Dynamic Replica Management in the Service Grid

Byoungdai Lee and Jon B. Weissman
Department of Computer Science and Engineering
University of Minnesota, Twin Cities
{blee, jon}@cs.umn.edu

Abstract

As the Internet is evolving away from providing simple connectivity towards providing more sophisticated services, it is difficult to provide efficient delivery of high-demand services to end-users due to the dynamic sharing of the network and connected servers. To address this problem, we present the service grid architecture that incorporates dynamic replication and deletion of services. We have completed a prototype implementation of the architecture in Legion. The preliminary results show that our architecture can adapt to dynamically changing client demands in a timely manner. The results also demonstrate that our architecture utilizes system resources more efficiently than static replication system.

1.0 Introduction

The popularity of the Internet has grown enormously since its early deployment in the 1970's. To support newly emerging applications in areas such as e-commerce and distributed high-performance computing for science and engineering, the Internet is evolving away from providing only simple connectivity towards providing more sophisticated services. As a result, diverse network services ranging from content-delivery provided by Web servers to high performance computational servers such as in NetSolve [9] are currently deployed across the Internet. However, due to the dynamic sharing of the network and connected servers, it is difficult to provide efficient delivery of high-demand services to end-users.

The Service Grid [8] is an infrastructure for generic service delivery that has been designed to address several bottlenecks of the current Internet. Most notably, lack of reliability, transparency, and efficiency in service delivery. Our solution is to perform **dynamic** replication and deletion of services in response to user demand and system outages. Replication is the process by which one or more copies of a service are made. Although the idea of replication is not new, it presents several interesting challenges in the context of network services. For example, both server loads and network closeness should be taken into account when selecting a site to host a newly created service replica. The main features of our architecture are as follows.

- *adaptive*: the Service Grid adapts to dynamic changes in client demand in a timely manner. When demand is high, a new service replica may be dynamically created. When demand is low, system resource utilization is increased by deleting the underutilized replicas from the system.
- *scalable*: information collection and replica management is distributed.
- *transparent*: clients do not need to consider the degree of service replication, where replicas are located in the network, and which replica is best for them, etc.

In this paper, we limit our discussion to the Service Grid replication mechanisms and focus on computationally-intensive high-performance services. We have completed a prototype implementation of our architecture in Legion [1], a wide-area object-based distributed computing system developed at the University of Virginia. The preliminary results show that our architecture can adapt to dynamically changing client demands in a timely manner. The results also demonstrate that our architecture utilizes system resources more efficiently than static replication systems.

2.0 Related Work

Many research groups [2][4][5][6][7][10][14][15][16][17] have developed algorithms and architectures for replica selection among statically replicated servers. Research into dynamic replication has been mostly limited to the Web environment. In Fluid Replication [3], clients monitor their performance while interacting with the service. When performance becomes poor, a replica is created automatically. To select a replica site, they employ a distance-based discovery mechanism. In contrast, our solution for replica creation considers the characteristic of the service in addition to communication. RaDaR [13] proposes a web hosting architecture that supports dynamic migration and replication of web pages. In their architecture, if utilization is above a threshold, the system decides to migrate the web page near the host where most of the clients requests are coming from. CgR[11] proposes replication for enhancing the current Web infrastructure. In their work, one primary server forms the root of a logical tree of replicated servers, which serve as part of the primary server's namespace. Using a client-side proxy, the client can access either the primary server or replicated servers. The Bio-Networking Architecture [12] is inspired by biological metaphors of population growth in nature and provides a highly distributed replication mechanism.

3.0 Architecture

Our architecture consists of three core components: Replication Manager (RM), Group Manager (GM), Site Manager (SM) (Figure 1). RM is the decision-maker for global replica selection, creation and deletion, and tracks the location and state of all the replicas. Clients belong to a particular GM, established at configuration time. When a client wants to access the service, it first contacts the GM assigned to it to receive binding information for a replica that can provide the best performance. Once receiving the binding information, the client can directly access the service replica without intervention by the GM.

Every site in the Service Grid runs a SM, whose primary job is to interact with the GM to determine the network performance between replicas and client groups. We believe that all hosts at one site are likely to see the same network performance when communicating with any host at a remote site. Therefore, the SM can reduce the overhead associated with collecting network performance since only one probe between sites is necessary to characterize the network performance between any pair of machines hosting a replica and a client respectively.

