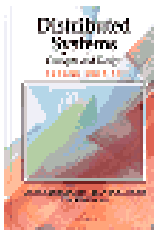


Originally published at pp. 584-97 of Coulouris, Dolllimore and Kindberg, Distributed Systems, Edition 2, 1994.

Amoeba

History and architectural overview

Amoeba is a complete distributed operating system design, including all the basic facilities that one would expect from a conventional operating system. It is currently being developed at the Vrije Universiteit in Amsterdam, where its design and implementation were begun in 1981, and it was previously developed jointly with the Centrum voor Wiskunde, also in Amsterdam. The version of Amoeba we shall discuss is version 5 [Tanenbaum *et al*. 1990, Tanenbaum 1992]. The Amoeba system model is an example of the *processor pool model* (Figure 1). The main components making up the architecture are the processor pool, workstations (SUN-3s and VAXstations are supported), X-terminals, servers and gateways. The gateways are used for connecting Amoeba sites over WANs.

The assumptions behind this architecture are that memory and processors are sufficiently cheap that each user will be allocated multiple processors, and each processor will have plenty of memory to run applications, without the need for backing store. Rather than allocating a multiprocessor to each user, processing power is largely concentrated in the processor pool, where it can be shared more flexibly among users and more economically housed and interconnected.

Design goals and chief design features

Three central design goals were set for the Amoeba distributed operating system, as follows.

Network transparency: All resource accesses were to be network transparent. In particular, there was to be a seamless system-wide file system, and processes were to execute at a processor of the system's choosing, without the user's knowledge.

Figure 1 The processor pool model.

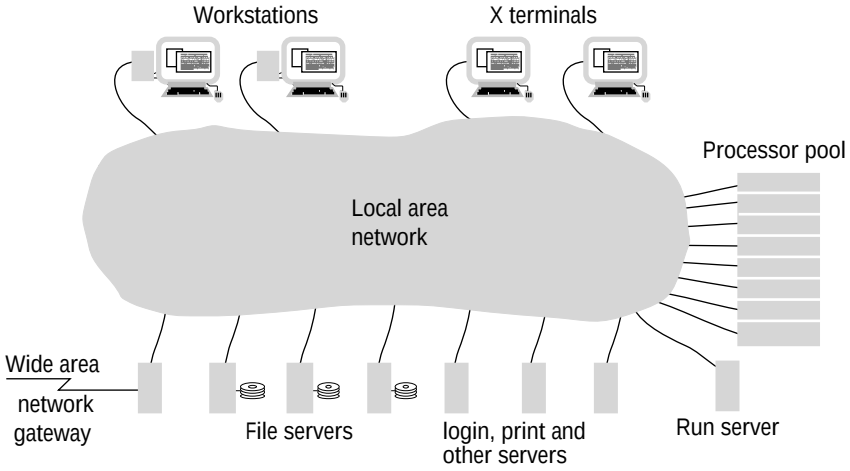
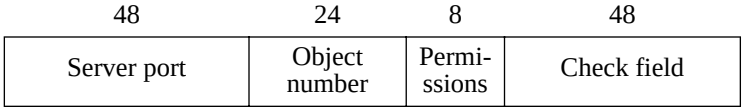


Figure 2 An Amoeba capability.



Field sizes are shown in bits.

Object-based resource management: The system was designed to be *object-based*. Each resource is regarded as an object and all objects, irrespective of their type, are accessed by a uniform naming scheme. Objects are managed by servers, where they can be accessed only by sending messages to the servers. Even when an object resides locally, it will be accessed by request to a server.

User-level servers: The system software was to be constructed as far as possible as a collection of servers executing at user-level, on top of a standard microkernel that was to run at all computers in the system, regardless of their role. An issue that follows from the last two goals, and to which the Amoeba designers paid particular attention, is that of protection. The Amoeba microkernel supports a uniform model for accessing resources using capabilities.

