Adaptive Remote Paging for Mobile Computers

Bill N. Schilit and Dan Duchamp

Department of Computer Science
Columbia University
500 West 120th St. Room 450
New York, N.Y. 10027

TR CUCS-004-91
February 1991

Abstract

There is a strong trend toward the production of small "notebook" computers. The small size of portable computers places inherent limits on their storage capacity, making remote paging desirable or necessary. Further, mobile computers can "walk away from" their servers, increasing load on network routing resources and/or breaking network connections altogether. Therefore, it is desirable to allow client-server matchups to be made dynamically and to vary over time, so that a client might always be connected to nearby servers. Accordingly, we have built a self-organizing paging service that adapts to changes in locale and that stores pages in remote memory if possible. We show empirically that there is no performance penalty for using our paging facility instead of a local disk. This suggests that portable computers need neither a hard disk nor an excessive amount of RAM, provided that they will operate in environments in which remote storage is plentiful. These are important facts because both a hard disk and large amounts of RAM are undesirable characteristics for very small portable computers.

1 Introduction

One of the strongest trends in the computer industry is the production of small "notebook" computers. Such computers are characterized by, among other things, powerful microprocessors, easy portability and limited storage capacity. The computing environment we envision is one where portable personal machines move with their owners but obtain computing services from an infrastructure of servers that are for the most part stationary. We make three suppositions:

1. That the reduced size of portable computers places inherent limits on their storage capacity, thus making remote paging desirable or necessary.
2. That mobile computers will regularly "walk away from" their servers, thus increasing load on network routing resources and/or breaking network connections altogether. Therefore, it is desirable to allow client-server matchups to be made dynamically and to vary over time, so that a client might always be connected to nearby servers.
3. That remote memory is faster than local disk, and that increases in network and processor speeds will continue to outstrip improvements in disk transfer rates. In an environment having an abundance of portable computers, the total amount of free remote memory could be quite substantial, and offers an inviting resource for remote paging.

Accordingly, as an experiment in mobile operation, we have designed and built a remote paging service which exhibits two novel features:

1. The service is "self-organizing." That is, it adapts to changes in its computing environment.
2. Pages are stored in memory if at all possible; the hypothesis is that remote memory can be faster than local disk.

Self-organization means that clients and servers dynamically find each other and match up without any predeclaration and that client-server bindings may vary over time due to changes in network load, server load, or server availability.

Incorporating adaptiveness into a remote paging facility requires a number of new and refined algorithms. The high level decisions leading to our design are presented in the next section. Section 3 describes the details of our Mach-based implementation. Sections 4 and 5 analyze the effectiveness of our code and of Mach, respectively.

2 Major Design Decisions

Key issues in the design of a self-organizing remote paging system are:

1. *Role playing*: which machines are clients and which are servers.
2. *Adaptation*:
   (a) which objects migrate from clients to servers,
   (b) how load is balanced among servers as operating conditions change.
3. *Resource location*: how a client locates servers.

Roughly speaking, we split our software into three pieces (depicted in Figure 1) mapping one-to-one with the issues mentioned above:

1. Each service machine runs a *paging server* (simply called a server). The server reads and writes pages of memory objects, as instructed by the kernel.
2. Each network runs a *broker*. The broker is a centralized resource allocator that knows about available space on servers. Fault tolerance is maintained by regenerating, or electing, a new broker when the existing one crashes.
3. Each client machine runs a *service organizer*. The service organizer is "glue" that locates remote paging storage. The organizer sets up a direct path between the client's kernel and one or more paging servers. Thanks to this, the kernel can use Mach's external memory object interface unchanged and a paging server need implement only a single interface for local and remote paging.

2.1 Division into Clients and Servers

Every machine runs both the service organizer and a paging server, and so every machine can be either client or server. Such hermaphroditic organization increases the self-organizing nature of the system and, for example, permits a number of portable computers to power-on simultaneously and then organize automatically into clients and servers based on supply and demand. Clients are not predisposed to use a particular server, but rather have the service organizer refer them to the "best" server at the time of their need. Both client and server can adapt to changes in locale or network configuration: a client may use several servers at once, and servers can migrate paging storage from one server to another.

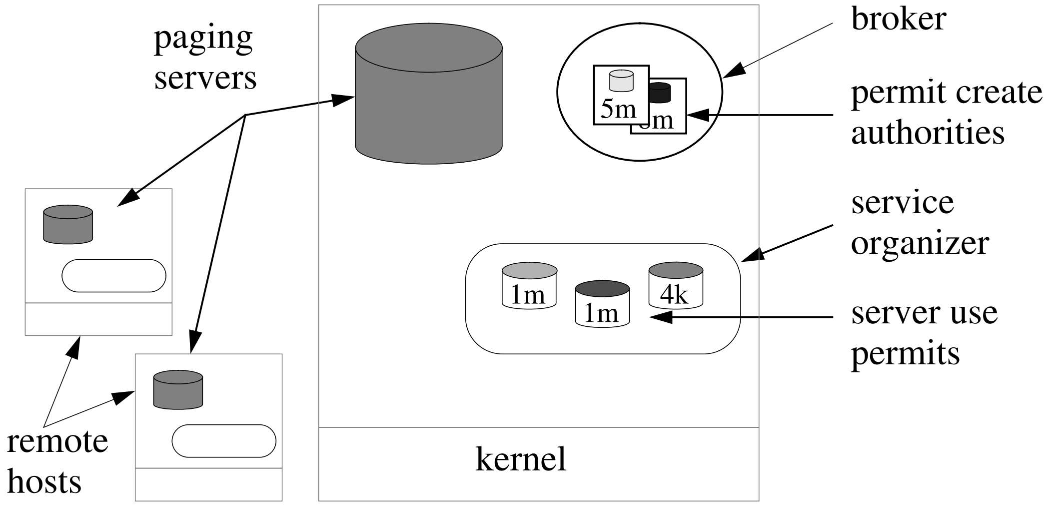

Figure 1: Adaptive Paging Overall Structure

2.2 Granularity of Paging Objects

Remote paging service is defined in terms of *paging objects*. A key issue is how to map client-side memory objects (e.g., pages) to server-supplied paging objects. An important question is what size paging objects should be. Coarse grain allocation has the advantage of limiting the overhead of creation and deletion of paging objects. Fine grain allocation makes it easier for objects to transparently migrate from one server to another.

Mach already has a protocol for paging memory objects, shown in Figure 2. We decided to use this interface across the network for several reasons. First, no local storage is wasted on tables for remapping Mach memory objects into remote paging objects. Further, it is not necessary to interpose extra code between the kernel and the paging server to remap memory object addresses to paging object addresses. Finally, creation and deletion costs can be amortized by migrating large long lived objects to the remote server and keeping short lived small objects locally.

2.3 Placement and Migration of Paging Objects

Whenever possible, a paging object is created within the paging server at the site whose kernel created the object. The reason is that a majority of paging objects are short-lived (c.f. Section 4.1), and it is desirable to have them live and die in the same location.

A paging server can be used alone or as part of a brokered collection. Initially, the organizer arranges that a paging server works alone to provide service only to local processes. If free space exceeds some threshold, requests are accepted from other sites. Once free space is mostly used up, the server picks objects to be *migrated* and asks the service organizer to arrange the destination; the service organizer does so, possibly with the help of the broker. At this point the organizer also revokes its commitment to serve other sites.

The objects that should be moved to remote storage are those whose migration costs the least in terms of movement and anticipated future references. The object selection algorithm, called LRU+SIZE (described in Section 3.3.2), prefers large idle objects.

Presently, the only reason a server decides to migrate some objects is if it begins to run out of space. We plan also to monitor link performance and initiate migration of objects away from servers that deliver poor performance. This feature will be a key to adapting to changes induced by movement.

