Brokering Algorithms for Composing Low Cost Distributed Storage Resources

Jimmy Secretan, Malachi Lawson and Ladislau Bölöni
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816

Abstract *In this paper we investigate the composition of cheap network storage resources to meet specific availability and capacity requirements. We show that the problem of finding the optimal composition for availability and price requirements can be reduced to the knapsack problem, and propose three techniques for efficiently finding approximate solutions. These include a dynamic programming algorithm, a heuristic and an genetic algorithm. The algorithms can be implemented on a broker that intermediates between buyers and sellers of storage resources. Finally, we show that a broker in an open storage market, using the combination of the three algorithms can more frequently meet user requests and lower the cost of requests that are met compared to a broker that simply matches single resources to requests.*

Keywords: Distributed Storage, Grid Computing

1 Introduction

Currently there is an abundance of idle storage resources, connected to the Internet. To make use of these idle resources, and to create opportunity for commercial providers to bring new storage services online, an open marketplace is needed to help connect buyers to sellers as well as to establish prices. This market approach to grid computing is beginning to emerge as a popular paradigm in the literature. A buyer in this economy would like to meet his availability and capacity requirements at the lowest cost. However, there might not be an exact match for the requirements of the users in the pool of available resources. To solve this problem, we propose a model in which the user assembles a resource of desired capacity and availability by combining resources which, individually, do not satisfy the user's requirements.

Note that a similar technique is used in local resources such as RAID arrays. The various RAID levels such as RAID0 (striping), RAID1 (mirroring), and so on are just different ways of combining storage resources to meet the storage, reliability, performance and cost demands. Our approach extends this model to a dynamic networked environment.

Our objective is to design a broker-based architecture for the efficient allocation of the resources. As the optimal composition problem has a non-polynomial complexity, we are interested in finding efficient approximate algorithms, with modest computational requirements.

This paper is organized as follows. In section 2, we discuss current work related to architectures and economics for distributed storage. In section 3, we introduce a notation to express composition of storage resources based on availability, capacity and cost. In section 4, we analyze the complexity involved in these resource optimization problems. In section 5, we setup various resource allocation experiments to test our algorithms. In section 6, we present the results of these experiments and discuss the significance of these results. Finally, in section 7, we discuss future enhancements to our work.

2 Related Work

Economic models for grid resource allocation will be beneficial or even necessary for data and processing grids to become common place. In [1], and [2], the authors present two complementary systems for economic resource allocation on the grid. The CPM (Compute Power Market) is intended to apply to low-end systems, while GRACE (Grid Architecture for Computational Economy), is intended for high end grid computing. Oceanstore [3] aims to build a data storage infrastructure on untrusted servers, using both redundancy and cryptography. It is built around a market concept whereby users would pay a monthly fee for persistent and reliable storage, supported by various storage providers. The prototype implementation of Oceanstore employs both Reed-Solomon and Tornado algorithms for encoding redundancy. Finally, the SX (Storage eXchange) system in [4] acts as a storage broker, allowing storage to be a tradeable resource. SX uses a double auction market model for open market trading and also allows storage to be exchanged. The system brokers storage requests by taking into account capacity, upload/download rate, and time frame of the storage reservation. The authors cite many other criteria that should be taken into account, including security, high-availability, fault-tolerance, reputation, consistency and operating environment.

3 Resource Composition

In the following, we define an algebra for resource composition. The two main attributes of a (simple or composed) storage resource are the capacity and availability. We define availability as the probability of a successful resource access. We assume that the availability of the individual resources is known to the broker (for instance, by counting the ratio of successful accesses to the total number of accesses). After some threshold time of unavailability, the user's storage system will be forced to re-allocate. For a storage resource request R_{req}, the brokering system aims to use the distinct storage resources of which it keeps track, R_i where $1 \leq i \leq n$, to meet the request. The broker may meet the request R_{req} either with one of the resources R_i, or with R_{comp}, a new resource composed from d distinct R_i's. The user specifies a capacity $C(R_{req})$ and an availability $A(R_{req})$ which he requires for his particular application. The broker suggests one of four ways to satisfy the user's request. First, it may suggest a specific resource R_i, that suits the user's needs. The other three approaches are based on combining several resources through additive composition ($AC(R_1...R_i)$), redundant composition ($RC(R_1...R_i)$) or distributed error correction ($DEC(R_1...R_i)$) to produce an R_{comp}. We summarize the notations in table 1.