The basic abstractions supported by the microkernel are processes and threads, and ports for communication. Each server is a multi-threaded, protected process. Server processes can occur singly, or in groups, as we shall discuss. Communication between processes at distinct computers running Amoeba on a network is normally via an RPC protocol developed by the Amoeba designers. This protocol is implemented directly by the kernel. Servers that have been constructed include several file servers and a directory server, which stores mappings of path-name components to capabilities for files and other resources.

All three goals have been achieved. They took precedence over any issues of compatibility with existing operating systems. In particular, while a UNIX emulation library exists, the emulation is at the source level and is not accurate.

Protection and capabilities

In Amoeba all resource identifiers are capabilities, implemented in the form shown in Figure 2. A capability is 128 bits long. It contains an identifier that is mapped at run-time onto a *server port*, and the *object number* is used to identify the object within that server. The two additional fields, the *permissions field* and *check field*, are used respectively to identify the types of accesses that the possessor of the capability is allowed to make, and to protect against forgery of the capability.

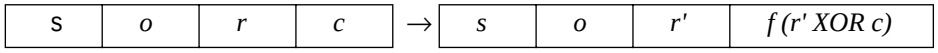
The permissions field requires integrity checks, to prevent users from forging capabilities or tampering with the permissions. Amoeba uses the check field for this purpose as follows.

When a client requests the creation of a new object, the server supplies a capability with all permissions set – an *owner capability* (the creator of an object can do with it what it likes). This capability contains: the identifier of the server port for receiving request messages; a new object number; a permissions field allowing all operations on the object; and a 48-bit random number in the check field. The server stores the owner capability with the new object's data.

Now, consider a client that attempts to forge a capability with all the permissions bits set. It can copy the server port identifier from another capability and guess an object number. However the client is unlikely to be able to guess the check field. There are 2^{48} – about 10^{14} – combinations of 48-bit wide fields. Generating and testing all these combinations by brute force would involve passing each guess in a message to the server, at about 2 milliseconds for each guess. That is about 2×10^{11} seconds, or about 6,300 years. The same argument can be applied to the 48-bit server port identifier, to show that a process not knowing the target process's port identifier is highly unlikely to succeed in guessing it using brute force.

Reduced capabilities ◊ Clients with owner capabilities often want to allow others to access their resources, but they do not necessarily want other clients to be able to perform all operations upon the resource. The client must be able to acquire *reduced capabilities* – legitimate capabilities with restricted rights – and the server must be able to test them for validity. An obvious solution is for the server to generate and store a different check field for each combination of rights for each object. This has two disadvantages: first it introduces further storage requirements at the server, and second, clients must request the creation of new capabilities by the server.

In the Amoeba algorithm, a client can use one-way function (see Chapter 7) to create a reduced capability from an owner capability. The client creates the reduced capability by using a one-way function *f* and the exclusive-or binary operator XOR, as follows:



The new capability thus has the same server port (*s*) and object number (*o*) fields, rights field set to the reduced rights *r'*, and check field equal to *f*(*r'* XOR *original-check-field*). In order to check a capability with reduced rights, the server performs the same computation to calculate the check field as the client, using the stored check field, the rights in the capability being checked and the same one-way function. If the capability is not a forgery, then its check field will match the result of this computation. The computation can be avoided by caching the results after the first time it is made. This avoids subsequent calculations for the same rights combination, when a direct comparison can be made of the check field with the cached value.

A client with a reduced capability cannot increase its rights using this method, since it does not know and cannot determine the original check field (because a one-way function was used). Note that only a client possessing an owner capability can fabricate a reduced capability. Clients possessing a reduced capability have to request the server (or another client with an owner capability) to fabricate a capability with yet fewer rights.

If servers use access control lists, an authentication protocol (see Chapter 7) would be required for each access. Capabilities avoid this (although authentication is required at some point in order to obtain existing capabilities in the first place).

Capabilities do not solve the problems of eavesdropping and replaying: an intruder can examine messages being sent over the network, and copy capabilities (or encrypted capabilities) out of them, to be used in malicious accesses to the corresponding resource.

