Online Reliability Improvement via Smart Redundancy in Systems with Faulty and Untrusted Participants

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

Abstract

Many software systems today, such as computational grids, include faulty and untrusted components. As faults are inevitable, these systems utilize redundancy to achieve fault tolerance. In this paper, we present two new, "smart" redundancy techniques: iterative redundancy *and* progressive redundancy. *The two techniques are efficient, adaptive, and automated. They are* efficient *in that they leverage runtime information to improve system reliability using fewer resources than existing methods. They are* automated *in that they inject redundancy in situations where it is most beneficial and eliminate it where it is unnecessary. Finally, they are* adaptive *in that they increase redundancy on-the-fly when component reliability drops and decrease redundancy when component reliability improves. We enumerate examples of systems that can benefit from our techniques but focus in this paper on computational grid systems. We present formal analytical and empirical analyses, demonstrating our techniques on a real-world computational grid and comparing them to existing methods.*

1. Introduction

Many software systems today, such as distributed data stores (e.g., Freestore), peer-to-peer A/V streaming applications (e.g., Skype), and volunteer computing systems (e.g., BOINC), consist of very large numbers of autonomous software and hardware participants that interact over untrusted networks. In such systems, failures of individual software and hardware nodes are relatively frequent because (1) the number of nodes in the network is very large, (2) nodes are not subjected to any significant dependability checking, and (3) malicious entities can easily join the system or compromise other participants. Because faults are unavoidable in this setting, these systems require robust fault-tolerance mechanisms to achieve the desired reliability. Usually, fault-tolerance in distributed systems is accomplished by exploiting *redundancy*.

Fault-tolerant systems is a well-studied and mature field of research, characterized by diverse approaches and techniques [17]. These techniques originated from the realm of hardware systems research, but the special properties of distributed software systems require that new, customized redundancy techniques be developed and applied. In this paper, we propose two new redundancy techniques—*progressive redundancy* and *iterative redundancy*—that exploit the properties of many of today's large-scale distributed systems to increase the reliability benefit produced by redundant computation. These techniques outperform traditional redundancy approaches because they leverage runtime information to inject redundancy where it is essential and eliminate it where it is unnecessary.

We specify, formally analyze, and empirically evaluate the two techniques. We demonstrate three key characteristics that make them superior to existing methods: efficiency, adaptivity, and automation. Our techniques are more *efficient* than existing methods because they produce the same level of system reliability at a lower cost in employed system resources (or, equivalently, higher reliability at the same cost). In fact, iterative redundancy is *optimal* with respect to the cost in that it is guaranteed to use the minimum amount of computation needed to achieve the desired system reliability. Our techniques are *automated* because they recognize the situations during a computation that can result in failures and inject additional redundancy to avert those situations. Our techniques can *adapt* on-the-fly to unstable environments with changing node reliabilities. Finally, our techniques are *simple* to understand and implement. To some degree, this latter quality was unexpected. In particular, we set out to design iterative redundancy to "do the right thing" via complex data manipulation at system runtime; however, in our investigation we arrived at a much simpler but equivalent algorithm.

Progressive and iterative redundancy are applicable to systems with the following three characteristics:

1) Systems that perform highly parallelizable computation, i.e., they execute many independent tasks that do not require the completion of one task to initiate others.
2) Systems that employ many independent resources (e.g., hardware nodes) that interact asynchronously.
3) Systems that have the ability to monitor resources and make task deployment decisions at runtime.

Clearly, our techniques do not apply to all systems (e.g., airplanes with redundant dedicated hardware modules), but the range of systems to which they do apply is quite large. Examples of distributed software systems that can benefit

* co-first authors

from our techniques include distributed data stores, p2p data-streaming applications, and computational grid systems.

In this paper, we specifically focus on applying progressive and iterative redundancy to *computational grids*. We use volunteer computing, an example of computational grids, to formally analyze the cost and performance impacts of our techniques and perform a rigorous, empirical evaluation on real-world systems. Computational grid systems pool hardware and software resources of many participating distributed computers to deliver computation as a standardized service in domains such as physics [18], astrophysics [19], bioinformatics and genomics [4], economics [16] and neuroscience [7]. It is imperative that these grids be able to withstand faulty and malicious nodes as anyone may volunteer to participate, regardless of quality of hardware, soundness of software, and intent. Today's grids aim to ensure the correct execution of each task through rudimentary redundancy: asking several independent machines to perform the same computation and comparing their returned results. However, this can be costly as replicating each task N times requires expending N times as many resources or suffering an N-times slowdown in performance. By comparison, iterative and progressive redundancy tolerate faults and attacks in a smarter and more efficient way, by obtaining information about executing tasks at runtime and deploying additional computations based on runtime decisions.

The remainder of this paper is organized as follows. Section 2 presents the definitions used in our work and the assumptions underlying it. Section 3 describes the redundancy algorithms and provides their theoretical analysis, while Section 4 presents the algorithms' empirical evaluation. Section 5 discusses our results. The paper is rounded out by an overview of the related work and a recap of our contributions.

2. Definitions and Assumptions

In this section, we define our system model, state the threat model we will consider, and enumerate the assumptions we make to aid the explanation and analysis of our techniques.

2.1. System Model

In describing our techniques, we will use the following nomenclature. A *computation* is the typically large problem being solved on a grid. A *task* is one of the parts of the computation that can be performed independently of the others. A *job* is an instance of a task that a particular node performs. With redundancy, each task will be executed as several identical jobs on distinct nodes. In our model of a computational grid, a *task server* breaks up a computation into a large number of tasks. The task server then assigns jobs to *nodes* in a node pool, ensuring that each node is chosen at random. After returning a response to an assigned job to the task server, each node is reinserted into the node pool and can again be selected and assigned a new job. New volunteer nodes join the pool while other nodes leave. This system model is depicted in Figure 1.

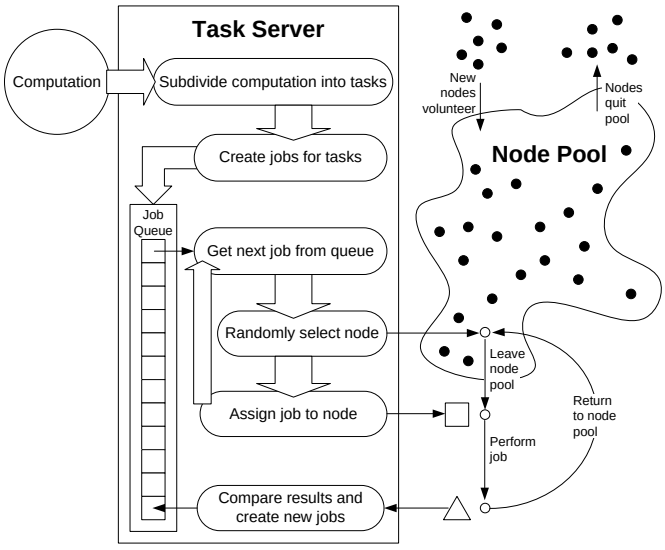


Fig. 1. A model of a computational grid.

This system model accurately represents a number of computational grid systems, including the BOINC grids [3], [5]. Other grid systems, e.g., MapReduce [10] use a similar model, though with a decentralized, often hierarchical task server.

