Smart Redundancy for Distributed Computation

Yuriy Brun[†], George Edwards[⋆], Jae young Bang[‡], and Nenad Medvidovic[‡]
[†] University of Washington, Seattle, WA, USA, brun@cs.washington.edu
[⋆] Blue Cell Software, Los Angeles, CA, USA, george@bluecellsoftware.com
[‡] University of Southern California, Los Angeles, CA, USA, {jaeyounb, neno}@usc.edu

Abstract

Many distributed software systems allow participation by large numbers of untrusted, potentially faulty components on an open network. As faults are inevitable in this setting, these systems utilize redundancy and replication to achieve fault tolerance. In this paper, we present a novel "smart" redundancy technique called iterative redundancy*, which ensures efficient replication of computation and data given finite processing and storage resources, even when facing Byzantine faults. Iterative redundancy is more efficient and more adaptive than comparable state-of-the-art techniques that operate in environments with unknown system resource reliability. We show how systems that solve computational problems using a network of independent nodes can benefit from iterative redundancy. We present a formal analytical analysis and an empirical analysis, demonstrate iterative redundancy on a real-world volunteer-computing system, and compare it to existing methods.*

1. Introduction

Many software systems today, such as distributed data stores (e.g., Freenet [11]) and peer-to-peer A/V streaming applications (e.g., Skype [6]), consist of large numbers of autonomous software and hardware participants that interact over untrusted networks. These systems utilize redundancy mechanisms to tolerate faults and achieve acceptable levels of reliability. In this paper, we focus on one subset of these systems: *distributed computation architectures* (DCAs), which solve massive problems by deploying highly parallelizable computations (i.e., sets of independent tasks) to dynamic networks of potentially faulty and untrusted computing nodes. Widely known and successful DCAs include grid systems (e.g., Globus [16]), volunteer-computing systems (e.g., BOINC [7]), and MapReduce systems (e.g., Hadoop [17]). DCAs are used extensively for diverse applications, including cryptanalysis [29], web analytics [13], and scientific simulations in fields such as physics [22], bioinformatics [5], and economics [20].

It is imperative that DCAs be able to withstand frequent failures since the entities in their networks are not subjected to any significant dependability checking and malicious entities can easily join the system or compromise other participants. Today's DCAs aim to ensure the correct execution of each task through *voting*: multiple independent worker machines perform the same computation and their results are checked for agreement. This technique is costly, however, as taking a vote among n workers requires expending a factor of n resources or suffering a factor of n slowdown in performance.

In this paper, we propose a new redundancy technique — *iterative redundancy* — that is based on voting but exploits the properties of DCAs to adapt to changing execution environments and improve reliability more efficiently than existing alternatives. More generally, iterative redundancy is applicable to systems that perform computations using a pool of independent processing resources, such that multiple resources have the ability to perform each task and the system may choose, at runtime, among the available resources at random. A key property of iterative redundancy is that it does not require knowing the reliability of the processing resources in the pool, which expands its applicability to systems for which this information cannot be determined. We describe iterative redundancy, formally analyze its cost and performance impacts, and perform a rigorous empirical evaluation on a real-world volunteer-computing system. We compare iterative redundancy to two alternatives:

- *Traditional redundancy*, also called *k-modular redundancy* [21]), which performs $k \in \{3, 5, 7, \dots\}$ independent executions of the same task in parallel and then takes a vote on the correctness of the result.

1063-6927/11 $26.00 © 2011 IEEE
DOI 10.1109/ICDCS.2011.25

- *Progressive redundancy*, which is an adaptation of a related technique from the area of self-configuring optimistic programming research [8].

We demonstrate that iterative redundancy is superior to both traditional and progressive redundancy because iterative redundancy is more *efficient* than both these alternative methods. Iterative redundancy produces the same level of system reliability at a lower cost in employed system resources (or, equivalently, higher reliability at the same cost). In fact, as we argue in Section 3.3, iterative redundancy is optimal with respect to the cost: it is guaranteed to use the minimum amount of computation needed to achieve the desired system reliability.

Finally, we discuss the relationship between iterative redundancy and several other types of redundancy techniques, including active replication [28], primary backup [9], checkpointing [26], and credibility-based fault tolerance [27]. In some cases, iterative redundancy can be used in conjunction with these techniques, as is the case with active replication. In other cases, iterative redundancy can be used in situations where these techniques cannot, as is the case with credibility-based fault tolerance.

The remainder of this paper is organized as follows. Section 2 presents the definitions and assumptions underlying our work. Section 3 describes the three redundancy techniques and provides their theoretical analysis, while Section 4 presents the empirical evaluation and results. Section 5 analyzes several threats to the validity. The paper concludes with an overview of the related work and a recap of our contributions.