A final disadvantage of capabilities in general, compared to access control lists, is that they cannot easily be retracted. If Smith and Jones have each been given capabilities to access a certain file, how is it possible to retract Jones's rights to access the file, but not Smith's? The only way is for the server to associate a different set of capabilities with the file, and to give a new capability to Smith, but to ensure that it is not given to Jones. However, if Smith decides to grant access to the file to Jones, then she has only to pass the capability to Jones, thus thwarting the owner's wishes.

Processing and communication

An Amoeba process consists of an execution environment together with one or more pre-emptively scheduled threads. An Amoeba address space consists of an arbitrary number of regions, which may be mapped into the address space of more than one process, enabling sharing.

Each process is associated with some threads, some ports and some memory segments. An Amoeba memory segment is an array of bytes, which can be mapped into a region. It might be stored, for example, in a file or in main memory.

Amoeba does not provide demand paging, swapping or any other scheme whereby mapped data can be non-memory-resident. The designers claim that workstations in the near future will have sufficient memory to enable most large programs to fit into it. Performance is enhanced and the kernel is simplified by the assumption that all mapped data is in memory.

A *process descriptor* is a data structure that describes a process: the layout of its address space, capabilities for the corresponding memory segments, the ports and the states of its threads (including state of execution, program counter, stack pointer and other registers). Given a process

descriptor, a process can be created. The address space, threads and ports can be created as kernel-managed resources; the data in the memory segments can be copied or mapped into the address space.

An executing process can be sent a signal that causes it to be suspended, and causes a process descriptor to be constructed. In principle, this can be used to recreate the process at another host computer, and free the resources at the original computer. This is a means of achieving process migration.

Most servers run at user-level, although some, such as the memory server, execute in the kernel for efficiency. Processes that execute at user-level but which have to access hardware resources such as device controllers do so through a message passing interface exported by the kernel. Of course, only processes in possession of the requisite capabilities can access these resources.

The kernel provides just three major system calls, which are similar to *doOperation*, *getRequest* and *putReply* introduced in Chapter 4, and are used in the same way. The equivalent of *doOperation* is called *trans*, and has at-most-once semantics (see Chapter 5). There is no asynchronous message *send* call in Amoeba. Those wishing to avoid the synchronous behaviour of *trans* must create a separate thread to make the call.

The definitions of the Amoeba communication calls are given in the ANSI C language. All three calls use a *Msg* data structure, which is a 32-byte header with several fields to hold capabilities and other items. Note that each request or reply message can consist of just a header or a header and an additional component.

---

*trans(Msg *requestHeader, char *requestBuffer, int requestSize, Msg *replyHeader, char *replyBuffer, int replySize)*
Client sends a request message and receives a reply; the header contains a capability for the object upon which an operation is being requested.

*get_request(Msg *requestHeader, char *requestBuffer, int requestSize)*
Server gets a request from the port specified in the message header.

*put_reply(Msg *replyHeader, char *replyBuffer, int replySize)*
Server replies.

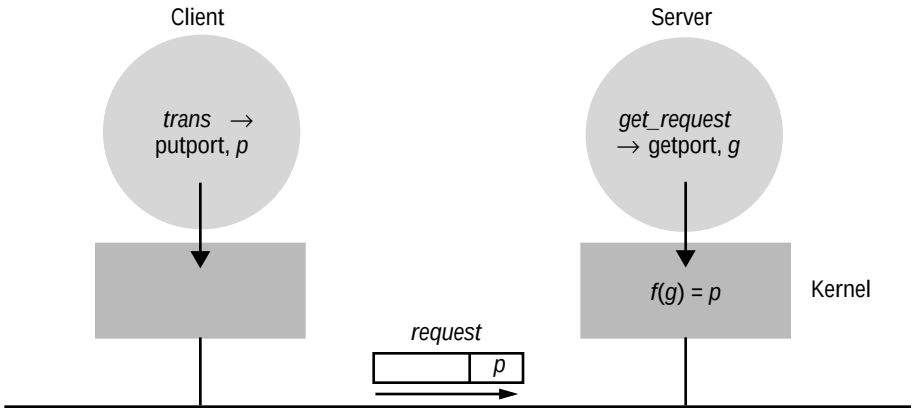
Several threads may receive messages from the same port. Amoeba automatically routes the message sent using *put_reply* to the sender of the corresponding call to *trans*. A thread cannot reply out of order to messages it has received, and must follow every call to *get_request* with a call to *put_reply*.

