The following paper was originally published in the
Proceedings of the USENIX 1997 Annual Technical Conference
Anaheim, California, January 6-10, 1997

Network-aware Mobile Programs

M.Ranganathan, Anurag Acharya, Shamik D. Sharma, and Joel Saltz
Department of Computer Science
University of Maryland

Network-aware Mobile Programs*

M.Ranganathan, Anurag Acharya, Shamik D. Sharma and Joel Saltz
Department of Computer Science
University of Maryland
College Park, MD 20740

Abstract

In this paper, we investigate network-aware *mobile programs, programs that can use mobility as a tool to adapt to variations in network characteristics. We present infrastructural support for mobility and network monitoring and show how* `adaptalk`*, a Java-based mobile Internet chat application, can take advantage of this support to dynamically place the chat server so as to minimize response time. Our conclusion was that on-line network monitoring and adaptive placement of shared data-structures can significantly improve performance of distributed applications on the Internet.*

1 Introduction

Mobile programs can move an active thread of control from one site to another during execution. This flexibility has many potential advantages. 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. Applications running on mobile platforms can react to a drop in network bandwidth by moving network-intensive computations to a proxy host on the static network. 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. Applications can use online information about their operating environment and knowledge of their own resource requirements to make judicious decisions about placement of computation and data.

For different applications, different resource constraints are likely to govern the decision to migrate, e.g. network latency, network bandwidth, memory availability, server availability. In this paper, we investigate *network-aware* mobile programs, i.e. programs that position themselves based on their knowledge of network characteristics. Whether the potential performance benefits of network-aware mobility are realized in practice depend on answers to three questions. First, how should programs be structured to utilize mobility to adapt to variations in network characteristics? In particular, what policies are suitable for making mobility decisions? Second, is the variation in network characteristics such that adapting to them can be profitable? Finally, can adequate network information be provided to mobile applications at an acceptable cost?

In order to adapt to network variations, mobile programs must be able to decide when to move, what to move and where to move. There are three types of network variations which may be cause for migration: (1) *population* variations, which represent changes in the distribution of users on the network, as sites join or leave an ongoing distributed computation; (2) *spatial* variations, i.e. stable differences between in the quality of different links, which are primarily due to the hosts' connectivity to the Internet; and (3) *temporal* variations, i.e. changes in the quality of a link over a period of time, caused presumably by changes in cross-traffic patterns and end-point load. 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 data. Dynamic placement decisions have two partially conflicting goals: maximize the performance improvement from mobility and minimize the cost of mobility. If an opportunity for improving performance presents itself, it should be capitalized upon; however, reacting too rapidly to changes in the network characteristics can lead to performance degradation as the performance gain may not offset the mobility cost.

We investigate these issues in the context of *Sumatra*, an extension of the *Java*[1] programming environment [10]

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

[1] *Java* is a registered trademark of Sun Microsystems.

that provides a flexible substrate for adaptive mobile programs. Since mobile programs are scarce, we developed a mobile chat server for our experiments. This application, called `adaptalk`, monitors the latencies between all participants and locates the chat server so as to minimize the maximum response time. 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.

To evaluate if mobile applications can take advantage of network-awareness, we examined the performance of `adaptalk` with and without mobility. Our evaluation had two main goals: (1) to determine the performance benefits, if any, of network-aware placement of the central chat server over a network-oblivious placement; and (2) to determine if dynamic placement based on online network monitoring provides significant performance gains over a one-time placement based on initial information. Our results are encouraging - they indicate that on-line monitoring and dynamic placement can significantly improve performance of distributed applications on the Internet.

The paper is organized as follows. Section 2 describes Sumatra and the programming model that it provides. Section 3 describes the design and implementation of Komodo. Section 4 describes the `adaptalk` application and the policy it uses to make mobility decisions. Section 5 describes our experiments and presents the results. Section 6 discusses the results and their implications. Section 7 describes related work and Section 8 provides our conclusions and plans for future work.

2 Sumatra: a Java that walks

Sumatra is an extension of the Java programming environment that supports adaptive mobile programs. Platform-independence was the primary rationale for choosing Java as the base for our effort. In the design of Sumatra, we have not altered the Java language. Sumatra can run all legal Java programs without modification. All added functionality was provided by extending the Java class library and by modifying the Java interpreter without affecting the virtual machine interface.

Our design philosophy for Sumatra was to provide the mechanisms to build adaptive mobile programs. Policy decisions concerning when, where and what to move are left to the application. The main feature that distinguishes Sumatra from previous systems [3, 11, 13, 23] that support mobile programs is that *all* communication and migration happens under application control. Furthermore, combination of distributed objects and thread migration allows applications the flexibility to dynamically choose between moving data or moving computation. The high degree of application control allows us to easily explore different policy alternatives for resource monitoring and for adapting to variations in resources. We believe that the space of design choices for adaptive mobile programs is yet to be mapped out and such flexibility is important to help explore this space.

Sumatra adds two programming abstractions to Java: *object-groups* and *execution engines*. An object-group is a dynamically created group of objects. Objects can be added to or removed from object-groups. All objects within an object-group are treated as a unit for mobility-related operations. This allows the programmer to customize the granularity of movement and to amortize the cost of moving and tracking individual objects. This is particularly important in languages like Java because every data structure is an object and moving the state one object at a time can be prohibitively expensive. An execution-engine is the abstraction of a location in a distributed environment. In concrete terms, it corresponds to an interpreter executing on a host. Sumatra allows object-groups to be moved between execution-engines. An execution-engine may also host active threads of control. Currently, multiple threads on the same engine are scheduled in a *run-to-completion* manner. We plan to implement other scheduling strategies in future. Threads can move between engines.

