

Injecting Robustness into Autonomic Grid Systems

Yuriy Brun, George Edwards, and Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{ybrun,gedwards,neno}@usc.edu

ABSTRACT

Autonomic computational grids are self-organizing software systems that pool the computational resources of large public networks to solve computationally-intensive problems. While autonomic grids can scale to networks far larger than centralized grids, they have not seen the same adoption and success in industry due to an incomplete treatment of fault tolerance. In this paper, we propose two complementary mechanisms, *progressive redundancy* and *redundant distribution*, that secure autonomic grids by injecting robustness into insecure, possibly malicious networks. Progressive redundancy replicates computation in a way that reduces the probability of failure while minimizing the associated cost. Redundant distribution allows for the replication of computation in decentralized networks. We formally define the class of grid technologies to which progressive redundancy and redundant distribution apply and evaluate the cost and benefit of using the techniques. Progressive redundancy and redundant distribution reduce the probability of system failure exponentially, at a linear cost in the execution speed of the system.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques;

C.4 [Computer Systems Organization]: Performance of Systems

General Terms

Reliability, Security

1. INTRODUCTION

Solving computationally intensive problems is integral to modern research in artificial intelligence [13], physics [32], astrophysics [33], bioinformatics and genomics [5], disease and drug design [6], economics [29], networking [42], neuroscience [12], system biology [44], and a number of other fields. To satisfy this extensive need for fast computation, researchers have turned to *computational grid systems*, which pool hardware and software resources to deliver computation as a standardized service. For example, computational grids can leverage public networks to combine

machines with idle cycles into single-interface distributed computing systems [4, 31, 34]. The organization of loosely coupled networked devices achieved by computational grids has permitted advances in science and engineering that would otherwise not be possible [9, 28, 36].

The design and implementation of grid infrastructures raises new problems in computer science and software engineering with respect to efficient distribution, scalability, security, and robustness of ultra-large-scale systems [19, 20]. In particular, while the Internet presents the possibility of creating grids with millions of machines, today's most successful grid technologies only scale to networks smaller than .001% of the Internet ($\approx 10,000$ machines) because they rely on centralized management and control.

The two main approaches to harnessing large networks for computation — the *master-worker model* [4, 15, 21] and *centralized scheduling* [7, 24, 37] — are both subject to scalability limitations. The master-worker model is constrained by the performance of the master and requires individual computational chunks to be fairly large to deal with the uncertainty of the underlying network. Centralized scheduling cannot scale to Internet-size networks because it is impossible to maintain an up-to-date view of the entire network and because scheduling algorithms require too much overhead for large networks. In other words, these approaches are aimed at, and perform best on, either relatively small networks with reliable nodes [18, 21, 35] or large static networks that can be modeled without much overhead [4, 31, 34].

Autonomic grid technologies present the opportunity for creating grids with 3–4 orders of magnitude more computational resources [14]. Recent proposals for grid technologies remove scalability and machine-reliability restrictions by leveraging biologically-inspired processes and structures to develop dynamic, amorphous distributed software systems that self-organize on a network and self-adapt to changes in the underlying hardware and network infrastructure [11, 14]. An *autonomic computational grid* is a computational grid that distributes highly parallelizable computation in a decentralized manner.

While autonomic grids promise a better infrastructure, they have not seen the same adoption and success in industry as traditional grid technologies [18, 21, 35, 31, 41]. We believe that autonomic grids surpass other grid technologies in several important aspects, but have not been adopted because they rely on the participation of untrusted hardware and network resources but lack the ability to deal with faulty and malicious actors.

In this paper, we propose two techniques, *progressive redundancy* and *redundant distribution*, that work together to make the distribution and execution of computation on grid systems robust. Our approach applies to a large class of grid technologies solving computationally intensive problems, such as all NP-complete prob-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

lems and many others. We develop techniques that leverage smart redundancy to ensure that a wide range of failures (even Byzantine failures of consorting malicious nodes) can be prevented, with high probability. Further, we show that the overhead on the system designer is minimal: one must only estimate the failure rate on the network (e.g., 10%) and specify the desirable certainty of computation (e.g., 99.999%), and the system automatically employs the right amount of redundancy. Finally, we formally argue that a *linear* reduction in computation speed results in an *exponential* decrease in the error rates. For instance, slowing down a system with failure rate of 10% by roughly a factor of 5 results in a probability of failure of roughly $(10\%)^5 = .001\%$.

In Section 2, we formally define the necessary properties of autonomous computational grids and describe examples of such systems, some of which are currently deployed on over a million real-world machines. Then, in Section 3, we present the theory behind progressive redundancy and redundant distribution. Next, in Section 4, we formalize our threat model and analyze the types of faults and attacks our technique can withstand. Finally, in Section 5, we evaluate the costs and benefits of progressive redundancy and redundant distribution on autonomous grids.

2. AUTONOMIC GRIDS

While there are several different types of grid technologies [19, 20], in this paper, we concentrate on *computational grids*, which coordinate federated hardware and software resources to solve complex computational problems. Put simply, these technologies combine the computational power of many distributed computers into a single computing system. Computational grids decompose large computations into smaller *subcomputations* and transparently distribute those subcomputations among grid nodes. The robustness mechanisms we describe in this paper are relevant to one particular type of computational grids: *autonomic computational grids*. In this section, we define the essential features of autonomous grids and describe several implementations that are in use today.

2.1 Definition of Autonomic Grids

A computational grid technology is called *autonomic* if and only if it possesses two key properties:

1. It distributes computation in a *decentralized* manner.
2. It decomposes computation into *predominantly nonblocking* subcomputations.

Each of the above two properties, which we now examine, has significant ramifications on the types of problems that autonomous grids can solve, the way autonomous grids are structured, and the behaviors exhibited by them.

2.1.1 Decentralized Distribution

A distribution module’s job is to assign a set of subcomputations to some nodes on a network. Centralized distribution modules can perform all the decisions regarding which nodes will execute which subcomputations; however, such modules exhibit two problems: they are single points of failure and they present bottlenecks for large distributions.

In decentralized distribution modules, each node only assigns subcomputations to a small constant number of other nodes. These modules may achieve complete distribution by: recursively dividing the subcomputations (e.g., the organic grid [14] and MapReduce [17]); developing iterative subcomputations, such that the node executing a subcomputation then distributes the next, possibly several, subcomputations (e.g., the tile style [11]); or by other means. By removing the central agent, decentralized distribution modules avoid single points of failure and bottlenecks. This property is

crucial to applying redundancy to autonomous grids because redundancy increases the number of subcomputations and thus is susceptible to delays in the presence of distribution bottlenecks.

2.1.2 Nonblocking Computation

Two subcomputations are *nonblocking* if and only if *their order of execution does not affect the results*. In other words, nonblocking subcomputations are trivially parallelizable; for instance, they do not require or expect any interaction in the form of shared data or message exchange.

A set of subcomputations is *predominantly nonblocking* if and only if *each pair of subcomputations within that set, with high probability, is nonblocking*. In the autonomous grids we describe in this paper, “high-probability” will imply that the number of blocking pairs of computations in a set is at most linear in the size of the input, while the number of nonblocking pairs is at least exponential in the size of the input. Our techniques perform best when there is a high probability that each pair is nonblocking, and are thus well-suited for autonomous grids, but they can still be applied to systems with some blocking subcomputations.

