$

and the acceptance region is below the line

$$S_m = \frac{\log \frac{1-\beta}{\alpha} + m \log \frac{1-p_0}{1-p_1}}{\log \frac{p_1}{p_0} - \log \frac{1-p_1}{1-p_0}}.$$

Fig. 1 gives a graphical view of the testing process.

The sample size is a itself a random variable, depending on the path defined by the successive samples. Thus, the average sample size $E_p(n)$, as a function of the actual error rate, is the test cost.

In the framework of result-checking, a sequential test has two decisive advantages over a classical one. The first one is that the average sample size is lower than the size of the equivalent classical one. The second and most important one is that the test is self-adaptive. Fig. 2 displays $E_p(n)$, for the sequential test, versus the actual error rate $p$, for various choices of $(p_0, p_1)$ at $\alpha = \beta = 0.05$; the horizontal lines are the corresponding sample sizes for the classical test. The sample size is much lower that the classical one for $p = 0$, goes through a maximum between $p_0$ and $p_1$, and drops sharply for high values of $p$. Closed expressions for $E_p(n)$ are available for $p = 0, p_0, p_1, 1$. In particular,

$$E_0(n) = \frac{\log \frac{\beta}{1-\alpha}}{\log \frac{1-p_1}{1-p_0}}$$

.

Let $L(p)$ be the probability of acceptance at the $p$ error rate. By construction, the test is built so that $L(0) = 1, L(p_0) = 1 - \alpha, L(p_1) = \beta$ and $L(1) = 1$. Fig. 3 displays $L_p$ versus the actual error rate $p$, for various choices of $(p_0, p_1)$ at $\alpha = \beta = 0.05$; between $p_0$ and $p_1$ is the zone where the test may give an erroneous result with a probability higher than what the confidence parameter allow for.

# 5 Batch testing

This section discusses the extreme case where no information is available about the workers. Thus the test has only a batch as a whole, whose quality must be assessed.

The simplest solution is the plain sequential test. The Dispatcher initiative is limited to the choice of the parameters of the test, $p_0, p_1, \alpha$ and $\beta$. The requirement of the end-user are not the same as the ones of a test: the end-user will provide only its error tolerance parameters, that is $p_a$, the maximum error rate, and $\epsilon$, the acceptable risk. To fulfill these requirements,the rejection parameters must be set to the user requirements: $p_1 = p_a$ and $\beta = \epsilon$. On the other hand, from the size of the batch and its own resource availability, the Dispatcher may decide on a desirable amount of resource devoted to individual checks, let say $k$ checks. Hopefully, the batch will be composed of correct works, so that $p$ will be near 0. A reasonable strategy is then to choose $p_0$ such that $E_0(n) = k$. This gives $p_0 = 1 - (1 - p_1)c^{1/k}$, where $c = \frac{1-\alpha}{\beta}$. Fig. 4 displays the values of $p_0$ (in the form of the ratio $p_0/p_1$) versus the values of $k$ for various values of $p_1$, $\alpha = \beta = 0.05$. Very large values of $k$, which are not of practical use, would be necessary to get a small uncertainity region for small $p_0$. Thus, this strategy can be applied only if

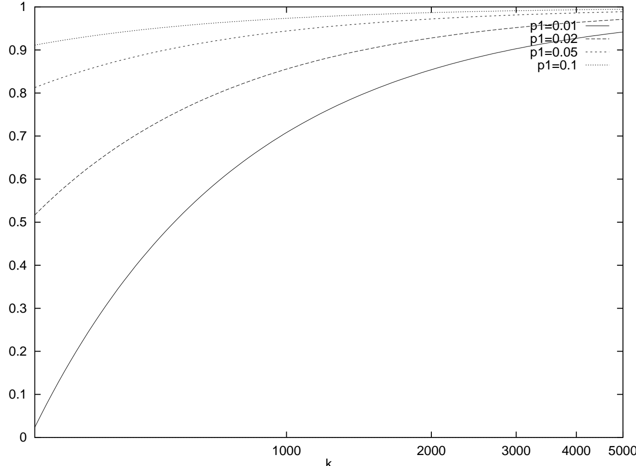


Figure 4: Rejection value at fixed $E_0(n)$

a large uncertainity region is acceptable, which means that the Dispatcher has a strong confidence in the fact that the defect rate is actually null. If it is not the case, the test will reject acceptable batches with a high probability.

