The Virtual Service Grid: An Architecture for Delivering High-End Network Services

Jon B. Weissman and Byoung-Dai Lee

Department of Computer Science and Engineering
University of Minnesota, Twin Cities
Minneapolis, MN 55455
WWW: www.cs.umn.edu/~jon
E-mail: {jon, blee}@cs.umn.edu

Abstract

This paper presents the design of a novel system architecture, Virtual Service Grid (VSG), for delivering high performance network services. The VSG is based on the concept of the Virtual Service which provides location, replication, and fault transparency to clients accessing remotely deployed high-end services. One of the novel features of the Virtual Service is the ability to self-scale in response to client demand. The VSG exploits network- and service-information to make adaptive dynamic replica selection, creation, and deletion decisions. We describe the VSG architecture, middleware, and replica management policies. We have deployed the VSG on a wide-area Internet testbed to evaluate its performance. The results indicate that the VSG can deliver efficient performance for a wide-range of client workloads, both in terms of reduced response time, and in utilization of system resources.

1.0 Introduction

A large array of wide-area application technologies for distributed high-performance computing including scientific problem-solving environments [7][8][18][19][23], computational Grids [12][14], and peer-to-peer environments [9][16] are emerging. These technologies are beginning to support a model in which diverse services are deployed in the network and made available to users and applications. In this paper, we focus on “high-end” services, services that require high communication, computation, or both. Services are highly heterogeneous ranging from network-intensive content-delivery (e.g. interactive visualization) to compute-intensive high-performance computing (e.g. coupled supercomputing, remote equation solvers, or genomic servers), to everything in between. For services that have high client demand, scalability is crucial.

The most common solutions for providing scalable network services are caching and replication. Caching and replication can promote greater reliability and improved performance as they each represent an alternate location for the service. Caching is useful when the service requested is simply a retrieval of server contents, which Web proxies and clients routinely perform. However, caching has the fundamental limitation that it is tied to data and not computation. If the service requested requires significant specialized processing (e.g. remotely solve a specific system of equations as in NetSolve [7]), then caching is not particularly helpful. Replication does not have this limitation as it allows an arbitrary service to be performed by multiple service replicas. Two types of replication are common: local and mirrored. Local replication uses a front-end service (e.g. `www.cnn.com`) that distributes the request to a replica within the same site. The service is usually implemented by a cluster of machines and disk resources within the local site [13][25]. Most commonly the client requests are sent to the replicas in a round-robin fashion to achieve load balance. The advantage of this approach is that replication is transparent to the client and performance improves for high demand services. The disadvantage is that it does not deal with network bottlenecks between a client and the local site. Mirrors address this issue by allowing a geographic multi-site distribution of replicas. The advantage of mirrors is that by positioning replicas at various places in the network it becomes possible to avoid single-site network bottlenecks and to select replicas that may be closer to clients. However, it is difficult to determine which mirror is best in a dynamic changing network, particularly if the services are heterogeneous. Network distance may not matter at all if the service is compute-intensive. Even if the service is network-sensitive, performance might be more sensitive to latency or bandwidth. For this reason, many systems leave replica selection to the client [34], or simply presume network performance is all that matters [1][3][4][5][24]. This assumption is reasonable for Web page access and content-delivery networks but not when performance is dominated by computation. A final difficulty with both replication schemes is that they are often ad-hoc, static, and added after-the-fact. Issues such as the number of replicas to create and where to place them are not well understood. Typically, a fixed pool of replicas is created based on an average expected degree of client access or worst-case assumptions. These systems are unable to adapt to a changing access pattern or network conditions. If the client load rapidly increases or decreases there is no mechanism for acquiring or releasing replicas. In addition, if performance worsens due to a sudden congestion spike between a client and the replica, the client either must suffer poor performance or manually request a new replica.

This paper presents a novel scalable architecture for the reliable delivery of high-end network services called the Virtual Service Grid (VSG). The VSG exploits network- and service-information to make adaptive dynamic replica selection, creation, and deletion decisions. We describe the VSG architecture, middleware, and replica management policies. The results indicate for two distinct workload patterns and three high-performance services, dynamic replica selection achieves superior performance to standard approaches such as random and round-robin selection. In addition, dynamic replication is shown to achieve both high resource utilization and low response time relative to static schemes. When dynamic replication is combined with a small amount of static pre-allocation, performance can be

further improved such that virtually all client requests achieve response time below a preset performance threshold. The design on the VSG is based on five motivating scenarios that are described next.

2.0 Motivating Scenarios

Scenario #1: application-dependent server selection

We illustrate the first scenario with a numeric solver service that solves a system of equations provided by the client (Figure 1). Such servers are common to several network-based scientific problem-solving environments such

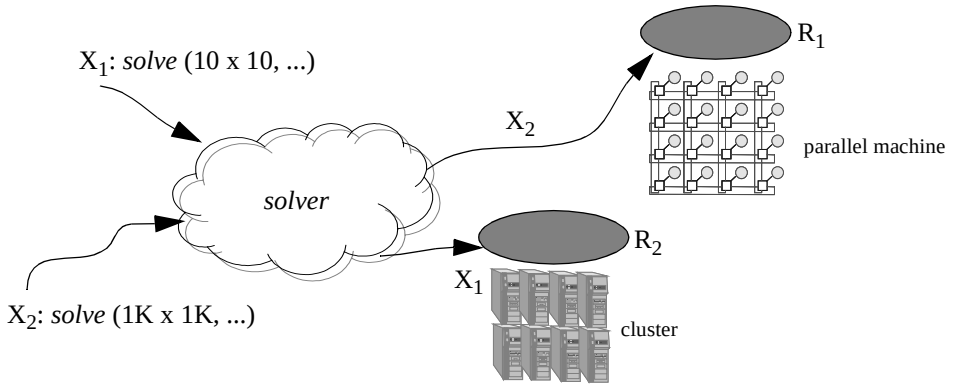

Figure 1: Scenario 1. Client requests X_i stream into the virtual *solver* service. The requests X_i can be from the same or multiple clients. Based on input parameters, requests are served by either replica R_1 or R_2.

as NetSolve [7] and Ninf [23]. Suppose that the solver service has replicas running across several machines, including a very fast small PC cluster of 8 nodes, and a large parallel machine consisting of 256 mid-range PCs. The performance of the solver service depends on the input equations. For small systems of equations, the smaller faster cluster performs best. For larger systems of equations, more parallelism can be extracted and the larger parallel machine is best.

Scenario #2: application- and network-dependent server selection

In the second scenario, a remote visualization service allows clients to interactively select a portion of a synthetic aperture radar image at different resolutions by drawing a box around the region of interest (Figure 2). The selected bytes are then delivered to the client. Examples of remote interactive visualization systems are SARA [26] and CMUs Dv project [21]. Typically, a client may begin by selecting a coarse low resolution image which might be small in size (number of bytes), and then select increasingly finer regions of greater size in order to locate a region of interest. The performance of the visualization service depends on the size of the image requested. For small images, the latency between the client and the replica may be the dominant factor, while for large images, the bandwidth between the client and the replica may be the dominant factor. If the server provides specialized filtering or processing of the images, then the compute power of the replica may also be important as in scenario #1, and if the images are on disk, disk performance may also be a factor.

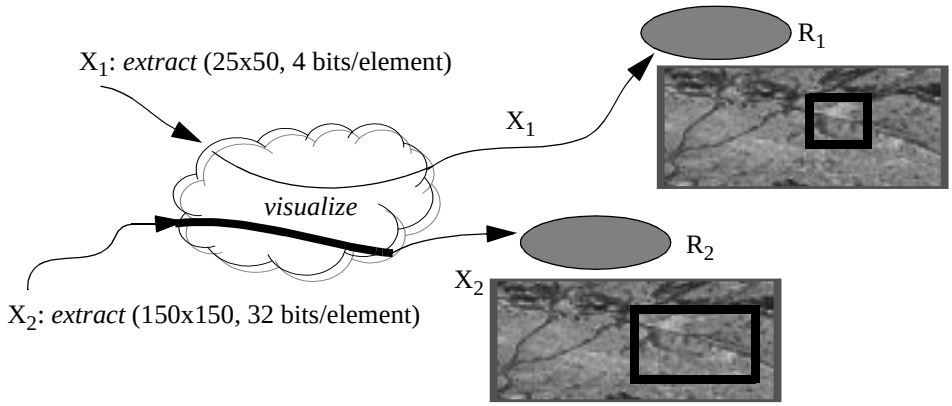

Figure 2: Scenario 2. Client requests X_i stream into the virtual *visualize* service from one or more clients. The latency sensitive request (X_1) is served by a replica R_1 that has the smallest latency to the client location. In contrast, a bandwidth sensitive request is served by R_2which has a higher bandwidth to X_2, illustrated by the darker line.

Scenario #3: client storm/calm

A particular service has become very popular very suddenly and then demand begins to fall off. For example, imagine the popularity of `www.espn.com` during the olympics or `www.microsoft.com` after announcing the availability of a critical software patch for windows version xyz. In this scenario, we would like the virtual service to "scale itself" in proportion to the perceived workload. As the demand begins to heighten, capacity grows to accommodate the requests, and when demand declines, the capacity is released, freeing up valuable system resources (Figure 3).

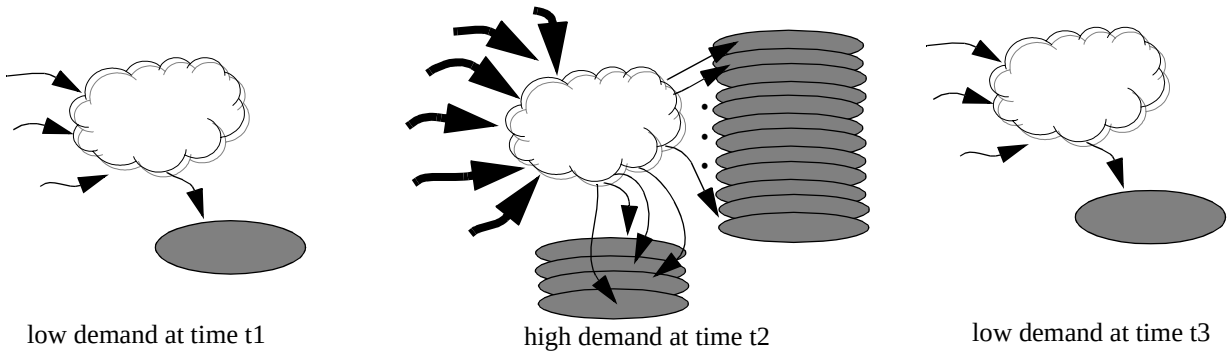

Figure 3: Scenario 3. Virtual service responds to demand spike by dynamically creating replicas and then removes replicas as demand reduces. Darker arrows indicate greater client demand.

Scenario #4: network and service adaptation

This scenario reflects the dynamic nature of the network in terms of communication and server performance. Suppose that in scenario #1, client X_1 wishes to solve a large number of small equations and a queue is forming at the preferred server replica R_2. In this case, it may be beneficial to send some of these requests to a different replica (R_1) because it may be less loaded, even if it does not offer the best runtime performance (Figure 4). Similarly, in scenario #2, a client X_1 is performing a series of latency-sensitive requests for image visualization. The preferred replica is R_1 due to the smaller expected latency. However, suppose that during the middle of these requests, a link along the net-

work path from X_1 to R_1 becomes congested, dramatically increasing the communication time. In this case, it may be beneficial to send subsequent requests to R_2. Similarly if the load on R_1 were to suddenly increase to this client (or other clients) it may also be preferable to use R_2.

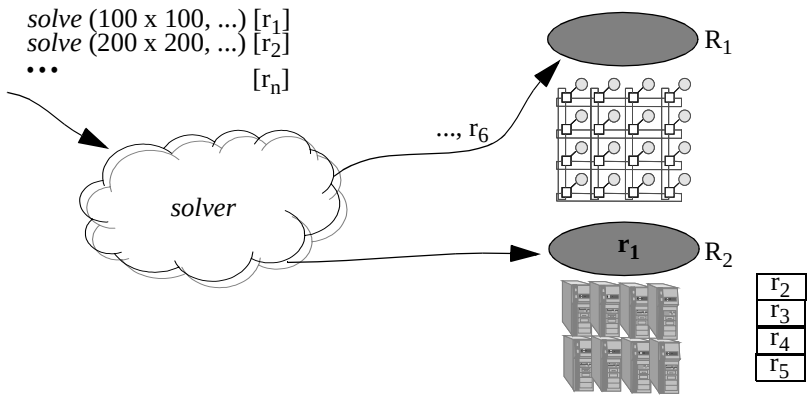

Figure 4: Scenario 4. Requests r_1 .. r_N are generated; preferred replica is R_2. As queue grows on R_2, requests begin to flow to R_1.

