Wide-Area Communication under Amoeba

*Robbert van Renesse**
Hans van Staveren

Vrije Universiteit
Amsterdam

ABSTRACT

This document describes the earliest Amoeba wide-area gateway service that makes it possible to execute Amoeba transactions over existing long-haul networks. Furthermore, the specification of the interface between network drivers and the gateway is given. Finally, we will propose how to use the gateway service.

1. INTRODUCTION

When given the task of implementing a gateway, we face some non-trivial problems. First, we want to have transparent communication between Amoeba systems in different countries all over Europe as part of a COST-11 project. Second, the Amoeba systems are running on heterogeneous computers, local-area networks, and network layers. For example, there are Amoeba operating systems running on several different processors, and there are Amoeba *layers* running under UNIX† using Berkeley sockets. Byte ordering is also a problem. Third, the wide-area links are far from optimal. Many sites do not have the authority to address the network interfaces directly (they are maintained by the institution, not by the department), the interfaces are different on all sites, and sometimes the only connection between a pair of sites is over several different networks. Fourth, Amoeba ports do not give a clue about the location of servers waiting for requests on that port, making server lookup close to impossible.

To minimize development costs we decided to use whatever software was available and not to change any existing network software. Because all the sites in the initial group of participants had UNIX we decided to implement the initial service by UNIX processes.

2. MODEL

We divide the Amoeba wide-area system into three layers, as shown in Fig. 1.

Amoeba layer	Amoeba clients and servers
Gateway layer	Gateway software
Network layer	Existing wide-area services

* Part of the work was done at the University of Tromsø, Norway.
† UNIX is a trademark of Bell Laboratories.

Fig. 1. Layers in the Amoeba wide-area system.

The highest layer, the Amoeba layer, contains the clients and servers. Clients and servers on the same Amoeba site communicate with each other directly, without using lower layers.

The middle layer, called the *gateway* layer, provides the means for making transparent wide-area transactions possible. This layer contains the new software that had to be written.

The lowest layer, the *network* layer, consists of whatever wide-area communication services are available. It usually provides virtual circuits.

2.1. The Amoeba Layer

The Amoeba layer can differ from site to site, as long as it supports the Amoeba transaction calls. We have made no changes to this layer whatsoever. Each site may have its own local-area network protocol and port location mechanism.

At the time that this document is being written we have two different Amoeba systems. First, we have an Amoeba operating system with links into several kinds of UNIX systems. Second, and most frequently used, there is an Amoeba library that uses UNIX BSD 4.2 sockets to carry out transactions. The gateway runs on both these systems, making communication between them possible.

2.2. The Network Layer

This layer consists of several heterogeneous networks, different in size, quality of service, and interface. Currently we only use X.25 for inter-country communication, usually via X.29 PADs. To reach the PADs, however, we often have to cross several intra- and inter-institute networks and port selectors.

To get from one UNIX machine to another, a program is needed that connects to the ‘‘outside.’’ Then a series of login procedures and network interface commands are needed to log on to the remote UNIX node. Although some, or maybe even all, networks on the path may provide reliable communication, the total line is usually not reliable. For example, PAD to PAD communication is reliable, but the V.24 communication between PAD and computer is not. A simple network protocol running on both nodes is sufficient for obtaining end-to-end reliability.

2.3. The Gateway Layer

The gateway layer functions as the glue between the Amoeba layer and the network layer. On the borders of the gateway layer we have abstracted away from the communication problems in the adjoining layers, so additional network protocols are not needed to achieve reliability.

The function of both adjoining layers is to provide reliable communication between processes. The interface to each layer differs, however. The Amoeba layer provides request/reply exchanges, while the network layer provides virtual circuits. Naming is also radically different. One task of the gateway layer is to convert between the two types of communication.

Another task of the gateway is to extract those requests and replies from the Amoeba layer that have to be sent to remote sites, and, conversely, to insert request and replies received from remote sites. To simplify matters, we use the term *message* as being either a request or a reply. The gateway's task then is to transfer messages to and from Amoeba systems by sending them over virtual circuits between the Amoeba sites.

3. DESIGN

This section only discusses the design of the gateway layer, since the other two layers were designed and implemented a long time ago, and we use them as they are.