The principal new operations provided by Sumatra are:

Object-group migration: Object-groups can be moved between engines on application request. As mentioned earlier, all objects within an object-group are treated as a unit for mobility-related operations. Objects in an object-group are automatically marshalled using type-information stored in their class templates. When an object-group is moved, all local references to objects in the group (stack references and references from other objects) are converted into *proxy references* which record the new location of the object. Some objects, such as I/O objects, are tightly bound to local resources and cannot be moved. References to such objects are reset and must be reinitialized at the new site. The class template for an object (and the associated bytecode) can be downloaded into an execution-engine on application request.

Remote method invocation: Method invocations on proxy objects are translated into calls at the remote site. Type information stored in class-templates is used to achieve RPC functionality without a stub compiler. Exceptions generated at the called site are forwarded to the caller. Sumatra does not automatically track mobile objects. Requesting a remote method invocation on an object that is no longer at the called site results in an *object-moved* exception at the calling site. To facilitate application-level tracking, the exception carries with it a forwarding address. The caller can handle the exception as it deems fit (e.g., reissue the request to the new location, migrate to the new location, raise a further exception and so on). This mechanism allows applications to locate mobile objects lazily, paying the cost of tracking only if they need to. It also allows applications to abort tracking if need be and pursue an alternative course of action.

Thread migration: Sumatra allows explicit thread migration using a `engine.go()` function that bundles up the stack and the program counter and moves the thread to the specified execution-engine. Execution is resumed at the first instruction after the call to `go`. To automatically marshal the stack, the Sumatra interpreter maintains a type stack parallel to the value stack, which keeps track of the types of all values on the stack. When a thread migrates, Sumatra transports with it all local objects that are referenced by the stack but do not belong to any object-group. Objects that belong to an object-group move only when that object-group is moved. Stack references to the objects that are left behind (i.e were part of some object-group) are converted to proxy references. After the thread is moved to the target site, it is possible that its stack contains proxy references that point to objects that used to be remote but are now local. These references are converted back to local references before the call to `go` returns.

Remote execution: A new thread of control can be created by *rexec*'ing the `main` method of a class existing on a remote engine. The arguments for new thread are copied and moved to the remote site. Unlike remote method invocation, remote execution is non-blocking; the calling thread resumes immediately after the `main` method call is sent to the remote engine. Remote execution is different from thread migration as it creates a new thread at the remote site that runs concurrently with the original thread; thread migration moves the current thread to the remote site without creating a new thread. Concurrent threads communicate using calls to shared objects. The thread initiating a remote execution can share objects with the new thread by passing it references to these objects as arguments to `main`.

Resource monitoring: Sumatra provides a resource-monitoring interface which can be used by applications to register monitoring requests and to determine current values of specific resources. This interface is similar to an object-oriented version of the Unix `ioctl()` interface. When an application makes a monitoring request, Sumatra forwards the request to the local resource monitor. If the monitor does not support the requested operation, an exception is delivered to the application.

Signal handlers: Sumatra allows applications to register handlers for a subset of Unix signals. Signals can be used by the external environment (the operating system or some other administrative process) to inform the application about urgent asynchronous events, in particular resource revocation. Using a handler, the application can take appropriate action including moving away from the current execution site.

2.1 Example

In this section, we provide a feel for the Sumatra programming model using a simple example. The task is to scan through a database of X-ray images stored at a remote site for images that show lung cancer. This task can be performed in two steps. In the first step, a computationally cheap pruning algorithm is used to quickly identify lungs that might have cancer. A compute-intensive cancer-detection algorithm is then used to identify images that actually show cancer.

One way to write a program for this task would be to download all lung images from the image server and do all the processing locally. If the absence of cancer in most lung images can be cheaply established, this scheme wastes network resources as it moves all lung images to the destination site. Another approach would be to send the selection procedure to the site of the image database and to send only the "interesting" images back to the main program. If the selection procedure is able to filter out most of the images, this approach would significantly reduce network requirements. A third, and even more flexible, approach would allow the shipped selection procedure to extract all the interesting images from the database but return only the *size* of the extracted images to the main program. If the size is too big, the program may choose to move itself to the database site and perform the cancer-detection computation there rather than downloading all the data. This avoids downloading most images at the cost of (possibly) slower processing at the server. On the other hand if the size of the images is small, the data can be shipped over and processed locally. Figure 1 shows code for the third approach. This program makes its decision to migrate in a rudimentary fashion; a more realistic version of this application would also take network bandwidth and the processing power available on both machines into consideration.

Sumatra assumes that a local resource monitor is available which can be queried for information about the environment. In the next section, we describe one such monitor which allows `Sumatra` applications to request information about network latency between any pair of sites that

```
.....
filter_object = new Lung_filter();
cancer_object = new Lung_checker(filter_object);
myengine = System.rpc.myEngine();

// Create a engine at the xray database site.
remote_engine = new Engine("xrays.gov");
// Send the lung_filter class to the remote engine
remote_engine.downloadClass("Lung_filter");
// Create a new object group.
objgroup = new ObjGroup("lung_filter_group");
// Add the lung_filter_object to the object group
objgroup.checkIn(filter_object);
// Move the object group to the database site
objgroup.moveTo(remote_engine);

// a remote method call selects interesting xrays
size = filter_object.query(db,"DarkLungs");

// Are there too many images to bring over?
if ( size > too_many_images ) {
  // Migrate thread, process images and return.
  remote_engine.go();
  result = cancer_object.detect_cancer();
  myengine.go();
}
else {
  // there are only a few interesting xrays. Fetch them
  // and process locally.
  objgroup.moveTo(myengine);
  result = cancer_object.detect_cancer();
}

// display result locally
System.display(result);
```

Figure 1: Excerpt of a Sumatra program that adaptively migrates to reduce its network bandwidth requirements

run the monitor.