Scenario #5: fault transparency

In the final scenario suppose one or more replicas become unavailable either due to replica crash, overloading at the replica machine, or network outage. Ideally, the client should not be aware of such failures even if it is waiting on a current request that has failed. Fault transparency is an extreme form of network adaptation, but it is more difficult because here the adaptation boundary must be within a request, while in scenario #4 it is at the start of a request. It is unacceptable if the client is blocked waiting on request that will never complete due to a failure that occurs during the processing of the request (Figure 5).

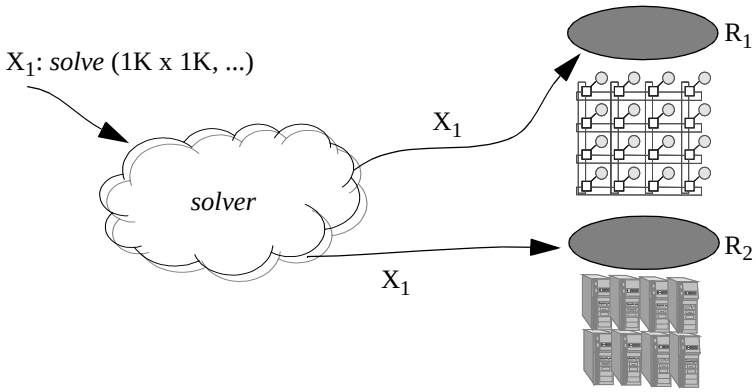

Figure 5: Scenario 5. During processing of request X_1, replica R_1 goes down. System regenerates request to replica R_2.

3.0 Virtual Service Grid

To achieve this vision, the virtual service must be scalable, predictable, adaptive, and efficient. The virtual service is an abstraction of a network service that is being designed from first principles. The term is an analogy to vir-

tual memory of computer systems. Virtual memory creates the illusion that the user owns the entire memory of the computer, while unbeknownst to the user, the operating system is smartly multiplexing, adapting, and sharing memory among many applications. The degree of sharing is controlled so that its impact is undetectable to the application. User programs also do not directly manipulate physical memory address but virtual addresses which gives the system greater flexibility. The virtual service creates the illusion that the user has much service capacity as is needed. But in reality, server and network resources are finite, and must be multiplexed and shared across multiple clients. As with virtual memory, client programs do not directly access specific replicas through network addresses, but rather through virtual addresses provided by the virtual service. The virtual service is ***stateless*** to provide greater reliability and scalability. Stateful virtual services are the subject of future work. Virtual services are contained with a VSG which provides support for their execution. The VSG consists of three primary components: system architecture, scalable information, and replica management.

3.1 VSG Architecture

The VSG architecture spans a mesh of sites within a wide-area network available to run replicas and VSG system components. Each site contains a number of host machines. Replicas and VSG components run on host resources within the site. The VSG contains four components: replica managers (RM), group managers (GM), client proxies (CP), and site managers (SM), Figure 6. The GM, RM, CP, and service replicas are specific to a virtual service. The SM can be used by different virtual services. The RM is the cornerstone of the virtual service. It is the decision-maker for *global* replica selection, creation, and deletion, and tracks the location and status of all replicas. The GM is a *local* decision-maker for replica management across a set of local clients. The CP is a local decision maker for replica management for a particular client.

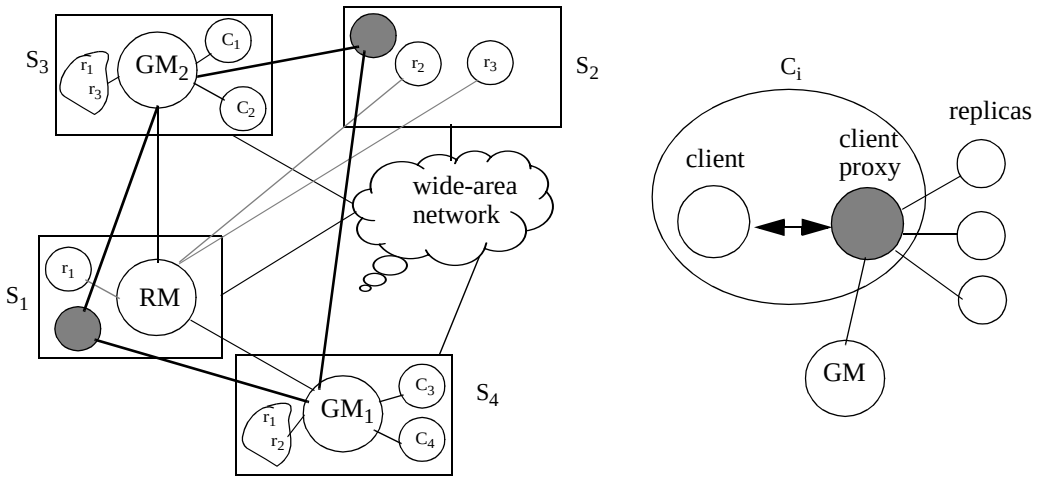

Figure 6: The Virtual Service Grid (VSG). Three replicas (r_1, r_2, and r_3) are running in two sites. An SM (filled circle) is running at each VSG site. Two clients sites S_3 and S_4 each contain a GM that has cached several replicas. Four clients are shown. On the right, the client proxy is expanded. The proxy runs on the same machine as the client and acts as a form of *virtual address* for the service.

The SM determines client-replica network performance between a remote site and a client site. Every site in the VSG runs a SM. A VSG can be configured to contain any number of GMs depending on the clients that wish to access virtual services. A client proxy is defined for each client with a group. Clients and their proxies belong to a particular GM, established at configuration time. Clients and GMs may run on machines anywhere in the network including sites outside the VSG perimeter. A single RM is assumed. However, in very large VSGs, replicating the RM may be necessary for scaling and protocols for maintaining consistency will be needed.

Each client service request is made to its proxy which in turn initiates the remote service request. The proxy also acts solely on behalf of the client, monitoring performance of prior requests, and providing fault transparency. On the first client request for a service, its proxy obtains a binding from the GM. The proxy provides a virtual address for the service and maps this virtual address onto one or more physical replica addresses. The proxy supports the ability to dynamically change a replica binding or re-selection. Re-selection is needed to adapt to changing network conditions (scenario 4) or to select a different replica in response to a change in input parameters (scenarios 1-2) or if a replica fails (scenario 5). As each client request passes through the proxy[1], it makes a decision whether to continue to use the current binding or to obtain a new one from the GM based on prior performance and current input values. The proxy can also provide fault transparency by monitoring the request. If the replica does not return a result in a timely manner to the proxy (based on expected performance), it can re-issue the request to another replica. Another option is to have the proxy automatically use several replicas in parallel for each client request (if n replicas are used, the system can withstand n-1 faults). For stateless virtual services, this kind of replicated execution is a viable solution provided there are sufficient network resources.

The GM maintains a cache of replicas allocated to it by the RM over time that are used by the local proxies. The cache simply stores the address of each replica and execution information provided by the replica periodically (to be described shortly). Clients access replicas through their GM when making service requests. One of the GMs is designated as the *primary* for a replica. Each GM periodically updates the RM with the state information of all replicas for which it is the primary. The GM and CP off load much of the traffic that would otherwise reach the RM, promoting scalability. A GM can be configured to adopt its own local replica policies based on the needs of its clients. For example, a GM could decide to enforce admission control if the load on the local replicas is predicted to eclipse a threshold, or could request to acquire a new replica from the RM. Collectively, the GMs and the RMs will prevent over-subscription of resources much like virtual memory prevents over-subscription of memory. The precise roles of each component in replica management are described in the next section.

Network performance from any client within the same GM to an outside site is assumed to be approximately the same. The GM interacts with the SM to perform periodic network probes to obtain a current picture of the network to determine latency and bandwidth based on fixed-size message exchange. This communication is done only between a GM and a SM that is running a replica already cached by the GM, or to select a site for creating a new replica. By structuring the network in terms of replica sites (controlled by SMs), and client sites (controlled by GMs), network

1. In our current prototype, the client proxy does not yet provide fault transparency or re-selection. In the remainder of this paper, we use the term client and client proxy interchangeably.

information can be collected in a scalable fashion. The alternative adopted by most Web server selection mechanisms is to form a network probe between **each** client and server replica. This scheme is not scalable to thousands of clients and hundreds of server sites.

In this paper, we focus on those run-time aspects of the Virtual Service Grid critical to handling our five scenarios. The VSG is presumed to contain all resources a-priori running the necessary system components. A model for negotiating for additional resources outside the perimeter of the VSG can be found in [31], but is not addressed in this paper. In addition, practical problems relating to production deployment such as naming and security are outside the scope of this paper though many viable solutions to these problems are well-known [6][11][27].

3.2 Scalable Information

Information is the cornerstone of the virtual service. Virtual services do not exist in a vacuum, rather they must interact and collect information from the wide-area environment. In particular, policies for replica selection, creation, and deletion all depend on network and service time information. The primary metric we seek to minimize is client-perceived response time ($T_{response}$). The estimated response time for a service request with input parameters $\vec{P}$ that would be delivered to the client in group G (a group is managed by the same GM) using replica R_i has three components: service time, waiting time, and network time: $T_{response} = T_{serv} + T_{wait} + T_{network}$. More formally (shown for a specific service):

$$T_{response}(G, R_i, \vec{P}) \approx \overline{T_{serv}}(G, R_i, \vec{P}) + T_{wait}(R_i) + T_{network}(G, R_i, \vec{P}) \quad \text{(Eq. 1)}$$

$$T_{wait}(R_i) = Q_i \cdot \overline{T_{serv}}(G, R_i, *) \quad \text{(Eq. 2)}$$

Q_i is the queue length at R_i

$$T_{network}(G, R_i, \vec{P}) = LAT(G, R_i) + BW(G, R_i) \cdot D_{size}(\vec{P}) \quad \text{(Eq. 3)}$$

LAT/BW is the latency and bandwidth respectively from G to replica R_i and D_{size} is the amount of data transmitted to/from the client

T_{serv} is the time to execute a service request at a replica. Service replicas will record and store the service times of each executed request over a time window in the past and track the current queue length for requests submitted to the replica, but not yet executed. For T_{wait}, we must estimate the service times of other requests on the queue which may depend upon their own input parameters (this is indicated by the * parameter). Replicas periodically provide $T_{service}$ and T_{wait} to the groups that are using them. This information will be used to predict the service time and ultimately, the response time, of future requests. If the service time depends on input parameters (e.g. the *solver* in Figure 1), then the input parameters must be stored along with the service time. Using this information, the service can construct a model that predicts the response time for a new request to a replica given the previous history of service execution times, the current queue length, and the amount of communication required between the client and the replica to transmit inputs and outputs.

When the request depends on input parameters that have not been seen earlier, estimation procedures will be required. For example, if a replica has previously solved a 10x10 system in t_1 time units, a 100x100 system in t_2 units, how long will it take to solve a 50x50 or 1000x1000 system? If t_1 was 5 and t_2 was 500 we might infer a quadratic dependence (factor of 10, produces execution factor of 100). Alternatively, the virtual service itself may provide cost estimation functions specific to the kind of service and its implementation. Now suppose that the response time is network-sensitive such as the *visualize* service. The response time is not specific to a replica, but depends on the position of the client relative to the replica. The time spent communicating to and from the replica as perceived by the client is the network time ($T_{network}$). Because the amount of data transferred (D_{size}) may depend on the input parameters, an accurate estimate of the communication time requires an accurate estimate of the amount of data transferred as well. The virtual service must also be able to collect and estimate this kind of information. For $T_{service}$ and $T_{network}$ actual runtime measurement will be required to adjust the predictions to amplify their accuracy. Some recent research has confirmed that augmenting static prediction models with runtime estimation can lead to more accurate estimation [10][33] and is the approach we have adopted. In many cases, exact accuracy may not be required as *relative accuracy* will be sufficient. To illustrate this concept, suppose the system is trying to decide between two replicas and uses a model to predict the wait, service, and network time for either replica. Exact prediction may not be necessary - if the best replica is desired, then a relative prediction that can be used to rank the replicas is sufficient.

3.3 Dynamic Replica Management

3.3.1 Dynamic Replica Selection

Dynamic replica selection is the process by which the system selects a replica for a client to serve one or more requests. Dynamic replica selection introduces the following questions: which replica is served to a client? when is a replica replaced or re-selected? Replica selection begins as a local process. A client proxy in group G makes a replica selection request to its local GM and the GM checks its replica cache first. The GM will first try to select a cached replica offering the best relative end-to-end performance (where i ranges over the cached replicas):

$$R_{best} = min_i \left(T_{response}(G, R_i, \vec{P}) \right) \quad \text{(Eq. 4)}$$

The GM is periodically provided with replica and network information to calculate Eqs. 1-4. Since the replicas periodically update the GM with this information, it is likely that such information will be up-to-date. However, the best replica may not be good enough. The replicas may be overloaded or network conditions may have rendered one or more unacceptable to the GM or to the proxy. Periodically, the GMs will compute the average response time for its clients (the client proxy reports the actual response time for each request). If the average response time is above a threshold α for this group (each group may have its own threshold), then the GM may contact the RM for a new replica. The decision made the by the GM is a local one. Acquiring a new replica also occurs when the GM's cache is empty or stale. Similarly, when a replica has been underutilized for a period of time within a group that falls below a utilization threshold β, it can be released back to the RM. We describe a general framework (called the P-Q algorithm) for deciding **when** to acquire and release replicas in the next section.

