An Architecture for Dynamic Allocation of Compute Cluster Bandwidth

John Bresnahan[1,2,3], Ian Foster[1,2,3]

[1]Math and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439
[2]Computation Institute, University of Chicago, Chicago, IL 60637
[3]Department of Computer Science, University of Chicago, Chicago, IL 60637

{bresnaha, foster}@mcs.anl.gov

***Abstract*—Modern high-performance computers are often implemented as clusters. A cluster is comprised of several machines, each representing a compute node. Local resource managers are used to grant access to these nodes. Once access is granted to a node, a user has exclusive access to the entire node including its local filesystem and network interface. In addition to serving as a computational cluster, these sites are often highly connected to a network: for example, sites participating in the US TeraGrid system have between ten and forty gigabits per second of available bandwidth and each computational node in a TeraGrid cluster typically has a one gigabit per second network interface. However, since it is a compute cluster, jobs are often computationally bound, in which case most of this bandwidth is unused.**

In order to stage in and out data sets, or to publish computational results, sites provide file transfer services. Achieving peak transfer speeds requires a significant number of nodes. Packet switching at top speeds put a load on a CPU that prohibits locating user compute jobs and site transfer processes on the same node. Thus, administrators partition their resources into compute nodes and transfer nodes. Due to fluctuating transfer requirements and the desire to avoid idle resources it is difficult to determine this partition statically. Ideally, the partitioning could be adjusted dynamically to suit immediate needs.

This paper studies this dynamic resource partitioning problem. We propose an architecture that allows transfer nodes to be acquired from the computational queue when needed, and returned when no longer required. The site administrator has a means to set a policy regarding how long a transfer node can be idle before being returned, and when to request additional nodes. We measure the costs associated with a prototype implementation of this architecture and study the impact of different policies on achieved performance

***Index Terms*—GridFTP, Dynamic Resource Allocation, Backend**

I. Introduction

Science has many computationally intense problems. To foster research, so-called “space shared, batch scheduled” systems have been developed that allow many users to obtain dedicated access to powerful computer processors, or nodes, via local resource managers (LRMs). An LRM is an interface to a run queue. LRMs allow jobs shared exclusive access to system resources. A user requests a certain number of nodes and the required wallclock time, and then waits until their request is serviced. Once the LRM can fulfill the request, the user’s job may use 100% of the resources associated with the allocated nodes.

On modern computational systems, like the TeraGrid [1], it is common that such systems are implemented as Linux clusters. Each compute node is a full Linux system. When a user gains access to a node, they acquire exclusive use of the entire Linux machine and all of its devices, including the network interface card (NIC) and filesystem. Despite such systems being mainly targeted at computationally bounded jobs, their network connectivity is typically quite good. In the case of TeraGrid clusters, each node has its own Gbit/s NIC that connects to a 10Gbit/s switch.

Jobs make use of the network by staging in data sets for processing and staging out results for analysis. Jobs also perform I/O using any of several network and parallel file systems such as NFS [2], PVFS [3], and GPFS [4]. Jobs may also establish local and wide area network connections to service application-specific needs. However, most jobs are CPU-bound, and thus a cluster typically does not use all available bandwidth simply to service the needs of general user jobs submitted to the queue. For example, the TeraGrid publishes the network utilization of their routers at https://network.teragrid.org/. Rarely are peaks of a twenty second average over 2 Gbit/s, and the majority of the time the network is idle.

Clusters also commonly run data transfer services so that external clients can access data stored on file systems attached to the cluster. This external access may be used, for example, to enable community access to result sets. On TeraGrid, GridFTP is used as the primary data transfer service because of its high performance and scalability: the Globus GridFTP implementation can easily scale to the 30-40Gbit/s of external bandwidth available at some TeraGrid sites—if sufficient nodes can be dedicated to data transfer tasks. However, because each individual node has only a 1 Gbit/s NIC, in order to fully utilize a 40 Gbit/s network, forty nodes must participate in a transfer with 100% of their NIC. In practice, sites are reluctant to dedicate that number of nodes to data transfer tasks, given that it is only occasionally that external clients require this data transfer rate. Thus, innovative uses of TeraGrid clusters, such as network replication applications, proxy applications, and transfer applications such as DRS [5] and SQUID [6], cannot be supported.

One solution to this problem is to dedicate nodes to the data transfer service. However, unless the site is constantly in need of peak data transfer requirements, some of these nodes will be idle. Typically, peak data transfer requirements are short lived. Assuming that the LRM always has some jobs in the wait state (which is typical for successful compute sites), this leads to a situation where the site has job requests waiting even though there are idle resources on which they could run. Since these idle resources could be diverted to servicing computational jobs, again the wait times are artificially long.

Another solution is to use a node for the data transfer service while that same node is servicing a user's job. However this approach is also not ideal. On modern processors, packet switching at gigabit rates requires a significant amount of the CPU. When any type of security processing, like packet signing or encrypting, is used the problem is greatly exacerbated. On a dual Itanium-64 1.5GHz UC TeraGrid node, we ran transfer tests for ten seconds and measured the CPU load using some common performance benchmarks: iperf [7], globus-url-copy [8], globus-xioperf [9], and netperf [10]. The results are shown in Table 1. In the best case, 8% of the CPU was used which is a nontrivial amount if the user is expecting a dedicated resource. Users who have acquired nodes from the queue have an expectation of exclusive access to the node's resources, co-locating a high performance transfer with a user's job would not only deny the user network cycles that they may require, but also CPU and memory bus cycles.

Table 1: CPU load imposed by different network benchmarks

Benchmark	CPU Load (%)	Transfer Rate (Mbit/s)
netperf	8	990
iperf	10	990
globus-url-copy	13	944
globus-xioperf	11	944
globus-xioperf (gsi integrity)	79	99
globus-xioperf (gsi private)	88	40

The correct solution to this data transfer problem, we believe, is to allow nodes to be transferred dynamically, in an on-demand fashion, between a pool of compute resources and a pool of data transfer resources. When data transfer requirements increase, nodes can be transferred from the compute pool to the data transfer pool; when the transfer load drops, or perhaps when the compute load increases, nodes can be returned to the compute pool and again used to service computational requests. In this way, we ensure that there are never idle resources when job requests (or data requests) wait to be serviced.

This paper proposes an architecture that allows such a dynamic partitioning of transfer nodes and compute nodes. We present an architecture that can work with any standard LRM on a computational cluster. We propose extensions to the Globus Toolkit(tm) GridFTP server [11] and develop implementation techniques to support this architecture. We use monitoring interfaces defined by the Web Services Resource Framework (WSRF) [12] to observe the state of the architecture in a non-disruptive and modular way, and allow site administrators to set policies controlling the partitions in a modular fashion. We performance test a prototype system to determine costs associated with our approach.

II. Related Work

We review work that has overlapping concepts and solves similar problems to that of dynamic allocation of compute cluster bandwidth. The majority of such work focuses on storage space or CPU scavenging. We also look at concepts in queuing theory. Because we are allocating resources to user requests there is overlap with queuing research. Additionally we review work regarding load predictions for both network transfers and computational jobs.

Shadan et al., in their paper "Dynamic Mirroring for Efficient Web Server Performance Management" [13], propose a solution for dynamically mirroring of web servers. In their approach, they have a set of cooperative web servers. As the main server's load gets unacceptably high, they transfer some portion of the data to a selected mirror site. This work introduces interesting concepts for WAN replication, but that is not the goal of our research. In our work we decouple a transfer resource from a storage resource. Further, they assume a static set of mirror servers and do not allow for a growing and shrinking pool of resources, which is an important part of our work.

Freeloader [14] is a distributed storage scavenging system. It takes advantage of unused desktop storage and available bandwidth to create a distributed storage cache for read only data. Freeloader asserts that desktop machines do not have the storage capacity to store large scientific data sets on local disk in their entirety, yet at the same time at least half of the local disk is typically empty. Its goal is to harness all empty disk space at a site into a single data cache with little additional expense. The work we present here shares the ideas of reclaiming idle bandwidth, and the idea of providing additional I/O power at little additional cost to the site; however we are not investigating storage systems. Our focus is on high utilization of network resources.

