

Distributed Resource Monitors for Mobile Objects*

M.Ranganathan, Anurag Acharya and Joel Saltz
Department of Computer Science
University of Maryland, College Park 20742
{ranga, acha, saltz}@cs.umd.edu

Abstract

We present our position on resource monitoring as three working hypotheses. First, a resource-aware placement of components of a distributed application can provide significant performance gains over a resource-oblivious placement. Second, effective mobility decisions can be based on coarse-grained monitoring. Finally, a simple and cheap distributed resource monitoring scheme can provide sufficient information for effective mobility decisions. We present a design for distributed resource monitors which we believe can provide effective resource information at an acceptable cost.

1. Introduction

Mobile programs can move an active thread of control from one site to another during execution. This flexibility has many potential advantages in a distributed environment. For example, a program that searches distributed data repositories can improve its performance by migrating to the repositories and performing the search on-site instead of fetching all the data to its current location. Similarly, an Internet video-conferencing application can minimize overall response time by positioning its server based on the location of its users. Other scenarios where mobile programs can be useful may be found in workflow management and wireless computing. The primary advantage of mobility in these scenarios is that it can be used as a tool to adapt to variations in the operating environment. To be able to utilize mobility in this manner, programs need to be able to obtain online information about their operating environment. Applications can use this information and knowledge of their own resource requirements to make judicious decisions about migration.

Before we present our position on resource monitoring for mobile applications, we would like to argue the need for mobility as an adaptation mechanism. An alternative

*This research was supported by ARPA under contract #F19628-94-C-0057, Syracuse subcontract #353-1427

adaptation mechanism, which places replicated servers at all suitable points in the network, could adapt to variations in resources and user distribution by coordination between servers and by using dynamically created server hierarchies. It is quite likely that for any particular application, such a strategy would be able to achieve the performance achieved by programs that use mobility as the adaptation tool. The advantage of mobility-based strategies is that it allows small groups of users to rapidly set up private communities on-demand without requiring extensive server placement. With online information about resource availability and quality, mobility-based strategies can automatically determine suitable sites for locating data-structures and computations that govern the performance of the application.

We present our position on resource monitoring as three working hypotheses. First, a resource-aware placement of components of a distributed application can provide significant performance gains over a resource-oblivious placement. Second, effective mobility decisions can be based on coarse-grained monitoring. Finally, a simple and cheap distributed resource monitoring scheme can provide sufficient information for effective mobility decisions. We present a design for distributed resource monitors which we believe can provide effective resource information at an acceptable cost.

The paper is organized as follows. First, we present results from a recent investigation on *network-aware* mobile programs, that is, programs that can use mobility as a tool to adapt to variations in network characteristics [6]. In this investigation, we studied how an object-oriented Internet chat server can take advantage of mobility to adapt to variations in network latency as well as variations in the distribution of clients. Next, we present our design for distributed resource monitors. Based on this design, we have implemented *Komodo*, a distributed network latency monitor. The chat server mentioned above uses latency information provided by *Komodo*. We found that *continuous* monitoring of UDP-level network latency between sixteen hosts consumed about 0.5% of CPU cycles and generated 256 bytes/second of network traffic. We also found that latency-sensitive ap-

plications that run for at most an hour or so gain little from continuous monitoring and that most of the gains for these applications can be achieved by *on-demand* determination of resource levels. On-demand monitoring would also be suitable for resources that are more expensive to monitor, for example, network bandwidth.

2. Network-aware dynamic placement of objects: a case study

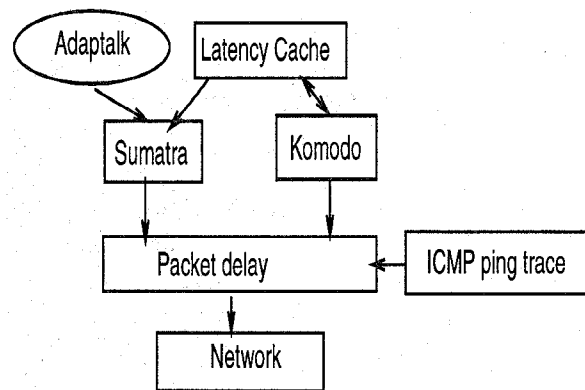
We studied network-aware dynamic placement in the context of an object-oriented Internet chat server. This application, called `adaptalk`, implements the chat server as a mobile object. It monitors the latencies between all participants and locates the *chat-board* object so as to minimize the maximum response time. `Adaptalk` was implemented in *Sumatra* [6], an extension of the *Java*¹ programming environment [2] that provides support for adaptive mobile programs, including object migration, thread migration and tracking of mobile objects. We selected this application since it is highly interactive and requires fine-grain communication. If such an application is able to take advantage of information about network characteristics, we expect that many other distributed applications over the Internet would be similarly successful. The resource that governs the migration decisions of `adaptalk` is network latency. To provide latency information, we have developed *Komodo*, a distributed network latency monitor (see section 3.1).

