A Network-Aware Distributed Storage Cache for Data Intensive Environments[1]

Brian L. Tierney, Jason Lee, Brian Crowley, Mason Holding

Computing Sciences Directorate
Lawrence Berkeley National Laboratory
University of California, Berkeley, CA, 94720

Jeremy Hylton, Fred L. Drake, Jr.
Corporation for National Research Initiatives, Reston, VA 20191

Abstract

Modern scientific computing involves organizing, moving, visualizing, and analyzing massive amounts of data at multiple sites around the world. The technologies, the middleware services, and the architectures that are used to build useful high-speed, wide area distributed systems, constitute the field of data intensive computing. In this paper we will describe an architecture for data intensive applications where we use a high-speed distributed data cache as a common element for all of the sources and sinks of data. This cache-based approach provides standard interfaces to a large, application-oriented, distributed, on-line, transient storage system. We describe our implementation of this cache, how we have made it "network aware," and how we do dynamic load balancing based on the current network conditions. We also show large increases in application throughput by access to knowledge of the network conditions.

1.0 Introduction

High-speed data streams resulting from the operation of on-line instruments and imaging systems are a staple of modern scientific, health care, and intelligence environments. The advent of high-speed networks is providing the potential for new approaches to the collection, organization, storage, analysis, visualization, and distribution of the large-data-objects that result from such data streams. The result will be to make both the data and its analysis much more readily available.

For example, health care imaging systems illustrate the need for both high data rates and real-time cataloging. Medical video and image data used for diagnostic purposes — e.g., X-ray CT, MRI, and cardio-angiography — are collected at centralized facilities and may be accessed at locations other than the point of collection (e.g., the hospitals of the referring physicians). A second example is high energy physics experiments, which generate high rates and massive volumes of data that must be processed and archived in real time. This data must also be accessible to large scientific collaborations — typically hundreds of investigators at dozens of institutions around the world.

In this paper we will describe how "Computational Grid" environments can be used to help with these types of applications, and how a high-speed network cache is a particularly important component in a data intensive grid architecture. We describe our implementation of a network cache, how we have made it "network aware," and how we adapt its operation to current network conditions.

2.0 Data Intensive Grids

The integration of the various technological approaches being used to address the problem of integrated use of dispersed resources is frequently called a "grid," or a computational grid — a name arising by analogy with the grid that supplies ubiquitous access to electric power. See, e.g., [10]. Basic grid services are those that locate, allocate, coordinate, utilize, and provide for human interaction with

1. The work described in this paper is supported by DARPA, Information Technology Office (http://www.darpa.mil/ito/ResearchAreas.html) and the U. S. Dept. of Energy, Office of Science, Office of Computational and Technology Research, Mathematical, Information, and Computational Sciences Division (http://www.er.doe.gov/production/octr/mics/index.html), under contract DE-AC03-76SF00098 with the University of California. This is report no. LBNL-42896.

the various resources that actually perform useful functions.

Grids are built from collections of primarily independent services. The essential aspect of grid services is that they are uniformly available throughout the distributed environment of the grid. Services may be grouped into integrated sets of services, sometimes called "middleware." Current grid tools include Globus [8], Legion [16], SRB [3], and workbench systems like Habanero [11] and WebFlow [2].

From the application's point of view, the Grid is a collection of middleware services that provide applications with a uniform view of distributed resource components and the mechanisms for assembling them into systems. From the middleware systems points of view, the Grid is a standardized set of basic services providing scheduling, resource discovery, global data directories, security, communication services, etc. However, from the Grid implementor's point of view, these services result from and must interact with a heterogeneous set of capabilities, and frequently involve "drilling" down through the various layers of the computing and communications infrastructure.

2.1 Architecture for Data Intensive Environments

Our model is to use a high-speed distributed data storage cache as a common element for all of the sources and sinks of data involved in high-performance data systems. We use the term "cache" to mean storage that is faster than typical local disk, and temporary in nature. This cache-based approach provides standard interfaces to a large, application-oriented, distributed, on-line, transient storage system.

Each data source deposits its data in the cache, and each data consumer takes data from the cache, often writing the processed data back to the cache. A tertiary storage system manager migrates data to and from the cache at various stages of processing. (See Figur e1.) We have used this model for data handling systems for high energy physics data and for medical imaging data. For more information see [15] and [14].