3 Komodo: a distributed network latency monitor

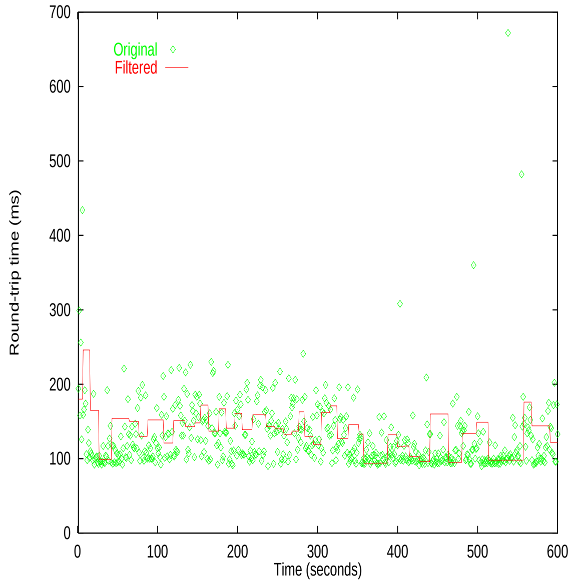
Komodo[2] is a distributed network latency monitor. The design principles of Komodo are: low-cost active monitoring and fault-tolerance. Active monitoring uses separate messages for monitoring; passive monitoring generates no new messages and piggybacks monitoring information on existing messages. An active monitoring approach is needed for `adaptalk` (described in the next section) as passive monitoring cannot provide information about links that are not used in the current placement but could be used in alternative placements. It is our working hypothesis that effective mobility decisions can be based on medium-term (30sec-few minutes) and long-term (hours) variations. At these resolutions, we believe that active monitoring can be achieved at an acceptable cost. This section briefly describes the design and implementation of Komodo. Further details about Komodo are presented in [18].

Komodo allows applications to initiate monitoring of network latency between any pair of hosts running the monitor; the application need not be resident on either of the hosts. Komodo is implemented as a user-level daemon that runs on every host participating in the computation. Applications pass monitoring requests to their local Komodo daemon. If the requested link includes the current host, the local daemon handles the request. Otherwise, it forwards the request to the daemon on the appropriate host. Daemons determine network latency by sending 32-byte UDP packets to each other. 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. Using UDP for communication may, occasionally, lead to loss of messages. Message loss can lead only to a short-term loss of efficiency. As we expect monitoring requirements to be coarse-grained, the effect of packet loss should be small. Note that message loss is also a sign of network congestion and as such may be useful information for applications.

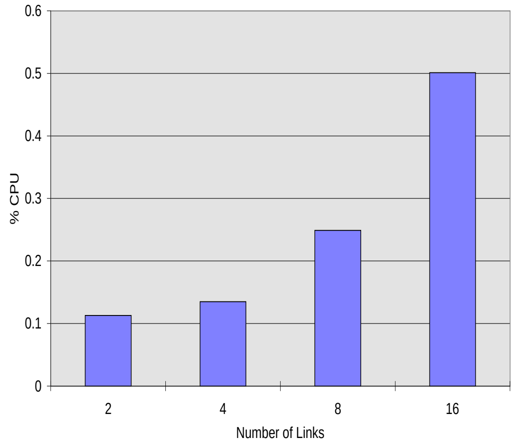
Applications that initiate a monitoring request can specify the frequency with which Komodo *pings* a link. Komodo enforces an upper bound on this frequency to keep the monitoring cost at an acceptable level. Applications need to refresh requests periodically to keep them alive; Komodo deactivates requests that have not been refreshed for longer than its *request-timeout* period.

Latency measures acquired by Komodo are passed through a filter before being provided to applications. This filter eliminates singleton impulses as well as noise within a jitter threshold (we use a jitter threshold of 10 ms, which is the resolution of most Unix timers). If the measure changes rapidly, a moving window average is generated. This filter was designed on the basis of our study of a large number of Internet latency traces (see Section 5.1) which 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 sharp jumps in latency that appear only for short time intervals; (3) occasionally, the latency measure

[2] Komodo dragons are a species of *monitor* lizards found on the island of Komodo which is close to both Java and Sumatra.



(a) Operation of the Komodo filter



(b) CPU utilization of Komodo

Figure 2: (a) The input to the filter is a 10-minute trace of one-per-second latency measures between `baekdoo.cs.umd.edu` and `lanl.gov`. 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 total running time. Each experiment was run for 1000 seconds with one ping per second for all links.

fluctuates rapidly; (4) for time windows of 10 seconds or larger, the mode value (with a 10 ms jitter threshold) dominates. To elaborate the last point, in most time windows, 70-90% of the latency values fall within a jitter threshold of the most common value. Our filter attempts to find the mode for a recent time window. If there is no stable mode (as happens occasionally), it returns the mean. Figure 2(a) illustrates the operation of the filter.

Each daemon maintains a cache of current latency estimates for all links it is currently monitoring. This cache is maintained in a well-known shared memory segment and can be efficiently read by all Sumatra applications executing on the same machine. Cooperating Komodo daemons forward latency information in response to persistent remote requests. A latency estimate for a request received from another host is forwarded only when a new filtered estimate (different from the previous filtered estimate) is generated and is piggybacked onto a ping reply if possible. Currently, Komodo is implemented in C.

To address concerns about the cost of active monitoring, we measured the CPU utilization of Komodo for varying number of links. Results in Figure 2 (b) show that the maximum CPU utilization for up to sixteen links is about 0.5 %. The amount of data transferred is 512 bytes/second. This experiment was conducted on Sparc 5 machines (110MHz,32 MB of memory) running SunOS Release 5.5.

4 `Adaptalk`: An adaptive internet chat application

`Adaptalk` is a relatively simple network chat application built using Sumatra and Komodo. It allows multiple users to have an online conversation; new participants can join an ongoing conversation at any point; multiple independent conversations can be held. To ensure that all participants see the same conversation and that new participants can join ongoing conversations, a central server is used to serialize and broadcast the contributions.