The gateway layer consists of a gateway server at each Amoeba site. Since it has to communicate with Amoeba layer processes, it runs as an ordinary Amoeba server. Since it has to communicate with remote gateway servers over virtual circuits, it needs to be able to set up and close down connections, and to send and receive data over the connections.

A gateway server consists of four parts, as shown in Fig. 2.

boxwid=2i boxht=0.5i A: box "Network agent" B: box with .sw at A.sw + (0, 1i) "Gateway switch" C: box with .se at B.nw + (0.25i, 0.5i) "Client agent" D: box with .sw at B.ne + (-0.25i, 0.5i) "Server agent" arrow from D.s to B.ne + (-0.75i, 0) arrow from B.nw + (0.75i, 0) to C.s line <-> from A.n to B.s

Fig. 2. Modules in the gateway server.

The server agent accepts requests from local clients and sends them to the gateway switch. Ultimately the gateway switch will send a reply back to the server agent, which will subsequently send it to the client. The server agent plays the role of the remote server to the client.

The client agent does the reverse. It waits for a request message from the gateway switch, does a local transaction to send the request to the appropriate server, and sends the reply back to the gateway switch. The client agent acts like the remote client to the server.

The network agent transfers messages between the gateway switch and the wide-area network.

The gateway switch decides where messages should go. When it gets a request message, it looks up the destination port in a table internal to the gateway server. If an entry for this port exists, it uses the address information therein to send the message to the appropriate remote site. If there is no entry for the port, it sends the request to the client agent, which will try to find a local server for the request. The request messages are tagged with the source address, to allow the replies to be routed to the originator of the request.

Reply messages are routed using their address tag, and can be sent by the gateway switch to either the network agent, or the server agent. The gateway switch can also generate reply messages when a wide-area connections fails. In that case it sends an error reply message back for every outstanding request on that (former) connection.

4. IMPLEMENTATION

Since the gateway server needs to be able to handle multiple clients and servers, drive multiple wide-area networks, and run the gateway switch, the server needs multiple Amoeba tasks. As mentioned earlier, the initial gateway servers run on UNIX systems supporting the Amoeba transaction interface. These UNIX interfaces do not allow tasks or shared memory. The real gateways, running on Amoeba will have these features.

Our implementation therefore simulates multiple tasks by having a UNIX process for each Amoeba task, and shared memory using UNIX pipes. To minimize conflicts (like waiting for more than one event), all messages to the gateway switch are sent using transactions, whereas messages *from* the gateway switch are sent over pipes. This reduces the conflicts to only one: sometimes a process in the network agent is interested in messages from both the gateway and a virtual circuit at the same time. This conflict is resolved by having the process regularly poll the pipe from the gateway switch.

The UNIX process structure is as follows. There is one process for each Amoeba task. The gateway switch is the overall parent process, and forks off the processes in the different agents of the gateway server. These child processes are called *stubs*. There is one pipe from the gateway switch process to each stub.

boxwid=1.5i boxht=0.5i A: box "Network stub" B: box with .se at A.se + (0, 1i) "Switch" C: box with .se at B.se + (0, 1i) "Client stub" D: box with .se at C.se + (0, 1i) "Client" E: box with .sw at A.se + (2i, 0) "Network stub" F: box with .se at E.se + (0, 1i) "Switch" G: box with .se at F.se + (0, 1i) "Server stub" H: box with .se at G.se + (0, 1i) "Server" arrow from D.s to C.n arrow from C.s to B.n arrow from B.s to A.n line from A.s down 1i line from A.s + (0, -1i) to E.s + (0, -1i) arrow from E.s + (0, -1i) to E.s arrow from E.n to F.s arrow from F.n to G.s arrow from G.n to H.s line dotted from D.s + (-1i, -0.25i) to H.s + (1i, -0.25i) line dotted from A.s + (-1i, -0.25i) to E.s + (1i, -0.25i) boxwid=2i box invis with .sw at D.se "Amoeba layer" box invis with .sw at B.se "Gateway layer" box invis with .sw at A.se + (0, -0.75i) "Network layer"

Fig. 3. How request messages travel through the system.