The high-speed cache serves several roles in this environment. It provides a standard high data rate interface for high-speed access by data sources, processing resources, mass storage systems (MSS), and user interface / data visualization elements. It provides the functionality of a single very large, random access, block-oriented I/O device (i.e., a "virtual disk"). It serves to isolate the application from tertiary storage systems and instrument data sources, helping eliminate contention for those resources

This cache can be used as a large buffer, able to absorb data from a high rate data source and then to forward it to a

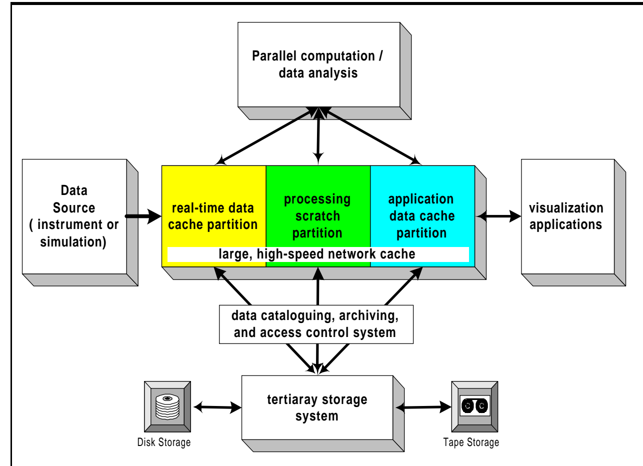

Figure 1 The Data Handling Model

slower tertiary storage system. The cache also provides an "impedance matching" function between a small number of high throughput streams to a larger number of lower speed streams, e.g. between fine-grained accesses by many applications and the coarse-grained nature of a few parallel tape drives in the tertiary storage system.

Depending on the size of the cache relative to the objects of interest, the tertiary storage system management may only involve moving partial objects to the cache. In other words, the cache may contain a moving window for an extremely large off-line object/data set. Generally, the cache storage configuration is large (e.g., 100s of gigabytes) compared to the available disks of a typical computing environment (e.g., 10s of gigabytes), and very large compared to any single disk (e.g. hundreds of ~10 gigabytes).

2.2 Network-Aware Applications

In order to efficiently use high-speed wide area networks, applications will need to be "network-aware"[6]. Network-aware applications attempt to adjust their demands in response to changes in resource availability. For example, emerging QoS services will allow network-aware applications to participate in resource management, so that network resources are applied in a way that is most effective for the applications. Services with a QoS assurance are likely to be more expensive than best-effort services, so applications may prefer to adjust rather than pay a higher price. Network-aware applications will require a general-purpose service that provides information about the past, current, and future state of all the network links that it wishes to use. Our monitoring system, described below, is a first step in providing this service.

3.0 The Distributed-Parallel Storage System

Our implementation of this high-speed, distributed cache is called the Distributed-Parallel Storage System (DPSS) [7]. LBNL designed and implemented the DPSS as part of the DARPA MAGIC project [18], and as part of the U.S. Department of Energy's high-speed distributed computing program. This technology has been successful in providing an economical, high-performance, widely distributed, and highly scalable architecture for caching large amounts of data that can potentially be used by many different users.

Typical DPSS implementations consist of several low-cost workstations as DPSS block servers, each with several disk controllers, and several disks on each controller. A four-server DPSS with a capacity of one Terabyte (costing about $80K in mid-1999) can thus produce throughputs of over 50 MBytes/sec by providing parallel access to 20-30 disks.

Other papers describing the DPSS in more detail include [23], which describes how the DPSS was used to provide high-speed access to remote data for a terrain visualization application, [24], which describes the basic architecture and implementation, and [25], which describes how the instrumentation abilities in the DPSS were used to help track down a wide area network problem. This paper focuses on how we were able to greatly improve total throughput to applications by making the DPSS "network aware."

The application interface to the DPSS cache supports a variety of I/O semantics, including Unix-like I/O semantics through an easy to use client API library (e.g. dpssOpen(), dpssRead(), dpssWrite(), dpssLSeek(), dpssClose()). The data layout on the disks is completely up to the application, and the usual strategy for sequential reading applications is to write the data "round-robin," striping blocks of data across the servers. The client library also includes a flexible data replication ability, allowing for multiple levels of fault tolerance. The DPSS client library is multi-threaded, where the number of client threads is equal to the number of DPSS servers. Therefore the speed of the client scales with the speed of the server, assuming the client host is powerful enough.

The internal architecture of the DPSS is illustrated in Figur e2. Requests for blocks of data are sent from the client to the "DPSS master" process, which determines which "DPSS block servers" the blocks are located on, and forwards the requests to the appropriate servers. The server then sends the block directly back to the client. Servers may be anywhere in the network: there is no assumption that they are all at the same location, or even the same city.