Domain Name Service (DNS) has a load balancing feature. It can be used to share host load by mapping server physical machines to the same domain name. Clients send the DNS a domain name, and the DNS returns to them an IP address.

When DNS-Round-Robin is in place, several IP addresses are mapped to the same domain name. The IP address that the DNS chooses to respond with is selected in a round robin fashion. While effectively distributing the physical machines to which a client connects, it does not distribute load. A client may lookup a server and not use it for a transfer, in which case some servers will be used for transfers more often than others. In our approach we only seek out a transfer node when we have a transfer request, therefore we have a good idea of the load it will incur. More importantly, the DNS approach does not take into account sharing of computational and transfer resources in a dynamically changing environment.

Our work overlaps with concepts prevalent in queuing theory. We are allocating resources to support a user load which is the fundamental problem that queues are designed to solve. Our work is not researching into the field of queuing theory. We are rather acting as a user of it from a layer above. However, it is important for us to understand how we are affected by some of its fundamental concepts. Additionally, ongoing work in this field will allow some important extensions to the architecture we propose here.

The backfill algorithms described in "Core Algorithms of the MAUI Scheduler" [15] and "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling" [16] play a role in node acquisition for our system. The proposed architecture will perform best when nodes can be obtained quickly. Further, most transfer jobs run for a relatively short amount of time. These two characteristics dovetail well into backfill. A queue maintains a list of jobs that it orders according to priority. To prevent starvation, the order in which the jobs are requested plays a strong role in this priority. The scheduler then matches jobs with nodes. Higher priority jobs receiver earlier start times. Since requests vary in wallclock time and number of nodes required, the process inevitably leaves time gaps between jobs. Backfilling searches the remaining lower priority job to find those that can fit into these time gaps. This can result in lower priority jobs running before higher priority jobs, but it does not delay any job from its originally scheduled time, it simply uses what would otherwise be idle time. Naturally jobs which request a small number of nodes and a short wallclock time are able to fit into the most gaps and thus have a higher likelihood of running at an earlier time. Our architecture takes advantage of this by always requesting the smallest number of nodes, and a relatively short wallclock time.

There has been much research on predicting resource load requirements [17] [18] [19], including network loads [20] [21] [22]. Our system can benefit from this work. If we were able to predict network loads quickly and accurately enough, we could set resources aside ahead of time for network traffic spikes. To allow for this possibility, we have designed into our architecture a way for prediction algorithms to be implemented in a natural and modular fashion. We discuss this topic when we introduce the controller program.

Fast access to nodes is one challenge that our architecture faces. We cannot instantly obtain a node because a user's job may be running on it. Therefore we must wait for a job to complete before the scheduler will even consider granting access to our system. Ideally we could suspend any running job, use its node for a transfer, and then restart that user's job when we are finished, all without disrupting the job or denying it wallclock time. Research into virtual machines [23] can be used to achieve this. By running each job in a virtual machine the entire system state can be suspended and even migrated to another host. As this research advances we expect it to provide solutions for immediate node access.

III. Architecture

Recall that our goal is to define an architecture which allows well connected computational clusters to service bandwidth greedy applications without incurring the cost of dedicated resources. We propose a modular architecture that uses as many existing components as possible. Monitoring software is used to observe the state of the stock components and provide enough information so that policy decisions can be made in a separate and noninvasive process. Our architecture defines the following components:

Nodes: Nodes are full computer systems complete with their operating system, CPU, memory, disk and NIC. A typical example is a PC running Linux. Nodes are used for servicing user requests. They must be appropriate for both computation and transfer with fast NICs (e.g., gigabit Ethernet cards) and fast modern CPUs. Nodes should have access to a shared file system. This is a common description for most computational clusters, as on the TeraGrid.

LRM: An admission control queue that has its own means for setting policy for granting access to nodes. The ability to set high priority in order to preempt jobs that have waited longer in the queue, and the ability to suspend running jobs are important features but not mandatory. Common examples of such LRMs are PBS [24], LSF [25], MAUI, and LoadLeveler [26]. A GRAM [27] interface to the LRM is recommended.

Transfer Service: This is the component responsible for transferring data in and out of the cluster. The transfer service comprises a frontend service and a set of backend services. The frontend service acts as a contact point for clients, and as a load balancing proxy to the backend services which move the data. Backend services run on nodes acquired from the LRM. The frontend must be instrumented with monitoring software so that an external process can observe its state to make decisions based upon the load.

Controller: This component is the brain of the system. It observes the frontend transfer service to determine when client load is high or when the backend transfer node pool is idle, at which point it interacts with the LRM to either acquire a new node for use in data transfer or release an idle node from the transfer pool back into the computational queue. The controller's decisions are based upon a policy set by the site administrator.

Figure 1 shows how these components interact. Clients contact the frontend transfer service with transfer requests. The frontend goes to the transfer node pool to find an available backend service. The backend service with the lowest load is selected to perform the transfer. If there is no available backend service, or all are too busy, the frontend waits for some amount of time and then asks again. The client is stalled on this process but is free to timeout without disturbing the system.

The controller observes the state of the frontend service. When the client load exceeds the available resources in the transfer pool, the controller requests a new backend service from the LRM. The faster this request can be serviced, the better the overall system's performance. Ideally, the controller would have special privileges allowing it to acquire nodes immediately, for example by suspending current user jobs or preempting other jobs that have waited longer. Typically transfers do not require large amounts of time so nodes acquired in this way could be returned to the compute pool quickly.

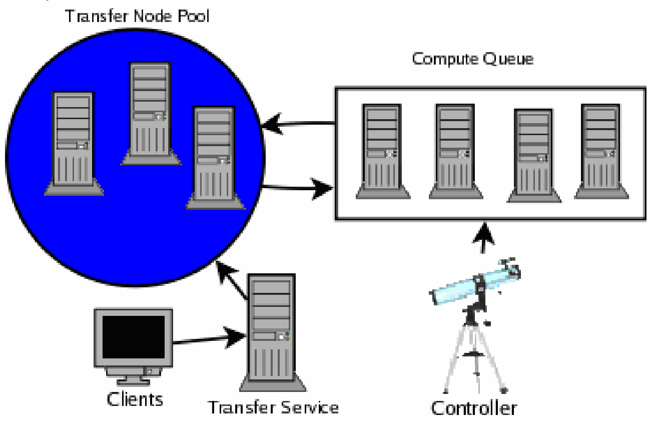

Figure 1: Component Interaction

In the absence of, or in addition to, special rights to the queue, the controller program can implement "caching" or retention policies, meaning that it holds nodes for some time before returning them. Similarly, "prefetching" policies can be implemented that attempt to ensure that there are always idle transfer nodes available: as resources in the transfer pool are consumed, additional nodes can be requested from the LRM. More sophisticated logic can also be implemented: for example, we might observe the state of the frontend service over a long time and make assumptions and predictions about client load spikes. The controller can also access log files and databases at the administrator's discretion. The optimal policies may be site specific, so it is crucial to the architecture that the administrators have a means to set sophisticated policy in ways suited to their specific needs.

The architecture provides another important source of flexibility. A client can interact directly with both the LRM and the frontend service. Using their own credentials and submission rights clients can request the same standard backend service that the controller requests from the LRM. The client can then request a transfer from the frontend service specifically requesting use of its backend service. Once that backend service has worked its way through the LRM's wait queue, the frontend will select it and the client's transfer will begin.

This feature allows the site to directly harness client knowledge. The clients know what transfer load they require. In this way they ask for resources directly from the LRM, therefore they gain access to transfer nodes in a fair and managed way. Further they maintain all of the conveniences of a standard transfer service interface because they are not required to stand up their own custom transfer services. If the user needs specific resources and times dedicated to their transfer they can arrange advanced reservations [28] in the same familiar manner as they would for any compute job. Further most LRMs have mechanisms to provide approximate start times. This can be useful information to transfer applications that need resource guarantees. This feature effectively provides a mechanism for scheduling data transfers. By running backend services directly from the LRM, instead of via an intermediate provisioning service (like the frontend and controller) all of the features of the LRM can be leveraged. Scheduling the transfer with fair sharing to site resources is then controlled in a more appropriate place, as is load prediction and security audit trails.