2.2. Threat Model

In this paper, we employ the most general and widely accepted threat model [13], [15], [17] that has been applied to numerous distributed systems [1], [2], [15]. The threat model includes Byzantine failures and allows for malicious nodes that collude and form cartels to try to mislead and break computations. There are two important statements to be made about this threat model:

- First, our threat model is at least as strong as those used by redundancy techniques currently deployed in grid systems [3], [8], [10]. On the one hand, our threat model is certainly not bulletproof. For example, if failures are perfectly correlated (meaning if one node fails on a task, all nodes will fail on that task), our techniques fail to increase system reliability. On the other hand, we make no assumptions that existing grid redundancy implementations do not make.
- Second, given that faults occur, our model assumes the worst possible case scenario: all faults are Byzantine faults. That is, malicious nodes may collude to return results that most hurt the reliability of the system. For example, colluding nodes might not only return the wrong result, but they might all return the same wrong result, making it hard to identify malicious nodes. Similarly, malicious nodes are aware of other nodes that failed and how they failed, and consequently are able to return the same wrong result as those failing nodes.

Figure 2 summarizes the faults our redundancy techniques help tolerate. These are the same faults that traditional redundancy helps tolerate, but our techniques do so more efficiently.

- 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).

Fig. 2. Faults handled by computational grids employing our redundancy techniques.

2.3. Assumptions

In this section, we state a number of assumptions about the nodes of the network deploying a computational grid. These assumptions will simplify the description and analysis of our redundancy techniques. In Section 5.3 we will relax these assumptions and demonstrate that our techniques still apply and, in some cases, perform even better on more general networks.

1) Every job sent to the node pool has the same probability of failure. While some nodes may be more reliable than others, the jobs are assigned to the nodes at random.
2) Node failures are independent of each other.
3) The result of every job is one of two possible values (e.g., "yes" or "no," as in NP-complete problems [23]). This assumption is quite common in fault-tolerance research [17]. Although perhaps counterintuitive, this assumption creates a worst-case scenario because all failing and malicious nodes report not only a wrong result but the same wrong result, making it difficult to differentiate wrong results from correct results.
4) The reliability of the client that receives the final result of the computation is excluded from the system's reliability.

3. Redundancy Algorithms

In this section, we specify and explain traditional, progressive, and iterative redundancy. To characterize the behavior of each technique, we derive formulae for two measures of their effect on systems: the *system reliability* $s(r)$ achieved through replication and the *cost factor* $c(r)$ of applying the redundancy technique (how many times more resources are required). Both of these measures are functions of the average reliability $r \in [0, 1]$ of the node pool; r can be defined as the fraction of time a job returns the correct response. For completeness, we present complex formulae for costs and reliabilities. As an aid to the reader, Figure 4 provides a graphical depiction of the costs and reliabilities. Further, in Section 4, we will verify the formulae's correctness experimentally. In Section 3.1, we describe traditional redundancy, which is currently used in distributed software systems, e.g., BOINC [3], [5]. In Sections 3.2 and 3.3, we describe our techniques of progressive and iterative redundancy, respectively.

3.1. Background: Traditional Redundancy

The *k-vote traditional redundancy* technique (sometimes called *k-modular redundancy* [17]) performs k independent executions (where $k \in \{3, 5, 7, \ldots\}$) of the same task 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; a different number can easily be used as the consensus in the algorithm we describe. Figure 3(a) graphically depicts the traditional redundancy algorithm.

Example. Suppose $k = 19$ and each node's reliability is $r = 0.7$. Distributing a single job for each task (i.e., not using any redundancy) results in a system reliability of 0.7. Using traditional redundancy results in a system reliability of $1-$ the chance that at least 10 of the jobs fail: $1 - \sum_{i=10}^{19} \binom{19}{i} 0.3^i 0.7^{19-i} = 0.97$, but the cost for this procedure is using 19 times as many resources.

Analysis. Recall the two measures of a redundancy technique: *system reliability* and *cost factor*. For k-vote traditional redundancy, we refer to the system reliability as $s^k_{TR}(r)$ and the cost factor as $c^k_{TR}(r)$. Traditional k-vote redundancy repeats every task k times, independently of r. Thus,

$$c^k_{TR}(r) = k. \tag{1}$$

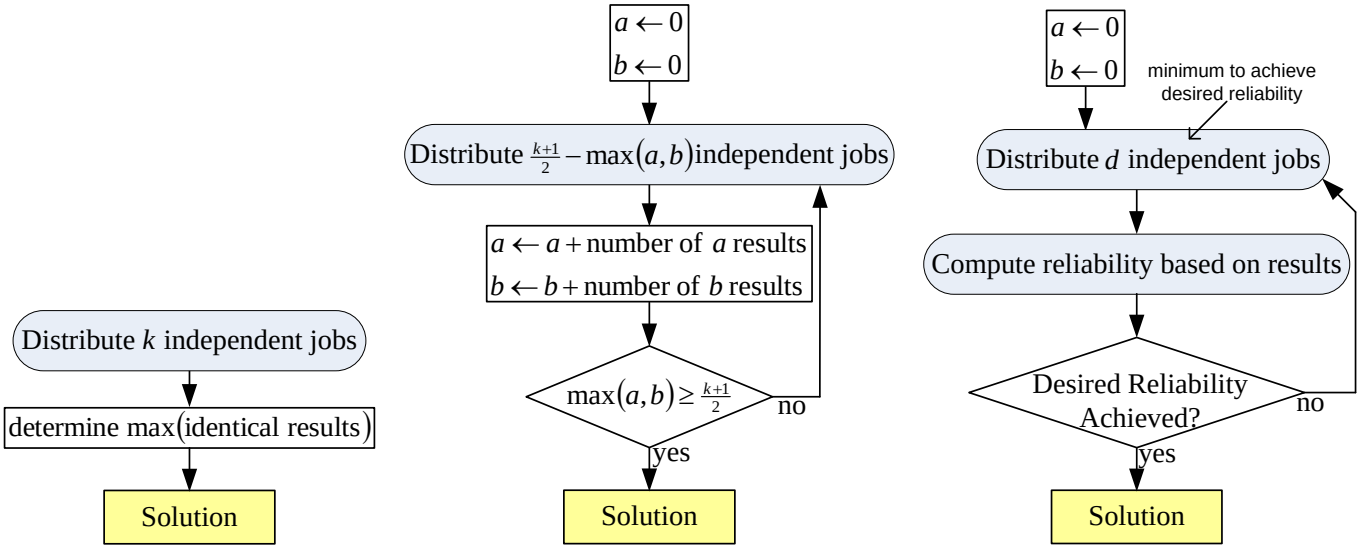
The reliability of a system with k-vote traditional redundancy is the probability that at least a consensus of jobs $\left(\frac{k+1}{2}\right)$ does not fail, in other words, the sum of the probabilities that only 0, 1, …, and $\frac{k-1}{2}$ jobs fail. Thus,

$$s^k_{TR}(r) = \sum_{i=0}^{\frac{k-1}{2}} \binom{k}{i} r^{k-i}(1-r)^i. \tag{2}$$

Figure 4 graphs the system reliability vs. the cost factor of redundancy techniques for a node pool of reliability $r = 0.7$. The reliability of a system employing traditional redundancy (labeled "TR") grows exponentially as the cost factor grows linearly.