When a GM requests to acquire a new replica from the RM which one is selected? The RM has global knowledge of all replicas, both *shared* (cached by multiple GMs) and *dedicated* (cached by a single GM). For shared replicas, the RM also knows which groups are currently using the replica. The RM will pick a replica that is predicted to deliver performance below the client group's threshold. If the replica is shared, the addition of this new group must not compromise the performance of the clients in the other groups using the replica (their performance threshold must not be exceeded). It is possible that no such replica exists and a new replica will have to be created (this is discussed in the next section). The RM uses the cost equations (Eq. 1-4) to help make this decision.

3.3.2 Dynamic Replica Acquisition and Creation

When client demand increases rapidly (as illustrated in scenario 3), the virtual service abstraction must transparently scale to provide performance independent of client load. Dynamic replica creation is required to scale capacity to increased demand[2]. The individual GMs decide when performance is unacceptable by selecting a threshold parameter α. When the average response time grows above α, then the GM may request to acquire a new replica (which may or may not cause the RM to dynamically create a new replica). The threshold α must be picked carefully - if it is too low, the GM will be requesting new replicas from the RM frequently. On the other hand, if it is too high, then client performance may suffer. Since this is an expensive global operation that involves global communication (to the RM), and possibly dynamic replica creation, a more flexible mechanism has been developed. We have developed an algorithmic framework that is applicable to both dynamic replica acquisition/creation and replica release/deletion called the P-Q algorithm.

The P-Q algorithm is given a threshold value (e.g. α), a threshold variable (e.g. response time), and two measured values: (1) p: number of consecutive time intervals with increasing (acquire) or decreasing (release) slope, and (2) q: the number of consecutive time intervals that the threshold is exceeded regardless of slope of the threshold variable. The threshold variable is the average response time for all clients within a group (for acquire) and the average utilization for all replicas used by a group (for release). In the latter case, the threshold is applied to each replica. The constants P and Q are parameters of the algorithm such that ($P \leq Q$). The algorithm is run at each discrete time interval Δt. The P-Q algorithm reports "yes" (e.g. acquire), if ($p \geq P \parallel q \geq Q$). The use of P is to enable a rapid response to a change in performance, but to prevent transient response. If the test on P fails, then rather than answer "no" we apply the Q test which is more forgiving (e.g. the requirement that the slope be monotonically increasing or decreasing is relaxed) but the bar is higher ($Q > P$). We illustrate the operation of the P-Q algorithm in Figure 5 ($P=3$, $Q=5$). At $t=6$ and $t=9$, the algorithm triggers an acquire based on Q and P respectively. The minimum delay for a "yes" is $P\Delta t$. When a "yes" is returned, the algorithm resets the p and q counters. The parameters values for P and Q control the degree of pro-activity for replica management. Increasing the amount of pro-activity or aggressiveness (by lowering P or Q) also increases the likelihood of transient response. Similarly, more conservative replica management will set

2. The issue of how to replicate a service is specific to the service at-hand and is outside the scope of this paper. A virtual service will ultimately be able to use VSG services to transmit code and data to a remote site for replica instantiation.

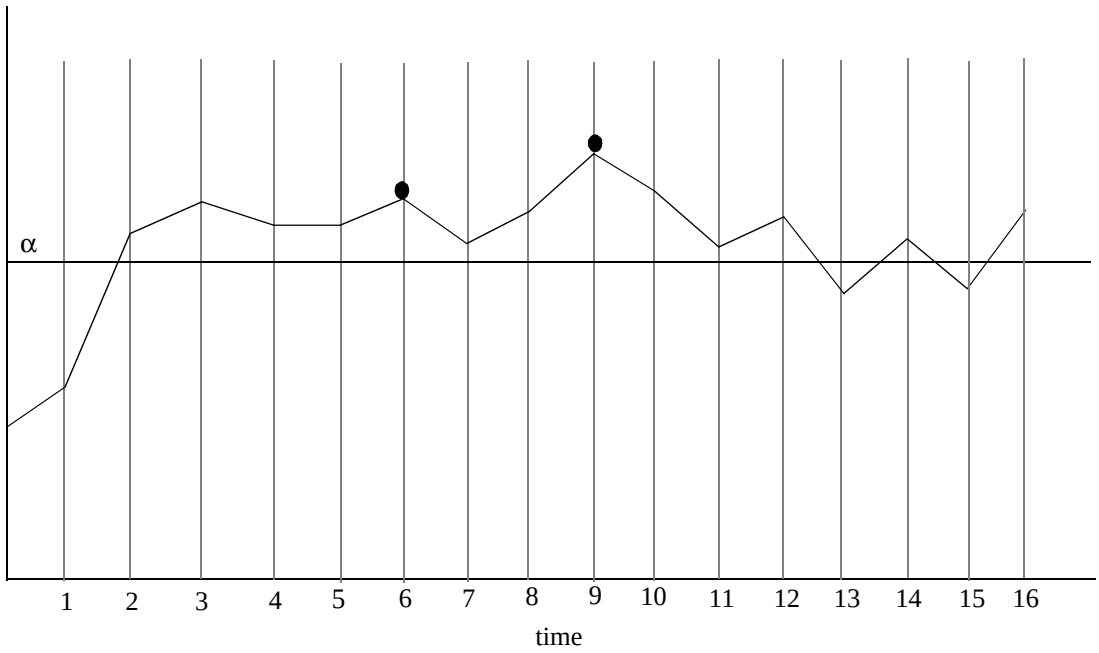

Figure 7: The P-Q algorithm (P=3, Q=5). Example of algorithm for acquire/create. At t=6, the algorithm reports "yes" since q≥Q, and at t=9, the algorithm reports "yes" since p≥P. The fluctuations after t=9 do not

the values higher. We present results indicating the sensitivity of several metrics (response time and utilization) to P and Q, and the threshold, for several workloads.

If the P-Q algorithm reports "yes" then an acquire request will be sent to the RM. When an acquire request for a new replica reaches the RM, it examines the most recent state of the replicas as provided by the GMs and picks the best available replica based on predicted response time (using Eq. 1-4). But in the case where the entire system is heavily loaded, it is possible that no available replica will offer acceptable performance to the current GM making the acquire request. At this point, the RM will create a new replica and allocate it to the group making the acquire request. Since creation is an expensive operation (both in time and resources), it is important that the P-Q algorithm filter out possible acquire requests that result from transient loads. For example, assuming wide-area network performance of 10 MB/s, a service that requires 1 GB of data and code to be transferred for dynamic replica creation, would take at least 1.6 minutes (not to mention site resources required to run the replica). On the other hand, if global client demand is increasing dramatically then the P-Q algorithm should not delay the replica creation substantially. While replica creation could introduce latency for the client group in the short run (e.g. 1.6 minutes as above), if the P-Q predicts that demand is growing, subsequent requests will be handled more rapidly amortizing the creation overhead. The GM maintains sufficient state to prevent race conditions on replica acquisition; it will not issue an acquire request while another is pending. However, the RM must be able to serve multiple acquire requests from different groups concurrently in order to scale to high client demand across the multiple groups.

If the RM decides to create a replica, the next question is where should the replica be located in the network? If the service is highly network-sensitive, then the position of the replica relative to the group becomes important. If the

service is not network-sensitive, then the location of the site matters less than the computational capability of the replica machine. Which group should be the "target"? The simple solution is pick the group that issued the acquire request. This is the current strategy. A different strategy would be to pick the group that has exceeded its local threshold α by the largest amount and would thus appear to get the greatest near-term benefit. Another possibility would be to select the group that is expected to issue the greatest number of future requests. For this option, the simple solution is to presume that the recent past is a good predictor of the future. The RM could pick the GM that has served the greatest number of client requests over a recent time window and use this as the target group. These strategies will be investigated in future work.

3.3.3 Dynamic Replica Release and Deletion

When demand declines, a group may choose to release a replica back to the RM. A replica that is released by all groups (idle) is a candidate for deletion. Replica release and deletion allow system resources to be reclaimed for allocation to other groups or services. Release is advantageous for two reasons: (1) once released, the group will not issue any requests to this replica which can promote the performance of other groups using the replica, and (2) once released, the replica need not report its status to this group reducing overhead. Replica deletion is advantageous if the resources are needed by other virtual services, or if the VSG owner is being charged for resources allocated whether they are in active use or not. Before releasing a replica, care has to be taken to ensure that some amount of capacity will still be available to the GM. The release decision is exactly analogous to the acquire decision, but instead of response time, the threshold variable of interest is utilization of the replica for clients within this group. We measure the utilization as the number of requests per a fixed amount of time serviced by a replica. We compare this utilization against a threshold β (here, falling below threshold is the trigger in contrast to acquire). Periodically, the GM checks on the possibility of releasing replicas in a manner similar to checking for acquisition. The P-Q algorithm is applied against each replica with the determination of p based on a monotonically decreasing slope. Since release is a global operation, we use the P-Q algorithm to prevent release of replicas to the RM based on a transient fall in demand. More importantly, if release is done prematurely, it could result in either a subsequent acquire by the same group (an expensive operation) or could lead to premature deletion of a replica (also an expensive operation, especially if the replica will need to be re-created soon after). The P-Q algorithm is given a threshold value (e.g. β) and uses two measured values: (1) p: number of consecutive time intervals with decreasing slope and (2) q: the number of consecutive time intervals that the utilization is below threshold (regardless of slope). As before, the constants P and Q are parameters of the algorithm such that ($P \leq Q$), and the algorithm is run at each discrete time interval Δt. The P-Q algorithm reports "yes" (e.g. release), if ($p \geq P \parallel q \geq Q$). However, in this case, if the algorithm reports "yes", the replica is only a candidate for release. Only one replica at most is released in a given cycle of the algorithm. Of the candidate replicas, we release the replica that is least frequently used over the most recent time interval Δt. Note that each GM can set a configurable parameter so that it controls the minimum number of replicas that it should maintain. More details of the P-Q algorithm can be found in [20].

Replica deletion is needed when the virtual service has allocated more resources than is currently needed. When a replica is idle (all groups have released it) for a period of time that exceeds a threshold, the RM can re-claim its

resources by deleting the replica. The RM is configured to maintain a minimum pool of idle replicas. When the number of idle replicas exceeds this limit, the replica that has been idle the longest is selected for deletion. Deletion runs the risk that new replicas may be needed in the future if demand increases. However, this risk must be weighed against the increased cost of holding resources for under-utilized replicas. This cost may include large amounts of storage, memory, and possible CPU resources (even in an idle state) for service replicas. Such resources could be made available to other services in large VSGs. In addition, the resources available to a VSG may be rented from resource providers, which may be charging for resources whether in active use or not.

4.0 Performance

4.1 VSG Prototype and Testbed

We have built a VSG prototype using the Legion system [14]. As an object-based distributed computing infrastructure, Legion provides fundamental services necessary for the VSG such as object creation/deletion, object naming and remote method invocation. Legion also provides a distributed network testbed in which our prototype can be easily deployed across multiple sites. The prototype consists of four core components: Replication Manager (RM), Group Manager (GM), Site Manager (SM), and Client Proxy (CP). When a client wants to access a service, it first sends the request to the its CP. After receiving a request from the client, the CP contacts GM to get a binding information of a local replica which is most likely to provide the best performance to the client. Once the CP acquires binding information for the best replica, it forwards the client request to the replica, waits for the return value and returns the result back to the client. In addition to forwarding client requests, the CP also reports the response time of the completed client requests to GM. With the response time reports from CPs within the same client group, the GM can decide to acquire or release replicas as needed. Note that CPs are created when clients first enter VSG and last during the lifetime of the clients.

For the VSG prototype, we have implemented two different high-performance services: **2D Matrix Multiplication Service** and **Jacobi Iteration Solver Service (JIS).** For the matrix service, we implemented two variations that allow us to isolate the impact of communication vs. computation: **MMS**, which assumes all matrices are stored at the replica server and thus requires no explicit communication (useful when the matrix product is to be used as part of another remote computation, e.g. multiplying a chain of matrices), and **MMSR,** which assumes matrices will be communicated between client and replica. The Jacobi iterative method is one of the simplest iterative techniques used to solve systems of equations of the form Ax = b that generate a sequence of approximations to the solution vector x. For JIS, clients send n x n matrix, n-vector, and the number of iterations as input parameters[3].

One of the important roles of GM is to find the best replica among its cached local replicas for the requesting clients. GM selects a best replica based on estimated response time. Response time is dependent on not only the capabilities of the resources but also the characteristics of the service. For example, since MMS does not involve a large

3. We pass the number of iterations to experiment with varying computational time for the service

amount of data communication, the processing time dominates the response time, while for MMSR and JIS, both processing time and data communication time should be taken into account to estimate the response time.