Messages are made by concatenating the Amoeba header and buffer, and prepending a small header containing the message type (request or reply), and an address tag used for routing replies (see Fig. 4(a)). A request message travels from client to server as shown in

Fig. 3. boxwid=1.5i boxht=0.25i A: box ht 1i "Amoeba buffer" "(Up to 2K bytes)" B: box ht 0.5i with .sw at A.nw "Amoeba header" "(32 bytes)" C: box with .sw at B.nw "Reserved" D: box with .sw at C.nw "Reserved" E: box with .sw at D.nw "Address tag" F: box with .sw at E.nw "Message type" G: box ht 1.5i with .sw at A.se + (2i, 0) "Message" H: box with .sw at G.nw "Low byte of size" I: box with .sw at H.nw "High byte of size" J: box with .sw at I.nw "Reserved" K: box with .sw at J.nw "Buffer type" "(a)" at A.s + (0, -0.1i) below "(b)" at G.s + (0, -0.1i) below

Fig. 4. (a) Amoeba request or reply message. (b) Buffer structure.

4.1. Communication Over Pipes

Communication over a pipe is as follows. First the gateway switch creates a capability for using the switch, and writes it onto the pipe. The capability identifies the stub to the switch when the stub later sends requests using this capability. All further communication is via *buffers* . The buffer structure is depicted in Fig. (b).

A buffer consists of a header and a message. The header contains the type and the size of the message. There are three types:

OPEN: open a virtual circuit. This message can only be sent to a network stub, requesting it to open a connection. The address is in the message.

CLOSE: close a virtual circuit. This requests a network stub to close the current connection.

DATA: the message is either a request or a reply. A server stub sends it to its client, a client stub sends it to some server, and a network stub sends it over its current connection.

The stub is required to send an acknowledgement for each of the buffers it receives, to prevent the switch from being blocked on a full pipe.

4.2. The Gateway Switch

The gateway switch is a true server process waiting for and executing requests. Its main task is to send messages, received in the requests, to the right stubs over the pipes. Furthermore, it manages the connections with remote gateways by requesting network stubs to set up or close virtual circuits. An additional feature is an error recovery scheme for broken connections.

4.2.1. Requests to the Switch

An Amoeba request consists of a request header and a request buffer. The contents of these for the different types of requests to the gateway switch are discussed in this section.

SEND: The request buffer contains either a request or a reply message from some stub. The switch has to route the message to the correct destination stub.

DONE: This is an acknowledgement for a buffer that was sent over a pipe. The switch can now send another buffer over the pipe without danger of blocking on a full pipe.

FAILED: Something went wrong at the stub that sent this request. In the current implementation this can only be a broken virtual circuit.

REMCONN: A remote gateway seeks contact with the local gateway. The buffer contains information about what type of network stub to use for communication; there is a different network stub for each type of network service. The switch is supposed to send an OPEN request to this stub.

INSTALL: Install a remote port in the table of ports. The address of the port is in the request buffer. The switch has to fork off a server stub to collect the local requests on that port.

CONNECT: Add a new network stub to the gateway server. The type of wide-area network is in the request buffer. The switch has to fork off a process for this stub.

SWEEP: Virtual circuits are closed by the gateway switch when they have not been used for a while. The sweep request is sent periodically to the gateway switch, which then scans its table of network stubs to see whether the connections there are being used.

SHUTDOWN: This requests the switch to ‘‘gracefully’’ stop all the gateway server activities. The switch sends a signal to all the stubs and exit itself.

The gateway switch acknowledges the requests to the sender by sending a reply. This reply may contain an error status if the switch was not able to execute the request.

4.2.2. Routing Messages

When a request message arrives at the gateway switch, it is routed to the proper stub by comparing the port in the header of the request to each port in the internal port table. If the port is there, the message is transferred to a network stub. The address in the table determines what type of network stub is to be used, and the network address on the corresponding network. If no entry for the port is found, the request message is routed to the client stub, which will try to find a server on the local network.

Associated with each stub is a message queue. When a virtual circuit is opened or a DONE message for the stub arrives, the switch scans this queue to see if there are messages to be transmitted to the stub. Each entry in the queue is uniquely identified by a small integer, and contains the address of the sender of the associated message. When a request message is sent over the virtual circuit, it is tagged with this identifier. The associated reply message will contain the same tag.