3.2. Progressive Redundancy

We call our first improvement over traditional redundancy *progressive redundancy*. Traditional redundancy sometimes reaches a consensus quickly but still continues to distribute jobs that do not affect the task's outcome. Progressive redundancy minimizes the number of jobs needed to produce a consensus: the k-vote progressive redundancy task server



(a) Traditional Redundancy (b) Progressive Redundancy (c) Iterative Redundancy

Fig. 3. Schematics of traditional (a), progressive (b), and iterative (c) redundancy techniques.

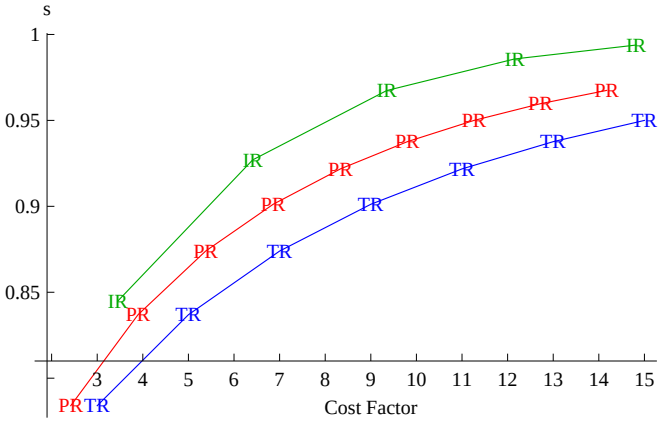


Fig. 4. The reliability of a system (s) approaches 1 exponentially, as a function of cost, for traditional, progressive, and iterative redundancy techniques (here, $r = 0.7$).

distributes only $\frac{k+1}{2}$ jobs. If all jobs return the same result, there will be a consensus and the results produced by any subsequent jobs of the same task become irrelevant. If some nodes agree, but not enough to produce a consensus, the task server automatically distributes the minimum number of additional copies of the job necessary to produce a consensus, assuming that all these additional executions were to produce the same result. The task server repeats this process until a consensus is reached. Figure 3(b) graphically depicts the progressive redundancy algorithm.

Example. As before, suppose $k = 19$ and $r = 0.7$. Using progressive redundancy, the system reliability is the probability that fewer than 10 (fewer than half) of the jobs fail, or 0.97, which is the same as traditional redundancy. As we will show in Equation (3), the cost of this procedure is using 14.2 times as many resources as a system without redundancy. This number is 1.3 times lower than the cost of traditional redundancy: while sometimes a task is distributed to as many as 19 nodes, many tasks reach the consensus earlier.

Analysis. For k-vote progressive redundancy, we use $s^k_{PR}(r)$ and $c^k_{PR}(r)$ to denote the system reliability and the cost factor, respectively. The cost factor of progressive redundancy is a bit more complex to compute than that of traditional redundancy. It is at least the consensus (since at least that many jobs must be distributed), plus the sum, for every integer i larger than the consensus up to k, of the probability that i jobs have not produced a consensus. Thus,

$$c^k_{PR}(r) = \frac{k+1}{2} + \sum_{i=\frac{k+3}{2}}^{k} \sum_{j=i-\frac{k+1}{2}}^{\frac{k-1}{2}} \binom{i-1}{j} r^{i-1-j}(1-r)^j. \quad (3)$$

The reliability of a system with k-vote progressive redundancy is the probability that at least a consensus of jobs $\left(\frac{k+1}{2}\right)$ do not fail, exactly the same as with traditional redundancy:

$$s^k_{PR}(r) = \sum_{i=0}^{\frac{k-1}{2}} \binom{k}{i} r^{k-i}(1-r)^i. \quad (4)$$

Figure 4 shows that for a given cost factor, progressive redundancy (labeled "PR") always achieves a higher system reliability than traditional redundancy.

3.3. Iterative Redundancy

Our second improvement over traditional redundancy is called *iterative redundancy*. While progressive redundancy considers the number of agreeing job responses and minimizes

the number of jobs needed to reach a consensus, iterative considers both agreeing and disagreeing job responses in attempting to reach a desired reliability. By leveraging this additional information, iterative redundancy is able to provide benefits beyond the traditional and progressive techniques.

Iterative redundancy distributes the minimum number of jobs for each task required to achieve a desired confidence level in the task result. To ensure that this minimum number is not exceeded, iterative redundancy first distributes only as many jobs as would be required to achieve the confidence threshold, assuming that all the jobs were to return the same result. Once the first set of results has been received, the algorithm reevaluates the situation and, if the desired confidence threshold has not been reached, performs another iteration by distributing the minimum number of additional jobs that would achieve the desired level confidence if all returned the majority result. This process is repeated until sufficient agreeing results have been received to reach the confidence threshold. Figure 3(c) graphically depicts the progressive redundancy algorithm.

Example. Suppose $r = 0.7$ and the desired system reliability is $s = 0.97$. Iterative redundancy uses s as the confidence threshold and calculates how many jobs' results must unanimously agree to be sure of the result with probability s. For example, if the task server distributes only one job, there is a $\frac{0.7}{0.7+0.3} = 0.7$ chance that the result is correct, but if the task server distributes four jobs and they all return the same result, there is a $\frac{0.7^4}{0.7^4+0.3^4} > 0.97$ chance that the result is correct. Thus, four is the minimum number of jobs that can achieve the confidence threshold, and the task server distributes four jobs in the first iteration. If all four jobs return the same result, the task is finished. If some jobs return a disagreeing result, the task server determines the minimum number of additional jobs that must be distributed to achieve the confidence threshold and produce the desired system reliability. For example, if three jobs return agreeing results and one returns a disagreeing result, the task server determines that at least two more jobs must return the majority result to achieve s; thus, the task server automatically distributes two more jobs. As we will show in Equation (5), the cost of iterative redundancy, for this particular example, is the use of 9.4 times as many resources as a system without redundancy. Note that this cost is 1.5 times less than the cost of progressive redundancy and 2.0 times less than the cost of traditional redundancy.

Intuitively, progressive redundancy is guaranteed to distribute the fewest jobs needed to achieve the consensus. Iterative redundancy is guaranteed to distribute the fewest jobs needed to achieve a desired system reliability. Thus far, in our description of iterative redundancy, we have avoided specifying how the technique determines this minimum number of jobs. Employing the basic intuition behind the algorithm, described above, we originally employed a relatively complex probability computation to determine the number of jobs to distribute in each iteration. However, during experimentation with the iterative redundancy algorithm, we discovered a much simpler implementation that still achieves the exact same behavior and benefits. We first describe the complex algorithm and then the simplified algorithm.

Complex algorithm. Suppose that, of $a + b$ jobs, a return one result with probability r, and b return another result with probability $1 - r$. Consider the confidence, denoted $q(r, a, b)$, that the a jobs reported the correct result. That confidence is the probability that a jobs are right and b jobs are wrong, divided by the probability that a jobs are right and b jobs are wrong plus the probability that b jobs are right and a jobs are wrong. So $q(r, a, b) = \frac{r^a(1-r)^b}{r^a(1-r)^b+(1-r)^a r^b}$. We can use this formula to determine, given some number b of jobs that have reported a result we believe to be wrong (i.e., a result that is in the minority), how many jobs must report the result we believe to be right (i.e., a result that is in the majority) for us to be s confident in the majority result. We denote that number $d(r, s, b)$. Thus, $d(r, s, b)$ is the minimum a such that $q(r, a, b) \geq s$. We can determine $d(r, s, b)$ by testing consecutive a values, or employing a more sophisticated and faster method (e.g., Newton's method [21]).