The GM maintains a cache of local replicas allocated to it by the RM over time. This replica pool is

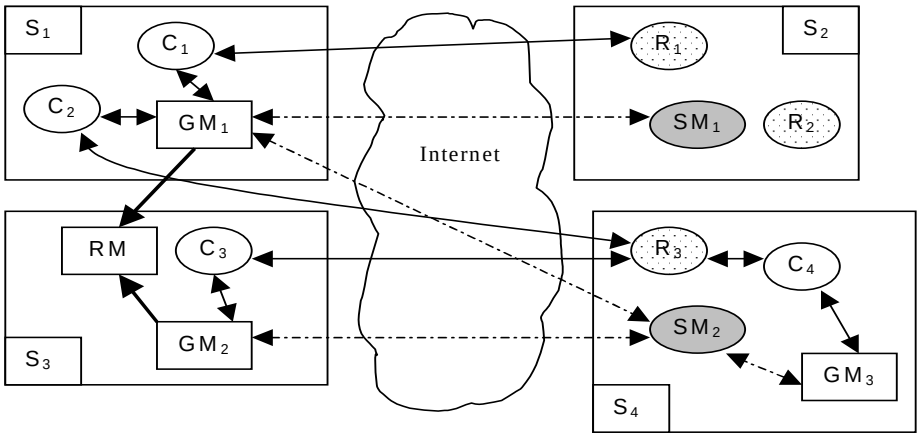


Figure 1: Service Grid Architecture: Three replicas (R_1, R_2, R_3) are running at two different sites (S_2, S_4). R_1 is dedicated to GM_1 and R_3 is shared across GM_1, GM_2 and GM_3. Since we chose GM_2 as the primary GM for R_3, only GM_2 needs to contact RM to update status information of R_3. Clients c_i belong to a group (GM_j), and may run on machines located outside the Service Grid perimeter.

available to the clients within the GM. Each replica maintains a list of GMs that are currently using the replica and reports its load status to them periodically so that the GMs can have up-to-date information on the replica status. Among all GMs that are sharing a replica, a primary GM is responsible for propagating the information to the RM so that the RM will also have an up-to-date global view of the system. With this protocol, the GM offloads much of the traffic that would otherwise reach the RM[1], promoting scalability. In addition to information collection and replica selection, the GM is also responsible for decision-making about when to acquire replicas and when to release replicas based on perceived performance and replica utilization.

Replica creation and deletion are initiated by the GMs in a distributed fashion. When the RM receives a replica acquisition request from the GMs, it decides whether to return an existing replica or to create a new replica based on the replica utilization by other groups. When a GM sends a replica release request and there is no other GM that is using the released replica, the RM puts that replica in an idle replica pool.

In addition to serving clients requests, a service replica periodically reports its status information to the GMs that are using it. However, the periodic status report may consume unnecessary network resources if the replica is not accessed frequently. To address this problem, the replica dynamically changes the window size for periodic status reporting. For example, if a replica is dedicated to a GM, this GM will require much less frequent updates. By selecting a larger window size, the network traffic between the GM and the replica can be reduced. On the other hand, if a replica is shared across many GMs, each GM cannot estimate the status of the replica accurately because it lacks global information. In this case, the replica sends its status more frequently so that each GM can have up-to-date information about the replica.

4.0 Replica Management

Both the GM and RM make replica management decisions to improve the performance that end-users will experience and to increase system resource utilization. To meet these requirements, algorithms for replica management must respond effectively to dynamically changing status of both client demand and replica utilization. For example, if a GM tries to acquire new replicas due to a transient short-term increase in clients demand, then valuable system resources may be wasted. The following sub-sections describe algorithms for both GM and RM replica management.

4.1 Replica Management in the GM

The GM runs three algorithms for replica management: replica selection, replica acquisition, and replica release. The challenge in designing algorithms for replica acquisition and replica release is that these algorithms should combine the goal of providing good performance to end-users with the goal of utilizing the system resources efficiently. The GM runs both the replica acquistion and replica release algorithms periodically based on a configurable time parameter.

4.1.1 Replica Selection

Replica selection is the process by which the GM selects a replica among its local cache of replicas that is predicted to provide the best performance for the requesting client. If the cache is empty, then replica acquisition is required. Replica selection in the GM is based on response time prediction. With up-to-date status information about replicas in its local cache, the GM can predict the response time of the service accurately.

Response time (T_{resp}) consists of four components: service time (T_s), waiting time (T_w), and communication time (T_c) and can be formulated as $T_{resp} = T_s + T_w + T_c$. The GM will select a replica that achieves a predicted minimum T_{resp} from its cached pool. Service time denotes the time necessary for completing the service at the replica. Waiting time is the time that the request will wait in the replica's waiting queue before being served by the replica and is computed by multiplying T_s by the replica queue length. The service time and the queue length in the replica is periodically updated by the replicas. Communication time is the time for sending the request to the replica and receiving the result from the replica. By periodically probing SMs associated with local replicas, the GM can maintain current network performance between replicas and a client site.