IV. Implementation Details

Our proposed architecture allows a site to use any LRM and node configuration that it already has in place. Our goal is to provide existing computational sites with insight into how they can better leverage their available bandwidth. An important subgoal is to be minimally disruptive to the existing system. The only new components that we require to implement this architecture are the transfer service and the controller.

A. Transfer Service

For the transfer service we use the GridFTP server released with the Globus Toolkit [29]. This versatile piece of software can be configured to run in a variety of modes [30]. For this paper we used the separation of processes mode, which means essentially that we configure it as a proxy server. A frontend service acts as the contact point for clients wishing to initiate data transfers; this service does not do any of the actual transporting of data. The actual shipping of bits is left to the backend services. The frontend and backend services communicate with each other via an internal protocol (typically over a LAN), which allows processes to be on different machines. The backend services are workers for the frontend and perform operations as they are directed.

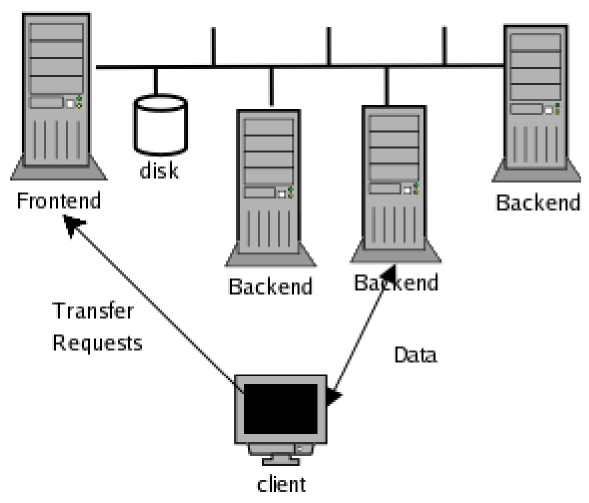

Figure 2: Separation of Processes

As shown in Figure 2, this setup allows the frontend service to select from a set of backends. All backend services have access to the same network and parallel filesystem and thus are functionally equivalent. In the current GridFTP implementation, incoming requests are allocated to backends in a round robin fashion. In effect, the frontend service acts as a load balancing proxy server. It is also free to select many backend services for use in a single transfer, in which case it divides the request by the number of selected backends and forwards the divided requests along appropriately. This feature is called striping. It is possible to mix striping and load balancing, but for simplicity we focus here on load balancing and reserve striping for future work.

B. GridFTP Modifications

The standard GridFTP server implementation requires that the frontend be given a static list of backend services. These are the only backends with which the frontend can communicate, and the list must be known at the time the frontend is started. Because we wish to create a system in which the pool of backends shrinks and grows throughout the frontend's lifetime, we had to extend the GridFTP server.

Specifically, we modified the frontend service to listen for a connection on a given TCP port. When a connection is made, the remote peer is authenticated via GSI [31]. If access is granted, the peer can register a new backend service with the frontend service. The newly registered backend service is now an available resource to be used in the proxy server setup. A timeout is associated with this registration. If no communication with the specified backend service occurs before the timeout, it is removed from the list of available resources. Similarly if an error occurs when communicating with the backend service, it is removed from the list of those available. In addition to registering services, a peer can also refresh or cancel an existing registration. The refresh allows idle backend services to remain in the available list and serves as a heartbeat monitor.

When clients requests transfers of the frontend, there must be at least one available backend or the transfer will fail. Additional modifications were made to cause the frontend to check repeatedly for available backends after a configurable delay. This functionality allows a client to use a backend that became available subsequent to the initial transfer request.

C. Monitoring

In order to monitor the internal state of the GridFTP server, we needed to instrument it with monitoring software similar to that described in CUMULVS [32]. We required that this monitoring software be minimally invasive and that it provide a network interface for observation. It is not acceptable for a monitored GridFTP server to be significantly slower or less robust than a non-monitored server. Observation via a remote process over a network interface allows experimentation with various policies without the risk of crashing or hanging the GridFTP server.

Each site has varying levels of logic needed to express its partition policies. Rather than using a specific policy language, we define the transfer service monitoring interface and a LRM submission interface, and leave it up to a specific site administrator to implement a controller process in whatever form is convenient. Languages such as Python, Java, and C can also be used to express logical policies. Further this allows maximal flexibility and full use of all available knowledge sources, including system logs, databases, and prediction toolkits [33] [34]. Since system administrators will be writing these scripts, it is important that it is easy to use the interfaces to both the transfer services internal state and to the LRM.

For these reason we followed the approach taken in GMonSteer [35] for monitoring the transfer service and GRAM as the LRM interface. We use WSRF by embedding a WS C Core hosting environment into the GridFTP server. Controller programs are written as clients to this service. Before we can explain the details of how this was done a brief discussion on WSRF is required.

D. WSRF

WSRF introduces the concept of a resource into the web services world. A resource is a collection of data types represented by user-defined XML. In overly simplistic (yet didactically useful) terms, the data representing the resource lives in the service. Along with the resource, WSRF provides some useful methods for accessing the state of the resource. A client can get the values of any elements of the resource by making a SOAP call to the GetResourceProperty operation. Additionally, a client can subscribe for notification of changes to a given resource. If we are working in a pull paradigm, a client may poll for changes using GetResourceProperty, and if we are in a push paradigm, the service signals the clients of data changes, via the notification mechanisms.

Web Services provides clients access to operations. Operations are remote method calls that a Web Service allows clients to invoke via SOAP. The resource and the operations associated with it are described in an XML document format called Web Services Description Language (WSDL). Stubs are created from the WSDL which makes client interaction with the service as easy as making a local function call. More information on WS C and WSRF can be found in the WS C Core web pages [36] and in "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," [37].

E. The Embedded Service

We identify the following list of internal state the frontend service must expose to the controller:

- Current client connections: The number of clients currently requesting or performing transfers.
- Max client connections: The maximum number of clients allowed to connect simultaneously.
- Available Backend Services: The number of backend services ready for use.
- Approximate load of each backend: a percentage associated with each backend that approximates how hard it is working.

We defined these states in a WSRF compliant WSDL document as resource properties. The WSDL documents serves as an interface definition for both clients and services. We use it to generate stubs for the controller programs and also to create a service for exposing these resource properties. Because it is important for the service to be light weight and have very low overhead interactions with the frontend service we used WS C Core.

WS C Core is a light weight hosting environment that can be embedded into the GridFTP frontend service. WS C Core compiles to native code allowing it to be linked into existing applications. From the WSDL we generate C bindings which handle the conversion of SOAP messages into C data structures. Using the bindings we create an embeddable service that observes the internal state of the GridFTP server and sets the resource properties appropriately as the state changes. The service is linked into a shared library which is loaded at runtime in the presence of an option. If loaded, the WS service starts providing information; if not loaded, the GridFTP server is left entirely unaffected.

F. Controller

The controller is a client to the embedded service. Via WSRF, it gets notifications from the frontend service when any of the resource properties defined above change. Stubs are generated from the WSDL that make each notification a simple function call. When a client connects or disconnects, or when a backend is added or removed from the transfer pool, or when the frontend service's exposed state otherwise changes, the controller is notified asynchronously. It then uses this information to decide when to start new backend service instances and when to kill existing ones.

The controller also observes the amount of time that any backend is idle and may, depending on the policy that it implements, decide to kill the backend service so that the associated node can be returned to the compute resource pool. The GridFTP server is fault tolerant and can handle unexpected backend service termination. Termination of one backend has no effect on any other, nor does it affect the frontend service. However, it may leave a client that requested a transfer with an error. Such errors are not a problem since GridFTP transfers can be restarted in such a way that little data needs retransmission.