Simplifying insight. While investigating iterative redundancy, we observed that whenever a task completed, the difference between the number of majority and minority results was constant. For example, if the algorithm first sought 6 unanimously agreeing results, but got 4 agreeing and 2 disagreeing results, the algorithm would distribute 4 additional jobs (in an effort to produce an 8 to 2 majority) to achieve the desired reliability. This phenomenon arises from the counterintuitive fact that, for all j, $q(r, a, b) = q(r, a + j, b + j)$. For example, 6 agreeing results and 0 disagreeing results instills the same confidence as 106 agreeing results and 100 disagreeing results. The key to properly understanding the reasoning is that the probability of a 106-to-100 decision split occurring may be low, but once the system is faced with a 106 to 100 decision split, it is irrelevant how unlikely such a situation was to happen in the first place; given that this unlikely situation has occurred, the relevant quantity is how likely the 106 jobs are to have been right rather than wrong.

Simple algorithm. Using this insight, we can greatly simplify the iterative redundancy algorithm. We only need to determine $d(r, s, 0)$ once and set that quantity to be the required minimum difference d between the number of jobs reporting the majority result and the number reporting the minority result. For example, if $d(r, s, 0) = 6$, the algorithm iterates, automatically distributing jobs until 6 more jobs have reported one result than the other. Even further, a user may specify the desired reliability improvement in terms of the d number, and then neither the user nor our technique need know the average reliability r of nodes in the node pool. This situation is parallel to the progressive and traditional redundancy techniques, in which the user specified a parameter k. Figure 5 specifies the entire iterative redundancy technique in pseudocode. Despite being much simpler and appearing to

```
COMPUTE(Task task, int d)
1   a ← 0
2   b ← 0
3   while a − b < d
4     deploy d − (a − b) task jobs on
5       independent, randomly chosen nodes
6     a ← a + number of a results returned
7     b ← b + number of b results returned
8     if a < b
9       a ↔ b
10  return a
```

Fig. 5. The iterative redundancy algorithm. The parameter d is a measure of how much redundancy the system should deploy.

be quite different from the original algorithm, this simplified algorithm deploys the same number of redundant jobs in every situation and accomplishes the exact same efficiency.

Analysis. For iterative redundancy with d, we use $s_{IR}^{d}(r)$ and $c_{IR}^{d}(r)$ to denote the system reliability and the cost factor, respectively. The cost factor of iterative redundancy is the sum, for every b, of the probability that the system distributes $(d+2b)$ jobs and receives $d+b$ of one result and b of the other, weighted by the cost $(d+2b)$. Thus,

$$c_{IR}^{d}(r) = \sum_{b=0}^{\infty}(d+2b)Pr\left[\begin{array}{l} d+2b \text{ jobs produce} \\ d+b \text{ identical results} \end{array}\right]. \quad (5)$$

Finally, the reliability of a system with iterative redundancy is the probability that d more jobs return the right result than the wrong result. Thus,

$$s_{IR}^{d}(r) = q(r,0,d) = \frac{r^d}{r^d + (1-r)^d}. \quad (6)$$

Figure 4 shows that for a given cost factor, iterative redundancy (labeled "IR" in Figure 4) always achieves a higher system reliability than both traditional and progressive redundancy. We should note that, while iterative redundancy may appear superior to progressive redundancy, we will show in Section 5.2 that progressive redundancy's higher predictability and bounds on task-response time make it preferable for systems with certain requirements.

4. Evaluation

Having explained the mechanisms behind progressive and iterative redundancy, in this section, we will analyze the costs and benefits of these techniques. In addition to some formal arguments based on Equations (1) through (6), each of these analyses will include data from a simulation of a distributed computational grid system and a deployment of the BOINC grid [3], [5] on the distributed PlanetLab platform [20].

We will first, in Section 4.1, describe our evaluation platforms for the simulation and BOINC deployments. We will then, in Section 4.2, evaluate the *efficiency* of our techniques; in Section 4.3, describe how our techniques *automate* making distributed systems more reliable by injecting redundancy on-the-fly into situations where it is most beneficial and eliminate it where it is unnecessary; and, in Section 4.4, discuss how our techniques *adapt* to variable node reliability.

4.1. Evaluation Platforms

We used two off-the-shelf platforms to evaluate our redundancy techniques: XDEVS [11] and BOINC [5].

XDEVS Simulation Environment. The XDEVS simulation framework [11] is a highly extensible discrete event simulator that is specialized for building simulations of software systems. Unlike other discrete event simulators, XDEVS provides a software-oriented programming model by supporting abstractions commonly used in software design models, such as components, interfaces, and resources, as first-class modeling entities. We modeled the task server as an XDEVS component and the node pool as an XDEVS resource. The jobs distributed to nodes in the XDEVS simulation do not solve any specific problem; rather, they perform simulated work for a simulated period of time. The XDEVS simulation engine, which is designed to incorporate domain-specific algorithms and constraints, ensured that the aspects of our system model enumerated in Section 2 were enforced.

Using XDEVS for empirical evaluation allowed us to rapidly implement each redundancy technique, flexibly experiment with system parameters such as the job reliability and amount of redundancy employed, and observe dynamic behavior not exposed by formal static analysis. To allow for comparison, all the data given in this section were generated from XDEVS simulation runs with

- at least 1,000,000 tasks and 10,000 nodes,
- job completion times that varied stochastically between 0.5 and 1.5 time units, according to a uniform distribution, and
- an average node reliability of 0.7 (except where explicitly noted otherwise, as in Section 4.3).

Each simulation run recorded the simulated time units required to complete the computation, the total number of jobs generated, the average number of jobs per task generated, the maximum number of jobs generated for any single task, the number of tasks that achieved a correct result, the average response time per task, and the maximum response time for any task.

BOINC Deployment. Our second empirical evaluation utilized the BOINC grid system, best known for applications such as SETI@home and Folding@home [3]. The BOINC server software [5] allows distribution of a custom problem to volunteering computers. To test our redundancy techniques, we (1) developed a custom task server that decomposes 3-SAT problems into individual tasks that test whether particular Boolean assignments satisfy a Boolean formula, and (2)

modified the job-assignment and result-validation procedures to employ iterative and progressive redundancy.

We deployed our BOINC server on a 200-node subset of PlanetLab [20], a globally distributed network of machines of varying speeds and resources. PlanetLab, distributed at 487 locations around the world, consists of 1,006 nodes, almost half of which are typically unresponsive. Of the responsive nodes, some are heavily overloaded or exceedingly slow.

In deploying BOINC on PlanetLab, we uncovered that BOINC employs two levels of redundancy: every task is deployed as k jobs, but also every job is deployed twice in case the node executing the job crashes and fails to return a result. BOINC is thus forced to waste considerable resources in order to avoid failures. Our techniques can handle nodes returning incorrect results as well as nodes not returning results (with proper time-out mechanisms), reducing the use of resources even further.

To allow for comparison, all the data given in this section were generated from BOINC executions on 200 nodes that solved 22-variable 3-SAT problems. Each problem was decomposed into 140 tasks. Three types of failures were present in the BOINC system:

- seeded failures that caused the wrong result to be returned 30% of the time,
- PlanetLab nodes becoming unresponsive, and
- all other unanticipated failures that PlanetLab nodes might experience.