When the reply message arrives at the switch, it locates the queue entry, and use the address therein to route the message to the proper recipient. At the same time, the queue entry is removed. Queue entries for reply messages are deleted as soon as the message has been transferred to the stub.

The address information in the queue entries consists of three fields: the stub where the message arrived from, the queue identifier in that message, and an *incarnation number*, that is used to detect and recover from broken virtual circuits.

Virtual circuits are reliable, in the sense that they transfer data without damaging the contents or the order in which the data was sent. However, virtual circuits can break for reasons like network congestion, or a crash of the remote host. The gateway can then reuse the stub to set up a new connection, possibly to another destination. To differentiate between the connections, there is an *incarnation number* associated with each stub, which is incremented with every new connection. The incarnation number in the queue entries can be used to prevent replies being routed to wrong destinations; if the incarnation numbers do not match, the message is simply discarded.

The queue can be used to generate error reply messages to the clients when a virtual circuit breaks. When a connection breaks, the stub sends a FAILED request to the gateway switch, which then generates replies for every request entry in the queue of the stub.

Saving the queue identifiers of requests in the queue for the destination stub permits requests and replies to be routed through several gateways. The port table in the gateway contains the address of the remote host that is only one hop away; the gateway there decides how to route request messages further. For reply messages there is enough information in the queue to route them back to the clients. Thus arbitrary network topologies are supported by this scheme.

4.2.3. Making connections

When a request message arrives at the gateway for a remote site for which no virtual circuit exists, a connection has to be set up. The gateway switch takes a free network stub and sends an OPEN request to this stub, which then makes the connection. Meanwhile, messages that arrive for this stub are queued.

The site that receives the call sends a REMCONN request to the gateway switch at that site, to tell it that somebody is trying to make contact. The switch then sends a special OPEN request to a free network stub to tell it to make the connection. The address used in the OPEN request is obtained from the REMCONN request to the switch.

When connections have not been used for a while, the switch will break the virtual circuit by sending a CLOSE request to the network stub. A connection is defined to be *idle* when there are no messages left to be sent over it, *and* there are no outstanding transactions, either to the remote site or from the remote site. There is an outstanding transaction to a remote site when the reply from that site has not yet been received. There is an outstanding transaction from a remote site when a reply for the remote request has not yet been sent.

The switch can test whether a connection is idle by checking whether the queue is empty, and whether the number of incoming requests minus outgoing replies is zero. At regular intervals the switch increments a counter variable associated with the connection when the idle test succeeds. When this variable exceeds some critical value, the switch will break the connection. The variable is reset whenever a message is sent over the connection.

The times at which the switch tests whether a connection is idle are determined by the arrival of sweep requests. These are sent at regular intervals by a special process in the gateway, called the *sweep* stub.

4.3. The Server Stub

A server stub is a server process that accepts requests for a remote server. There is at least one stub per remote server, because the Amoeba primitives do not allow specifying more than one port. Soon this may change however (see the Saper called "Connecting

Amoebas Using Gateways"). More that one stub may exist for one remote server to allow for multiple clients using the same remote structure.

Server stubs are created by the gateway switch when the switch receives an INSTALL request. A capability for the gateway switch is sent over the pipe between the stub and the switch to enable the stub to send requests to the switch. Now the server stub goes through a loop as depicted in Fig. 5, getting requests, processing them, and sending back replies.

```
from_switch(&cap);
for ever
do
   getreq(&request);
   to_big_endian(&request);
   to_switch(&cap, SEND, &request);
   from_switch(&reply);
   to_switch(&cap, DONE, NIL);
   from_big_endian(&reply);
   putrep(&reply);
done
```

Fig. 5. Operation of a Server Stub.

Before a request message is sent to the gateway switch, the stub converts the Amoeba header in the message to a defined word order, namely *big endian* (lowest numbered byte is most significant). Similarly, when it gets a reply message from the switch, it converts the header to the native byte order before it is sent to the client. This way the wide-area network byte order is defined, and there are no byte order problems in the Amoeba headers, like it is in the local case. Of course, this does not solve the byte order problem in the data buffer of the messages, but fortunately these are usually ascii strings.