We studied the performance of `adaptalk` for three kinds of network variations: (1) *population variations*, which represent changes in the distribution of users on the network, as sites join or leave an ongoing conversation; (2) *spatial variations*, i.e. stable differences between in the quality of different links, which are primarily due the host's connectivity to the Internet; and (3) *temporal variations*, i.e. changes in the quality of a link over a period of time, which are presumably caused by changes in cross-traffic patterns and end-point loads. Spatial variations can be handled by a *one-time placement* based on the information available at the beginning of a run. Adapting to temporal and population variations requires *dynamic placement* which needs a periodic cost-benefit analysis of current and alternative placements of computation and objects.

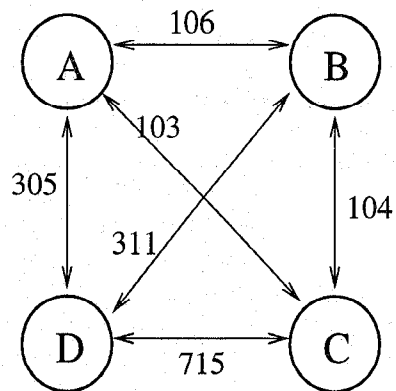
We performed all our experiments on four Solaris machines on our LAN. To simulate the characteristics of long-haul networks, we used Internet ICMP ping traces to delay all packets. This approach also allowed us to perform repeatable experiments.

We collected latency traces for 45 hosts on the Internet: 15 popular .com web-sites (US), 15 popular .edu web sites (US) and 15 well-known hosts around the world.

¹Java is a registered trademark of Sun Microsystems



(a) Organization on each host



(b) Avg. Latency (in ms) between hosts

Figure 1. Experimental Setup. Four local machines on a LAN were used to simulate four remote machines on the Internet by adding delays to packets. ICMP ping traces between real Internet hosts were used to generate the delays, so as to capture real-life temporal variations in latencies.

These hosts were pinged from four different locations in the US. The study was conducted over several weekdays, each host-pair being monitored for at least 48 hours. We used the commonly available ping program and sent one ping per second. We have a total of over 150 48-hour traces from which we picked six trace-segments over the noon-2pm EDT period for our experiments. Noon is the beginning of the daily latency peak for US networks and the end of the daily latency peak for many non-US networks. Hosts participating in the selected traces include: `java.sun.com`, `home.netscape.com`, `www.opentext.com`, `cesdis.gsfc.nasa.gov`, `www.monash.edu.au` and `www.ac.il`. This setup makes the four local machines behave like four far-flung machines on the Internet. Figure 1 shows the experimental setup used for the experiments.

To evaluate the effect of changing user distribution we

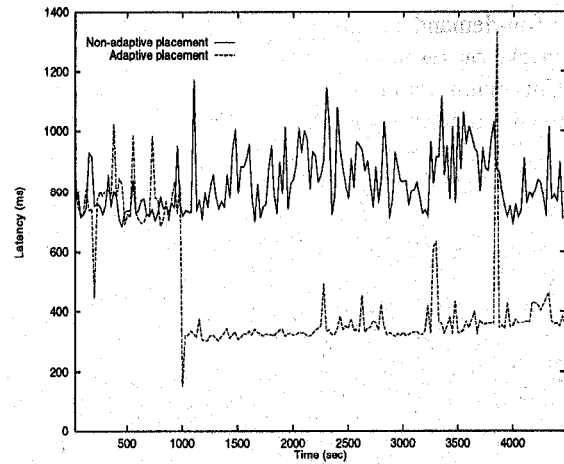
used the following workload: hosts C and D initiate a conversation; host B joins after 15 minutes, and host A joins after another 15 minutes. Each host sends a sequence of 70-character sentences with a 5-second think time between sentences. To evaluate the effect of spatial and temporal variations, the workload consisted of all 4 hosts jointly initiating a conversation which runs for 75 minutes. As before, each host generates a new sentence every 5 seconds.

