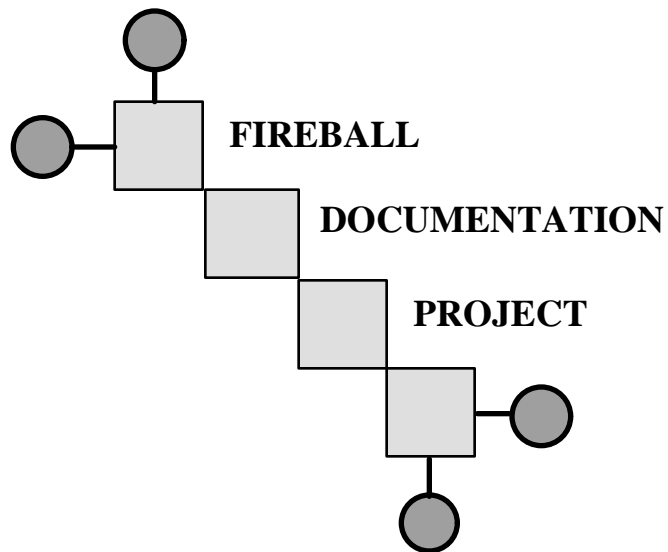


Interprocess-Communication Manual
Client and Server Programming
Fireball-Amoeba 5.6.0

Date: Sun Dec 17 13:48:33 GMT 2000



Client/Server Tutorial

Writing Servers and Clients

Servers are used to perform specialized functions and form the basis of centralized object management. There are many standard servers delivered with Amoeba. These include the Bullet File Server, the Soap Directory Server, the Virtual Disk Server and the Run Server (which does load balancing at process creation-time), among many others. To develop basic applications and port utilities from other operating systems the standard servers are usually adequate. However for some applications this is not enough. This section gives an indication of how to write your own servers and clients and points out some of the pitfalls to avoid.

Servers normally have a fixed set of commands that they accept. They sit in a loop accepting commands, executing them and sending a reply. To simplify the client's interaction with a server it calls a procedure, known as a *stub*. This packages the command and data in the manner required by the server and sends it to the server using a remote procedure call (RPC) and then awaits a reply. The procedure also unpacks any data in the reply and hands it back to the client. The stub allows details of the server interface and the possible byte-order differences between client and server's processors to be hidden from the programmer. Below is a description of how to write servers, their stubs and the clients that use them.

The following description of how to write clients and servers begins by showing what the C code actually looks like for a client/server interface. The reason for this is purely educational. In fact it is seldom necessary to write the server loop and client stub code yourself. AIL (the Amoeba Interface Language) can be used to generate this code automatically. The implementation of the commands must be written by the programmer of course. One advantage of this is that if the RPC interface ever changes, then recompiling the client/server interface with AIL will probably be sufficient to port the server to the new RPC interface. Another advantage is that it is in principle easier to write a specification of the interface in AIL than to write the code for every interface stub.

Before writing servers and clients it is important to understand the system of F-boxes described in many of the papers about Amoeba. These have been implemented in software. The idea of the F-box is that it prevents someone starting a server which intercepts RPCs intended for some other (important) server by deliberately listening to the same port.

The F-box mechanism works as follows. When a *getreq* call is done it listens to the *get-port* for the server. This is a port known only to the server. The server's kernel passes this port through the routine *priv2pub(L)* to encrypt the port. The result is known as the *put-port* and the kernel accepts requests sent to the *put-port*. *Priv2pub* is a one-way function so it is practically impossible to deduce the *get-port* given the *put-port*.

Before doing a *getreq* the server needs to publish the *put-port* through which it can be found by clients. Therefore it also passes the *get-port* through *priv2pub* and publishes the result in the directory server in a place accessible to its clients. In principle it is not possible to listen for requests on the public port unless the *get-port* is known.

This actually provides very little security if it is possible for someone to boot modified Amoeba kernels on your network. They can add a primitive for listening to *put-ports* or modify a kernel binary to avoid the conversion of *get-port* to *put-port* when *getreq* is called. The same deficiency occurs if the F-box is implemented in hardware and it is possible to connect to the network without an intervening F-box.

Servers

Most servers have the same basic structure, although exceptions do exist. Each has a main program which initializes any global variables, publishes the port of the server so that clients can use it and starts one or more instances of the server loop. The server loop is typically an infinite loop with a *getreq* call at the top, a switch on the command to be executed and a *putrep* call at the bottom. If the server is multithreaded then resource locking may also take place within the infinite loop as well. It is a good idea to be able to see the matching pairs of lock/unlock and *getreq* and *putrep*. Unmatched pairs can be difficult to debug so it is better to keep them in the same function wherever possible.

If a server is expected to have more than one client then it is important to make it multithreaded. This allows the server to take advantage of any possibility for parallelism in handling requests. As soon as one thread gets a request, another will be ready to run with the next RPC that arrives. It also means that there is a high probability that the server is listening to its port and so locating of the server will not fail. Each thread in the server that handles clients will normally run the same code. Of course, there may well be several other threads inaccessible to clients performing different tasks within the server.

The best way to understand the structure of a server is by looking at one. Below is an example of a server written in C. The example is of a trivial server which when sent two long integers reverses their order in the message and sends them back to the client. This is a ridiculous thing for a server to do since it is trivially done in the application itself but it does demonstrate the important aspects of writing servers. Note that some unimportant details have been left out to avoid obscuring the structure.

We begin with the include file *this_server.h* which is particular to this server. It defines the request buffer size `REQ_BUFSZ`, the command codes for the commands that the server accepts, the relevant rights bits for the server and the number of threads that the server should run. They are in an include file since some of this information is also needed by the client or for tuning.

Note that it is not acceptable to use just any command codes. You must begin at the value `UNREGISTERED_FIRST_COM` defined in the file *cmdreg.h* as explained in the chapter *Paradigms and Implementations*.

```
/* the include file this_server.h */
#define NTHREADS      5
#define STACKSZ      0x4000
#define SERVER_CAP    "/home/this_server/default"
#define REQ_BUFSZ    (2*sizeof(long))

/* commands accepted by the server */
#define DO_SWAP      UNREGISTERED_FIRST_COM
#define DO_NOTHING   (DO_SWAP+1)

/* rights bits in capabilities */
#define RGT_DONOTHING ((rights_bits)0x01)
#define RGT_DOSWAP    ((rights_bits)0x02)
```

Before we look at how the server loop works it is worth knowing how to invoke multiple threads all running the server loop. This is done using the thread interface routines (see *thread(L)*). The use of *priv2pub* is also demonstrated.

```
#include "amoeba.h"
```

```
#include "stderr.h"
#include "module/rnd.h"
#include "module/name.h"
#include "this_server.h"
void server_loop();
capability get_cap;
port check_field;
main()
{
    capability put_cap;
    errstat err;
    int i;

    /* initialize the server's get capability */
    uniqport(&get_cap.cap_port);
    uniqport(&check_field);
    if (prv_encode(&get_cap.cap_priv, (objnum) 0,
                  (rights_bits) 0xFF, &check_field) != 0) {
        printf("Could not initialize capability\n");
        exit(1);
    }
    priv2pub(&get_cap.cap_port, &put_cap.cap_port);
    put_cap.cap_priv = get_cap.cap_priv;

    /* publish put capability in directory server */
    if ((err = name_append(SERVER_CAP, &put_cap)) != STD_OK) {
        printf("Cannot append to %s: %s\n", SERVER_CAP,
              err_why(err));
        exit(1);
    }
    /* start the NTHREADS server_loop threads */
    for (i = 0; i < NTHREADS-1; i++)
        if (!thread_newthread(server_loop,
                              STACKSZ, (char *) 0, 0))
            printf("Cannot start server thread\n");
    server_loop();
    /*NOTREACHED*/
}
}
```

Note that if *main* exits then the process will die. Therefore it is important that *main* either goes into an infinite loop (in this case by also executing the server loop) or goes to sleep (for example, on a mutex).

In the *server_loop* routine it is assumed that the initialization of global variables has taken place in the main program which started it. In particular the port listened to by the server should have been initialized and published under the name defined by *SERVER_CAP* in the include file *this_server.h*. The directory where this capability is to be found must be writable by the person starting the server. Otherwise the new capability cannot be registered.

```
#include "amoeba.h"
#include "cmdreg.h"
#include "stderr.h"
#include "this_server.h"
extern capability get_cap;
extern errstat do_swap(), do_nothing();
/*ARGSUSED*/
void
server_loop(param, size)
```

```
char * param;
int size;
{
    header hdr;
    char reqbuf[REQ_BUFSZ], repbuf[REQ_BUFSZ];
    bufsize n;
    for (;;) {
        hdr.h_port = get_cap.cap_port;
        n = getreq(&hdr, reqbuf, REQ_BUFSZ);
        if (ERR_STATUS(n)) {
            printf("getreq failed: %s\n", err_why((errstat) n));
            exit(1);
        }
        switch (hdr.h_command) {
        case DO_SWAP:
            hdr.h_status = do_swap(&hdr, reqbuf, n, repbuf);
            break;
        case DO_NOTHING:
            hdr.h_status = do_nothing(&hdr, reqbuf, n, repbuf);
            break;
        default:
            hdr.h_status = STD_COMBAD;
            hdr.h_size = 0;
            break;
        }
        putrep(&hdr, repbuf, hdr.h_size);
    }
}
```

The routine *err_why* (see *error(L)*) produces a human readable error message for standard error codes. For unknown error codes it merely returns the string

```
amoeba error nn
```

where *nn* is the error code returned. Note that *getreq* takes a *get-port* as parameter in the *hdr* variable. However when it returns the *hdr* contains the *put-port* and other fields sent by the client so it is important to reinitialize *hdr* before each call to *getreq*.

The routine *do_swap* will check the capability in the header, take the *n* bytes of data in the request buffer and perform the command specified by *DO_SWAP*. It fills the reply buffer to be returned to the client and returns the status of the command. It returns the status for clarity. If every entry in the switch sets the return status of the command it is easy to check that all pathways return a status. Other arrangements are also possible.