DPSS performance, as measured by total throughput, is optimized for a relatively smaller number (a few thousand)

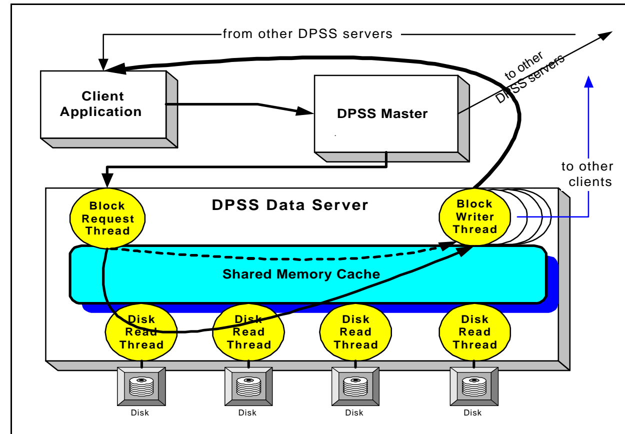

Figure 2: DPSS Architecture

of relatively large files (greater than 50 MB). Performance is the same for any file sizes greater than 50 MB. We have also shown that performance scales well with the number of clients, up to at least 64 clients. For example, if the DPSS system is configured to provide 50 MB/sec to 1 client, it can provide 1 MB/sec to each of 50 simultaneous clients. The DPSS master host starts to run out of resources with more than 64 clients.

Because of the threaded nature of the DPSS server, a server scales linearly with the number of disks, up to the network limit of the host (possibly limited by the network card or the CPU). The total DPSS system throughput scales linearly with the number of servers, up to at least 10 servers.

The DPSS provides several important and unique capabilities for data intensive distributed computing environments. It provides application-specific interfaces to an extremely large space of logical blocks; it offers the ability to build large, high-performance storage systems from inexpensive commodity components; and it offers the ability to increase performance by increasing the number of parallel disk servers.

DPSS data blocks are available to clients immediately as they are placed into the cache. It is not necessary to wait until the entire file has been transferred before requesting data. This is particularly useful to clients requesting data from a tape archive. As the file moves from tape to the DPSS cache, the blocks in the cache are immediately available to the client. If a block is not available, the application can either block, waiting for the data to arrive, or continue to request other blocks of data which may be ready to read.

The DPSS is dynamically reconfigurable, allowing one to add or remove servers or disks on the fly. This is done by storing the DPSS hardware resource information in a Globus Metacomputing Directory Service (MDS)[5] formatted LDAP database, which may be updated

dynamically. Software agents are used to monitor network, host, and disk availability and load, storing this information into the LDAP database as well. This information can then be used for fault tolerance and load balancing. We describe this load balancing facility in more detail below.

4.0 Network-Aware Adaptation

For the DPSS cache to be effective in a wide area network environment, it must have sufficient knowledge of the network to adjust for a wide range of network performance conditions and sufficient adaptability to be able to dynamically reconfigure itself in the face of congestion and component failure.

4.1 Monitoring System

We have developed a software agent architecture for distributed system monitoring and management. We call this system Java Agents for Monitoring and Management (JAMM) [13]. The agents, whose implementation is based on Java and RMI, can be used to launch a wide range of system and network monitoring tools, extract their results, and publish them into an LDAP database. These agents can securely start any monitoring program on any host and manage the output of any monitoring data. For example, we use the agents to run *netperf* [19] and *ping* for network monitoring, *vmstat* and *uptime* for host monitoring, and *xntpdc* for host clock synchronization monitoring. These results are uploaded to an LDAP database at regular intervals, typically every few minutes, for easy access by any process in the system. We run these agents on every host in a distributed system, including the client host, so that we can learn about the network path between the client and any server.

4.2 TCP Receive Buffers

The DPSS uses the TCP protocol for data transfers. For TCP to perform well over high-speeds networks, it is critical that there be enough buffer space for the congestion control algorithms to work correctly [12]. Proper buffer size is a function of the network bandwidth-delay product, but because bandwidth-delay products in the Internet can span 4-5 orders of magnitude, it is impossible to configure the default TCP parameters on a host to be optimal for all connections [21].

To solve this problem, the DPSS client library automatically determines the bandwidth-delay product for each connection to a DPSS server and sets the TCP buffer size to the optimal value. The bandwidth and delay of each link are obtained from the agent monitoring results which are stored in the LDAP database.

There are several open issues involved in obtaining accurate network throughput and latency measures. One issue is that the use of past performance data to predict the future may be of limited utility. Another issue is whether to use active or passive measurement techniques.

Network information such as available bandwidth varies dynamically due to changing traffic and often cannot be measured accurately. As a result, characterizing the network with a single number can be misleading. The measured bandwidth availability might appear to be stable based on measurements every 10 minutes, but might actually be very bursty; this burstiness might only be noticed if measurements are made every few seconds.

These issues are described in more detail in [17] and [27]. We plan to adopt techniques used in other projects such as NWS, once they are proven to be sound.