When services require a large amount of data communication such as in MMSR and JIS, the response time that clients would experience is affected by the communication time significantly. Therefore, GM needs to maintain the current network status between the client group and its local replicas. To collect network information, GM periodically probes SMs with sample messages. Latency is computed by sending 8 bytes messages and bandwidth by sending 64 Kbyte messages, and measuring round-trip time from the SM to the GM. While this mechanism can track the status of the network with modest consumption of system resources, it cannot predict the communication time accurately for much larger messages in terms of absolute values, particularly on the commodity Internet. On the other hand, if the GM uses sample messages of the actual data size for bandwidth prediction, it can perform relatively accurate prediction but this might consume a significant amount of system resources. To address this problem, we combine these two strategies together. That is, the GM runs two different network probe processes with different frequencies: 1) a network probe with 64 Kbyte message, and 2) a network probe with a sample message of the same size of actual data. Formally, the communication time is predicted as follows.

---

t : the last time when probe 1) is done

s : the last time when probe 2) is done

B(x) : comm. time measured with a 64 Kbyte message at time x
A(x) : comm. time mesaured with actual data at time x

$$Predicted\ Communication\ Time\ = \frac{A(s)}{B(s)} \times B(t)$$

By making the frequency of 2) much less than that of 1), we can achieve both relatively accurate communication time prediction and reasonable consumption of system resources. This strategy is feasible for fixed size services that use a single (or small number of different) message sizes. For services that do not, we use the closest message size that we have recorded and interpolate the predicted cost. When the RM receives a replica acquisition request from GM, it first examines the pool of available replicas not currently used by the group making the request. The RM determines whether an existing replica can provide predicted performance below the group's threshold while not compromising the performing of other groups sharing the replica. To do this, we first assume that recent history will be a good indicator of future requests. With this assumption, we use the average queue length of the candidate replica to represent the future workload of the groups sharing the replica. We add to this the estimated workload from the requesting group. It is a difficult task to accurately predict the future workload from the new group. In our prototype,

we assume the new group will generate a new request in each time window. Therefore, we set R_{queue} to be 1 and Q_{queue} to be the average queue length of the replica over the recent past.

α*: the threshold of the requesting group*

S_{queue}*: estimated queue length by the sharing group*

R_{queue}*: estimated queue length by the requesting group*

$$(S_{queue} + R_{queue}) \times T_{serv} + T_{network} \leq \alpha$$

If there is no replica that satisfies the above condition, then the RM creates a new replica. When the RM decides on the target host for the new replica, it considers the processing capability of the host, and the performance of the communication path if the service involves a large amount of data communication. For measuring the processing capability, RM uses benchmarked data for hosts if they were not used before to host services. Otherwise, the average service time of other replicas that are running on the host is used. In the current prototype, we allow multiple replicas on a single host. In general, the reported service time reflects any competition for the CPU whether from other replicas or other processes[4].

The Legion network contains a variety of distributed hosts with different relative processing power (Table 1). In addition, the UTSA machines are connected into VSG via commodity Internet while other machines are connected by vBNS. Our testbed includes a diverse range of resources in terms of processing speed and network connectivity. We now show that the VSG makes good use of the characteristics of system resources to improve performance and system resource utilization.

4. A VSG can also be configured to limit a single replica per host.

Table 1: VSG Testbed. Benchmark data is for a matrix multiply of 400x400 sized matrices.

Host Name	O/S	Benchmarked Processing Time(ms)
University of California, Berkeley (UCB)		
u6.cs.berkeley.edu	SunOS	79193.893
u7.cs.berkeley.edu	SunOS	78950.285
u8.cs.berkeley.edu	SunOS	78241.115
u9.cs.berkeley.edu	SunOS	79670.393
University of Minnesota (UMN)		
juneau.cs.umn.edu	Linux	11776.624
sitka.cs.umn.edu	Linux	11687.796
University of Texas, San Antonio (UTSA)		
fearless.cs.utsa.edu	SunOS	28974.354
pandora.cs.utsa.edu	SunOS	28877.360
wolf.cs.utsa.edu	SunOS	27998.673
University of Virginia (UVA)		
centurion172.cs.virginia.edu	Linux	18049.087
centurion173.cs.virginia.edu	Linux	18054.539
centurion174.cs.virginia.edu	Linux	18032.441
centurion175.cs.virginia.edu	Linux	18036.933
centurion176.cs.virginia.edu	Linux	18040.933

4.2 Results

4.2.1 VSG Overhead

The first issue we examined was the overhead inherent to the VSG. The primary overheads include service object creation, deletion, and acquisition of binding information. If a new replica is required and the time to create a new replica is significantly high, then the cost may outweigh the benefit. So it is important to characterize this overhead. Furthermore, if the CP or GM takes too much time to acquire the binding information for a replica, then simpler algorithms such as round-robin or random selection may be better. So, it is important to characterize the overheads inherent in dynamic replica management.

Binding information acquisition

The cost of acquiring binding information is shown in Figure 8. Since the GM is located in the client group, the communication time between GM and client is very small. Since service replicas and the GM periodically transmit information in the background, there is no need for any additional information collection when binding requests arrive to the GM.

Service Object Creation and Deletion

When the RM decides to create new replicas, it first determines the best location among the available hosts. If the services does not involve significant data communication the RM can immediately choose a best host. However,

if the service requires a large amount of data communication, the RM determines the network performance between the requesting client group and the new host by launching communication probes.

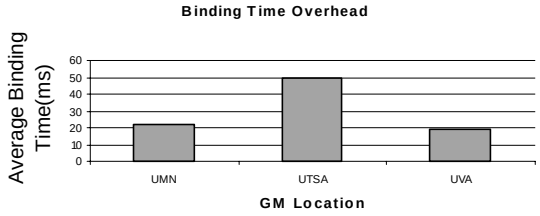

Figure 8: Binding overhead

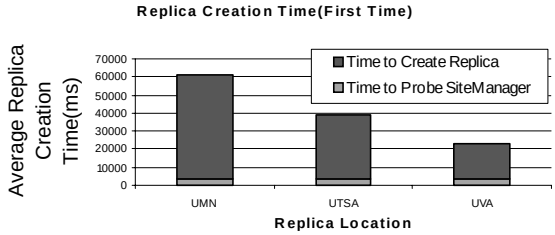

Figure 9: Service Object creation time -first time

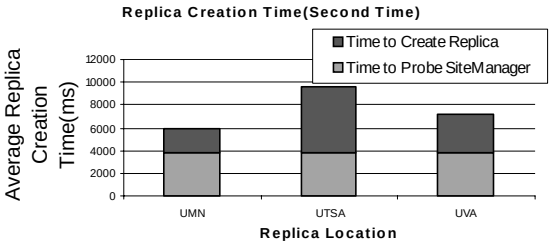

Figure 10: Service Object creation time - second time

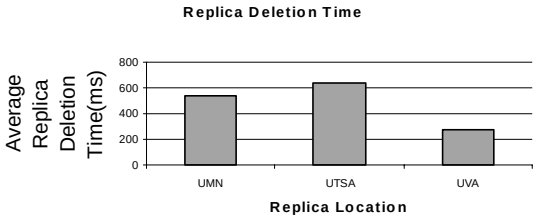

Figure 11: Service Object deletion time

Once the RM decides where to host a new replica, it calls a Legion library function to create the new replica. In Legion, the *ClassObject* is responsible for creating new objects. When the *ClassObject* receives a new object creation request, it first sends the binaries to a *HostObject* on the target location if the target location was not used before to host objects of the same class. The *HostObject* then creates a process for the service on the machine. Therefore, the time to create new replicas consists of two parts: time for selecting a new location and Legion overhead (time to transfer binaries and create processes). The object creation overhead is dominated by Legion overhead (Figure 9,10). This is because large binaries must be transferred to the target location. We show results for JIS in which the binaries are 32 Mbytes for Linux and 20 Mbytes for Solaris. However, once the binaries are cached at a site, Legion overhead is reduced significantly. At this point, the time for collecting network information accounts for the bulk of the creation overhead (Figure 10). When replicas are idle for a long time, RM deletes the replicas from the system (Figure 11). To do so, the RM calls another Legion library function. All results shown in subsequent sections include binding, creation, and deletion overhead in the presented data.

4.2.2 Response Time Prediction

Accurate prediction of response time is the cornerstone of effective replica management. The experimental results show that response time prediction can be done with high accuracy. For each high performance service, we first describe the response time model used for prediction.

MMS/MMSR

A dominant factor in response time for MMS is processing time. For this reason, network information for latency only needs to be collected (to initiate the request and to receive an indication that the service is finished). For

MMSR, network information for bandwidth is needed (400x400 doubles is 1.2 MB of result data). Since matrix multiplication takes $O(n^3)$ we computed the unit time for a service request as follows:

$$Unit\ Time\ =\ \frac{T_{serv}}{Matrix\ Size^3}$$

When a replica needs to report status information to the GMs, it returns the average of the unit time recorded along with the snapshot of the wait queue. The GM predicts the response time as follows ($T_{network}$ is dominated by latency for MMS and bandwidth for MMSR):

$$Tresponse\ =\ T_{wait} + Matrix\ Size^3 \times Unit\ Time + T_{network}$$

JIS

As in MMSR, the matrix and vector size are fixed for these experiments. For the experiment, each client sends a 400x400 matrix and vector of size 400. However, each client can choose the number of iterations randomly and hence, the service time can vary. To solve Ax = b, JIS runs the following computation for each iteration:

$$x[i]\ =\ \frac{1}{A[i,i]}\left(b[i] - \sum_{j \neq i} A[i,j]x[j]\right)$$

Since each iterations takes $O(n^2)$, we compute the unit service time as shown below. Since JIS includes a large amount of data communication for sending parameters and returning results, communication time for bandwidth is also used.

$$Unit\ Time\ =\ \frac{T_{serv}}{k \times Matrix\ Size^2}$$

$$Tresponse =\ T_{wait} + k \times Matrix\ Size^2 \times Unit\ Time\ + T_{network}$$

k : the number of iterations

Figure 12 shows response time prediction for MMS, MMSR, and JIS for both fixed size and variable size matrices. For MMS, each client chooses matrix size randomly among 200x200, 300x300, 400x400 and 500x500. In our experiments, the GM predicted the response time in the vast majority of cases within a 10% accuracy. For those cases where accuracy was above 10%, the GM was still able to make effective ranking decisions for replica selection.

4.2.3 Selection Policy Comparison

The next question we investigated is whether prediction-based replica selection could outperform simpler policies such as round-robin and random selection. For this experiment, we created three replicas on UMN, UTSA and UVA sites, and clients are deployed in the UMN site. We compared the average response time of each strategy as the

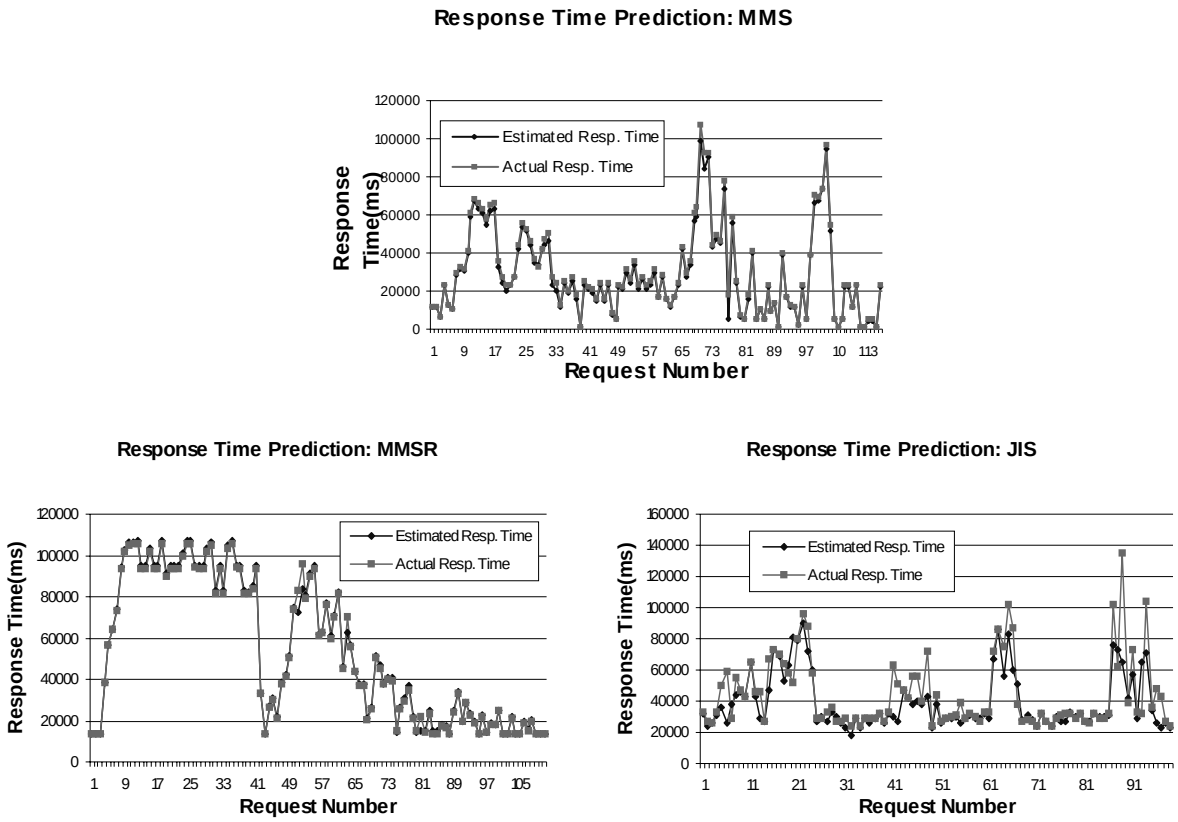

Figure 12: Response Time Prediction for different service types

number of clients increases. However, clients that use JIS can select the number of iterations randomly. As Figure 13 shows, prediction-based replica selection algorithm outperforms both round-robin and random for all service types. In prediction-based selection, the GM was able to consider the characteristics of the resources and select a replica that was predicted to provide the best performance. In round-robin and random selection, a replica is selected without regard for the capabilities of the replicas and the connectivity between replicas and client groups. In our scheme, a faster replica is more likely to receive requests, while in the other schemes, requests tend to be more even spread throughout the system. As the number of clients increase, the performance gap begins to widen. This highlights the importance of using information when services are heterogeneous, have significant computational demands, and client load is high.