2. Definitions and Assumptions

This section defines our model of a DCA, states the threat model we use, and enumerates the assumptions we make to aid the explanation and analysis of iterative redundancy.

2.1. System Model

In this paper, we use the following nomenclature. A *computation* is the typically large problem being solved by a DCA. 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 DCA, 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 a job to the task server, each node rejoins the node pool and can again be selected and assigned a new job. New volunteer nodes may join the pool while other nodes may leave.

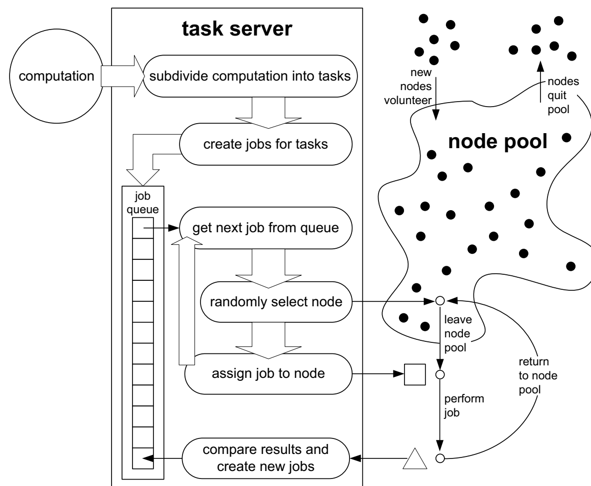


Figure 1: A model of a DCA.

Figure 1 depicts this system model. The model accurately represents a number of DCAs, including the BOINC family of volunteer-computing systems [4], [7]. Section 6 discusses some specific distributed systems to which our techniques apply.

2.2. Threat Model

In this paper, we employ the Byzantine failure model, which is the most general and widely accepted threat model [15], [19], [21], [23] that has been applied to numerous distributed systems [1], [3], [19]. The model includes Byzantine failures and allows for malicious nodes that collude and form cartels to try to mislead and break computations. Byzantine nodes may try to report incorrect results or not report a result at all. For the purposes of this paper, we assume a node that does not report a result in a timely fashion to have failed. There are two important statements to be made about this threat model:

1. Our threat model is at least as strong as those used by redundancy techniques currently deployed in DCAs [4], [10], [13]. 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), all redundancy techniques fail to increase system reliability. On the other hand, we make no assumptions about failures that existing implementations of DCAs do not make. In particular, we assume whether a node fails to be a function of that node, and not of the computation it is performing.

2. 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 a wrong result, but 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.

In a system with voting, the Byzantine failure model can be applied by assuming that the result of every job is one of two possible values. 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.

2.3. Assumptions

In this section, we state five assumptions about the nodes of the network on which a DCA is deployed. These assumptions simplify the description and analysis of the three redundancy techniques, and help to define the class of systems to which the techniques apply. Section 5.3 discusses relaxing these assumptions and demonstrates that iterative redundancy still applies and, in some cases, performs even better on more general networks.

1. Every job sent to the node pool has the same probability of failure because, even though some nodes may be more reliable than others, the jobs are assigned to the nodes at random.

2. The reliability of nodes cannot be determined. This assumption creates a constraint on the redundancy technique, but expands the class of systems to which the technique can be applied.

3. Node failures are independent of each other. That is to say, which nodes fail is independent. However, once nodes do fail, they are allowed to collude, following the Byzantine failure model.

4. The result of every job is one of two possible values (e.g., "yes" or "no"), but the result cannot be easily verified, as in decision NP-complete problems [30]). This assumption is derived from the Byzantine failure model, as described above.

5. 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 three redundancy techniques: the state-of-the-practice traditional redundancy, the state-of-the-art progressive redundancy, and our novel iterative redundancy. To characterize the behavior of each technique, we derive formulae for two measures of their effect on systems: the *system reliability* $\mathbb{R}(r)$ achieved by and the *cost factor* $\mathbb{C}(r)$ of applying the redundancy technique. 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 the somewhat complex formulae for cost and reliability of each technique. As an aid to the reader, Figure 3 provides a graphical depiction of the costs and reliabilities. Further, in Section 4, we verify the formulae's correctness experimentally.

3.1. Traditional Redundancy

The *k-vote traditional redundancy* technique (sometimes called *k-modular redundancy* [21]) performs $k \in \{3, 5, 7, \ldots\}$ independent executions 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. Modern implementations of DCAs, including BOINC [4], [7] and Hadoop [17], rely on traditional redundancy. Figure 2(a) graphically depicts the traditional redundancy algorithm.