4.3 Load Balancing

The DPSS can perform load balancing if the data blocks are replicated on multiple servers. The DPSS master uses status information in the LDAP database to determine how to forward a client's block request to the server that will give the fastest response. A minimum cost flow algorithm [1][9] is used by the DPSS master to optimize the assignment of block requests to servers.

Our approach is to treat load balancing as a combinatorial problem. There is some number of clients and servers. Each client must be assigned to one or more servers without any server being overloaded.

The minimum cost flow approach is a good match for the combinatorial nature of the problem, but there are several practical challenges to overcome. In particular, the minimum cost flow algorithm is an offline algorithm; the number of blocks each client will request must be know in advance in order to generate a flow of blocks from servers to clients for a given period. However, client arrivals and departures are unpredictable, and for some clients, the request rate and the amount of data requested is also variable. Our solution is to run the algorithm each time a client request arrives, using the actual request for the current client and estimates for every other client. The algorithm itself is fast (less than 1 ms for typical graphs), so this solution is workable.

We model the DPSS load balancing problem as a transportation problem [1] (p. 99). Each server has a supply of blocks that must be delivered to the clients. The network is represented as a bipartite graph, where each node is a client or server and each edge is a network path from server to client. Each edge has a per-block cost and a maximum capacity. The algorithm finds a flow of blocks from servers to clients that minimizes the total cost. It is defined for a balanced network, where the total demand is equal to the

total supply. For the DPSS, this situation occurs only when the clients have saturated the servers. To create a balanced problem, we introduce a ghost client and a ghost server that have infinite capacity and high-cost links to other servers and clients, respectively. Supply or demand is assigned to one of the ghosts to create a balanced problem.

We assign a cost and capacity based on the assumption that network latency is the dominant factor affecting application performance, so that selecting servers with the lowest latency will maximize application performance. The total latency from a client's request to its receipt of the first tile from a server is affected by three different network paths: the paths from client to master, master to server, and server to client. The master obtains the latencies from these three paths from the LDAP database. The total delay for the edge cost is the sum of the three latencies, the processing delay at the master and server, and the transmission delay of a data block across the link between server and client. Data blocks are large (typically 64KB), so the transmission delay is non-trivial, even across a high-speed network.

One limitation of this approach is that the graph does not represent the actual network topology. Several edges in the graph may actually share the same bottleneck link in the real network, but the graph does not capture this information. The minimum cost flow algorithm could accommodate a more detailed model of the network, but the monitoring system only collects information about host-to-host performance.

The edge capacity is set to the bandwidth obtained from the LDAP database. This capacity may be reduced based on the degree of replication of the data blocks. When data is loaded into the DPSS, blocks are distributed acros n servers and each block is replicated m times, where $m <= n$. If we assume blocks are uniformly distributed to servers, then it is unlikely that any one server will store more than m/n percent of the blocks requested. The actual edge capacity assigned is the minimum of the bandwidth and m/n percent of the data requested by the client.

The bandwidth data from the LDAP database is also used to set the server's supply. The supply at a server is the total bandwidth available to all clients. This bandwidth must be determined heuristically because the monitoring system only reports the maximum bandwidth available to each client. We might naively assume that the total bandwidth is the sum of the bandwidth available to each client. If several clients share the same bottleneck link, however, the total bandwidth will be less. We conservatively assume that all clients share the same bottleneck link and set the total bandwidth to the maximum bandwidth available to any one client.

The load balancing implementation maintains a graph data structure that is modified whenever clients arrive or leave. The edge costs are recomputed every three minutes based on data from LDAP. We use the CS2 [4] minimum cost flow solver. For a particular request, the solver determines what proportion of the blocks will be delivered by each server. Each block must be looked up in the block database to determine which specific servers it is loaded on. A stride scheduler [26] chooses one of the available servers based on the proportions assigned by the solver.

5.0 Results

5.1 TCP Buffer Tuning

Table 1 shows the results from dynamic setting of the TCP receive buffer size. This table illustrates that buffers can be hand-tuned for either LAN access or WAN access, but not both at once. It is also apparent that while setting the buffer size big enough is particularly important for the WAN case, it is also important not to set it too big for the LAN environment.