The advantage of Amoeba's message format is that many requests or replies consist of only a few bytes of data, which can be packed in the header alone, without an extra message component being necessary. The kernel is optimized to pre-allocate buffers of the right size to hold message header data, and to provide a fast path to the network or another local process's message queue for this data. Otherwise, the kernel has to be prepared to allocate a buffer for an arbitrary amount of data on each call.

The Amoeba kernel finds out that a given port is being used when it handles a call to *get_request* that contains the port's identifier. Several servers, for example, several instances of the same service, are allowed to use the same port. A client using the port identifier will reach at most one such server, but there is no explicit means given to the programmer to control which one. However, a server connected to the same LAN as the client will be chosen in preference to any that reside on a neighbouring LAN.

Port identifiers for system services are allocated to servers under the control of system administrators. Where capabilities refer to persistent objects – ones that can outlive the execution of any particular server process that manages them – the same port identifier is used each time a server runs. Otherwise, clients would have to rebuild their capabilities according to the latest server port. For short-term or private communication, however, processes can generate random port identifiers and pass them in messages to the processes with which they need to communicate.

Figure 3 Putports and getports.



A security problem addressed by Amoeba is the possibility of malicious processes being able to masquerade as legitimate servers. Amoeba provides a mechanism to guarantee the authenticity of the server listening on a port.

Amoeba distinguishes between *putports* and *getports* (Figure 3). Clients use ordinary port identifiers called putports. Servers use a related secret identifier called a getport when they call *get_request*. The Amoeba kernel passes this getport through a one-way function, *f*. The result is matched with the putports being used by clients attempting to reach the server. Therefore, only processes that know the getport g such that $f(g) = p$ can service requests sent to putport p. The putport p can be made public but because f is a one-way function, g cannot be determined from knowledge of p: g is a secret which the system administrators will reveal only to those server processes that should know it.

In addition to RPC, version 5 Amoeba added the notion of process groups, and included a reliable multicast delivery service to these groups. A description is available at www.cdk3.net/coordination. This form of communication is designed only for communication between members of a group of servers in order, for example, to implement a fault-tolerant service. Clients are expected to continue to use RPC communication with one of the servers, so that replication is transparent.

Communication implementation

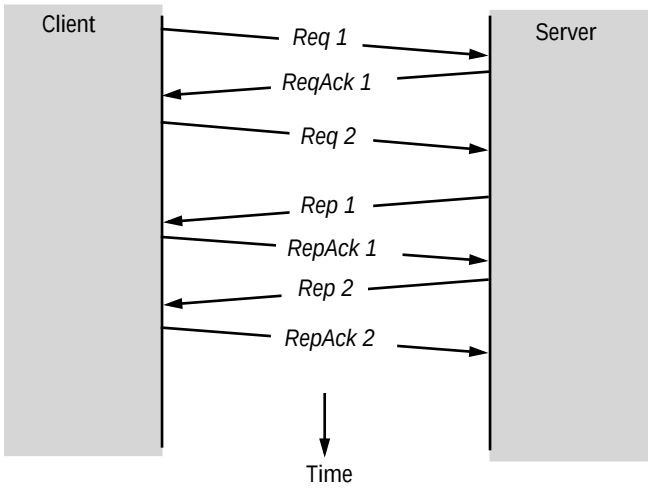
The Amoeba designers decided to include network communication in the kernel as a basic service. They considered the extra context-switching costs that would be incurred through use of a separate, user-level network manager process to be prohibitive.