Now we shall consider the structure of the routine *do_swap*. The structure of the data that it expects in the request buffer determines what data the client must send to it. Since the server should also provide the client programs with stub routines to communicate with the server, the routine *do_swap* will largely determine the client stubs. However, we first need a way of validating capabilities. Below is an example routine. Note that it is important to distinguish between a bad capability and a capability with insufficient rights when generating an error code.

```
#include "amoeba.h"
```

```
#include "cmdreg.h"
#include "stderr.h"
#include "module/prv.h"
extern port check_field;
errstat
check_cap(priv, required_rights)
private * priv;
rights_bits required_rights;
{
    rights_bits rights;
    if (prv_number(priv) != 0 ||
        prv_decode(priv, &rights, &check_field) != 0)
        return STD_CAPBAD;
    if ((rights & required_rights) != required_rights)
        return STD_DENIED;
    return STD_OK;
}
```

In this example the server has only one capability of interest, namely the capability for the server itself. There are no object capabilities. Note well that the capability is checked against the original *check_field* and not against the *get_cap*. If you have more than one object then the original check field for each object should be stored with the per-object information and not the *get* or *put* capability. The *check_field* can be used to generate the capability, but the converse may not be true. The *prv(L)* routines should always be used to check capabilities to protect programs from future changes to data structures.

```
#include "amoeba.h"
#include "cmdreg.h"
#include "stderr.h"
#include "module/buffers.h"
#include "this_server.h"
errstat check_cap();
errstat
do_swap(hdr, req, reqsz, rep)
header * hdr;
char * req;
bufsize reqsz;
char * rep;
{
    char * p;
    char * q;
    long vall, val2;
    errstat err;
    hdr->h_size = 0; /* in case of error */
    if ((err = check_cap(&hdr->h_priv, RGT_DOSWAP)) != STD_OK)
        return err;
    q = req;
    p = rep;

    /* get the two longs from the request buffer */
    q = buf_get_long(q, req + reqsz, &vall);
    q = buf_get_long(q, req + reqsz, &val2);

    /* if q is 0 then the request buffer did not contain two longs! */
    if (q == 0)
        return STD_ARGBAD;
}
```

```
/* now put them into the reply buffer in reverse order */
p = buf_put_long(p, rep + REQ_BUFSZ, val2);
p = buf_put_long(p, rep + REQ_BUFSZ, val1);

/* if p is 0 then the buffer of the server is too small */
if (p == 0)
    return STD_SYSERR;
hdr->h_size = p - rep;
return STD_OK;
}
```

The routine *do_swap* uses the *buf_get* and *buf_put* routines (see *buffer(L)*) to receive and send data. These ensure that no byte-order problems occur between client and server. The stub routines should also use the corresponding routines. The *buf_get* and *buf_put* routines are very pleasant to use since it is not necessary to check for errors until the last operation is completed. They check for buffer overflow and thus prevent overrunning array bounds. They return the next free position in the buffer if they succeed and the null pointer if they fail. If given a null pointer as a buffer argument then they immediately return a null pointer. Thus if any of the calls returns a null pointer then all subsequent calls will also do so and the error test need only be done once at the end of a series of *get* or *put* calls.

In this example the server expects two longs in the request buffer. It swaps them and inserts the result into the reply buffer using *buf_put_long* and returns the total size of the reply buffer to the *server_loop* which sends the reply.

Clients

It is now time to consider what is needed on the client side. It is possible to write a client program which knows all about the data formats that the server expects. However it is almost certainly simpler to provide the client with single routine to call which hides the details of marshaling the data and sending it to the server. One reason for doing this is that the implementation of the RPC is then hidden from the client program and any of the planned changes to that interface need only be modified in a few stub routines and the programs should then continue to function after recompilation. If AIL is used then recompilation of the sources should be sufficient.

A stub routine for *DO_SWAP* might look like the following.

```
#include "amoeba.h"
#include "cmdreg.h"
#include "stderr.h"
#include "module/buffers.h"
#include "this_server.h"
errstat
swap_longs(svr, val1, val2)
capability * svr;
long * val1;
long * val2;
{
    header hdr;
    bufsize t;
    char buf[REQ_BUFSZ];
    char * p;
    hdr.h_port = svr->cap_port;
    hdr.h_priv = svr->cap_priv;
    hdr.h_command = DO_SWAP;
    p = buf_put_long(buf, buf + REQ_BUFSZ, *val1);
```

```
p = buf_put_long(p, buf + REQ_BUFSZ, *val2);
if (p == 0) /* REQ_BUFSZ is too small! */
    return STD_SYSERR;
t = trans(&hdr, buf, REQ_BUFSZ, &hdr, buf, REQ_BUFSZ);
if (ERR_STATUS(t))
    return ERR_CONVERT(t);
if (ERR_STATUS(hdr.h_status))
    return ERR_CONVERT(hdr.h_status);
p = buf_get_long(buf, buf + t, val1);
p = buf_get_long(p, buf + t, val2);
if (p == 0) /* server returned garbage */
    return STD_SYSERR;
return STD_OK;
}
```

In general it is important that stub routines return an error status. This makes it much simpler for the client to deduce the cause of errors. Returning a null pointer or something similar on failure complicates things and a single *errno*-like variable is complicated and cumbersome in a multithreaded process.

Note once again the use of the *buf_put* and *buf_get* routines for marshaling data. There are routines for various data types. It is important that the server and the stub marshal the data in the same way or the results may be unfortunate.

In this example we used the same buffer to send data as to receive it. Similarly we used the same *header* struct to send and receive information. This is perfectly acceptable practice but the following should be noted. The fields *h_command* and *h_status* occupy the same position in the header struct. (The field name *h_status* is actually a `#define`.) Therefore if a stub has a loop of RPCs rather than a single RPC then the *h_command* field must be reinitialized each time around the loop.

There are some fields in the *header* struct which are intended for small amounts of out of band data. These could have been used to send and receive small pieces of data. In this example we have two longs and they do not both fit into the header. All byte swapping of header fields is done automatically by the kernel and if only one or two small integers need to be sent then it is better to use the header and give the parameter `NILBUF` and buffer size zero (0) since this will lead to greater efficiency in the marshaling of data.

The macros `ERR_STATUS` and `ERR_CONVERT` are also important. In the current version of Amoeba the error status returned by *trans* or in *hdr:h_status* is an unsigned 16-bit integer. It is intended that this be changed to a signed 32-bit integer in a future release. However, error codes are signed. To avoid massive changes later, stubs return a signed 32-bit integer and the grizzly details of the current implementation are thus hidden from the clients and servers. However it is necessary to make conversions that deal correctly with sign extension. Therefore the macro `ERR_STATUS` is provided to determine whether what was returned is indeed an error and `ERR_CONVERT` is provided to correctly convert the unsigned 16-bit integer to a signed 32-bit quantity.

Now let us consider what the main program of the client looks like. One of the things it needs to do is to get the capability for the server it needs to talk to. In general this is not done by the stub routine because there may be more than one instance of a particular server and the main program should be able to choose the server. It gets the server's capability by doing a *name_lookup* of the name that the server should have used to publish its capability. In this case it is the name defined by `SERVER_CAP` in *this_server.h*.

Having converted the command line arguments to integers, the next step is to call the stub routine. This is a normal procedure call, but the stub then takes the server's *put* capability and the two integers and puts them into a message which it then sends out. The message it gets back has the two numbers swapped around and it sets them into the original variables, but now in the reverse order.

Thus the client might be implemented as follows:

```
#include "amoeba.h"
#include "stderr.h"
#include "stdlib.h"
#include "this_server.h"
#include "module/name.h"
main(argc, argv)
int argc;
char * argv[];
{
    errstat err;
    capability svr;
    long num1, num2;
    if (argc != 3) {
        printf("Usage: %s num1 num2\n", argv[0]);
        exit(1);
    }
    /* get the capability for the server */
    if ((err = name_lookup(SERVER_CAP, &svr)) != STD_OK) {
        printf("%s: lookup of %s failed: %s\n",
            argv[0], SERVER_CAP, err_why(err));
        exit(2);
    }
    num1 = strtol(argv[1], (char **) 0, 10);
    num2 = strtol(argv[2], (char **) 0, 10);
    /* send the command to the server */
    if ((err = swap_longs(&svr, &num1, &num2)) != STD_OK) {
        printf("%s: swap_longs failed: %s\n",
            argv[0], err_why(err));
        exit(3);
    }
    printf("new number order is %d %d\n", num1, num2);
    exit(0);
}
```

Using AIL

Although it is possible to write the server loop and client stubs by hand it is also possible to generate the stubs and server loop using the Amoeba Interface Language (AIL). All the details of marshaling and unmarshaling data are handled by AIL. The precise details of how to use AIL are described in *ail(U)* and the chapter *Programming Tools* in this volume of the manual.

An important thing to watch out for is that AIL-generated servers or clients do *not* necessarily cooperate with hand-written servers or clients. The reason for this is that AIL may decide to marshal some of the parameters of an operation into the header. Another optimization it currently performs is the marshaling of parameters to and from request and reply buffers. Rather than marshaling the parameters in a byte-order independent way (using the *buf_put* routines, described earlier) it sends them over in the original byte-order. Additionally, one of the header fields is initialized to allow the server to see whether it is necessary to byte-swap parameters contained in the buffer.

In the next two sections we will reimplement the server described in the preceding sections. Only this time we will show how much of the labour required can be transferred to AIL.

Writing a Server with AIL

In this section we will show how the example swap server can be implemented using AIL. The most important part of the AIL specification is the interface itself. An interface is defined by means of *class* definitions. For each command included in the class, the class definition specifies the *prototype* of the corresponding client stub. For example, the interface for the integer swap server could be specified as follows:

```
/* class file swap.cls */
#include "cmdreg.h"
#include "this_server.h"
class swaplong [DO_SWAP .. DO_NOTHING] {
    do_swap [DO_SWAP] (*,
        in out long first,
        in out long second
    );
    do_nothing [DO_NOTHING] (*,
        in out long first,
        in out long second
    );
};
```

It defines the class *swaplong* which consists of commands in the range from DO_SWAP up to DO_NOTHING. The client stub for the command DO_SWAP has three parameters: the "*", representing the capability for the object on which the operation is performed, and two long parameters that are passed by reference (the *in out* clauses are responsible for that). Note that although the server in this case is not really object-based, the "*" is still required in order to address the server.

The class presented here is very simple; in general it may also define class-specific types and constants. Moreover, a class can inherit procedures, types and constants from other classes, thus making it very easy to write a server supporting a subset of the standard server commands (see *std(L)*). In fact all servers should do this. This is outside the scope of this introduction however. For details see the AIL manuals. To make use of the AIL class definition, the command *ail(U)* must be told what to do with it. To generate the server main-loop which was hand-coded earlier, the following file should be given as argument to *ail*:

```
/* file swapsvr.gen */
#include "swap.cls"
generate swaplong {
    client_interface;
    server;          /* Generate server main loop */
};
```

The C source for the server main-loop will be produced in the file *ml_swaplong.c*. This file will include a header file *swaplong.h* which is generated as well.