`Adaptalk` is divided into three modules: handling keyboard events, managing the chat screen and coordinating the communication between participants. Each component is implemented by a separate object-group. Each host participating in the conversation runs two execution-engines, one houses the `screen` object-group and the other houses the `keyboard` object-group. The central server is implemented as a separate shared object-group, the `msgboard`, which can be placed on any host participating in the conversation. Each message issued by a participant starts from a keyboard object which invokes a remote method on the msgboard. The msgboard serializes incoming messages and issues a series of remote-execution requests, one per participant, which update the screen objects on all participants. In this case, remote execution is preferred to remote method invocation as there is no useful return value and remote execution allows fast one-way communication.

Individual messages in `adaptalk`, and most other chat applications, consist of single lines of characters, usually no more than 50-60 characters. The goal of a chat application is to provide a short response-time to all participants so that a conversation can make quick progress. The response-time for a particular participant depends on the latency between it and the central server. Given the latencies of all the links, the primary knob that `adaptalk` can turn to maintain a low response-time for all participants is the position of the central server.

4.1 Mobility policy

There are two main features of the `adaptalk` mobility policy. (1) continuous tracking of the instantaneously *most-suitable-site* and (2) deferral of server-motion till the potential for a significant and *stable* performance advantage has been seen. The first feature allows it to quickly take advantage of opportunities for optimization; the seconds helps ensure the gain is greater than the cost. The goal of `adaptalk` is to minimize the maximum response-time seen by any participant. The suitability of a participating machine as the location of the msgboard is characterized by the maximum network latency between it and all other participants. The machine that achieves the lowest measure is designated the most-suitable-site.

`Adaptalk`'s migration policy is shown in pseudo-code in Figure 3. This algorithm is run at the location that hosts the msgboard and recomputes the most-suitable-site each time a new message is posted by any participant. The msgboard maintains an array of counters, one for each potential location, which keep track of the number of times each location is found to be the most-suitable-site. The msgboard moves whenever: (1) the current site receives a very low score ($<$ loss_threshold) over a given period (the decision_cycle); or (2) a different site receives more than a threshold score (the win_threshold). The first condition is used to quickly move away from locations that provide poor performance; the second condition is used to move the msgboard to locations that consistently promise better performance. The counters are reset whenever the msgboard moves, the decision_cycle completes or a participant enters or leaves the conversation.

We expect three types of variations in the network characteristics which may be cause for migration: (1) *population* variations, which represent changes in the distribution of users on the network, as participants join or leave an ongoing conversation; (2) *spatial* variations, i.e. stable differences between latencies of different links; and (3) *temporal* variations, i.e. changes in the latency of a link over a period of time. `Adaptalk`'s migration policy can adapt to all three types of variations. Consider the case with a fixed number of participants with significant spatial variation in network latency and little temporal variation. In this case, the migration algorithm rapidly recognizes the best location for the msgboard, but waits until this choice has been ratified over some period of time (count[newloc] $>$ win_threshold) before moving it. As shown in Section 5, this policy allows `adaptalk` to effectively insure itself against poor *initial placement*. Once a good location has been found, the msgboard does not move, unless temporal variations or changes in population distribution cause another node to become a substantially better location (i.e. count[newloc] $>$ win_threshold) or the current host to become a substantially bad choice (i.e. count[curr_engine] $<$ loss_threshold && rounds % decision_cycle == 0). In such cases, the msgboard will move during the conversation. After initial experiments with `adaptalk`, we set the win_threshold to be $25 \times n$, the loss_threshold to be $12 \times n$ and the decision_cycle to be $50 \times n$. Here, n is the number of participants. The length of the decision_cycle was set large enough to amortize the cost of movement in cases where large temporal variations or fluctuations in population distribution cause frequent repositioning.

```
........
Get the all to all latency map from Komodo;
Find the site s that would minimize the max
  latency for messages posted to msgboard;
count[s] = count[s] + 1; rounds++;
let w be the site with the largest count;
let curr_engine be the engine which
   currently houses msgboard;
// Found a clear cut winner.
if (count[w] > win_threshold) return w;
else if (rounds % decision_cycle == 0) {
// Is the current engine an ok location ?
if (count[curr_engine] > loss_threshold) {
    clear count for each host;
    return curr_engine;
 } else {
    // Current engine is a bad location.
    set new_host to the host with the
         maximum count;
    clear count for each host;
    return new_host;
 }
} else return null; // cycle not yet over.
```

Figure 3: Decision Algorithm for *msgboard* placement used in Adaptalk. This algorithm is run at the location where the *msgboard* resides each time a message is posted.

5 Evaluation

To evaluate the performance impact of network-aware adaptation on the Internet, we performed two sets of experiments. First, we monitored round-trip times for 32-byte ICMP packets between a large set of host-pairs over several days. The goal of these experiments was to study the spatial and temporal variation in network latency on the Internet. Results from this study are presented in section 5.1.

Second, we measured the performance of three versions of `adaptalk` over long-haul networks, using traces collected during the Internet study. Our evaluation had two main goals: (1) to determine if network-aware placement of components of an application distributed over multiple hosts on the Internet provides significant performance gains over a network-oblivious placement; and (2) to determine if dynamic placement based on online network monitoring provides significant performance gains over a one-time placement based on initial information. Results from this study are presented in section 5.3.

5.1 Variations in Internet latency

We selected 45 hosts: 15 popular `.com` web-sites (US), 15 popular `.edu` web sites (US) and 15 well-known non-US hosts. These host 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. This resolution was acceptable as our goal was to discover medium-term (30sec/minutes) and long-term (hours) variations.

The conclusions of our study, briefly, are: (1) there is large spatial variation in Internet latency (the per-hour mean latency varied between 15 ms and 863 ms for US hosts and between 84 ms and 4000 ms for non-US hosts); (2) there is a large and stable variation in the latency of a single host-pair over the period of a day (maximum daily variation in per-hour mean latency for US hosts was 550 ms and for non-US hosts was 5750 ms); (3) There is a lot of jitter in the latency measures but in most cases, the jitter is small; (4) there are isolated peaks in latency that appear only for a single time interval; (5) for time windows of 10 seconds or larger, the mode value (with a 10 ms jitter threshold) dominates (in most time windows, 70-90% of the latency values fall within a jitter threshold of the most common value); (6) the moving-window mode changes quite slowly.