2.2 Industrial Autonomic Grid Systems

In this section, we summarize the architectures of four autonomous grids, one of which is currently deployed on over a million nodes on the Internet. While this list of autonomous grids is not complete, we will use these examples throughout the remainder of the paper to illustrate how our redundancy techniques work and how they can be utilized to improve the robustness of autonomous grid systems. In all four examples, our redundancy techniques result in more-efficient fault tolerance than the mechanisms currently in use.

2.2.1 The Organic Grid

The *organic grid* [14] is used to solve easily parallelizable problems, such as NP-complete problems, and problems susceptible to dynamic programming, such as nucleotide-nucleotide alignment and matrix multiplication. The organic grid decomposes computational tasks into subtasks and assigns each subtask to a mobile agent, whose job is to autonomously locate a node with adequate resources to perform the subcomputation.

One *origin* node begins the computation by distributing subtasks to several other nodes that, in turn, divide those subtasks into smaller subtasks, and so on. The resulting structure is a tree that acts as a platform for the autonomous mobile agents to schedule tasks in a decentralized manner without needing global network information.

The organic grid can tolerate nodes in the middle of the tree becoming unresponsive, because each node maintains a small list of its ancestors; if a node fails, its children become reincorporated into the tree structure through their grandparent nodes. Currently, the organic grid offers no fault-tolerance mechanism for incorrect subcomputation results, or even corrupted or lost results due to network communication failures, which we believe is a significant reason for its lack of adoption in industry. However, since the organic grid deals with predominantly nonblocking subcomputations and employs a decentralized scheduler, it is susceptible to our redundancy techniques.

2.2.2 MapReduce

MapReduce [17], another well-known and commercially available autonomous grid technology, is applicable to similar problems and uses a similar decentralized distribution algorithm to those of the organic grid. While there are some notable differences, (in particular, MapReduce does not use mobile agents to find available and

resource-sufficient nodes), for the purposes of our fault-tolerance techniques, we treat MapReduce and the organic grid in the same manner.

In order to use MapReduce, the developer designs two functions: `Map` and `Reduce`. The `Map` function takes a problem and divides it into a small number of subproblems that can be solved in parallel. The `Reduce` function takes the solutions to several subproblems and combines them into a solution to the original problem. The MapReduce infrastructure then handles taking a single problem, distributing it by recursively dividing it into smaller and smaller subproblems, and recursively combining the solutions to the subproblems into the final answer. Commercial uses of MapReduce include computing Google’s PageRank and as many as a thousand MapReduce jobs are executed on Google’s clusters daily [17].

2.2.3 The BOINC Systems

BOINC is an autonomic grid platform currently deployed on over a million computers. Volunteers join the network by running a BOINC client and choosing problems to participate in solving. Entities who wish to submit problems to the network apply for access; if accepted, they send computations to volunteers who have agreed to help with that problem. Examples of BOINC applications include SETI@home, LHC@home, Folding@home, Quantum Monte Carlo at Home, Malariacontrol.net, Climateprediction.net, PrimeGrid, and many others [3, 8].

BOINC applications are autonomous actors responsible for the subdivision, scheduling, and distribution of work. Each application implements a scheduling server that sends subcomputations to nodes that have requested it, while ensuring that each node has sufficient resources to solve the subcomputation and has not already submitted a solution to that subcomputation. BOINC applications decompose computation into entirely independent (completely nonblocking) subcomputations. While every application is autonomous, of the autonomic grids we describe in this paper, the BOINC systems are the most centralized. In some ways, these systems represent one extreme of autonomic grids.

BOINC utilizes the traditional redundancy algorithm (described further in Section 3) to achieve robustness and fault tolerance. Each application specifies the number of times to replicate each subcomputation, the number of results required for a *quorum*, and the minimum number of agreeing results for a *consensus*.

2.2.4 The Tile Style

The tile architectural style [11] is used to solve NP-complete problems, such as 3-SAT, determining the structure of proteins, and optimally allocating resources. The tile style pushes the second property of autonomic grids to the limit. It decomposes NP-complete problems into the smallest possible subcomputations (on the order of complexity of simple 2-input binary gates) and distributes those subcomputations onto a network. The subcomputations are iterative, so the result of each subcomputation is one or more subcomputations that are then distributed onto other nodes. As the nodes perform and distribute the subcomputations, they provide an autonomic decentralized distribution service. Note that the nature of the problems tile style tackles, NP-complete problems, is integral to the decomposition into iterative subcomputations.

Current implementations of the tile style rely on trustworthy nodes and do not employ fault tolerance. Proposals for making the tile style fault and adversary tolerant have argued that the tile style is susceptible to both traditional redundancy and biologically-inspired fault-tolerance techniques [10]. The redundancy techniques we define in this paper build on the existing proposed ideas and are more efficient than the technique proposed in [10].

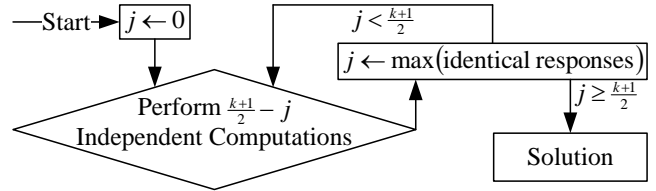


Figure 1: Progressive redundancy technique.

3. FROM REDUNDANCY TO ROBUSTNESS

We have developed two mechanisms to inject robustness into autonomic grid systems. Each mechanism leverages one of the fundamental properties of autonomic grids described in Section 2. The first, *progressive redundancy*, which we discuss in Section 3.2, allows for replication of components in a way that reduces the probability of failure while minimizing the associated cost. Progressive redundancy achieves this benefit by utilizing the predominantly nonblocking nature of autonomic grid computations. The second, *redundant distribution*, which we discuss in Section 3.3, allows for the replication of distributing components in complex, multilevel networks. Redundant distribution takes advantage of the autonomic grids’ decentralized distribution algorithms. We begin, however, in Section 3.1, by stating our assumptions about network nodes.

3.1 Assumptions

In this section, we state a number of assumptions about the nodes of the network deploying an autonomic grid. These assumptions make it easier to explain our techniques and analyze their effects. However, in Section 5.3, we will relax these assumptions and demonstrate that both progressive redundancy and redundant distribution still apply, and, in some cases, perform even better on more general networks.

For the meantime, our assumptions are:

- all nodes have the same probability of failure,
- the probabilities of nodes failing are independent of each other,
- all nodes are equally preferred,
- the result to a computation is a single bit¹ and
- the reliability of the client that receives the final result of the computation is excluded from the system’s reliability.