4.2.4 Performance Comparison

The next issue we investigated was the performance of different replica management strategies, including the dynamic scheme described earlier. We deployed clients on three different locations: UMN, UTSA and UVA. For the evaluation, we generated two workloads: “stair-shape” and “high-demand” to examine performance under a variety of workload settings (Figure 14). Each point along the x-axis represents the number of total client requests generated between *t* and *t-1* (in units of minutes). The high-demand workload is relatively static because the number of total cli-

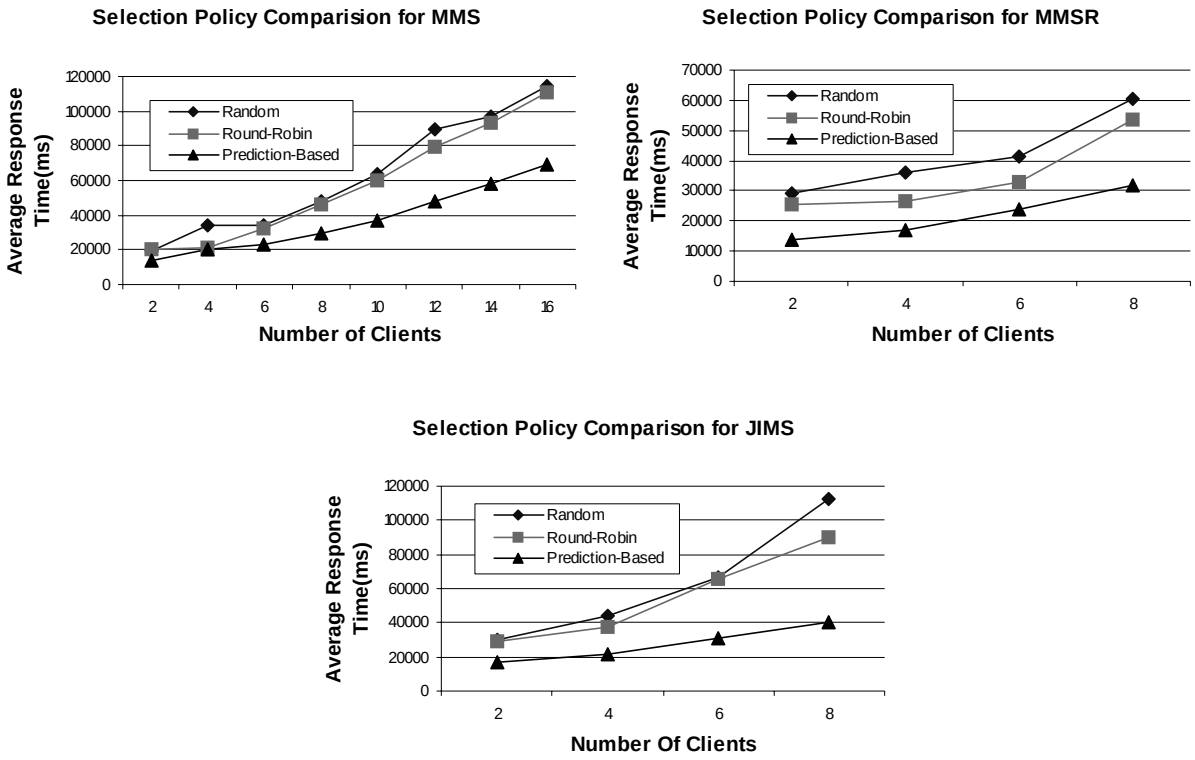

Figure 13: Selection policy comparison for each service type

ent requests does not change significantly during the lifetime of the experiment. However, in the stair-shape workload, the number of clients requests varies over time to model both demand growth and decline. Note that for each experiment using the same workload model, the actual number of client requests generated at a certain time point may not be the same, but the overall shape of the workload is the same.

We compared the performance of four different replica management schemes: Fully-Dynamic, Dynamic-Hybrid, Static-2 and Static-8. In the fully-dynamic strategy, no replicas are pre-created. When the GM is created, it acquires an initial replica from the RM and always maintains at least one replica in its local cache. More replicas may be created based on demand. Static-*x* represents a static pre-allocation of *x* replicas created across multiple sites. No

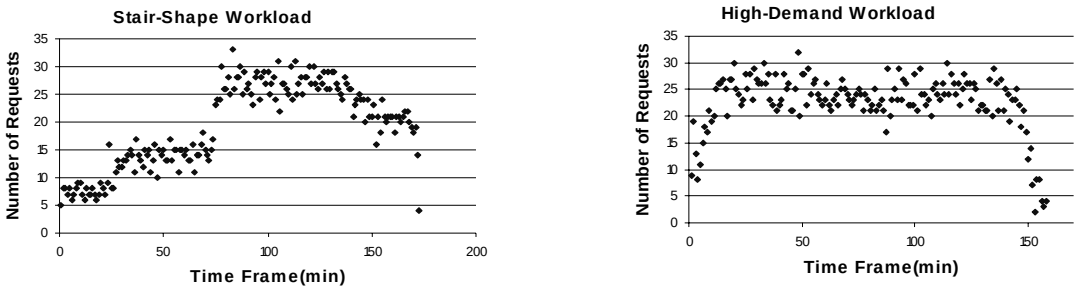

Figure 14: Two different types of workloads for the experiments

additional replicas will be created. In the dynamic-hybrid strategy, we statically pre-create 2 replicas and then allow the system to dynamically replicate if necessary (the GM always maintains at least two replicas even if they are underutilized). In the prototype, the replica acquisition/release algorithms are run every 2 minutes and the service replica transmits its status every 10 seconds if it is dedicated to a single GM, otherwise it transmits its status every 1 second to each GM that is using it. To predict communication time accurately, the GM for MMSR and JIS runs probe 1) every 2 minutes and probe 2) every 6 minutes. The GM for MMS runs a network probe for latency every 2 minutes. We now present results for each workload in turn. For each workload we indicate the parameters used to configure the GM for each service (Table 2,3). For different workloads, different parameters were selected to highlight the performance sensitivity to the selected parameters.

Table 2: Configurable parameters of GM for stair-shaped workload

	MMS			**MMSR**			**JIS**		
Parameter	**UMN**	**UTSA**	**UVA**	**UMN**	**UTSA**	**UVA**	**UMN**	**UTSA**	**UVA**
Threshold (sec)	65	55	85	65	55	85	65	60	80
P (acquisition)	3	2	3	3	3	3	3	3	4
Q (acquisition)	7	5	7	6	5	7	5	4	7
Utilization	5	3	5	3	3	3	3	3	3
P (release)	4	4	4	4	4	4	4	5	4
Q (release)	9	9	9	9	9	9	9	8	9

Table 3: Configurable parameters of GM for high-demand workload

	MMS			**MMSR**			**JIS**		
Parameter	**UMN**	**UTSA**	**UVA**	**UMN**	**UTSA**	**UVA**	**UMN**	**UTSA**	**UVA**
Threshold (sec)	65	55	85	65	55	85	65	60	80
P (acquisition)	3	3	3	3	3	3	3	3	4
Q (acquisition)	6	5	7	6	5	7	5	4	7
Utilization	3	3	3	3	3	3	3	3	3
P (release)	4	4	4	4	4	4	4	5	4
Q (release)	9	9	9	9	9	9	9	8	9

We now present results for the stair-shape and high-demand workloads. In each case, we present the performance of each replica management strategy in terms of delivered average response time to clients, and the average replica utilization under each scheme.

Stair-Shape Workload

The response time measured at the client sites under each replica management scheme for each service type is shown (Figure 15). Static replication with 8 replicas achieves the lowest response time as it uses a large number of replicas. However, it suffers from low utilization (Figure 16). As shown in Figure 15, both dynamic replication and

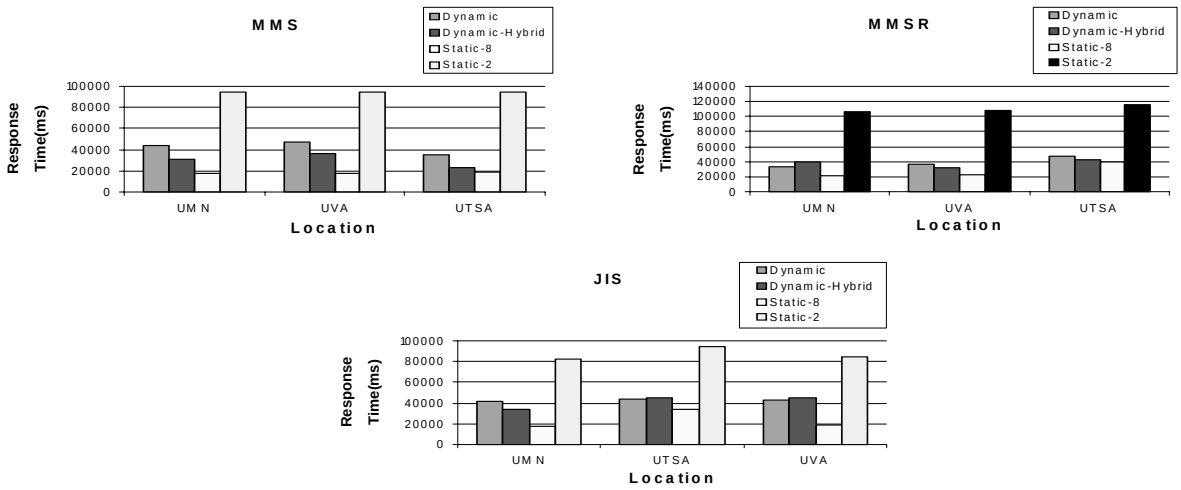

Figure 15: Comparative performance for different replica acquisition policies

static-8 meet the performance threshold requirement (see Table 2 for threshold). Observe that dynamic-hybrid provides better performance as compared with fully dynamic during the first demand peak period (Figure 17). We show results for MMSR only in the interest of brevity (the same pattern was seen for the other services). The static pre-allocation allows the system to more quickly serve the early burst of requests. However, this scheme may suffer from low utilization if the number of client requests is very low. In addition, observe how the dynamic schemes generally offer smooth response time irrespective of client load, but with static schemes such as static-2 the response time tends to mirror the workload shape.

