2

Conf-9004179--3

LBL-22767

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

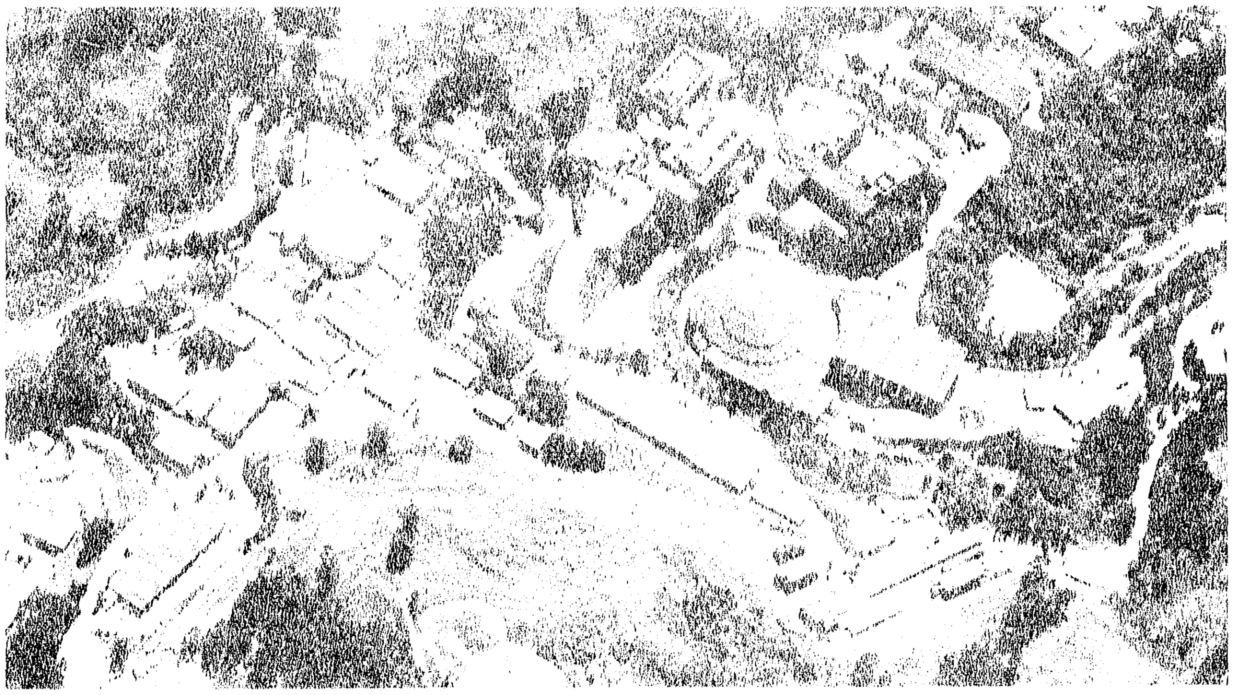
Information and Computing Sciences Division

Presented at the Cray User's Group, Toronto, Canada, April 9–13, 1990, and to be published in the Proceedings

Video Movie Making Using Remote Procedure Calls and 4BSD Unix Sockets on Unix, UNICOS, and MS-DOS Systems

D.W. Robertson, W.E. Johnston, D.E. Hall, and M. Rosenblum

March 1990



Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098

LBL--22767

DE92 010255

March 1, 1990

VIDEO MOVIE MAKING USING REMOTE PROCEDURE CALLS AND 4BSD Unix SOCKETS ON Unix†, UNICOS†, AND MS-DOS SYSTEMS

David W. Robertson, William E. Johnston, Dennis E. Hall, and Mendel Rosenblum[1]

Advanced Development Group
Information and Computing Sciences Division
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

ABSTRACT

We describe the use of the Sun Remote Procedure Call and Unix socket interprocess communication mechanisms to provide the network transport for a distributed, client-server based, image handling system. Clients run under Unix or UNICOS and servers run under Unix or MS-DOS. The use of remote procedure calls across local or wide-area networks to make video movies is addressed.

[1]The first three authors can be reached via USMail at: Lawrence Berkeley Laboratory, Bldg. 50B, Rm. 3238, Berkeley, CA 94720. Email addresses are: dwrobertson@lbl.gov, wejohnston@lbl.gov, dehall@lbl.gov. The work presented in this paper is supported by the Director, Office of Energy Research, Office of the Scientific Computing Staff, of the U.S. Department of Energy, under contract DE-AC03-76SF00098. Any conclusions or opinions, or implied approval or disapproval of a company or product name are solely those of the authors and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory, or the U.S. Department of Energy. Trademarks are acknowledged by †.

