An Experimental Prototype for Scalable Server Selection

Mohamed-Vall O. Mohamed-Salem, University of Montreal
Jun Chen and Gregor v. Bochmann, University of Ottawa
Johnny W. Wong, University of Waterloo

Abstract

An experimental prototype for server selection using an independent brokerage service is described. This prototype is composed of four main components: instrumented Apache Web servers, monitoring agents, a QoS broker, and client emulator. The role of the broker is to distribute client sessions to a set of replicated servers. It is designed to support different types of selection policies and has the capability to collect performance data from the servers. We include in our description the technique used to instrument Apache servers and our implementation of the server-broker protocol that we have developed. Our implementation of the QoS broker and the technique used to collect data for the performance parameters of interest are also described. We use our prototype to study the performance of server selection algorithms under realistic conditions. The experimental environment and an analysis of the experimental results are presented.

1. Introduction

In recent years, we have seen a significant growth in the development and deployment of distributed applications over the Internet. These include electronic commerce applications that support the dissemination of information regarding a company's products, and the sale of goods and services. For large-scale deployment, the server must be able to scale to many users. A common approach to improve scalability is server replication. An important challenge in replicated server architecture is how a client may locate an appropriate server without being aware of the specific details of replica organization, and how can this process scale to a large number of users and replica.

In our recent work [1], we investigated the delegation of server selection functionality to an independent brokerage service. Specifically, we studied the use of a "broker" to distribute load to a set of replicated servers. Our replicated server architecture has the following properties:

- Scalable to a large number of clients.
- Support for the provision of quality of service.
- Support for feedback mechanisms by which up-to-date performance information on system components is available.

In our architecture, a broker is used to assign clients to servers at the beginning of their sessions. After server selection, each client may interact directly with the server for a period of time specified by the broker. This is different from other approaches where server selection is done on a per user request basis (see for example [2]). Our architecture allows the broker to gather information about the status of each server and use such information for load balancing purposes. Performance data are collected by monitoring agents at the servers and sent to the broker. The protocol between the broker and the monitoring agent is quite straightforward. It simply involves the periodic transmission of performance data to the broker. Our architecture also allows for a flexible organization of resources used by web sites. The broker could be at the server site under the same authority as the replicated servers. This is applicable, for example, to sites with heavy load and high degree of replication. Different sites may also share the same broker. In this case, the broker could be an independent brokerage service that

manages the assignment of servers for affiliated sites.

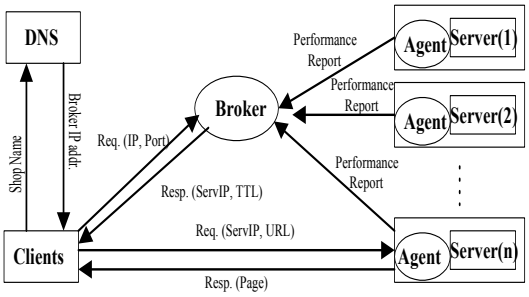

Figure 1: Basic architecture and interactions between its components

The negotiation between broker and client is carried out at the beginning of a session. The protocol between the client and the broker is illustrated in Figure 1. The client sends a server selection request to the web site address, specified by an URL. This URL is mapped by the DNS to the IP address of the broker. The broker, upon receiving the client's request, selects a server and returns the IP address of that server, together with a quantum size, to the client. The client then sends its requests to this IP address; it also caches the IP address to be used for subsequent requests. The cache entry will be deleted when the quantum expires. Note that the above protocol needs to be implemented at the client and we assume that this is possible. Note also that the client-broker protocol can be extended to include QoS negotiation, which can be based on client classification, client performance requirement, or server availability.

We have developed a prototype for the above architecture and conducted experiments to evaluate its performance. This prototype is composed of the following four components: instrumented Web servers, monitoring agents, QoS broker, and client emulator. The objective of the experiments is to evaluate the performance of server selection policies in a real environment.

This paper is organized as follows. Section 2 describes the technique we have used to instrument the servers and the reporting protocol between monitoring agent and broker. We also describe in the same section, the various server performance parameters of interest to our study and how they are collected. Section 3 describes the implementation of the QoS broker and its public access points. Section 4 contains a description of the various server selection policies implemented by the broker and used in our experiments. In Section 5, we describe the test environment and analyze the experimental results. Finally, Section 6 contains a summary of our findings and some concluding remarks.