Unlike processes, paging objects are easy to move: they have few dependencies on their computing environment, and the ones they do have, such as byte order and page size, are transparent to the client. The

```
memory_object_create(old_memory_object, new_memory_object, new_object_size,
    new_control_port, new_name, new_page_size)
```
Supply server with responsibility for a kernel created memory object.

```
memory_object_terminate(memory_object, memory_control, memory_object_name)
```
Indicate to server that no further calls will be made on the memory object.

```
memory_object_data_request(memory_object, memory_control, offset, length, desired_access)
```
Request data from the server managed memory object.

```
memory_object_data_write(memory_object, memory_control, offset, data, data_count)
```
Write data to the server managed memory object.

```
memory_object_data_provided(memory_control, offset, data, data_count, lock_value)
```
Supplies the kernel with data from a memory object.

```
memory_object_data_error(memory_control, offset, size, error_value)
```
Inform the kernel of data retrieval failure.

Figure 2: External Memory Manager Protocols (subset)

problem of preserving the correctness of operations applied to paging objects while they undergo migration can be handled by deferring the operations until the object has been moved and then forwarding the operations to the new server. This approach should cause only minor delays because the objects that are moved are precisely those that receive little use.

3 Implementation

3.1 Mach Overview

We decided to build on Mach because of its external memory management interface, its support for migrating services, and its availability on portable machines. By now Mach is well known [10, 1]. Nevertheless, in order to provide a self-contained paper, the rest of this section briefly describes the kernel facilities that are important to our work.

In Mach, resources are contained within a *task* which may have several computational *threads*. The task-thread division makes it easy to share data structures among potentially blocking threads of execution.

The *port* is a communication channel over which messages are sent and received. Each task has a kernel-administered port name space: many tasks may access a port, each using its own name. Associated with a port name are *access rights*. *Send rights* permits a task to write messages to the port; *receive rights* allows a task to read messages from the port. Any number of tasks may hold send rights to a port, but only one task may hold receive rights. Port rights may be transferred in messages between tasks using Mach's typed inter-process communication (IPC) mechanism. Since at most one task may hold receive rights at any time, transferring receive rights first removes the rights from the sender before transmitting them to the receiver.

A program may elect to receive *notification messages* for events such as the deletion of a port for which it holds receive rights. Notifications facilitate garbage collection. The *backup port* mechanism allows a task to recover a port being deleted by another task. Backup ports help implement resilient services by allowing a manager to obtain the service port and restart servers when they crash.

A port is often used to represent an object within a server; i.e., as a capability. One protocol employing capabilities is the *external memory management interface* (EMMI) [13]. In this case an unprivileged task provides backing storage for *memory objects* in response to the kernel's paging requests. The EMMI is show in Figure 2. Memory objects not serviced by a user supplied manager are considered *anonymous* and sent to Mach's *default pager*. The *default memory object protocol* is a part of the EMMI allowing the kernel to efficiently create and handoff memory objects (i.e., *paging objects*) to the default pager.

Messages can be transparently forwarded between tasks; for example, RPC takes place this way. A per-host *netmsgserver* task receives messages on a proxy port and sends them over the network to a peer netmsgserver which in turn transfers them to the remote task. This makes ports location-transparent.

In order for two tasks to communicate, they must first obtain a common communication port. The *netnameserver* allows programs to register and lookup string-to-port associations.

Central to our implementation is the ability to transfer port rights between tasks, both locally and across the network. This allows us to migrate a paging object from one host to another by transferring receive rights for a port representing the object. As long as the recipient can provide the same protocol and services as the sender, the change in location is unnoticed.

3.2 System Structure

The kernel, paging servers, organizer, and broker communicate using a number of message protocols, depicted in Figure 3 and described below:

(a) *kernel → organizer.* Mach's *Default Memory Object Protocol* is used by the kernel to create a paging object. The organizer intercepts the request and determines the target server. After the server object is created, read and write messages pass directly between the kernel and the server.

(b) *kernel ↔ server.* Using Mach's *Memory Object and Memory Control Protocols*, the kernel interacts with the server for reading, writing, and terminating paging objects.

(c) *organizer ↔ server.* The *Paging Object and Paging Control Protocols* allow the organizer to create and migrate server paging objects.

(d) *organizer ↔ broker.* The *Allocation Protocol* is used by the organizer to obtain server storage through the broker. A capability to access the storage, called a *permit* is returned.

(e) *broker ↔ server.* When receiving an organizer request for storage, the broker determines an appropriate server and forwards the *Allocation Protocol* request over a private port. The broker is informed when allocations or capacities at servers change through the *Update Protocol.*

(f) *broker ↔ broker.* When brokers discover one another they send a takeover bid message. This is the *Broker Election Protocol.*

Mach allows a privileged task to set the port for the default pager. In the future we will use a per-task pager port. Currently our test software links in a special version of the virtual memory allocation system call `vm_allocate()` to simulate the setting of the kernel default pager port. Our `vm_allocate()` locates the service organizer port through the netnameserver, creates the paging object, and maps it into the user address space with `vm_map()` using the arguments provided by the `vm_allocate()` call. The main drawback of this method, aside from the need to relink, is the inability to back kernel-created memory objects. The benefit is ease of debugging and the localization of the effect of software malfunctions during development.

3.3 Service Organizer

The two primary functions of the organizer are:

1. To select a server and create paging objects at the kernel's request.

2. To supply a new server and initiate migration of paging objects to a remote site in response to server requests.

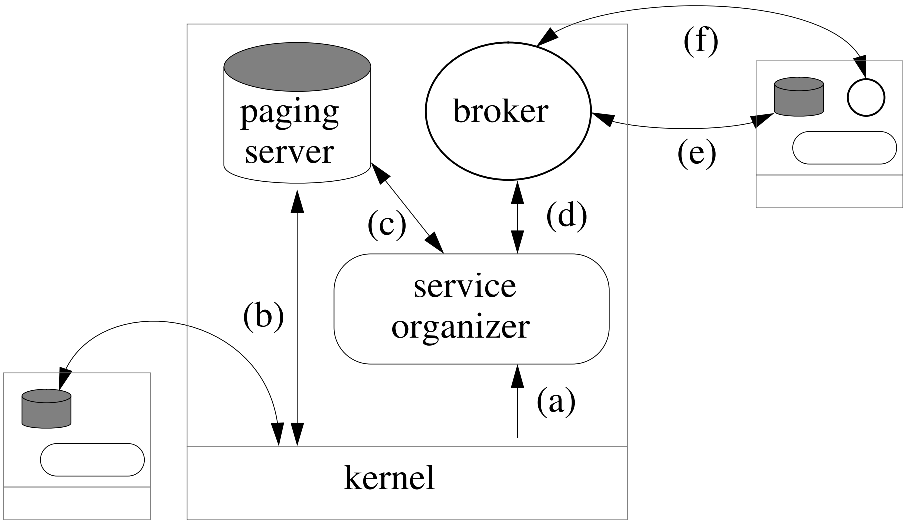

Figure 3: Adaptive Paging Protocols

Creation and placement of paging objects is a "late binding" operation in the sense that the organizer decides at execution time the server on which to place the paging object. The kernel invokes the organizer using the default memory object protocol. The organizer in turn uses the allocation protocol to learn from the broker of potential servers.

Intercepting object creation requests allows the organizer to track current storage allocations at each of its servers. If a paging server later needs to move an object to another site, the organizer uses its allocation data to locate a site with sufficient capacity.

3.3.1 Paging Object Creation

Creation of a new paging object causes the organizer to first search its data structures for any known servers whose usage is sufficiently below allocation. If no existing server has the required storage, the organizer communicates with the broker, first trying to enlarge the allocation of the known servers and then requesting a new server that can meet the request. Enlarging an allocation on a server already in use is desirable since it limits the number of servers employed by a client and hence avoids exacerbating the fault tolerance problem. The organizer always requests slightly more storage than is needed.

The organizer obtains allocations from the broker in the form of *permits*. A *permit* is a capability to use a fixed amount of storage at a particular server. In order to create a new paging object at a server, a permit must be presented. Because the broker holds exclusive rights to create server permits, all allocation requests must go through a server's broker.