Each execution recorded the time required to complete the computation, the total number of jobs generated, the average number of jobs per task generated, the maximum number of jobs generated for any single task, and the number of tasks that achieved a correct result.

4.2. Efficiency

Iterative, progressive, and traditional redundancy techniques all improve the reliability of computational grids. In Section 3, we described our expectations for the performance of our techniques. In this section, we present empirical data on simulated grid systems and on BOINC grid systems deployed on PlanetLab to (1) confirm our theoretical predictions and (2) demonstrate the efficiency of our techniques as compared to traditional redundancy.

Figure 6 shows our empirical data from the XDEVS simulations supporting the claim that our techniques outperform traditional redundancy in the number of jobs and total time to execute the computation. This data (for $r = 0.7$) closely agrees with our analytical predictions from Section 3; in fact, the graph is almost identical to the one from Figure 4.

The exact cost factor improvement that our techniques exhibit over traditional redundancy depends on r. Figure 7 demonstrates the improvement of iterative and progressive redundancy, as a function of r, over traditional redundancy. Progressive redundancy is most helpful for high r. If r is close to 0.5, the cost factor of k-vote progressive redundancy is

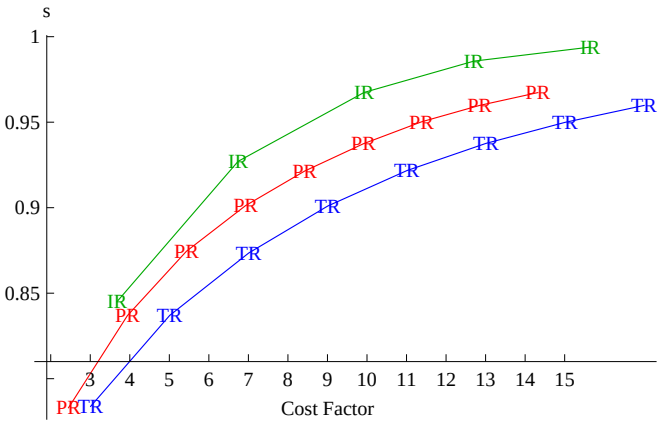


Fig. 6. Experimental results from the XDEVS simulations, shown here for $r = 0.7$.

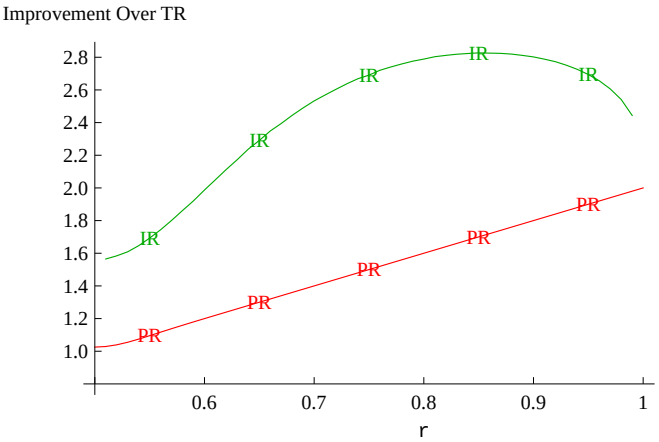


Fig. 7. The ratio improvement in cost factor for progressive (PR) and iterative (IR) redundancy over traditional redundancy. The benefit of each technique depends on the nodes' reliability r.

close to k because, most likely, the nodes just barely reach the consensus. If, however, r is close to 1, progressive redundancy reaches the consensus quickly and shows greatest benefit over traditional redundancy. For r approaching 1, progressive redundancy uses 2.0 times fewer resources than traditional redundancy.

Iterative redundancy follows a similar trend. It is more efficient for larger r, but it is at least 1.6 times as efficient even for r close to 0.5. Iterative redundancy's efficiency peaks at 2.8 times that of traditional redundancy for $r \approx 0.86$. As r approaches 1, the efficiency of iterative redundancy decreases slightly to $\approx$ 2.4 times that of traditional redundancy. We hypothesize that this decrease exists because, when almost all nodes are reporting correct results, utilizing runtime information to make redundancy decisions is somewhat less beneficial than when nodes' behavior is highly variable. More precisely, as r increases, the cost $c_R^k(r)$ to produce a constant increase in $s_R^k(r)$ decreases linearly for traditional redundancy,

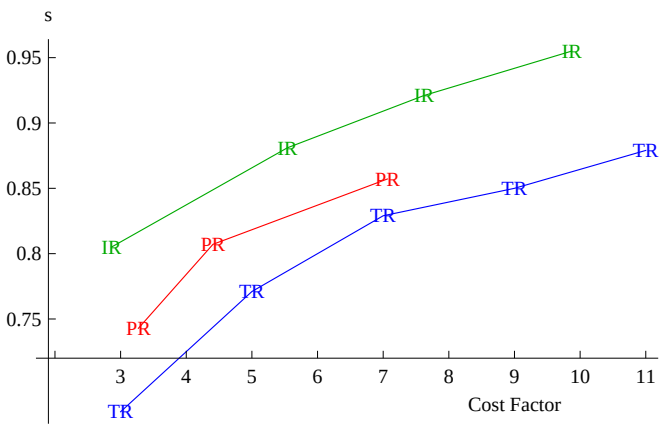


Fig. 8. BOINC deployment experimental results comparing the efficiency of redundancy techniques.

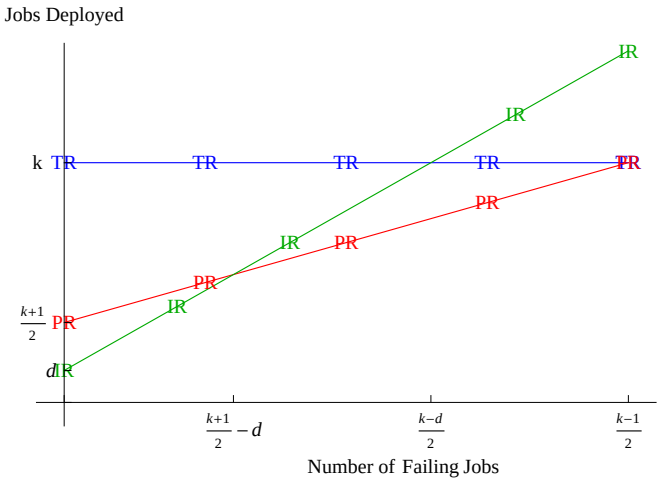


Fig. 9. Iterative redundancy (IR) injects more redundancy than progressive (PR) and traditional (TR) redundancy into the cases with the most failing jobs.

but approaches a constant for iterative redundancy. We intend to conduct further experiments to test this hypothesis. Also, note that even for values of r very close to 1, iterative redundancy performs much better (2.4 times better) than traditional redundancy.

In our next set of experiments, we deployed our redundancy techniques on a BOINC grid running on PlanetLab. Since we seeded some faults, we knew the reliability of the nodes would be no higher than $r = 0.7$. However, due to the other PlanetLab failures, we were unaware of the actual value of r. This scenario accurately represented typical real-world deployments. Figure 8 depicts the system reliability as a function of the cost factor of each technique. These data points are averages of multiple executions. Iterative redundancy, as we predicted, outperformed the other redundancy techniques, delivering the highest system reliability at the lowest cost in resources. Progressive redundancy also outperformed traditional redundancy.