2. Collection of Performance Data

In our prototype, each server is associated with an agent that monitors its performance and periodically reports this information to the broker. Figure 2 shows the entire monitoring system, which consists of an instrumented Apache Web server, a monitoring agent that resides on the same machine on which the server is running, and a remote QoS broker.

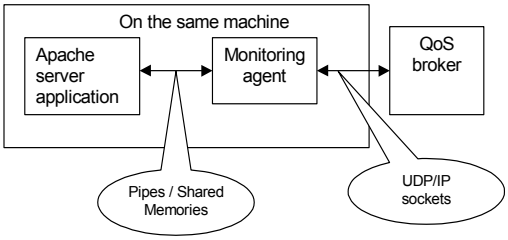

Figure 2: Monitoring system

2.1 Instrumentation of Servers

The parameters that need to be measured at the server are gathered using function calls inserted as probes within the code of the server. Triggers are sent to the monitoring agent when these function calls are executed. The agent would thus know when each transaction started and when it is completed. It is also possible to receive other information from the server like for instance the identity of the client and details about the requested pages. The communication between the server and the agent is done by means of standard inter-process communication (IPC) mechanisms like for instance pipes or shared memory.

We use in our prototype an application programming interface (API) called Application Response Measurement (ARM). The ARM API is made up of a set of functions that are contained in a

shared library. Calls to these functions are inserted at appropriate places in the code. Performance measurement agents that support the API implement these functions. The advantage of this approach is the isolation of server instrumentation from monitoring actions that need to be taken while an application is running. Application providers have the freedom to choose any monitoring agent that best meets their need without the application needing any changes. The ARM API that we are using has the following functions:

- *arm_init*: This function is used in the initialization phase of an application. In our implementation of the API, we use this function to initiate the communication between the monitoring agent and the monitored Web server.
- *arm_getid*: This function is used to obtain a unique identifier that is used to characterize a class of transactions. It allows us to monitor different types of request that a server is handling. However, in our implementation, we do not use this feature and hence do not differentiate between the types of request that the Web server is processing.
- *arm_start*: This function signals the start of execution of an instance of a transaction of a particular class. It returns a unique identifier that is passed as a parameter for two other functions: *arm_update* and *arm_stop*.
- *arm_update*: This is an optional function that can be called any number of times after the start of a transaction and before it stops. It can be used to signal the progress of a transaction.
- *arm_stop*: This function signals the completion of a transaction.
- *arm_end*: This function is used by an application to do the global clean up of all resources used by the ARM API. In our implementation, when this message is received, the monitoring agent sends a message to the broker signaling that the server application is exiting.

Figure 3 shows the instrumentation of an Apache server using the ARM API to monitor the start and the completion of a client request.

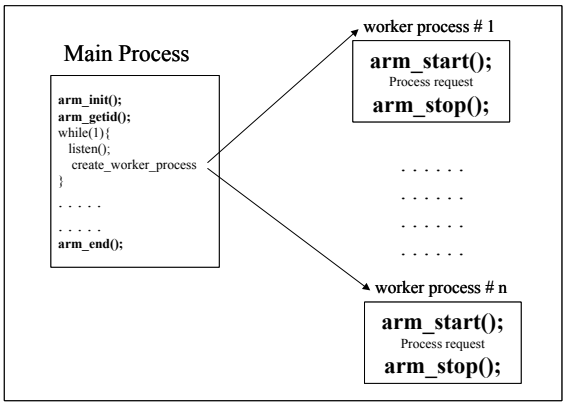

Figure 3: Instrumentation of an Apache server

2.2 Computation of Performance Data

Data on the servers' run-time behavior are collected by monitoring agents and periodically transmitted to the broker. Our monitoring agent collects data for the following parameters, which are used in our server selection algorithms [1]:

2.2.1 Mean response time of server

The agent has information about when a transaction starts and when it stops. These correspond to the time at which it receives triggers for *arm_start()* and the corresponding *arm_stop()*, respectively. The difference between the two values gives the response time for that particular transaction. During each measurement interval, the agent keeps track of the number of requests completed by each server and the sum of the response times for these requests. These two values are then used to compute the average response time of the server for that particular measurement period.