Additive composition combines two or more smaller resources into one larger one. All of the involved resources are required to be available for the full storage resource to be considered available. The capacity and availability of an additively composed resource $R_{comp} = AC(R_1...R_i)$ is given by the following formulas:

Table 1: Notations for Analysis

Notation	**Description**
R	Set of storage resources, each labeled R_i where $1 \leq i \leq n$. Each R_i storage resource includes a capacity, an availability and a price. There is only one resource per network entity even though it may be divisible.
$C(R_i)$	Gives the storage capacity of the resource R.
$A(R_i)$	Gives the availability probability of resource R, $0 \leq A(R) \leq 1$
$P(R_i)$:	Price per unit of storage for a storage resource.
R_{req}	A description of the resource required by the user.
R_{comp}	The resource being composed by the broker for the requesting user.
d	Number of resources chosen to be part of R_{comp}.
n	Number of resources accounted for in the broker.
$AC(R_1...R_i)$	The result of additive composition of resources R_1 through R_i.
$RC(R_1...R_i)$	The result of redundant composition of resources R_1 through R_i.
$DEC(R_1...R_i)$	The result of distributed error composition of R_1 through R_i.

$$C(R_{comp}) = \sum_{i=1}^{d} C(R_i) \geq C(R_{req})$$
$$A(R_{comp}) = \prod_{i=1}^{d} A(R_i) \geq A(R_{req})$$

If we wish to additively compose a resource, the sum of the storage in the resources must be greater than or equal to our desired storage, and the availabilities multiplied together must be greater than or equal to our required availability.

Redundant composition allows us to take two or more resources that are sufficiently large, and put them together redundantly. This is used whenever the user requires higher availability than is offered by the currently available storage resources. Therefore, if we have N-modular redundancy, only one of the resources is required to work for a user to be able to access his data. When composing a redundant storage resource, $R_{comp} = RC(R_1...R_i)$, The capacity is constrained and the availability is given by

$$C(R_i) \geq C(R_{req}) = C(R_{comp})$$
$$A(R_{comp}) = 1 - \prod_{i=1}^{d} (1 - A(R_i)) \geq A(R_{req})$$

It should be noted that the only the needed storage portion can be separated from $R_1...R_i$, leaving the rest for other allocations.

Finally, *Distributed Error Correction (DEC)* based storage can be used. This is analogous to the RAID5 disk drive composition method. For this model, we must have at least 3 resources. All but the last resource stores some portion of the data. The last resource stores the XOR of all of the other stores. In this way, the system can lose any one resource and still maintain the user's data. For availability, we only allow the possibilities of all or all but one of the resources being unavailable. Therefore, composing a DEC resource, $R_{comp} = DEC(R_1...R_i)$, with d individual resources, we require that

$$C(R_i) \geq C(R_{req})/(d-1)$$
$$A(R_{comp}) =$$
$$\prod_{i=1}^{n} A(R_i) + \sum_{i=1}^{d}(1 - A(R_i)) \prod_{j=1}^{n,j\neq i} A(R_j) \geq A(R_{req})$$

Suppose a user is looking for a storage resource of arbitrary size and availability, for the lowest possible price. This, then, becomes the central challenge of the proposed system.