3.3.2 Paging Object Migration

When local server space is low, objects are selected for migration. The server keeps paging objects ordered in an LRU queue. Instead of simply taking the object from the head of the queue, the algorithm (LRU+SIZE) chooses from among the first several entries. Given a goal amount of space to free up, the algorithm proceeds as follows: the queue is scanned from the head until objects whose sizes sum to S have been found. The value for S is a function of the goal but must be at least as large. A small S results in a preference for strict LRU, whereas a large S results in more weight given to large objects. At this point the largest objects from those selected, whose combined size meets the goal, are migrated to the remote server.

LRU+SIZE is implemented using a modified heapsort, which acts like a priority queue removing entries until the total size of the elements removed are greater than S. In our case we set S to be twice the goal

```
paging_object_create(server_permit, new_paging_object, new_object_size,
     new_memory_control, new_memory_name, new_memory_page_size, new_paging_control)
```
Accept responsibility for an organizer created memory object.

```
paging_object_forward(paging_object, paging_control, new_server_permit)
```
Request from organizer to forward object to new server.

```
paging_object_forward_complete(paging_control, paging_object, new_server_permit)
```
Indicates to organizer the completion of a forward call.

```
paging_object_abdicate_request(paging_control, paging_object)
```
Request to organizer to offload a server paging object.

Figure 4: Paging Object & Paging Control Protocols

amount we wish to offload. The value is somewhat arbitrary and we have not made tests to tune it. This algorithm is similar to the swapping algorithm used in Berkeley UNIX [6].

Once the objects to offload have been selected, the paging server requests migration by sending the `paging_object_abdicate_request` message to the organizer. The organizer can ignore this message. If the request is accepted, the organizer first selects a destination capable of holding the paging object. The server is informed of the new location and is asked to forward the memory object. Upon completion of the forwarding, the new site notifies the organizer. These steps are shown in Figure 5.

3.4 Paging Server

The server stores paging objects in either a disk partition or paging file, the size of which is specified at runtime.

The protocols used by the paging server include the EMMI, used to read and write pages; the service organizer protocol to control paging object creation and migration; and the allocation and update protocols for providing and controlling storage access.

A paging object providing reading and writing of server storage through the EMMI must be created with a permit. The broker decides which servers should be used and provides permits for their use. When receiving an allocation request, the broker channels the message to the selected server over a private port and obtains a permit satisfying the request. The server's private port is called the *permit create authority* and is passed to the broker during the election process.

The forwarding of paging objects between servers involves three steps. First a new paging object is created at the remote host. Second, pages are written to the remote paging object. And third, the ports representing the object are handed off to the remote server. Kernel request that might have occurred during the first two steps are queued by Mach and transferred with the paging object ports to the remote system by the netmsgserver. Location independence of Mach ports means this new paging object can be used without any changes to the data structures at the client.

3.4.1 Keeping Paging Objects in Memory

Our paging server aims insofar as possible to use idle physical memory to store remote pages. Application control over resident memory is not available under Mach 2.5. However, we take advantage of the fact that an external memory manager can track which of its pages are currently cached in physical memory: those that have been written but for which the kernel has not yet sent a `memory_object_data_write` message. In this way a server maintains a cache that grows and shrinks according to local memory demands, similar to Sprite's variable sized cache [12].

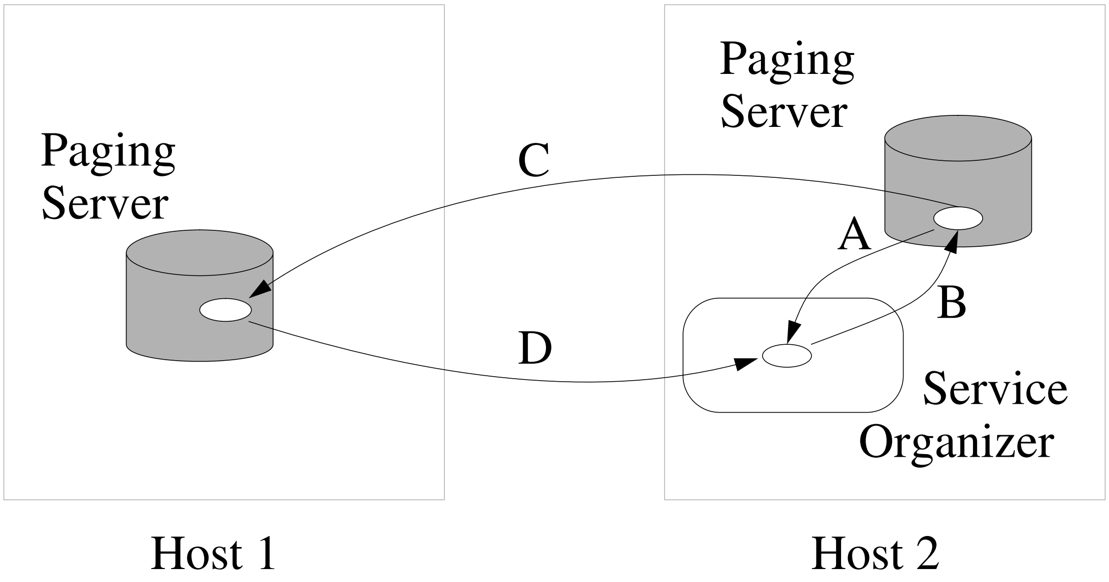

Figure 5: Forwarding a Paging Object

The server creates a memory object the size of physical memory and registers with the kernel as the external memory manager for that object. When a (remote) paging object is sent to the server, the page is written both to disk and to the cache. Paging object read requests that miss in the cache have the side effect of promoting the relevant pages into the cache. Because it is the external memory manager for the object, the paging server is informed if the kernel chooses to page out part of the object. In this case, the paging server simply removes the kernel mapping for the requested page, and removes the page from the cache. The page is not written to disk since it is already available on secondary storage.

3.5 Broker

A single broker maintains a record of the capacity and allocations of each paging server on its network. Every server periodically updates the broker's record of its capacity. When a service organizer requests storage, the broker creates a permit at some server and returns a capability for the permit to the requester.

The broker accepts two types of storage requests: allocations and reallocations. For an allocation the selection policy is to use the server with the maximum available space. This method is used because free space often reflects a corresponding ability to provide service whereas minimum space used may mean nothing.

A policy of strict balancing across servers might cause a client to use a large number of servers, creating a fault-tolerance problem. To avoid this situation, a request to the broker can also modify a permit, enlarging or reducing the allocation. Similarly, a request for a new permit can contain a server hint so that clients can indicate preference for local servers. In the case of reallocation the requester supplies an existing permit along with a delta amount and the broker tries to rewrite the permit.

Permits are not permanent. Reasons for destroying a permit include: the client deleted it, the client rebooted, or the server wants to revoke the allocation it represents. When a permit is destroyed by the client, the server eventually receives Mach's notification message. Because the broker has no way to know that the storage is now available, the server sends an update message with the allocation amount of the permit.

3.5.1 Election and Regeneration

The broker election process is not unlike the real world: brokers meet, bid, and the winner takes over the loser's customers. In this case the customers are those servers for which the broker holds permit create authority. An unusual feature of our election method is that there is no phase during which all candidates communicate and cast their votes. Instead there is a series of two-candidate tournaments eventually resulting in a single broker. Entry of a new server into the global allocation causes only a single round of the tournament. An example sequence of election rounds is shown in Figure 6. The advantages of a tournament style election method are its simplicity and the fact that brokers can provide allocations during the election.