5.2 Experimental Setup

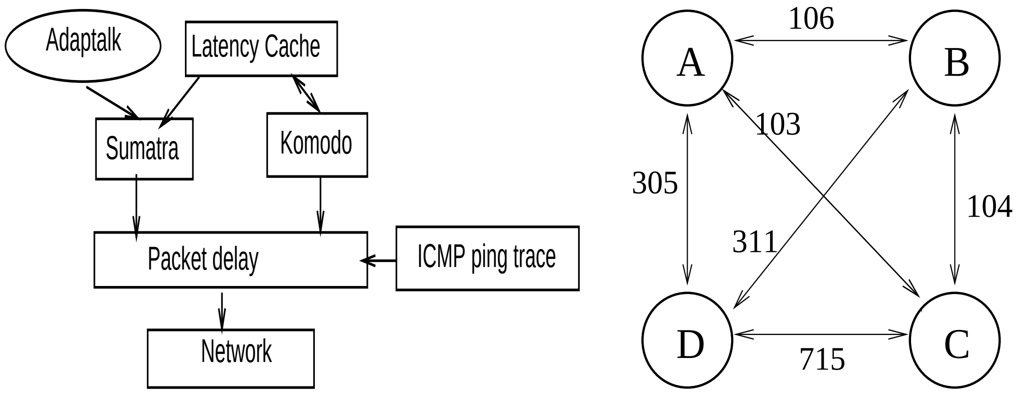
Having established that there are significant spatial and temporal variations in network latency on the Internet, we examined how well `adaptalk` could adapt to these variations.

To simulate the characteristics of long-haul networks, we decided to run our experiments over a low-latency LAN and delay all packets based on the ICMP `ping` traces described above (see Figure 4 (a)). This approach also allowed us to perform repeatable experiments. To ensure that delaying packets instead of using a real network does not skew the latency measures, we performed a simple test. Free-running Komodo monitors were installed at `bookworm.cs.umd.edu` and `jarlsberg.cs.wisc.edu` and were used to collect UDP latency measures between this host-pair. In parallel, a trace of ICMP ping times between these two hosts over the same period (5000 sec) was collected. This trace was later fed into trace-driven Komodo monitors running on two hosts on our LAN. The latency measures reported by the trace-driven monitors matched quite well with the actual latency measures reported by free-running monitors. The average of the actual latency measures was 128 ms (std dev = 64); the average of the values reported by the trace-driven monitors was 144 ms (std dev = 68).

We performed all our experiments on four Solaris machines on our LAN. We picked six trace-segments from the Internet study and used them to delay packets between the machines. All these segments were over the noon-2pm EDT period. We selected this period since noon is the approximate beginning of the daily latency peak for US networks as well as the approximate end of the daily latency peak for many non-US networks. These traces were selected to approximate the network latency spectrum observed in the Internet study. 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 4 (b) shows the configuration used for the experiments.

5.3 Experiments

We performed a series of experiments to evaluate the benefits of adapting to various types of network variations. The experiments consisted of running three different versions of the chat server. The first version, called static-placement, had no migration support and no network-awareness. The location of the msgboard was chosen in a network-oblivious fashion. The second version was a stripped-down version of *adaptalk*, called one-shot-placement. It used network information from Komodo to find the best initial placement for the msgboard, and used mobility support to move it there. After initial placement,



(a) Organization on each host (b) Avg. Latency (in ms) between hosts

Figure 4: 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.

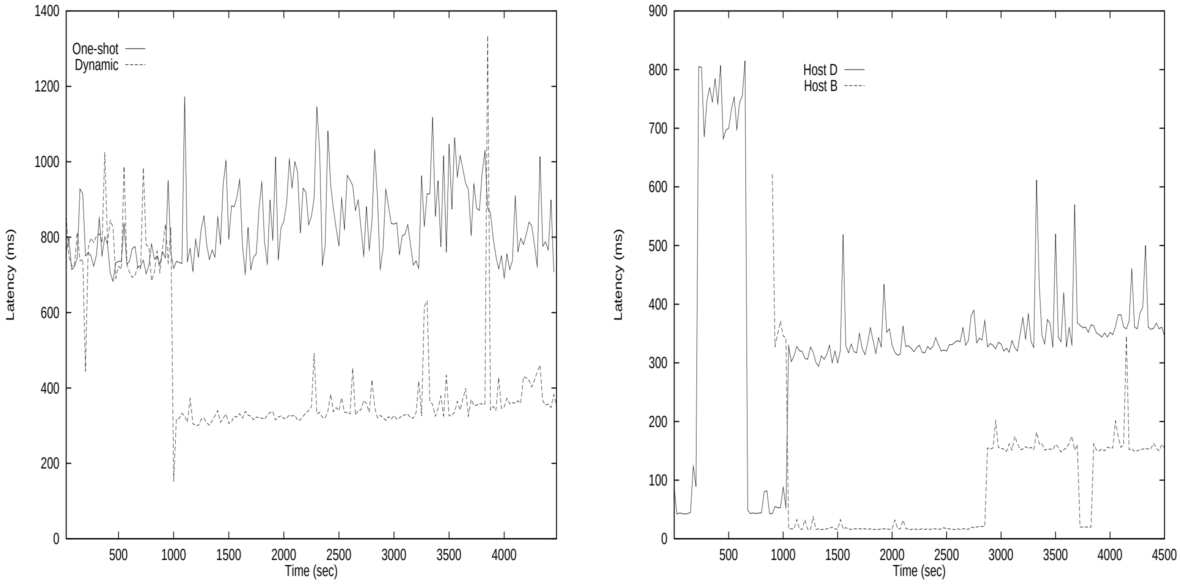
migration decisions and network-awareness were turned off. The third version, called dynamic-placement, was the full-fledged `adaptalk`, as described in section 4. It used on-line monitoring and dynamic placement to position the msgboard.