The main-loop generated is called *ml_swaplong*, and it has a single parameter supplying the *get-port* the server should be listening to. The code for the function *server_loop* suddenly becomes very simple:

```
#include "amoeba.h"
```

```
#include "cmdreg.h"
#include "stderr.h"
#include "this_server.h"
errstat ml_swaplong();
capability get_cap;
void
server_loop()
{
    errstat err;
    err = ml_swaplong(&get_cap.cap_port);
    /* should never return */
    printf("getreq failed: %s\n", err_why(err));
    exit(1);
}
```

As soon as a request has been accepted by *ml_swaplong* it will unmarshal the parameters of the command and call a user-supplied function taking care of the actual implementation of the command. After that, it will marshal the result values (as specified by the client interface), return the reply to the client, and wait for a new request.

The implementation functions for our swap server are called *impl_do_swap* and *impl_do_nothing*. The code for *impl_do_swap* now becomes:

```
errstat
impl_do_swap(hdr, first, second)
header *hdr;
long *first, *second;
{
    errstat check_cap();
    long temp;
    errstat err;
    if ((err = check_cap(&hdr->h_priv, RGT_DOSWAP)) != STD_OK)
        return err;
    temp = *second;
    *second = *first;
    *first = temp;
    return STD_OK;
}
```

Observe that, compared with function *do_swap* in the hand-coded example, the code is now almost trivial, because the programmer is not bothered with the marshaling of input and output data. Combined with a completely analogous definition of *impl_do_nothing* and the functions *main* and *check_cap* taken from the original implementation, our AIL-based server is now complete.

Writing a Client with AIL

Now we turn our attention to the client side. We have two options here. When a client program only consists of a command-line interface to a command supported by the server, AIL is able to generate the program all by itself. The other possibility is to let AIL only generate the client stubs, writing the rest of the program by hand. Although the latter option is a bit more work, it is also more flexible.

We will first show an AIL specification generating the entire client program *do_swap*:

```
#include "swap.cls"
```

```
generate swaplong {
  client_interface;
  client_stubs(do_swap);
  command(do_swap);
};
```

The clause *client_interface* causes a header file *swaplong.h* containing C prototypes for the client stubs to be generated. The client stub for the operation *DO_SWAP* is requested by the *client_stubs* clause. It will be produced in the file *do_swap.c*. The rest of the program, which takes care of parsing the arguments, calling the stub and printing the results is requested by the *command* clause. It will be produced in the file *cmd_do_swap.c*.

It must be noted that the command produced is a direct reflection of the *do_swap* operation as defined in the class definition. More specifically, it requires the user to specify the name of the swap server capability as the first argument. The resulting program has the following interface:

```
$ swap /home/this_server/default 0 1
first=1
second=0
```

The other possibility is to use AIL only to generate the client stub *do_swap*. The AIL specification becomes:

```
#include "swap.cls"
generate swaplong {
  client_interface;
  client_stubs(do_swap);
};
```

The main program is then the same as in the hand-written case, except that the client stub is called *do_swap* rather than *swap_long*s.

Interface stubs for remote procedure calls

(in libraries: libamoeba.a, libkernel.a, libamunix.a)

Name

rpc - the interface stubs for remote procedure calls

Synopsis

```
#include "amoeba.h"

bufsize  trans(request_hdr,
               request_buf,
               request_size,
               reply_hdr,
               reply_buf,
               reply_size);

bufsize  getreq(hdr, buf, size);

void     putrep(hdr, buf, size);

interval timeout(length);
```

Description

The three calls *getreq*, *putrep*, and *trans* form the remote procedure call (RPC) interface for interprocess communication. All point to point communication is, at the lowest accessible level, structured along the client/server model; that is, a client thread sends a request message to a service, one of the threads in one of the server processes gets the request, carries it out and returns a reply message to the client. There is also the possibility for group communication. This is described in *grp(L)*. See also *rawflip(L)*.

For historical reasons, a remote operation (a request followed by a reply) is called a *message transaction* or just a *transaction*. A client thread invokes a transaction by a call to *trans*. A server thread receives a request via a call to *getreq* and returns a reply by calling *putrep*. *Trans*, *getreq*, and *putrep* are blocking; that is, a *trans* suspends a thread until the request is sent, carried out and a reply is received; *getreq* suspends a thread until a request has been received and *putrep* suspends a thread until the reply has been received by the appropriate client thread's kernel.

A request or reply message is described to the transaction system calls by means of three parameters: a pointer to a *header*, a pointer to a *buffer* and the *size* of that buffer. The *header* is a fixed-length data structure, containing addressing information, status information, an operation code and some parameters. The *buffer* is an 8-bit-character array whose *size* may vary from 0 to *30000decimal*

bytes. The parameters of *getreq* specify where in memory the header and buffer of the request are to be received. The parameters of *putrep* specify the reply to be sent and the parameters of *trans* specify the request to be sent and where in memory the reply is to be received.

The following sections explain the port- and capability-based addressing mechanism used to specify services and objects, the exact structure of request and reply messages, the three system calls and the failure semantics of RPCs.

Ports and Capabilities

Each object is both identified and protected by a *capability*. Capabilities have the set of operations that the holder may carry out on the object coded into them and they contain enough redundancy and cryptographic protection to make it infeasible to guess an object's capability. Objects are implemented by server processes that manage them. Capabilities have the identity of the object's server encoded into them (the Service Port) so that, given a capability, the system can find a server process that manages the corresponding object. The structure of capabilities and ports is defined in the standard include file *amoeba.h*:

48	24	8	48 bits
Port	Object	Rights	Random

```
#define PORTSIZE 6

typedef struct {
    int8    _portbytes[PORTSIZE];
} port;

typedef struct {
    int8    prv_object[3];
    uint8   prv_rights;
    port    prv_random;
} private;

typedef struct {
    port    cap_port;
    private cap_priv;
} capability;
```

A server thread identifies itself to the system by giving its *port* in the header of the *getreq* system call. The port that it gives must be the *get-port*, also called the *private port*, (see *priv2pub(L)* for more details). A client identifies the object of its transaction- and with it the service that manages that object- by giving the object's capability in the *port* and *private* fields of the request header. The port that the client has is the *put-port* and this is also the port returned in the header of the *getreq* call when a message arrives. This is illustrated in the example below.

Both client and server thus specify a port and this port is used by the system to bring client and server together. To prevent other processes from impersonating a particular server, a port has two appearances called get-port and put-port. A server needs to specify a get-port when it does a *getreq* call and a client has to specify a put-port (as part of the capability) when it makes a *trans* call. Get-port and put-port are related via a one-way function, *F*, which transforms a get-port into a put-port. The function was chosen so that it is 'impossible' to compute a get-port from a put-port. The library function *priv2pub* does the one-way function transformation. The system guarantees that a client's request message containing put-port *P* in its capability will only be delivered to a server thread which specified get-port *G* in its *getreq* call, where $P = F(G)$.

Message Structure

Both request and reply messages consist of two parts, a header and a buffer. A message with an empty buffer is a *header-only* message. The header has a fixed length and a fixed layout:

```
typedef uint16  command;
typedef uint16  bufsize;

typedef struct {
    port        h_port;          /* 6 bytes */
    port        h_signature;     /* 6 bytes */
    private h_priv;             /* 10 bytes */
    command h_command;          /* 2 bytes */
    int32      h_offset;         /* 4 bytes */
    bufsize h_size;             /* 2 bytes */
    uint16     h_extra;          /* 2 bytes */
                                /* total 32 bytes */
} header;

#define h_status h_command
/* alias: reply status */
```

The meaning of the fields is as follows:

h_port

The port field contains the port-part of the capability of the object on which the request should be carried out. The port field is interpreted only in *trans* and *getreq*. With *trans* it contains the put-port of the service for which the request is intended. When calling *getreq* it contains the get-port of the service offered. When *getreq* returns it contains the put-port sent by the client. In replies, the contents of this field are passed from server to client uninterpreted.

h_signature

This field is reserved for future use, possibly connected with security.

h_command

This field is normally used in requests to hold an operation code which specifies which operation to carry out. In replies, it usually conveys the result of the operation back to the client (e.g., an error status). The field is not interpreted by the operating system.

`h_offset`

The name of this field was inspired by the use of transactions for reading and writing file objects. In the standard file read/write interface, it is used to specify offsets into a file. This 32-bit field may, however, be used as the application programmer sees fit. It is not interpreted by the operating system.

`h_size`

`h_extra`

These fields can be used to communicate arbitrary 16-bit quantities between client and server. They are not interpreted by the operating system.

Request and reply messages consist of a header and optionally a buffer. The size of the buffer may be between 0 and *30000decimal*

bytes. Requests and replies, or buffers in which requests and replies are to be received, are specified via three parameters, the address of the header, the address of the buffer, and the size of the buffer. The address of the header must always point to a valid area in memory (that is, an area that is mapped in readable or read/writable, depending on whether it is used for sending or receiving). If the length of the buffer is zero, the buffer address is ignored. If the length of the buffer is non-zero, the buffer address must point to an area of memory of that length, wholly contained in a single memory segment. In other words, buffers may not straddle segment boundaries. The segment in which a buffer lies must be readable or read/writable depending on whether it is used for sending or receiving a message.

When a request or reply is received that is larger than the available buffer, no error code is returned. The number of bytes actually received in the buffer is returned. Truncation can only be detected if the *h_size* field is used to transmit the length of the buffer sent. If this is done properly, an example test for truncation is:

```
if (ERR_STATUS(stat = getreq(&hdr, &buf, cnt))) {
    /* error */
    ...
}

if (stat != hdr.h_size) {
    /* truncation */
    ...
}
```

Programmers are strongly encouraged to use the *h_size* field to communicate the true length of a message buffer to the receiver.

getreq

```
bufsize
getreq(hdr, buf, size)
header *hdr;
bufptr buf;
```

```
bufsize size;
```

A server thread uses *getreq* to receive a request from a client. After processing the request, it must return a reply by a call on *putrep*, or forward the request to another server using *grp_forward* (see *grp(L)*). A server thread may carry out only one request at a time. Successful *getreq* and *putrep* calls must therefore alternate. A server thread may always, even while serving (that is, after *getreq* and before *putrep*), use *trans* calls to have operations carried out by other servers.

A server thread must provide the get-port on which it receives in the *h_port* field of the header parameter. The code for receiving a request thus becomes:

```
header.h_port = server_get_port;
status = getreq(&header, buf, size);
```

Getreq blocks a thread until a request is received (or until the thread is alerted by a signal). The returned value, *status*, contains the number of bytes in the buffer when the call is successful, or a negative error code. If the request attempted to pass more than *size* bytes, the message placed in the area pointed to by *buf* is truncated to *size*. Errors are discussed in the next section.