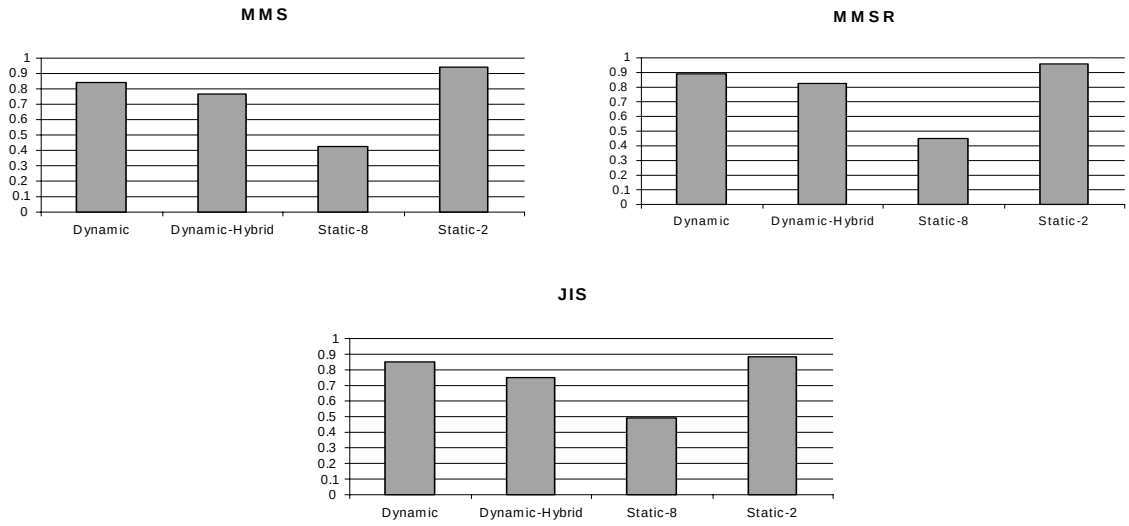

Figure 16: Comparative utilization for different replica acquisition policies

Next we show the percentage of client requests completed within the threshold that each GM defined (Figure 18). As the graph shows, the gap between dynamic and static-8 becomes narrower when compared with the average response time (Figure 15**)**. Although the average response time for dynamic replication is higher than for static-8, most of individual requests are completed within the threshold. In this sense, dynamic replication can achieve acceptable levels of performance to static-8 but at lower cost.

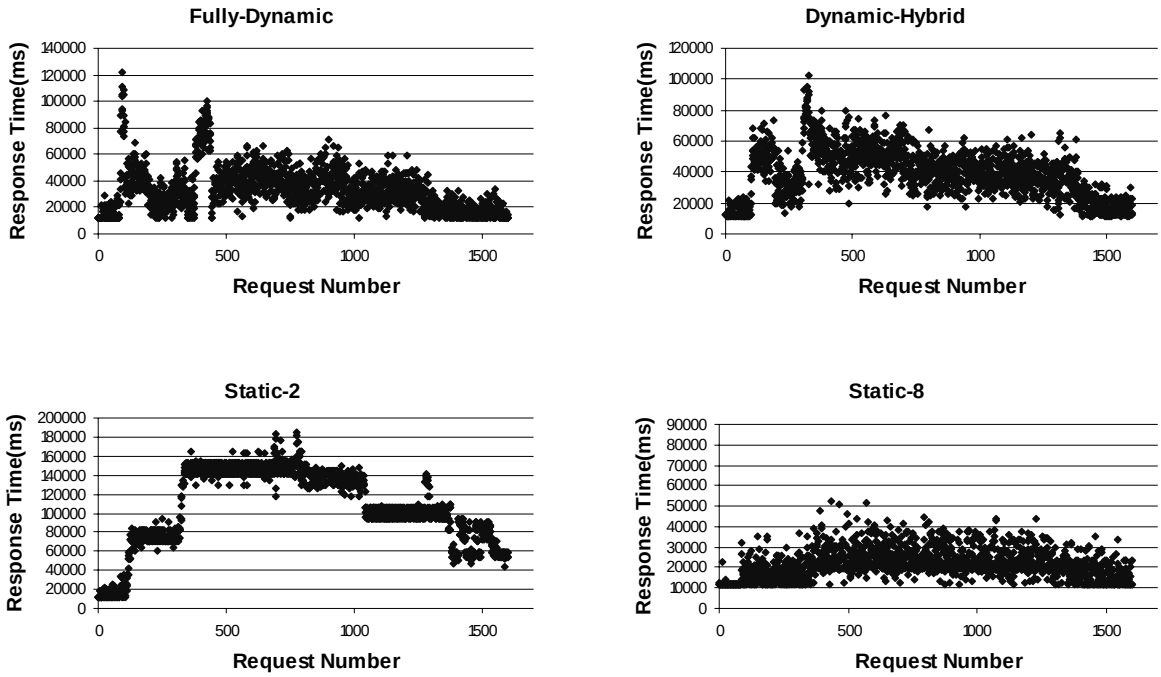

Figure 17: MMSR: Measured response time under 4 replica acquisition strategies

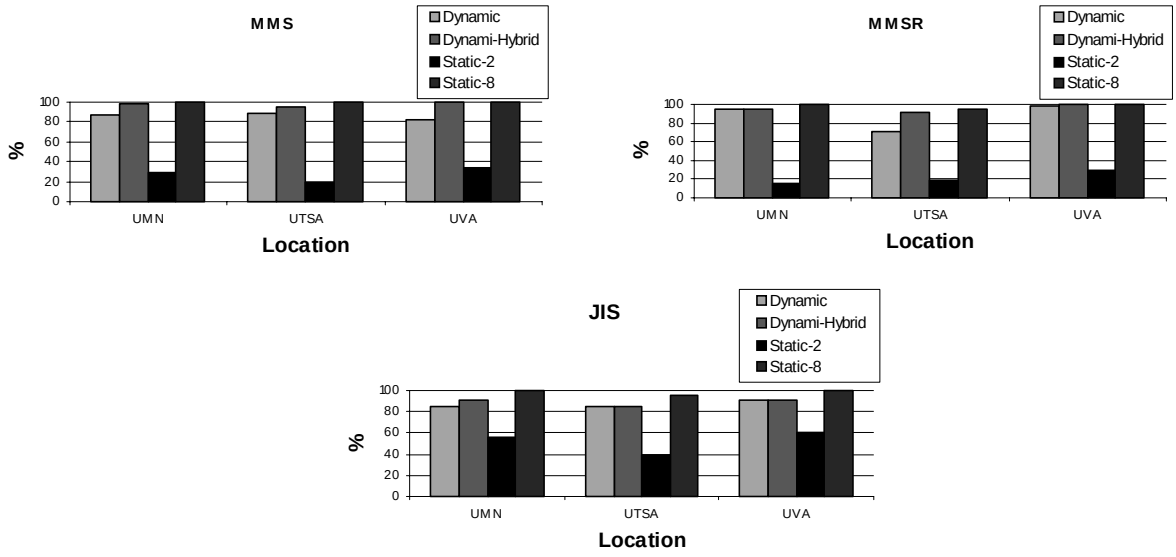

Figure 18: Percentage of client requests below performance threshold under different replica acquisition policies

High-demand Workload

The high demand workload keeps request volume high throughout the lifetime of the experiment. In such a high demand setting, static-8 offers the best response time overall (Figure 19). As in the stair-shape workload, dynamic replication can meet performance objectives at a lower cost (Figure 20). The replicas become overloaded faster and much earlier than in the stair-shaped workload, therefore all dynamic schemes suffer an initial performance degradation early on until sufficient replicas can be created to handle the load (again we show a representative graph for MMSR, Figure 21).

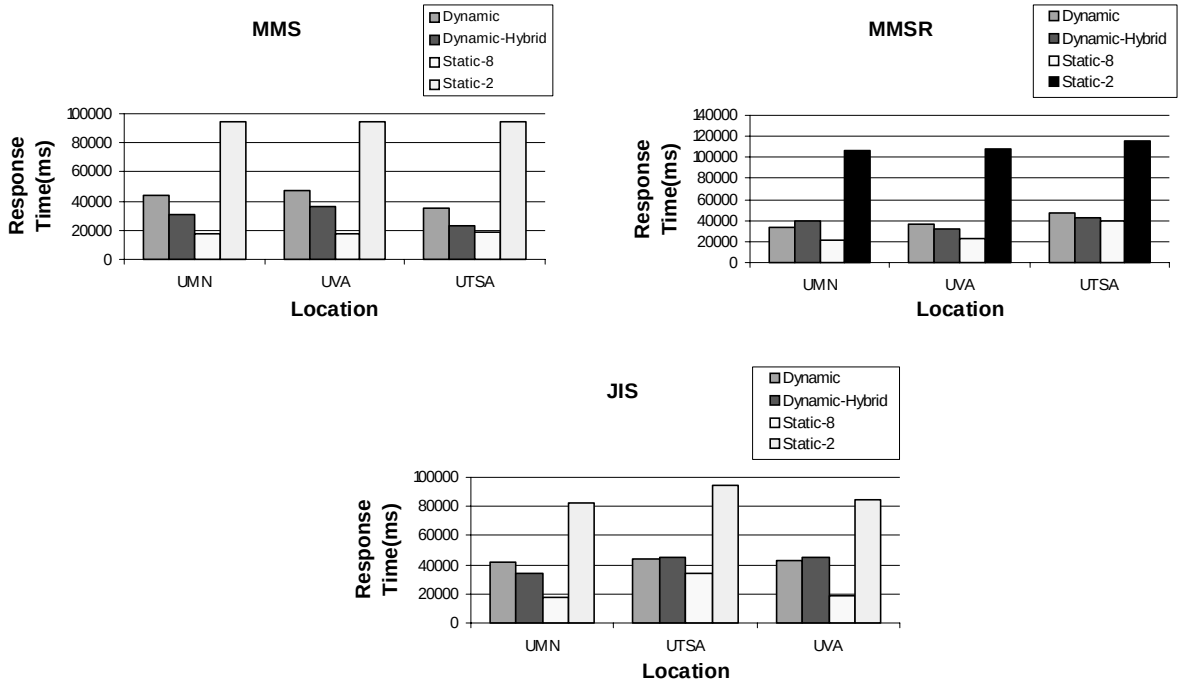

Figure 19: Comparative performance for different replica acquisition policies

Interestingly, the percentage of client requests below threshold in JIS is relatively low compared to other service types (Figure 22). This is the result of service time dependence on the number of iterations. If a large number of small requests (small number of iterations) are interspersed with large requests (large number of iterations), then the P-Q algorithm is delayed in generating new replicas. The reason is that the small requests are generally below threshold which prevents the early-start property of the algorithm (replicating based on P). While large requests are above threshold, enough smaller requests (below threshold), delay the replica creation. This highlights the importance of properly selecting P and Q as discussed in the next section. It also suggests to us that a single threshold may not be sufficient for highly variable services.

4.2.5 Sensitivity to Time Parameters

The final question we examined was the role of the parameters selected for the GMs in the VSG. Time parameters defined in the GM play a critical role in performance and utilization. In particular, the behavior of the GM

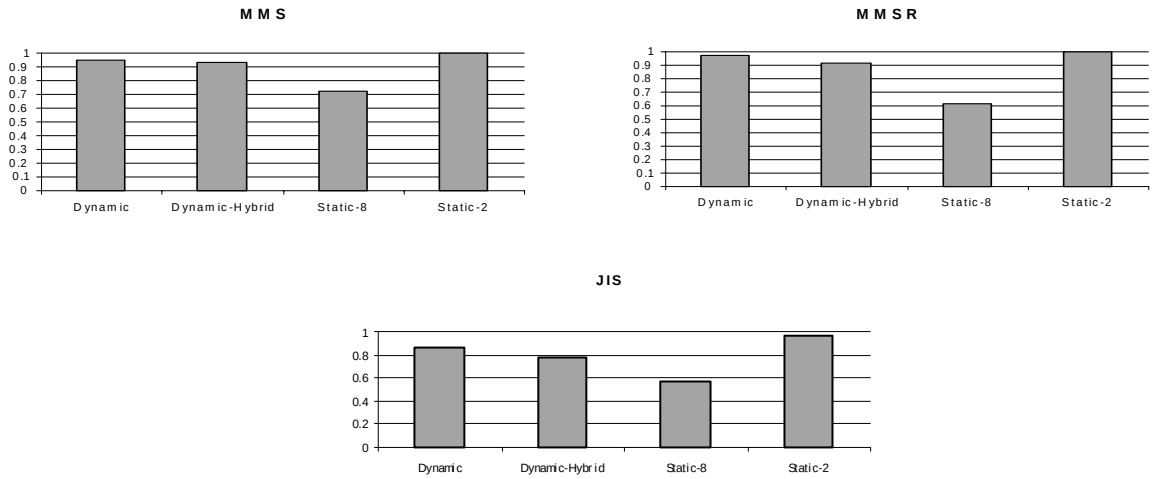

Figure 20: Comparative utilization for different replica acquisition policies

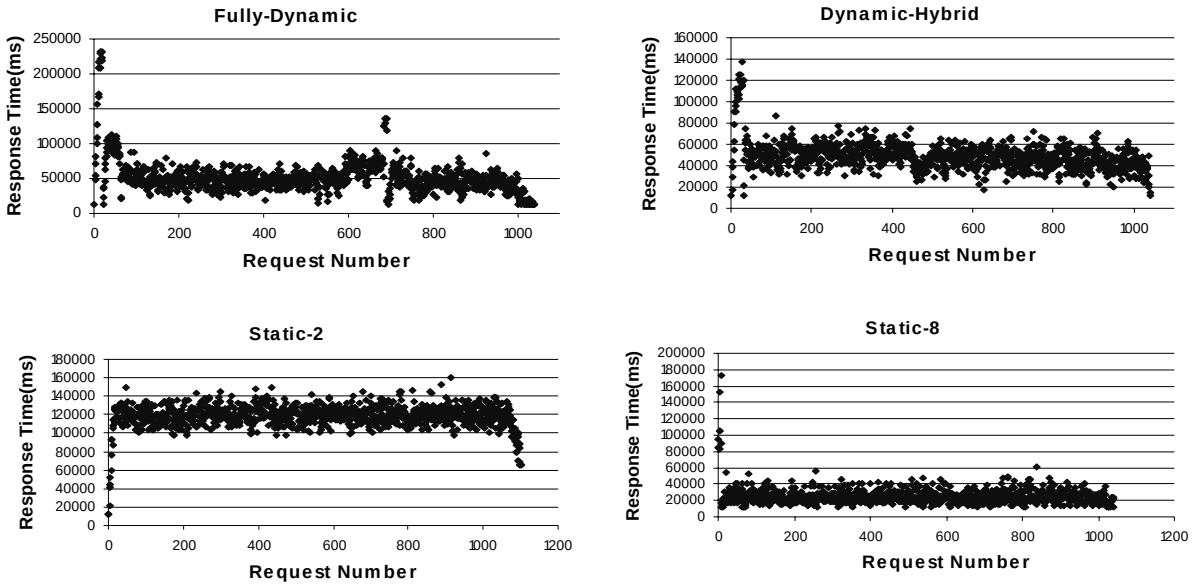

Figure 21: MMSR: Measured response time under 4 replica acquisition strategies

depends on the values of P and Q for replica acquisition and release. To examine the sensitivity of our algorithms to those parameters, we used two different (micro) workloads for replica acquisition and release, respectively (Figure 23). In both workloads, the number of client requests increase or decrease significantly at a point in time. Since the acquisition and release algorithms depend only on the average response time, we used MMS for the experiments. During highest peak of client demand, a total of 8 clients are active while during low demand, 4 clients are active. The maximum threshold is 65 sec and the minimum utilization threshold is 3.