The performance of static-placement depends on the location of the msgboard. If static-placement chooses the same location as one-shot-placement, both would have the same performance. On the other hand, since static-placement is network-oblivious, it is just as likely to place the msgboard at the worst possible location. As the performance of one-shot-placement already provides a rough upper-bound on the performance of static-placement, we deliberately chose the worst initial placement when running the static-placement version.

Adapting to Population Variation: To evaluate the effect of changing user distribution we used the following workload: A conversation was initiated between hosts C and D. Host B joins the conversation after 15 minutes, and host A joins 15 minutes thereafter. Each host sends a sequence of 70-character sentences with a 5-second think time between sentences. With only two hosts initiating the conversation, there is no difference between the best and worst initial placements for the msgboard and both static-placement and one-shot-placement perform identically (both place the msgboard on host D). Figure 5 (a) plots the maximum latency over all hosts for the one-shot-placement version. Note that even after new hosts join the conversation there is no noticeable difference in maximum latency. continues to In contrast, dynamic-placement adapts to the changing population. Soon after host B joins the conversation, the adaptive placement policy moves the msgboard there, causing a drop in the maximum latency. After host A joins the conversation, the msgboard moves between hosts A and B in response to temporal fluctuations. This can be seen from the variation in latency for host B in Figure 5 (b). These movements help keep the maximum latency steady even in the presence of temporal fluctuations.

Adapting to Temporal and Spatial Variation: In this case the client population is assumed to be stable. The workload consists of all 4 hosts jointly initiating a conversation which runs for 75 minutes. As before, each host generates a new sentence every 5 seconds. In this case, the network-oblivious (static-placement) version places the chat server on host D. The network-aware (one-shot-placement) version uses latency information provided by Komodo to determine that host B is a much better placement. For the dynamic-placement version, initial placement is less important as it should be able to recover from a bad initial placement. For this version, we place the msgboard at host D, the worst-possible location.

To avoid clutter, Figure 6 shows the performance of these three versions in two different graphs. Figure 6 (a) compares the maximum latency (over all participants) for the dynamic-placement and static-placement versions. As seen from the sharp drop on the left end of the graph, the dynamic-placement version is successfully able to move the msgboard away from its bad initial placement to a more suitable location. Figure 6 (b) compares the average maximum latency (over all participants) for the dynamic-placement and one-shot-placement versions. It shows that once the dynamic-placement 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 work-



(a) one-shot vs. dynamic for population variation. Max latency (over all participants) vs time.

(b) Latency variations for hosts B and D Jumps signify movement of the server.

Figure 5: Adapting to population variation. Hosts C and D initiate the conversation. Host B joins after 900 seconds and host A joins 1800 seconds after the beginning. The one-shot-placement version places the chat server at host D. The dynamic-placement version migrates the server when new hosts join.

load 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. Note that at the far right of graph Figure 6 (b), temporal variation in the link latencies do allow the dynamic-placement version to do better than the one-shot-placement version.

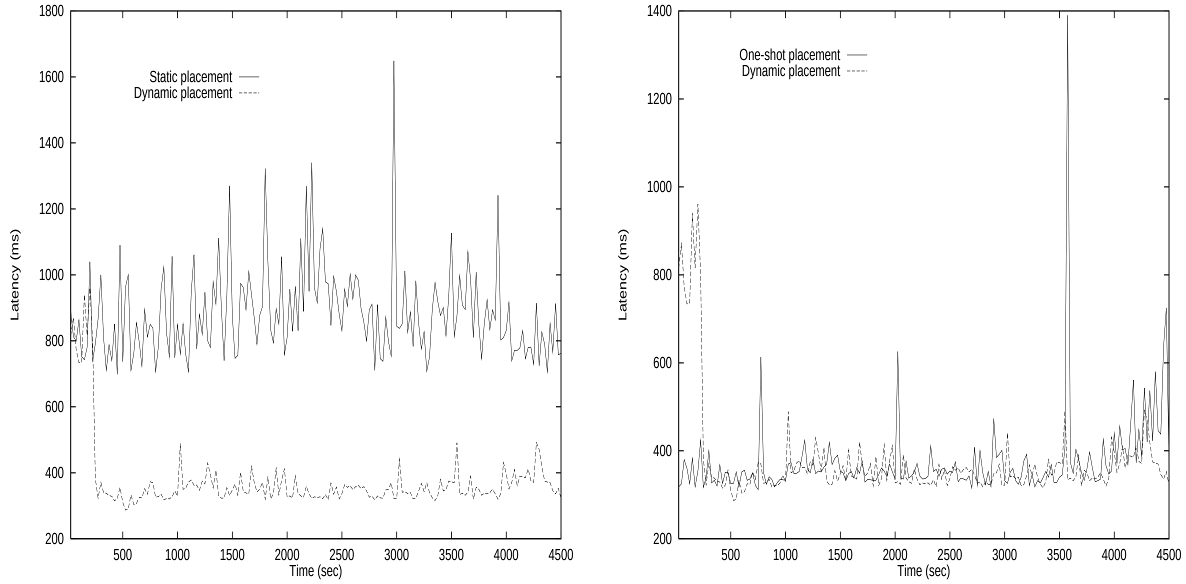
6 Discussion

In the introduction, we raised three questions with respect to network-aware mobility. First, how should programs be structured to utilize mobility to adapt to variations in network characteristics? Second, is the variation in network characteristics such that adapting to them proves profitable? Finally, can adequate network information be provided to mobile applications at an acceptable cost?