The controller uses GRAM to start up new backend service instances via the LRM. GRAM is used because it is a convenient and easy to use interface to many different LRMs. The ease of use is important for enabling the creation of site specific policies. With GRAM starting up a new backend is a simple function call available in many programming languages. Additionally GRAM is a network protocol which allows the controller to run on an entirely separate system from that of the LRM. No CPU cycles or other resources are diverted from the frontend and thus we can expect that the performance of the transfer service will be minimally affected.

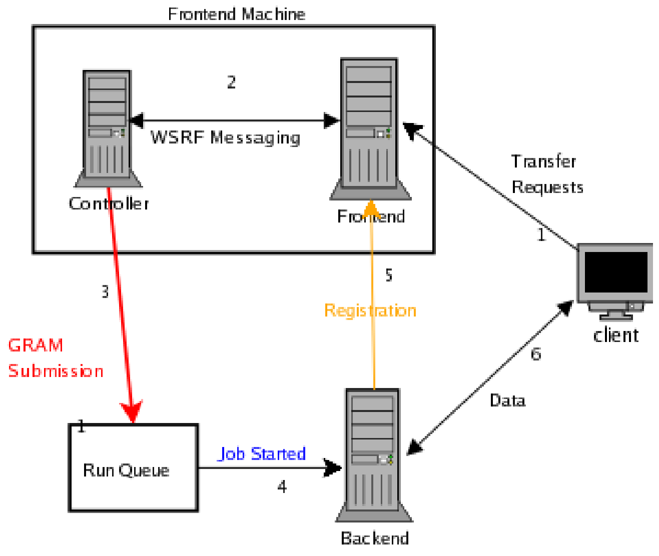

Figure 3: Event Sequence

Figure 3 shows the event sequence of our architecture. A client connects to the frontend service and requests a transfer (line 1). If an unused backend service instance has already been registered with the frontend service it is used, if not the frontend waits for one to be registered. The controller receives a WSRF notification when the client connects telling it that the state of the frontend service has changed (line 2). The controller may then use the GRAM interface to the LRM to start up a new instance of a backend service (line 3). The newly requested backend service will not start immediately, and the time it takes to start will vary with contention in the LRM's wait queue. When it does start, it registers itself with the frontend service (line 5). The frontend service can now use it and the clients can begin transferring data (line 6).

The decision as to when to start up a new backend service instance is entirely up to the logic of the controller. In some cases the controller will wait until a client request prompts the need for a new backend. In other cases it may start a new instance based on a prediction algorithm guessing that a transfer request will be made in the near future. Its modular placement in the architecture allows it to be noninvasive to the rest of the system. It simply observes the running state and injects new resources when its policy allows.

V. Experimental Studies

To establish a proof of concept for this architecture we developed a prototype system. We implemented all modifications to the GridFTP server specified above. We created controllers to implement some basic policies, described in detail below. All tests were run using the University of Chicago TeraGrid system, in which each node has a 1.5GHz Dual Itanium processor, 4 GByte RAM and Gbit/s Ethernet. We use globus-url-copy as the client to the transfer service in all of our experiments.

A. *Simple LRM*

The University of Chicago TeraGrid system operates a PBS LRM with which our prototype can interact. However, the cluster is generally well used and the wait times for nodes can be high. As this wait time is entirely dependent on outside variables, we could not use it in controlled experiments. To address this problem, we implemented our own simple LRM for the purposes of experimentation. We then simulated a compute cluster by leasing TeraGrid nodes from PBS and using those nodes for our own purposes within our simulated LRM.

We acquire one node from PBS to be the service provider. On it we run the GridFTP frontend service, the controller, the GRAM service, and the simple LRM. An additional set of nodes is acquired and designated as the compute/transfer nodes for our simple cluster. One interface to the queue is a script to be passed to the command line of the job to remotely run. The node on which the user job will be executed is chosen via round robin selection. The job is then remotely executed on that machine via ssh [38]. In order to simulate the nontrivial queue traversal time that is typically encountered in queuing systems, the script will sleep before remotely starting the user's job. The sleep time is set in a configuration file that can be changed dynamically.

In addition to providing direct access to our simple LRM by calling the submission script, we also provide a GRAM interface. The GRAM ManagedJobFactoryService is designed to be configured to interact with a variety of LRMs, including LSF, PBS, and others. We extended it to run against our simple LRM as well. This way all of the XML messaging overhead to the simple queue is the same as it would be to any other queue. The only difference is the remote execution via ssh versus the proprietary mechanisms used by the specific LRM.

B. *Policies*

We evaluate our prototype system using four different policies, which we now describe. Many other policies can be imagined (and can be implemented in our architecture); we choose these policies to provide insight into the efficiency and applicability of the architecture.

1) *Zero Idle*

The zero idle policy does not allow a backend node to be idle for any time. Thus, nodes can neither be pre-fetched nor cached. Each time a client connects to the frontend service and requests a transfer, a request for a new backend service instance is submitted to the LRM. When the transfer completes the backend service immediately ends and the node it was running on is released to the LRM for use in scheduling other jobs. When using this policy, the amount of time it takes to start a transfer is entirely dependent upon the wait time of the LRM and amount of time it takes for the notification to reach the controller.

2) *Infinite Idle*

The infinite idle policy is the exact opposite of the zero idle policy in terms of how it handles idle backend services. A set of backend service instances are started in advance and added to the transfer pool. They are allowed to be idle for the lifetime of the frontend service and therefore the nodes on which they run are never returned to the computational pool. With this policy there will be idle backend services ready to handle a client request at all times, until the system reaches maximum capacity.

3) *N-Idle*

The N-Idle policy is a simple approach to prefetching and caching backend transfer nodes. The policy attempts to have N backend services idle at all times. When the system starts up, N backend services are created. These backend services are idle until a transfer is requested. When the transfer is requested one of the N backend services is immediately allocated to service the transfer request. Since there are now N – 1 idle backend services, the controller implementing this policy requests the creation of a new backend service in an attempt to maintain N. Since creation of a new backend service takes time, there will not always be N idle, however the sum of idle and requested backend services will always equal N.

4) *Time Cache*

The Time Cache policy allows a backend service to continue running and servicing transfer requests for the entire wallclock duration. Unlike the Zero Idle policy which returns the node to the LRM immediately upon completion of its first and only transfer request, the Time Cache policy will continue to wait for additional transfer requests until its lease from the LRM times out. The assumption is that the time and effort required to obtain a node from the LRM is greater than that of a single transfer and thus it is worth caching for its wallclock time. The maximum amount of idle time is never greater than the wallclock time.

The first two policies represent two extremes. Infinite idle represents an optimal policy from the perspective of connection time. Backend services are always ready, and so our architecture adds no additional overhead. Infinite idle is equivalent to the GridFTP proxy server without our modifications, with backend services statically assigned to a frontend and always available to it. In contrast, zero idle provides the worst case scenario for connection time. Since backend services must be created for every transfer request, the maximum amount of latency is added to every connection. Conversely, since backend services are never idle, zero idle provides the best possible case when minimizing idle transfer resources, and infinite idle is the worst possible case. These two policies are base cases for evaluating the architecture.

The N-idle policy is a hybrid of zero-idle and infinite-idle. It attempts to optimize connection times by having backend services immediately available for transfers while at the same time putting an upper bound on idle resources. While the number of idle backend services is bounded, the number of working nodes is not, and thus the policy can scale up to support high transfer loads. Because the value of N is configurable, the site administrator can decide on an appropriate balance between idle resources and low connection times.

Administrators may want more sophisticated policies to address specific needs. Our architecture provides access to information that allows arbitrarily complex policies to be defined, based on intimate knowledge of the transfer service and a site's history. We leave the exploration of such policies for future work.