3.2 Progressive Redundancy

The first technique we propose in this paper, *progressive redundancy*, leverages the predominantly nonblocking nature of subcomputations to minimize the amount of work required to produce a reliable result. To explain and demonstrate how progressive redundancy works, we contrast it with the *k-vote traditional redundancy* technique (sometimes called *k-modular redundancy* [30]). Traditional redundancy performs *k* independent executions (where $k \in \{3, 5, 7, \dots\}$) of the same computation in parallel, and then takes a vote on the correctness of the result. If at least some minimum number of executions agree on a result, a *consensus* exists, and that result is taken to be the solution. To simplify the subsequent discussion, we use $\frac{k+1}{2}$ (i.e., a majority) as the minimum number of matching results required for a consensus, but different

¹The assumption that the result of a computation is a single bit is quite common in fault-tolerance research [30]. Although perhaps counterintuitive, this assumption creates a worst-case scenario because all failing nodes will report not only a wrong answer, but the same wrong answer, thus making it difficult to identify failing nodes. In Section 5.3, we will discuss the implications of relaxing this assumption.

p	k	$f_{TR}^k(p) = f_{PR}^k(p)$	$c_{TR}^k(p)$	$c_{PR}^k(p)$
.1	3	.028	3	2.18
	5	.009	5	3.32
	7	.003	7	4.44
p	k	$\sum_{i=0}^{\frac{k-1}{2}} \binom{k}{i} p^i (1-p)^{k-i}$	k	$\frac{k+1}{2} + \sum_{i=\frac{k+3}{2}}^k \sum_{j=i-\frac{k+1}{2}}^{\frac{k-1}{2}} \binom{i-1}{j} p^j (1-p)^{i-1-j}$

Figure 2: Comparison between the traditional and progressive redundancy techniques’ costs.

minima can readily be applied to the algorithm we describe. Traditional redundancy produces an exponential drop in the failure rate, but requires k times as many resources, such as hardware nodes or CPU cycles, whereas progressive redundancy produces the same drop in the failure rate, but requires fewer resources.

Progressive redundancy minimizes the number of computations needed to produce a consensus. Rather than having k nodes perform the computation, only $\frac{k+1}{2}$ nodes do so. If all these nodes agree, the results produced by any subsequent repetitions of the computation are irrelevant; a consensus already exists. In this case, progressive redundancy does not distribute that particular subcomputation to any additional nodes. If some nodes agree, but not enough to produce a consensus, progressive redundancy distributes the minimum number of additional copies of the subcomputation necessary to produce a consensus, assuming that all these additional executions were to produce the same result. Progressive redundancy repeats this process until a consensus is reached. Figure 1 graphically depicts the progressive redundancy algorithm.

Let us consider an example. Traditional 15-vote redundancy executes the same computation on 15 independent nodes, reports the majority answer, and succeeds whenever at least 8 nodes execute without failure. Progressive 15-vote redundancy first asks only 8 nodes to perform the computation. If all 8 nodes report the same answer, the results of the remaining 7 nodes are irrelevant. If only 5 of the nodes report the same answer, progressive redundancy asks 3 more nodes to perform the computation. If of those 11 nodes only 6 agree on an answer, progressive redundancy then asks another 2 nodes, and so on, until 8 nodes agree on an answer.

Progressive redundancy is more efficient than traditional redundancy in terms of the amount of computation that must be performed, but may require additional time to complete. Today, typical systems that leverage redundancy to detect and correct faults have blocking computations [22, 30]: if these systems experience a fault, they must correct that fault before moving on. Thus, delaying the completion of a subcomputation causes idle nodes and wastes computational power. The key property of autonomic grids that makes them susceptible to progressive redundancy is that their subcomputations are predominantly nonblocking. This means that there is virtually no cost to delaying the completion of an individual subcomputation because all available nodes can be given other subcomputations to execute in the meantime, and no nodes are left idle (as we formally argue in Section 5.1). In other words, progressive redundancy allows dynamic failure detection and recovery: failures are corrected (via additional replication) only when they have occurred. In traditional scenarios with blocking computations, dynamic failure detection incurs a high cost by delaying all future computations whenever nodes disagree. The nonblocking nature of autonomic-grid subcomputations allows the successful use of dy-

namic failure detection.

To compare traditional and progressive redundancy techniques, we are interested in two measures of their effect on systems: the *probability of failure* of replicated subcomputations and the *cost factor* of applying the redundancy technique (how many times slower the system executes), both functions of the probability p of failure of a single component. For k -vote traditional redundancy, we will refer to the probability of failure as $f_{TR}^k(p)$ and the factor cost as $c_{TR}^k(p)$. For k -vote progressive redundancy, we will use $f_{PR}^k(p)$ and $c_{PR}^k(p)$, as appropriate². Note that a commonly used term in fault-tolerance literature, *mean time to failure* for receiving a particular result, is identically the inverse of the failure probability, in terms of the length of the execution of each subcomputation. In other words, the mean time to failure of getting the answer to a subcomputation is $(f_{PR}^k(p))^{-1}$ multiplied by the time necessary to execute an average subcomputation.

The following simple example shows quantitatively how progressive redundancy outperforms traditional redundancy.

With 3-vote traditional redundancy, if each node has the probability .1 of failing, then the probability that the 3-node system fails is $f_{TR}^3(.1) = .1^3 + 3(.1^2) .9 = .028$, which is a big improvement over a single node’s .1 probability of failure. However, the cost of this improvement is a factor of $c_{TR}^3(.1) = 3$ in the amount of computation performed.

With 3-vote progressive redundancy, the probability of failure is again $f_{PR}^3(.1) = .1^2 + 2(.1) .9(.1) = .028$, but the expected cost of this improvement is only a factor of $c_{PR}^3(.1) = 2 + 2(.1) .9 = 2.18$. Of course, whereas traditional redundancy took 1 time unit to complete, progressive redundancy requires 2 time units if the results of the first two nodes disagree.

Figure 2 shows the probability of failure and the expected cost factor for a few example k -vote traditional and progressive redundancy techniques, as well as the general case. We provide a rigorous, formal analysis of the costs and benefits of progressive redundancy in Section 5.

3.3 Redundant Distribution

The second technique we propose in this paper, *redundant distribution*, exploits the decentralized distribution mechanisms of autonomic grids to incorporate redundancy into complex networks of computing nodes. Recall that by the first property of autonomic grids, computing nodes distribute computation to other nodes without the aid of a centralized entity (possibly by dividing sets of subcomputations into smaller subsets and distributing those subsets,

²We note that the factor cost $c_{TR}^k(p)$ is actually independent of p , while $c_{PR}^k(p)$ is not.

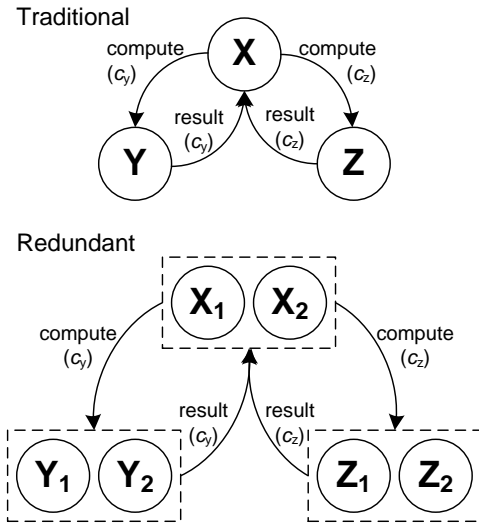


Figure 3: Sample computation-flow graphs for an autonomic grid without redundancy and with 3-vote progressive redundancy. In these examples, each node divides its sets of subcomputations into a pair of subsets, though different distribution mechanisms may use a different, perhaps variable, number of subsets.

developing iterative subcomputations, or other means). The results of subcomputations, when necessary, are returned from the “child” node back to the “parent” node that assigned the work to be performed. To use redundancy, both steps in this process must be replicated: each subcomputation must be distributed to multiple independent child nodes, and results must be returned to, and a consensus found by, multiple parent nodes. Such redundancy masks and tolerates faults within, both nodes that perform computation and nodes that compare results. In other words, the system avoids critical points by exploiting redundancy at all levels. We call the concept of injecting redundancy into the distribution process *redundant distribution*. Note that redundant distribution can leverage progressive redundancy by dynamically deploying the appropriate number of nodes to detect and correct failures.