If the buffers are too large, throughput may decrease because the larger receive buffer allows the congestion window to grow sufficiently large that multiple packets are lost (in a major buffer overflow) during a single round trip time (RTT), which then leads to a timeout instead of a smooth fast retransmit/recovery. [20]

Table 1

buffer method	network	Total Throughput
hand tune for LAN (64KB buffers)	LAN	33 MBytes/sec
	WAN	5.5 MBytes/sec
hand tune for WAN (512 KB buffers)	LAN	19 MBytes/sec
	WAN	14 MBytes/sec
auto tune in DPSS library	LAN	33 MBytes/sec
	WAN	14 MBytes/sec
LAN RTT = 1 ms over OC-12 (622 Mbit/sec) network WAN RTT = 44 ms over OC-3 (155 Mbit/sec) network		

5.2 Load Balancing

We first ran a series of tests to verify that latency is the dominant factor in determining which server to use in the load balancing algorithm. Figure 3, Figur e4, and Fig ure5 show the results of using dynamic load balancing varying one factor at a time. In Figure 3 we used servers with the same load and latency, and varied the available network throughput (the first two servers were on OC-3, the third on 10BT ethernet, and the fourth on 100BT ethernet). In Figur e4 we used DPSS servers with the same network throughput and latency, but varied the server CPU power available by using servers with other jobs running

simultaneously. The first server had no load, the second had 33%, the third had 50%, and the fourth had 66% load. For the test shown in Figu re5, we used servers with the same network throughput and load, but with varied latencies, where the first server had a latency of 0.5 ms, the second and fourth of 40 ms, the third of 2 ms.

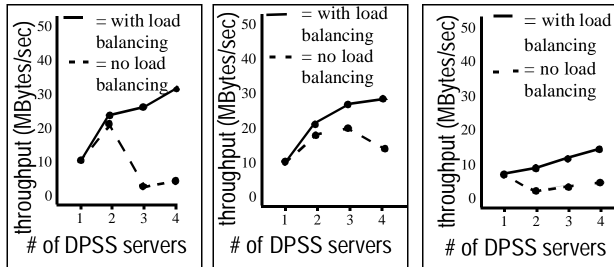

Figure 3: varied network speed

Figure 4: varied server CPU

Figure 5: varied latency

Note that the results for the DPSS without load balancing actually decrease starting with the third server in Figur e3, which is on 10 Mbit/sec ethernet, because without load balancing total throughput is constrained by the speed of the slowest server. The same thing happens in Fi gure4 and Figure 5. Therefore, the total throughput of the system with no load balancing is the speed of the slowest server, times the number of servers. The overall throughput in Figure 5 is less than in Figures 3 and 4 because the servers used were considerably less powerful, and were connected by networks with a throughput of only 12 Mbits/sec.

5.3 Test Environment

We used the testbed environment shown in Figure 7 to evaluate the effectiveness of load balancing in the DPSS. The network performance in the testbed limits the performance improvements we could achieve with load balancing. This section outlines the basic network performance data obtained from the monitoring system, based on its *netperf* and *ping* measurements. These measurements are performed every five minutes.

The platforms and operating systems varied from host to host, which caused each host to have slightly different performance characteristics. Server A at LBNL is a 2-CPU UltraSparc 2 running Solaris 2.6, while Server B is 2-CPU 200 MHz Pentium II running Solaris X86. The sustained bandwidth from Server A to Client A is 112 Mbps, but only 80 Mbps to Server B.

A network configuration problem limited the bandwidth between Server C and Client B to 11 Mbps, although Server C achieved 107 Mbps to Client A and Client B to Server A achieved 56 Mbps. All West Coast hosts can ping each other with 0 or 1 ms delay. Server D and Client C, both in Kansas, achieved 85 Mbps throughput; the delay is 3ms.

The DS3 link between the West Coast sites and the machines in Kansas could sustain up to 14 Mbps. The delay for pings across the DS3 link ranged from 35ms to 44ms, depending on the specific hosts involved.

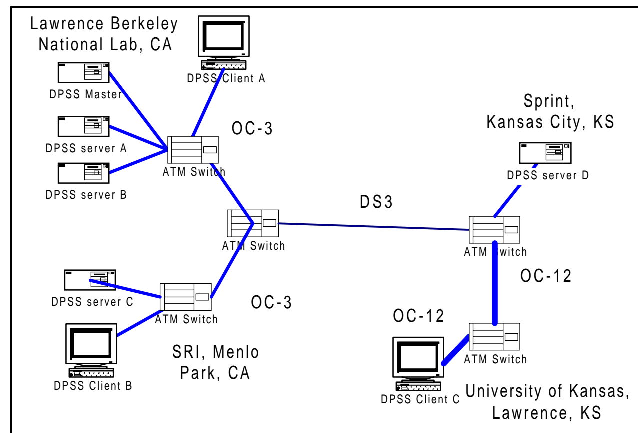

Figure 6: Test configuration

We performed a series of tests using a test client that reads a single 150MB file from the DPSS. The file is loaded in 2400 64KB data blocks, and the client requests 64 data blocks (4 MB) at a time. The client waits until all 64 blocks are delivered before issuing the next request.

We compare the load balancing code, which we call the minimum cost master, to a previous version of the DPSS master, called the greedy master. We also vary the degree of replication and the number of clients. The greedy master uses a two-part server selection strategy. For 75% of the requests, it uses a greedy strategy, selecting the server that has the highest bandwidth connection to the client. For the other requests, it randomly selects a server.