C. Metrics

We use three criteria to evaluate the system: site bandwidth utilization, connection time, and user bandwidth. The most important metric is site bandwidth utilization. Our main goal in defining this architecture is to use idle network cycles. If we can provide transfer services with access to large amounts of bandwidth, we believe existing applications will make use of it and additional use cases will be discovered, and thus we are successful. In our experiments we measure bandwidth by running daemons on the client machines. These daemons read the raw network usage from the Linux /proc filesystem [39] and use that information to track the number of bytes the machine has transferred over the life time of our experiments. We take samples every second and then post process the results to calculate bandwidth over the experiments time interval. Because only our jobs are running on the client machines we know the bytes transferred are a result of our experiments. Our goal is to utilize all available bandwidth so we must count bytes sent under the IP protocol stack. We are not measuring end to end throughput. If resends are needed by TCP due to packed loss we count the resends as additional bytes. Our results would otherwise appear to have used less network resources than were actually used. This issue has very little effect because all experiments are done on an under utilized LAN where packet loss is infrequent.

The second metric is connection time. We define connection time as the latency from when a client first connects to the transfer service until the time the transfer begins. Before a transfer can begin the frontend must find an available backend service. In the worst case, this time will include the latency of the WSRF notification message updating the controller that the frontend services state changed, the entire length of the LRM's wait queue, and the latency of the backend service request to the LRM. This time has the potential to be high and could have a significant effect on overall system throughput. Further, while this architecture is targeted at batch transfers, sites may wish to support interactive transfer services as well, in which case connection times need to be low. We measure the connection time with a plug in to globus-url-copy. The time interval begins immediately before the TCP connect function is called, and ends when FTP login message "230 User logged in" [40] is received.

The last metric is achieved user bandwidth. Our goal is to provide a transfer service that can exploit large amounts of idle network cycles. In order for this service to be useful, we must have applications that wish to use it. To make the service attractive to applications we must be able to service each transfer request with enough bandwidth to satisfy its needs. How much bandwidth is needed is a subjective measure. Ideally the application would be the bottleneck in the transfer and not the transfer service. As 100 Mbit/s Ethernet cards become outdated and Gbit/s Ethernet cards become standard, even on laptops, we aim to provide near Gbit/s transfers to clients.

VI. Results

We present results for five sets of experiments using our simple LRM. The first experiment evaluates the validity of the LRM we used by comparing it to the PBS system used on the UC TeraGrid. We then measure the connection time as a function of increasingly heavy loads. This experiment is designed to analyze the overhead introduced by our system. We similarly measure the connection time to see how the LRM wait queue times affect it. Following this we measure how our various policies behave under lighter, and more naturally occurring, client loads. The final experiment shows the bandwidth that our system can consume and the throughput it can deliver to individual clients.

In all experiments we took care to prevent any one iteration of tests from affecting the next iteration. For the zero idle and infinite idle policies this is not a problem. Those two basic policies are not affected by previous client access patterns. However, the n idle policy can be greatly affected by previous access patterns. If a previous test is still holding a backend service, or if the backend service that was started to replace a used backend service has not yet begun, our results could be corrupted. To protect against this, we verified that the system was in a stable state before running any iteration of our tests. When one test finished we waited for all resources associated with previous transfers to be cleaned up and for all requested backed services to start before we began the next iteration of tests.

A. Simple LRM Validation

Our first results validate our simple queuing system. Since we intend that our architecture work with standard and widely used LRMs such as PBS, we must show that our system provides a reasonable approximation of the behavior of a standard system like PBS. The difference in which we are interested is the amount of time it takes to execute a job once a node is available. With our simple queue, this is the time it takes to run ssh, which involves simply connecting to and authenticating with a daemon on the remote node, and then executing the job via this connection. To measure this cost we used ssh to remotely run a dummy job, "/bin/true," one hundred times in a row. We took that time and divided it by one hundred to determine the average.

With a full featured LRM there is much more involved. The submission program must connect and authenticate with the LRM's gateway service. Once the system accepts the submission it must add the submission to its list of jobs waiting to run. Then, it must apply its scheduling algorithm to determine if and where the job can be scheduled. Even in situations where there are resources immediately available, these steps can take some time. We measured this time interval using PBS on the TeraGrid. We took the time when starting the submission program and again when the job actually started. This interval defined our access time. We repeated this one hundred times and again took an average. Because we are using a live queue we need to make sure that our jobs could be scheduled right away. We did not want to end up with inflated times due to a congested wait queue. To accomplish this we obtained an advance reservation for the node in question, thus guaranteeing that it was always ready for our job.

The results of these measurements are shown in Table 2. Since PBS is much more complicated than our simple LRM, the average access time is orders of magnitude larger. The simple queue access time is never more than a second while the PBS, times range from 1 second to 64 seconds with a mean of 25.28 seconds and a standard deviation of 10.7 sec. To account for this difference in average times, we have our simple LRM wait 25 seconds with each submission in addition to the configured wait time.

Table 2: Queue access times

Queue	Avg Time (seconds)
PBS	25.28
Simple queue	0.79

B. Connection Time

The next set of experiments measure the connection time. We configure our simple queue with four nodes in its resource pool, and compare performance for three policies: infinite idle, zero idle using a GRAM interface, and zero idle with direct access to our simple LRM. Each client request involves a one-byte file transfer from /dev/zero to /dev/null. This strategy minimizes the use of the network for file transfer so that connection time is not disrupted by data transfer traffic, and eliminates any potential bottlenecks due to filesystem usage.

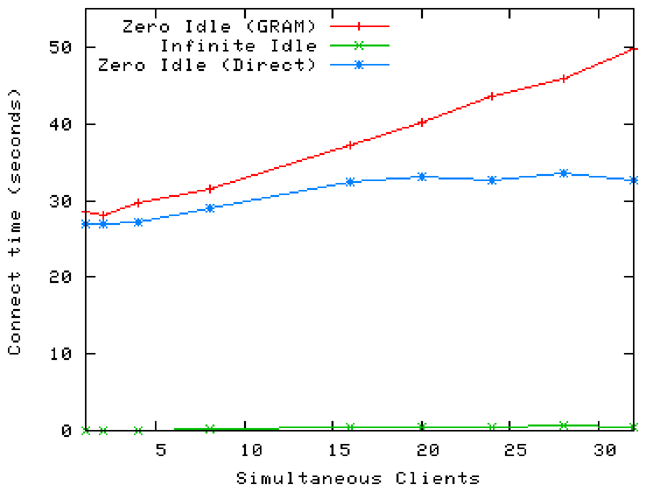

Figure 4: Connection Time

We evaluate the connection time under an increasing workload of simultaneous client requests, ranging from one to thirty-two. This allows us to analyze connection times as they vary from light client loads to heavy client loads. One client at a time is the lightest possible load that is interesting for this experiment. Thirty-two clients with Gbit/s NICs all transferring at the same time would use over 30Gbit/s of the bandwidth, which is heavy load on the spectrum in which we are interested.

The graphs in Figure 4 show the results of this experiment. While the infinite idle policy overhead grows as the client load increases, it is never significant. As the graph shows it only minimally rises above the X-Axis when compared to the other two and effectively can be considered zero. The difference between the zero idle policy with GRAM (Zero Idle GRAM) and zero idle with direct calls to the simple queue (Zero Idle Direct) expose the latency added by the GRAM Web Service interface. In both cases, the overhead introduced increases with client load, however in the direct case the overhead increased only slightly beyond the wait time in the queue and in the GRAM case it increases steadily with load. We attribute this effect to XML serialization within GRAM and the single point of contact for the queue job submission. This additional overhead becomes irrelevant as wait times increase. The ease of use of the GRAM API and the convenience of the abstraction to many LRMs outweighs the added overhead for our purposes.

C. Wait Time

We next measure the effects of wait time. Figure 5 shows the results. As is expected, connection latency increases proportionately with wait time. Sites that are interested in providing an immediate and interactive transfer service will find the usefulness of the architecture decreases as wait time increases and will therefore be interested in the n-idle policy described below. Although our architecture is targeted at batch transfer services, we propose solutions for interactive services as well. The first and simplest way is to allow preemption. If the controller can get immediate access to a node by preempting or suspending other jobs, then the queue wait time is no longer relevant. While preemption can be handled with high priorities, and backfilling can allow faster access to the queue for short running jobs of this nature, it may not provide immediate access. Suspending running jobs can provide immediate access; however this is a complicated issue and typically requires virtual machines. Since this is not always possible, we use pre-fetching and caching of backend services.