To minimize the overhead of failed batches, a two-step process can be considered. If we first run the batch, and run the test only after the batch is complete, when the batch fails, all the failed jobs will contribute to the overhead. Thus, what we seek is a strategy that allows to stop the batch as soon as possible if it appears that the batch will fail. This can be implemented through a test of the earliest results. However, a saboteur can appear at any time, thus the earliest results are not necessarily a sample of the actual error rate. We thus propose a two-steps algorithm. In the first step, only a few jobs, say $N$, are launched, and all are checked; these jobs are randomly chosen in the batch. This test is a sensor of the system reaction to this particular batch. If $N$ is not large enough for the test to complete, new jobs are launched until the test finally completes. If this first test succeeds, the whole batch is launched; when finished, a new test is run on the result set. The batch is accepted only if both tests succeed. If either of the tests fails, it shows that the GC is in serious malfunction, and a corrective action should be undertaken, as described in section 7. After this action, the whole process is restarted.

The choice of $N$ is a tradeoff. Any reasonable choice of $N$ is bounded above by $\mathcal{E} = \text{Sup}_{p\in[0,1]}(E_p(n))$, the maximum over all error rates of the expected sample size. There is no close formula for this maximum, which is attained between $p_0$ and $p_1$. However, if the results are homogeneous, either good (at small $p$) or bad (at large $p$), the actual sample size is smaller than $\mathcal{E}$, and indeed much smaller at large $p$. Hence the sequential test should be computed sequentially, leading to $N = 1$. In this way, as soon as a decision can be reached, the test stops, and the number of redundant computations is kept at its exact minimum (as a function of the precision parameters). On the other hand, in the extreme case of this section, where it is assumed that contributor are fully anonymous, the only strategy available to the Global Computing system scheduler is to launch jobs in parallel: in order to corrupt simultaneously running jobs, a saboteur should get control over many machines. Conversely, if the test jobs are scheduled sequentially, the saboteur could more easily catch and corrupt all or most of them, and lead to an incorrect anticipation of the behavior of the whole batch. As a tradeoff, the *factoring* strategy [15] used in load-balancing can be used for the first test. Initially, $\mathcal{E}/2$ jobs are scheduled in parallel. If the test is not complete, half of the remaining test jobs ($\mathcal{E}/4$) are scheduled at the next step and so on.

One could consider that, as two tests are run on the same set, the $\beta$ parameter could be adjusted : as the two tests are independent, they could be run under $\beta' = \beta^{1/2}$. This is not correct, because only the second test actually assesses the error rate of the whole batch as a result of the contribution of the various and unknown workers. Hence, if the batch is actually a zero-default one, the overall test will use a sample of average size at best $E_0(\alpha, \beta, p_0, p_1)$, doubling the cost with respect to the first algorithm.

Choosing between the two tests is another decision process, that can use an analogous methodology: each successful test is a positive result (a 0), each failed one is a negative result (a 1). Starting with the optimistic hypothesis that batches will be accepted, the first testing procedure is used, until the rejection region is reached. Then the Dispatcher switches to the second procedure. This strategy provides a second level of adaptive behavior, and a more graceful degradation of the system throughput.

# 6 Blacklisting

The name blacklisting has been coined by [12], as a class of algorithms allowing to reject a saboteur. If saboteurs can be identified, there is an evident advantage at eliminating them, at least for the duration of a batch. However, as explained before, we do not want to define a saboteur on a only-one error basis. This section explores a weak form of blacklisting, where we eliminate only the workers which have a too high error rate.