Example: Suppose each node's reliability is $r = 0.7$ and $k = 1$ (i.e., there is no redundancy). Then the system distributes just a single job for each task and has the system reliability of 0.7. Using, instead, $k = 19$ 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 $\mathbb{R}_{TR}^k(r)$ and the cost factor as $\mathbb{C}_{TR}^k(r)$. Traditional k-vote redundancy repeats every task k times, independently of r. Thus,

$$\mathbb{C}_{TR}^k(r) = k. \quad (1)$$

The reliability of k-vote traditional redundancy is the probability that at least a consensus of jobs $\left(\frac{k+1}{2}\right)$ does

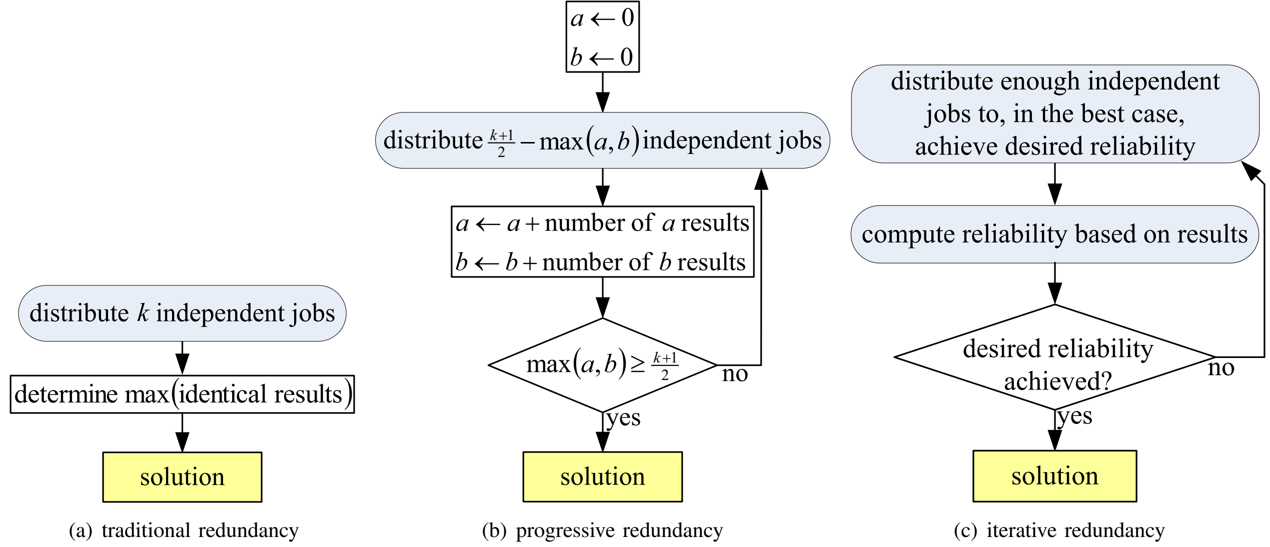


Figure 2: Schematics of (a) traditional, (b) progressive, and (c) iterative redundancy techniques.

not fail, in other words, the sum of the probabilities that only 0, 1, ..., and $\frac{k-1}{2}$ jobs fail. Thus,

$$\mathbb{R}_{TR}^{k}(r) = \sum_{i=0}^{\frac{k-1}{2}} \binom{k}{i} r^{k-i}(1-r)^{i}. \qquad (2)$$

Figure 3 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”) approaches 1 exponentially as the cost factor grows linearly.

3.2. Progressive Redundancy

As part of our research into redundancy techniques, we discovered a self-configuring optimistic programming technique [8] that can be redesigned to apply to DCAs. We have leveraged this scheme to develop *progressive redundancy*. While, to our knowledge, progressive redundancy is not used today in any deployed distributed systems, we introduce it here because, in a sense, it represents a “midway” point between traditional and iterative redundancy, both in terms of the achieved system reliability as a function of cost, and in helping to better explain iterative redundancy to a reader who is familiar only with traditional redundancy.

The key to progressive redundancy is the observation that 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 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 2(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 smaller 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 $\mathbb{R}_{PR}^{k}(r)$ and $\mathbb{C}_{PR}^{k}(r)$ to denote the system reliability and the cost factor, respectively. The cost factor of progressive redundancy 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,

$$\mathbb{C}_{PR}^{k}(r) = \frac{k+1}{2} + \sum_{i=\frac{k+3}{2}}^{k} \sum_{j=i-\frac{k+1}{2}}^{\frac{k-1}{2}} \binom{i-1}{j} r^{i-1-j}(1-r)^{j}. \qquad (3)$$