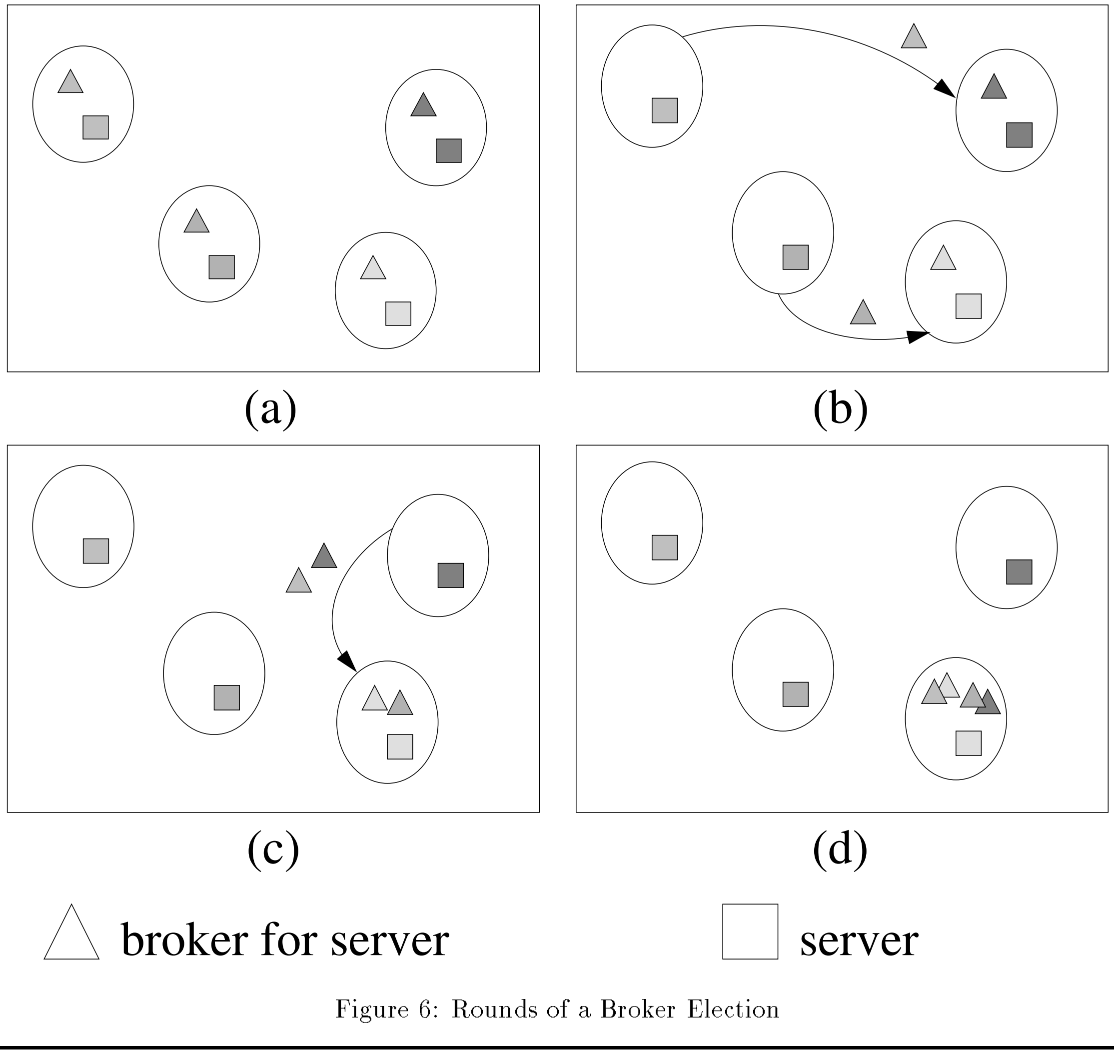

Figure 6: Rounds of a Broker Election

Election consists of two concurrent activities: accepting bids from remote candidates, and generating bids for remote candidates. A bid is simply a remote procedure call containing a bid value and returning the loser's broker state on a successful takeover. For a bid value we use the number of servers controlled by the broker, so big brokers take over little ones. Some coordination is necessary between the process sending bids and the process receiving bids. For example, it would be erroneous to make and win a bid just after losing to another broker. The way we achieve the coordination is to shut down the thread accepting bids before allowing a takeover to complete.

A single broker is quickly elected, and if it crashes it is a serious matter. Broker regeneration is achieved by restarting the election process on each server after noticing that the broker has crashed. Recreating a

```
takeover_bid(broker_port, candidate_port, bid_a, bid_b,
    status, server_ports, server_auths, server_tokens, server_allocs)
```
Try to takeover another broker.

```
broker_update(broker_port, server_port, delta_capacity, delta_allocation)
```
Update broker upon change in server capacity.

```
permit_allocate(authority_port, server_port, permit_port, amount, result)
```
Request for a new paging server permit.

```
permit_reallocate(authority_port, server_port, permit_port, delta_amount, result)
```
Request for a change in existing paging server permit.

Figure 7: Broker Election, Update & Allocation Protocols

broker for the local server requires knowledge about the capcitity and current allocations. The fact that permits are created on servers means that the outstanding allocations can be easily calculated.

4 Evaluation

We have developed a prototype of our design. The software consists of approximately 8,000 lines of C and about 400 lines of MIG interface declarations. The largest pieces are the server and broker (2736 lines), and the organizer (1241 lines).

Our prototype runs on Mach 2.5 on Toshiba 5200 portable computers configured with 8 MB of memory, a 100 MB hard disk, and attached to a 10Mb Ethernet. The Toshibas have a 20MHz Intel 386 processor and an 8-bit AT bus connection to a WD 8003 Ethernet controller.

We ran four experiments:

1. Statistical analysis of paging object size and lifetime, as measured from a real load. We sought to demonstrate that our LRU+SIZE algorithm would find a "natural" set of objects to migrate.

2. Breakdown of RPC costs.

3. Time required to retrieve a page from several different paging mechanisms. Page retrieval is the "common case" for a pager. We sought to show that there is no performance penalty for paging from remote memory, even using the message-based Mach external memory management interface.

4. Time required to migrate a 1-page object. Migration is, we hope, the "uncommon case."

Each of these experiments is described and analyzed in a following section.

4.1 Paging Object Size and Lifetime

In order to gain information on the kernel's use of paging objects we traced events within the default pager. We took care to analyze only paging activity, discarding events related to memory mapped files. Statistics for two traces are shown in Figure 1. Trace 1 covers a period of approximately 27 hours that is characterized by a moderate interactive load including editing and compilations. The second trace occurred over a shorter period (3.5 hours) during which simultaneous kernel builds provided heavy load. The results of these traces can be seen in Table 1 and Figures 8–11 A few facts are apparent.

First, as evidenced by Figures 8 and 10, there is a good-sized pool of large, long-lived objects: among the objects of above-median size, 75% percent live longer than 10 minutes (11% in the heavy load case). These objects provide a good pool for LRU+SIZE to select from.

	Trace	
	1	2
duration (minutes)	1605	207
objects	88	439
reads	1838	15549
writes	463	16099
average object size[1]	77	87
average object usage[1]	20	28
average object lifetime (mins)	869	25

[1] sizes in pages of 4KB

Table 1: Default Pager Statistics

Second, the size of the "paging working set" — that is, the sum of the memory_object sizes in steady state — is likely not very large. For Trace 2 this figure is less than 6MB, while for Trace 1 the number is well under 4MB.[1] These figures are not authoritative, however. Our traces can tell us about object creation and use during the period of the trace but not before. We suppose that a large number of large, long-lived objects are created shortly after boot time when the system initializes. The size of these objects must be added to our computed numbers.

The last fact we notice from our traces is that the object usage rate (percent of allocation used) is quite low; only 25%-30% of the pages in a memory object actually get written.

Under heavy load an anomaly occurs: over half the paging objects created are never written. This is caused by the pageout daemon. The daemon fetches pages from the inactive page queue and writes them to their external memory object. If an external memory object does not yet exist, one is created; but instead of immediately writing out the page, it is instead put back onto the active list. Under the load induced in our test, a large number of 1-page memory objects are created by short-lived processes that terminate before the page becomes a candidate for replacement. Fortunately, these anomalous "zero use objects" do not appreciably affect the results drawn from the trace, and so we filter them out of Trace 2 statistics.

4.2 RPC Latency

Remote paging is an asynchronous protocol, and so it should be expected to be somewhat slower than synchronous RPC. The two reasons for this are that fewer packets are exchanged during an RPC, and that both processes are waiting for the other at the right moment. A remote paging operation of course also includes processing time specific to the paging server. So RPC time is a lower bound on remote paging latency.

