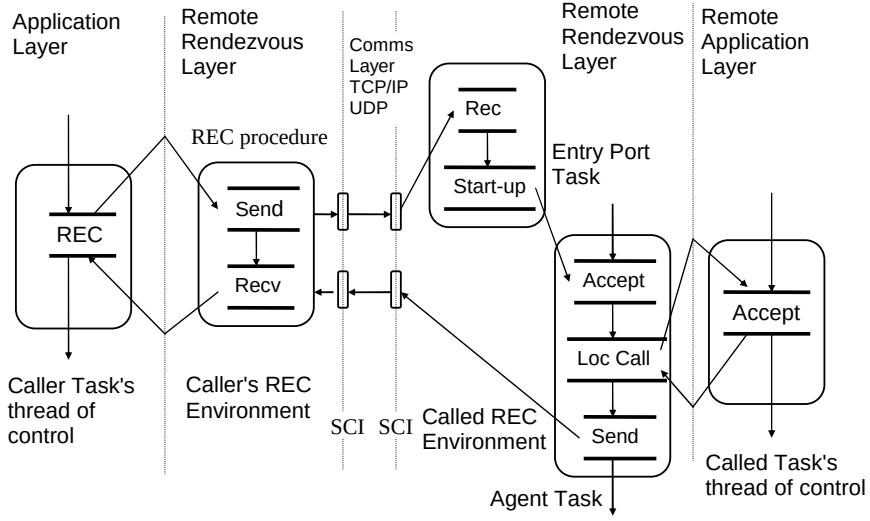
DISTRIBUTED PROCESSING SOFTWARE ENVIRONMENTS

The environment or framework for supporting distributed processing can usefully be divided into three main approaches:

- Distributed processing is supported via a concurrent programming language through a transparent interface (e.g. REC in Ada 83 or RSC in Ada 95, other examples are provided in Occam and RMI in Java).
- Distributed processing is supported via a non-concurrent language through a library of routines or APIs that can be called to provide distributed communication primitives (e.g. RPC, PVM and even NFS).
- Distributed processing is supported via *middleware* - language independent object brokers (e.g. CORBA, DCOM and WWW specific through HTTP/CGI).

Remote Entry Calls (REC) in Ada 83

This approach was developed by Atkinson (1988) in the DIADEM (Distributed Ada Demonstrated) project. It provides a layered communication model for the Ada remote rendezvous:



Note that the aim of this approach is to make the only layer visible to the calling task environment the remote rendezvous layer → allows the lower level layers to be replaced without changing the application code.

The remote rendezvous layer maintains a map of software (or virtual) nodes and physical nodes which includes the physical port address. Also note that no explicit mechanism is provided for the distribution of code to the virtual nodes.

Remote Subprogram Calls (RSC) in Ada 95

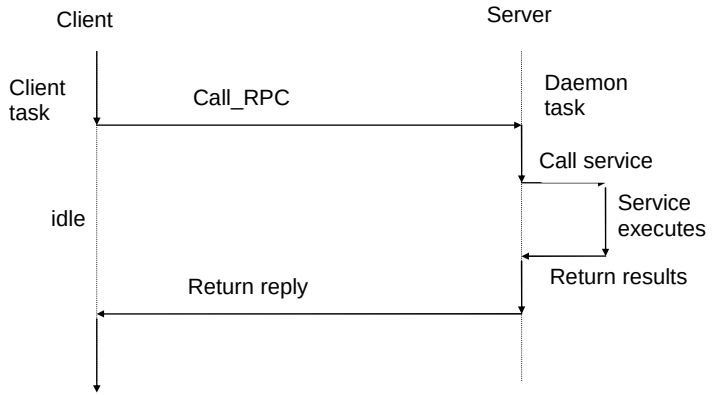
Ada 95 effectively removes the software management problem inherent in code distribution by defining *partitions*. All code for the distributed system is contained in one logical program, and the target nodes for the code are defined as separate partitions in the one program. All references to data or subprograms calls across partitions are *remote accesses*.

A *remote subprogram call* (RSC) invokes the execution of a subprogram in another partition. A partition communication subsystem (PCS) provides for inter-partition communication through a package called **System.RPC** which uses the system's Remote Procedure Call primitives.

A *Partition_ID* is used to access each partition and synchronous RPC (*Do_RPC*) and asynchronous RPC (*Do_APC*) calls are supported. An **Ada.Streams** package is also provided to support the naming of remote subprograms, the parameters sent to a remote subprogram and the results returned from a remote subprogram.

Remote Procedure Calls (RPC) under UNIX

The required communication layers are provided by system RPC services, which are based on a client/server model. The communication protocol is invisible in RPC, but knowledge of the remote processor, RPC program, RPC procedure and version number is required:



Under UNIX a typical RPC call is:

char **host*, **in*, **out*;
u_long *prognum*, *versum*, *procnum*;
xdrproc_t *inproc*, *outproc*;

callrpc (*host*, *prognum*, *versnum*, *procnum*, *inproc*, *in*, *outproc*, *out*);

where:
host is the address of host machine name
prognum and *procnum* are the program and procedure number defined in the server
versnum is the version number of the program
in and *out* are addresses of the data structures holding the remote procedure parameters and return results respectively
inproc and *outproc* are functions for encoding/decoding the procedures parameters and its results via XDR (eXternal Data Representation)

The RPC model couples the application far more tightly to the underlying communication layer than the REC or RSC models and is less portable, but has the advantage of less run-time overhead.