In autonomic grids, the flow of subcomputations and their results forms a graph of network nodes. We call this graph the *computation-flow graph*. Depending on the mechanisms used to recruit participants and distribute computation, the graph will have different properties (e.g., the number of neighbors of each node and the sparseness of the graph). Figure 3 shows notional depictions of such graphs. To employ redundant distribution, each node in the graph must be replaced with a *group* of nodes. Each “parent” group recruits “child” groups to distribute subcomputations, just as each parent node recruits other child nodes in an autonomic grid without redundancy. Each group checks to ensure that a consensus exists within the group whenever it performs a function, such as executing a subcomputation or comparing results.

The number of nodes in a group is determined by progressive redundancy. That is, in 3-vote progressive redundant distribution, each group starts out with 2 nodes, and grows to become 3 nodes only when a fault occurs within that group. Figure 3 illustrates a graph of this type. All nodes within a child group perform the same subcomputations and report results back to the parent group that assigned those subcomputations. Conversely, all nodes within a parent group receive the results of the same subcomputations back from child groups and compare them to find the consensus solution.

One key problem in implementing such an algorithm is determining who is responsible for detecting a failure within a group and recruiting additional nodes for a group. While several solutions to this problem are possible, the one we use makes each parent group responsible for each of its child groups. Whenever a node completes a task, such as recruits child nodes and reaches a consensus of results, it lets its parent group know. The parent group ensures that every time a step is completed, it is completed by a consensus of the group. If failures cause a lack of consensus, the parent group recruits extra nodes for the child group, as dictated by progressive redundancy. In this way, every k -sized group must have at least $\frac{k+1}{2}$ failures in order to cause a failure of the entire group.

The second key problem in implementing redundant distribution is the procedure for recruiting nodes. It is important that a malicious node in a group cannot guide the recruitment of child groups towards other malicious nodes. It is also important that malicious nodes can neither impersonate other nodes nor change the content of existing messages. We leverage existing solutions from cryptography and secure multiparty computation literature for these problems. Yao’s garbled protocol allows for multiple nodes to compute a function without sharing their data [43]; secure peer-to-peer pseudorandom-number generation allows unbiased selection of child nodes [23]; and public-key encryption and digital signatures allow for secure exchange of data [40].

Computation-flow graphs depend on the autonomic grids’ distribution mechanisms, and thus their redundant distribution implementations differ. While the overall structure (depicted in Figure 3) is consistent, some grids use distinct variations. For example, the BOINC systems have a 2-level graph: the autonomic server for each problem being solved distributes the subcomputations directly to the groups of child nodes that execute them (multiple problems form disconnected graphs). Tile-style systems have graphs with nodes with varying numbers of children, from 1 to possibly hundreds. The number of children is typically determined by the speed of the node and the types of iterative subcomputations it performs. Organic-grid and MapReduce systems’ graphs most closely resemble those in Figure 3, with each node having at least two, but not too many children (the exact number is determined by the nature of the computation).

The following example, illustrated in Figure 3, quantitatively compares redundant distribution to traditional distribution.

With traditional distribution, node X must perform computation c , which consists of nonblocking subcomputations c_y and c_z . X asks nodes Y and Z to perform subcomputations c_y and c_z , respectively. Nodes Y and Z perform these subcomputations and return the results to X , which then combines those results into a result to c . If each node has the probability .1 of failing, then the probability of computation c failing is $1 - .9^3 = .271$.

With 3-vote redundant distribution, the nodes X_1 and X_2 form a group that must perform computation c . X_1 and X_2 decide to distribute subcomputation c_y to the group $\{Y_1, Y_2\}$ and subcomputation c_z to the group $\{Z_1, Z_2\}$. Y_1 and Y_2 then perform c_y and return the result to the group $\{X_1, X_2\}$. If Y_1 or Y_2 fails, then $\{X_1, X_2\}$ will not receive a consensus of c_y results and will find a new Y_3 to perform c_y . The process $\{Z_1, Z_2\}$ follows to solve c_z is symmetrical. After reaching a consensus of c_x and c_y results, $\{X_1, X_2\}$ reports the result to its parent group. If either X_1 or X_2 fails, the parent group will then know to recruit X_3 for the group. If each node has the probability .1 of failing, then the probability of computation c failing is the probability of at least one group failing, $1 - .972^3 = .082$.

We provide a rigorous, formal analysis of the costs and benefits of redundant distribution in Section 5.

4. TOLERATED FAULTS AND ATTACKS

While redundancy techniques cannot remove all possible types of faults in grid systems, they do alleviate a large set of faults. That set is complementary to the set of faults autonomic grid systems already resist by being decentralized. In this section, we describe the types of faults autonomic grids and redundancy techniques alleviate. Figure 4 summarizes these faults.

We employ the most general widely accepted threat model [22, 26, 30] that has been applied to numerous distributed systems [1, 2, 26]. The threat model includes Byzantine failures and allows for malicious nodes that collude and form cartels to try to mislead and break computations. Even in the face of this powerful threat model, progressive redundancy and redundant distribution reduce the probability of failure exponentially (and increase the mean time to component failure exponentially), while decreasing the computational speed linearly.

In an autonomic grid that uses no redundancy, every subcomputation is distributed by some node D to be executed by some node E that returns the result to D . There are three possible scenarios: (1) D receives the correct result, (2) D receives an incorrect result, and (3) D never receives the result. While faults may occur in each of these scenarios, only the latter two actually exhibit faulty behavior and are of our concern.

Faults that may cause D to receive an incorrect result include the subcomputation getting corrupted on the way to E ; E malfunctioning while executing the subcomputation; E reporting the wrong result on purpose (possibly maliciously); E colluding with other malicious nodes to report the wrong result; and the result getting corrupted on the way back to D . In turn, each of these faults may be caused by a hardware or software fault on the network; on either D , E , or both nodes; or by a malicious cartel of nodes. One interesting aspect of autonomic-grid redundancy is that it can treat each of these cases identically and is resilient to all of them. This statement is true because we have already assumed the worst possible case in Section 3.2: a redundancy technique fails to prevent a fault whenever the majority of the nodes executing a subcomputation reports an incorrect result. But since those incorrect results may not be identical, some of these failures may be detectable and preventable. In other words, we have already assumed that whenever a node fails, it fails in the worst possible way, either by chance or through colluding with other failing nodes. Still, as described in Section 5, even under this very broad assumption, progressive redundancy reduces the probability of failure exponentially while imposing a linear cost on the speed of the system.