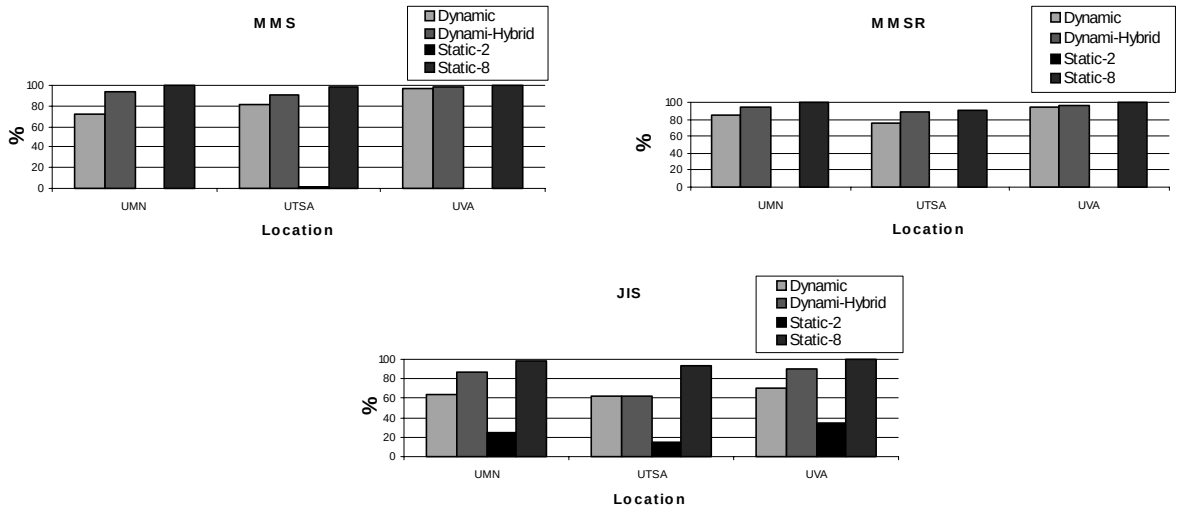

Figure 22: Percentage of client requests below performance threshold under different replica acquisition policies

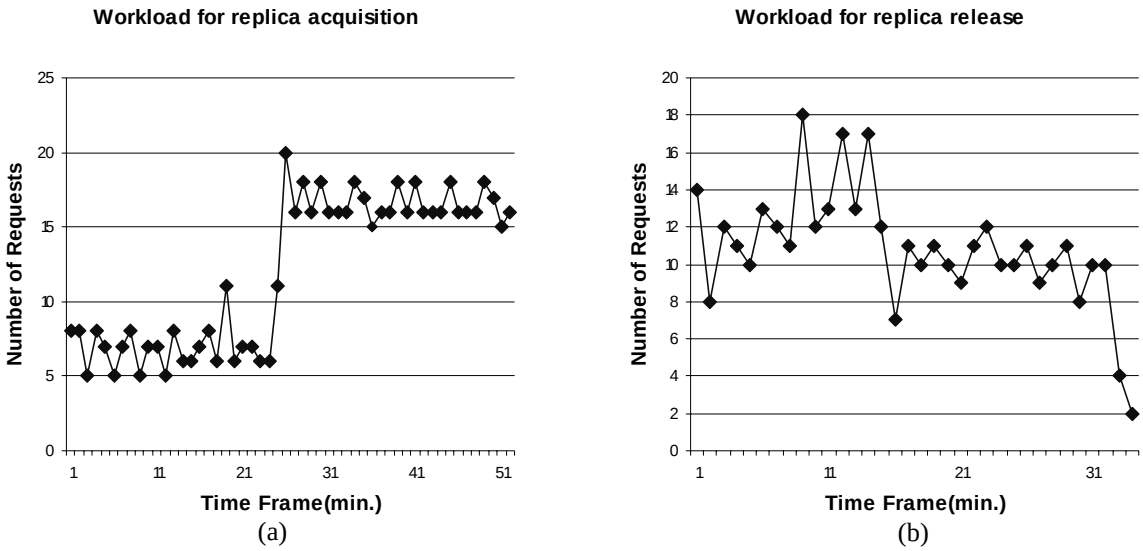

Figure 23: Workload for replica acquisition and release

Replica Acquisition

In this experiment, we set the thresholds for release high enough to prevent release to isolate the behavior of acquisition. Depending on the values of P and Q for acquire, the amount of time that the average response time is above the threshold varies (Figure 24). Smaller values of P and Q reduce the response time because they increase the probability of replica creation. Therefore, if the demand peak continues, it is desirable to make these values small. However, if the demand peak is transient, the GM may acquire unnecessary replicas. In the workload above (a), after the number of requests increases significantly, it remains for a long time. Smaller values of P and Q help to reduce the average response time (Figure 25). For this high demand workload, replica utilization approaches 100%.

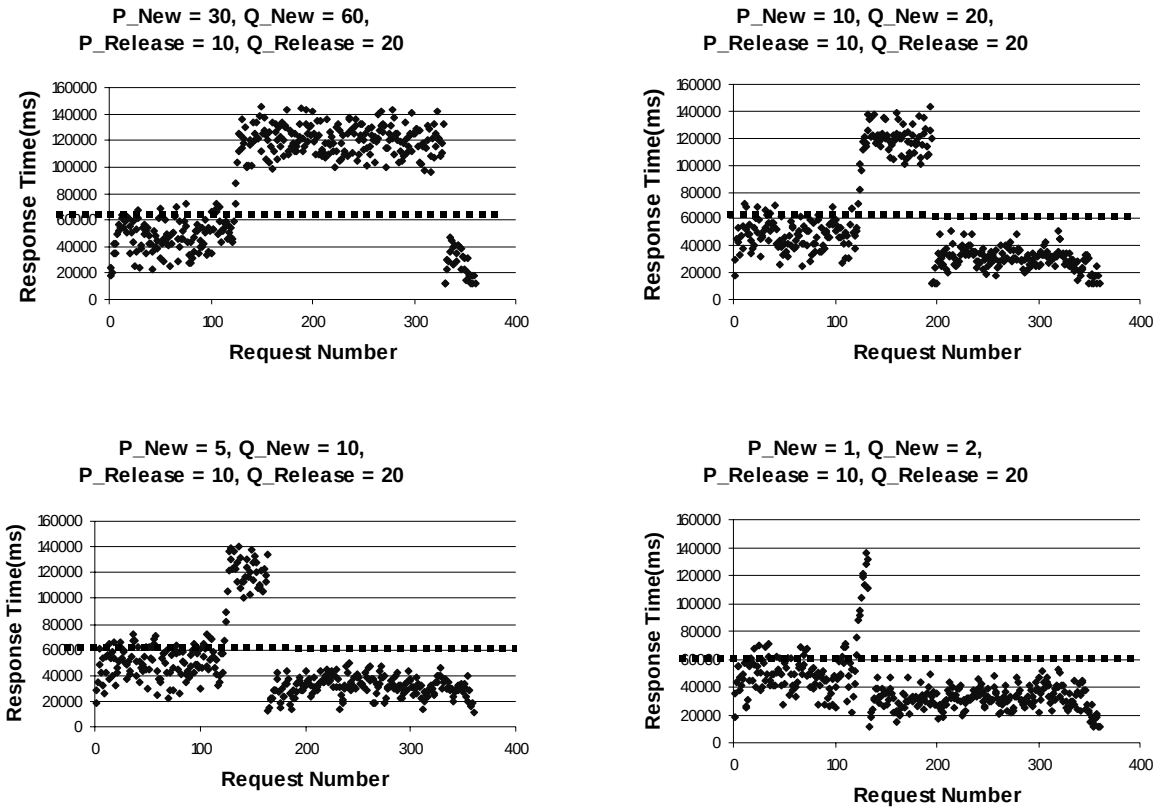

Figure 24: Sensitivity to P and Q values for replica acquisition

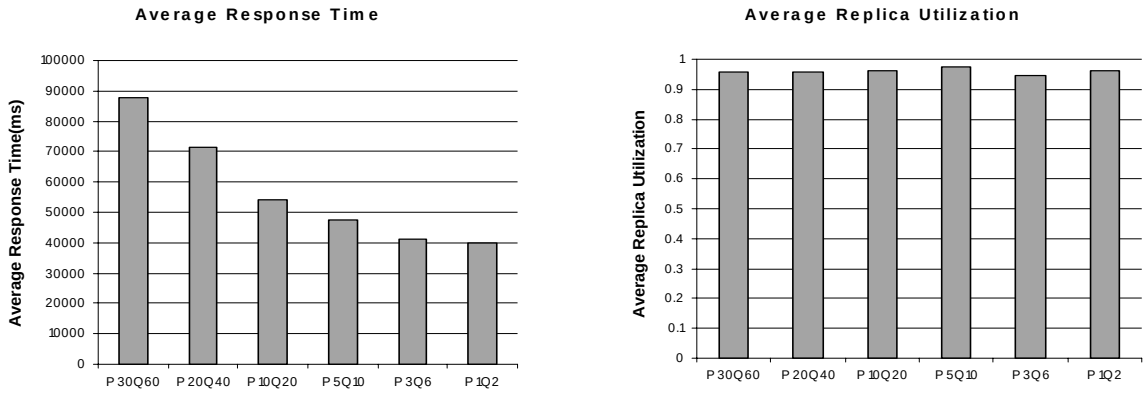

Figure 25: Comparative performance and utilization with different P and Q values for replica acquisition

Replica Release

In this experiment, we use a fixed value for P and Q for acquisition and vary P and Q for replica release. Since P and Q values for replica acquisition are the same, a new replica is acquired at similar time point in each run during phase 1 (the first portion of the graph up to the solid vertical line). However, depending on the P and Q values for replica release, different behavior will result (Figure 26). In the most aggressive case (P = 1, Q = 2), replica acquisition