The kernel is responsible for implementing network communication only over what is described as a *local internet*. This consists of a few LANs interconnected by gateways or bridges. Amoeba RPC has been implemented with considerable attention to its performance, and exhibits some of the best null RPC delay and RPC bandwidth figures that have been measured [van Renesse *et al*. 1989].

Amoeba relies upon additional protocols implemented externally to Amoeba for message transmission over WANs. The request-reply protocol is implemented in two layers: the RPC layer, which implements a Request-Reply protocol that provides at-most-once RPC and the FLIP (Fast Local Internet Protocol) layer – see below. The reason for this division is that, while RPC is intrinsic to the Amoeba design, some additional general communication services were felt to be needed, including group communication, security, support for process migration and operation over connected networks.

The FLIP protocol ◊ The FLIP layer provides a datagram service that transmits messages of up to a gigabyte to destinations called FLIP ports, and deals with the location of FLIP ports. FLIP ports are intermediaries between Amoeba ports and physical addresses. Each process is associated with

Figure 4 Amoeba multi-packet protocol.



a unique FLIP port identifier. Even if two servers use the same port, they will each possess a unique FLIP port.

If a service has several instances and one of them migrates, its clients will attempt to relocate it. Amoeba guarantees that the same instance of the service is located. not one of its peers. This is assured because the FLIP port is specified in the search algorithm. Even though the peer implements the same service, the original server could possess state relevant to its clients' operations, so it is important that the original server continues to service the same clients. Also, it is important that retransmissions of client requests are not picked up at a different server and treated as if they are fresh, when they could have been executed already at a peer that had failed or was migrating.

Request-Reply protocol ◊ The RPC layer implements at-most-once call semantics over FLIP, using the RRA protocol, introduced in Chapter 4. It retries request messages and filters duplicate requests. The RRA protocol acknowledges the reply message, so that the server's data does not need to be retained.

When the user-level request or reply data is too big to fit into a single packet, FLIP uses a multipacket protocol. The service is not reliable, although the protocol does acknowledge all the packets in a multipacket message, except the last. Figure 4 shows a multipacket request message in packets *Req1* and *Req2* and a multipacket reply message in packets *Rep1* and *Rep2*. FLIP sends the client's first request packet *Req1*, then waits for an acknowledgement *ReqAck1* from the server before sending the next request packet, *Req2*. Similarly FLIP sends the server's first reply packet *Rep1*, then waits for an acknowledgement *RepAck1* from the client before sending the next reply packet, *Rep2*.

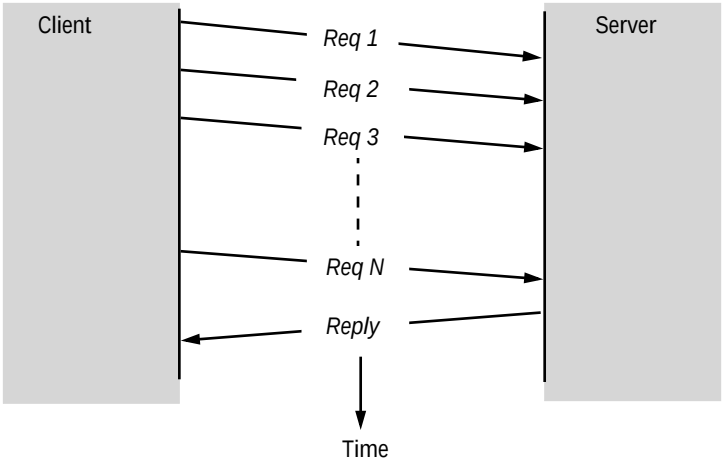
As in the case when both the request message and the reply message fits into a single packet, it is the responsibility of the RPC layer to provide at-most-once call semantics over FLIP's multipacket datagrams.

The overall effect of the RPC layer over FLIP multipackets is very similar to Birrel and Nelson's multipacket protocol, except that Amoeba acknowledges the last reply packet whereas Birrel and Nelson [1984] assume that the next request message will do as an acknowledgement.