Upon successful completion, the header pointed to by *hdr* contains a copy of the header given in the corresponding *trans* call of a client. Therefore the *h_port* field will contain the put-port which was used by the client. A subsequent call to *getreq* must be sure to reinitialize *h_port* to the get-port.

putrep

```
void
putrep(hdr, buf, size)
header *hdr;
bufptr buf;
bufsize size;
```

After processing a request, a server must return a reply using *putrep*. All the fields (including the *h_signature* field) of the header pointed to by *hdr* are sent to the client unmodified. As a matter of convention, the *h_command* field is used as a *status* field and is, in fact, referred to as *h_status*. It is important that all paths through the server actually set an error status rather than returning the original command code.

It is the conventional wisdom to allocate command and error-status codes centrally in order to prevent strange things from happening when one mistakes one type of object or server for another. It also simplifies object inheritance using AIL. Currently the central allocation makes use of the files *cmdreg.h* and *stderr.h*.

Every request must be accompanied by a reply. It is therefore a programming error to call *getreq* when a call to *putrep* can legally be made (and vice versa). The blocking nature of *trans* automatically prevents a client thread from being involved in more than one transaction. As a consequence, a thread can be involved in at most two transactions at a time: once as a server and once as a client.

trans

```
bufsize
trans(request_hdr, request_buf, request_size,
       reply_hdr, reply_buf, reply_size)
header *request_hdr;
bufptr request_buf;
bufsize request_size;
header *reply_hdr;
bufptr reply_buf;
bufsize reply_size;
```

Trans attempts to deliver a request to a server and passes back the server's reply. *Trans* blocks until the transaction has either failed or a reply is received. *Trans* takes six parameters, three to specify the request to be sent to the server and three to specify the memory locations to place the reply. The request header pointed to by *request_hdr* must contain the port of the service to be called in the *h_port* field. The system guarantees not to deliver a request to a server that has not specified the associated get-port in the *h_port* field of its *getreq* call. Note that there is only a negative guarantee. The system cannot guarantee to deliver a request; it can merely make an effort. The system does guarantee, however, that a request will be received at most once and by no more than one server. *Trans* has **at-most-once** semantics.

When a *trans* fails, the contents of the reply header and reply buffer are undefined and may have been changed. The contents of the request header and request buffer are guaranteed to be unchanged, unless they overlap the reply header or reply buffer. It is common practice to use the same header and buffer for request and reply in one transaction.

The other fields in the header, except possibly *h_signature*, are not interpreted by the system. But again, convention dictates the use of the *h_private* field for the private part of the capability of the object to which the request refers (the port part is in the *h_port* field), the *h_command* field for a unique request code, the *h_status* field for a unique error reply code, and the *h_size* field for the size of request and reply. The value returned by *trans* contains the number of bytes in the buffer when the call is successful, or a negative error code. If the reply attempted to pass more than *reply_size* bytes, the message placed in the area pointed to by *reply_buf* is truncated to *reply_size*.

timeout

```
interval
timeout(length)
interval length;
```

Before a request can be delivered by the system, the client's kernel must know where to send the request. It must *locate* a server. Details of the locate mechanism are not relevant here, but what is relevant is that there is a minimum time that the kernel will spend trying to locate a suitable server. This per-thread locate timeout can be modified by the *timeout* system call. Its argument *length* is the new minimum timeout in milliseconds. It returns the previous value of the timeout. The default timeout is system dependent; it is typically 5 seconds. It is not a good idea to set the timeout below 2 seconds (*length=2000*) since it may result in locate failures even though the server is actually available.

Error Codes and Failure Semantics

All three calls, *getreq*, *putrep* and *trans* can fail in a number of ways. A distinction must be made between programming errors and failures. An error is caused by faults in the program, for instance, calling *getreq* when a *putrep* is still due. A failure is caused by events beyond the control of the programmer: server

crashes, network failures, etc.

Errors are caused by program bugs. A program with an error in it should be debugged. Errors, therefore, cause an exception, which, when not fielded by the program, causes the program to abort. When they are fielded, an interrupt routine is called to handle the error. In addition, the system calls also return a negative error code. Under Amoeba the exception is `EXC_SYS`. Under UNIX the exception generated is `SIGSYS`.

All failures cause *trans* and *getreq* to return a negative error code. The include file *stderr.h* provides the mapping between the error codes and the names used in programs. The table below gives an overview of the errors and failures, in which calls they can occur (T for *trans*, G for *getreq* and P for *putrep*), the code returned on *getreq* or *trans* (*putrep* does not return a value), and whether an exception is raised when an error occurs.

Value returned	Exception	Which call	Error description
<code>RPC_NOTFOUND</code>		T	Server not found
<code>RPC_ABORTED</code>		T G	Signal received while blocked
<code>RPC_FAILURE</code>		T	Server or network failure
<code>RPC_FAILURE</code>	*	T G P	Buffer length > 30000
<code>RPC-FAILURE</code>	*	G	Getreq while serving
<code>RPC_BADADDRESS</code>		T G P	Use of invalid pointer
<code>RPC-BADPORT</code>		T G	Use of a NULL port
	*	P	Putrep while not serving
<code>RPC_TRYAGAIN</code>		T	Out of resources

Server not found

When locating a server fails, the `RPC_NOTFOUND` error code is returned on *trans* calls. The reason can be that the server is not yet ready, or that there is no server at all. In any case, the request will not have been delivered anywhere. When this error code is returned, it is sometimes sensible to wait a few seconds and try once more. The server may have called *getreq* in the meantime.

Signal received while blocked

If a thread has indicated interest in a signal and that signal is raised while the thread is in a *getreq* or the locate stage of a *trans* call, the call is broken off and `RPC_ABORTED` is returned. No useful information should be expected in the receive header and buffer. Threads can be aborted in system calls as a consequence of debugging. For servers, a sensible action following an `RPC_ABORTED` error code is to restart the *getreq* or *trans* operation.

Server or network failure

When a request has been sent and no reply is received and the server's kernel does not respond to "are-you-alive?" pings, `RPC_FAILURE` is returned. The server may or may not have carried out the request; the server may even have sent a reply, but then it has not been received. The course to take when this failure happens depends very much on the semantics of the attempted operation.

Buffer length > 30000

The buffer length is an unsigned integer. It can not be less than zero. If it is larger than the maximum size of a request or reply, an exception is raised and `RPC_FAILURE` is returned if the exception is handled.

Getreq while serving, putrep while not serving

Programming errors, so an exception is raised. Putrep does not return a value, so when the exception is caught no error code is returned.

Use of invalid pointers

Using pointers to headers or buffers wholly or partially in non-existent memory, to receive headers or buffers in read-only memory, or to buffers straddling segment boundaries, an exception is raised. If the exception is caught, *trans* and *getreq* additionally return `RPC_BADADDRESS`.

Use of a NULL-port

When you forget to fill in the *h_port* field on *getreq* or *trans*, the system returns a `RPC_BADPORT` failure. This is not an exception, because ports are often filled in with values retrieved from remote sources such as the directory server. It seemed unreasonable to ask application programmers to write code to check for NULL-ports when the system has to check for them anyway.

Out of resources

This error can only occur under UNIX or UNIX-like systems. It is used when the Amoeba driver could not get access to enough resources to send the message. It is guaranteed that the request has not reached any server. If enough resources are freed the request can be safely repeated.

Two helpful macros are available for manipulating returned status codes: `ERR_STATUS` and `ERR_CONVERT`. They provide the proper casts from the 16-bit unsigned integers to signed integers for testing the sign bit. The macros are defined in *amoeba.h*.

Signals

The previous section specified that when a *trans* is blocked and a signal is received while the system is still trying to locate the server, *trans* will return the error code `RPC_ABORTED`. When the location of the server is known and the RPC has been sent to the server and no reply has been received as yet, the reaction of the system to signals to be caught is altogether different. In this case signals are propagated to the server. Thus the server will receive the signal and can choose to react accordingly. It can, as per default, ignore the signal or catch it. If the server ignores the signal the signal will have no effect in both server and client. If the server handles the signal it can choose its own way of reacting, for example by returning an error reply with *putrep*. If the server itself catches signals and is waiting for a reply to a *trans* the signal will again be propagated.

Signals can abort calls to *getreq*, but can not abort calls to *putrep*. See also *signals(L)*.

Example

The example below shows the typical server main loop for a very simple file server and a sample of client code for calling that server for a read operation and a write operation. It is assumed that the client's capability has the put-port associated with the server's get-port.

```
#include "amoeba.h"
#include "cmdreg.h"
#include "stderr.h"

/* Server threads typically sit in an endless loop
 * fielding and carrying out client requests
 */
for (;;) {
    /* Put the server's get-port in the header.
     * When the request
     * arrives, it will be replaced by the put-port of the
     * client's request.
     */

    hdr.h_port = getport;
    size = getreq(&hdr, buf, 100);
    /* Check for errors. */
    if (ERR_STATUS(size)) {
        /* The macro ERR_CONVERT casts the 16-bit
         * unsigned into an ordinary signed int.
         */
        if (ERR_CONVERT(size) == RPC_ABORTED) {
            /* When debugging a program, signals are used to
             * checkpoint it. These abort getreq calls. When
             * this happens, just try again.
             */
            continue;
        }
        /* Other errors are bad */
        fprintf(stderr, "Getreq error (%s)\n",
                err_why(ERR_CONVERT(size)));
        exit(-1);
    }
}
```

```
    }

    /* Control gets here when a request is successfully
     * received.
     * Dispatch to the implementation routines for the
     * different request types.
     */

    switch (hdr.h_command) {

    case F_READ:
        /* Read implementation code goes here For the example,
         * simulate the result of a successful read operation.
         */
        size = hdr.h_size = 100;
        hdr.h_status = STD_OK;
        break;

    case F_WRITE:
        /* Same for write */
        size = hdr.h_size = 0;
        hdr.h_status = STD_OK;
        break;

    default:
        /* Not a legal request type, return
         * an error code
         */
        hdr.h_status = STD_COMBAD;
    }

    /* Return a reply. No error codes are returned */
    putrep(&hdr, buf, size);
}
}
```