Here also, we propose a two-step method. Each

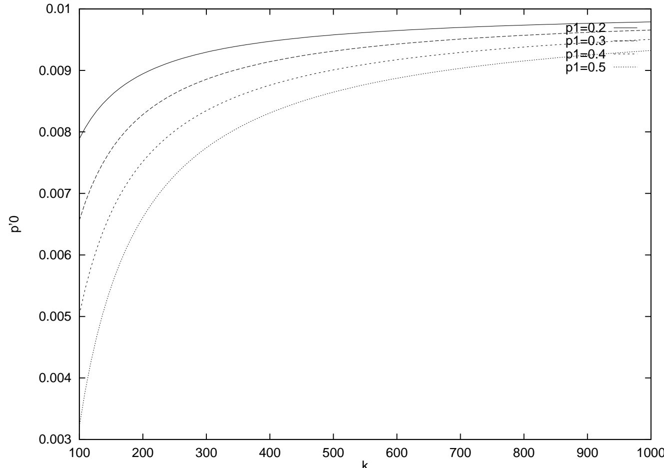


Figure 5: Rejection value for the second step of the blacklisting algorithm

worker takes a piece of a batch $B$; on this piece, a sequential test is run, with $p_0 = p_a$, and a large value of $p_1$. More precisely, the objective of this step is to eliminate the saboteurs that fake a serious amount of results. The exact values of $p_1, \alpha$ and $\beta$ will depend on the size of the piece of work allotted to a worker, as in the previous section. All the jobs performed by a worker which has failed the test are eliminated, and reserved for another batch. This is the reason for the choice of $p_0 = p_a$: we do not want to eliminate pieces that comply with the user requirement, so we tune the test so that it accepts (whp) a set of jobs with an actual defect rate less than $p_0$.

At the end of this first test, the remaining jobs, which come only from the workers that have not been eliminated, form an improved batch $B'$. At this step, to assess the quality of $B'$, we can reuse the results of the jobs tested, thus at no cost, to conduct a second test on $B'$. For this new test, we must set $p_1 = p_a$, and $\beta = \epsilon$. As the naive saboteurs have been eliminated, and because we expect that the naive and honest ones are the overwhelming majority, the most frequent case will be the one where the remaining workers are nearly perfect. Thus, a likely efficient choice for $p_0$ will be the one such that $E_0(n') = kE_0(n)$, where $k$ is the number of remaining workers. Fig. 5 shows the values of $p_0$ as a function of $k$, assuming that $\alpha' = \alpha$ and $\beta' = \beta$, for $p_a = 0.01$, and various choices of $p_1$. For $p_1 = 0.2$, which gives a sample size on the order of 10 for each worker, it suffices to keep slightly more than 200 workers to get a final test which is subject to err only for $0.009 < p < 0.01$.

An optimization of this scheme would be to allow for the early rejection of the previous section. However, in this case, the amount of work allotted to each worker may be varying. A worker may have passed the test, and be allotted new jobs, which offers an opportunity to new errors. Integrating the two schemes will be the subject of future work.

# 7 Actions

## 7.1 Corrective actions

The simplest corrective action is to force the worker to reload the code. In the XtremWeb framework we are currently developing [21], this is easy: each time a worker asks for work, it gives information about the codes it has previously downloaded, and the dispatcher is responsible for uploading the code of the job it plans to commit to the worker. For a honest worker, uploading is avoided, if the code is already present, as it wastes network and server bandwidth. For a saboteur, the dispatcher will upload the code, and otherwise monitor its results as usual. This is the only possible action when the workers are anonymous.

When workers can be identified, the average size of the test can be used to estimate the error rate of a worker, through Wald first lemma stating that $E_p(S_N) = pE_p(N)$. Dealing with honest workers and naive saboteurs is straightforward. An error rate between the acceptance and rejection threshold is much more penalizing, as it requires a very high number of checks. This behavior strongly points towards a denial of service attack (assuming the job checking algorithm is mostly correct). First, if the saboteur fakes only a few results, it may be expected that the overall computation will not be seriously hampered; hence, its purpose is only to hamper the GC system itself, by increasing the checking overhead. Second, and more important, this attack may show that the saboteur has a precise knowledge of the appropriate parameters of the checks. In this case, if the reloading code strategy does not modify the system behavior, it would become necessary to assume that the security of the dispatcher itself has been compromised.

## 7.2 Reliable workers

At bootstrap, there must be some external guarantee for the workers which are in charge of individual job checking. This could be a serious limitation to the overall throughput of the GCS, which can only deliver a batch when the testing process is complete. In permanent regime, and when workers can be identified, their reliability performance can be traced over time, and tested for zero default. Workers that have a negligible probability of faking the results can be used as a

support for the testing process itself. Then, only these workers have to be checked on the definitely reliable ones. Here also, the system will self-adapt to the workers behavior: the number of workers it can actually use will increase with the number of honest workers.

# 8 Related Work

Checking the correctness of a computation has been the subject of a lot of theoretical work, since the seminal paper of Blum [16]. Two main approaches have been considered. The first one considers the problem of checking the result of a computation on a particular input, through the use of some properties relating the inputs to the outputs [14]. In this paper, we consider result-checking algorithms only at the level of individual job checking, assuming that a batch checker is not available. However, as explained before, the availability of job checkers are not a requirement for our scheme.