Parallel Virtual Machine (PVM)

PVM is a set of libraries that emulates a hetrogeneous parallel and/or distributed computing framework.

Principle Features:

- User-configured host pool - application tasks execute on a dynamically selected set of machines
- Low-level access - application programs can exploit specific capabilities of particular machines
- Task-based parallelism - independent sequential threads of control alternating between computation and communication
- Explicit message-passing model - tasks cooperate by explicitly sending and receiving messages
- Heterogenous machine architecture - support for different machine types, networks, data types and applications
- Multiprocessor support - uses native message passing on multiprocessors

PVM system components:

- daemon (*pvmd3*) - runs on all machines comprising the virtual machine
- library of PVM interface routines (*libpvm3.a*) - set of primitives needed for intertask cooperation, creating tasks and modifying the virtual machine

PVM tasks:

- Each task is identified by a *task identifier* (TID)
- Applications or parts of applications may be composed of a *group of tasks* and the task joining a group is assigned an *instance identifier* for that group

PVM programming methodology:

- All tasks correspond to precompiled programs and their object files are accessible to all machines in the pool
- Execution is initiated by running a master program which usually starts all other PVM tasks (although tasks can be started manually)
- Tasks interact with each other via message passing, explicitly identifying each other with the system assigned TIDs

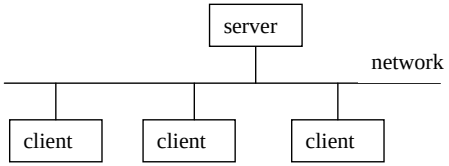
Example:

```
main(){
    int cc, tid, msgtag;
    char buf[100];

    printf("Task t%x \n", pvm_mytid());
    cc = pvm_spawn("hello_other", (char**)0, 0,"",1,&tid);
    if (cc ==1 ) {
        msgtag = 1;
        pvm_recv(tid, msgtag);
        pvm_upkstr(buf);
        printf("From t%x: %s\n", tid, buf);
    } else
        printf("Cant start hello_other\n");
    pvm_exit();
}
```

```
#include "pvm3.h"
main() {  /* hello_other program mainline */
    int ptid, msgtag;
    char buf[100];

    ptid = pvm_parent();
    strcpy(buf,"Hello world from ");
    gethostname(buf + strlen(buf), 64);
    msgtag = 1;
    pvm_initsend(PvmDataDefault);
    pvm_pkstr(buf);
    pvm_send(ptid, msgtag);
    pvm_exit();
}
```


NETWORKED FILE SYSTEMS

The first networked file systems required users to log onto a central machine on which the shared files were located. As these file servers quickly became overloaded, a way was needed to share files on several machines.



Unix System V introduced *RFS* which had good UNIX semantics but poor performance. Research at CMU lead to the *Andrew file system*, which was commercialised and eventually became part of the Distributed Computing Environment (DCE). The most successful protocol has been the *network file system (NFS)* designed and implemented by Sun Microsystems.

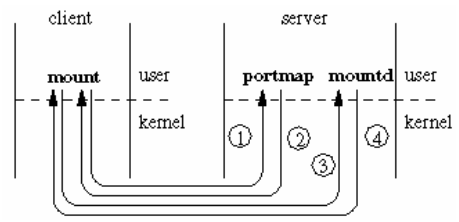
NFS approaches operate as a client server application:

- The server receives *remote-procedure-call* (RPC) requests from its various clients.
- The parameters must then be *marshalled* into a message.
- Marshalling includes replacing pointers by the data to which they point and converting binary data to the network byte order.
- The message is unmarshalled at the server and processed as a local file system operation.
- The result must be similarly marshalled and sent back to the client. which then splits up the result and returns it to the calling process.

The NFS protocol is also *stateless*:

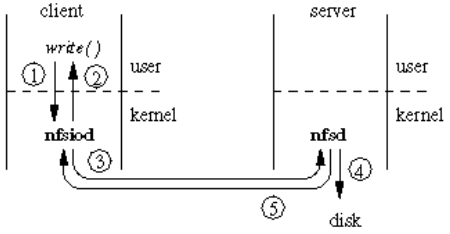
- The server does not maintain any information about which clients it is serving or about the files that they currently have open
- Read requests include the identity of the user, file handle, offset in the file to begin the read, and the number of bytes to be read.
- The server then opens the file, verifys that the user has read permission, seeks to the appropriate point, reads the contents, and closes the file.
- In practice, the server also caches recently accessed file data but the file handle allows the server to reopen the file if it is pushed from the cache.