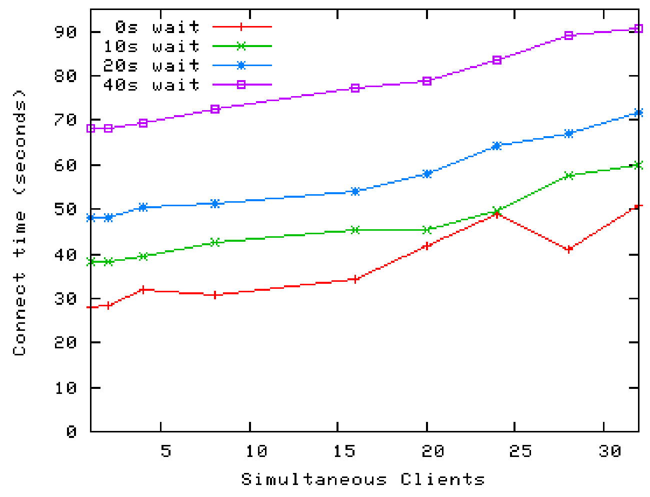

Figure 5: The Effect of Load on Connect Time

We perform the same study using the N-Idle policy and set N=4. Figure 6 shows the results. The Connect time for up to four simultaneous clients is low, no matter what the wait time of the queue. As the number of clients connecting exceeds the value of N, the connection time increases similarly to the zero-idle policy yet remains lower on average due to the immediately available backends servicing up to N requests.

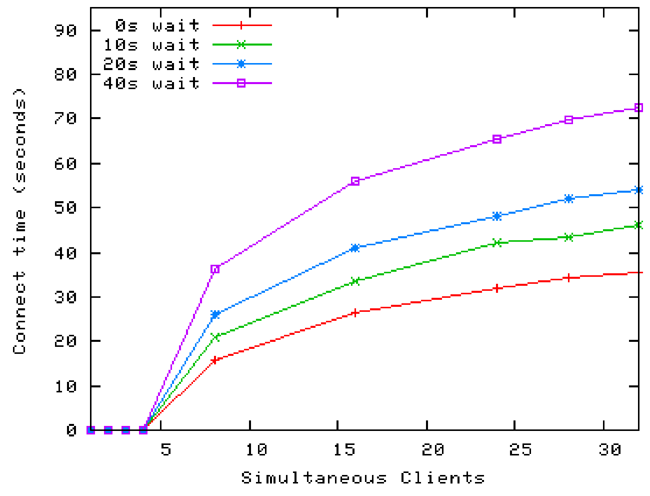

Figure 6: Connection Time for Various Waits

1) *Lighter Loads*

Many clients connecting at the exact same time constitutes a heavy load on the system. In practice this situation will typically not occur: there will be some spacing of client requests and thus a lower load. In this experiment we measure the effects of the n-idle and zero idle policies under lighter client loads.

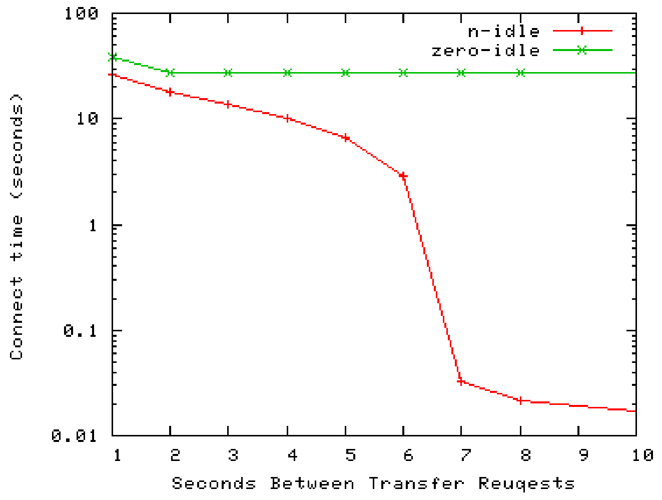

Figure 7: Lighter Loads

Figure 7 shows the results of an experiment in which we run various client loads against both the zero idle policy and the N=4-idle policy. The graph shows the connection time as a function of the delay between client requests. The zero-idle policy maintains a steady delay that is equal to the backend service start time. The N-idle connection times start off high because the system is overloaded; however they drop off quickly as the interval between requests increases. As requests are spaced out enough they allow enough time for previous requests to finish before the next starts. Recall that the time it takes to finish a request involves the time through the wait queue as well as the transfer time. With the N-Idle policy N of the requests can be serviced immediately and thus finish very quickly. Once finished the backend service's node is returned to the LRM's compute pool. The backend service that was started to replace the consumed backend will not be ready until it works its way through the wait queue.

The point at which the connection time drops significantly in this case is seven seconds between transfers. The zero-idle line shows that the node access time is slightly below 30 seconds. With seven seconds between transfers and four pre-fetched nodes the system has 7*4 = 28 seconds to acquire a node before a new client will be stalled waiting for it. This results in the following formula:

N = queue wait time / time between clients

Network load prediction research can be used here to determine a value for N dynamically as the system runs based on the expected time between transfer requests.

Instead of looking at the results to find an ideal value for N we can alternatively look at the ideal time between transfer requests. If client connection requests are intercepted and spaced far enough apart, the overall connection time could improve for short transfer times or for bursts in client transfer requests. This leaves interesting open questions relating to transfer time prediction. We leave this for future work.

D. Throughput

As stated, the goal of this proposed architecture is to maximize bandwidth usage for batch data transfers. The delay time to the start of the transfer is secondary to maximizing utilization of available bandwidth. However, the wait time to acquire a node can affect the overall system throughput. Figure 8 shows this.

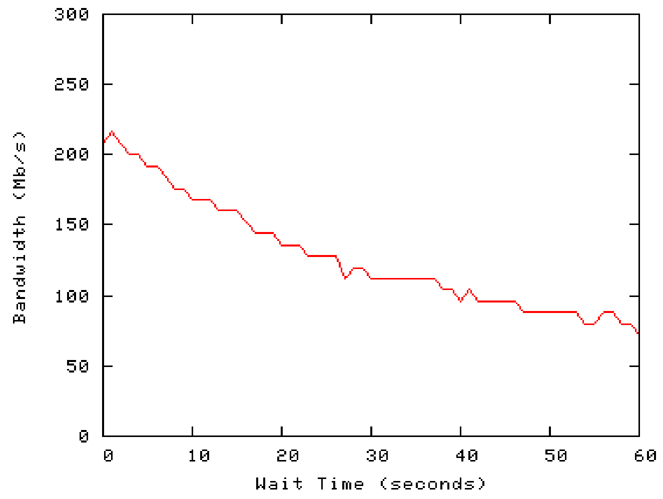

Figure 8: Single Client Throughput

In the experiment we used the Zero-Idle GRAM policy and repeatedly ran a single client transfer request of a 1 GByte file, one hundred times in sequence. We measured the overall throughput used by the system as a function of an increasing queue wait times. The maximum possible bandwidth available for consumption is one Gbit/s. Our results show that we only achieve 20% of the maximum with no wait, and as the wait time increases the bandwidth utilization decreases. The histograms in Figure 9 provide insight into why. Peak transfer rates are reached only over the small interval after a backend service is allocated to a transfer and before the transfer is finished. The peak times are followed by idle intervals spanning the wait time plus the connection time. The longer the wait time, the longer the network idle time and thus the lower overall throughput.

E. Caching

One approach to eliminating the time gaps between transfers involves using the *Time Cache* policy. We demonstrate this with an experiment using four backends and four client machines. A client is run on each machine transferring a series of one Gigabyte files in serial. Each backend service has a wallclock time of one minute.