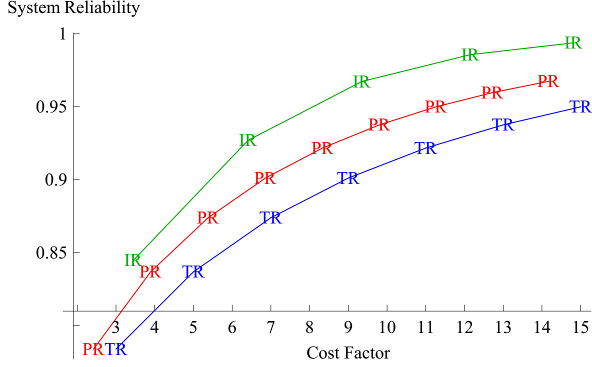


Figure 3: The reliability of a system approaches 1 exponentially, as a function of cost, for traditional (TR), progressive (PR), and iterative (IR) redundancy techniques (here, $r = 0.7$).

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:

$$\mathbb{R}_{PR}^{k}(r) = \sum_{i=0}^{\frac{k-1}{2}} \binom{k}{i} r^{k-i}(1-r)^{i}. \tag{4}$$

Figure 3 shows that for a given cost factor, progressive redundancy (labeled "PR") always achieves a higher system reliability than traditional redundancy.

3.3. Iterative Redundancy

DCAs typically execute jobs asynchronously and have (1) access to runtime information about system reliability and (2) the ability to alter task deployment decisions based on that information. We leveraged this observation to develop *iterative redundancy*.

Iterative redundancy distributes the minimum number of jobs required to achieve a desired confidence level in the result, assuming that all the jobs' results agree. Then, if all jobs agree, the task is completed. However, if some results disagree, the confidence level associated with the majority result is diminished because of the chance that the disagreeing results are correct. In other words, the apparent risk of failure of the task is increased. The algorithm then reevaluates the situation and distributes the minimum number of additional jobs that would achieve the desired level of confidence. This process repeats until the agreeing results sufficiently outnumber the disagreeing results to reach the confidence threshold. Figure 2(c) graphically depicts the progressive redundancy algorithm.

Example: Suppose $r = 0.7$ and the desired system reliability is $\mathbb{R} = 0.97$. Iterative redundancy uses $\mathbb{R}$ as the confidence threshold and calculates how many jobs' results must unanimously agree to be sure of the result's correctness with probability $\mathbb{R}$. 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. Four is the minimum number of jobs that can achieve the confidence threshold in this example, so the task server distributes four jobs. If all four jobs return the same result, the task is finished. However, 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 (with no additional jobs returning the minority result) to achieve $\mathbb{R}$. The task server then 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 to achieve a consensus. In contrast, iterative redundancy is guaranteed to distribute the fewest jobs needed to achieve a *desired system reliability*. Thus far, we have avoided specifying how the technique determines this minimum number of jobs. The basic intuition described above leads to an algorithm that requires (1) numerous relatively complex probability computations and (2) node reliability as an input parameter. Requiring the availability of node reliabilities violates one of the assumptions we stated in Section 2.3. We made this assumption because, for many systems, it is not practical to obtain this information. For example, in a volunteer-computing system, the system has no information about new volunteers. If the system collects information about the reliability of nodes over time, malicious nodes that have developed a bad reputation can change their identity. For iterative redundancy, we have devised an algorithm that does not require knowledge of node reliability and can thus be applied to a wider class of systems than credibility-based fault tolerance and blacklisting [27]. We first describe the naïve, complex algorithm (that requires node reliabilities) and then the simplified algorithm (which does not).

Complex algorithm: Suppose that, of $a+b$ jobs, a return one result with probability r, and b return another result with probability $1-r$. The confidence, denoted $q(r,a,b)$, that the a jobs reported the correct result 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 $\mathbb{R}$ confident in the majority result. We denote that number $d(r,\mathbb{R},b)$. Thus, $d(r,\mathbb{R},b)$ is the minimum a such that $q(r,a,b) \geq \mathbb{R}$. We can compute $d(r,\mathbb{R},b)$ by testing consecutive a values or employing Newton's method [25].

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. Thus, for this example, the algorithm was attempting to achieve a difference of 6 between agreeing and disagreeing results. This phenomenon arises from the somewhat counterintuitive fact (related to Bayes's Theorem), 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 understanding the reasoning is that while the probability of a 106-to-100 decision split occurring may be low, 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 correct. Theorem 1 and its proof formalize this observation.