We measured an RPC that consists of a small request and a 4KB response. The average time for this operation is 33 ms. The latency breakdown is summarized in Table 2. We computed the fraction of the latency due to the network and bus transfer of the 4KB response, and we measured values for the Mach IPC and TCP costs. The computed values are based on rated 10Mb Ethernet throughput and a 1μs/byte rating of the AT bus. The number and sizes of network packets were observed with a network "snooper" running on a fast machine capable of picking up all packets. The bus transfer cost is doubled because the 4KB block is moved over the bus at both client and server. The Mach IPC costs were measured by running

[1] Working set size is approximated by multiplying object creation rate (number of objects divided by length of the trace) by average object lifetime by average object usage.

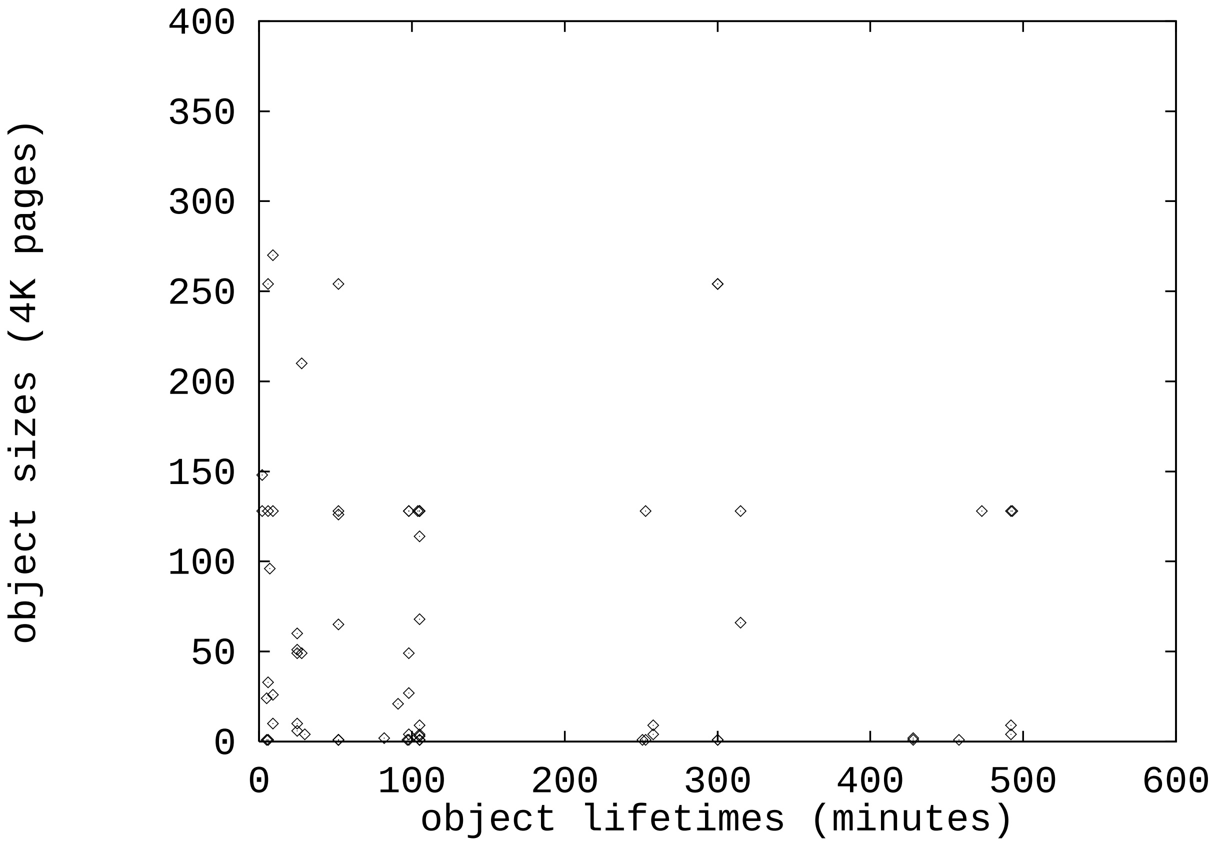

Figure 8: Trace 1 – Allocated Object Sizes vs. Lifetime

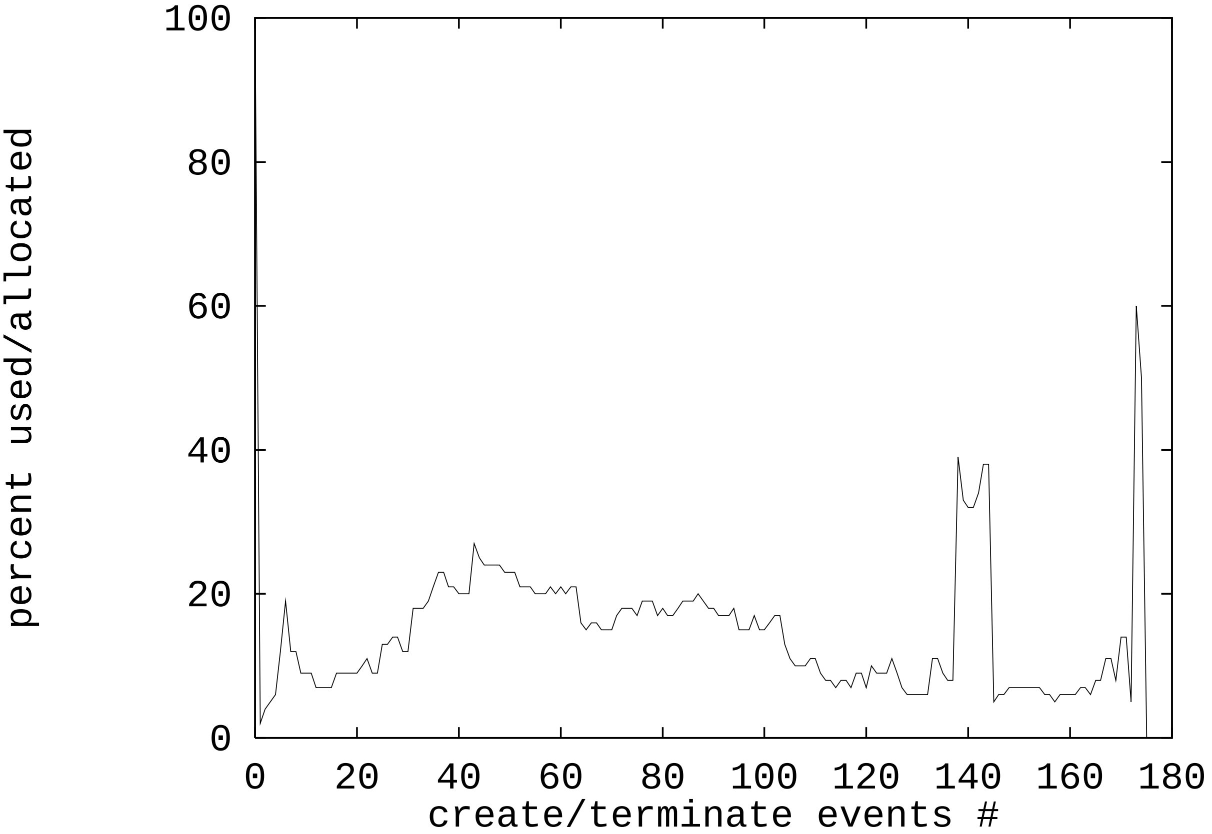

Figure 9: Trace 1 – Object Usage

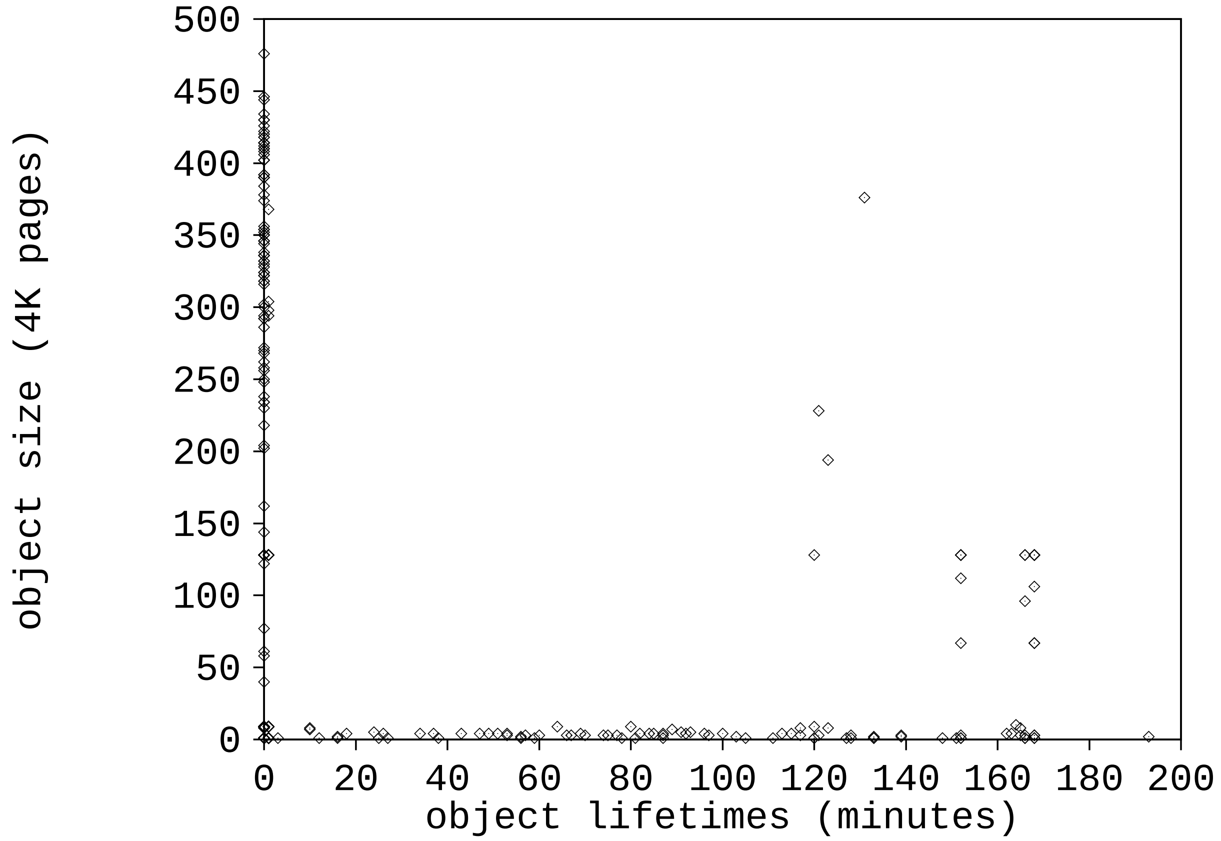

Figure 10: Trace 2 – Allocated Object Sizes vs. Lifetime

the test program locally. These costs are also doubled because message forwarding (via the netmsgserver) adds a local IPC on both sites.

We also measured TCP packet processing time, since Mach's netmsgserver uses TCP for data transport. TCP costs were measured using another program that sent the equivalent of the observed traffic in loopback mode. Using only these computations and measurements we are able to account for about 90% of the observed time. The remainder can be attributed to the netmsgserver, network interrupt processing, context switch and scheduling overhead.