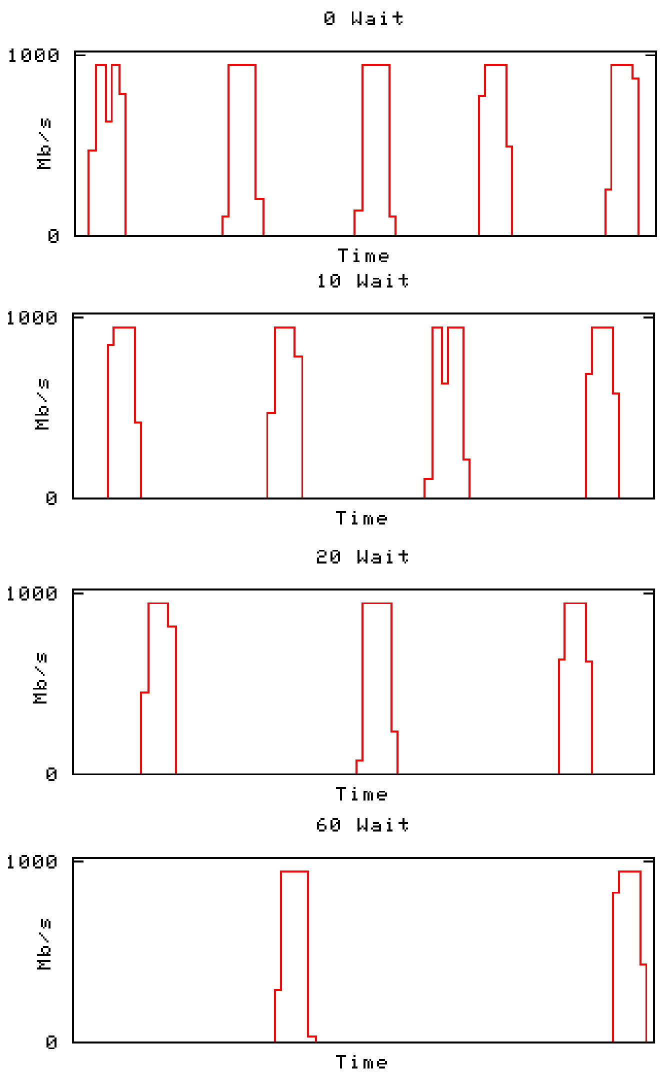

Figure 9: Single Client Transfer Histograms

The graph in Figure 10 shows the results of the first 60 seconds of this experiment. Each colored bar represents the through put achieved by a client machine in a two seconds time step. Figure 10 shows the network is only idle at the beginning when we are requesting new nodes. The remainder of the time the backend services are cached and therefore ready to perform transfer requests immediately. Were the experiment to continue on for another minute we would see another drop in performance as the cached backends wallclock time expired. This ratio of cached time to fetch time can be adjusted by requesting longer and shorter wallclock times. Longer wallclock times will result in higher overall bandwidth utilization under heavy client loads. Conversely, shorter wallclock times reduce the potential backend idle time under light client loads.

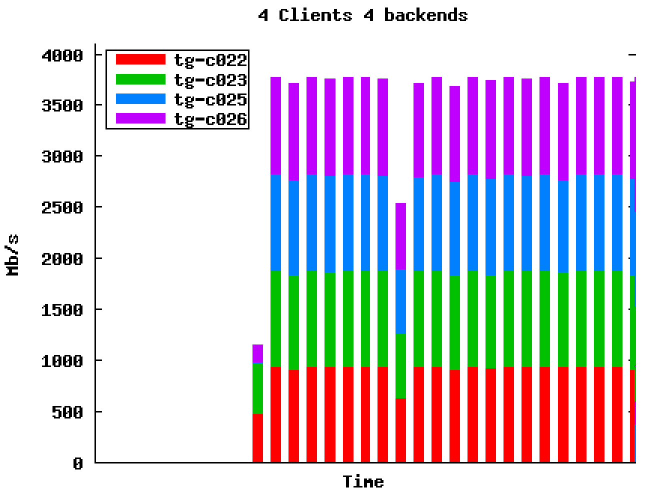

Figure 10: Time Cached Bandwidth

The results of this experiment show that our architecture performs well according to the bandwidth utilization metric as it can scale up to network speeds. Figure 10 also shows how the architecture is measured against the final metric of achieved user throughput. Each client receives transfer rates of near gigabit speeds which is the maximum available to it.

VII. Conclusions and Future Work

We have designed and implemented an architecture that allows computational sites to allocate transfer nodes dynamically in response to client request load. A site can use this architecture to utilize more of its network resources without leaving compute nodes idle. By providing a transfer service with a significant amount of available bandwidth, sites can better service their communities and increase their user base. Further, causing a glut of useable network cycles can enable existing and future higher level applications to take advantage of them. We envision this work as a launching pad for research into such applications.

The majority of related work in this area focuses on storage system scavenging. Bandwidth usage is a part of a distributed storage system but when the focus is on the storage as opposed to the bandwidth itself different attributes get accentuated. Our goal is to focus strictly on idle bandwidth cycles to enable present and future applications that move data sets around effectively, making the network itself a storage system or capitalize on it for efficient replication. By providing a service with a scheduled interface to predictable amounts of bandwidth we can enable an array of applications. Some examples include:

- Fast Write Services: An application may need to get data out of its memory buffers as quickly as possible so that more data can be brought in for processing. The application may or may not care about the location the final storage point of the data, because its primary focus is on writing the data to whatever sink can receive it the fastest. If the fastest endpoint is not the final endpoint, an out-of-band service can later move it to its final destination.
- Replica Services: It can be advantageous to have multiple copies of datasets. Replicas not only provide redundancy in case of loss but can also assist with load balancing and transfer optimization. Clients looking to read data can be directed to any replica, thus lightening the load on other sources. Clients can also gain performance increases by reading different parts of the data from multiple replica locations. Applications that provide these replica services could use the bandwidth we provide as another point of replication.
- Co-scheduling: The architecture allows for users to submit backend jobs directly to the LRM. This is an important aspect of this architecture that warrants additional comment. This was not directly studied in the experimental studies section largely because its performance is tied to that of the run queue and its value is not in connection time or transfer time but rather in the provided scheduling information. By allowing users to start their own backends we give them all the resources and features of the LRM. This includes most notably start time prediction and notification. Further, once the backend starts it will have an entire dedicated resource with a predictable portion of the sites bandwidth. With this information applications can begin to co-schedule [41] both sides of the transfer.

Our prototype evaluation has shown that this system can be effective. It takes advantage of otherwise idle network cycle without causing unnecessary delays to the compute wait queue. When faced with heavy client loads our system can dynamically scale up to peak client transfer demands and release the resources as the load decreases.

While our system is targeted at batch transfers, for which connection time is less crucial, our evaluation shows that the architecture can provided a reasonable interactive service under moderate client loads. We achieve this performance by pre-fetching nodes and/or caching nodes by allowing them to remain idle after completing transfers and before new transfers start. Idle resources do unnecessarily detract from the compute queue but we have show how to put limits on the amount of idle resources without preventing the system from scaling up under heavier loads.

For batch transfers we have shown that the system can scale up to high levels of network utilization. Since we used the zero-idle policy in our experiments the high bandwidth utilization that we achieved did not come at the expense of idle computer resources. While we did divert resource from the run queue for transfers we did not leave them idle.

In the future we would like to study the effects of a steady pulsing of clients in more detail. As we found in our evaluation of the n-idle policy, when client transfer requests are spaced out over time, the connect times drop significantly. We would like to explore specifically how and why this happens. It is possible that we could intercept client requests before sending them to the GridFTP frontend service, and then allow connections only at regular intervals using token bucket or some similar algorithm. While this strategy would impose a delay on the connection time, the delay may be low, regular, and predictable, and thus acceptable. A key aspect of this study involves predicting transfer times.

In additional future work we plan an in depth study of various caching policies. The effects of caching policies on wait time and idle time under a variety of real and simulated client loads would grant further insight into the usefulness of this architecture. As the production systems use this architecture we can simulate actual work loads from system logs to complete this study.