Faults that may cause D to never receive a result include the subcomputation never making it to E or getting corrupted along the way; E receiving the subcomputation but never executing it; E receiving the subcomputation but deciding (possibly maliciously) not to respond; and E receiving and executing the subcomputation and sending the result to D , but the result never making it to D or getting corrupted along the way. In turn, each of these faults may be caused by a hardware or software fault on the network; or on either D , E , or both nodes. In each of these cases, autonomic grids already (without the use of redundancy) resort to a time-out mechanism that allows D to assign the task to another node if E does not return the result promptly. However, to our knowledge there has not been a sufficient analysis of the cost of these faults. Redundant distribution, using the predominantly nonblocking property of autonomic-grid subcomputations, ensures this cost is negligible. As we describe in Section 5.1, with an overwhelmingly

- Undelivered network messages (hardware and software).
- Corrupted network message (hardware and software).
- Unresponsive nodes due to failure (hardware and software).
- Unresponsive nodes due to malice or corruption (typically software).
- Infinite-loop-executing nodes (typically software).
- Malfunctioning nodes that report incorrect results (software or hardware).
- Malicious nodes that report incorrect results on purpose (typically software).
- Malicious colluding nodes that report coordinated misleading results (typically software).

Figure 4: Faults handled by autonomic grids and redundancy techniques.

high probability, if a node in an autonomic grid needs to wait for a subcomputation to finish, it can execute other nonblocking subcomputations, thus wasting virtually no time due to these faults.

It is important to note that, while certain important faults we have described here and listed in Figure 4 lend themselves to redundancy techniques, autonomic grids with redundancy additionally allow the use of other common fault-tolerance techniques that may handle different types of faults. For example, techniques that deal with distributed denial of service attacks can be used on autonomic grids with redundancy, as well as on autonomic grids without redundancy, and on traditional grids.

5. EVALUATION

In this section, we evaluate the cost and benefit of using progressive redundancy and redundant distribution techniques, compare these techniques to traditional redundancy, and provide a real-world example of injecting fault tolerance into an autonomic grid. These analyses are specific to autonomic grids and rely heavily on autonomic grids' decentralized distribution and predominantly nonblocking subcomputations. We do, however, conclude this section by demonstrating that our techniques are applicable to a broad class of networks and failures by relaxing our prior assumptions.

5.1 Redundancy Cost

Since autonomic grid systems distribute predominantly independent subcomputations onto network nodes in a decentralized manner, there are four phenomena that affect the speed of computation when applying progressive redundancy and redundant distribution.

- I. The cost of repeating subcomputations.
- II. The cost of distributing the additional subcomputations.
- III. The cost of redundantly comparing results and voting.
- VI. The cost of waiting for a blocking subcomputation.

We now analyze each of these costs with the aim of showing that the overall cost will be no more than a factor of c_{PR}^k , when using the k -vote progressive redundancy and redundant distribution techniques. We achieve this goal by arguing that every time-consuming step in an organic grid system without redundancy is performed no more than c_{PR}^k times in the redundant system, and each node spends virtually no more time being idle in the redundant system than it does in the system with no redundancy.

I. Perhaps the most straightforward cost is the cost of executing the redundant subcomputations. Since, in expectation, progressive redundancy performs each subcomputation c_{PR}^k times, the redundant system will spend, in expectation, c_{PR}^k times as much time performing the subcomputations as the system with no redundancy.

II. To estimate the impact of scheduling and distributing the redundant subcomputations, we leverage the fact that autonomic grids use decentralized scheduling mechanisms that distribute subcomputations independently of each other. In the system with no redundancy, a single node distributes parts of its computation onto several other nodes. In the system with redundancy, a group of nodes uses the secure multiparty computation procedure to distribute the parts onto several groups of nodes. The amount of computation and communication necessary is proportional to the size of the groups, and thus in expectation is c_{PR}^k times larger than required in the system with no redundancy.

III. As the results of subcomputations percolate up the computation-flow graph, nodes in the system without redundancy must combine those results into a suitable package to forward to their parent nodes. Groups of nodes in the system with redundancy combine, in expectation, c_{PR}^k copies of each result, which involves taking a majority vote on the correctness of the result. Since checking whether two results are identical requires minimal computation, combining multiple results will take at least as long as comparing two results, and thus, in expectation, each node will perform no more than c_{PR}^k times as much computation in comparing and combining the results.

IV. The cost of waiting for a particular subcomputation to complete can be nonzero only if there exist blocking subcomputations (in many autonomic grids, all subcomputations are nonblocking [4, 14, 31, 34]). In autonomic grids with some blocking subcomputations (e.g., the tile style [11]), subcomputations are predominantly nonblocking, which means that if a node has two subcomputations to perform, A and B , and A is blocked by another subcomputation C that has not completed, with high probability (exponential in the size of the input for the tile style), B is not blocked by C , and thus the node can execute B and experience no waiting cost. Thus the waiting cost depends on the size of the queue of subcomputations of each node. If the queue is at all larger than just one or two subcomputations, the probability that a node can execute a different subcomputation rather than wait for a blocking one to complete is extremely high. (In the tile style, each node's queue contains tens of thousands of subcomputations, thus we expect the chance of waiting to be negligible.) In the highly unlikely event that a node must wait for a blocking computation to complete, the expected waiting time is only linear in the time necessary to execute a single subcomputation. Therefore, the expected overall cost due to waiting for blocking subcomputations is zero for autonomic grids with completely nonblocking subcomputations, such as BOINC systems, and virtually zero for autonomic grids with predominantly nonblocking subcomputations, such as the tile style.

5.2 Redundancy Benefit

The benefit of employing progressive redundancy and redundant distribution is that the probability of the entire system failing (reporting the wrong result) is reduced. In Section 3.2, we have argued that a group of nodes, each with a probability p of failing, employing k -vote progressive redundancy, has the probability of failing (and as we show shortly, that probability diminishes exponentially in k).

The idea that each component in an autonomic grid can be replaced by a group of components, such that the group is exponentially (in k) more reliable than the original component, is central to our argument that progressive redundancy and redundant distribution inject reliability into autonomic grids. However, the exact analysis of the failure rates of the overall system is different for distinct computation-flow graphs, such as the one shown in Figure 3. The probability of the bottom-level group failing is $f_1 = f_{PR}^k(p)$,

but the next-level group fails, in the worst case, whenever any one of its children groups fails or when a majority of its own nodes fails.

Thus its probability of failing is $f_{PR}^k \left(p + \sum_{\text{children}} Pr[\text{child fails}] \right)$.

Therefore, the probability of system failure increases linearly, by a factor of roughly $f_{PR}^k(p)$, in the number of levels of the computation-flow graph. Because autonomic-grid computation-flow graphs have branching factors greater than 1 (typically much greater), the probability of system failure grows linearly as the amount of necessary computation grows exponentially. Since $f_{PR}^k(p)$ drops exponentially in k , the system can accumulate arbitrarily little failure probability at each level.