5.4 Performance Result

Our experiments show that replication and load balancing increase the throughput of the DPSS by a factor of 2.17 compared to the use of unreplicated data. Using replicated data, load balancing increases throughput by 33% compared to the greedy strategy. In the testbed configuration, the minimum cost master sustained a peak throughput of 128 Mbps to three clients using a fully replicated data set; without replication, the peak throughput was only 59 MBps. With replicated data, throughput for the greedy master was 96 Mbps

We experimented with load balancing and replication using four different data set configurations. When a data set is loaded on the DPSS, each data block can be loaded on one or more servers. We label a configuration as x-by-y, where x is the total number of servers used and y is the number of servers each block is loaded on. For example, a data set that is fully replicated on four servers is a 4x4 configuration. For our tests, one configuration was 4x4. For

the other tests, three servers (A, C, D) were used. One configuration was fully replicated (3x3), one was replicated twice (3x2), and was not replicated (3x1). Since we expect this type of storage cache to be mainly used with very large data sets, in practice the data will likely only be replicated at most twice.

We performed two series of tests. The first series measured a single client accessing the DPSS to provide a performance baseline. The second series measured three clients using the DPSS simultaneously.

Load balancing has a greater effect when multiple clients are running, because a small number of clients can produce more load than a single server can handle. The master dynamically adapts to the load generated by clients to reduce contention at busy servers.

Our first test measured the sustained throughput a single client could achieve for each data set. The test was intended to show the best possible bandwidth that a client could achieve for a given configuration, which provides a baseline for future tests.

The test client ramps up to maximum performance over the first two or three block requests. We exclude this ramp up from the results by measuring the performance over the last half of the requests. Figure7 s hows the results for these tests, using clients on hosts A, B and C in Figur e6. They show that the clients' performance is governed by the quality of their connections to the servers. For the 4x4 data set, Client A does the best, while Client C, which has a slow connection to the local servers, does the worst.

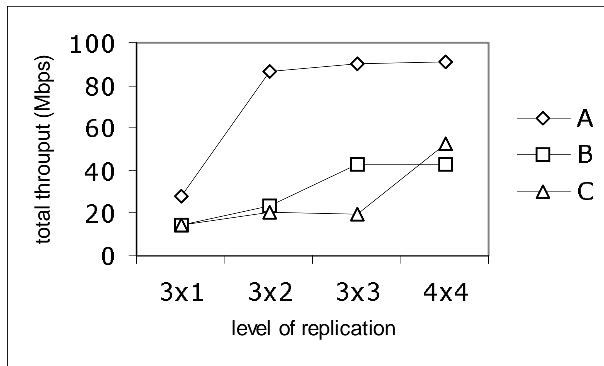

Figure 7: Results for 1 client

Our second test measured the total sustained throughput of the DPSS when three clients were running simultaneously. These tests require measuring the effectiveness of the load balancing algorithm at allocating clients to servers without overloading servers. Figure shows the results for the 3-client tests. It shows the total throughput delivered to each client.

The total throughput for the 4x4, 3-client test is 75% of the sum of the single client tests. Clients A and B both used Server A in the single client test. When they run

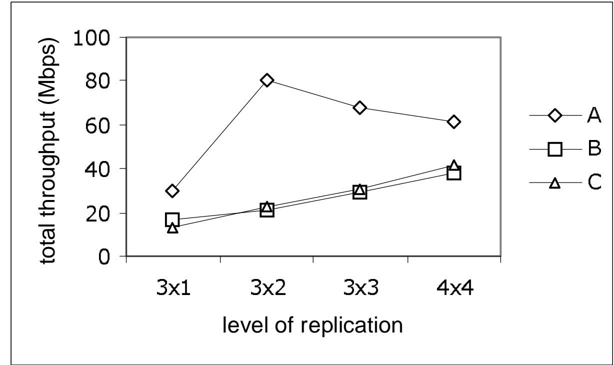

Figure 8: Results for 3 simultaneous clients

simultaneously, their total bandwidth requirements are more than the server can handle. Some of the excess demand is handle by Server B. As a result, the total throughput is lower.

The 3x2 case shows the benefits of the minimum cost flow algorithm. No server has a copy of all the data blocks, so each client must use two or more servers. This limited replication increases contention at servers and reduces the aggregrate throughput of the DPSS. For the 3x2 data set, it can only get two-thirds of the data from Server D. The rest of the data comes from Server A, which lowers Client C's throughput.