4.1.2 Replica Acquisition

When a service is overloaded or the network is congested, end-users will suffer increased response time. By dynamically acquiring additional replicas from the RM, the GM can provide better performance to its end-users. The acquired replica may be a replica that already exists or a newly created one. This decision is made by the RM based on the degree of sharing for the existing replica and system resource utilization.

1. Replicating the RM for very large Service Grids is the subject of future work.

```
ReplicaAcquisition(R_curr, R_prev, T_threshold, P, Q, M)
{
  /*
  Suppose current time is t
  R_curr : average response time of local replicas over current time window
  R_prev : average response time of local replicas over previous time window
  T_threshold: maximum threshold of average response time
  P: the number of consecutive time windows such that
       R(t) ≥ R(t-1) ≥ … ≥ R(t-P) and R(i) ≥ T_threshold, where
       R(i) : average response time at time window i ---------- (1)
  Q : the number of consecutive time windows such that R(i) ≥ T_threshold, t-Q ≤ i ≤ t ----- (2)
  p, q : variables for storing the number of time windows that satisfy (1) and (2), respectively
  */
  if (R_curr ≥ T_threshold)  {
    if (R_curr ≥ R_prev)  {
      p++;
    else
      p = 0;
    q++;
  } else
    p = q = 0;

  if (p ≥ P or q ≥ Q)
    send "replica acquisition" request to the RM
}
```

Figure 2: Replica Acquisition Algorithm.

Periodically, the GM computes the average response time T_{resp} for all local replicas over a recent time window. When the replica reports its status information, it sends not only the current snapshot of the waiting queue but also the average processing time of the requests that it served during the time window (see below).

Since the calculated response time includes both server load and network performance, we believe that it is a good indicator that represents performance that clients would experience. Once the average response time of each replica is computed, the GM next applies the replica acquisition algorithm to decide if it needs to acquire an additional replica from the RM.

The algorithm is based on a response time threshold ($T_{threshold}$) and two parameters, P and Q ($Q>P$), that control the degree of aggressiveness of replication (Figure 2). Each GM is free to select these parameters differently. In particular, the threshold will be client- and service-specific. With P, the GM can avoid acquiring unnecessary replicas due to temporary network congestion or transient increase of client demand. P requires that response time be monotonically increasing above the threshold for P consecutive time epochs. If the test

$$P_i(\textit{Performance of } R_i) = \left(\frac{\sum \textit{Average Processing Time of } R_i}{n_i}\right) + \left(\frac{\sum \textit{Communication Time between GM and } R_i}{m_i}\right)$$

$$P_{overall} = \frac{\sum P_i}{k}, \; k \textit{ replicas in the GM cache}$$

Assumptions :

1. *Each replica R_i reports its load status n_i times during the GM's time window*
2. *The GM probes each SM associated with R_i , m_i times during the GM's time window*

```
ReplicaRelease(U_threshold, P, Q)
{
  /*
  Suppose current time is t
  NumR: number of currently cached replicas within this group
  U_threshold: minimum utilization
  P: the number of consecutive time windows such that
       U(t) ≥ U(t-1) ≥ … ≥ U(t-P) and U(i) ≥ U_threshold, where
       U(i) : utilization of a replica at time window i – (1)
  Q : the number of consecutive time windows such that U(i) ≥ U_threshold, t-Q ≤ i ≤ t (2)
  M is the minimum number of replicas to retain within the group
  */

  if (NumR <= M) return;
  replia_bins = empty /* replica_bins contains candidate replicas for release */
  for (each local replica)  {
     U_curr <- utilization of the replica over current time window
     U_prev <- utilization of the replica over previous time window

     if (U_threshold ≥ U_curr)  {
       if (U_prev ≥ U_curr)  {
          p++;
       else
          p = 0;
       q++;
      } else
         p = q = 0;

      if (p ≥ P or q ≥ Q)
        insert the replica into replica_bins
   }

   find a replica in replica_bins whose utilization is the smallest and return the replica
}
```

Figure 3: Replica Release Algorithm.

on *P* fails, we apply the *Q* test which is less restrictive. *Q* requires that the response time be simply above threshold for *Q* consecutive time epochs. Smaller values of *P* and *Q* lead to more aggressive replication. *P* allows an immediate response to rapidly growing demand, while *Q* permits some performance fluctuation and is more conservative (*Q*>*P*). Counters for *P* and *Q* are reset when the measured response time goes below threshold to help eliminate the risk of a transient response.

4.1.3 Replica Release

If the GM caches more replicas than it needs to meet its current threshold, some replicas may be idle and system resources would in turn be wasted. The GM will release unnecessary replicas back to the RM as long as clients requests can be serviced within the response time threshold.