Our experience with Sumatra and `adaptalk` provides some early insights about application structure suitable for adaptive mobile programs. First, the migration policy should be cheap so that applications don't have to analyze the tradeoffs of the migration decision itself. An easy-to-compute policy allows frequent decisions and rapid adaptation to changes in the environment. We believe that an easy-to-compute migration policy was key to `adaptalk`'s ability to quickly find good locations for the chat server. Second, good modularization helps an application take advantage of mobility. Modularization is important for all distributed applications but it is more so for mobile programs as they have to make online decisions about the placements of different components. Third, to be resource-aware, remote accesses should be split-phase; the first phase delivers an *abbreviation*, a small and cheaply computed metric of the data (for example, size, number of data items, thumbnail sketch etc) and the second phase actually accesses the data. This allows the application to change its data access modality for the second phase (retrieve remotely, request filtering, move to data location) based on the value of the abbreviation and knowledge of its own requirements. This insight comes from our experience with writing other applications in Sumatra; `adaptalk` does not benefit from this as the size of all messages is small.

An important question that needs further investigation is where the control for mobility decisions should be placed – whether mobility decisions should be made by a central controller that keeps track of the state of all links or by multiple local controllers that use information only from a small subset of the links. Centralized decisions are likely to be more expensive than distributed decisions (the latter need less information and less synchronization) but could yield better performance (as they use global information).

To answer the second question, we evaluated the prof-



(a) Dynamic vs static placement

(b) Dynamic vs one-shot placement

Figure 6: Maximum latency (over all participants) vs time in `adaptalk`. The one-shot-placement and the static-placement are computed based on latency information available when the conversation is initiated. The client population is stable throughout the experiment.

itability of adapting to changes in the user-distribution as well as spatial and temporal variations in network latency. Adapting to changes in user-distribution led to significant gains allowing `adaptalk` to find better placements as more users came online. Support for mobility allows applications built around a central data-structure to recover from a poor initial placement of this structure by repositioning it to a more suitable location. Adapting to temporal variations alone did not not lead to significant benefits over the period of an hour. In light of this experience, we expect that a simpler migration policy for `adaptalk` for short periods would consider migration only when users join or leave the conversation, rather than on every message as is currently done. Since long-term variation of latency could be as large as 550 ms (US hosts) and 5750 ms (non-US-hosts), longer conversations could still benefit from adapting to temporal variations.

Our experiments with Komodo illustrate that cheap active monitoring can provide network information that can be profitably exploited. Though it would be best to use Komodo as a stand-alone system supplying network information to many distributed applications, its cost is so low that one can contemplate rolling Komodo into individual applications such as `adaptalk` without overloading the network. Active monitoring was needed for `adaptalk` as it needed information about links that are not used in the current placement but could be used in alternative placements. Other applications that change the location of computation but do not change the pattern of communication would not need active monitoring as they could piggyback monitoring information on existing messages. An example of such an application would be an information access program on a mobile platform which moves primarily between this platform and a proxy host on the static network. Active monitoring, as implemented in Komodo, will not be as cheap for applications that are bandwidth-sensitive and not latency-sensitive. We are currently investigating methods to cheaply estimate Internet bandwidth.

In this paper, we have considered Internet hosts that are static. If the platform is mobile and is able to switch between multiple wireless networks [14], the temporal variation in latency could be greater and more abrupt. In these cases, adapting to short-term temporal variations could provide a significant benefit even for latency-sensitive applications.

System stability is a potential concern for programs whose components are mobile. We believe that system stability is a property of the application and not the underlying system support. Accordingly, Sumatra does not provide automatic tracking. Instead, it provides support (in the form of *object-moved* exceptions) that allows applications to track mobile objects (see section 2 for details). We have not yet encountered stability problems in any of our applications.

Finally, we would like to argue the need for mobility as an adaptation mechanism. An alternative adaptation mechanism, which places replicated servers at all suitable points in the network, could adapt to spatial, temporal and population variation by handing off control between servers and by using dynamically created hierarchies of servers. It is quite likely that for any particular application, such a strategy would be able to achieve the performance achieved by programs that use program 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.

7 Related work

Process migration and remote execution have been proposed, and have been successfully used, as mechanisms for adapting to changes in host availability [5, 7, 15, 21, 24]. Remote execution has also been proposed for efficient execution of computation that requires multiple remote accesses [6, 8, 22] and for efficient execution of graphical user interfaces which need to interact closely with the client [2]. Both these application scenarios use remote execution as a way to avoid using the network. Most proposed uses of Java [10] also use remote execution to avoid repeated client-server interaction. In these applications, decisions about the placement of computation are hardcoded. To the best of our knowledge, Sumatra (together with Komodo) is the first system that allows distributed applications to *monitor* the network state and *dynamically* place computation and data in response to changes in the network state. We also believe that our experiment with `adaptalk` is the first attempt to determine if the variation in Internet characteristics is such that it is profitable for applications to adapt to them.

Network-awareness is particularly important to applications running on mobile platforms which can see rapid changes in network quality. Various forms of network-awareness have been proposed for such applications. Application-transparent or system-level adaptation to variations in network bandwidth has been successfully used by the designers of the Coda file system [17] to improve the performance of applications. The Odyssey project on mobile information access plans to provide support for application-specific resource monitoring and adaptation. The primary adaptation mechanism under consideration is change in data fidelity [20]. Athan and Duchamp [1] propose the use of remote execution for reducing the communication between a mobile machine and the static network. In all these systems, location of the various computation modules is fixed; adaptation is achieved by changing the way in which the network is used.

Several systems have been built which permit an executing program to move while it is in execution - for example Obliq [3], Agent TCL [11], Emerald [13], Telescript [23] and TACOMA [12].The primary distinction between these systems and Sumatra is that in Sumatra, *all* communication and migration happens under application control. Complete application control allows us to easily explore different policy alternatives for resource monitoring and for adapting to variations in resources.

Several studies have been performed to determine end-to-end Internet performance. Sanghi et al [19] and Mukherjee [16] have studied network latency. Their observations show that while round trip times show significant variability with sharp peaks, there exist dominant low frequency components. This is consistent with our observations that in a time window of reasonable size, the mode value usually dominates and that the mode value changes slowly.