We found that for conversations lasting up to one hour or so, adapting to spatial variations and variations in the number and location of the clients achieves most of the gains. Figure 2 (a) plots the maximum latency over all hosts for the population variation experiment. It shows that an adaptive placement policy rapidly adapts to the changing population workload. Soon after host B joins the conversation, the *chat-board* object moves there, causing a drop in the maximum latency. In contrast, the maximum latency remains high throughout the conversation for a non-adaptive policy. Figure 2 (b) compares the maximum latency (over all participants) for an adaptive placement policy that moves the *chat-board* in response to online latency variations and a non-adaptive placement policy that places the *chat-board* object based on network information available at the beginning of the conversation and keeps it there for the entire conversation. To handicap the adaptive policy, its experiment is started with the *chat-board* in the worst-possible location. The graph in Figure 2 (b) shows that once the adaptive version moves the server to a more suitable location, the performance of the two versions is largely equivalent. This implies that adapting to short-term temporal variations in a steady population workload does not provide much performance advantage over one-shot network-aware placement. It may, however, still be advantageous to adapt to long-term temporal variations. We note that at the far right of Figure 2 (b), temporal variation in the link latencies do allow the adaptive placement policy to do better.

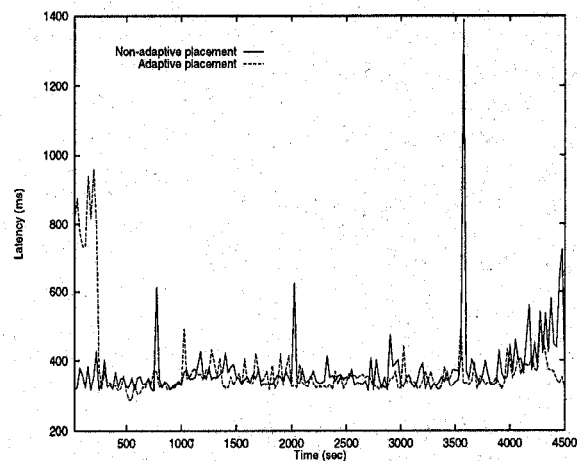
3. Design of a distributed resource monitor

For different applications, different resource constraints are likely to govern the decision to migrate - for example network latency, network bandwidth, server load (as in number of server connections available), CPU cycles etc. We propose a single user-level monitor for all resources. Using a single monitor facilitates applications that might need information about multiple resources. It also reduces communication requirements for distributed monitoring as information about multiple resources can be sent in the same message. Based on our hypothesis that coarse-grained monitoring is adequate for mobility decisions, we believe that a portable user-level monitor can provide acceptable performance.

In our design, each host runs a monitor daemon which



(a) Adapting to population variation.



(b) Adapting to spatial and temporal variation

Figure 2. Adaptation to population, spatial and temporal variations. The graphs plot maximum latency over all participants vs time.

communicates with peers on other hosts. The monitoring daemons are loosely-coupled and use UDP for communication as well as for monitoring the network. A simple timeout-based scheme is used to handle lost packets and re-transmissions. Using UDP may result in packet loss leading to temporary lack of accuracy. We do not expect this to be a problem as we expect movement decisions to be based on fairly coarse grained observations of system resources.

Applications register monitoring requests with the local daemon. If the resource mentioned in the request can be monitored from the current host then the local daemon handles the request. Requests that cannot be handled locally, for example, network latency between two remote sites, are forwarded by the local daemon to the daemon on the appropriate host.

Applications can request the current availability of a re-

source (on-demand monitoring) or they can request periodic checks on resource availability (continuous monitoring). Continuous monitoring applies a application-specified filter to eliminate jitter in resource levels. Eliminating jitter in an application-specific manner helps reduce the communication requirements without impacting application performance. Data corresponding to remote requests for continuous monitoring is forwarded to the requesting sites as and when the filtered value of the resource changes. Requests for continuous monitoring may also specify a sampling frequency, subject to an upper bound.