2.2.2 Mean service time at server

The mean response time includes the mean waiting time and mean service time. An Apache server is available in a multi-threaded architecture for the different Windows platforms and a multi-process architecture for the UNIX platform. For the multi-threaded architecture, the server uses a main process which manages a FIFO queue for waiting jobs, and a set of worker threads which continuously remove and process one job at a time from the head of the queue. Our instrumentation is done in such a way that it allows us to measure the

time spent by each request in the waiting queue. The mean service time can then be obtained as the difference between the mean waiting time and the mean response time.

In the multi-process architecture, a new process is created for each new connection. The master process at the server does not implement a waiting queue for jobs and it is therefore not possible to isolate the waiting time from the service time. In this case, we measure the mean service time at each server off-line and statically feed this information to the agent and the broker. The measurement is done as follows. For each server that we use in our experiments, we measure the mean response time when the machine is serving only one request at a time and when no other jobs are running on it. The load generator is configured to emulate one single client for a long enough time period to ensure that documents of different sizes are fetched. The client does not experience any waiting time and hence the mean response time can be used as an estimate for the mean service time.

2.2.3 Mean request arrival rate

The load on each server is calculated as the number of requests received during a measurement interval. Every time that the agent receives a trigger for an *arm_start(),* the total number of received requests is incremented by one. The request arrival rate is computed by dividing this total number by the length of the measurement interval.

2.2.4 Utilization of a server

The utilization of a server in a measurement interval is calculated as the fraction of time that the server is busy servicing requests. It is computed as the ratio of the total service time in a measurement interval over the length of the interval.

2.2.5 Mean think time

The mean think time is defined to be the time between consecutive requests from the same client. It should be noted that from the perspective of the server, each request could be for an individual object, such as a static file, an image, a dynamic request, etc. Each client is identified by its IP address, which is determined within the Apache server code itself and passed to the monitoring agent. The monitoring agent keeps track of all the think times of each client during a measurement interval, and uses this information to compute an overall mean think time at the monitored server.

2.3 Performance Reporting Protocol

Communication with the broker is carried out using UDP packets. A proprietary reporting protocol is used between monitoring agents on the server side and the broker. This protocol allows the agent to register with the broker, send performance updates, and sign off when the server is going down. As soon as the agent is activated on the server side, it registers itself with the broker by sending information on the server it is monitoring (address and characteristics) to the broker. When the broker accepts the registration, it returns a unique identifier to be used for all subsequent interactions and a desired length for the measurement interval. During the lifetime of the agent, the collected data is summarized in a report that is sent to the broker at the end of each measurement interval. The message exchanges between the broker and the agent are defined as follows (see Figure 4):

- *Registration*: A monitoring agent sends this message to register the machine on which it is running. It is sent each time the agent is restarted. It contains the identification of the agent and the necessary information on the server associated with it.
- *Registration Acknowledgement*: This message acknowledges the reception of a *Registration* request. It contains control information that the broker would like to communicate to the monitoring agent, like for instance the length of the measurement interval.
- *Status Query*: This message is used to request the status of a server; it can be sent at any time.
- *Update*: This message is the periodic report sent by the monitoring agent to the broker. It contains a summary of the performance data collected during the last measurement period. This message is sent at the end of a measurement period or in response to a *Status Query* message from the broker.
- *Stop*: The stop message is sent by an agent requesting the broker to stop using the server.

It is typically sent when a server is going down for maintenance. Upon reception of this message, the broker returns a *Stop Acknowledgement* and stops assigning clients to this server.

- *Stop Acknowledgement*: This message is sent by the broker to confirm the reception of a *Stop* message. After the reception of such a message, an agent stops all data exchanges with the broker.

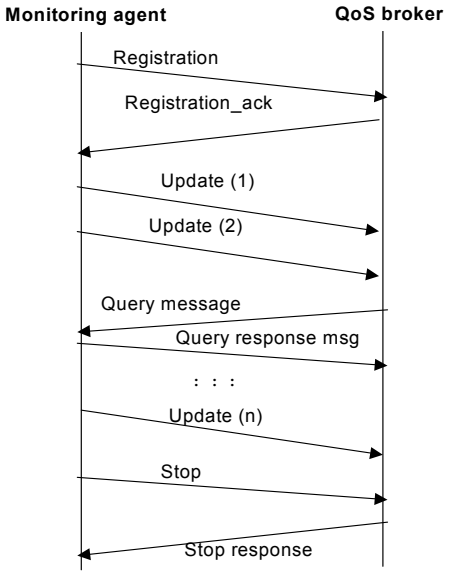

Figure 4: Protocol between monitoring agents and the broker

3. Quality of Service Broker

Our broker implementation is composed of two main modules: a performance monitor and a negotiator. These two modules run in parallel as separate threads inside the address space of the broker, and share a common data structure that contains information on servers.