The measurements in Figure 8 allowed us to estimate the reliability of PlanetLab nodes. Each technique's executions regularly reported costs and system reliabilities consistent with $r \in (0.64, 0.67)$. Our seeded faults lowered r to 0.7 and the naturally occurring PlanetLab faults were responsible for the difference. The consistency of the derived node reliabilities, both among multiple trials with different parameters and across all three techniques, provides strong evidence for the validity of these experiments.

4.3. Automation

While traditional redundancy injects the same amount of redundancy regardless of the scenario, progressive and iterative redundancy automatically determine which situations require more or less redundancy. For example, suppose we employed 15-vote traditional redundancy to execute two tasks: A and B. We would then execute 15 A jobs and 15 B jobs, even if all A jobs returned the same result while only 8 B jobs returned one result and 7 returned the other. In the end, we would be very sure of the A result but much less sure of the B result. Had we used iterative redundancy instead, our technique would have automatically determined, after executing just a few jobs, that the A jobs are achieving a higher reliability and would have spent more resources on the B jobs. In the end, we would have been equally confident in the A and B results.

To illustrate the automation provided by iterative and progressive redundancy, consider how the behavior of each algorithm differs in a "lucky" situation (in which nearly all jobs return agreeing results) vs. an "unlucky" situation (in which some jobs return results that disagree with the majority). Figure 9 shows, for each algorithm, the relationship between the amount of redundancy employed (i.e., the total number of jobs distributed) and the number of jobs that have returned a disagreeing result. While we have produced this graph through formal analysis and empirical experiments, Figure 9 uses the most exact theoretical data and symbolic labels on the axes to make it more instructive. For traditional redundancy, the amount of redundancy is constant regardless of how many nodes disagree with the majority. When progressive redundancy is used, the total number of jobs distributed is equal to the number of disagreeing nodes plus $\frac{k+1}{2}$ (the number of nodes in the majority). Finally, the iterative redundancy technique distributes a total number of jobs equal to twice the number of disagreeing nodes plus d.

Therefore, progressive and iterative redundancy provide an automated mechanism for applying additional redundancy in situations where some jobs have failed. These same mechanisms allow the progressive and iterative redundancy techniques to automatically keep the confidence level achieved for various tasks relatively constant.

4.4. Adaptation

Iterative and progressive redundancy adapt to a changing environment by automatically increasing the number of jobs

per task when node reliability drops and decreasing the number of jobs per task when node reliability increases. For progressive redundancy, the number of jobs per task is bounded and varies between $\frac{k+1}{2}$ and k. For example, when node reliability is close to 0.5, progressive redundancy tends to use about k jobs for each task, while when node reliability approaches 1, it uses about $\frac{k+1}{2}$ jobs per task. For iterative redundancy, the number of jobs per task is unbounded.

To illustrate this quality, we conducted a number of experiments in which the reliability of the nodes varied over time. Figure 10 shows three executions of a single system over time using each of the three techniques. Node reliability (top), varies between 0.75 and 0.95, and is the same for all three executions. However, the average jobs per task (middle), and the percentage of tasks that return a correct result (bottom), are quite different for each technique. Traditional redundancy keeps a constant number of jobs per task, but as the reliability r of the underlying nodes drops, the percentage of tasks resulting in a correct result drops significantly. Progressive redundancy allows the jobs per task to vary within a predefined range to adjust to changes in node reliability; however, it is still not immune to drops in r, as the system reliability dips a fair amount. Iterative redundancy has the largest variations in jobs per task but keeps the number of tasks that result in a correct result fairly constant.

These graphs also illustrate how iterative redundancy outperforms progressive redundancy in terms of cost factor. When node reliability peaks, progressive redundancy "maxes out" system reliability and produces 100% correct results, but it cannot reduce the jobs per task below $\frac{k+1}{2}$. At these times, progressive redundancy is "wasting effort" by asking more nodes than are needed. Iterative redundancy, on the other hand, is able to reduce the jobs per task to as low as 2.

One interesting side effect of progressive redundancy being less adaptive than iterative redundancy is that in certain situations, progressive redundancy is more predictable in terms of bounds on the response time. We will discuss in Section 5 some scenarios that may make progressive redundancy preferable to iterative redundancy.

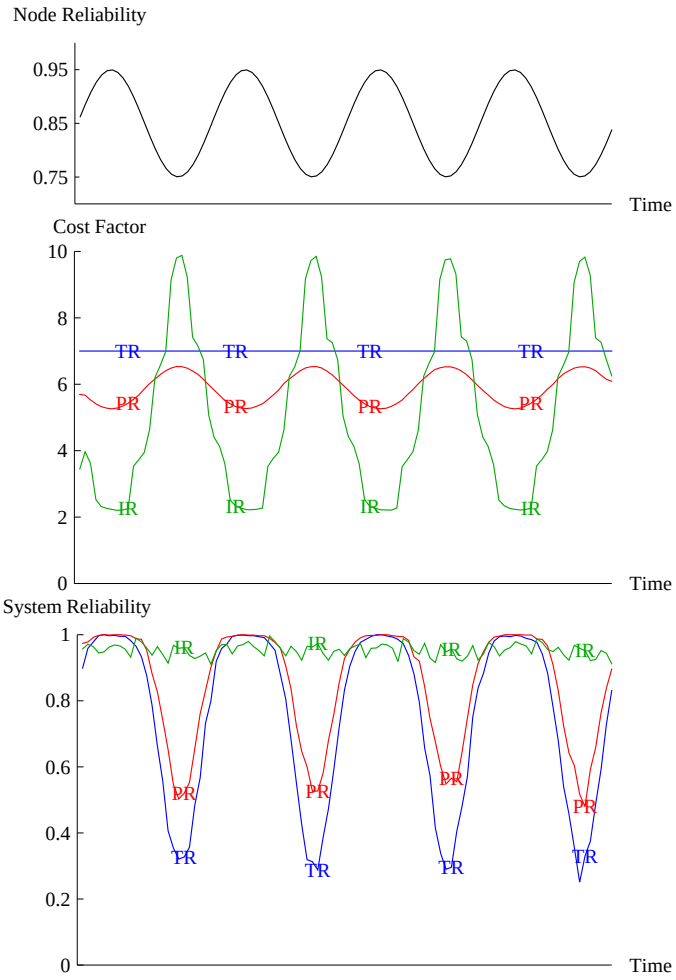


Fig. 10. Traditional, progressive, and iterative redundancy techniques react differently to a changing node reliability r. As r changes (top), traditional redundancy keeps a constant cost while progressive and iterative redundancy adapt by executing more jobs (middle). When r drops, however, traditional redundancy allows system reliability to drop significantly, progressive redundancy allows system reliability to drop slightly, while iterative redundancy keeps system reliability fairly constant (bottom).

5. Discussion

In this section, we discuss how our techniques respond to poor estimates of the reliability of nodes, the possible shortcomings of our techniques, and the effect of relaxing of our earlier assumptions.