Each daemon supports a limited number of monitoring requests. This limit applies to both local and remote requests. Together with the limit on sampling frequency, this ensures the monitoring load on individual hosts is within acceptable limits.

Each request has an application-specified *time-to-live*. There is an upper bound on the *time-to-live* which allows the daemons to clean-up requests made by applications or hosts that have since crashed. Applications need to refresh requests within the *time-to-live*. Requests that are not refreshed are dropped. If a daemon runs out of table space, the least recently requested entry is ejected.

The interface for registering the monitoring requests and accessing the resulting information resembles the familiar `ioctl()` interface in Unix.

Monitoring requests are passed from applications to the local daemon using a well-known Unix domain socket. The resource information is made available by the daemon in a read-only shared memory segment. This allows applications to rapidly access the latest available monitoring information.

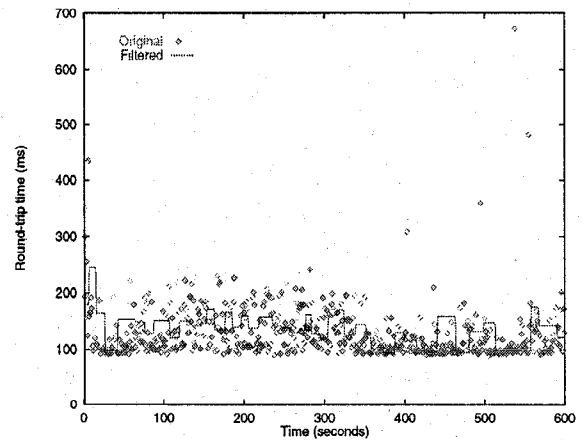
3.1 A distributed network latency monitor

Based on the design described above, we have developed *Komodo*, a distributed network latency monitor. We plan to extend *Komodo*, in the immediate future, to monitor network bandwidth, CPU cycles and server load (number of available server connections).

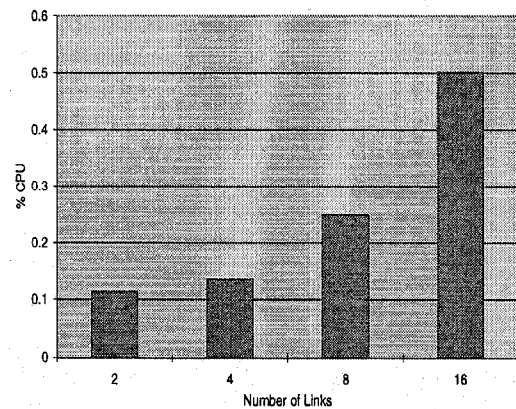
Each *Komodo* daemon monitors the UDP-level latency on a network link for which it has received monitoring requests, by sending a 32-byte UDP packet to the daemon on the other end of the link of interest. If an echo is not received within an expected interval, (the maximum of the ping period or five times the current round trip time estimate) the packet is retransmitted.

To eliminate short-term variation in latency measures, we developed a filter based of our study of a large number of Internet ICMP latency traces (see [6]). This study revealed that: (1) there is a lot of short-term jitter in the latency measures but in most cases, the jitter is small; (2) there are occasional jumps in latency that appear only for a single ping; and (3) for some traces, the latency measure fluctuates

rapidly. Accordingly, the *Komodo* filter eliminates singleton impulses as well as noise within a jitter threshold. If the measure changes rapidly, a moving window average is generated. Figure 3(a) illustrates the operation of this filter.



(a) Operation of the Komodo filter



(b) CPU utilization of Komodo

Figure 3. (a) The input to the filter is a 10-minute trace of one-per-second latency measures between `baekdo.cs.umd.edu` and `lanl.gov`. The jitter limit was 10 ms. Note that the four single-ping impulses towards the right end have been eliminated. (b) The CPU utilization is computed by dividing the (user+system) time by the wallclock time. Each experiment was run for 1000 seconds with one ping per two seconds for all links.

To quantify the cost of monitoring, we measured the CPU utilization of *Komodo* for varying number of links. Results in Figure 3 (b) show that the maximum CPU utilization for up to sixteen links is about 0.5 %. The amount of data

transferred is 256 bytes/second. Also, up to sixteen links, the CPU utilization scales linearly.