We show that the problem of finding the optimally priced additively and redundantly composed storage resources, can be reduced to the classical knapsack problem. The classical problem is cast as follows. There are several items i, numbered from 1 to n to be placed in a knapsack of capacity c. Each item has an associated price, p_i and an associated weight, w_i. We intend to maximize the value of the knapsack contents while staying within the capacity. If we assume that an item must be taken or not taken, the problem is then referred to as the 0-1 Knapsack problem. For integer knapsack capacity, a pseudo-polynomial approach does exist that uses dynamic programming, taking $O(nc)$ time.

However, suppose that, in addition to not exceeding the capacity of the sack, one must not exceed the a dimension of length either. This is referred to as the 2 dimensional knapsack problem. This can also be extended to an arbitrary dimensionality, becoming the d-dimensional knapsack problem.

Redundant composition is the easiest to reduce to the knapsack problem. The items are the storage resources, R_i, and the price p_i is instead $P(R_i)$. The capacity c is represented by $C(R_{req})$ and the weights are represented by $A(R_i)$.

First, we select all sellers that have $C(R_i) \geq C(R_{req})$. First, we rewrite the equation of availability for redundant composition as:

$$\prod_{i=1}^{d}(1 - A(R_i)) \leq (1 - A(R_{req}))$$

For reasons that will become clear later, we need both sides to be greater than 1. A good way to guarantee this is by multiplying by $\frac{1}{(1-min(A(R_i)))^d}$

This yields:

$$\prod_{i=1}^{d} \frac{1-A(R_i)}{(1-A(R_{min}))^d} \leq \frac{1-A(R_{req})}{(1-A(R_{min}))^d}$$

Now, we intend to turn this into a linear weight function. We do this by taking the natural logarithm. Then we add a term x_i either equal to 0 or 1, depending on whether the resource is included, and sum over all n instead of d. This gives:

$$\sum_{i=1}^{n} x_i ln(\frac{1-A(R_i)}{(1-A(R_{min}))^d}) \leq ln(\frac{1-A(R_{req})}{(1-A(R_{min}))^d})$$

Now this has become the 0/1 knapsack problem, with a weight function where $w_i = ln((1 - A(R_i))/(1 - A(R_{min}))^d)$ and $c = ln(((1 - A(R_{req}))/(1 - A(R_{min}))^d))$.

If we try sufficient numbers of d from 2 to n, and use the aforementioned dynamic programming method, this is solvable in pseudo-polynomial time, $O(n^2 \log \frac{1-A(R_{req})}{(1-A(R_{min})^n)})$.

We find an analogous problem for the case of additive composition. In this case, the resources that are selected, when put together, must meet or exceed $C(R_{req})$. We also must not exceed the availability constraints. As stated before, the availability that must be met is

$$\prod_{i=1}^{d} A(R_i) \geq A(R_{req})$$

By taking the reciprocal and the natural logarithm, we yield an arrangement that is compatible with the knapsack weight equation, as before:

$$\sum_{i=1}^{d} ln(\frac{1}{A(R_i)}) \leq ln(\frac{1}{A(R_{req})})$$

However, we must add the additional constraint that

$$\sum_{i=1}^{d} C(R_i) \geq C(R_{req})$$

Again, this yields a 2-dimensional knapsack problem, which is not easily solvable.

Finally, the most difficult problem is the DEC arrangement, which is complex integer programming problem, not easily tractable.

4 Allocation Algorithms

The system is designed around a centralized broker that serves as the marketplace for buyers and sellers to meet. As nodes with resources to sell first come online, they register themselves with the broker according to a unique, persistent ID. This node's ID is used by the broker to keep track of whether or not it is currently online, how much its resource asking price is, how much of a resource it currently has, its history of uptime, and its transaction history. A buyer, also with a unique account ID, then approaches the broker, looking for a resource with specific capacity and availability. The broker then searches its list of available sellers.