5.1. A Priori Knowledge of Node Reliabilities

We set out to design iterative redundancy to achieve a desired system reliability s, given a particular reliability of the nodes. In some sense, we expected that the algorithm will need to know each job's reliability r in order to correctly approximate how much redundancy to inject. As we already described in Section 3.3, during our exploration of iterative redundancy we discovered that a much simpler algorithm that does not use r or s performed exactly the same actions as our original goal algorithm. The user only needed to specify how much improvement was needed (or how high a cost in execution time she was willing to pay) and the algorithm used those resources to achieve the highest possible system reliability.

In addition to our theoretical analysis, we wanted to verify our discovery experimentally. To that end, we used our XDEVS simulation engine to simulate multiple sets of three systems attempting to achieve a given system reliability s: the first system in each set overestimated the reliability r, the second estimated r correctly, and the third underestimated r. Figure 11 shows that, for a fixed cost factor, iterative redundancy produced the same system reliability regardless

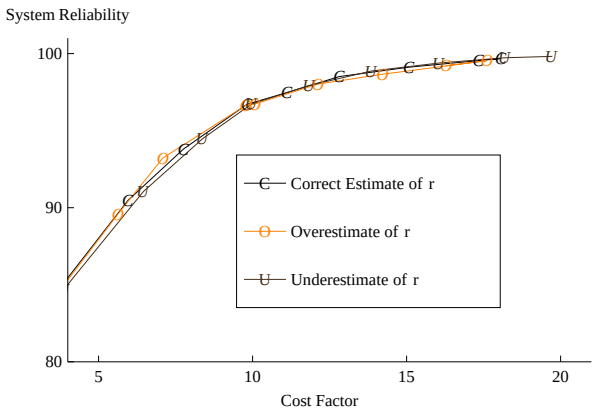


Fig. 11. The iterative redundancy algorithm is robust to poor estimates of r.

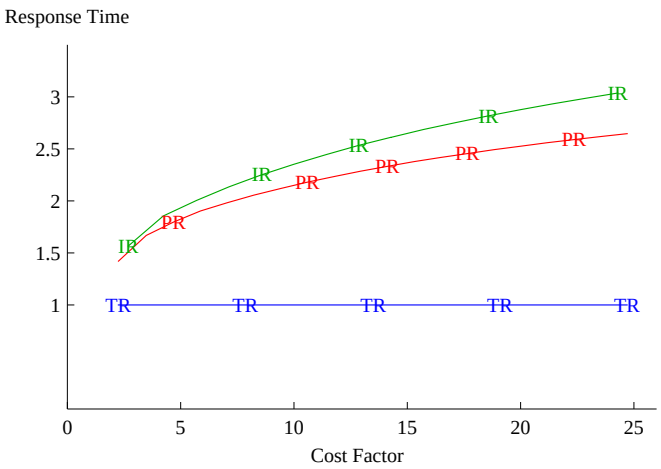


Fig. 12. The average response time for tasks using traditional, progressive, and iterative redundancy.

of the estimated r, supporting our theoretical conclusions.

5.2. Possible Shortcomings

We have concentrated on minimizing the number of jobs needed to execute in order to complete computations. However, we have thus far ignored one aspect of our redundancy techniques that may be important in some realms. Using traditional redundancy, a task server can deploy all k jobs at once. Meanwhile, using progressive or iterative redundancy, the task server must deploy several jobs and wait for the responses before possibly choosing to deploy more. Therefore, our techniques can increase the response time for a particular task. In the realm of computational grids with a large number of independent tasks, the increased response time does not present a problem because the nodes can always execute jobs related to other tasks [3], [10]. However, some applications may pose requirements on the response time for particular tasks.

A task server employing traditional redundancy attempts to start all the jobs related to a single task at once, in a single wave. In contrast, a task server employing progressive redundancy may wait for several waves of jobs to finish before deploying more; however, it guarantees that there will be no more than $\frac{k-1}{2}$ such waves. Iterative redundancy makes no such guarantees, and while it is very unlikely, any one task may require arbitrarily many waves of jobs.

Figure 12 demonstrates the average response times for tasks using traditional, progressive, and iterative redundancy techniques collected from XDEVS simulations. The response time depends on the cost factor. For the instances measured, progressive redundancy took between 1.4 and 2.5 times longer and iterative redundancy between 1.4 and 2.8 times longer to respond than traditional redundancy. Thus, progressive redundancy offers a lower average response time and a lower upper bound on response time than iterative redundancy.

In addition to response time, some domains concerned with privacy may want a hard limit on the number of times a task may be replicated and deployed. Traditional and progressive redundancy can provide such limits, while iterative redundancy cannot. It is possible to impose an artificial limit on the number of replicas. While we have not yet fully investigated this option, our intuition is that such a limit would reduce both, the system reliability and the cost factor. However, these effects would be minor because they only affect the low-probability cases that use a large number of replicas.

The idea of artificially limiting the number of replicas highlights an interesting research direction. In this paper, we have identified a new dimension along which redundancy techniques can vary: the degree to which runtime information about the reliability of individual tasks is leveraged to increase or decrease replication of those tasks on-the-fly. This dimension captures a trade-off between the response time for each individual task and the performance of the system as a whole. Viewed from this dimension, traditional redundancy techniques represent the extreme of this spectrum where individual task response time is minimized but performance of the system as a whole is suboptimal. Iterative redundancy represents the other extreme of the spectrum and progressive redundancy lies within the spectrum. Iterative redundancy optimizes the performance of the system as a whole while increasing the response time of individual tasks. Other techniques along this spectrum (such as iterative redundancy with an artificial limit on the number of replicas) may provide interesting insights into the trade-off and remain the focus of our future work.

5.3. Relaxing Assumptions

We have made several assumptions, described in Section 2.3, that allowed us to more clearly explain how iterative and progressive redundancy can inject robustness into distributed systems such as computational grids. We had assumed that every job sent to the node pool had the same probability of failure, that those failures were independent, and that the result of every job was one of two possible values. In this section,

we explain that redundancy can apply to computational grids deployed on networks without these assumptions, and in some cases can even benefit from relaxing the assumptions.

We have already described, in Section 5.1, that the user does not need to know r to use iterative and progressive redundancy. However, knowing r can help calculate the reliability of the systems employing those redundancy techniques. An improved estimate of r will result in a more accurate calculation of system reliability. It is also possible to leverage information about distinct r values for every job and dependency between job failures in calculating the system reliability. Equations (1) through (6), as well as the analysis in Section 4, reflect the assumption that each job has an equal probability of failure. We made this assumption based on the fact that many computational grids (e.g., BOINC [3] and MapReduce [10]) assign jobs to nodes from the node pool at random; therefore, from the node reliability perspective, every job submitted to the job queue has the same probability of failure. In some circumstances and for certain classes of jobs, it may be possible to estimate the distinct reliability information for different classes of nodes and jobs. Further, probabilities of node and job 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 job schedulers can use the additional information to decrease the probability of failure. The only necessary change to Equations (1) through (6) is the replacement of r with appropriate reliabilities of the relevant nodes. For example, if r_c denoted the probability of a particular job c failing, Equation (3) would become

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

The final cost and probability of failure would then depend on the probability distribution, as well as the computation-flow graph of the grid. This statement opens a number of 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 may be possible to design an algorithm that requires more redundancy when using unreliable nodes and less redundancy when using reliable nodes, or that assigns more weight to reliable nodes. Such an algorithm might decrease both the cost and the probability of failure.