4. Discussion

There is a fundamental choice in the design of a resource monitoring interface – whether applications are required to poll or whether they are notified asynchronously. This is, of course, a common dilemma for designers of system support for parallel and distributed systems. Which of these alternatives is preferable depends on: (1) the relative costs of polling and notification, (2) the frequency at which applications need the information provided by the underlying system, and (3) whether the information corresponds to resources *managed* by the underlying system.

Another important choice is the granularity at which resource change is to be monitored. The simplest alternative is to track every change. This is impractical as most resource levels have some jitter which usually has little impact on application performance. The next simplest alternative is to use a jitter threshold and track only those changes that larger than this threshold. Jitter-threshold-based schemes work well if changes in the resource levels are usually stable. Transient changes (usually just spikes) in the resource levels can cause spurious responses. The alternative is to augment the jitter-based scheme with a filter that eliminates transients. This allows the applications to track only the stable changes.

Noble, Price and Satyanarayanan [5] as well as Badrinath and Welling [1] propose notification-based schemes for tracking resource changes for mobile hosts. Both propose an interface that allows applications to specify a jitter threshold.

Our experience with *adaptalk* and Internet latency traces has indicated that coarse-grained monitoring is adequate for effective monitoring decisions for latency-sensitive mobile applications on the Internet. Whether this is true for applications whose performance is sensitive to other resources remains to be seen. Note that if this is indeed true, the issue of cost becomes less important. We did find, however, that polling can be implemented very cheaply (accessing a shared memory segment) compared to notification (signals and signal handlers). A polling-like approach has also been used by Mummert *et al.* [4] in the design of mechanisms to exploit weak connectivity for mobile file access in the *Coda* file system. Individual components of the *Coda* system support cooperate in monitoring the bandwidth and maintain the information in a shared location. In their system, however, the monitoring information is used to adapt the system support and is not exported to the applications.

There are two situations in which a notification-based approach may be preferable to a polling-based approach. First, if the system support not only monitors the resource levels

but also allocates and revokes resources [7]. For example, if the underlying system supports bandwidth allocation, it may need to revoke a previous allocation to accommodate later requests or to accommodate multiple existing requests within a smaller bandwidth. Second, if the platform is mobile and may be able to switch between multiple wireless networks [3]. In such scenarios, rapid reaction is likely to be more important than efficiency of communication.

5 Conclusions and Future Work

Our experiments with a simple latency-sensitive application has shown that resource-aware adaptation combined with program and object mobility can be an important strategy for dealing with variations in resource quality in an Internet environment.

In the immediate future, we intend to extend the implementation of Komodo, our prototype distributed resource monitor, to handle network bandwidth, CPU cycles, server connections and other resource and to experiment with applications that can exploit this information. Applications that we are considering include multi-database queries over the Internet, sequence servers and resource-aware pre-fetching for web clients and mobile applications that run on mobile hosts. We believe that there is a class of long running applications over the Internet for which resource-aware mobility could provide flexibility and performance which would take a lot more effort to achieve by other means. In our research, we plan to study such applications and understand their structure and requirements.

References

- [1] B.R.Badrinath and G. Welling. Event delivery abstractions for mobile computing. Technical Report LCSR-TR-242, Rutgers University, 1995.
- [2] J. Gosling and H. McGilton. The Java language environment white paper. Technical report, Sun Microsystems, 1995.
- [3] R. Katz. The case for wireless overlay networks. Invited talk at the ACM Federated Computer Science Research Conferences, Philadelphia, 1996.
- [4] L.B.Mummert, M.R.Ebling, and M.Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the 15th. A.C.M Symposium on Operating Systems Principles*, Dec. 1995.
- [5] B. D. Noble, M. Price, and M.Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. *Proceedings of the Second USENIX Symposium on Mobile and Location Independent Computing*, Feb. 1995.
- [6] M. Ranganathan, A. Acharya, S. Sharma, and J. Saltz. Network-aware mobile programs. In *Proceedings of USENIX'97*. To appear.
- [7] M. Satyanarayanan. Fundamental challenges in mobile computing. Invited Lecture at the Fourteenth ACM Symposium on Principles of Distributed Computing, 1995.