Flow control ◊ We turn now to considerations of flow control, by comparing FLIP with another protocol, VMTP. FLIP is an example of a *stop-and-go* protocol: it does not transmit a next packet until the previous one has been acknowledged. By contrast, the VMTP protocol [Cheriton 1986] (developed by David Cheriton, the V system's principal designer, also for request-reply exchanges) is a *burst protocol*: it allows for data belonging to long messages to be sent in bursts of packets called *packet groups* (Figure 5). Instead of acknowledging each packet, VMTP acknowledges packet groups, which can contain up to 16 kilobytes of data.

The advantage of a burst protocol over a stop-and-go protocol is that greater throughput can be achieved, because data can be transmitted concurrently with acknowledgements. The

Figure 5 Sending a packet group in VMTP.



disadvantage is that packets transmitted *back-to-back* – that is, as fast as the sending operating system can send them through its network controller – can sometimes result in the receiving computer dropping packets, necessitating a recovery action. Packets are dropped either because the receiving network controller fails to respond sufficiently quickly after the previous packet to be able to accept a new one, or because the kernel runs out of buffer space for incoming packets. At worst, every other packet might be dropped by a network controller.

There are two ways to alleviate this situation. First, VMTP acknowledges packet groups with a small bit map describing which packets, if any, were not received. Only these packets have to be retransmitted, instead of retransmitting the whole group. This is known as *selective retransmission*. Second, it is possible for the sender to introduce a small delay called an *inter-packet gap* between packets, reducing the tendency to overrun the receiver. The value of this delay can be adjusted dynamically. A sender could increase the delay until acknowledgements tell it that all packets are arriving at the receiver, or until the rate at which packets are dropped has reached an acceptable level.

Timer management ◊ Protocols normally include actions to be taken when timers expire, in order to cope with the non-arrival of expected messages. A timeout has to be arranged for every message that is sent reliably. Setting up a timeout can occur at the beginning of the wait for an acknowledgement, and so does not add directly to client latency; but it does represent a cost to the system as a whole. Additionally, the timeout has to be cancelled in the common case in which an acknowledgement arrives before the timer expires. The kernel therefore has to provide efficient facilities for this purpose.

Amoeba optimizes timeout handling by placing the responsibility for it with a single kernel thread. This thread periodically sweeps a list of protocol data structures, of which there is one for every transaction in progress. If it notices that nothing has occurred since its last sweep, then it initiates a timeout action (that is, usually it retransmits a message). The sweeper thread is run with a low priority to minimize its interference with other system operations, and the times at which timeout actions occur are not very accurate. This is of little significance in general, since the values of timeout periods are only guesses anyway.

Locating ports and FLIP ports ◊ As the Amoeba kernel does not monitor the passage of capabilities in messages, when a client first attempts to use a capability, the port it refers to might not have been seen before at that computer. It is the responsibility of the Amoeba kernel to locate the port. Whenever a process makes a first attempt to receive a message, the Amoeba kernel generates a new FLIP port and associates it with the process at that computer. When a client calls *trans*, the RPC layer looks up a mapping from a port identifier to a FLIP port identifier, in a cache of entries of the following form:

Port	FLIP port	Network address

The network address field is a hint as to the current FLIP port's location (and therefore the port's location).

We shall refer temporarily to ports as 'RPC ports', to distinguish them from FLIP ports. Recall that different client computers may use different servers which employ the same RPC port identifier. The associations between RPC ports and FLIP ports can thus differ between computers. However, the association between FLIP ports and network addresses is unique. If no entry is found for the given RPC port identifier, then a location algorithm is run.

The RPC port location algorithm broadcasts a location message containing the RPC port identifier. The RPC layer in any kernel that hosts a process using the given RPC port identifier will know a FLIP port identifier for a corresponding local process, and will respond with this identifier and the network address. This response packet is used to set up a cache entry for the port at the sender. If several kernels respond with FLIP ports for the given RPC port identifier, then responses after the first are ignored, thus retaining an entry for the computer that responded most quickly.