As a utilization metric, the number of requests that the replica has served within this group over the time window is used. This is the local utilization. The replica may be actively used by other groups. As in replica acquisition, the GM should not respond to a transient decrease in client demand. We apply the same principle within the release algorithm as in replica acquisition. The difference is that the algorithm should be applied against each local replica. Figure 3 shows the replica release algorithm.

As in the replica acquisition algorithm, there are three configurable parameters: $U_{threshold}$, *P* and *Q*. If there are multiple underutilized replicas, the GM selects the least frequently used replica and releases it. Release does not mean that the replica is deleted. Deletion is a decision that is ultimately up to the RM, analogous to replica creation. A replica that has been released by all GMs is idle, and is a possible candidate for deletion by the RM. In the current version of the release algorithm, it will release at most one replica each time it is run (at each time interval). In addition, the GM indicates the minimum number of replicas it wishes to keep cached (*M*) irrespective of their utilization.

Table 1: Service Grid testbed

Host Name	O/S	Benchmarked Processing Time (ms) for 400x400 matrix mult
University of Minnesota (UMN)		
juneau.cs.umn.edu	Linux	11776.624
sitka.cs.umn.edu	Linux	11687.796
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
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

4.2 Replica Management in RM

The RM must perform the following replica management tasks, replica acquisition and replica release. Replica acquisition 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 groups threshold while not compromising the performing of other groups sharing the replica. If these criteria cannot be met, the RM will create a new replica. Replica release simply indicates to the RM that the replica has been removed from the GMs cache. The RM notifies the replica of this change which allows the replica to eliminate any status updates to this GM savings network resources. In addition, the RM periodically checks the status of idle replicas (replicas released by all GMs). The RM is configured to maintain a minimum pool of idle replicas in the system. When this limit is exceeded, the RM will delete the replica that has been idle for the longest period of time.

5.0 Experiments

We have built a Service Grid prototype using the Legion system, a wide-area object-based distributed computing infrastructure [1]. In our Service Grid prototype, a service replica is implemented by a Legion

Table 2: Configurable parameters for GM

Parameters	UMN	UVA	UTSA
maximum threshold (ms)	65000	85000	55000
P (replica acquisition)	3	3	2
Q (replica acquisition)	7	7	5
minimum utilization	5	5	3
P (replica release)	4	4	4
Q (replica release)	9	9	9

object and accessed via remote method invocation. In this paper, we present results for a matrix multiplication service. In the prototype, the replica acquire/release algorithms are run every 2 minutes, 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.

We present data for a subset of our larger testbed (Table 1). Clients are deployed across three sites (two of which are Service Grid sites), University of Minnesota (UMN), University of Texas at San Antonio (UTSA) and University of Virginia (UVA). In our experiments, UMN has 8 clients, UVA has 8 clients and UTSA has 5 clients. We generated a synthetic workload of client requests to the matrix multiply service that is shaped to contain both demand growth and decline (Figure 4).

5.1 Response Time Prediction

Accurate prediction of response time is critical to replica management. The experimental results show that the response time prediction can be done with high accuracy for this service, often within 10% (Figures 5). However, predictions can drift when the rate of request generation exceeds the rate of information exchange. We are implementing an adaptive mechanism for controlling the interval for information collection.

5.2 Performance Comparison

To compare performance of our replica management scheme, we selected several parameter values for experimentation (Table 2). When a GM is created, it acquires an initial replica from the RM and always maintains at least one replica in its local cache (i.e. M=1).

We compare performance against pure static replication (2 and 8 replicas), and hybrid static/dynamic replication. In the hybrid approach, two replicas are statically replicated on UMN and UVA and each GM starts with these two replicas. In dynamic and hybrid schemes when client demand increases, the GM acquires additional replicas, and when clients demand decreases, the GM releases underutilized replicas until the number of local replicas is at most two. Replica acquisition may return an existing replica or may result in the creation of a new replica depending on the current system state.

Figures 6-7 show the response time measured at the client sites under the different schemes. Static replication with 8 replicas achieves the lowest response time as it uses a large number of replicas. However, it suffers from very low utilization (Figure 8). In contrast, static replication with 2 replicas cannot meet demand in the high-demand regimes and the response time is very poor. In fact, the response time peak follows the workload peak in duration.

Both dynamic and hybrid schemes are able to achieve performance below threshold (the performance objective) at a lower cost (higher utilization) Interestingly, they have the effect of “shortening” the impact of the workload peak. It appears that 4 static replicas would likely achieve the best overall performance given this workload, but in general this cannot be predicted apriori. In addition, suppose that only enough resources for two replicas were available at the time this service was initially deployed. A static scheme cannot exploit newly available resources were they to become available later. Dynamic replication schemes make much more efficient use of system resources because they replicate only when necessary and release resources when demand declines. The hybrid scheme achieves a good balance of low response time and good utilization and appears to be a promising approach.