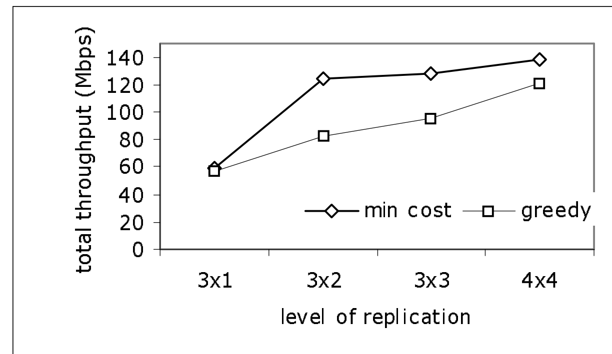

Figure 9: Comparison with "greedy" method

Figur e9 shows the performance of the minimum cost flow load balancing method vs. the greedy server selection method, showing the sum of the throughput from all three clients. The greedy strategy does not perform as well as the minimum cost master, because it does not take into account latency, the amount of bandwidth needed to satisfy requests, or the number of clients using the same server. Each problem occurs in practice. For the 3x2 test, Client C had to use some servers on the West Coast. The highest bandwidth link also had the highest latency; the load balancing algorithm got better performance by choosing a server with lower bandwidth and lower latency. In other

tests, the greedy master causes a single server to be overloaded by choosing the same server for several clients.

Overall, load balancing and replication increase the bandwidth delivered to applications. Replication allows clients to use multiple servers, but without load balancing clients are still limited by the speed of the slower server. The minimum cost flow approach to load balancing increases the total throughput of the system by adapting to varying client demand. A summary of the total throughput results for the 3 server, 2-way replicated scenario is shown in Table 2 below.

Table 2

number of clients	no replication (3x1)	greedy master (3x2)	min flow master (3x2)
1	60 Mbps	71 Mbps	87 Mbps
3	60 Mbps	82 Mbps	124 Mbps

5.5 Future Work

Our work on the DPSS is continuing in several areas.

We plan to test the DPSS in a larger testbed, with an OC-12 wide area link and more clients. This will allow us to evaluate the minimum cost master under heavier load, where we expect load balancing to have a greater impact.

The specific minimum cost flow algorithm used is also being studied. We currently model load balancing as a single-commodity flow problem, which requires that different data sets used by clients as be loaded on the same servers and with the same degree of replication. A multi-commodity flow algorithm, while computationally more expensive, would not impose this requirement.

We would also like to investigate the use of a more detailed network model to drive the minimum cost flow algorithm. One improvement would be to monitor performance between end hosts and internal network nodes, which would allow the master to detect contention on shared links. We also plan to experiment with other network monitoring methods, such as passive methods, for collecting network throughput information.

6.0 Related Work

Research on automatic TCP receive buffer tuning in the operating system is being done by the Pittsburgh Supercomputer Center [21]. While this is a very nice approach, it will currently only help applications running on hosts with a customized version of the NetBSD operating system. Our approach can be used by any distributed application on any Unix system.

Other network-aware application or middleware projects include the Remulac project [6][22], which is also doing network monitoring and application adaptation based on the current network conditions. This project focuses on designing a middleware API for applications to use; we focus instead on issues for data intensive computing.

The goals of our network monitoring system are similar to that of the NPACI Network Weather Service (NWS)[27]. NWS is a distributed system that allows one to periodically monitor various network and computational resources. NWS then dynamically forecasts the performance of these resources based on their past performance. The advantage of our system is that the agent mechanism makes it very easy to dynamically change what is being monitored, and our use of LDAP to publish results makes it easy for any application to access the results. The advantage of NWS is that it maintains historical information and can do predictions of future performance. This ability to predict future performance would be extremely valuable for this system, and we plan to try to incorporate NWS into our JAMM system.

Our monitoring system is also somewhat similar to the Globus network performance tool, GloPerf, which supports dynamic measurement and publication of network and bandwidth information.

7.0 Conclusions

We have described a data architecture for widely distributed data intensive applications, which is centered around the use of a high-speed network-aware cache. We believe that this type of network-aware data cache will be an important architectural component to building effective data intensive computational grids.

We have also shown that adding network-awareness to a distributed system can greatly improve its performance. We are using real time network and system monitoring information as input to a load balancing algorithm. While more testing is needed to validate the utility of this particular load balancing algorithm in more general network environments, clearly this type of monitoring will be required to enable high performance grid-like systems.

Finally, in order to achieve best performance we have shown that data should be replicated. There is clearly a cost involved in doing this, namely the cost of additional servers/disks. However, this cost is relatively small, and much less that the cost of enhancing other resources such as network capacity. Moreover, replication has the added benefit of providing redundancy for fault tolerance.

8.0 References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. Network Flows: Theory, Applications, and Algorithms. Prentice-Hall: Englewood Cliffs, N.J., 1993.