[illegible]

1. INTRODUCTION

The computer modelling of physical phenomena often results in a sequence of images, which must be displayed fairly rapidly to gain insight into the dynamics of the process being modelled. Displaying such a sequence rapidly enough requires either the computational power for real-time display or the ability to record the results a frame at a time and then play back the frames in real time. In the past, film recorders were used to record movies of the results of computer modelling on 16 mm or 35 mm film. Their use involved the expenditure of much time, money, and equipment. The film medium is no longer in the mainstream of most AV operations. The recording process is less time-consuming if videotape instead of film is used. However, the equipment necessary to record a single computer-generated image at a time on videotape is usually expensive. The least expensive video animation system involves using IBM PC compatible microcomputers.

The philosophy of the LBL Video Animation Project has been that scientific movies resulting from computer simulation would be made more frequently if scientists had access to a low-cost video movie making system. Movies produced with this system, while not of broadcast quality, would enable insight into the results of computer modelling. The system put together to reach this goal is described in Johnston, et al. [7].

The main hardware components of this video workstation include an IBM PC compatible microcomputer, a 16-bit color frame buffer, and an Ethernet controller. Video recording is done through the use of a video animation controller and a videotape recorder (VTR), or the use of a videodisk player. The Ethernet controller has associated software allowing communication over the network using the TCP/IP [17] protocol. The frame buffer is supplied with software implementing graphics primitives (i.e. for points, lines, etc.). The video animation controller is supplied with software that permits the use of simple commands like "edit in a certain frame at location xxx and record this frame until location yyy". The videodisk player used is a Panasonic TQ2026 model. It has a repertoire of one-line commands that can be sent to it over a serial line that allow seeking to a specified frame, playing at half speed, etc. Several current videodisk models have an animation controller associated with them.

2. MOTIVATION FOR USING REMOTE PROCEDURE CALLS

The video workstation uses an IBM PC compatible to allow inexpensive video movie making. However, both the modelling of physical phenomena, and the generation of images representing the modelling are often too time-consuming or too memory-intensive to perform on such a machine. The problem thus becomes how to deliver images resulting from scientific modelling to the video workstation for display, and how to control the video recording from the more powerful computer generating the images.

The first movie made using the video workstation, depicting 2D flow over a backward-facing step, was produced in a convoluted manner. The flow field representing the numerical result of fluid modelling was generated on a Cray XMP-48 [18]. It was moved by tape to a VAX-VMS system, and then to a VAX-Unix system. The positions of flowing particles used to ascertain the dynamics of the flow were determined from the flow field at each time step with software written and run on the VAX-780 Unix system. The positions and colors of the particles, along with the position of the boundary, were recorded in a graphics metafile. This metafile was then sent to the PC using FTP [3]. The amount of data being sent over the network was so large that the movie was made in sections. After transfer, a program [7] was started on the PC to read the metafile, display the colored particles, and control the recording process.

This three-step process requires the user to know something about two operating systems, and the file transfer program that operates between them. Another drawback is that human intervention is required after each step is completed in order to start the next step, making it difficult to leave the process running unattended for long periods.

The three-step mechanism could be avoided if the process generating the image on the more powerful CPU could communicate directly with a process on the PC that controls display and recording on the video workstation. However, inter-process communications (IPC) programming is difficult. In Unix, for example, the low-level IPC primitives are sockets and calls to send and receive data. Limited to them, the user has to develop higher-level ways of converting from one machine data representation to another, ensuring security, dealing with network byte ordering, and various other details [14].

3. RPC'S BETWEEN CLIENT AND SERVER

An attractive approach to hiding these details from the user is that of remote procedure calls (hereafter referred to as RPC's). A remote procedure call is similar to a conventional procedure call, but is made between processes which are potentially on separate machines. An RPC made by one machine (the client) causes the invocation of a procedure on another (the server) through the mediation of the RPC package. The RPC package communicates the arguments across the network, handles data format conversion, and finds the desired procedure on the server [1]. Another way of looking at the client-server relationship is that of a master process (on the client) controlling the slave process (on the server). The server performs no actions on its own; it waits for incoming RPC calls from the client to tell it what to do. The implementation described here uses the Sun RPC library [14]. The Sun RPC implementation uses the Berkeley Unix socket interface to TCP/IP to provide the underlying network transport [14, 17].