Below is the client code for calling the server to do a read and then a write operation.

```
#include "amoeba.h"
#include "stderr.h"

/* Set transaction timeout to 30 seconds */
timeout(30000);

/* Fill in transaction header */
hdr.h_port = cap.cap_port; /* Server port */
hdr.h_priv = cap.cap_priv; /* Rest of file cap */
hdr.h_command = F_READ;
hdr.h_size = 100; /* Read 100 bytes .. */
hdr.h_offset = 0; /* .. from beginning of file */

/* Call the server; request buffer is empty,
 * reply buffer is large enough to hold requested bytes
 */
size = trans(&hdr, NILBUF, 0, &hdr, buf, 100);

/* Size holds positive number of bytes read or
```

```
* negative error code.
*/
if (ERR_STATUS(size)) hdr.h_status = size;

/* Even if the transaction succeeds, the server may
 * return an error code of its own. Convention is that
 * the only success code is STD_OK (zero).
 */
if (hdr.h_status != STD_OK) {
    /* Print an error message. The macro ERR_CONVERT
     * casts the error code into an int. The function
     * err_why converts standard error codes into a
     * string.
     */
    fprintf(stderr, "1st trans failed (%s)\n",
            err_why(ERR_CONVERT(hdr.h_status)));
    exit(-1);
}
/* The server returns number of bytes read in hdr.h_size.
 * If the reply is truncated, or if the server is faulty
 * the number of bytes received may not correspond to what
 * the server claims was sent.
 */

if (size != hdr.h_size) {
    /* In the case of file read, the client may be
     * assumed not to ask for more than the buffer
     * can hold, so this code is only reached if
     * the server is faulty.
     */
    fprintf(stderr, "message truncated\n");
    exit(-1);
}

/* Read trans succeeded, try a write trans.
 * Again, fill in header fields. Never trust what
 * server left intact.
 */
hdr.h_port = cap.cap_port;
hdr.h_priv = cap.cap_priv;
hdr.h_command = F_WRITE;
hdr.h_size = 100;
hdr.h_offset = 0;

/* This time, request buffer is not empty */
size = trans(&hdr, buf, 100, &hdr, NILBUF, 0);

/* Size returned should be zero (empty reply buffer) */
if (ERR_CONVERT(size) != 0) hdr.h_status = size;
if (hdr.h_status != STD_OK) {
    fprintf(stderr, "2nd trans failed (%s)\n",
            err_why(ERR_CONVERT(hdr.h_status)));
    exit(-1);
}
```

See Also

error[L] , grp[L] , priv2pub[L] , signals[L] .

RPC server example

This example code was extracted from the object manager *om* server source code.

```
void
server_loop()
{
    char      *cp, *buf;
    header    h;
    bufsize   n, reply_size;
    errstat   err;
    static capability mycap;
    /* Is static just for initialization to 0 */
    capability pubcap;
    capability oldcap;
    port      check;
    char      pathbuf[257];
    char      *mycap_path;

    buf=malloc(OMSIZE);
    if(buf==0)
    {
        sys_log(SYS_FATAL,
            "%s: server_loop: can't malloc buffer!\n",
            progname);
        my_exit(-1);
    }

    /* On startup, create a unique cap and store it.
    */
    uniqport(&mycap.cap_port);
    uniqport(&check);

    priv2pub(&mycap.cap_port,&pubcap.cap_port);

    if (prv_encode(&pubcap.cap_priv,
        (objnum)0,
        PRV_ALL_RIGHTS,
        &check) < 0){

        sys_log(SYS_FATAL,
            "%s: can not prv_encode server cap\n",
            progname);
        my_exit(1);
    }

    /* check for old server cap ... */

    if ((err = name_lookup(mycap_path, &oldcap))
        == STD_OK)
    {
        err= name_replace(mycap_path,&pubcap);

        if ( err != STD_OK && err != STD_NOTFOUND ) {
            sys_log(SYS_FATAL,
                "%s: name_replace %s failed (%s)\n",
                progname, mycap_path, err_why(err));
            my_exit(1);
        }
    }
}
```

```
    /* Fall through on NOTFOUND & OK */
}

switch( err ) {

case STD_NOTFOUND:
    if ((err = name_append(mycap_path,
                          &pubcap))
        != STD_OK) {
        sys_log(SYS_FATAL,
                "%s: name_append %s failed (%s)\n",
                progname, mycap_path,
                err_why(err));
        my_exit(1);
    }

case STD_OK:
    break ;

default:
    sys_log(SYS_FATAL,
            "%s: name_lookup %s failed (%s)\n",
            progname, mycap_path, err_why(err));
    my_exit(1);
}

/* Server loop: */

for (;closing==0;) {

    /* the get port */

    h.h_port = mycap.cap_port;

    reply_size = 0;

    n = getreq(&h, buf, OMSIZE);

    if (ERR_STATUS(n)) {
        err = ERR_CONVERT(n);
        if (err == RPC_ABORTED) {
            continue;
        }
        sys_log(SYS_FATAL, "%s: getreq failed!\n");
        my_exit(1);
    }

    switch (h.h_command) {

        /* std_info request */

        case STD_INFO:
            (void) strcpy(buf, "object manager");
            h.h_size = reply_size = strlen(buf);
            h.h_status = STD_OK;
            break;

        /* std_status request */

        case STD_STATUS:
            cp = buf;

```

```
(void) strcpy(cp, "Object Manager:\n\n");
cp += strlen(cp);
sprintf(cp, "status=%s", om_server_status);

if ((reply_size = strlen(buf)) > OMSIZE)
    reply_size = OMSIZE;

h.h_size = reply_size;
h.h_status = STD_OK;
break;

/* the server shutdown command */

case STD_EXIT:
{
    rights_bits rights;
    h.h_size=0;

    /* the client needs full rights !!!*/

    if(prv_decode(&h.h_priv,
                 &rights,
                 &check) ||
        (rights != PRV_ALL_RIGHTS))
    {
        h.h_status=STD_DENIED;
    }
    else
    {
        closing=1;

        h.h_status=STD_OK;
    }
    break;
}

/* unknown command */
default:
    h.h_status = STD_COMBAD;
    break;
}

putrep(&h, buf, reply_size);
}

my_exit(0);
}
```

ar - ASCII representation of common Amoeba types

(in libraries: libamoeba.a, libamunix.a)

Name

ar - ASCII representations of common Amoeba types

Synopsis

```
#include "amoeba.h"
#include "module/ar.h"

char * ar_cap(cap_p)
char * ar_port(port_p)
char * ar_priv(priv_p)

char * ar_tocap(s, cap_p)
char * ar_toport(s, port_p)
char * ar_topriv(s, priv_p)
```

Description

To aid in debugging a set of routines has been provided to produce human-readable representations of capabilities and various subparts thereof.

The first three functions return a pointer to a string containing the ASCII representation of the bytes in a *capability*, *port* and *private* (see *rpc(L)*), respectively. Note that the string is in static data and subsequent calls by other threads or the same thread will overwrite the contents of the string. Each routine has its own static data so there are no interactions between the individual routines.

The latter three functions convert a string of the format returned by the first three routines to a capability, port and private, respectively. The *ar_to* functions assign to the capability, port or private, pointed to by the second parameter. They return a pointer to the character beyond the last character of the string. If the string is supposed to contain no extra characters, you should check that the returned pointer points to a NUL character.

If the string passed to an *ar_to* function is illegal, NULL is returned. The format used is explained below.

Functions

ar_port

```
char *  
ar_port(port_p)  
port *port_p;
```

Ar_port returns a pointer to a string with the six bytes in the port represented as *x:x:x:x:x:x*, where *x* is the value of a byte in hexadecimal.

ar_priv

```
char *  
ar_priv(p)  
private *p;
```

Ar_priv uses the format *D(X)/x:x:x:x:x:x*, where *D* is the object number in decimal, *X* is the rights in hexadecimal, and *x* is a byte from the check field, in hexadecimal.

ar_cap

```
char *  
ar_cap(cap_p)  
capability *cap_p;
```

The format used by *ar_cap* is the concatenation of the formats of *ar_port* and *ar_priv* respectively, separated by a slash. That is, *x:x:x:x/x/D(X)/x:x:x:x:x*.

ar_tocap

```
char *  
ar_tocap(s, cap_p)  
char *s;  
capability *cap_p;
```

Ar_tocap takes a string *s* in the format produced by *ar_cap* and makes a capability that matches that representation in the capability pointed to by *cap_p*. It returns a pointer to the first character after the parsing of *s* stopped.

ar_toport

```
char *
```

```
ar_toport(s, port_p)
char *s;
port *port_p;
```

Ar_toport takes a string *s* in the format produced by *ar_port* and makes a port that matches that representation in the capability pointed to by *port_p*. It returns a pointer to the first character after the parsing of *s* stopped.

ar_topriv

```
char *
ar_topriv(s, priv_p)
char *s;
private *priv_p;
```

Ar_topriv takes a string *s* in the format produced by *ar_priv* and makes a private part of a capability (i.e., object number, rights and check field) that matches that representation in the capability pointed to by *priv_p*. It returns a pointer to the first character after the parsing of *s* stopped.

Examples

The following code prints out an array of capabilities:

```
#include "amoeba.h"
#include "module/ar.n"
printcaps(caps, n)
    capability caps[];
    int n;
{
    int i;

    for (i = 0; i < n; ++i)
        printf("Cap #%d: %s\n", i, ar_cap(&caps[i]));
}
```

The following function prints “hello world” and stores the port consisting of the bytes 1, 2, 3, 4, 5, 6 in *p*.

```
Hello()
{
    port p;

    printf("%s\n", ar_toport("1:2:3:4:5:6hello world", &p));
}
```

}

See Also

c2a(U) , rpc[L] .

buffer - getting and putting data in arch-independent format

(in libraries: libamoeba.a, libamunix.a)

Name

buffer - getting and putting data in architecture-independent format

Synopsis

```
#include "amoeba.h"
#include "module/buffers.h"

char *buf_get_cap(p, endp, &val)
char *buf_get_capset(p, endp, &val)
char *buf_get_int16(p, endp, &val)
char *buf_get_int32(p, endp, &val)
char *buf_get_objnum(p, endp, &val)
char *buf_get_pd(p, endp, &val)
char *buf_get_port(p, endp, &val)
char *buf_get_priv(p, endp, &val)
char *buf_get_right_bits(p, endp, &val)
char *buf_get_string(p, endp, &val)
char *buf_put_cap(p, endp, &val)
char *buf_put_capset(p, endp, &val)
char *buf_put_int16(p, endp, val)
char *buf_put_int32(p, endp, val)
char *buf_put_objnum(p, endp, val)
char *buf_put_pd(p, endp, &val)
char *buf_put_port(p, endp, &val)
char *buf_put_priv(p, endp, &val)
char *buf_put_right_bits(p, endp, val)
char *buf_put_string(p, endp, val)
```

Description

These functions get data from, or put data into, a character buffer in a format that is independent of any machine architecture; in particular of byte-order dependencies. They are primarily useful in loading and unloading RPC buffers (and reply buffers). All of them have the same calling sequence and return value, except for the type of the last argument.

In each case, *p* should point to the place within the buffer where data is to be stored or retrieved, and *endp* should point to the end of the buffer. (That is, the first byte after the buffer.)

For the *buf_get* functions, the *val* argument should be the address where the data retrieved from the buffer is to be stored. It must be a pointer to the type of value expected by the function. (See the individual function descriptions, below.)