Although we expected the byte-wide AT bus to be responsible for a large portion of the time, it is surprising to see that TCP is the major cost. We can think of two causes. First, we observed that a fixed TCP window size of 4KB was causing servers to interrupt their transmission of data (which was slightly more than 4KB) and wait for an ack. It seems desirable for the netmsgserver to better control the window, perhaps according to the protocol. Second, the netmsgserver uses its own message acknowledgements which cause TCP acks to be transmitted for netmsgserver acks.

4.3 Paging Benchmark

We measured the time to fault in random pages using a number of different configurations, including:

- Mach's default pager using a local file
- Our pager using a local file
- Our pager run remotely using a file

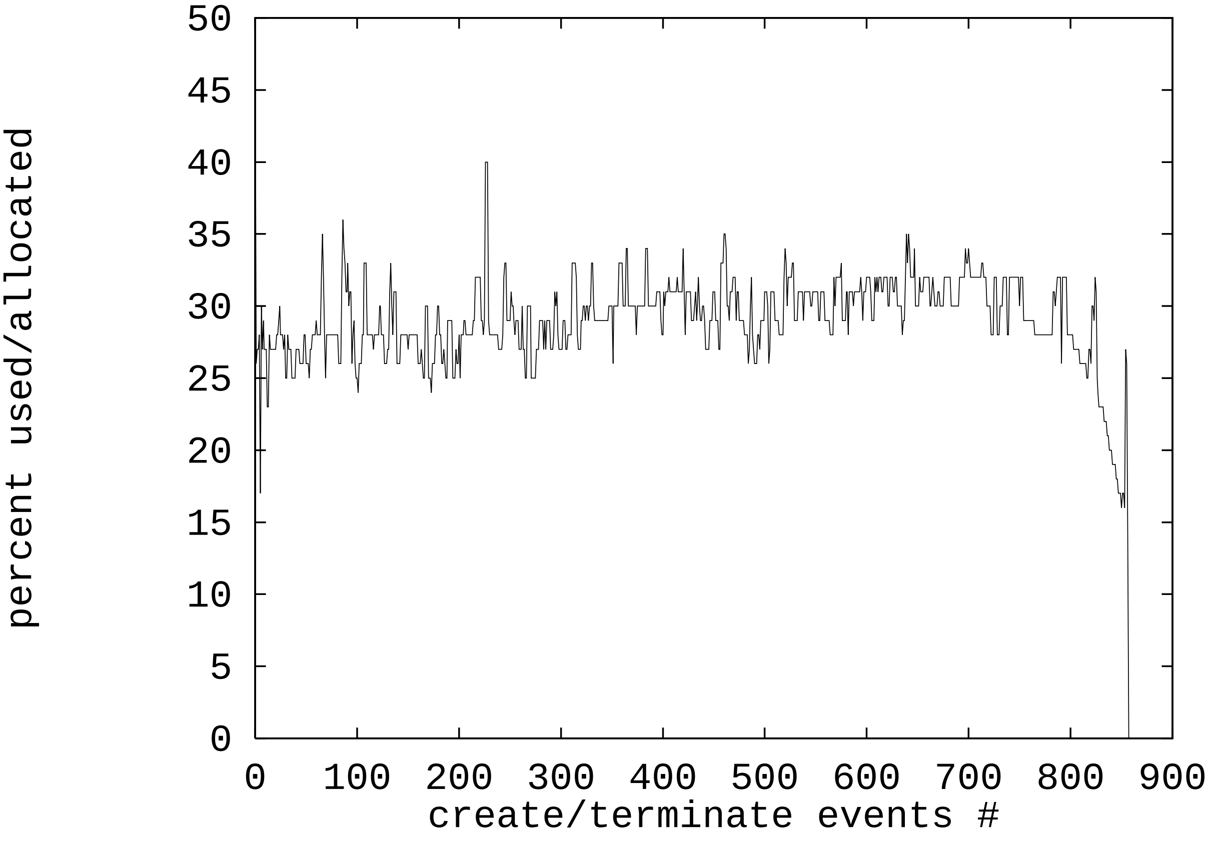

Figure 11: Trace 2 – Object Usage

	RPC Size			
	0K	2K	4K	8K
packets	3	4	7	11
bytes[1]	426	2528	4738	9050
AT bus[2] $\times$ 2	0.8	5.0	9.5	18.1
Ethernet[3]	0.3	2.0	3.8	7.2
TCP loopback	6	9	12	19
Mach IPC $\times$ 2	2	4	4	4
Total	9.1	20.0	29.3	48.3
Measured	11	22	33	52

[1]including all network headers
[2]based on 1 μs/byte AT bus
[3]based on 10 Mb/sec (0.8 μs/byte) Ethernet

Table 2: Synchronous RPC (in ms)

- Our pager run remotely using memory

For comparison we measured the time to directly read 4K pages from a local file and remote NFS file using random access. These disk read times should be the minimum a pager needs to access the same media, though the Mach pager, existing in the kernel address space, benefits from directly issuing VFS operations instead of system calls. Figure 12 summarizes the results.