FLIP broadcasts a message to all destinations within a distance called a *hop count*. The hop count is transmitted in the broadcast message but decremented by one at every gateway computer the message encounters, and not propagated beyond that computer if the hop count reaches zero.

The port location algorithm broadcasts the location message with a hop count of one, which reaches the local internet where hardware broadcast can be used. If there is no response, the hop count is incremented by one and the broadcast attempted again. This is repeated if necessary until either the RPC port is found or the entire local internet has been traversed.

Amoeba does not piggy-back the request data in the port location packet, so the request message has then to be sent to the known address. Piggy-backing the request data would be more efficient, but then the request could be executed at more than one server, which might have undesirable effects.

The RPC layer passes the FLIP port as the address of the datagram it is sending. The FLIP layer consults the cache to determine what network address to use. If the cache entry proves stale, then this is detected on use: the receiving computer will send a negative acknowledgement, telling the sending computer that no such FLIP port is known there. The FLIP layer then has to resort to broadcasting to locate the FLIP port afresh. If the FLIP port is relocated, then the cache is automatically updated.

Discussion of main Amoeba features

In summary, the design of the Amoeba kernel is based on an object-based client-server model, in which as many system services as possible are implemented by user-level processes or groups of these. Amoeba supports this model with a few key abstractions: multi-threaded processes and RPC communication using ports and process groups. Amoeba implements protection by uniformly supporting capabilities for protected access to resources. It has achieved its goal of network transparency, and the majority of its servers execute at user-level. Its optimized RPC implementation is fast by present-day standards. However, implementing RPC in the kernel means that it is provided whether it is required or not; and providing a particular RPC protocol as the only directly supported communication protocol is restrictive.

We have studied Amoeba because it is a consistently designed system based on just a few design principles and goals. However, Amoeba has limitations which might inhibit its general acceptance. First, its assumption that memory will become sufficiently cheap to avoid the need for demand-paged virtual memory seems questionable. The memory requirements of applications (and, equally importantly, useful combinations of applications) are not to be underestimated. The main advantages of virtual memory in the face of these requirements are that it provides a graceful degradation when memory capacity is exceeded, and it allows the programmer to be relatively unconcerned with physical memory limitations. As an example of virtual memory being perceived as a requirement in the 1990s, Windows NT is the operating system kernel developed by Microsoft for workstations and servers to support MS-DOS and other operating environments such as Windows [Custer 1998]. Windows NT includes virtual memory as a distinctive feature that its predecessors lacked.

Secondly, Amoeba does not support binary-level compatibility with any generally-used operating system such as UNIX or DOS, and provides only a partial, library-level emulation of UNIX. This limits severely the current utility of Amoeba. Users are reluctant to give up their software base, unless improved functionality or quality of service justify the migration costs.

One of the problems in emulating UNIX on Amoeba – even at the source code level – is the difficulty that arises when trying to implement UNIX-style *user*/*group*/*other* semantics for file permissions using capabilities alone. These semantics, deriving from access control lists, are administratively much more convenient than having to distribute capabilities. We have already pointed out, for example, that preventing a user Smith from accessing a particular file while continuing to allow Jones to access it is awkward in a capability-based system. Amoeba stores capabilities for files in directories managed by directory servers. How is a directory server to decide which capability (that is, what rights), if any, is to be granted to a particular client when the client presents the file pathname? It would need to know to which of the *user*/*group*/*other* categories the client belonged for the particular file concerned. But this is impossible to determine without proof of the client's user and group identities. Since Amoeba capabilities can be copied freely between processes (and are subject to eavesdropping), they cannot fulfil this purpose by themselves.

The moral of this is that some servers must be free to implement access control lists and to require authentication of clients. Capabilities are not generally applicable. Nor do they provide security unless messages are encrypted and replays are detected, because of eavesdropping (see Chapter 7).

On the other hand, Amoeba has been successful in reminding us of the advantages of simplicity that originally inspired the designers of UNIX. Amoeba's simple microkernel interface is maintainable and relatively easy to understand and conform to.