Theorem 1. *Let $r \in [0..1]$ and $a,b \in \mathbb{Z}_{\geq 0}$. Then $\forall j \in \mathbb{Z}_{\geq 0}$, $q(r,a,b) = q(r,a+j,b+j)$.*

Proof:

$$
\begin{aligned}
q(r,a+j,b+j) &= \frac{r^{a+j}(1-r)^{b+j}}{r^{a+j}(1-r)^{b+j}+(1-r)^{a+j}r^{b+j}} \\
&= \frac{r^j}{r^j}\frac{r^a(1-r)^{b+j}}{r^a(1-r)^{b+j}+(1-r)^{a+j}r^b} \\
&= \frac{(1-r)^j}{(1-r)^j}\frac{r^a(1-r)^b}{r^a(1-r)^b+(1-r)^a r^b} \\
&= \frac{r^a(1-r)^b}{r^a(1-r)^b+(1-r)^a r^b} \\
&= q(r,a,b)
\end{aligned}
$$

□

Theorem 2 exhibits an even stronger notion: no matter what r is, if a (potentially biased) coin is flipped $2b+d$ times and lands heads up $b+d$ times, the probability the coin is biased to land heads more often than tails depends only on d and is independent of b.

Theorem 2. *Let X be a Bernoulli random variable and let $d \in \mathbb{Z}_{\geq 0}$. Then there exists c such that, for all $b \in \mathbb{Z}_{\geq 0}$, if out of $2b+d$ samples of X, exactly $b+d$ are T (and b are F), then the probability that $P(X) \geq \frac{1}{2} = c$.*

Proof: There are two possibilities: either $P(X) \geq \frac{1}{2}$ or $P(x) < \frac{1}{2}$. If $P(X) \geq \frac{1}{2}$, the probability that exactly $b+d$ samples are T is $\binom{2b+d}{b+d}P(X)^{b+d}(1-P(X))^b$. If $P(x) < \frac{1}{2}$, the probability that exactly $b+d$ samples are T is $\binom{2b+d}{b}P(X)^b(1-P(X))^{b+d}$. Then,

$$
\begin{aligned}
&P\left(P(X) \geq \frac{1}{2}\right) = \\
&= \frac{\binom{2b+d}{b+d}P(X)^{b+d}(1-P(X))^b}{\binom{2b+d}{b+d}P(X)^{b+d}(1-P(X))^b+\binom{2b+d}{b}P(X)^b(1-P(X))^{b+d}} \\
&= \frac{P(X)^{b+d}(1-P(X))^b}{P(X)^{b+d}(1-P(X))^b+P(X)^b(1-P(X))^{b+d}} \\
&= \frac{P(X)^d(1-P(X))^b}{P(X)^d(1-P(X))^b+(1-P(X))^{b+d}} \\
&= \frac{P(X)^d}{P(X)^d+(1-P(X))^d}
\end{aligned}
$$

Let c be that value. Note that c does not depend on b. Thus, c is identical for all b. □

Simple algorithm: Using this insight, we can greatly simplify the iterative redundancy algorithm. We only need to determine $d(r,\mathbb{R},0)$ 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 $d(r,\mathbb{R},0) = 6$, 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

```
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 result a
```

Figure 4: The iterative redundancy algorithm.

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 4 specifies the entire iterative redundancy algorithm in pseudocode. Despite being much simpler than and appearing to 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 as defined above, we use $\mathbb{R}^d_{IR}(r)$ and $\mathbb{C}^d_{IR}(r)$ 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,

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

Note that for non-trivial d, $\mathbb{C}^d_{IR}(r) \approx \frac{d}{2r-1}$.

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,

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

Figure 3 shows that for a given cost factor, iterative redundancy (labeled “IR”) always achieves a higher system reliability than both traditional and progressive redundancy.

4. Evaluation

This section analyzes the costs and benefits of the three redundancy techniques. In addition to some formal arguments based on Equations (1) through (6), each analysis includes data from a discrete event simulation of a DCA and a deployment of the BOINC volunteer-computing system [4], [7] on the distributed PlanetLab platform [24].

In Section 4.1, we describe our evaluation platforms for the simulation and BOINC deployments. In Section 4.2, we evaluate the *efficiency* of the techniques.