6.0 Conclusions

We described a new architecture for the efficient delivery of high-demand network services, the Service Grid. To achieve scalable, reliable, and adaptive performance, the Service Grid performs dynamic service replication and deletion in response to changing client demand. It implements an algorithmic framework for dynamic replica management that controls the degree of aggressiveness in creating and removing replicas. A Service Grid prototype was built using the Legion system and preliminary results for a compute-intensive service indicate that dynamic replica management can be done to meet end-user performance goals at a much lower cost than fully static replication. Future work includes investigating the sensitivity of both response time and utilization to the parameters of the replica management algorithms.

7.0 References

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

[2] 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

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

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

[5] Ellen W. Zegura et al., “Application-Layer Anycasting: A Server Selection Architecure and Use in a Replicated Web Service”, *IEEE/ ACM Transactions on Networking*, Vol.8, No. 4, August 2000

[6] James D. Guyton and Michael F. S., “Locating Nearby Copies of Replicated Internet Servers”, University of Colorado, TR CU-CS-762-95

[7] Jaspal Subholk et al., “Automatic Node Selection for High Performance Application on Networks”, *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1999

[8] Jon B. Weissman and Byoungdai Lee, “The Service Grid: Supporting Scalable Heterogeneous Services in Wide-Area Networks”, *Proceedings of Symposium on Applications and the Internet*, January 2001

[9] H.Casanova and J. Dongarra, “NetSolve: A network server for solving computational science problems”, University of Tennessee TR CS-95-313, 1995

[10] Martin Aeschlimann et al., “Preliminary Report on the Design of a Framework for Distributed Visualization”, *Proceedings of Parallel and Distributed Processing Techniques and Applications*, August 1999

[11] M. Baentsch, et al., “Enhancing the Web Infrastructure- From Caching to Replication”, *IEEE Internet Computing, Vol. 1, No. 2*, March 1997

[12] Michael Wang et al., “The Bio-Networking Architecture: A Biologically Inspired Approach to the Design of Scalable, Adaptive and Survivable/Available Network Applications”, *Proceedings of Symposium on Applications and the Internet*, January 2001

[13] M. Rabinovich et al., “A Dynamic Object Replication and Migration protocol for an Internet Hosting Service”, *IEEE International Conference on Distributed Computing Systems*, May 1999

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

[15] Sandra G. Dykes et al., “An Empirical Evaluation of Client-side Server Selection Algorithms”, *Proceedings of IEEE Infocom ‘00*, March 2000

[16] 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

[17] Y. Amir et al., “WALRUS - a Low Latency, High Throughput Web Service Using Internet-wide Replication”, *Proceedings of the 19th International Conference on Distributed Computing Systems Workshop*, June 1999