For the *buf_put* functions, there are two possibilities: if the value to be put in the buffer is an integral type, or a character pointer, the value itself should be passed as the *val* argument. For larger, structured types, a pointer to the value should be supplied. (See the individual function descriptions, below.)

On success, the functions return a pointer to the next byte in the buffer following the value that was retrieved or inserted. Thus, in normal usage, a simple sequence of calls to *buf_get* functions or to *buf_put* functions can be used to get or put a sequence of values, in order. All that is necessary is to provide the output of a previous call as the first argument of the next call (see the example, below.)

The end pointer is used to detect whether there is sufficient space in the buffer to get or put the specified value. If not, NULL is returned to indicate failure. To avoid the necessity of an error check after each of a sequence of calls to *buf_get* or *buf_put* functions, this overflow error propagates: if a NULL pointer is passed as the *p* argument to any of the functions, it returns NULL.

Note that these routines guarantee not to require more than *sizeof(datatype)* bytes to store the data in the buffer. If similar routines are needed to marshal a *union* then the union should be encapsulated within a *struct* with an extra field which specifies the field of the *union* which is really being sent.

Functions

buf_get_cap

```
char *
buf_get_cap(p, endp, valp)
char *p, *endp;
capability *valp;
```

A capability is retrieved from the buffer pointed to by *p* and copied to the capability pointed to by *valp*.

buf_get_capset

```
#include "capset.h"

char *
buf_get_capset(p, endp, valp)
char *p, *endp;
capset *valp;
```

A capability-set is retrieved from the buffer pointed to by *p* and copied to the capability-set pointed to by *valp*. The suite for the capability-set is allocated by this function. The suite must not contain more than MAXCAPSET entries.

buf_get_int16

```
char *  
buf_get_int16(p, endp, valp)  
char *p, *endp;  
int16 *valp;
```

An architecture-independent 16-bit integer is retrieved from the buffer pointed to by *p* and copied to the 16-bit integer pointed to by *valp*, in the form required by the local host architecture.

buf_get_int32

```
char *  
buf_get_int32(p, endp, valp)  
char *p, *endp;  
int32 *valp;
```

An architecture-independent 32-bit integer is retrieved from the buffer pointed to by *p* and copied to the 32-bit integer pointed to by *valp*, in the form required by the local host architecture.

buf_get_objnum

```
char *  
buf_get_objnum(p, endp, valp)  
char *p, *endp;  
objnum *valp;
```

An object number (as found in capabilities) is retrieved from the buffer pointed to by *p* and copied to the object number pointed to by *valp*.

buf_get_pd

```
char *  
buf_get_pd(p, endp, valp)  
char *p, *endp;  
process_d *valp;
```

A process descriptor stored in an architecture-independent format is retrieved from the buffer pointed to by *p* and stored in the process descriptor structure pointed to by *valp*. All necessary allocation of sub-structures is performed by this function. See *process_d(L)* for more details.

buf_get_port

```
char *  
buf_get_port(p, endp, valp)  
char *p, *endp;  
port *valp;
```

A port is retrieved from the buffer pointed to by *p* and copied to the port pointed to by *valp*.

buf_get_priv

```
char *  
buf_get_priv(p, endp, valp)  
char *p, *endp;  
private *valp;
```

The private part of a capability (object number, rights and check field) is retrieved from the buffer pointed to by *p* and copied to the place pointed to by *valp*.

buf_get_right_bits

```
char *  
buf_get_right_bits(p, endp, valp)  
char *p, *endp;  
rights_bits *valp;
```

The rights bits of a capability are retrieved from the buffer pointed to by *p* and copied to the place pointed to by *valp*.

buf_get_string

```
char *  
buf_get_string(p, endp, valp)  
char *p, *endp;  
char **valp;
```

The char pointer specified by *valp* is modified to point to the character string beginning at location *p* in the buffer, and *p* is advanced to just beyond the NULL-byte that terminates the string. If no such NULL-byte is found, however, NULL is returned and *valp* is left undefined.

WARNING: This behavior is often not what is wanted, especially when the transaction buffer is to be re-used. Most of the time, it will be necessary to copy the characters of the string out of the buffer to a safer place. This is not done by *buf_get_string* nor is any storage allocated for the string, beyond the transaction buffer itself.

buf_put_cap

```
char *  
buf_put_cap(p, endp, valp)  
char *p, *endp;  
capability *valp;
```

The capability pointed to by *valp* is copied into the buffer pointed to by *p*.

buf_put_capset

```
#include "capset.h"  
  
char *  
buf_put_capset(p, endp, valp)  
char *p, *endp;  
capset *valp;
```

The capability-set pointed to by *valp* (including the suite sub-structure) is copied into the buffer pointed to by *p*. The suite must not contain more than MAXCAPSET entries.

buf_put_int16

```
char *  
buf_put_int16(p, endp, val)  
char *p, *endp;  
int16 val;
```

Val is stored in the buffer pointed to by *p*.

buf_put_int32

```
char *  
buf_put_int32(p, endp, val)  
char *p, *endp;  
int32 val;
```

Val is stored in the buffer pointed to by *p*.

buf_put_objnum

```
char *
```

```
buf_put_objnum(p, endp, val)
char *p, *endp;
objnum val;
```

Val, which should be an object number as found in a capability, is stored in the buffer pointed to by *p*.

buf_put_pd

```
char *
buf_put_pd(p, endp, valp)
char *p, *endp;
process_d *valp;
```

The process descriptor pointed to by *valp* is copied into the buffer pointed to by *p*. See *process_d(L)* for more details.

buf_put_port

```
char *
buf_put_port(p, endp, valp)
char *p, *endp;
port *valp;
```

The port pointed to by *valp* is copied into the buffer pointed to by *p*.

buf_put_priv

```
char *
buf_put_priv(p, endp, valp)
char *p, *endp;
private *valp;
```

The private part of a capability (object number, rights and check field), pointed to by *valp*, is copied into the buffer pointed to by *p*.

buf_right_bits

```
char *
buf_put_right_bits(p, endp, val)
char *p, *endp;
rights_bits val;
```

Val, which should be the rights bit of a capability, is copied into the buffer pointed to by *p*.

buf_put_string

```
char *  
buf_put_string(p, endp, valp)  
char *p, *endp;  
char *valp;
```

The string pointed to by *valp* is copied into the buffer pointed to by *p*, including the NULL-byte terminator.

Example

The following code copies a capability, *c*, a 32-bit integer, *i*, a character string, *s*, into a transaction buffer, then retrieves them into the variables *c2*, *i2*, and *s2*.

```
capability c, c2;  
int32 i, i2;  
char *s, *s2;  
char buffer[1000];  
char *p, *ep;  
  
name_lookup(path, &c);  
i = 17;  
s = "foobar";  
  
p = buffer;  
ep = buffer + sizeof buffer;  
p = buf_put_cap(p, ep, &c);  
p = buf_put_int32(p, ep, i);  
p = buf_put_string(p, ep, s);  
if (!p) {  
    error("Buffer overflow during put");  
}  
  
p = buffer;  
ep = buffer + sizeof buffer;  
p = buf_get_cap(p, ep, &c2);  
p = buf_get_int32(p, ep, &i2);  
p = buf_get_string(p, ep, &s2);  
if (!p) {  
    error("Buffer underflow during get");  
}
```

name - directory manipulation

(in libraries: libamoeba.a, libamunix.a)

Name

name - name-based functions for manipulating directories

Synopsis

```
#include "module/name.h"

errstat cwd_set(path)
errstat name_append(path, object)
char * name_breakpath(path, dir)
errstat name_create(path)
errstat name_delete(path)
errstat name_lookup(path, object)
errstat name_replace(path, object)
```

Description

This module provides a portable, name-oriented interface to directory servers. It has the advantage of being independent of the particular directory server in use, but it does not make available all the functionality of every directory server, e.g., SOAP. It is similar to the module containing the same set of functions but with the prefix “dir_” instead of “name_”. The main difference is that the “dir_” functions take an extra argument, *origin*, which is the capability for a directory relative to which the given *path* is to be interpreted.

In each function, the *path* should be a “path name” that specifies a directory or directory entry. To these functions, a directory is simply a finite mapping from character-string names to capabilities. Each (name, capability) pair in the mapping is a “directory entry”. The capability can be for any object, including another directory; this allows arbitrary directory graphs (the graphs are not required to be trees). A capability can be entered in multiple directories or several times (under different names) in the same directory, resulting in multiple links to the same object. Note that some directory servers may have more complex notions of a directory, but all that is necessary in order to access them from this module is that they satisfy the above rules.

A path name is a string of printable characters. It consists of a sequence of 0 or more “components”, separated by “/” characters. Each component is a string of printable characters not containing “/”. As a special case, the path name may begin with a “/”, in which case the first component starts with the next character of the path name. Examples: *a/silly/path*, */profile/group/cosmo-33*.

If the path name begins with a “/”, it is an “absolute” path name, otherwise it is a “relative” path name.

The interpretation of relative path names is relative to the current working directory. (Each Amoeba process has an independent capability for a current working directory, usually inherited from its creator -- see *cwd_set* below.)

In detail, the interpretation of a path name relative to a directory *d* is as follows:

- (1) If the path has no components (is a zero-length string), it specifies the directory *d*;
- (2) Otherwise, the first component in the path name specifies the name of an entry in directory *d* (either an existing name, or one that is to be added to the mapping);
- (3) If there are any subsequent components in the path name, the first component must map to the capability of a directory, in which case the rest of the components are recursively interpreted as a path name relative to that directory.

The interpretation of absolute path names is the same, except that the portion of the path name after the “/” is interpreted relative to the user’s “root” directory, rather than to the current working directory of the process. (Each user is given a capability for a single directory from which all other directories and objects are reached. This is called the user’s root directory.)

The components “.” and “..” have special meaning. Paths containing occurrences of these components are syntactically evaluated to a normal form not containing any such occurrences before any directory operations take place. See *path_norm(L)* for the meaning of these special components.

Errors

All functions return the error status of the operation. They all involve transactions and so in addition to the errors described below, they may return any of the standard RPC error codes.

STD_OK:

the operation succeeded;

STD_CAPBAD:

an invalid capability was used;

STD_DENIED:

one of the capabilities encountered while parsing the path name did not have sufficient access rights (for example, the directory in which the capability is to be installed is unwritable);

STD_NOTFOUND:

one of the components encountered while parsing the path name does not exist in the directory specified by the preceding components (the last component need not exist for some of the functions, but the other components must refer to existing directories).