4.1. Evaluation Platforms

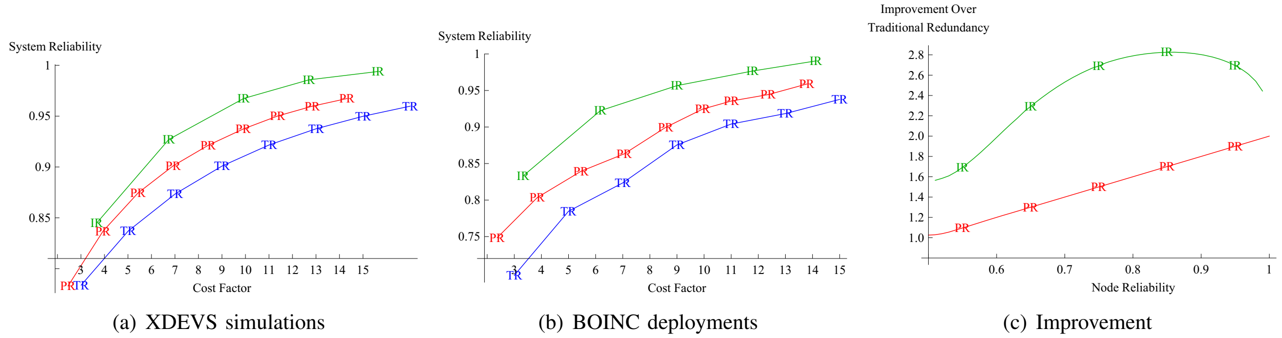
We used off-the-shelf platforms to evaluate the redundancy techniques: XDEVS [14] and BOINC [7].

XDEVS Simulation Environment. The XDEVS simulation framework [14] is a highly extensible discrete event simulator specialized for simulating software systems. The jobs distributed to nodes in our XDEVS simulations 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 enforce constraints on system behavior, ensures that our system model described in Section 2 is accurately represented.

Using XDEVS 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 (1) at least 1,000,000 tasks and 10,000 nodes, (2) job completion times that varied stochastically between 0.5 and 1.5 time units, according to a uniform distribution, and (3) an average node reliability of 0.7.

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 volunteer-computing system [4], [7]. BOINC is a popular DCA currently deployed on over a million machines. Examples of BOINC applications include SETI@home, Folding@home, Malariacontrol.net, and Climateprediction.net. The BOINC server software [7] allows distribution of a custom problem to volunteering computers. To compare the three redundancy techniques, we (1) developed a custom task server that decomposes 3-SAT [30] problems into individual tasks that test whether particular Boolean assignments satisfy a Boolean formula, and (2) modified the job-assignment and result-validation



(a) XDEVS simulations (b) BOINC deployments (c) Improvement

Figure 5: Experimental results from the (a) XDEVS simulations and (b) BOINC deployments, for $r = 0.7$. (c) The ratio improvement in cost factor for progressive (PR) and iterative (IR) redundancy over traditional redundancy varies with r.

procedures to employ iterative and progressive redundancy.

We deployed BOINC on a 200-node subset of PlanetLab [24]. The PlanetLab testbed consists of $\sim$1,000 machines of varying speed and resources, distributed at $\sim$500 locations around the world.

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:

1) seeded failures that caused the wrong result to be returned 30% of the time,

2) PlanetLab nodes becoming unresponsive, and

3) all other unanticipated failures that PlanetLab nodes might experience.

Each execution recorded the time 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

In Section 3, we described our expectations for the performance of the redundancy techniques. In this section, we present empirical data from, first, simulated systems executed in XDEVS and, then, from BOINC systems deployed on PlanetLab to confirm our theoretical predictions.

Figure 5(a) shows empirical data from the XDEVS simulations that supports the claim that iterative redundancy outperforms traditional and progressive redundancy in the number of jobs and time to execute the computation. The data (for $r = 0.7$) closely agrees with our analytical predictions. The exact cost factor improvement of iterative redundancy depends on r.

Figure 5(c) 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 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 the nodes' behavior is highly variable. More precisely, as r increases, the cost $\mathbb{C}_R^k(r)$ to produce a constant increase in $\mathbb{R}_R^k(r)$ decreases linearly for traditional redundancy, but approaches a constant for iterative redundancy. We intend to conduct further experiments to test this hypothesis.

In our next set of experiments, we deployed the redundancy techniques on a BOINC system 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 represents typical real-world deployments. Figure 5(b) 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 5(b) allowed us to estimate the reliability of PlanetLab nodes. The executions consistently reported costs and system reliabilities consistent with $0.64 < r < 0.67$. Seeded faults lowered r to 0.7 and naturally occurring PlanetLab faults were responsible for the difference. The consistency of the derived node reliabilities, among multiple trials with different parameters and across all techniques, provides strong evidence for the validity of the experiments.

5. Threats to Validity

We now discuss several perceived and actual threats to validity. Section 5.1 discusses why predicting the reliability of nodes is difficult and comes at a cost (and that it is a significant advantage that iterative redundancy does not require such predictions). Section 5.2 addresses the issue of response time, which is one measure in which traditional redundancy outperforms iterative and progressive redundancies. Section 5.3 analyzes how relaxing the assumptions we made in Section 2.3 affects our analysis, at times, improving our results.