The interactions between the client and server daemons when a remote file system is mounted are shown below:



1. The client's mount process sends a message to the well-known port of the server's **portmap** daemon, requesting the port address of the server's **mountd** daemon.
2. The server's **portmap** daemon returns the port address of its server's **mountd** daemon.
3. The client's **mount** process sends a request to the server's **mountd** daemon with the pathname of the file system that it wants to mount.
4. The server's **mountd** daemon requests a file handle for the desired mount point from its kernel (the file handle is returned to the client's **mount** process if successful and the *mount* system call is performed).

Interaction at I/O completion:



1. The client's process does a *write* system call.
2. The data to be written is copied into a kernel buffer on the client, and the *write* system call returns.
3. An **nfsiod** daemon awakens inside the client's kernel, picks up the dirty buffer, and sends the buffer to the server.
4. The incoming write request is delivered to the next available **nfsd** daemon running inside the kernel on the server. The server's **nfsd** daemon writes the data to the appropriate local disk, and waits for the disk I/O to complete.
5. After the I/O has completed, the server's **nfsd** daemon sends back an acknowledgment of the I/O to the waiting **nfsiod** daemon on the client. On receipt of the acknowledgment, the client's **nfsiod** daemon marks the buffer as clean.

Server Message Block (SMB)

SMB is a protocol for sharing files, printers, serial ports, and communications abstractions such as named pipes and mail slots between computers. SMB first appeared in the IBM PC (1985) and then as a Microsoft's network file sharing protocol (1987). Microsoft and others subsequently developed the protocol further.

SMB is a client-server, request-response protocol. The only exception being when the client has requested opportunistic locks (oplocks) and the server subsequently has to break an already granted oplock because another client has requested a file open with a mode that is incompatible with the granted oplock. In this case, the server sends an unsolicited message to the client signalling the oplock break.

Servers make file systems and other resources (printers, mailslots, named pipes, APIs) available to clients on the network.

Clients connect to servers using TCP/IP (actually NetBIOS over TCP/IP as specified in RFC1001 and RFC1002), NetBEUI or IPX/SPX:

<table>
<tr><td colspan="3">OSI</td><td colspan="2"></td><td>TCP/IP</td></tr>
<tr><td>Application</td><td colspan="4" rowspan="2">SMB</td><td rowspan="2">Application</td></tr>
<tr><td>Presentation</td></tr>
<tr><td>Session</td><td>NetBIOS</td><td rowspan="3">NetBEUI</td><td>NetBIOS</td><td>NetBIOS</td><td rowspan="2"></td></tr>
<tr><td>Transport</td><td>IPX</td><td>DECnet</td><td>TCP/UDP</td><td>TCP/UDP</td></tr>
<tr><td>Network</td><td></td><td></td><td>IP</td><td>IP</td></tr>
<tr><td>Link</td><td>802.2, 802.3,802.5</td><td>802.2/ 802.3,802.5</td><td>Ethernet V2</td><td>Ethernet V2</td><td>Ethernet or others</td></tr>
</table>

After connection, clients send commands (SMBs) to the server to access file shares, open files, read and write files, etc.

The actual protocol variant the client and server will use is negotiated using the *negprot* SMB which must be the first SMB sent on a connection. The first protocol variant was the Core Protocol which could handle a basic set of operations that included:

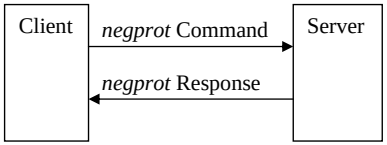
- connecting to and disconnecting from file and print shares
- opening and closing files and print files
- reading and writing files

- creating and deleting files and directories
- searching directories
- getting and setting file attributes
- locking and unlocking byte ranges in files.

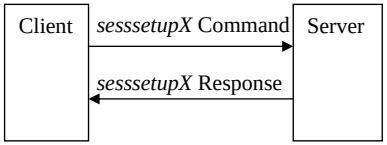
Samba and Microsoft's Common Internet File System (or CIFS) use the latest variants of this protocol. There are many other implementations by different vendors (some variants introduced new SMBs and some changed the format of existing SMBs or responses).

A typical SMB protocol exchange

The client sends a *negprot* SMB to the server, listing the protocol dialects that it understands. The server responds with the index of the dialect that it wants to use, or 0xFFFF if none were acceptable. Dialects more recent than the Core and CorePlus protocols supply information in the response to indicate their capabilities.



The client can then proceed to logon to the server with a *sesssetupX* SMB. The response indicates whether or not they have supplied a valid username password pair and if so, can provide additional information. The UID in the response must be submitted with all subsequent SMBs on that connection to the server.



The client can then proceed to connect to a tree. The client sends a *tcon* or *tconX* SMB specifying the network name of the share that they wish to connect to, and the server responds with a TID that the client will use in all future SMBs relating to that share. Having connected to a tree, the client can now open a file with an open SMB, followed by reading it with read SMBs, writing it with write SMBs, and closing it with close SMBs.