The assumption that the result of every task 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 tasks, for example: the first asks whether $2^2 = 4$ and the second asks for the result of 2^2. For the first task, all nodes that fail and report the wrong result will report "no", possibly making it difficult to distinguish between the correct and incorrect result. For the second task, nodes may report distinct integers, and it may be possible to determine that the correct result is 4 even if more than half of the nodes fail, because the plurality (though not the majority) will report the correct result.

Iterative and progressive redundancy are naturally applicable to systems with tasks with non-binary results. The probabilities of failure and costs of execution we have presented are upper bounds for non-binary systems, and all our analysis applies as is. 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 aspect of non-binary results is that two non-identical results 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 jobs' results is problem-specific, and the distributing nodes must be equipped with the proper comparison algorithms.

6. Related Work

In this section, we discuss several computational grid technologies to which our reliability techniques apply and then contrast our techniques with existing fault-tolerance approaches.

We have used BOINC [3], [5] to evaluate our work in Section 4. BOINC is currently deployed on over a million computers. Examples of BOINC applications include SETI@home, LHC@home, Folding@home, Malariacontrol.net, Climateprediction.net, and many others [3], [5]. BOINC applications are autonomous actors responsible for the subdivision, scheduling, and distribution of work. BOINC utilizes the traditional redundancy algorithm (as we described in Section 3.1) to achieve robustness and fault tolerance.

MapReduce [10] is another well-known autonomic grid technology, in which the developer designs two functions: `Map` and `Reduce`. The `Map` function takes a computation and divides it into a small number of tasks that can be solved in parallel. The `Reduce` function takes the solutions to several tasks and combines them into a solution to the original problem. The MapReduce infrastructure handles taking a single computation, distributing it, and combining the solutions into the final result. The best known use of MapReduce is computing Google's PageRank [10].

Reliability is a well-studied field in software systems and some approaches have emerged to leverage redundancy to improve a system's reliability. Most of these approaches fall into one of two categories: primary backup [6] and active replication [12]. Primary backup uses multiple servers to improve the reliability of a service. One server is designated as the primary server while the others act as backups. The primary-backup architecture handles on-the-fly updates of the

backups to ensure limits on losses from primary-server failures, while keeping the cost of updates among the servers low. Primary backup is widely used in commercial fault-tolerant systems [6]. Our iterative and progressive redundancy techniques complement primary backup by specifying, at runtime, how many backups should exist to guarantee a particular level or reliability.

Active replication removes the centralized control of primary backup and minimizes losses that occur when a subset of the replicas fail. Active replication incurs a high cost associated with keeping all the replicas synchronized [12]. Again, our iterative and progressive redundancy techniques complement active replication by specifying, at runtime, how many replicas should exist to guarantee a particular level or reliability. While primary backup and active replication propose mechanisms for implementing redundancy mechanisms in distributed systems, our techniques help make those mechanisms more efficient.

Hwang [14] 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 a service to detect crash failures (and an extension to allow the system designer to specify other failures and how to detect them) and a failure-handling framework that enforces designer-defined policies [14]. Traditional checkpoint techniques can also be applied to grid systems to log partially completed work and prevent data and computation loss in cases of crash failures. Checkpoints can be effective when individual subcomputations take a long time to complete [22].

Finally, autonomous agents capable of detecting failing components and initiating on-demand replication allow autonomic fault tolerance, although the developer has to implement fault-specific detection mechanisms into these agents [9].

7. Contributions

We have presented two techniques, iterative and progressive redundancy, that improve on existing reliability techniques by leveraging runtime information. We identified several types of systems to which our techniques apply and concentrated in this paper on computational grids. Our techniques are more efficient than existing methods in their use of resources, automated because they inject redundancy in situations where it is beneficial and eliminate it where it is unnecessary, and adaptive because they increase redundancy on-the-fly when component reliability drops and decrease redundancy when component reliability improves. In addition to a rigorous theoretical analysis of our techniques, we presented an empirical evaluation based on two deployments: a discrete event simulator XDEVS and the BOINC grid system. In the process of conducting and evaluating this work, a spectrum of possible redundancy techniques has begun to emerge, with the two extreme points on that spectrum being traditional redundancy and iterative redundancy. We have identified a number of interesting research questions, which will frame our future work.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP05)*, Brighton, UK, 2005, pp. 59–74.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, "Consensus with Byzantine failures and little system synchrony," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN06)*, Philadelphia, PA, USA, 2006, pp. 147–155.

[3] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID04)*, Pittsburgh, PA, USA, 2004, pp. 4–10.

[4] J. Andrade, L. Berglund, M. Uhlén, and J. Odeberg, "Using grid technology for computationally intensive applied bioinformatics analyses," *In Silico Biology*, vol. 6, no. 0046, 2006.

[5] BOINC, "The Berkeley open infrastructure for network computing," http://boinc.berkeley.edu, 2009.

[6] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," pp. 199–216, 1993.

[7] R. Buyya, S. Date, Y. Mizuno-Matsumoto, S. Venugopal, and D. Abramson, "Neuroscience instrumentation and distributed analysis of brain activity data: a case for eScience on global grids," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 15, pp. 1783–1798, 2005.

[8] A. J. Chakravarti and G. Baumgartner, "The organic grid: Self-organizing computation on a peer-to-peer network," in *Proceedings of the 1st International Conference on Autonomic Computing (ICAC04)*, New York, NY, USA, 2004, pp. 96–103.

[9] A. De Luna Almeida, J.-P. Briot, S. Aknine, Z. Guessoum, and O. Marin, "Towards autonomic fault-tolerant multi-agent systems," in *Proceedings of the 2nd Latin American Autonomic Computing Symposium (LAACS07)*, Petropolis, RJ, Brazil, 2007.

[10] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI04)*, San Francisco, CA, USA, December 2004.

[11] G. Edwards and N. Medvidovic, "A highly extensible simulation framework for domain-specific architectures," Center for Software Engineering, University of Southern California, Tech. Rep. USC-CSSE-2009-511, 2009.

[12] P. Felber and A. Schiper, "Optimistic active replication," in *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS01)*. Phoenix, AZ, USA: IEEE Computer Society, 2001, pp. 333–341.

[13] A. D. Friedman and P. R. Menon, *Fault Detection in Digital Circuits*. Prentice Hall, 1971.

[14] S. Hwang and C. Kesselman, "A flexible framework for fault tolerance in the grid," *Journal of Grid Computing*, vol. 1, no. 3, pp. 251–272, September 2003.

[15] P. Jalote, *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.

[16] T. Kimoto, K. Asakawa, M. Yoda, and M. Takeoka, "Stock market prediction system with modular neural networks," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN90)*, June 1990, pp. 1–6.

[17] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. Elsvier, Inc., 2007.

[18] M. Lamanna, "The LHC computing grid project at CERN," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 534, no. 1-2, pp. 1–6, 2004.

[19] C. B. Laney, *Computational Gasdynamics*. Cambridge University Press, 1998.

[20] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A blueprint for introducing disruptive technology into the Internet," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 59–64, 2003.

[21] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge University Press, 2007.

[22] S. B. Priya, M. Prakash, and K. K. Dhawan, "Fault tolerance-genetic algorithm for grid task scheduling using check point," in *Proceedings of the 6th International Conference on Grid and Cooperative Computing (GCC07)*, 2007, pp. 676–680.

[23] M. Sipser, *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.