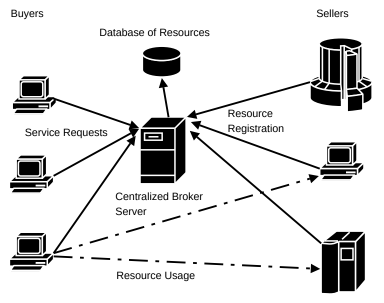

Figure 1: Design of the system. The sellers may be home PC users, companies/organizations or dedicated storage providers. Solid lines represent communication related to the brokering process, between buyers, sellers and the broker. The interrupted lines represent the peer to peer usage of the storage resources.

The pseudo-code for the general broker's algorithms is presented in figure 2. The kth user submits a request R^k_{req}. By executing the appropriate functions, the broker returns $SingleResource$, $RCResource$, $ACResource$, $DECResource$, which are total allocations, made with their respective algorithms, with prices P_{single}, P_{AC}, P_{RC}, P_{DEC} respectively. These total allocations are composed of several individual resource allocations. An individual resource allocation is represented by a two-tuple, $I_i = (R_i, C^i_{alloc})$, with R_i representing the seller resource from which it comes, and C^i_{alloc} representing a storage amount actually allocated from R_i. To find the cheapest single resource, $SingleResource$, the FINDLOWESTSINGLE function is used. It simply does a linear search through the list of sellers to find the seller with the lowest price that still meets the minimum requirements. FINDLOWESTRC implements the standard dynamic programming solution for solving the 0/1 knapsack problem, in order to solve the transformed redundant composition problem. SORTBYDIFFICULTY sorts the list of currently queued requests by the difficulty criterion, that is $C(R^k_{req})/(1 - A(R^k_{req}))$ with easiest requests first. The final lowest cost resource found is referred to as $LowestCostResource$ with price P_{lowest}.

```
SortByDifficulty($R_{req}$);
foreach $R^k_{req}$ do
    $SingleResource$, $P_{linear}$ = FindLowestSingle($R_i$, $R^k_{req}$);
    $ACResource$, $P_{AC}$ = FindLowestAC($R_i$, $R^k_{req}$);
    $RCResource$, $P_{RC}$ = FindLowestRC($R_i$, $R^k_{req}$);
    $DECResource$, $P_{DEC}$ = FindLowestDEC($R_i$, $R^k_{req}$);
    $LowestCostResource$, $P_{lowest}$ = FindLowest($SingleResource$, $P_{linear}$,
    $ACResource$, $P_{AC}$, $RCResource$, $P_{RC}$,$DECResource$, $P_{DEC}$);
end
```

Figure 2: Pseudocode for the broker's general algorithm.

The broker's algorithm searches through the available resources, as well as the possible composed resources, that optimally meet the users requirements. We now present each of the search functions in turn.

As mentioned before, finding the optimal lowest cost additively composed configuration is a very hard two dimensional knapsack problem. Therefore, we design a heuristic that takes into account knowledge of the domain. It is known that all sellers with $A(R_i) < A(R_{req})$ can be automatically excluded, because they would immediately cause the availability of the allocation to drop below $A(R_{req})$. After that, resources are added with regard to the AC-criterion, which gives priority to sellers that would likely be more favorable. Because the availability quickly drops as resources are additively composed, priority needs to be given to resources that have a high availability per unit price. Also, priority should be given to larger stores, because fewer of them are needed, resulting in less decrease in availability. However, this is only important insofar as it is as large as the space required. The AC criterion is therefore defined to be: $\sigma(R_i, R^k_{req}) = \frac{A(R_i)min(C(R_i),C(R^k_{req}))}{C(R_{req})P(R_i)}$. In the following pseudocode, FINDALLGREATEROREQUAL simply returns a list of sellers with availability greater than or equal to the specified availability and SORTBYSTRIPINGCRITERION sorts the list descending by the calculated σ. $currentAllocation$ is a temporary variable for a set of individual resource allocations.