We obtained our measurements after creating a memory object larger than available memory, writing all its pages, flushing them from memory, and then reading them in random order. This method ensures that the pages being read are located either remotely (in the case of our pager) or on disk.

The measurement of our pager for remote memory tests ensured memory references by mapping all pages of a large paging object into a single remote page. In all cases access was made through the kernel by using a mapped object, the kernel requested pages of size 4K and random access references to the memory object were made.

4.4 Migration Time

We measured migration cost by having the organizer repeatedly forward a paging object between a local and remote server under the control of a test program. The time measured is the complete roundtrip from the forward request until the completion notification, including disk reads and writes, averaged over a 1000 repetitions. The 96 ms time to migrate a 1-page (4KB) object is only slightly longer than the 94 ms needed for asynchronous remote disk reads (c.f. Figure 12). A summary of migration times for various sized objects is shown in Figure 13.

4.5 Conclusions

There is no performance penalty for using an adaptive remote paging facility instead of a local disk. This suggests that portable computers need neither a hard disk nor an excessive amount of RAM, provided that they will operate in environments in which remote storage is plentiful. These are important facts because both

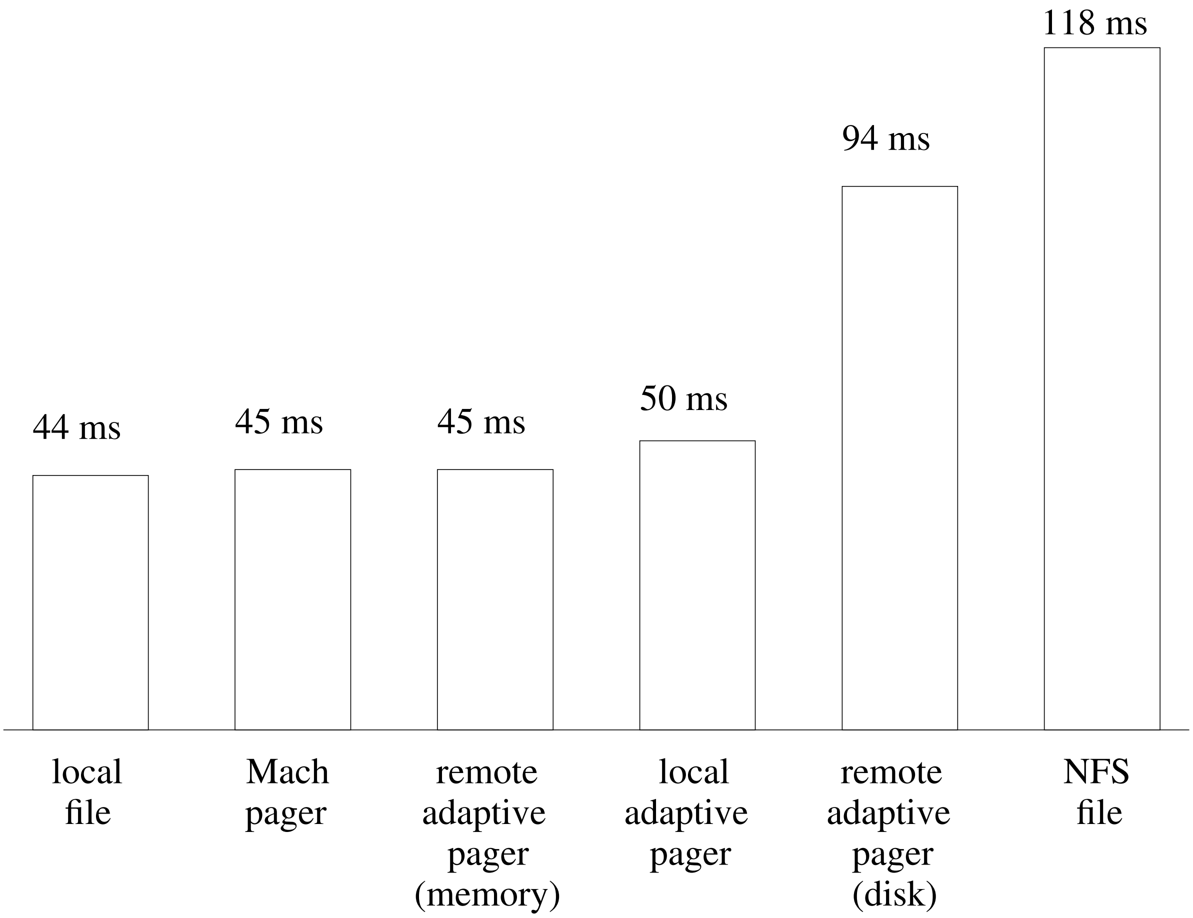

Figure 12: Random 1-page (4KB) Reads

a hard disk (power consumption, weight, reliability, cost) and large amounts of RAM (cost) are undesirable characteristics for very small portable computers.

Regenerating a centralized allocation task using a tournament style election is not unreasonably complicated. This means that the benefit of a centralized scheme, scalability and monitorability, can be had for little added cost. Centralized resource allocation is suitable in many cases where resources are moderately stable and update messages are infrequent.

Basing a remote paging system on EMMI has advantages and disadvantages. The design is simpler, code and data are smaller at the client, but making good use of server allocations is difficult. Since allocated size is usually a gross overestimate, allocating server storage based strictly on the potential size of objects means a large portion of remote server storage will not be used. If strict allocations are to be maintained at the server, then interceding between client and server to remap each memory object operation into a single per server paging object seems desirable. This is especially true given the low cost of Mach IPC relative to network operations. A major drawback is that remapping requires even more resident memory at the client.

We have not addressed the question of ensuring that the processing of paging operations do not themselves cause page faults. This situation, which is a consequence of external memory management in general, is complicated by the many components involved in our paging mechanism. For example it would be potentially fatal to pageout part of the netmsgserver to a remote location since the netmsgserver is required to return the page to the local host. One approach we are looking at is to identify and lock into memory those program pages needed for local disk paging, and to ensure that any network components, if they are paged at all, are only sent to local disk. This approach would further justify changes to Mach to allow a per task default