Functions

name_append

```
errstat
name_append(path, object)
char *path;
capability *object;
```

Name_append adds the *object* capability to the directory server under *path*, where it can subsequently be retrieved by giving the same name to *name_lookup*. The path up to, but not including the last component of *path* must refer to an existing directory. This directory is modified by adding an entry consisting of the last component of *path* and the *object* capability. There must be no existing entry with the same name.

Required Rights:

SP_MODRGT in the directory that is to be modified

Error Conditions:

STD_EXISTS: name already exists

name_replace

```
errstat
name_replace(path, object)
char *path;
capability *object;
```

Name_replace replaces the current capability stored under *path* with the specified *object* capability. The *path* must refer to an existing directory entry. This directory entry is changed to refer to the specified *object* capability as an atomic action. The entry is not first deleted then appended.

Required Rights:

SP_MODRGT in the directory that is to be modified

name_breakpath

```
char *
name_breakpath(path, dir)
char *path;
capability *dir;
```

Name_breakpath stores the capability for the directory allegedly containing the object specified by *path* in *dir*, and returns the name under which the object is stored, or would be stored if it existed. It is intended for locating the directory that must be modified to install an object in the directory service under the given *path*. In detail, *name_breakpath* does the following:

If the *path* is a path name containing only one component, *name_breakpath* stores the capability for either the current working directory (if the path name is relative) or the user's root directory (if the path name is absolute) in *dir*, and returns the *path* (without any leading "/").

Otherwise, the *path* is parsed into two path names, the first consisting of all but the last component and the second a relative path name consisting of just the last component. *Name_breakpath* stores the capability for the directory specified by the first in *dir* and returns the second. The first path name must refer to an existing directory.

cwd_set

```
errstat
cwd_set(path)
char *path;
```

Cwd_set changes the current working directory to that specified by *path*.

Required Rights:

NONE

Error Conditions:

STD_NOTFOUND: the directory does not exist

name_create

```
errstat
name_create(path)
char *path;
```

Name_create creates a directory and stores its capability in a directory server under *path*. If *path* is a relative path name, the new directory is created using the server containing the current working directory, otherwise the new directory is created using the server containing the user's root directory. (Note that either of these might be different from the server that contains the directory which is to be modified by adding the new directory.)

The path up to, but not including the last component of *path* must refer to an existing directory. This directory is modified by adding an entry consisting of a new directory, named by the last component of *path*. There must be no existing entry with the same name.

Required Rights:

SP_MODRGT in the directory that is to be modified

Error Conditions:

STD_EXISTS: name already exists in the directory server

name_delete

```
errstat
name_delete(path)
char *path;
```

Name_delete deletes the entry *path* from the directory server. The object specified by the capability associated with *path* in the directory entry is not destroyed. If desired, it should be separately destroyed (see *std_destroy(L)*).

Required Rights:

SP_MODRGT in the directory that is to be modified

name_lookup

```
errstat
name_lookup(path, object)
char *path;
capability *object;
```

Name_lookup finds the capability named by *path* and stores it in *object*.

Examples

```
/* Change the name /home/abc to /home/xyz: */
capability mod_dircap, object;
char *old_name = "/home/abc";
char modify_dir[NAME_MAX+1];
char *entry_name = name_breakpath(old_name, &mod_dircap);

if (entry_name != NULL) {
    strncpy(modify_dir, old_name, entry_name - old_name);
    modify_dir[entry_name - old_name] = '\0';
    if (cwd_set(modify_dir) == STD_OK &&
        name_lookup(entry_name, &object) == STD_OK &&
        name_append("xyz", &object) == STD_OK &&
        name_delete(entry_name) == STD_OK) {
        fprintf(stderr, "Successful name change\n");
        return;
    }
    fprintf(stderr, "Error -- no name change\n");
}
```

}

See Also

direct[L] .

priv2pub - convert private to public port

(in libraries: libamoeba, libamunix.a)

Name

priv2pub - convert private port (get-port) to public port (put-port)

Synopsis

```
#include "amoeba.h"

void
priv2pub(priv, pub)
port *priv; /* in */
port *pub; /* out */
```

Description

The Amoeba publications describe how a one-way function is applied by the F-box to convert a get-port to a put-port. This routine is used to implement the F-box in software. The *priv2pub* function is a one-way function (in that it is extremely difficult to invert) which is used to encrypt the port of a server.

Priv2pub converts a private port (also known as a *get-port*) to a public port (also known as *put-port*). A private port is the port used by a server in its *getreq* call; a public port is the port used by a client of that server in its *trans* call (see *rpc(L)*). When *getreq* is called the kernel calls *priv2pub* to encrypt the private port and only accepts requests to the encrypted port. Thus it is difficult for someone to pretend to be a server for which they do not know the *get-port*. Note that when *getreq* returns the *header* argument will contain the *put-port* since that is what the client sent.

It is legal for the *priv* and *pub* arguments to point to the same location. Neither should be a NULL-pointer.

See Also

one_way[L] , **rpc[L]** .

prv - manipulate capability private parts

(in libraries: libamoeba.a, libamunix, libkernel.a)

Name

prv - manipulate capability private parts

Synopsis

```
#include "amoeba.h"
#include "module/prv.h"

int prv_decode(prv, prights, random)
int prv_encode(prv, obj, rights, random)
objnum prv_number(prv)
```

Description

These functions are used by servers to make and validate capabilities and to extract the object number from a capability.

Functions

prv_decode

```
int
prv_decode(prv, prights, random)
private *prv;
rights_bits *prights;
port *random;
```

Prv_decode is used to validate a capability when the original random number is known. It operates on the private part of the capability. It checks the check field in *prv* and if the check field is valid it returns in *prights* the rights in the capability. It returns 0 if it succeeded and -1 if the check field was invalid. If *prv_decode* fails then the capability was not valid and should be rejected.

prv_encode

```
int
prv_encode(prv, obj, rights, random)
```

```
private *prv;
objnum obj;
rights_bits rights;
port *random;
```

Prv_encode builds a private part of a capability in *prv* using the object number *obj*, the rights field *rights* and the check field pointed to by *random*. It returns 0 if it succeeded and -1 if the *rights* or the *obj* argument exceeded the implementation limits.

prv_number

```
objnum
prv_number(prv)
private *prv;
```

Prv_number returns the object number from the private part *prv*.

Warnings

The current implementation restricts the object number to 24 bits and the rights field to 8 bits.

Example

The following function implements the `std_restrict` operation:

```
errstat
impl_std_restrict(cap, mask, newcap)
capability *cap;
rights_bits mask;
capability *newcap;
{
    objnum obj;
    rights_bits rights;
    struct foobar {
        port random;          /* Random checkword */
        ...
    }

    *foo, *FindFoo();

    /* Find object: */
    obj = prv_number(&cap->cap_priv);

    if ((foo = FindFoo(obj)) == NULL)
        return STD_CAPBAD;    /* Never heard of */
    /* Validate: */

    if (prv_decode(&cap->cap_priv, &rights, &foo->random) != 0)
```

```
        return STD_CAPBAD;        /* Do not trust cap */

/* Build new cap: */
rights &= mask;
newcap->cap_port = cap->cap_port;

if (prv_encode(&newcap->cap_priv, obj, rights, &foo->random) != 0)
    return STD_SYSERR;        /* Should not happen */
return STD_OK;
}
```

See Also

rpc[L].

std_params programming interface

Name

std_params - standard client-server interface for setting or controlling server run time parameters in a human friendly way

Synopsis

```
#include <stdcmd.h>

errstat std_getparams (capability server*,
                      char *parambuf,
                      int  bufsize,
                      int  *paramlen,
                      int  *nparams);
errstat std_setparams (capability server*,
                      char parambuf*,
                      int  paramlen,
                      int  nparams);
```

Description

The std_params interface provides an easy way to get or set server parameters. You can use the std_params[A] util to show or change settings.

Programming Interface

Client interface

std_getparams

```
errstat
std_getparams(capability *server,
               char   *parambuf,
               int    bufsize,
               int    *paramlen,
               int    nparams);
```

This client request function returns in *parambuf* (of size *bufsize*) the parameter list supported by the server with server capability *server*.

Buffer format:

```
for each parameter:
    name\0
    type\0
    description\0
    current value\0
```

std_setparams

```
errstat
std_setparams(capability *server,
              char *parambuf,
              int paramlen,
              int nparams);
```

This function set one or more server parameters *nparams*. The parameter list is stored in the buffer *parambuf* of length *paramlen*.

Buffer format:

```
for each parameter:
    name\0
    new value\0
```

To store or retrieve the parameters from the buffer, use the *buf_put_string* and *buf_get_string* functions (**buffer[L]**).

All values, independent of their meaning (long, char, string...) are always stored as strings!

Server interface

For the server side, no library functions exist. It's the work of the programmer to implement the *std_params* interface in the server service loop.

You must add the *STD_GETPARAMS* and *STD_SETPARAMS* commands in your service loop and extract the following parameters:

```
hdr.h_command = STD_GETPARAMS
hdr.h_extra   = maximal size of parambuf on
                the client side
buffer

-> put the parameters in the right order with
    buf_put_string

return:
hdr.h_extra = paramlen (size of parambuf)
hdr.h_size  = nparams  (total number of parameters)

hdr.h_command = STD_SETPARAMS
hdr.h_extra   = paramlen
hdr.h_size    = nparams

buffer

-> get the parameters from the buffer with
    buf_get_string
```

Examples

Client side

Setting only one parameter each time shows the next example. The server capability was retrieved with the *name_lookup* function for example (**name[L]**):

```
errstat
do_std_setparam(cap, param, value)
capability *cap;
char *param;
char *value;
{
    errstat err;
    size_t length;
    char *buf;
    char *bufp, *end;

    length = strlen(param) + strlen(value) + 2;
    buf = (char *)malloc(length);
    if (buf == NULL) {
        return STD_NOSPACE;
    }

    bufp = buf;
    end = &buf[length];

    bufp = buf_put_string(bufp, end, param);
    bufp = buf_put_string(bufp, end, value);

    if (bufp == NULL) {
        err = STD_NOSPACE;
    } else {
        err = std_setparams(cap, buf, (int)length, 1);
    }
    free(buf);

    return err;
}
```