3.1 Performance Monitoring

The monitor module implements the protocol described in Section 2.3. It listens on a known port for individual servers to join, receives performance data about each server, and stores these data for the negotiator. When a server registers at the broker, a new entry for the server is created and added into the server table. The broker maintains two types of information for each server: static information such as the name and address of the server, service port, etc., and dynamic information that is regularly updated by monitoring agents as performance data are collected.

3.2 QoS Negotiator

The public interface of the broker is accessible to the outside world only through the negotiator. This interface allows clients to negotiate the choice of a server for their sessions. Each client request specifies the name of the server cluster where one of the servers is to be selected. This name must be among those managed by the broker. Upon receiving a selection request from the client, the negotiator invokes the scheduling policy to select a server for the client session.

Our prototype is built on the assumption that software on the client side is capable of negotiating with the broker for the server selection. This software is structured as an intermediate engine that allows standard browsers to communicate properly with the broker. This engine listens on the service port of the server cluster and intercepts incoming requests from clients. For each request, the intermediate software engine interacts with the broker to choose a server. The engine then uses the standard HTTP temporal redirection mechanism to redirect the user to the selected server.

At the heart of the negotiator, a scheduler handles server selection. The broker may implement any number of server selection policies. Each policy implements its own scheduler. At the starting time of the broker, the list of the available policies is loaded, and the administrator of the broker can dynamically activate the desired selection policy and to enter the necessary parameters through the user interface. Examples of policies are described in the next section.

4. Server Selection Policies

We distinguish between “static” algorithms and “dynamic” algorithms depending on whether data on run-time server behavior are used in server selection decisions or not. For static algorithms, run-time data are not used, and the algorithms under consideration are:

- *Round robin* (RR): This algorithm is similar to that reported in [2]. Suppose there are *N* servers. Servers are selected in cyclic order, i.e., server 1, server 2,…, server *N,* repeatedly.
- *Random selection* (RAN): In this algorithm, a server is randomly selected from the cluster of servers; each server has the same probability of being selected.

The dynamic algorithms implemented in our prototype require the availability of measurement data for the following parameters for each server:

- mean response time (r)
- mean service time (s)
- utilization (u)
- mean think time (h)
- mean request arrival rate (λ).

These parameters are measured by monitoring agents as discussed in Section 2. Two dynamic algorithms are considered. They are:

- *Least active sessions* (LAS): At a given server, the number of active sessions can be estimated by $NS = \lambda(r + h)$ and the maximum number of active session that the server can support by $NS_{\max} = 1/s$. Under LAS, the server with the smallest value of $NS / NS_{\max}$ is selected.
- Least utilization (LU): The server with the lowest utilization, estimated by u, is selected.

The details of these two algorithms can be found in [1].

5. Experimentation

Our experimental environment is as follows. A broker that manages the assignment of users to servers runs on its own machine. Each server machine runs an instrumented Apache Web server and a monitoring agent that collects performance data.

Two server clusters were used in our experiments. The first cluster, referred to as homogeneous cluster, consists of three servers with very similar capacities. The mean service time of a request at each of these servers is around 0.0035 seconds and all server machines are connected to the same 100 Mb/s LAN. The second cluster is heterogeneous. It is composed of three servers with noticeably different capacities. The mean service times at these servers (denoted by S1, S2, and S3) are 0.003, 0.006, and 0.015 seconds respectively. Server S2 is located at the University of Montreal while the other two (S1 and S3) and the load generation machine are on the same 100 Mb/s LAN at the University of Ottawa.