While we cannot enumerate the system-failure rates for all possible computation-flow graphs, as one example, Figure 5 shows the probability of failure of a system with a binary computation-flow graph, employing 7-vote progressive redundancy and redundant distribution. The figure assumes $p = .1$ and demonstrates that as the amount of computation grows exponentially, the probability of system failure grows only linearly.

Figure 6(a) illustrates the costs of traditional and progressive redundancies, $c_{TR}^k(p)$ and $c_{PR}^k(p)$, respectively, as a function of the acceptable probability of failure, assuming the probability of a single component failing is $p = .1$. For any given k , and thus for any given acceptable failure rate $f_{PR}^k(p)$, the ratio of $c_{TR}^k(.1)$ and $c_{PR}^k(.1)$ is 1.80, indicating that a progressive-redundancy system executes 1.80 times faster than the same system using traditional redundancy to achieve the same acceptable probability of failure. (Figure 6(c) illustrates these ratios for all p .)

Figure 6(b) shows the linear growth of the costs of progressive and traditional redundancies. Intuition dictates that progressive redundancy is most efficient when the probability p of a single component failing is low because in those situations, progressive redundancy is likely to receive sufficiently many correct answers without asking too many nodes to execute the subcomputations. If p is close to .5, likely all k nodes have to be asked to receive a majority consensus. Figure 6(c) confirms this intuition: progressive redundancy outperforms traditional redundancy by as much as a factor of 2 for low p , and at least 5%, as p approaches .5.

Let us consider an example of using the tile style [11] (with a bi-

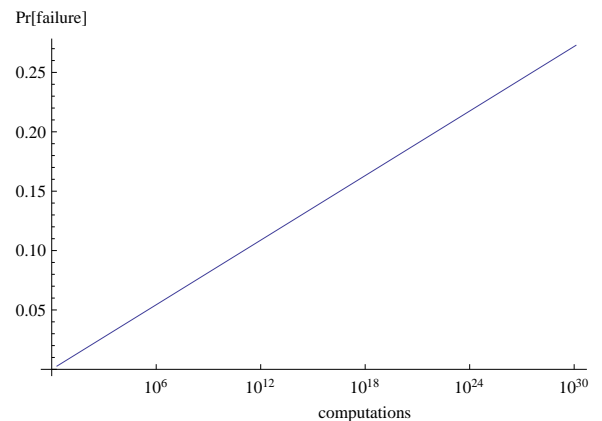


Figure 5: The failure probability of an autonomic-grid system with a binary computation-flow graph, employing 7-vote progressive redundancy and redundant distribution, as a function of the amount of computation required to solve a problem, assuming individual-node failure probability $p = .1$.

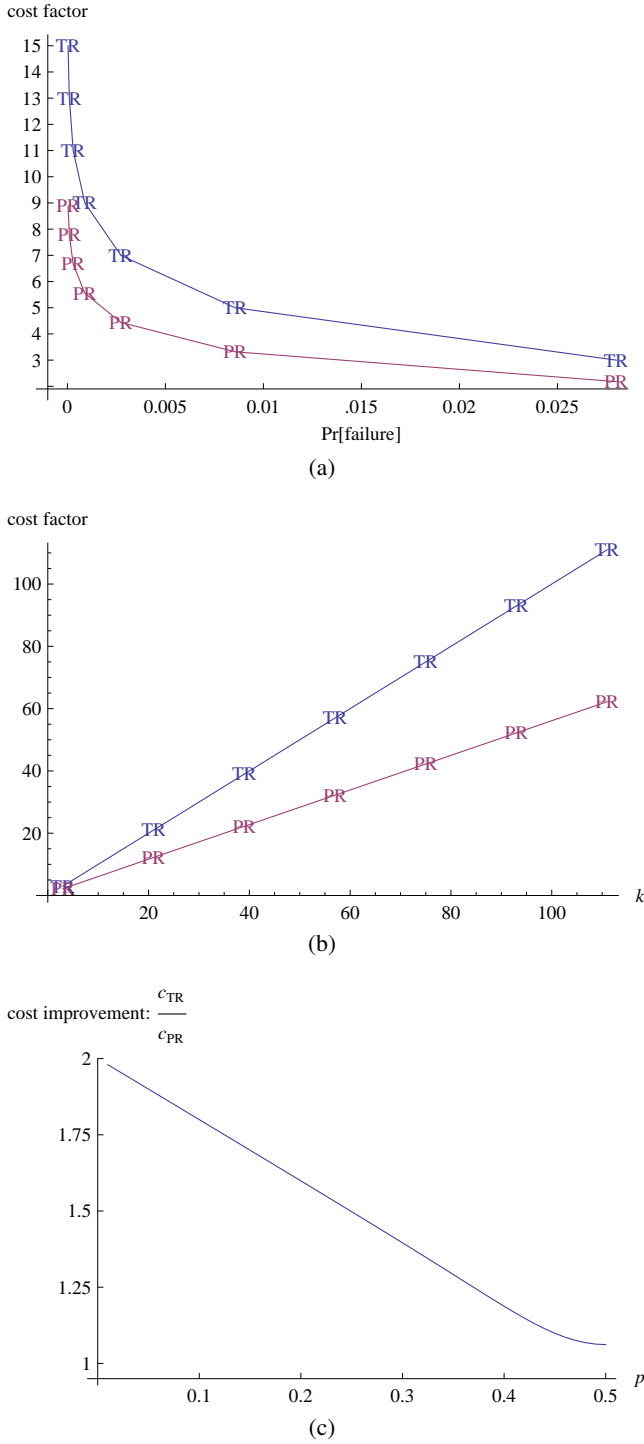


Figure 6: The cost, as a function of the probability of failure, diminishes exponentially for both traditional and progressive redundancy (a), though faster for progressive redundancy. These costs grow linearly in k (b). While (a) and (b) assume $p = .1$, in the general case (c), progressive redundancy outperforms traditional redundancy by up to a factor of 2 and is most effective for small p , though as p increases, progressive redundancy remains more efficient than traditional redundancy. (Note that for $p > .5$, redundancy is an ineffective technique.)

nary computation-flow graph) to solve a real-world problem. Empirical data from [11] shows that solving a 38-variable 100-clause instance of 3-SAT, such as ones solved in the circuit-design industry, on a public network of 1 million computers, would require 2.1 days. (Note that solving such a problem on a single computer would take over a year, creating a need for computational grids, and the size of the underlying network requires the decentralized approach of autonomic grids.) Because some fraction of the underlying public machines may be faulty and malicious, such a system without redundancy is extremely likely to fail. For example, if a single node's probability of failure is $p = 1\%$, the entire system has a $1 - .99^{10^6}$ chance of failure; that is less than 10^{-4000} chance of success. (Imaging attempting to string together 1 million successful subcomputations, each of which has a 1% chance of failing). If, however, the system employs 7-vote progressive redundancy, it will execute in under a week (2 weeks for traditional redundancy), and have a chance of success of 71.1% (Imagine stringing together 1 million successful groups-of-7-subcomputations, but each group only fails if 4 or more of its members fail.) Similarly, 11-vote progressive redundancy would take under 2 weeks (a month for traditional redundancy), but guarantee success with probability higher than 99.95%. This example illustrates both the need for use of autonomous grids to solve such highly computationally-intensive problems, and the impact redundancy can have on such projects.