5.1. Predicting Node Reliability

We have already shown that it is not necessary to estimate r to use iterative redundancy. The user only needs to specify how much improvement is needed (or how high a cost in execution time is acceptable) and the algorithm uses the available resources to achieve the highest possible system reliability. However, in some circumstances, it may be possible to estimate the reliability of the node pool as a whole or the distinct reliability of different classes of nodes and jobs.

Numerous techniques, such as spot-checking of results, blacklisting, and computing node credibility [27], have been proposed as mechanisms to determine the reliability of nodes and utilize that information to improve system reliability. For example, in an attempt to use node reliability knowledge, BOINC has recently added *adaptive replication*, which prevents replication of a task if a trusted node returns its result. However, these techniques have various shortcomings. For example, Byzantine faults cannot be reliably spot-checked, and malicious nodes can earn credibility and fool schemes for rating credibility. Moreover, these techniques incur performance penalties of varying severity. For example, spot-checking requires distributing jobs to which the result is already known, while estimating node credibility requires storing and updating the past behavior of every node. In a large system, these performance costs are significant and we regard iterative redundancy not needing mechanisms for estimating node reliability as a significant advantage.

5.2. Response Time

We have focused on minimizing the jobs needed to to complete computations reliably. However, we have thus far ignored one aspect of iterative redundancy that may be important in some domains. 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, these techniques can increase the response time for a particular task. In the realm of DCAs, the number of tasks is far larger than the number of nodes, so the increased response time does not present a problem because the nodes can always execute jobs related to other tasks [4], [13]. In other words, no node will ever be idle and all nodes processing capability will be fully utilized. 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.

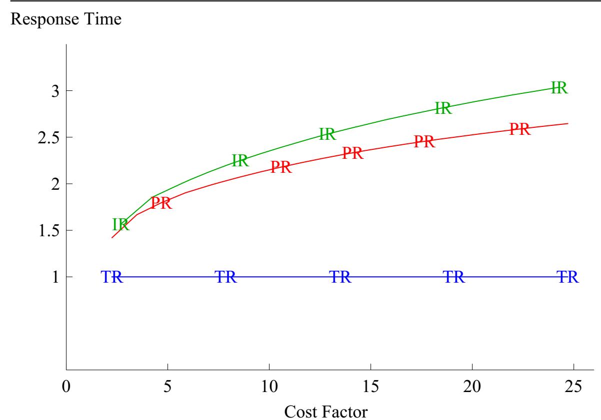


Figure 6: The average response time for tasks using traditional (TR), progressive (PR), and iterative (IR) redundancy.

Figure 6 graphs the average response times for tasks using the three redundancy techniques, as computed by 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.

5.3. Relaxing Assumptions

We made several assumptions, listed in Section 2.3, that helped to clarify how and why iterative redundancy works. We 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. This section explains how redundancy can apply to DCAs deployed on networks without these assumptions, and, in some cases, can even benefit from their relaxation.

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 DCAs (e.g., BOINC [4] and Hadoop [17]) 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. However, for some other types of systems, this assumption might not hold. In these cases, 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 reliability of a particular job c, Equation (3) becomes

$$\mathbb{C}_{PR}^{k} = \frac{k+1}{2} + \sum_{i=\frac{k+3}{2}}^{k} \sum_{j=i-\frac{k+1}{2}}^{\frac{k-1}{2}} \binom{i-1}{j} \prod_{c=1}^{j} r_c \prod_{c=j+1}^{i} (1-r_c)\,.$$

The final cost and probability of failure would then depend on the probability distribution.

We have so far assumed that job failures are independent. However, in some cases, probabilities of job failures 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. If the dependencies among job failure probabilities are known, job schedulers can use the additional information to decrease the probability of failure, using a scheme based on the complex form of the iterative redundancy algorithm or credibility-based fault tolerance [27]. However, if the dependencies are unknown, iterative redundancy can still be used. The analysis of the algorithm would again change as above, with r being replaced with the specific reliabilities of the relevant nodes.

The assumption that the result of every task is a single bit, as in decision NP-complete problems, has simplified our analysis thus far, but it actually turns out to be the worst-case scenario. Compare two types of tasks: 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 redundancy is naturally applicable to systems that perform 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. BOINC uses homogeneous redundancy, an approach that sorts nodes into equivalence classes that report identical answers, to resolve this issue.

6. Related Work

This section contrasts iterative redundancy with existing approaches. Space limitations prevent discussion of the extensive literature in distributed systems.

We based progressive redundancy on a self-configuring optimistic programming technique [8] aimed at component-based systems. Such systems allow for asynchronous job scheduling; however, they focus on minimizing response time and typically allocate finite resources to each task. DCAs relax these limits, which allows iterative redundancy to deploy jobs without a

priori knowledge of node reliability or a bound on the number of jobs.