Golding [9] and Carter and Crovella [4] have studied mechanisms to estimate end-to-end Internet bandwidth. Golding's results indicate that attempts to predict bandwidth using previous observations alone is unlikely to work well. Carter and Crovella propose the use of round trip times for short packets to estimate network congestion. They propose to use the network congestion information to estimate changes in network bandwidth (assuming the inherent bandwidth of the link has been previously computed by flooding the link). Their results indicate that it might be possible to estimate the *change* in network bandwidth using information about the *change* in network latency.

8 Conclusions and future work

This paper is a first step in demonstrating that distributed programs can use mobility as a tool to adapt to variations in their operating environment. Our exploration of network-aware mobile programs lead us to the following conclusions. First, network-aware placement of components of a distributed application can provide significant performance gains over a network-oblivious placement. For short term applications (applications that run for an hour or so), exploiting spatial variations as well as variations in the number and location of the clients achieves most of the gains. For longer-running applications, exploiting temporal variations might be worthwhile. Second, effective mobility decisions can be based on coarse-grained monitoring. This allows cheap active monitoring without losing effectiveness. Finally, there is significant spatial and temporal variation in Internet latency which can be effectively adapted to by mobile programs.

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. One future direction we would like to pursue is to identify such applications and understand their structure and requirements. Some of the examples we intend to study include network-bandwidth-aware data-combination on the Internet, custom combination of periodically generated large-volume datasets (such as weather information) and resource-aware pre-fetching for web clients.

Another direction that we plan to explore is efficient distributed monitoring of other resources, in particular network bandwidth and server availability. We are investigating cheap methods of estimating network bandwidth. An important question that we are investigating is how accurate resource estimates need to be in order to benefit from resource aware mobility and how the accuracy of estimation affects performance.

Network-awareness is very important to applications running on mobile platforms which can see rapid changes in network quality. The nature of these variations is significantly different from those that we have seen in our Internet study. We are planning to extend our work to mobile platforms. In particular, we would like to understand the monitoring requirements and the mobility algorithms in a mobile computing environment.

The focus of our work has been to make distributed applications achieve better performance using mobility as a tool to adapt to resource variations. We have therefore not as yet addressed the important issues of security and resource-use containment in our implementation. We plan to look into both these issues.

Acknowledgments

We would like to thank Mustafa Uysal, Manuel Ujaldon and anonymous referees for their suggestions. We would also like to thank John Kohl, our shepherd.

References

[1] A. Athan and D. Duchamp. Agent-mediated Message Passing for Constrained Environments. In *Proceedings of the USENIX Mobile and Location-independent Computing Symposium*, pages 103–7, Aug 1993.

[2] K. Bharat and L. Cardelli. Migratory Applications. In *Proceedings of the Eighth ACM Symposium on User Interface Software and Technology*, pages 133–42, Nov 1995.

[3] L. Cardelli. A Language with Distributed Scope. In *Proceedings of the* 22^{nd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1995.

[4] R. Carter and M. Crovella. Dynamic Server Selection using Bandwidth Probing in Wide-Area networks. Technical Report BU-CS-96-007, Boston University, March 1996.

[5] J. Casas, D. Clark, R. Konuru, S. Otto, and R. Prouty. MPVM: A Migration Transparent Version of PVM. *Computing Systems*, 8(2):171–216, Spring 1995.

[6] S. Clamen, L. Leibengood, S. Nettles, and J. Wing. Reliable Distributed Computing with Avalon/Common Lisp. In *Proceedings of the International Conference on Computer Languages*, pages 169–79, 1990.

[7] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–85, Aug 1991.

[8] J. Falcone. A Programmable Interface Language for Heterogeneous Systems. *ACM Transactions on Computer Systems*, 5(4):330–51, November 1987.

[9] R. Golding. End-to-end performance prediction for the Internet (Work In Progress). Technical Report UCSC-CRL-92-26, University of California at Santa Cruz, June 1992.

[10] J. Gosling and H. McGilton. The Java Language Environment White Paper, 1995.

[11] R. Gray. Agent TCL: A Flexible and Secure Mobile-agent System. In *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96)*, July 1996.

[12] D. Johansen, R. van Renesse, and F. Schneider. An Introduction to the TACOMA Distributed System Version 1.0. Technical Report 95-23, University of Tromso, 1995.

[13] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(2):109–33, February 1988.

[14] R. Katz. The Case for Wireless Overlay Networks. Invited talk at the ACM Federated Computer Science Research Conferences, Philadelphia, 1996.

[15] M. Litzkow and M. Livny. Experiences with the Condor Distributed Batch System. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Huntsville, Al., 1990.

[16] A. Mukherjee. On the dynamics and significance of low frequency components of Internet load. *Internetworking: Research and Experience*, 5(4):163–205, Dec 1994.

[17] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, December 1995.

[18] M. Ranganathan, A. Acharya, and J. Saltz. Distributed Resource Monitors for Mobile Objects. In *Proceedings of the Fifth International Workshop on Operating System Support for Object Oriented Systems*, pages 19–23, October 1996.

[19] D. Sanghi, A.K. Agrawala, O. Gudmundsson, and B.N. Jain. Experimental Assessment of End-to End Behavior on Internet. Technical Report CS-TR-2909, University of Maryland, June 1992.

[20] M. Satyanarayanan, B. Noble, P. Kumar, and M. Price. Application-aware adaptation for mobile computing. *Operating Systems Review*, 29(1):52–5, Jan 1995.

[21] J. Smith. A Survey of Process Migration Mechanisms. *Operating Systems Review*, 22(3):28–40, July 1988.

[22] J. Stamos and D. Glifford. Implementing Remote Evaluation. *IEEE Transactions on Software Engineering*, 16(7):710–22, July 1990.

[23] J. White. Telescript Technology: Mobile Agents. *http://www.genmagic.com/Telescript/Whitepapers*.

[24] E. Zayas. Attacking the Process Migration Bottleneck. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 13–24, November 1987.