The second approach is *property testing* [17], which has a very strong relation to the problem considered here. A property tester states probabilistically whether an object actually possesses a property, or is far for any object possessing it. Thus, a property tester exhibits the same parameters (false alarm and false negative) than a statistical test. In the literature, property tester usually consider non-numeric properties (linearity of functions [18, 19], or graph partition problems such as $k$-colorability [17]) In our framework, the tested property is to have a defect rate lower than a prescribed threshold, and the distance is described by the $p_1$ parameter.

In the framework of the Global Computing projects, the sabotage problem has been considered by the Bayanihan one [12]. Our work shares the idea of using dynamically built credibility to verification process. The main difference are that [12] 1) does not use the framework of statistical tests, and thus does not take into account the risk of false alarm ; 2) is based on blacklisting.

# 9 Conclusion

Global Computing Systems are exposed to a new type of attacks, where authentication is not relevant, and network security techniques are not sufficient. In this paper, we have considered the case where no information is available about the properties of the collective result of a batch, and thus the result-checking algorithms cannot be looked for. We think that this situation may be frequent. The massive trivial parallelism offered by GCS is especially well-suited to Monte-Carlo computations, and indeed GCS have been built for or applied to such applications [20, 21]. In this case, the overall computation result is the probability distribution of natural processes on which little is known, thus only too crude reliability tests can be defined. We have analyzed the likely behavior of a GCS, which is bimodal, and devised probabilistic methods which self-adapt to the behavior of the computing entities.

Future work will go in two directions. The first one is a typical case study, for a Monte-Carlo application in astrophysics, for which the slack in individual result checking is a reality. The second one is to explore the potential of the sequential analysis core idea, that is to build dynamically the sample, in the field of property testing.

# References


[1] C. Germain et al. Global Computing Systems. In *SciCom01*. LNCS 2179. Springer, 2001

[2] D. Anderson and al. A New Major SETI Project Based on Project Serendip Data and 100,000 Personal Computers. In *5th Intl. Conf. on Bioastronomy*, 1997.

[3] G. Kan. Gnutella. In *P2P: Harnessing the power of disruptive technologies. A. Oram Ed.* O'Reilly, 2001.

[4] A. Langley. Freenet. In *P2P: Harnessing the power of disruptive technologies. A. Oram Ed.* O'Reilly, 2001.

[5] David R. Karger and Matthias Ruhl. Finding Nearest Neighbors in Growth-restricted Metrics. In *STOC '02*.

[6] I. Foster. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *IJSA*, 15(3), 2001.

[7] I. Foster, C. Kesselman, J. Nick, S. Tuecke. Grid Services for Distributed System Integration. *Computer*, 35(6), 2002.

[8] S. Raman and S. McCann. A model, Analysis and Protocol Framework for for Soft State-based Communication. *Computer Communication Review*, 29(4). 1999

[9] D. Molnar. The SETI@home problem

[10] A. Wald. *Sequential Analysis*. Wiley Pub. in Math. Stat.1966

[11] D. Siegmund. *Sequential Analysis*. Springer series in Statistics. 1985.

[12] L. Sarmenta Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. *FGCS*, 18(4). 2002

[13] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *9th Heterogeneous Computing Workshop*, 2000.

[14] M. Blum and H. Wasserman. Program result-checking: A theory of testing meets a test of theory. In *35th FOCS*. 1994

[15] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: a method for scheduling parallel loops. *Comm. ACM*, 35(8). 1992.

[16] M. Blum and S. Kanna. Designing programs that check their work. In *21th STOC*. 1989.

[17] O. Goldreich, S. Goldwasser, and D. Ron Property Testing and its Connection to Learning and Approximation. *JACM*, 45(4). 1998

[18] M. A. Kiwi, F. Magniez, M. Santha. Approximate Testing with Relative Error. In *31th STOC*. 1999

[19] F. Ergun et al. Spot-checkers. In *30th STOC*. 1998.

[20] L. Loewe. Evolution@home: Experiences with Work Units that Span More than 7 Orders of Magnitude. In *Workshop on Global and P2P systems at 2nd IEEE CCGrid*. 2002.

[21] G. Fedak et al. XtremWeb: a generic Global Computing System. In *Workshop on Global Computing on personal devices at 1st IEEE CCGrid*