Getting the parameter list from the server shows the next example code:

```
struct param {
    char *par_name;
    char *par_type;
    char *par_descr;
    char *par_value;
};

errstat
do_get_params(cap, ret_params, ret_nparams)
capability *cap;
struct param **ret_params;
int *ret_nparams;
{
    errstat err;
    int i, paramlen, numparams;
    struct param *params;
    char *parambuf;
    char *bufp, *end;

    parambuf = (char *)malloc((size_t)MAX_PARAM_LEN);
    if (parambuf == NULL) {
        return STD_NOSPACE;
    }
}
```

```
    }

    err = std_getparams(cap,
                       parambuf, MAX_PARAM_LEN,
                       &paramlen, &numparams);

    if (err != STD_OK) {
        return err;
    }

    params = (struct param *)
        malloc((size_t)(numparams *
                       sizeof(struct param)));

    if (params == NULL) {
        free(parambuf);
        return STD_NOSPACE;
    }

    bufp = parambuf;
    end = &parambuf[paramlen];

    for (i = 0; i < numparams && bufp != NULL; i++) {
        char *s;

        bufp = fetch_string(bufp, end,
                            &params[i].par_name);
        bufp = fetch_string(bufp, end,
                            &params[i].par_type);
        bufp = fetch_string(bufp, end,
                            &params[i].par_descr);
        bufp = fetch_string(bufp, end,
                            &params[i].par_value);
    }

    free(parambuf);

    if (i < numparams) {
        free(params);
        /* also freeing the strings here isn't worth
         * the trouble
         */
        return STD_NOSPACE;
    }

    *ret_params = params;
    *ret_nparams = numparams;
    return STD_OK;
}

/*
 * Get server parameters (name, type, description,
 * value).
 */

static char *
fetch_string(buf, end, str)
char *buf, *end;
char **str;
{
    char *bufp, *s;

    bufp = buf_get_string(buf, end, &s);
    if (bufp != NULL) {
```

```
*str = (char *)malloc((size_t)(strlen(s) + 1));
if (*str == NULL) {
    fprintf(stderr, "could not allocate
                    string\n");
    exit(1);
}
(void)strcpy(*str, s);
}
return bufp;
}
```

Server side

The next example code was extracted from the om server source code.

The service loop:

```
{
    ...

    n = getreq(&h, buf, OMSIZE);
    if (ERR_STATUS(n)) {
        err = ERR_CONVERT(n);
        if (err == RPC_ABORTED) {
            continue;
        }
        sys_log(SYS_FATAL,
                "%s: getreq failed!\n");
        my_exit(1);
    }
    switch (h.h_command) {

        ...

        case STD_GETPARAMS:
        {
            int N; /* in redeclared (h_extra) */
            int paramlen; /* out redeclared (h_extra) */
            int nparams; /* out redeclared (h_size) */
            /* out parambuf not declared */
            if (n != 0) {
                h.h_status = STD_ARGBAD;
                break;
            }
            N = (short) h.h_extra;
            h.h_status = om_std_getparams(&h,
                                         buf,
                                         /* bound: */ OMSIZE,
                                         N,
                                         &paramlen,
                                         &nparams);
            if (h.h_status == 0) {
                h.h_extra =
                    (unsigned short int) paramlen;
                h.h_size =
                    (unsigned short int) nparams;
                reply_size = paramlen;
                /* for parambuf */
            }
            else

```

```
        reply_size=0;

        break;
    }

    case STD_SETPARAMS:
    {
        int paramlen; /* in redeclared (h_extra) */
        int nparams; /* in redeclared (h_size) */
                    /* in parambuf not declared */
        paramlen = (short) h.h_extra;
        nparams = (short) h.h_size;
        h.h_status = om_std_setparams(&h,
                                     buf,
                                     /* bound: */ OMSIZE,
                                     paramlen,
                                     nparams);

        reply_size=0;
        break;
    }

    default:
        h.h_status = STD_COMBAD;
        break;
}
putrep(&h, buf, reply_size);
}
```

And the *std_getparams* interface implementation:

1. Convert a long variable to string and append the string to the parambuf

```
/*
 * std_getparams implementation
 */

static char *
put_long_var(buf, end, name, min, max, type, descr, value)
char *buf, *end; /* buffer start and end */
char *name; /* name of the variable */
long min, max; /* minimum and maximum value (to
                create type string) */
char *type; /* the "currency" of the type */
char *descr; /* a short description telling what
               the param does */
long value; /* the current value */

/*
 * Put the string representation of an integer variable
 * in a buffer,
 * in the format required by std_getparams().
 */

{
    register char *bufp = buf;

    /* name\0 */
    bufp = buf_put_string(bufp, end, name);

    /* type\0 */
```

```
if ((bufp != NULL) && (end - bufp >= 20 +
    strlen(type))) {
    /* type is actually a subrange: */
    sprintf(bufp, "%ld..%ld %s", min, max, type);

    /* skip value until after nul byte */
    bufp = strchr(bufp, '\0');
    bufp++;
}

/* description\0 */
bufp = buf_put_string(bufp, end, descr);

/* value\0 */
if ((bufp != NULL) && (end - bufp >= 10)) {
    sprintf(bufp, "%ld", value);

    /* skip value until after nul byte */
    bufp = strchr(bufp, '\0');
    bufp++;
}

return bufp;
}
```

2. Setup and build the parambuf list

```
errstat
om_std_getparams(h,buf, max_buf, n, len, nparams)
header *h;
char *buf;          /* to put params in */
int max_buf;
int n;             /* length of client buffer */
int *len;         /* length of buffer returned */
int *nparams;    /* number of parameters returned */
{
    int      np;
    char     *bufp, *end;
    errstat  err;

    if (n < 0) {
        return STD_ARGBAD;
    }
    if (n > max_buf) {
        n = max_buf;
    }
    bufp = buf;
    end = &buf[n];

    np = 0;
    bufp = put_long_var(bufp, end, NAME_OMSLOWDOWN,
        0,1, "0/1",
        INFO_OMSLOWDOWN,
        om_slowdown);
    np++;
    bufp = put_long_var(bufp, end, NAME_OMWAIT,
        0,1, "0/1",
        INFO_OMWAIT,
        om_wait);
    np++;
    bufp = put_long_var(bufp, end, NAME_OMSLEEP,
```

```

                                MIN_OMSLEEP, MAX_OMSLEEP, "sec",
                                INFO_OMSLEEP,
                                delay_between_start_of_passes);
    np++;

    if (bufp == NULL) {
        return STD_OVERFLOW;
    }

    *nparams = np;
    *len = bufp - buf;
    return STD_OK;
}

```

Now the *std_setparams* implementation:

1. Set a server parameter variable with the string fetched from the parambuf list. Do additional sanity and bound checks.

```

/*
 * std_setparams implementation
 */

static errstat
set_long_var(var, strvalue, min, max)
long *var;
char *strvalue;
long min, max;
{
    char *after_val;
    long value;

    /* first convert string to integer */
    value = strtol(strvalue, &after_val, 0);
    if (after_val == strvalue) {
        /* could not get integer out of strvalue */
        return STD_ARGBAD;
    }

    /* perform range check */
    if (value < min || value > max) {
        return STD_ARGBAD;
    }

    *var = value;
    return STD_OK;
}

```

2. And extract all parameters from the parambuf list

```

errstat
om_set_param(param, val)
char *param;
char *val;
{
    errstat err = STD_NOTFOUND;
    /* unknown parameter name */

    if (strcmp(param, NAME_OMSLOWDOWN) == 0) {
        err = set_long_var(&om_slowdown, val, 0, 1);
    }
}

```

```
} else if (strcmp(param, NAME_OMSLEEP) == 0) {

    err = set_long_var(&delay_between_start_of_passes,
        val,MIN_OMSLEEP, MAX_OMSLEEP);

    if(err == STD_OK)
        omsleepchange=1;
} else if (strcmp(param, NAME_OMWAIT) == 0) {
    int    iarg;

    err = set_long_var(&iarg,val,0,1);
    if(err == STD_OK && om_wait == 0 && iarg == 1)
    {
        om_wait = 1;
        sema_down(&om_semawait);
    }
    if(err == STD_OK && om_wait == 1 && iarg == 0)
    {
        om_wait = 0;
        sema_up(&om_semawait);
    }
} else
    err=STD_ARGBAD;

if (err != STD_OK) {
    sys_log(SYS_ERROR,
        "%s: refused to set parameter \"%s\" (%s)\n",
        progname,param, err_why(err));
}

return err;
}
```

See also

rpc[L], buffer[L], name[L]

uniqport - generate a random port

(in libraries: libamoeba.a, libamunix.a)

Name

uniqport - generate a random port

Synopsis

```
#include "module/rnd.h"

void uniqport(p)
void uniqport_reinit()
```

Description

Routines to generate unique RPC and group communication ports and check fields.

uniqport

```
void
uniqport(p)
port * p;
```

Uniqport produces a non-NULL, random *port*. In general the result should be unique to the system. It is useful for servers that need to create new check fields or a new port to listen to. It uses a random number server to get the seed for the random number generator.

uniqport_reinit

```
void
uniqport_reinit()
```

Uniqport_reinit causes the next call to *uniqport* to choose a new seed for the generation of random numbers.

Environment Variables

If set, `RANDOM` determines which random number server to use when generating the seed. Otherwise the default random server is used.

Warnings

It is possible that the port generated is not unique within the system. If it is to be used as a *port* then it may be necessary to test to see if anyone is already listening to that port before using it. This is done by doing a *std_info* (see *std(L)*) on the port with a short locate timeout (say 2 seconds).

Example

```
#include "amoeba.h"
#include "module/rnd.h"

port listen;

uniqport(&listen);
```

See Also

rnd[L], **rpc[L]**.

Table of Contents

Client/Server Tutorial	2
Writing Servers and Clients.....	2
Servers	3
Clients	7
Using AIL	9
Writing a Server with AIL	10
Writing a Client with AIL.....	11
Interface stubs for remote procedure calls	13
Name.....	13
Synopsis.....	13
Description.....	13
Ports and Capabilities	14
Message Structure.....	15
Error Codes and Failure Semantics	18
Signals.....	20
Example	21
See Also	23
RPC server example.....	25
ar - ASCII representation of common Amoeba types	28
Name.....	28
Synopsis.....	28
Description.....	28
Functions.....	28
Examples.....	30
See Also	31
buffer - getting and putting data in arch-independent format	32
Name.....	32
Synopsis.....	32
Description.....	32
Example	38
name - directory manipulation.....	39
Name.....	39
Synopsis.....	39
Description.....	39
Errors	40
Functions.....	40
Examples.....	43
See Also	44
priv2pub - convert private to public port	45
Name.....	45
Synopsis.....	45
Description.....	45
See Also	45
prv - manipulate capability private parts	46
Name.....	46

Synopsis.....	46
Description.....	46
Functions.....	46
Warnings	47
Example	47
See Also	48
std_params programming interface	49
Name.....	49
Synopsis.....	49
Description.....	49
Programming Interface	49
Client interface.....	49
Server interface.....	50
Examples.....	50
See also	57
uniqport - generate a random port	58
Name.....	58
Synopsis.....	58
Description.....	58
Environment Variables.....	59
Warnings	59
Example	59
See Also	59