The loop through each of the R_i is an $O(n)$ process, while the sort is an $O(n \log n)$ process, making

```
Procedure: FindLowestAC
($R_i$, $R^k_{req}$)
$sellersWithMinA$ = FindAllGreaterOrEqual($R_i$, $A(R_{req})$);
SortByStripingCriterion($sellersWithMinA$, $C(R^k_{req})$);
real $accumulatedSpace = 0$;
real $accumulatedAvailability = 1$;
boolean $done$ = false;
boolean $valid$ = true;
$currentAllocation = \emptyset$;
while $\exists R_i \in R$ and $done$ != true do
    if $accumulatedSpace + C(R_i) \geq C(R^k_{req})$ then
        $done$ = true;
    end
    if $accumulatedAvailability * A(R_i) < A(R^k_{req})$ then
        $done$ = true;
        $valid$ = false;
    end
    if $valid$ then
        currentAllocation = currentAllocation $\cup$ I;
        $accumulatedSpace$ += $C(R_i)$;
        $accumulatedAvailability$ *= $A(R_i)$;
    end
end
return currentAllocation;
```

Figure 3: Heuristic algorithm for finding an additively composed set of resources.

this algorithm work in $O(n \log n)$.

As stated before, finding the optimal distributed error composition configuration a complex integer programming problem. However, there are a number of meta-heuristics to which we can turn. Tabu search and genetic algorithms have both been successfully applied to knapsack problems [5]. While this is a harder problem to solve than the traditional knapsack problem, it is relatively easy to frame in terms of a genetic algorithm. We form our chromosome with a binary gene for each available seller. The GA then tries to maximize a fitness function which we have engineered to reward solutions that meet the minimum size, whose sellers have the minimum required space and solutions that meet the requested availability. The fitness function is as follows:

$$Fitness(DEC) = \frac{D_1}{A(R_{req})}(\min A(DEC), A(R_{req}) + D_2(1 - \frac{d_{in}}{d}) + D_3(ProperNumber(d)) + \frac{D_4}{P}$$

where D_1, D_2, D_3 and D_4 are positive constants, P is the price per unit of the entire allocation, d is the number of total sellers in the allocation and d_{in} is the number of resources with insufficient space to be part of the allocation.

We also introduce two functions:

$$ProperNumber(d) = \begin{cases} 1 & 3 \leq d \leq DEC\_MAX \\ 0 & otherwise \end{cases}$$

Where $ProperNumber$ rewards solutions that are within a probably range of valid DEC solutions. DEC_MAX (in our case 10) is a parameter of the broker. It is chosen as a practical limit of the number of different R_i resources that can be part of R_{req}. The the number of storage resources d that are part of R_{req} could conceivably be as large as n, but we use DEC_MAX to find practical solutions and computationally simplify the task. Because this GA can produce individuals that are not valid solutions, they are checked for validity before being added to the list of possible solutions.

5 Experimental Setup

The clearing of a broker with both the standard allocation algorithm (linear search of single resources to meet requests) and our improved algorithm (standard allocation search plus the three improved searches) was evaluated using a Java simulation. It was assumed that several requests as well as several producers were queued up by the broker, and cleared all at once. This evaluates not only the ability of the algorithms to make a single allocation more efficient, but also their ability to make the whole set of allocations more efficient. Each clearing included 200 sellers and 50 buyers.

The parameters that were swept in the simulation included the average storage and availability requested. These parameters allowed the simulation to generate a set of requests whose requested storage and requested availabilities were normally distributed around the provided average storage and availability.