The load on servers is generated using a modified version of the tool SURGE developed at Boston University [4]. This tool allows the creation of real workload files at servers and contains an emulator that mimics the behavior of normal Web clients. The SURGE software can generate a sequence of URL requests, which exhibit representative distributions for document popularity, document size, request size, embedded document count and think time. We modified the tool to include the negotiation phase with the QoS broker. The total number of documents stored at each Apache server is 2000. Different sizes are generated by the tool, as explained in [4]. For the case of heterogeneous cluster, the total load generated during our experiments is around 660 requests/sec by 1650 concurrent clients. This yields an average utilization of the cluster of 82%. On the other hand, the total load generated for the case of homogeneous cluster is around 1080 requests/sec by 2700 concurrent clients, and the average utilization of the cluster is 85%. In each experiment, the measurement data are collected for periods of 600 seconds.

5.1 Results

As expected, policies such as RR and RAN perform well for the case of homogeneous servers. The broker achieves very good load balancing without using any dynamic information on the status of servers. We thus conclude that with a homogeneous cluster, simple policies such as RR and RAN should be used.

For the case of heterogeneous cluster, the performance achieved by the four policies is shown in Figure 5. We observe that RR and RAN are inferior to the dynamic algorithms LAS and LU with respect to mean response time. This can be explained as follows. RR and RAN do not take the difference in server capacities into consideration when a server is to be selected. This results in the

creation of load imbalance among the servers. As shown in Figure 6, the slowest servers S2 and S3 are highly utilized, while the fastest server S1 operates at a modest level of utilization (under 70% of its capacity).

The two dynamic algorithms LU and LAS learn the effective capacity of the servers through performance data and dynamically adjust the distribution of load. LU always dispatches sessions in a way to keep servers utilized at the same level. This results in good load balancing (see Figure 7). The corresponding results for LAS are shown in Figure 8. We observe that LAS is less capable of load balancing when compared to LU. One should note, however, LAS is based on the ratio of the estimated number of active sessions at a server and the estimated maximum number of sessions that the server can handle. The number of active sessions increases proportionally with the mean response time. A slower server has fewer active sessions that it can handle and hence fewer new sessions assigned by the broker. It follows that LAS achieves the best response time and hence the best throughput at the slowest server, among the four policies tested. Figure 9 shows the improvement of the response time for the slowest server achieved by LAS compared to that achieved by LU.

Under LAS a larger fraction of the load is shifted to the faster server. Specifically, the fastest server handled 68.5% of the requests, followed by 24.5% at the second fastest server. The corresponding values under LU are 59% and 30% respectively. Finally, Figure 10 summarizes the performance of the four different policies for a cluster of heterogeneous servers. In an earlier study, we have evaluated the performance of the four policies presented in this paper using extensive simulation [1]. The conclusions obtained in [1] are very similar to those obtained by experimentation.

6. Conclusion and Discussions

We have built a prototype for our architecture and used this prototype to evaluate the performance of four server selection policies in a real environment. The experimental results showed that with homogeneous servers, the performance of these policies is very similar. A static algorithm such as RR or RAN is therefore effective in load balancing for the case of a homogeneous cluster. For heterogeneous servers, however, static algorithms are not effective, and dynamic algorithms (e.g., LAS and LU) that make use of performance data perceived represent a robust and more predictable alternative.

Dynamic algorithms require the server to be instrumented and to have the capability of reporting performance data. We have adapted for our prototype the standard instrumentation ARM API and developed a monitoring agent to be associated with each server. Instrumentation of servers consists of placing functions at the appropriate places in the source code. The instrumentation and the associated agents provided us with capabilities in QoS management and scalability that we could not have otherwise.

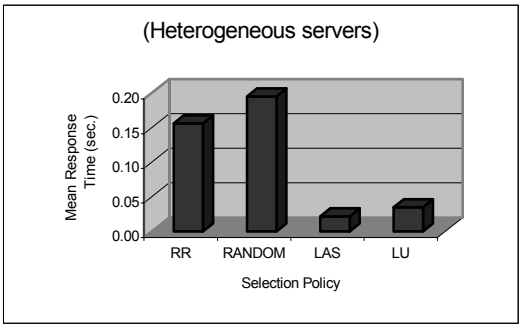

Figure 5: Mean response time achieved by the various policies

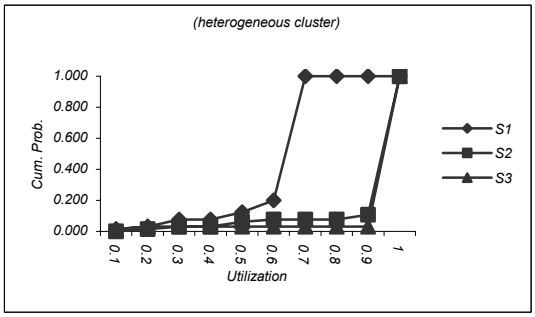

Figure 6: Probability of saturation of servers using RR

Acknowledgement

This work was supported by the Canadian Institute for Telecommunications Research under