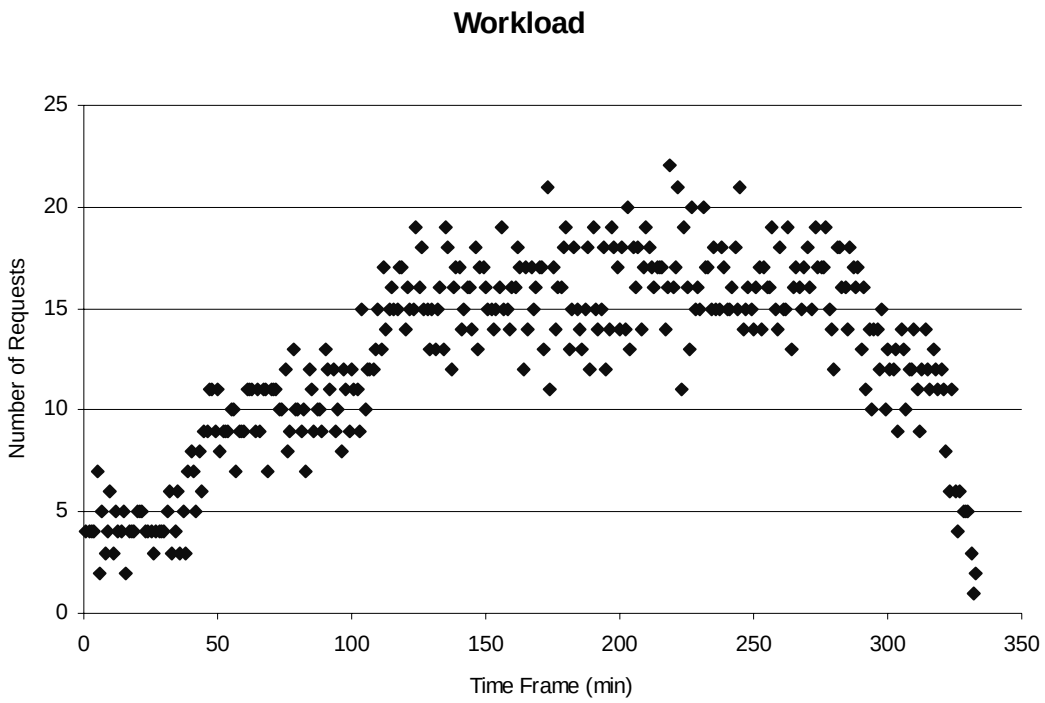
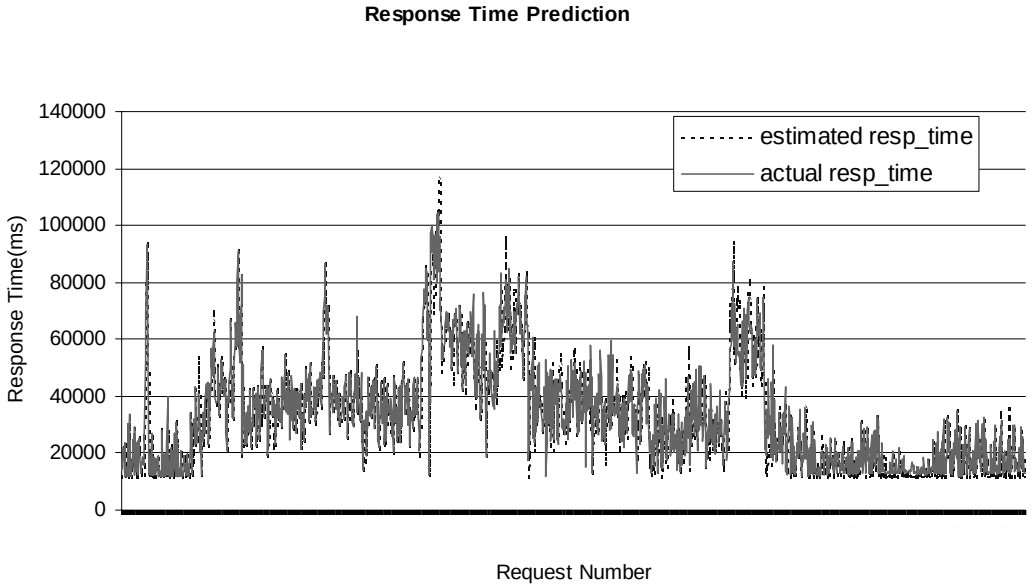


Figure 4: Stair-shaped client workload. Each point at *t* on x-axis in the graph represents the number of total client requests generated between *t* and *t-1* minute.



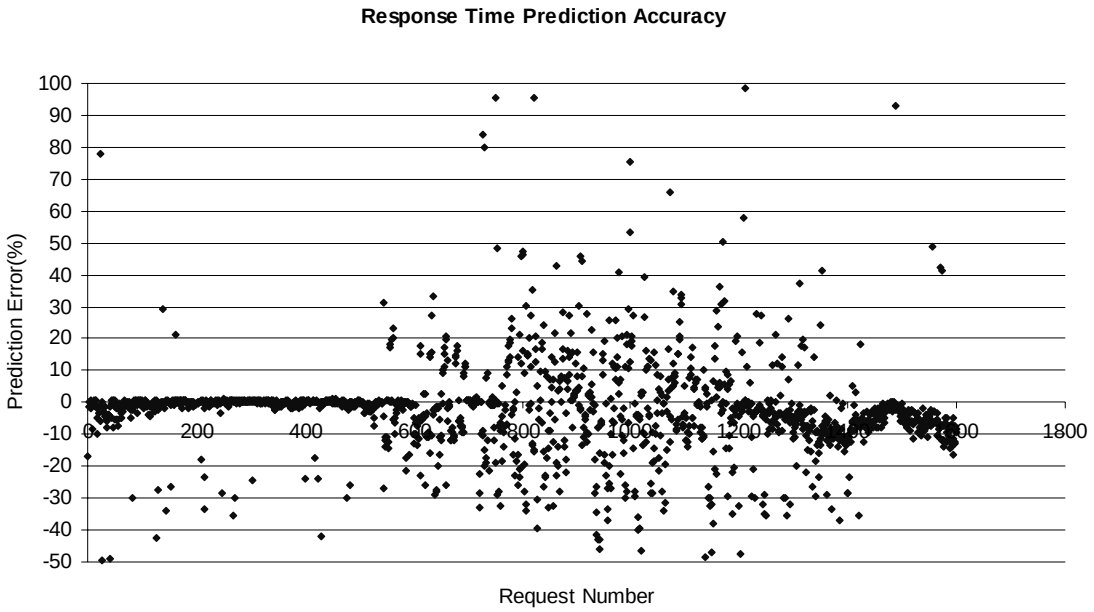


Figure 5: Accuracy of response time prediction.