Data for the storage providers were taken from the research done in [6]. In this paper, the authors provided some statistical summaries of the host attributes for hosts that participate in the SETI@Home project. In order to generate the list of sellers to be used in the simulation, 50,000 random values of free disk space on the hosts (referred to as d_free in the host database) were selected from the database of SETI@Home hosts. Unfortunately, there is no host availability data provided on per-host basis by the publicly available host database. Therefore, using summary statistics provided in [6], we attempted to generate some reasonable values. The aforementioned paper gives three important variables to calculate the overall availability of the host. The first variable is $on\_fraction$. This is the fraction of time that the SETI@Home client runs. The next variable of interest is $connected\_fraction$. This is the amount of time that the client program has access to an Internet connection. Finally, there is $active\_fraction$ which indicates how much of the time that the client is allowed to run. This is because the client is set to stay inactive when the host PC is otherwise busy. For our application we consider the average availability to be the product of these three variables: $\overline{A} = (on\_fraction)(connected\_fraction)(active\_fraction) = (0.81)(0.83)(0.84) = 0.5647$ Using this average as the average of a Gaussian random distribution with a standard deviation of 0.1, availability values were generated to paired with randomly selected disk space values from the SETI@Home host database. Finally, prices per Gigabyte of space were generated for each resource pair R_i. We assume that the pricing per unit of storage space is dictated by the availability, which becomes asymptotically more expensive as availability approaches one:

$$P \sim 1/(1 - A(R_i))$$

To provide random variations in pricing, this calculated price is multiplied by a normally generated random price factor with average of 1.0 and standard deviation of 0.1. For each clearing iteration within a run with particular values for the two parameters, 200 producers were randomly selected from the loaded seller pool, and 50 consumers were randomly generated. For each set of parameters, this clearing iteration was performed 10 times and averaged. The consumers were randomly generated with average requested availabilities ($\overline{A}(R_{req})$) in the range of 0.3 to 0.99 and average requested capacity ($\overline{C}(R_{req})$) of 1GB, 10GB, 100GB and 300GB. With these parameters, consumer requests were generated by Gaussian random generators, with the given averages and standard deviations of $\frac{\overline{A}(R_{req})}{8}$ and $\frac{\overline{C}(R_{req})}{2}$ for the required availability and capacity respectively. These were chosen under the assumption that the availability required by users would not vary as widely as the space required (most users want "pretty good" availability). The GA responsible for finding Distributed Error Compositions run for a 100 generations with a population size of 100. The default JGAP mutation rate of 0.1 was used. The constants for the fitness function were, $D_1 = 25.0, D_2 = 25.0, D_3 = 200.0, D_4 = 50.0$.

6 Results

The percentage of allocation successes are shown in Figure 4. Both the original search and the new composition method are overlaid. A similar graph is shown for pricing in Figure 5. Figure 6 shows the percentage of the time the specified allocation type was chosen as the lowest cost allocation fulfilling the user's needs.

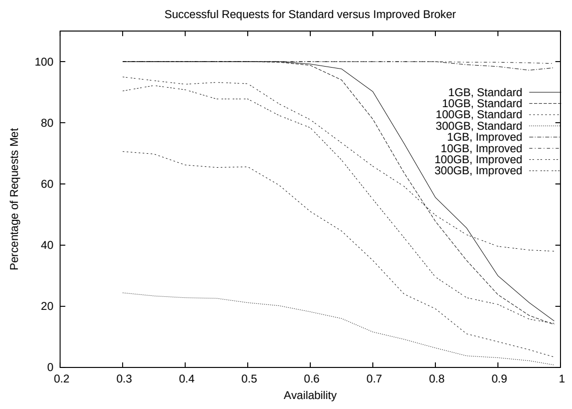

Figure 4: Percentage of successful allocations, using single resources and multiple resources.

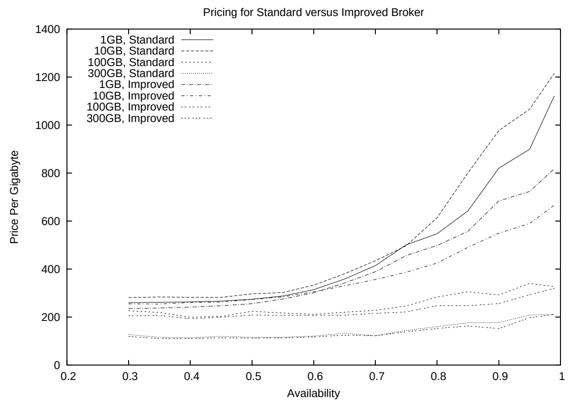

Figure 5: Prices for allocations with single and multiple resources.