While both traditional and progressive redundancies are impactful, the factor of 2 difference in speed between the two techniques can represent millions of dollars in costs for, for example, pharmaceutical companies that wish to predict structures of proteins to guide the design and development of new drugs for diagnosis and treatment.

5.3 Relaxing Assumptions

Thus far, we have made several simplifying assumptions that allowed us to most clearly explain how progressive redundancy and redundant distribution can inject robustness into autonomic grids. We had assumed that all nodes are equally reliable and independent, that the same information is available to us about all nodes, and that the results of all subcomputations are a single bit. In this section, we explain that redundancy can apply to autonomic grids deployed on networks without these assumptions, and in some cases, can even benefit from the relaxations of these assumptions. Note that it does not make sense to relax the client-reliability assumption, because the system can never be more reliable than the node submitting the original computation.

5.3.1 Probability Distribution

The formulae in Figure 2 reflect the assumption that each node has an equal probability p of failing. For some networks, that may be all the information available to a system; however, other networks may provide distinct reliability information for different classes of nodes, or even for each node. Further, probabilities of node failure may depend on each other: e.g., if a node in one part of the world fails because of a natural disaster, others near it are more likely to fail as well. In such cases, the autonomic grid scheduler can use the additional information to decrease the probability of failure. The only necessary change to the formulae in Figure 2 is the replacement of the portions that include exponentials in p and $1 - p$ with products of the appropriate probabilities of failures of the relevant nodes. For example, c_{PR}^k would become

$$\frac{k+1}{2} + \sum_{i=\frac{k+3}{2}}^k \sum_{j=i-\frac{k+1}{2}}^{\frac{k-1}{2}} \binom{i-1}{j} \prod_{c=1}^j p_c \prod_{c=j+1}^i (1-p_c),$$

where p_c denotes the probability of node c failing.

The final cost and probability of failure would then depend on the probability distribution, as well as the computation-flow graph of the autonomic grid. This statement opens a number of future-research questions, such as whether there exists an optimal distribution algorithm to minimize both the cost and probability of failure. We foresee, however, a balance between cost and robustness. One example that leads us to this hypothesis is that following the naïve algorithm of asking the most reliable nodes first would likely minimize the probability of failure, but increase the cost because the reliable nodes would be overworked. However, it is conceivable that some algorithm that requires more redundancy when using unreliable nodes and less redundancy when using reliable nodes, or assigns more weight to reliable nodes, would, in fact, decrease the cost and the probability of failure.

5.3.2 Local Information

We had assumed that every node on the network has the same knowledge about the reliability of the network. In real-world systems, it is more likely that every node has intimate knowledge about its local neighbors and relatively little knowledge about distant portions of the network. Further, during the course of computation, each node may collect information about other nodes it uses for subcomputations, such as the frequency of disagreement with others, thus generating information that is not available globally.

Because autonomic grids distribute subcomputations locally, redundancy is naturally applicable to such systems. Every distribution decision is made locally, and each node should use the most accurate information available. Depending on the kind and amount of information that nodes can collect at runtime, it may be possible to reduce the probability of failure and expected cost of progressive redundancy, though it remains future work to explore both the information-collection algorithms and the most effective uses of that information.

5.3.3 Non-Binary Results

The seemingly strong and influential assumption that the result of every subcomputation is a single bit has simplified our analysis thus far, but it actually turns out to be the worst-case scenario. Compare two types of computations: the first asks whether $2^2 = 4$ and the second asks for the result of 2^2 . For the first computation, all nodes that fail and report the wrong answer will report “no”, possibly making it difficult to distinguish between the correct and incorrect answer. For the second computation, depending on the way in which nodes fail, some may report distinct integers, and it may be possible to determine that the correct answer is 4 even if more than half of the nodes fail, because the plurality (though not the majority) will report the correct answer.

Progressive redundancy and redundant distribution are naturally applicable to systems with subcomputations with non-binary results. The probabilities of failure and costs of execution in Figure 2 are upper bounds for non-binary systems, and all our analysis applies as is. As we describe in Section 4, for all (binary and non-binary) systems with malicious nodes that collude to try to cause failures, our analysis gives tight bounds on the failure probabilities and execution costs. It is possible to develop a threat model that is weaker than ours and analyze non-binary systems that disallow cooperation between malicious nodes; however, such an analysis is unlikely to produce meaningful improvements on the bounds we present.

Another important issue that arises with non-binary results is that two non-identical answers may actually represent the same information (e.g., evaluations of $\sqrt{2}$ may return slight differences in the least significant bits). In such cases, the comparison of subcompu-

tation results is problem-specific, and the distributing nodes must be equipped with the proper comparison algorithms.

6. RELATED WORK

We have already discussed work related to autonomic grids and some fault-tolerance mechanisms [3, 10, 14, 17] specific to those grids in Section 2. However, in the broader class of grids, a major challenge in work on fault tolerance is the breadth of faults that may occur in distributed systems. In this section, we summarize work in two specific topics: fault tolerance in traditional computational grids and autonomic fault-tolerance mechanisms.

Hwang [25] proposed a method for injecting “smarter” fault tolerance into grids that suggests the possibility of handing a wide variety of faults within distributed systems. This work provides (1) a service to detect crash failures (and an extension to allow the system designer to specify other failures and how to detect them) and (2) a failure-handling framework that enforces designer-defined policies [25].

Globus grid middleware [18] includes a fault-detecting service that can handle node and network-link crashes. This service allows the detection of component failures and component replication to restore functionality. However, this detection is typically expensive and is applicable to heavyweight components [27]. Similarly, in systems with components capable of reporting their own failures, or with easily detectable failures, component replication can ensure sustainability and other qualities of service [38].

Traditional checkpoint techniques can be applied to grid systems to log partially completed work and prevent data and computation loss in case of crash failures. Checkpoints can be effective when individual subcomputations take a long time to complete [39].

Autonomous agents capable of detecting failing components and initiating on-demand replication allow somewhat autonomic fault tolerance, although the developer has to implement fault-specific detection mechanisms into these agents [16]. Nevertheless, this work is a step in the right direction, as Internet-sized systems’ complexity does not allow for centralized managers, and thus these systems must manage themselves autonomically.

7. CONTRIBUTIONS

We have presented two techniques, progressive redundancy and redundant distribution, that together inject robustness into autonomic computational grids, such as MapReduce [17], the organic grid [14], BOINC systems [3], and the tile style [11]. Progressive redundancy detects failures and dynamically allocates resources to remove them only when necessary. Redundant distribution incorporates progressive redundancy into the decentralized distribution protocols of autonomic grids, providing protection from a wide class of hardware and software failures on computational nodes and the network. Our threat model includes Byzantine faults created by cartels of malicious nodes, cooperating to sabotage a computation. Still, our techniques reduce the probability of failure exponentially at a linear cost in computation speed. This cost is up to two times lower than traditional redundancy techniques. Because today’s decentralized grid technologies either depend on robust underlying networks or use inefficient fault-tolerance techniques, our contribution should significantly increase industry’s adoption of autonomic grids.