Primary backup [9] and active replication [28] are two popular redundancy architectures. Primary backup uses multiple servers to improve the reliability of a service — one server designated as primary. 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 [9]. Iterative redundancy complements primary backup by specifying, at runtime, how many backups should exist to guarantee the maximum reliability for a given cost.

Active replication removes the centralized control of primary backup and minimizes losses that occur when some replicas fail. Active replication incurs a high cost associated with keeping all replicas synchronized [28]. Iterative redundancy complements active replication by specifying, at runtime, how many replicas should exist. While primary backup and active replication propose mechanisms for implementing redundancy in distributed systems, iterative redundancy improves the efficiency of those mechanisms.

Credibility-based fault tolerance [27] uses probability estimates to efficiently detect erroneous results submitted by malicious volunteers in volunteer-computing systems. The probability calculations used by credibility-based fault tolerance resemble the complex form of the iterative redundancy algorithm. However, credibility-based fault tolerance does not incorporate our simplifying insight that allows the algorithm to function without any estimates of node reliability. As a result, credibility-based fault tolerance is forced to rely on spot-checking with blacklisting. However, Byzantine faults cannot be reliably spot-checked, and malicious nodes can earn credibility and fool schemes for rating credibility.

Hwang and Kesselman [18] proposed a method for injecting fault tolerance into grids that handles a wide variety of faults within distributed systems. This work uses a service to detect crash failures (and an extension to allow the system designer to specify how to detect other failures) and a failure-handling framework that enforces designer-defined policies [18].

Traditional checkpoint techniques can also be applied to DCAs 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 [26]. Further, using checkpoints and replication together can reduce the number of replicas needed to detect Byzantine failures [2] over what the standard Byzantine agreement protocols [28] require.

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 [12].

7. Contributions

We presented a novel technique, iterative redundancy, that improves on existing reliability techniques by leveraging runtime information. We identified several types of systems to which iterative redundancy applies and concentrated in this paper on DCAs. Iterative redundancy is more efficient than existing methods in its use of resources. In addition to a rigorous theoretical analysis of iterative redundancy, we verified iterative redundancy's efficiency with an empirical evaluation based on two deployments: the XDEVS discrete event simulator and the BOINC volunteer-computing system.

Acknowledgments

This material is supported by the National Science Foundation under Grant numbers 0820170, 0905665, and 0937060 to the Computing Research Association for the CIFellows Project.

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] A. Agbaria and R. Friedman, "A replication- and checkpoint-based approach for anomaly-based intrusion detection and recovery," in *Proceedings of the Second International Workshop on Security in Distributed Computing Systems (SDCS05)*, 2005, pp. 137–143.

[3] 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.

[4] 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.

[5] 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.

[6] S. A. Baset and H. Schulzrinne, “An analysis of the skype peer-to-peer Internet telephony protocol,” in *Proceedings of the 25th Conference on Computer Communications (IEEE Infocom06)*, Barcelona, Spain, 2006.

[7] BOINC, “The Berkeley open infrastructure for network computing,” http://boinc.berkeley.edu, 2009.

[8] A. Bondavalli, S. Chiaradonna, F. D. Giandomenico, and J. Xu, “An adaptive approach to achieving hardware and software fault tolerance in a distributed computing environment,” *Journal of Systems Architecture*, vol. 47, no. 9, pp. 763–781, 2002.

[9] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, “The primary-backup approach,” in *Distributed Systems*, 2nd ed., 1993, pp. 199–216.

[10] 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.

[11] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, “Freenet: A distributed anonymous information storage and retrieval system,” in *International Workshop on Design Issues in Anonymity and Unobservability*, 2001, pp. 46–66.

[12] 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.

[13] 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, 2004.

[14] 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.

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

[16] “The Globus alliance,” http://www.globus.org, 2005.

[17] “Hadoop,” http://hadoop.apache.org, 2009.

[18] 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, 2003.

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

[20] 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)*, 1990, pp. 1–6.

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

[22] 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.

[23] L. Lamport, R. Shostak, and M. Pease, “The Byzantine generals problem,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, pp. 382–401, 1982.

[24] 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.

[25] 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.

[26] 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.

[27] L. F. G. Sarmenta, “Sabotage-tolerance mechanisms for volunteer computing systems,” *Future Generation Computer Systems*, vol. 18, no. 4, pp. 561–572, 2002.

[28] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” *ACM Computing Surveys*, vol. 22, pp. 299–319, 1990.

[29] A. Setiawan, D. Adiutama, J. Liman, A. Luther, and R. Buyya, “GridCrypt: High performance symmetric key using enterprise grids,” in *Proceedings of the 5th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT04)*, Singapore, 2004.

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