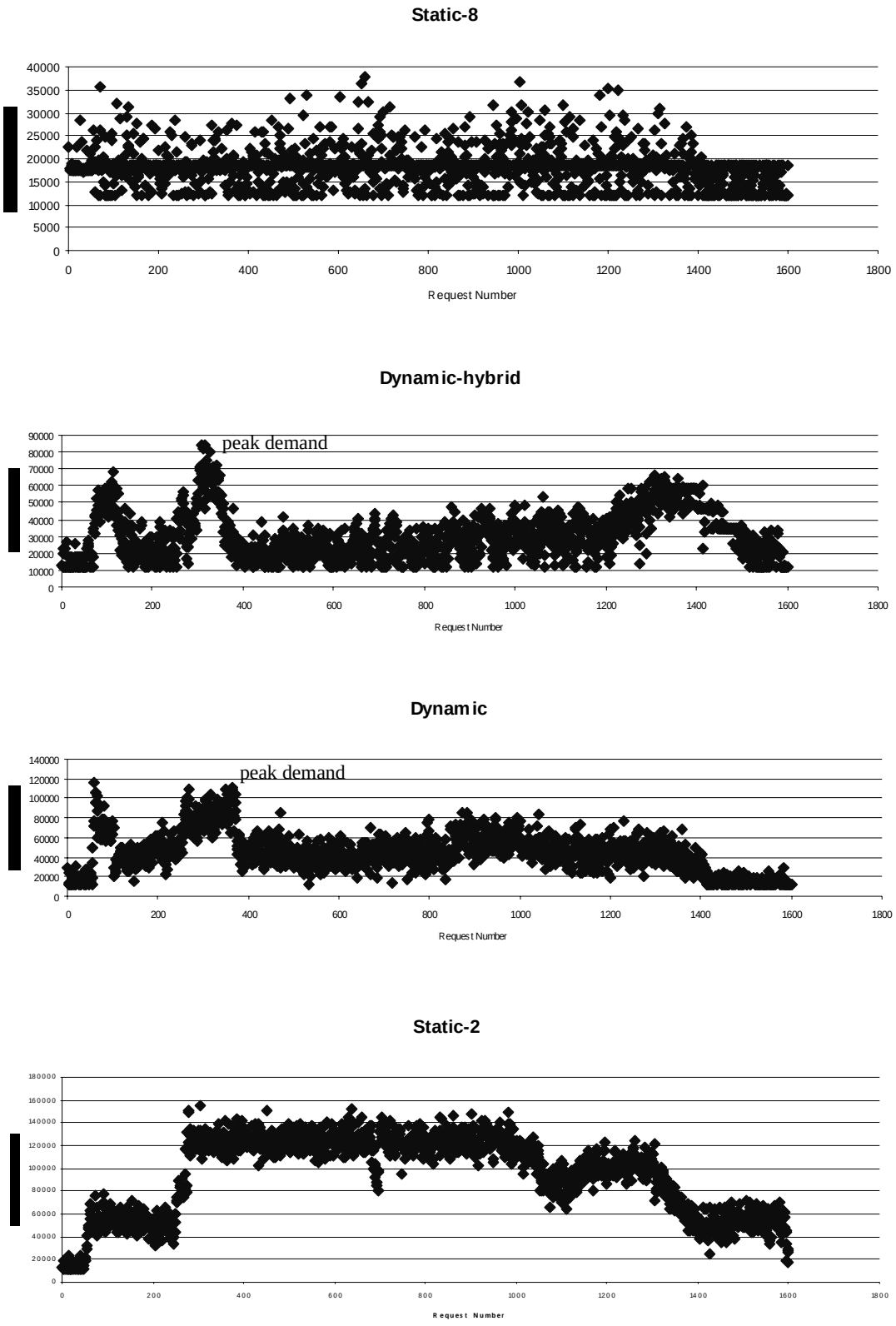


Figure 6: Measured response time under 4 replica acquisition strategies**.**

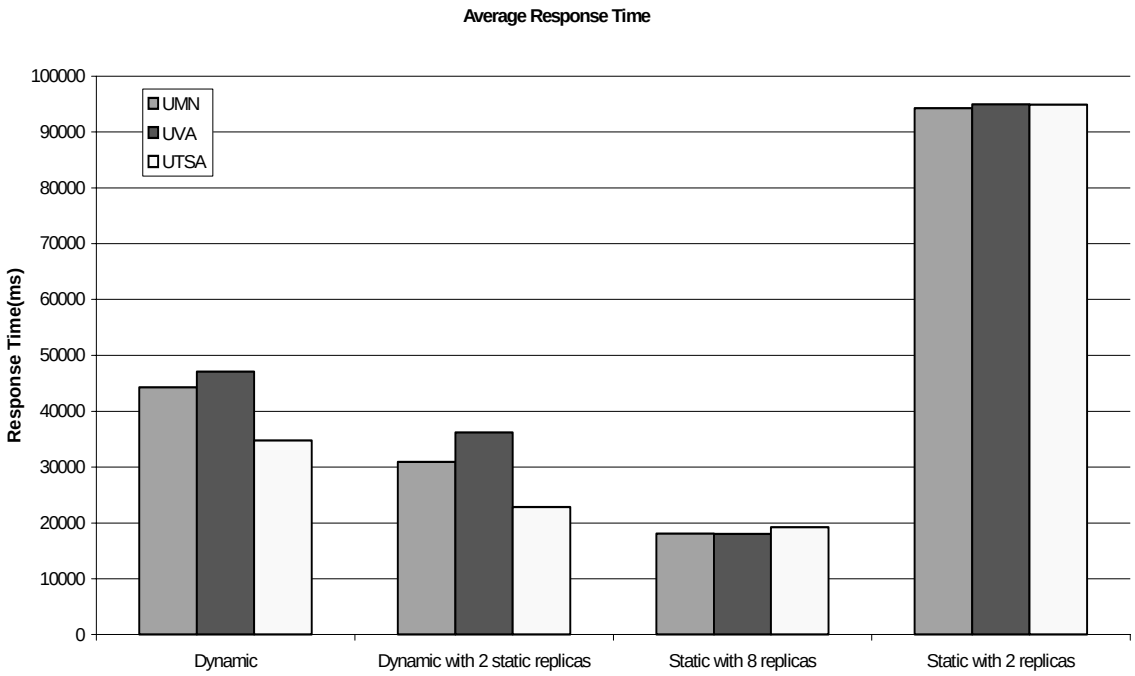