We would also like to explore this problem from a different perspective. Instead of starting a backend service on a node and fully utilizing all of the node's resources, we could attempt to share co-located transfers and user jobs on a single node. As we noted earlier, at full transfer speeds co-location is too disruptive. However, if our transfer rates were throttled down so that their network and CPU utilization did not noticeably detract from a user's job, we could indeed co-locate them. One way to accomplish this is to set a maximum bandwidth cap to a rate significantly lower than the nodes NIC speed. This will prevent the network card from being over utilized and also prevent the CPU from being overloaded with packet switching tasks. Since the bandwidth of any one node will be greatly limited we will use many nodes in a coordinated striped transfer to achieve higher throughput rates.

Finally, we plan to enhance our system to take advantage of GridFTP's striped transfer abilities. In the system presented in this paper we allocate a single backend service instance to every data transfer request. To provide an immediate service to clients we will allocate two backend service instances to each transfer request. The first will be shared by all transfers. It will be running at all times waiting for to service requests. This will allow for transfers to immediately start but because the node's NIC will be shared by all, the transfer rates will be low. A second backend service instance will be acquired from the LRM and added as a second stripe when it begins execution. While waiting for the second to start some progress on the transfer can be made and when it does start transfer rate will be at full speed. This feature will allow for significant improvement over connection times and slight improvements in transfer performance.

REFERENCES

[1] "TeraGrid web page," http://www.teragrid.org.

[2] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow, "The NFS Version 4 Protocol," 2000. [Online]. Available: citeseer.ist.psu.edu/shepler00nfs.html

[3] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters," in Proceedings of the 4th Annual Linux Showcase and Conference. Atlanta, GA: USENIX Association, 2000, pp. 317–327. [Online]. Available: citeseer.ist.psu.edu/294296.html

[4] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in Proc. of the First Conference on File and Storage Technologies (FAST), Jan. 2002, pp. 231–244. [Online]. Available: citeseer.ist.psu.edu/schmuck02gpfs.html

[5] A. Chervenak, R. Schuler, C. Kesselman, S. Koranda, B. Moe, "Wide Area Data Replication for Scientific Collaboration,." in Proceedings of 6th IEEE/ACM International Workshop on Grid Computing (Grid2005), November 2005.

[6] D. Wessels and K. Claffy, "ICP and the Squid Web Cache," IEEE Journal on Selected Areas in Communication, vol. 16, no. 3, pp. 345–357, 1998. [Online]. Available: citeseer.ist.psu.edu/wessels97icp.html

[7] IPerf. http://dast.nlanr.net/Projects/Iperf."

[8] "globus-url-copy" http://www.globus.org/toolkit/docs/4.0/data/gridftp/rn01re01.html

[9] J. Bresnahan, R. Kettimuthu, and I. Foster, "XIOPerf : A Tool for Evaluating Network Protocols," in Proceedings of the Third International Workshop on Networks for Grid Applications, 2006.

[10] "Netperf." http://www.netperf.org/netperf/NetperfPage.html .

[11] "GT4.0:GridFTP:" http://www.globus.org/toolkit/docs/4.0/data/gridftp/"

[12] 12 K. Czajkowski, D. F. Ferguson, J. F. I. Foster, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe, "The WS-Resource Framework," 2004.

[13] S. E. Shadan, B. H. Far, and J. Cheng, "Dynamic Mirroring for Efficient Web Server Performance Management," The IEICE Transactions on Communication, vol. E85-B, no. 8, pp. 1585–1595, 2002.

[14] S. Vazhkudai, X. Ma, V. W. Freeh, J. W. Strickland, N. Tammineedi, and S. L. Scott, "Freeloader: Scavenging Desktop Storage Resources for Scientific Data," in SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing. Washington, DC, USA: IEEE Computer Society, 2005, p. 56.

[15] D. Jackson, Q. Snell, and M. Clement, "Core Algorithms of the MAUI Scheduler," Lecture Notes in Computer Science, vol. 2221, pp. 87–??, 2001. [Online]. Available: citeseer.ist.psu.edu/jackson01core.html

[16] A. W. Mu'alem and D. G. Feitelson, "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling ". IEEE Trans. Parallel & Distributed Syst. 12(6), Jun 2001.

[17] L. Yang, J. Schopf, and I. Foster, "Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decisions in Dynamic Environments," 2003. [Online]. Available: citeseer. ist.psu.edu/yang03conservative.html

[18] W. Smith, V. Taylor, and I. Foster, "Using run-time predictions to estimate queue wait times and improve scheduler performance," in Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph, Eds. Springer Verlag, 1999, pp. 202–219.

[19] W. Smith, I. Foster, and V. Taylor, "Predicting Application Run Times Using Historical Information," Lecture Notes in Computer Science, vol. 1459, pp. 122–??, 1998.

[20] S. Vazhkudai and J. M. Schopf, "Using Disk Throughput Data in Predictions of End-to-End Grid Data Transfers," in GRID '02: Proceedings of the Third International Workshop on Grid Computing. London, UK: Springer-Verlag, 2002, pp. 291–304.

[21] S. Vazhkudai and J. Schopf, "Predicting Sporadic Grid Data Transfers," 2002. [Online]. Available: citeseer. ist.psu.edu/vazhkudai02predicting.html

[22] R. Wolski, "Dynamically Forecasting Network Performance Using the Network Weather Service," Cluster Computing, vol. 1, no. 1, pp. 119–132, 1998.

[23] Keahey, K., I. Foster, T. Freeman, X. Zhang, D. Galron, "Virtual Workspaces in the Grid", Europar 2005, Lisbon, Portugal, September, 2005.

[24] R. L. Henderson, "Job Scheduling Under the Portable Batch System," in IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing. London, UK: Springer-Verlag, 1995, pp. 279–294.

[25] S. Zhou, "LSF: Load Sharing in Large-Scale Heterogenous Distributed Systems". In Proc. Workshop on Cluster Computing, 1992.

[26] IBM Corporation, IBM LoadLeveler: User's Guide, 1993.

[27] "GT4.0WS_GRAM," http://www.globus.org/toolkit/docs/4.0/execution/wsgram/

[28] W. Smith, I. Foster, and V. Taylor, "Scheduling With Advanced Reservations," pp. 127–132.

[29] "The Globus Toolkit. http://www.globus.org/toolkit ."

[30] 30 "Gridftp : System Administrator's Guide. http://www.globus.org/toolkit/docs/4.0/data/gridftp/admin-index.html ."

[31] [31] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, "A Security Architecture For Computational Grids," in 5th ACM Conference on Computer and Communications Security Conference, 1998, pp. 83–92.
[32] [32] G. Geist, J. Kohl, and P. Papadopoulos, "Cumulvs: Providing Fault Tolerance, Visualization, and Steering of Parallel Applications," 1996.
[33] [33] P. Dinda and D. O'Hallaron, "An Extensible Toolkit for Resource Prediction in Distributed Systems," 1999.
[34] [34] R. Wolski, "Dynamically Forecasting Network Performance Using the Network Weather Service," Cluster Computing, vol. 1, no. 1, pp. 119–132, 1998.
[35] [35] "A Monitoring and Steering Framework Using WS C Core," http://www-unix.mcs.anl.gov/ bresnaha/gmonsteer.pdf.
[36] [36] "GT 4.0: C ws core. http://www.globus.org/toolkit/docs/4.0/common/cwscore/index.pdf
[37] [37] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," Int. J. High Perform. Comput. Appl., vol. 15, no. 3, pp. 200–222, 2001.
[38] [38] "Openssh," http://www.openssh.com/
[39] [39] "An Overview of the Proc Filesystem, http://linuxgazette.net/issue46/fink.html ."
[40] [40] "RFC 959 - File Transfer Protocol, http://www.faqs.org/rfcs/rfc959.html"
[41] [41] K. Czajkowski, I. T. Foster, and C. Kesselman, "Resource Coallocation in Computational Grids," in HPDC, 1999.