The client must first specify the address of the server to communicate with it. Sun RPC uses the Internet mode of addressing. In this mode the address of a socket (communication point on the client and server) is composed of a pair of numbers: the address of the machine on the network, and a receiving process identifier, the port. The two protocols of communication which are used by Sun RPC are UDP (User Datagram Protocol) and TCP (Transmission Control Protocol). TCP is a stream, or connection oriented protocol, and data transmitted by a single RPC using TCP can be of any length up to $2^{31} - 1$ bytes (in Sun RPC) [4]. UDP is a datagram protocol. The data transferred by a single RPC is limited by the maximum packet length. This length is system dependent, but modern Unix systems, including Sun, typically allow 8K [14]. UDP transmission is not error-free, while TCP is guaranteed to be reliable, at the cost of error-checking overhead [17].

Because internal data representation on various machines differs, data is converted to XDR (eXternal Data Representation) before sending it over the network. Machine dependencies may extend beyond different byte ordering, for example, for floating point data types. Once it reaches the other machine, RPC uses XDR routines to convert from the XDR representation to that particular machine's representation.

The main scenario to be discussed in this paper is that of running an RPC animation server on an IBM PC compatible under the MS-DOS operating system. The server is started on the PC and listens for incoming RPC calls to service. The client (running on a Sun or a Cray) performs the necessary protocol to establish communication with the PC and then uses RPC's to control the video workstation. Once it is done, it closes the connection, allowing other clients to use the PC video server.

While the PC does not run Berkeley Unix, the Ethernet board [13] has associated software which emulates sockets, though the semantics of the operations on sockets are slightly different. In addition, a Microsoft compiler and run-time library is used, which provides a similar C environment to that of Berkeley Unix. The availability of these two items on the PC made possible porting most of the server portion of the Sun RPC package to the PC. The details of the changes necessary to make the port, which requires some familiarity with the Berkeley socket library, are given in Appendix B.

A tutorial description of Sun's implementation of RPC's and of BSD inter-process communication is also given in Appendix B. Knowledge of Sun's implementation would help in understanding how a port might be achieved with a different Ethernet controller installed on the PC.

Alternatively, the server can run on a window based workstation, such as a Sun workstation. On the workstation the image is displayed in a window on the screen. There can be more than one window open, so more than one server can run on the workstation.

One problem was encountered because the window-based servers utilize an 8-bit color look-up table. There is no difficulty in decompressing an image when color quantization had been used; color quantization generates the look-up table, and converts each pixel color into an index into the table. However, flickering between frames can occur if the color map changes. Also, since the color map changes, frame-to-frame differencing cannot be used on this window-based server. (The PC workstation server does not use a color map; pixels that do not change are not written over from frame to frame.)

The compressed raster images from the rendering module can be saved on disk by the window-based server. If the images are part of a movie, then when the movie frame generation is compl[illegible] preview program (Anima) can be used to display a portion or the entire sequence on the workstation in forward or reverse at various speeds. The speed of decompressing and displaying a 512x400 image is several frames per second on a Sun 4-110. The color map problems mentioned above are more noticeable if not corrected, because of the rate of display.

4. VIDEO MOVIE MAKING USING RPC'S

To make a video movie, each image in the sequence of images to make up the movie must be displayed in the hardware frame buffer attached to the PC, at which point the associated video recorder can record that frame. The approach taken on the client to produce an image describing the results of scientific modelling is to scan convert graphics primitives (points, lines, text, and polygons) into a software frame buffer residing in main memory. See Robertson [15] for the details. Alternatively, there is an interface to accept an existing image (for example from remote sensing). In all cases, the result on the client is a software representation of a specific frame buffer format (TARGA [21] in the current implementation). This software frame buffer is transferred to the workstation using RPC's, either directly after the image generation step or from Sun disk using Anima (see previous section), and written into the PC frame buffer several scan lines at a time.

The TARGA software frame buffer requires storage of about 400 kilobytes of data, and the bandwidth observed in the Ethernet connecting a Sun 4-110 and the PC server is roughly 50-60 kilobytes/second with the TCP protocol. This network/system bandwidth, taken together with the compression-decompression time, usually makes it advantageous to compress the frame buffer before transmission. A detailed explanation of the types of compression used is given in Texier [22], Johnston [8], and Robertson [15]. After the data is compressed, the RPC package calls a special XDR routine, *xdr_opaque*, to take the data and place it in a buffer with no conversion for transferral (at this point the data is of type **char**, so byte ordering is not an issue).