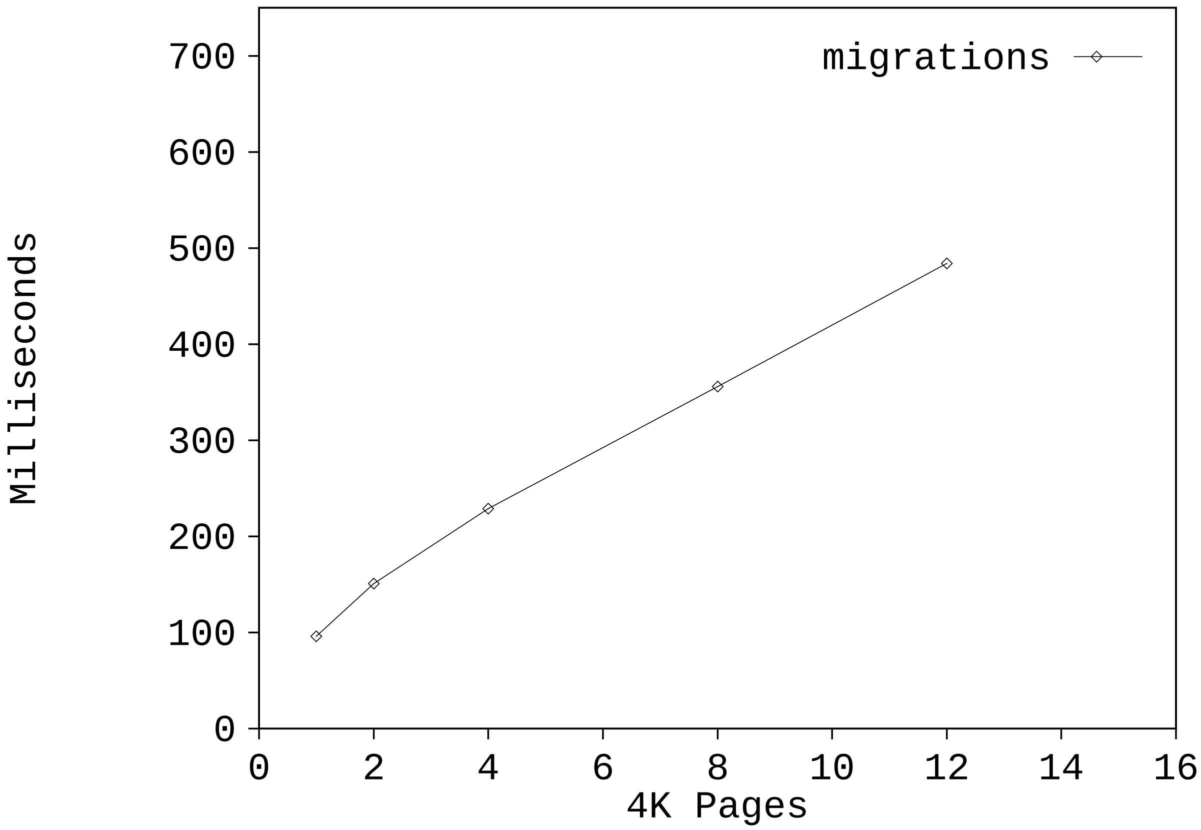

Figure 13: Forwarding Paging Objects

paging, which in the case of the netmsgserver would only allow local paging.

5 Mach Critique

This section discusses our experiences implementing a distributed application using the Mach 2.5 operating system. Overall, we found Mach to be a good development environment with light weight threads, port forwarding, the message interface generator, and the concealment of location and heterogeneity being major benefits. We did however develop the following criticisms, some of which are currently being addressed by the Mach implementers.

EMMI implementation. The Mach kernel currently uses EMMI to send and request a single page at a time. Furthermore, the protocol itself requires any larger amount of data to be contiguous. Our traces show a substantial number of messages one after the other writing the same memory object.

Name Service. The Mach name server interface is not well suited as a low level component of a more sophisticated service because it lacks multi-host responses to lookups. If a name is registered on two hosts then a lookup will (arbitrarily) respond with only one of those names. Indeed, if a name is registered locally, then it is impossible to find other hosts with the same registered name unless those hosts are already known. Our solution — devised to support the broker election — was to change the interface of the name lookup call to include a new form of wild card which will find a remote name even when a local name exists. If Mach clients are regularly reimplementing multi-valued name lookup then perhaps it should become part of the netnameserver.

Transparency not always desirable. Our software would like to track average roundtrip message times for the purpose of server placement and also message timeout values. However, we cannot monitor the roundtrip costs of messages without adding an extra IPC in the critical path whose sole purpose would be to forward and time the remote call. Ideally, the netmsgserver should be able to answer queries about a port's expected roundtrip cost, or even provide a call-back when average latency goes over a predefined level.

Multicast IPC. Mach has no port-oriented multicast mechanism. Although it is possible to send datagrams to multicast addresses, port rights cannot be transferred this way. Instead one must employ a two-step process of sending datagrams containing host names to multicast addresses and then querying the remote hosts' netnameservers to obtain a port for use by Mach RPC.

Physical memory locking. Exporting control over physical memory has been designed, but was not available in version 2.5. A prime customer of such an interface would be an external pager task since it must respond to paging requests without deadlocking itself in a page fault condition. Since the interface itself was unavailable our experience controlling page faults generated during operation is limited.

Network backup ports. For the most part interaction with remote hosts is transparent by virtue of the netmsgserver. However, the backup port facility is not supported over the network. When the server and manager are on different hosts, only the manager's site can save the port from destruction, with all other hosts seeing the port destroyed.

6 Related Work

Our work is unique in combining adaptive load balancing and remote paging, even though neither technique is by itself novel. However, we do add a new twist to each technique:

- our paging load is balanced among a changing set of sites,
- paging objects are stored in memory insofar as possible.

Adaptive remote paging might be viewed as a subset of adaptive process placement and migration, a subject on which there is much earlier work [8, 9, 11]. However, process placement/migration systems have not had major impact because — at least in Unix-based systems — so many processes are short-lived; furthermore, many long-lived processes are long-lived precisely because they manage (and so must reside near) some data set. So short-lived processes should not be moved because to do so would be inefficient, and long-lived processes should not be moved because they have no need to move. We address a more limited but more tractable problem.

Adaptive remote paging might also be viewed as distributed virtual memory [7, 5] without the sharing. Certainly the ability to share common pages among several processes is a major advantage, but along with sharing comes the problem of maintaining consistency. Addressing this problem has proven to require overhead inefficiencies and a good deal of complex code for a relatively uncommon situation. Also, distributed VM systems usually require a static partitioning of the shared address space.

One previous work [2] suggests the use of remote memory as a "new model of computation." The system described therein exploits a predeclared set of dedicated "remote memory servers," and concentrates on the network protocols necessary for interoperability. We get a certain (lesser) degree of interoperability from Mach and instead concentrate on resource location and load balancing issues. Comer's work is similar to ours in that the paging protocol has been separated from the file system.

Dynamic binding of servers to clients has been discussed in [4], which is mostly a proposal. Kazar's idea arose, like ours, from the contemplation of operating in a large, highly variable distributed computing environment.

A major work that addresses the issue of mobility is the Emerald object-based system [3]. This work provides language primitives allowing the programmer to hand-control migration of objects that can be data or processes. The Emerald runtime system then implements process and data migration.

7 Summary

We have demonstrated the design of a paging mechanism that allows portable computers possessing little storage to operate by using the storage of nearby computers as they move.[2] When remote memory is available, there is no performance penalty for using such a system versus using a more conventional local paging disk, and, counting expected upgrades in the capabilities of small computers, paging to remote memory should soon far surpass the performance of local disk paging.

As immediate future work, we plan to install pagers on a per-task basis, to add code that will migrate objects whenever paging performance is noticed to degrade, and to address the page fault circularity question raised by external memory managers in general.

Longer term future work will include investigating whether this sort of adaptability mechanism can subsume other services (e.g. the file system), and the possibility of sharing objects at servers.

8 Acknowledgments

This work was supported by IBM and the New York State Science and Technology Foundation Center for Advanced Technology in Computer and Information Systems.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *USENIX Association Summer Conference proceedings, Atlanta 1986*. USENIX Association, 1986.

[2] Douglas Comer and James Griffioen. A new design for distributed systems: the remote memory model. In *Proceedings, Usenix Summer Conference*, pages 127–135, Anaheim, California, June 1990.

[3] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Trans. Computers*, 6(1):109–133, February 1988.

[4] M. L. Kazar. Workstation operating systems: Invoking remote services. In *Proceedings: Workshop on Workstation Operating Systems*. IEEE Computer Society Technical Committee on Operating Systems, November 1987.

[2] Sources are available by anonymous ftp from internet site `not.revealed` in directory `pub/adaptive-paging`.

[5] P. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson, and B. L. Stumpf. The architecture of an integrated local network. *IEEE Trans. Selected Areas of Communication*, SAC-1(5):842–857, November 1983.

[6] Samuel J. Leffler, Marchall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and implementation of the 4.3BSD UNIX operating system.* Addison-Wesley, Reading, Mass., 1989.

[7] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Computers*, 7(4):321–359, November 1989.

[8] Barton P. Miller Michael L. Powell. Process migration in demos/mp. In *Proceedings of the Ninth Symposium on Operating System Principles*, pages 110–119. ACM, 1983.

[9] D. Nichols. *Multiprocessing in a Network of Workstations.* PhD thesis, Carnegie-Mellon, February 1990. Available as CMU Technical Report CMU-CS-90-107.

[10] Richard F. Rashid. Threads of a new system. *Unix Review*, 4(8):37–49, August 1986.

[11] Marvin M. Theimer and Keith A. Lantz. Finding idle machines in a workstation-based distributed system. *IEEE Transactions on Software Engineering*, 15(11):1444–1458, November 1989.

[12] Brent B. Welch. Naming, state management and user-level extensions in the sprite distributed file system. Phd thesis, University of California, Berkeley, April 1990.

[13] Michael Wayne Young. Exporting a user interface to memory management from a communication-oriented operating system. Thesis CMU-CS-89-202, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1989.