Figure 7: Comparative performance for different replica acquisition policies.

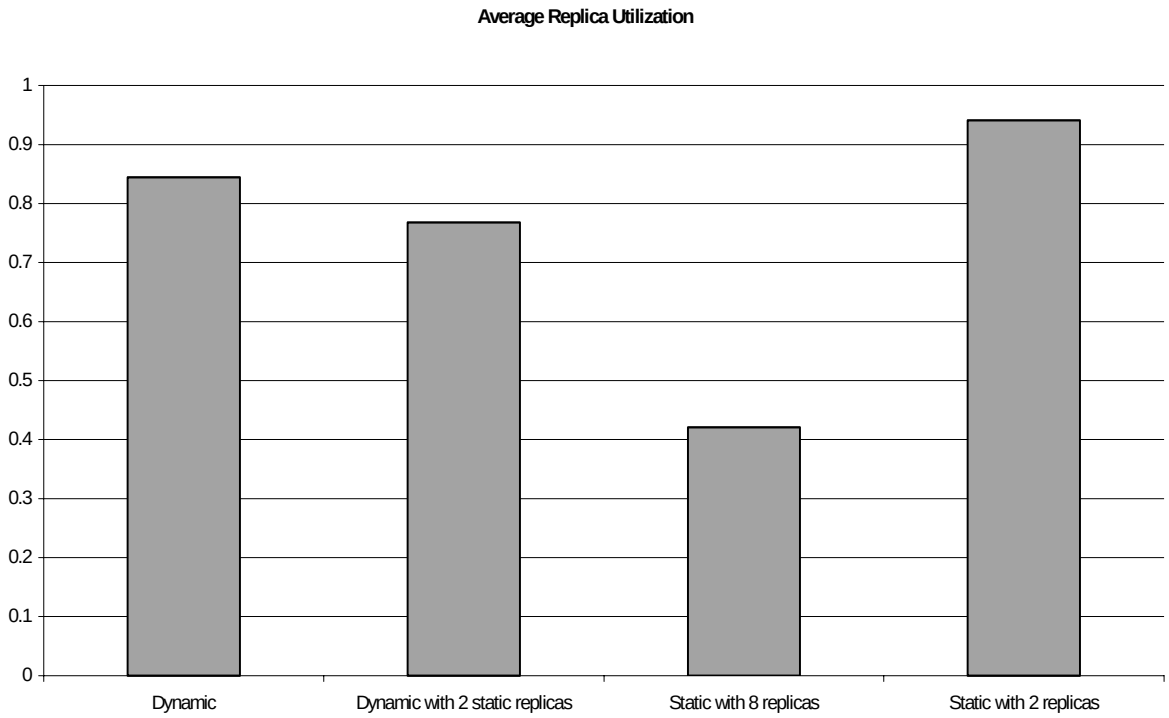


Figure 8: Comparative utilization for different replica management policies**.**