We have provided rigorous theoretical analyses of the execution-time cost and failure rates of systems employing progressive redundancy and redundant distribution. We are currently pursuing several directions of future research, including (1) empirically demonstrating the effects of our techniques on autonomic grids, (2) using

model checking to verify that our techniques are robust under our threat model, and (3) improving the cost of progressive redundancy even further by injecting extra redundancy into low-confidence sub-computations, which progressive redundancy already identifies.

8. REFERENCES

- [1] M. Abd-El-Malek, et al. Fault-scalable Byzantine fault-tol. services. In *Symp. Oper. Sys. Prin.*, pages 59–74, 2005.
- [2] M. K. Aguilera, et al. Consensus with Byzantine failures and little sys. synch. In *DSN*, pages 147–155, 2006.
- [3] D. P. Anderson. BOINC: A sys. for public-resource comp. and storage. In *GRID*, pages 4–10, 2004.
- [4] D. P. Anderson, et al. SETI@home: An exper. in public-resource comp. *Comm. ACM*, 45(11):56–61, 2002.
- [5] J. Andrade, L. Berglund, M. Uhlén, and J. Odeberg. Using grid technology for computationally intensive applied bioinformatics analyses. In *Silico Biology*, 6(0046), 2006.
- [6] B. Berger and T. Leighton. Protein folding in the hydrophobic-hydrophilic (HP) is NP-complete. In *Conf. Comp. Mol. Bio.*, pages 30–39, 1998.
- [7] F. Berman et al. Adaptive computing on the grid using AppLeS. *Tr. Parallel and Dist. Sys.*, 14(4):369–382, 2003.
- [8] BOINC. The Berkeley open infrastructure for network computing. <http://boinc.berkeley.edu>, 2009.
- [9] G. R. Bowman, X. Huang, Y. Yao, J. Sun, G. Carlsson, L. J. Guibas, and V. S. Pande. Structural insight into RNA hairpin folding intermediates. *JACS*, 130(30):9676–9678, 2008.
- [10] Y. Brun and N. Medvidovic. Fault and adversary tolerance as an emergent property of distributed systems’ software architectures. In *Eng. Fault Tolerant Sys.*, pages 38–43, 2007.
- [11] Y. Brun and N. Medvidovic. Preserving privacy in distributed computation via self-assembly. Technical Report USC-CSSE-2008-819, USC, 2008.
- [12] R. Buyya, et al. Neuroscience instrument. and dist. analysis of brain activity data: a case for eScience on global grids. *Concurrency and Comp.: Practice and Experience*, 17(15):1783–1798, 2005.
- [13] M. Campbell, A. J. Hoane, and F. Hsu. Deep blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.
- [14] A. J. Chakravarti and G. Baumgartner. The organic grid: Self-organizing computation on a peer-to-peer network. In *ICAC*, pages 96–103, 2004.
- [15] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and performance of an enterprise desktop grid system. *J. Parallel and Dist. Comp.*, 63:597–610, 2003.
- [16] A. De Luna Almeida, et al. Tow. auton. fault-tol. multi-agent sys. In *Lat. Amer. Autonomic Comp. Symp.*, 2007.
- [17] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Symp. Operating System Design and Implementation*, 2004.
- [18] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *Conf. on Network and Parallel Computing*, pages 2–13, 2005.
- [19] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the grid: An open grid serv. arch. for dist. sys. integ. In *Open Grid Serv. Infr. Working Group*, 2002.
- [20] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Intl. J. High Perform. Comp. Appl.*, 15(3):200–222, 2001.
- [21] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A comp. manag. agent for multi-institutional grids. In *Symp. High Perform. Dist. Comp.*, 2001.
- [22] A. D. Friedman and P. R. Menon. *Fault Detection in Digital Circuits*. Prentice Hall, 1971.
- [23] J. A. Garay, P. Mackenzie, and K. Yang. Efficient and secure multi-party computation with faulty majority and complete fairness. In *Cryptology ePrint Archive*, 2004.
- [24] A. S. Grimshaw, W. A. Wulf, and Legion team. The Legion vision of a worldwide virtual comp. *Comm. ACM*, 40(1):39–45, 1997.
- [25] S. Hwang and C. Kesselman. A flexible framework for fault tolerance in the grid. *J. Grid Comp.*, 1(3):251–272, 2003.
- [26] P. Jalote. *Fault Tolerance in Dist. Sys.* Prentice Hall, 1994.
- [27] H. Jin, D. Zou, H. Chen, J. Sun, and S. Wu. Fault-tol. grid arch. and prac. *J. Comp. Sci. & Tech.*, 18(4):423–433, 2003.
- [28] P. M. Kasson and V. S. Pande. Control of membrane fusion mechanism by lipid composition: Predictions from ensemble molecular dynamics. *PLoS Comput. Bio.*, 3(11):e220, 2007.
- [29] T. Kimoto, K. Asakawa, M. Yoda, and M. Takeoka. Stock market prediction system with modular neural networks. In *Conf. Neural Networks*, pages 1–6, 1990.
- [30] I. Koren and C. M. Krishna. *Fault-Toler. Sys.* Elsevier, 2007.
- [31] E. Korpela, et al. SETI@home — massively dist. comp. for SETI. *IEEE MultiMedia*, 3(1):78–83, 1996.
- [32] M. Lamanna. The LHC comp. grid proj. at CERN. *Nuclear Instr. and Mtd. in Physics Res.*, 534(1-2):1–6, 2004.
- [33] C. B. Laney. *Comp. Gasdynam.* Cambr. Univ. Press, 1998.
- [34] S. Larson, et al. *Folding@Home & Genome@Home: Using Dist. Comp. to Tackle Prev. Intrac. Prob. in Comp. Bio.* 2002.
- [35] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor — A hunter of idle workstations. In *Conf. Dist. Comp. Sys.*, pages 104–111, 1988.
- [36] D. Lucent, V. Vishal, and V. S. Pande. Protein folding under confinement: A role for solvent. *National Academy of Sciences*, 104(25):10430–10434, 2007.
- [37] M. M. Morgan and A. S. Grimshaw. Genesis II — standards based grid computing. In *Symp. Cluster Comp. and the Grid*, pages 611–618, 2007.
- [38] A. Nguyen-Tuong. *Integrating Fault-Tolerance Techniques in Grid Applications*. PhD thesis, University of Virginia, 2000.
- [39] S. B. Priya, M. Prakash, and K. K. Dhawan. Fault tolerance-genetic algorithm for grid task scheduling using check point. In *Conf. Grid and Cooperative Comp.*, pages 676–680, 2007.
- [40] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM*, 21(2):120–126, 1978.
- [41] A. Setiawan, D. Adiutama, J. Liman, A. Luther, and R. Buyya. GridCrypt: High performance symmetric key using enterprise grids. In *Conf. Parallel and Dist. Comp., Applications and Technologies*, 2004.
- [42] S. Vedantham and S. S. Iyengar. The bandwidth allocation problem in the ATM network model is NP-complete. *Information Processing Letters*, 65(4):179–182, 1998.
- [43] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167, 1986.
- [44] Y. Zhang, et al. Genome-scale comp. approaches to memory-intensive appl. in sys. bio. In *Conf. Supercomputing*, pages 12–25, 2005.