4.1. THE MOVIE-MAKING CLIENT

Most of the work involved in using RPC's to display and record images was done to implement the server on the PC based video workstation. Only minor changes had to be made to the way Sun RPC's are used on the client side. No changes had to be made to the Sun RPC software at the lowest user level and below. The Sun library can be linked in as is. Some changes are made in the way Sun RPC is utilized.

The client runs on the user system and provides the user interface to the movie system. Most of the user-level calls on the client that control the video server look like regular library calls. There is no evidence that an RPC is being used unless the RPC call fails. The client's use of RPC's is not exactly like RPC calls in Sun RPC because in Sun RPC a connection is opened and closed every time a RPC is made when the TCP protocol is used. The way Sun RPC was modified, in porting it to the PC, made this way of handling the closing of a connection undesirable (see Appendix B). Instead, the connection is created once at the beginning of a session involving the video server, is reused for all RPC's, and is only destroyed when the client process is finished. Similarly, only one server program (dispatch routine) is available on the PC, since the connection must be closed to communicate with a different dispatch routine.

The movie-making client provides a library of subroutines upon which a variety of higher-level graphics-generation algorithms can sit. The client has run on several different systems that provide the 4BSD Unix, socket based IPC mechanisms, and enough of the Unix run-time library to support the Sun RPC library. (For example, it has run under 4.3BSD Unix and Cray UNICOS.) For the availability of code giving an example of the client calls in a program, and the code for the underlying client, the server, and the modified Sun RPC library, see Appendix A.

The client module has been made as data-independent as possible. It neither knows nor cares how a particular image has been generated; it only needs to know what type of compression to apply and where to send the resulting byte stream. The content, independent of the format, can be a remote sensed image, the result of scan converting polygons making up an object, etc. The storage format of this image is currently in a 15 bit TARGA format (5 bits R, 5 bits G, and 5 bits B).

Since this module does not know the characteristics of the image that it is dealing with, it is not possible to automatically select an optimal compression technique. Several compression techniques are provided, some lossy, some lossless, some suitable for synthetic images (those generated by scan converting the graphics primitives generated by many visualization algorithms) and some suitable for natural (remote sensed) images. Generally speaking, the compression techniques are: (1) block truncation coding (BTC) [2]; (2) Heckbert's median cut color map algorithm in conjunction with BTC [5]; (3) frame-to-frame differencing [22]; and (4) Lempel-Ziv coding [10]. A detailed analysis of the characteristics of all of the useful combinations of these is too tedious here [22], but a few rules of thumb are useful. If the final display is to be a video movie, fairly inexact compression works well. Usually a combination of BTC and the color map algorithm is used[2], with Lempel-Ziv cascaded if the images are being sent via a wide-area network. For natural images color map compression and/or Lempel-Ziv compression are sometimes useful.

Usually images to be displayed by the animation previewer, mentioned in section 3, are compressed using only BTC and a color map by the client. Lempel-Ziv decompression for each frame will slow the playback speed by more than half. If a movie is not going to be viewed for some period of time, the UNIX "compress" function, which performs Lempel-Ziv compression, is used on the entire sequence, which has resulted in an observed compression rate on the already compressed images of between 2 and 40 to one.

The typical sequence of client calls is (1) *scry_open*; (2a) *scry_set_compress*, and/or (2b) *scry_set_record* (if the image(s) will be recorded), and/or (2c) *scry_set_copynum* (if the image(s) will be recorded on videotape); (3) *scry_send_frame* for each image to be displayed; and (4) *scry_close*. 2a through 2c do not necessarily have to occur in that order.

scry_set_record provides a *PREVIEW* option. Often a user will want to look at the images to make up a movie before recording them, in case there are mistakes. With the *PREVIEW* option set,

[2] BTC encoding divides the frame buffer (raster image of 15 bits per pixel) into 4x4 blocks. Two "best" colors are chosen to represent the block. These two colors, along with a bitmap which has 1's for pixels closer to one color and 0's for pixels closer to the other, are the compressed version of the original 4x4 block of pixels. Using Heckbert's color map approach, 256 colors are selected which are most representative of the colors found in the uncompressed image ($2^{15} - 1$ colors). The two best colors found by BTC encoding are represented by pointers into a lookup table containing 256 colors [5].

images are displayed and not recorded.

scry_open, which establishes communication with the video workstation, requires the user to specify the Internet name or address of the server, the protocol used, and the remote program number. The client can also communicate with a Sun based server as mentioned above. On the Sun, several servers may run at the same time, displaying images from different clients. The RPC program number identifies which server (and therefore which window) the image is to be sent to. The PC server runs only one remote program, which has a fixed RPC program number. If the user specifies the PC's program number, a fixed port number is chosen. Having the port number built in guarantees that a call will not be made to the non-existent port mapper on the PC.