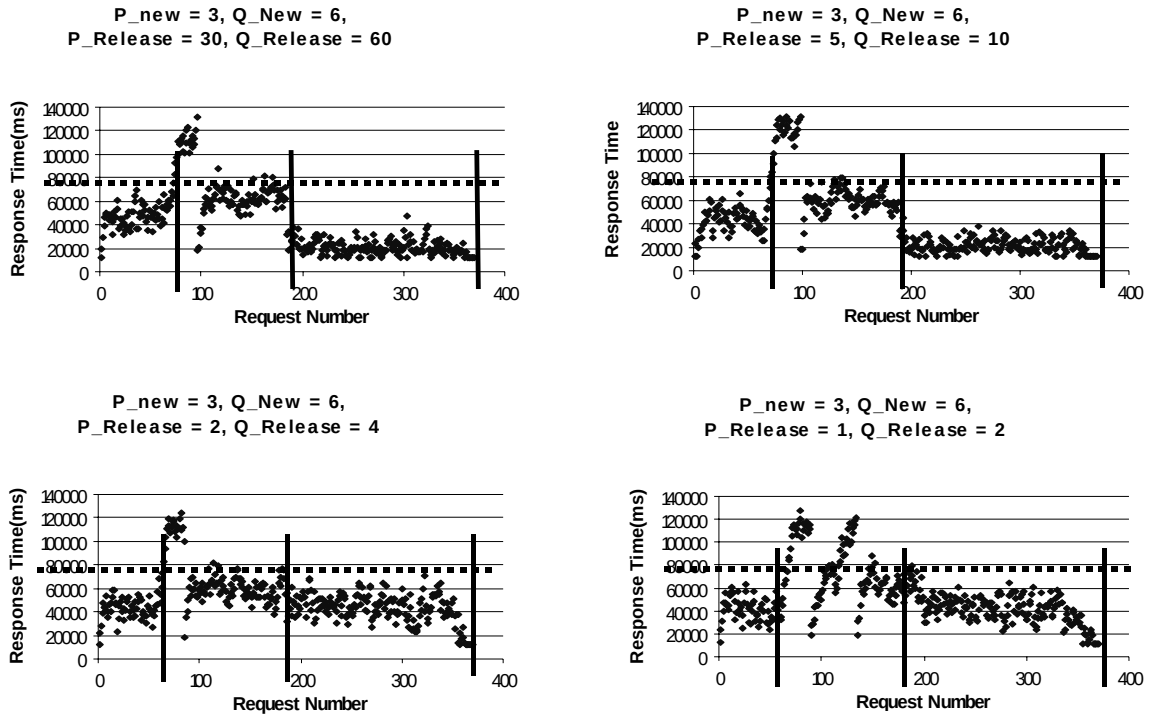

Figure 26: Sensitivity to P and Q values for replica release

and release is continued during demand peak period (phase 2). In the other three cases, two replicas serve client requests during the second phase. In the third phase, since the number of requests decrease, one replica is enough to service every client request. So with P=2, Q=4, the GM releases the unnecessary replica. However, with P=30, Q=60 or P=5, Q=10 the GM still holds the underutilized replica. Figure 27 shows the average response time and replica utilization. Larger P and Q values result in smaller average response time because underutilized replicas are not released (and can be utilized later if demand increases). But because of this, replica utilization goes down. The important point to stress is that P and Q values for both acquisition and release represent a trade-off. For systems unwilling to tolerate low utilization due to the high cost of resources, then aggressive release (low P and Q for release) may be desirable. On the other hand, if good performance and client satisfaction is the principle goal then low P and Q values for acquisition and high P and Q values for release are needed. The most appropriate choices, however, depend on the characteristics of the workload. Adjusting the values of P and Q to achieve some level of "QoS" (both in terms of response

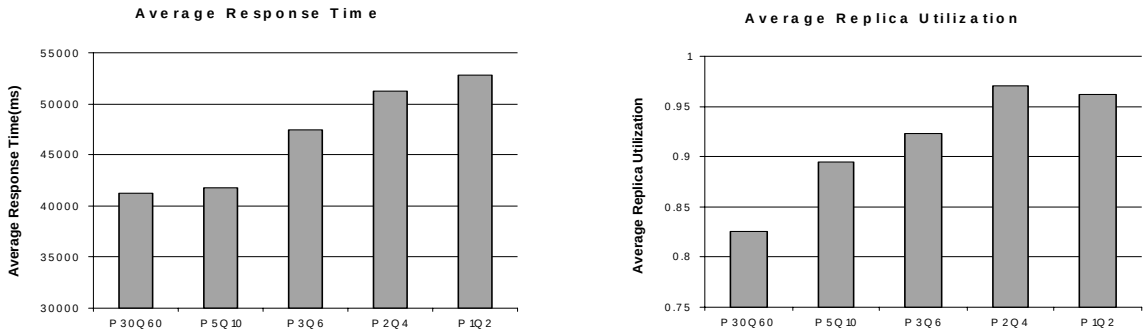

Figure 27: Comparative performance and utilization with different P and Q values for replica release

time and utilization) in response to changing workload characteristics is a promising and fascinating area of future research.

5.0 Related Work

A number of projects are exploring replica selection [1][4][5][10][13][24][25][27][27][33][34]. Scala-server projects [13][25] are studying mechanisms to support server replication within single-site clusters. In our work, which is complementary, multiple replicas may be created within a single site or across multiple sites. The focus of other research [4][5][10][33] is on accurate bandwidth characterization and measurement, which is also complementary to our work. Several other projects are working on replica selection in wide-area environments. WALRUS [3] is a Web-based server selection infrastructure that is designed to run without requiring modification to the network infrastructure (DNS, Web servers, and Web client software). Their server selection algorithms consider communication performance first (geographic locality), then load balancing. The server selection policy does not consider the possibility that services may be heterogeneous. However, this model has practical value due to its compatibility with existing Web infrastructure.

The WebOS and Smart Clients projects [27][34] defer server selection to the client. When a user wishes to access a service, a bootstrapping mechanism is used to receive service-specific applets designed to access the service. The client-side applet makes a service-specific choice of a physical host to contact based on an internal list of server sites. The focus of these projects is on replica selection mechanisms. WebOS does support dynamic replication creation, but assumes that their software infrastructure and Web tree are already loaded on the remote system.

Active Names [27] maps service names to a chain of mobile programs that can customize how a service is located and how its results are transformed and transported back to the client. In this model, they demonstrated that end-to-end information can be easily gathered while resolving the service name and the importance of using the information when selecting replicated servers. While our work is also focused on support for constructing customized replica selection policies, we do not implement our model via the naming system.

Fluid Replication is one of the very few systems that support automated replica creation [24]. Clients monitor their performance while interacting with the service and when performance becomes poor, a replica is created auto-

matically. To select a replica site, they employ a distance-based discovery mechanism. In contrast, our solution for replica creation considers the characteristics of the service in addition to communication.

Dv provides a visualization service for massive scientific data sets that are stored at remote sites [21]. Their work is based on the notion of an active frame, an application-level transfer unit that contains program and associated application data. The visualization process is divided into multiple stages which can be executed in pipelined manner. On each stage, an incoming active frame is processed by the active frame server and the intermediate data and the program associated with it are forwarded to the next active frame server, which could be on a different machine. However, this work is at an early stage and they have not yet tackled the dynamic scheduling problem of selecting the best server to use at each stage.

The Bio-Networking Architecture is inspired by the biological metaphors of population growth in nature and provides a highly distributed replication mechanism [28]. Their work is similar to ours in that there is no master entity that collects information and controls the actions of others. Replication, migration and destruction is based on the concept of energy units which appears to have useful properties for economic-based resource management. In contrast, the metric that drives our research is response time and utilization.

6.0 Conclusion

We have presented a new mechanism for the delivery of high-end network services to client applications, the virtual service grid. The VSG provides scalable performance irrespective of client demand using dynamic replica management techniques for replica selection, acquisition, and release. Efficient algorithms were presented for each facet of replica management. These algorithms were based upon response time prediction models that achieved an accuracy of 10% or better. Replica selection using our response time prediction model out-performed random and round-robin replica selection for three high-performance services. Novel algorithms for dynamic replica acquisition and release (P-Q algorithm) based upon the use of throttle thresholds was also presented. We then compared the performance of several replica management schemes: fully dynamic replica management (using P-Q), static, and a combined approach. The results indicate that for two distinct workload patterns and three high-performance services, the dynamic approach offers the highest resource utilization and good performance (80% below response time threshold). When dynamic replication is combined with a small amount of static pre-allocation, performance improves to more than 90% below threshold. Our conclusion is that such a combined approach is the most promising model for delivering effective network services at reasonable cost.

Future work lies in several areas. The performance sensitivity to P and Q depends on the workload. Techniques for adaptively selecting these parameters are an interesting area of future work. The virtual service abstraction is based on stateless services. Extension to stateful services and issues relating to replica consistency will also be examined. A complete implementation of the per-client proxy mechanism for replica re-selection and fault transparency is also planned. Finally, the issue of replica placement becomes important when the client requests are not evenly dis-

persed but geographically clustered. We plan to investigate techniques for replica placement based on predicting the location of future client accesses based on prior accesses.

7.0 Bibliography

[1] Akamai: `http://www.akamai.com`.

[2] ACCESS: `http://access.cs.washington.edu`.

[3] Y. Amir et al., “Seamlessly Selecting the Best Copy from Internet-Wide Replicated Web Servers,” *Proceedings of the 12th International Symposium on Distributed Computing*, September 1998.

[4] R. L. Carter and M. E. Crovella, “Server Selection using Dynamic Path Characterization in Wide-Area Networks,” *Proceedings of IEEE Infocom '97*, April 1997.

[5] R. L. Carter and M. E. Crovella, “Dynamic Server Selection using Bandwidth Probing in Wide-Area Networks,” Boston University Technical Report BU-CS-96-007, March 1996.

[6] E. Belani et al., “The CRISIS Wide Area Security Architecture,” *Proceedings of the USENIX Security Symposium*, 1998.

[7] H. Cassanova and J. Dongarra, “Netsolve: A Network Server for Solving Computational Science Problems,” *International Journal of Supercomputing Applications and High Performance Computing,* Vol. 11, no. 3, 1997.

[8] J. Czyzyk, M. Mesnier, and J. Moré, “The NEOS Server,” *IEEE Journal on Computational Science and Engineering*, 5 (1998).

[9] Entropia: `http://www.entropia.com.`

[10] M. Faerman et al., “Adaptive Performance Prediction for Distributed Data-Intensive Applications,” *Proceedings of SC 99*, November 1999.

[11] A. Ferrari et al., “A Flexible Security System for Metacomputing Environments, “University of Virginia CS TR-98-36, 1998.

[12] I. Foster and C. Kesselman, “Globus: A Metacomputing Infrastructure Toolkit,” *International Journal of Supercomputing Applications*, 11(2), 1997.

[13] A. Fox et al., “Cluster-Based Scalable Network Services,” *Proceedings of the Symposium on Operating Systems Principles*, 1997.

[14] A.S. Grimshaw and W. A. Wulf, “The Legion Vision of a Worldwide Virtual Computer,” *Communications of the ACM,* Vol. 40(1), 1997.

[15] A.S. Grimshaw, E.A. West, and W.R. Pearson, “No Pain and Gain! - Experiences with Mentat on Biological Application,” *Concurrency: Practice & Experience*, Vol. 5, issue 4, July, 1993.

[16] Groove: `http//www.groove.net`.

[17] M. Harchol-Balter, A.B. Downey, “Exploiting Process Lifetime Distributions for Dynamic Load Balancing,” *SIGMETRICS*, 1996.

[18] E.N. Houstis et al., “Enabling Technologies for Computational Science: Frameworks, Middleware, and Environments,” Kluwer Academic Publishers, 2000.

[19] N. Kapadia and J.B. Fortes, “PUNCH: An Architecture for Web-Enabled Wide-Area Network-Computing.” *Cluster Computing*, September 1999.

[20] B. Lee and J.B. Weissman, “Dynamic Replica Management in the Service Grid,” to appear in *IEEE 2nd International Workshop on Grid Computing*, November 2001.

[21] J. Lopez and D. O'Hallaron, “Run-time support for adaptive heavyweight services,” *Proceedings of the 5th Workshop on Languages, Compilers and Run-time systems* (LCR 2000), May 2000, Rochester, NY.

[22] NPACI-NET: `http://legion.virginia.edu/npacinet.html`

[23] H. Nakada, M. Sato, and S. Sekiguchi, “Design and Implementations of Ninf: towards a Global Computing Infrastructure,” *Journal of Future Generation Computing Systems*, Metacomputing Issue, 1999.

[24] B. Noble et al., “Fluid Replication,” *Proceedings of the Network Storage Symposium*, 1999.

[25] V. Pai et al., “Locality-Aware Request Distribution in Cluster-based Network Servers,” *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[26] SARA: `http://www.cacr.caltech.edu/SDA/digital_puglia.html`

[27] A. Vahdat et al., “Active Names: Flexible Location and Transport of Wide-Area Resources,” *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1999.

[28] M. Wang and T. Suda, “The Bio-Networking Architecture: A Biologically Inspired Approach to the Design of Scalable, Adaptive, and Survivable/Available Network Applications,” *2001 Symposium on Applications and the Internet* (SAINT 2000), January 2001.

[29] Web trace archive: `http://www.web-caching.com/traces-logs.html`

[30] J.B. Weissman, “Gallop: The Benefits of Wide-Area Computing for Parallel Processing,” *Journal of Parallel and Distributed Computing*, Vol 54(2), November 1998.

[31] B. Lee and J.B. Weissman, “The Service Grid: Supporting Scalable Heterogeneous Services in Wide-Area Networks,” *2001 Symposium on Applications and the Internet* (SAINT 2000), January 2001.

[32] J.B. Weissman, “Fault Tolerant Wide-Area Parallel Computing,” *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, International Parallel and Distributed Processing Symposium IPDPS*, May 2000.

[33] R. Wolski, “Forecasting Network Performance to Support Dynamic Scheduling,” *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, 1997.

[34] C. Yoshikawa et al.,” Using Smart Clients to Build Scalable Services,” *Proceedings of the USENIX Technical Conference*, January 1997.