the Networks of Centres of Excellence Program of the Government of Canada, and by the IBM Toronto Laboratory Centre for Advanced Studies. The authors would like to thank Radha Subrahmanyan for the implementation of the monitoring agent and the instrumentation of the Apache server on Windows platform, and Chun Bafu for the work on porting the code on a UNIX platform. The authors would like also to thank Richard Bell for his help in setting up the hardware for the experimentation.

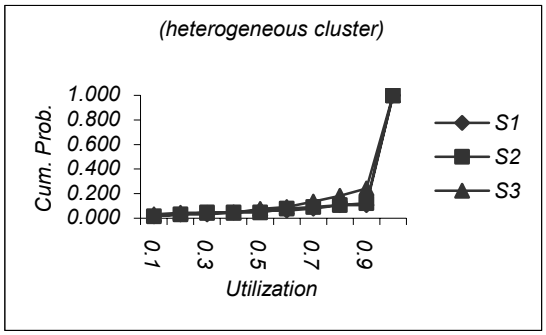

Figure 7: Probability of saturation of servers using LU

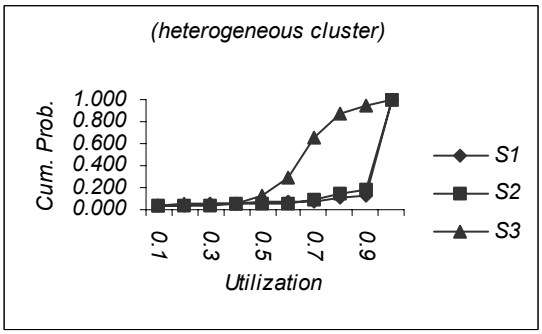

Figure 8: Probability of saturation of servers using LAS

References

[1] M. Salem, J.W. Wong, and G.v. Bochmann: "*A Scalable Load-Sharing Architecture for Distributed Applications*", Proc. SoftCom '2001, Split, Croatia.

[2] M. Colajanni, P.S. Yu, D.M. Dias, "Analysis of task assignment policies in scalable distributed Web-server systems", IEEE Trans. on Parallel and Distributed Systems, vol. 9, no. 6, 1998.

[3] R. Jain: "The Art of Computer Systems Performance Analysis", John Wiley & Sons, 1991.

[4] P. Barford and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation", Proc. ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 151-160, July 1998.

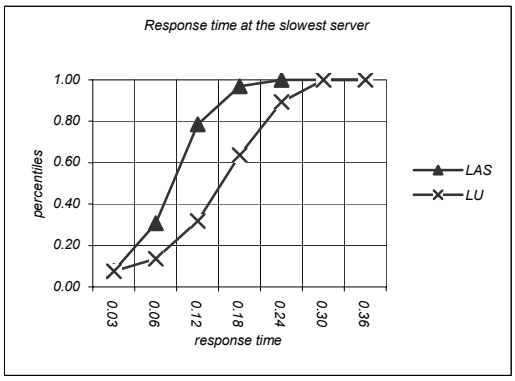

Figure 9: Response time at the slowest server in heterogeneous cluster

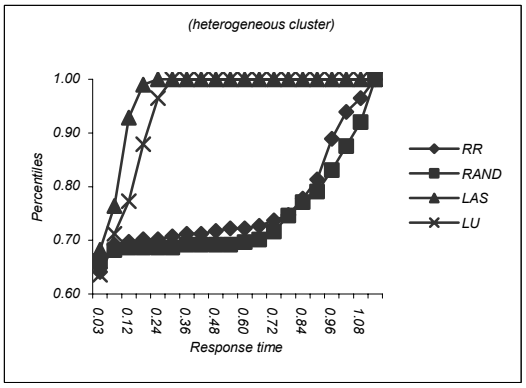

Figure 10: Response time achieved by the different policies