4.4. The Client Stub

The client stub gets request messages over the pipe from the gateway switch, and does transactions using these requests, as shown in Fig. 6. Like the server stub it converts messages to and from big endian to solve the byte order problem for Amoeba headers. Furthermore, it automatically generates an error reply message when a transaction fails to notify the remote client.

```
from_switch(&cap);
for ever
do
   from_switch(&request);
   to_switch(&cap, DONE, NIL);
   if (fork() == CHILD)
   then
      from_big_endian(&request);
      if (trans(&request, &reply) == FAIL)
```

```
        then
           reply.status = ERROR;
        fi
        to_big_endian(&reply);
        to_switch(&cap, SEND, &reply)
   fi
done
```

Fig. 6. Operation of a Client Stub.

Although, as far as the gateway switch is concerned, there is only one client stub, this stub consists of several processes enabling multiple simultaneous transactions. For this purpose, the client stub forks off a process for every transaction it has to perform.

4.5. The Network Stub

This stub manages the connection over wide-area networks. The gateway allows several types of network stubs, one per type of network service. A difficulty in implementing these stubs is that messages come from two sides, the gateway switch and the network, and the stub might want to wait for both sides at the same time.

```
from_switch(&cap);
for ever
do
   from_anywhere(&request);
   switch (request.type)
   case OPEN:  open_connection(request);
   case CLOSE: close_connection();
   case DATA:  if (request.source == NETWORK)
                  to_switch(&cap, SEND, &request);
               else    /* source == SWITCH */
                  to_switch(&cap, DONE, NIL);
                  to_network(&request);
               fi
   end_switch
done
```

Fig. 7. Operation of a Network Stub.

The task of the network stub is mainly to pass messages between gateway switch and wide-area network, as shown in Fig. 7. Additional tasks are setting up and closing down connections. The different types of stubs originate from the different network driver libraries used by the stubs.

5. USAGE

Now we have designed and implemented a gateway service for wide-area transactions, we will describe how the service can be used to build a wide-area distributed system. This section consists of two parts, one suggesting how to do the management of the gateway servers, the other giving advice on the implementation of global services.

5.1. Management

The gateway service as described offers a raw, low level service, that higher level services can use in building a user-friendly basis for making a wide-area distributed system. The high-level services we identify in this section are port distribution, monitoring, and accounting.

5.1.1. Port Distribution

On local Amoeba systems, clients locate services by broadcasting locate requests over the local network. The node that is running the service will respond with its node address. Clearly, this scheme is not possible on the scale of international networks, for they do not provide an efficient, cheap, broadcasting service. Even if it would, the enormous amounts of global locate requests would increase the load on all the machines on the network.

Therefore a different port location method has to be used on wide-area networks. Our gateway service allows the registration of remote ports on local gateways. Thus if a client is locating a known remote port, the gateway answers on behalf of the remote server. We would like to have a service for making ports ‘‘known’’ at the sites where the service is to be offered. Such a collection of sites is called a *domain*, and making a port known is called *publishing*.

This service could be implemented by having a *port publishing* server running at each Amoeba site, each one with its own unique port that is known at every other site. Each port publishing server accepts PUBLISH and INSTALL requests:

PUBLISH: A local Amoeba server wants to have its port installed at all the gateways in the specified domain.

INSTALL: A remote port publishing server wants to install a port at the local gateway.

When an Amoeba server wants to make itself known in a domain, it send a PUBLISH request to the local port publishing server, which will send INSTALL requests to each port publishing server in the specified domain.

A disadvantage of this method is that every port publishing server in the known domains has to be registered at the gateway service. Furthermore, creating, changing, and deleting domains is difficult, because the domains are large and the management is distributed over the complete wide-area network.

This can be resolved by having separate *domain managers*. The ports of the domain managers are known at every site. PUBLISH requests are sent to the appropriate domain manager, which will send install requests to all the sites in the domain it manages. This scheme can easily be made hierarchical, to allow large domains being split up into smaller ones, making the management of domains more efficient and flexible.