[2] Erol Akarsu, Geoffrey C. Fox, Wojtek Furmanski, Tomasz Haupt, "WebFlow - High-Level Programming Environment and Visual Authoring Toolkit for High Performance Distributed Computing," Proceedings of IEEE Supercomputing '98, Nov. 1998.

[3] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, Michael Wan, "The SDSC Storage Resource Broker," Proc. CASCON'98 Conference, Nov.30-Dec.3, 1998, Toronto, Canada. (http://www.npaci.edu/DICE/SRB/)

[4] Boris Cherkassky and Andrew Goldberg. CS2. http://www.star-lab.com/goldberg/soft.html.

[5] K. Czajkowski, I. Foster, C., Kesselman, N. Karonis, S. Martin, W. Smith, S. Tuecke. "A Resource Management Architecture for Metacomputing Systems," Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing,1998.

[6] DeWitt, T. Gross, T. Lowekamp, B. Miller, N. Steenkiste, P. Subhlok, J. Sutherland, D., "ReMoS: A Resource Monitoring System for Network-Aware Applications" Carnegie Mellon School of Computer Science, CMU-CS-97-194. http://www.cs.cmu.edu/afs/cs/project/cmcl/www/remulac/remos.html

[7] DPSS: http://www-didc.lbl.gov/DPSS

[8] Globus: See http://www.globus.org

[9] Andrew V. Goldberg. An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm by A. V. Goldberg, Journal of Algorithms, Vol. 22, pp. 1-29, January 1997.

[10] Grid: *The Grid: Blueprint for a New Computing Infrastructure*, edited by Ian Foster and Carl Kesselman. Morgan Kaufmann, Pub. August 1998. ISBN 1-55860-475-8. http://www.mkp.com/books_catalog/1-55860-475-8.asp

[11] Habanero: http://www.ncsa.uiuc.edu/SDG/Software/Habanero/

[12] V. Jacobson, "Congestion Avoidance and Control," Proceedings of ACM SIGCOMM '88, August 1988.

[13] JAMM: http://www-didc.lbl.gov/JAMM/

[14] William E. Johnston. "Real-Time Widely Distributed Instrumentation Systems," In *The Grid: Blueprint for a New Computing Infrastructur* . Edited by Ian Foster and Carl Kesselman. Morgan Kaufmann, Pubs. August 1998.

[15] William E. Johnston, W. Greiman, G. Hoo, J. Lee, B. Tierney, C. Tull, and D. Olson. "High-Speed Distributed Data Handling for On-Line Instrumentation Systems," Proceedings of ACM/IEEE SC97: High Performance Networking and Computing. Nov., 1997. http://www-itg.lbl.gov/~johnston/papers.html

[16] Legion: See http://www.cs.virginia.edu/~legion/

[17] Bruce Lowekamp, B. Miller, N. Sutherland, D. Gross, T. Steenkiste, P. Subhlok, J., "A Resource Query Interface for Network-Aware Applications," 7th IEEE Symposium on High-Performance Distributed Computing, IEEE, July 1998, Chicago

[18] MAGIC: "The MAGIC Gigabit Network." See: http://www.magic.net

[19] Netperf: http://www.netperf.org/

[20] Vern Paxson, private communication.

[21] J. Semke, J. Mahdavi, M. Mathis, "Automatic TCP Buffer Tuning," Computer Communication Review, ACM SIGCOMM, volume 28, number 4, Oct. 1998.

[22] P. Steenkiste, "Adaptation Models for Network-Aware Distributed Computations," 3rd Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC'99), Orlando, January, 1999.

[23] Brian Tierney, William E. Johnston, Hanan Herzog, Gary Hoo, Guojun Jin, Jason Lee, Ling Tony Chen, Doron Rotem. "Distributed Parallel Data Storage Systems: A Scalable Approach to High Speed Image Servers," ACM Multimedia '94 (San Francisco, October 1994). http://www-itg.lbl.gov/DPSS/papers/

[24] Brian Tierney, W. Johnston, H. Herzog, G. Hoo, G Jin, and J. Lee, "System Issues in Implementing High Speed Distributed Parallel Storage Systems," Proceedings of the USENIX Symposium on High Speed Networking, Aug. 1994, LBL-35775. http://www-itg.lbl.gov/DPSS/papers.html.

[25] Brian Tierney, W. Johnston, G. Hoo, J. Lee, "Performance Analysis in High-Speed Wide-Area ATM Networks: Top-to-Bottom End-to-End Monitoring," IEEE Network, May, 1996, Vol. 10, no. 3. LBL Report 38246, 1996. http://www-itg.lbl.gov/DPSS/papers.html

[26] Carl A. Waldspurger and William E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management, Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995.

[27] Rich Wolski, Neil Spring, and Jim Hayes, "The Network Weather Services: A Distributed Resource Performance Forecasting Service for Metacomputing," http://nsw.npaci.edu/