We find that giving the broker the ability to compose new resources based on multiple available ones increases its ability to serve user demands, as seen in Figure 4. The way the experiments are, the new method must perform at least as well as the old method, because the new method subsumes the old one. As $A(R^k_{req})$ becomes larger, the new method has a distinct advantage, as the prospect of finding a single resource to meet the user's requirements becomes less and less likely.

Figure 5, shows that the pricing benefit of the algorithm improves as availability required increases. When the broker has the ability to compose resources with high availability from resources with

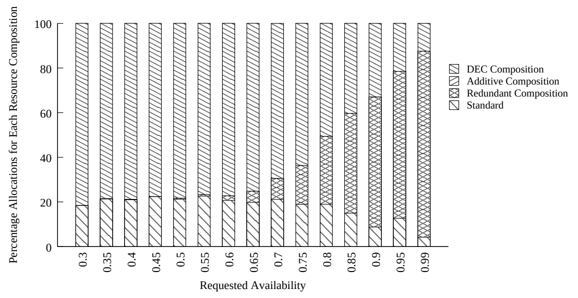

Figure 6: Percentage of compositions for $\overline{C}(R_{req}) = 300.0GB$

lower availability, it allows the broker to satisfy the users' request with cheaper resources. Again, the lowest cost resource is chosen, so we must at least meet the standard method.

These algorithms are also effective at dispatching a queue of requests to be cleared, because on average, the requests are still dealt with more efficiently.

As expected, the percentage of a particular resource composition is greater on certain parameters than others. Figure 6 shows, an example 300GB allocations. Standard allocations at low availability can be cheaper, and both DEC and standard allocations require less redundancy (and therefore, potentially less cost) than redundant composition. This makes them dominant at low required availabilities. As availability requirements are more stringent, anything but redundant composition has less and less chance of meeting it. Mostly absent from Figure 6 are additive compositions. Most of the time, additive compositions were not chosen at all, and they were chosen a maximum of a few percent of the time (not shown). This is because the availability of an additively composed resource declines rapidly, and the average availability of the seller host pool is already somewhat low.

7 Conclusions/Future Work

In this paper we proposed an approach to combine cheap network storage resources to achieve the desired availability and capacity requirements of consumers. We found that although the optimal allocation problem is complex, the combination of several approximation techniques can be used by the storage brokers improve the service provided to the users. Our future work involves both improving the performance of the proposed solutions, as well as extending them to a more general problem. An immediate extension, but on that is considerably more difficult, is to consider the network bandwidth through which the storage is available.

Acknowledgment

This work was supported in part by a National Science Foundation Graduate Research Fellowship as well as National Science Foundation grants CRCD: 0203446, CCLI: 0341601, DUE: 05254209, and IIS: 0647120. The authors would also like to thank Advanced Micro Devices, Inc. (AMD) for their generous donation of equipment used in this research.

References

[1] R. Buyya and S. Vazhkudai, "Computer power market: Towards a market-oriented grid," in *The 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, May 2001, pp. 574–581.

[2] R. Buyya, D. Abramson, and J. Giddy, "An economy driven resource management architecture for global computational power grids," in *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, June 2000.

[3] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000, pp. 190–201.

[4] M. Placek and R. Buyya, "Storage exchange: A global trading platform for storage services," in *Proceedings of the 12th International European Parallel Computing Conference (EuroPar 2006)*. Springer-Verlag, August 2006.

[5] P. Chu and J. Beasley, "A genetic algorithm for the multidimensional knapsack problem," *Journal of Heuristics*, vol. 4, no. 1, pp. 63–86, June 1998.

[6] D. P. Anderson and G. Fedak, "The computational and storage potential of volunteer computing," in *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, May 16-19 2006, pp. 73–80.