5.1.2. Monitoring

An important feature of the gateway server is the generation of logs of its operation. The amount of logging can be changed by setting the *log level* of the server: the higher the log level, the more events are logged. The gateway logs arrived and sent messages, connection set up, errors, and, if the log level is high enough, the low level network protocol.

An interesting service for the management of a wide-area distributed system is the automatic collection of the log files from the different gateways, filtering out the information that is actually required. Such information could be used to find bottlenecks in the network, determine the availability and efficiency of wide-area Amoeba services, or find bugs in the system.

There are several approaches to setting up such a service. For example, it could consist of monitoring servers at each site, accepting requests to give the logging information according to some conditions, like: ‘‘Give me the number of messages sent to server X during the last week.’’ Finding what conditions should be supported in these queries will make an interesting research project.

5.1.3. Accounting

A wide-area connection is expensive. So are processing cycles and resources. Many managers of Amoeba sites would like to know which local users use the network, and which remote sites are using local resources. Furthermore, they would like to know which local users are responsible for the bills the Amoeba site gets from remote sites, for using their resources. What is needed is an accounting service that keeps track of who used which remote resources, and the amount of network traffic they used for accessing the resources.

There are two principal ways of setting up such a service: it could keep records of who has done what, and send the bills afterwards, or it could charge the users before doing the requested operations. Then there is the problem of whom is to be charged for a transaction: the client or the server. Again, this has to be divided into communication and resource costs.

However the users are charged, the service has to authenticate the users first. This can be done using the digital signature field in the Amoeba header; the capability for objects give no clue about the identity of clients. Since it is impossible to make each user known at each site, gateways should act as one client to the remote gateways. The local gateway could then keep track of the local users that are responsible for the communication and service costs.

5.2. Wide-Area Services

The gateway should be a non-service, completely transparent to Amoeba users. However, we were not able to do this for several reasons:

- automatic port location is inefficient in a wide-area network;
- To keep the total buffer size in our system down, and because of limitations in UNIX pipes, we had to limit the Amoeba message buffers to 2048 bytes;
- wide-area communication is expensive and should be accounted for;
- the usage of remote resources should be accounted for.

In this section we propose how to use the gateway service to obtain optimal performance. First, however, we identify some wide-area applications:

- mail service;
- file transfer;
- remote execution;
- distributed database;
- banking system;
- airline reservation system;
- weather forecast;
- distributed processing.

This last point may require some extra attention.

Distributed processing is a feasible application, as long as the communication overhead is low compared to the processing. In wide-area networks this means that it is only worthwhile to execute a process at a remote site when it will run there for a while without exchanging messages with the local site. For example, this might happen if there are several independent jobs that require considerable computation.

In Amoeba systems it is not unusual for applications to generate new ports on the fly. On local Amoeba systems this does not present any difficulties, because ports can be found by broadcasting locate requests. Using our gateway service, however, it is more difficult. A newly generated port will have to be published in the domain where it is used, generating considerable overhead and consuming many resources (a server stub at each site). Furthermore, the lifetime of such a port is usually short, and it will have to be removed when it is not used any more.

The number of published ports should be kept down to a minimum for the same reason. This means that generally the wide-area services have to be more powerful than in the local case, offering more operations on a larger variety of objects. For example, if the mail service, file transfer service, and the remote execution service are merged into one larger service, it consume only one server stub at each site, instead of three.

Since the gateway server keeps a virtual circuit open as long as there are outstanding transactions over the connection, it is undesirable to have long transactions. First, because one has to pay for connections even if they are not used, and second, because they consume network resources that can be used for other transactions. Service time should therefore be kept low, that is, less than a few minutes, which is the usual time for unused connections to be closed.

6. CONCLUSIONS

Wide-area transactions are feasible, and need not be difficult to implement. We did so using existing local and wide area network services, by implementing a small message switching service. We suggested how the service can be made into a high-level service by adding port distribution mechanisms. Furthermore, by adding monitoring and accounting the wide-area Amoeba service can be made into a manageable system.

7. ACKNOWLEDGEMENTS

We would like to thank Sape Mullender for suggesting a high-level gateway service, and Jennifer Steiner for the careful reading of our document.