The user also specifies the protocol, that is UDP or TCP. The server must be invoked with an argument specifying whether it will accept calls using the UDP or TCP protocol. Thus the user must specify the matching protocol. This is a disadvantage -- if the server is expecting TCP and the client is sending using UDP, the client program will time out with no clue as to what has happened. If it is the opposite error, the client process will exit when a connection is attempted, saying it cannot establish the connection. This design choice was made to avoid an initial UDP negotiation via RPC's which specifies which protocol the client wants, and then closing the server socket and reopening a TCP connection if the client desires TCP.

Nine remote procedures are available on the server. RPC's are identified by name, for example *INITPROC* (actually an integer constant which the RPC-handling dispatch routine on the server uses to find the correct code to execute).

scry_open potentially makes the first RPC call, *AUTHPROC*, which at present is *#ifdef*'d out. If used, it sends a structure describing the user [4] to the server, which then checks to see if that is the same person currently controlling it, and then sends back a yes/no reply. It is used to avoid usage conflicts, i.e., if the reply is "no", the client exits. It is not desirable to generate a few frames of the phase space of a heavy-ion beam in the middle of a fluid-flow animation. At present, it is not used, because *CLOSEPROC*, which relinquishes control of the server, is not guaranteed to be called. *CLOSEPROC* is either made by *scry_close* or on the detection of some signals such as segmentation fault[3].

scry_set_record makes the recorder initialization call (the RPC call identified by the constant *INITPROC*). This causes the video recorder to seek to the frame number specified as an argument to *scry_set_record*. If the server is connected to a videodisk recorder, the frame number sent can be ignored and the frame number set to the beginning of the first block of frames sufficient for the sequence of images. The type of video recorder and the starting frame number is returned to the client. If the client finds that it is communicating with a server with a videodisk, the number of copies recorded per image set by *scry_set_copynum* is ignored, instead being set to one. The videodisk can trivially play back a sequence at any frame rate.

The rest of the RPC's are made as a result of *scry_send_frame*. They are made depending on the type of compression used. *scry_send_frame* sends a compressed image to the PC server to be displayed. (Alternatively, the image can be sent to a user workstation server to be displayed and recorded on disk). The call identified by *SHUTDOWN* is provided for use by the animation playback editor, Anima, when it is communicating with a PC connected to a videodisk recorder. *SHUTDOWN* takes the videodisk out of the recording state so that *INITPROC* can be called again and a new

[3] However, if the client is running under dbx, and the user says quit without first saying kill, *CLOSEPROC* will not be called. If *CLOSEPROC* is not called, the same user running a client from a different machine, not to mention other users, will all be bumped off when they attempt to begin a session. The server has to be interrupted and started again. This annoyance could be tolerated if the video recorders were in almost continuous use, but this is not the case at present.

sequence recorded with Anima.

NULLPROC is a constant defined by Sun. It is used for a remote procedure which takes no arguments and receives no arguments in reply. However, in the RPC protocol, the fact that a procedure returns is an indication that the server has sent a low-level reply indicating that it is processing or has completed executing its portion of the code. This determination is used by the client to test the PC server to ensure it is not otherwise busy (see section 4.2).

RECORDPROC records a frame once it has been entered into the server hardware frame buffer. It is called when no compression, or color map compression only, is used. In the other cases, where more substantial compression is used, there is a fair amount of control information to be received after the compressed image has been stored in main memory on the PC. In those cases the recording information (beginning and ending frames to record on) is bundled with the rest of the control information and received by a separate RPC.

COLORSEND sends a group of color map compressed scan lines to the server, which converts each pixel representation from color map index to RGB form, and places the result in the hardware frame buffer. The number of scan lines sent is dependent on the protocol. As mentioned in Appendix B, 2K packets seem to work best when UDP is used. With UDP, 4 scan lines are sent at a time. With TCP 50 scan lines are sent at a time (there are 400 scan lines total), since there is effectively no limit to the amount of data that can be sent with TCP.

When color map compression is used, pixels are represented by indices into a color map. In several cases the color map itself is sent to the PC by *MAPPROC*. *MAPPROC* is not used when *DISPLAY* is called with the TCP protocol, since all information for display and recording is sent with that particular RPC.

SENDUDP sends, using the UDP protocol, a group of scan lines that has been compressed using BTC, or BTC and a color map. Again, the size of a UDP packet is the limiting factor. The compressed scan lines are stored in PC memory until the *DISPLAY* call occurs.

DISPLAY triggers the decompression, display, and (optionally) recording of an image that has been compressed using BTC, and any combination of frame-to-frame differencing, color map compression, and Lempel-Ziv compression.

4.2. IMPLEMENTATION ISSUES

RPC's were first implemented on the PC using the UDP protocol on a local area network (Ethernet). Since UDP is not reliable, the Sun RPC package provides some mechanisms for reliability. In particular, the client re-transmits data if an acknowledgement to a RPC is not received within a set time limit. A problem arises in the proper setting for this time limit. If the time limit is too long excessive delays in displaying the data may result when a packet of data is lost because the client will wait to re-transmit until the time-out period expires. If the time limit is too short, the slowness of the PC server in performing an RPC may cause duplicate calls, because the client could re-transmit several times before the server has a chance to acknowledge. Based on experience over many runs, the retransmission delay was chosen as 4 seconds for a local-area network.

At first this 4 second setting caused problems when transferring the software frame buffer because one RPC was used to transfer one scan line, resulting in 400 RPC's per image. One or two scanlines per image were sometimes lost, causing a delay of 4 or 8 seconds. This problem is alleviated when compression is used, allowing dozens of scan lines to be sent in a packet via an RPC. The percentage of lost packets might be the same, but with compression there are many fewer packets being sent.

The problems associated with using UDP are eliminated when using the RPC library with the TCP protocol. No packets are lost. The price is a slower rate of transmission over the network, since additional tasks such as retransmission of data upon detection of error are performed by layers of the protocol below the socket level. In the version of the software currently being worked on, the option of using the UDP protocol has been eliminated. In practice, it was not much used, and had the aforementioned reliability problem.

Another advantage of TCP became evident when the client side of the package was ported to a Cray X/MP-14, UNICOS system. When the Cray is heavily used there is a sizeable delay after each RPC. (The 2-megaword executable gets swapped out on this small memory Cray.) Even with compression, the amount of data associated with the software frame buffer exceeds the 8K limit imposed by UDP. Since the compressed image of the torus and particles [15] is on the order of 25K bytes, several RPC's have to be made to transfer the data, with the accompanying delays. With TCP the entire compressed frame buffer is sent with one RPC. There is still a small delay evident after a certain amount of data (estimated at 10K) is sent, but the total transfer time is much smaller.

Normally the server does not reply to a RPC, allowing the client to continue, until it has finished all its processing. However, some operations are time consuming. For example, recording a frame on videotape takes about 15 seconds. In this case, to allow the client to be generating the next frame while the recording process is going on, the server sends a reply to the client's RPC before the recording is started.

A problem with sending a reply to the client before the recording process is finished is that the "record" command is initiated by a write to the PC's *com1*, and there is no blocking. If the next image is generated before recording is finished, it is possible that half of the old frame and half of the new will be recorded. (The hardware frame buffer is not cleared for each frame, merely written over.) To prevent this, a busy wait routine is executed on the PC to effect a "block" for the time needed to complete the recording process.

On a VTR, the recording takes a long time because a preroll of the videotape transport is required. This preroll is necessary on most VTR's for single-frame recording in order to stabilize the mechanical tape transport mechanism [6]. The time for this preroll can be modified by changing parameters that are sent to the animation controller. When using the wait routine, the wait time must be adjusted if the preroll time is changed.

If a videodisk is used instead of a VTR, recording only takes 0.5 seconds. However, the server still sends the reply before recording. The Panasonic TQ2026 videodisk has a repertoire of character-sequence commands to control it. The commands are sent over a serial line and the videodisk can be instructed to send a reply back over the com line indicating that it is finished recording.

Several seconds per frame can also be saved if computation on the client is overlapped with decompression and display of the image data on the server. This is done by having the server return from a client RPC before the decompression and recording are complete. When this is done, the PC server is busy performing these activities and cannot respond to an incoming RPC. Both with decompression, and with the recording RPC (where the PC is busy-waiting or waiting for the videodisk to say it's done) packets can be re-transmitted with UDP because the timeout period expires. It is useful to make the *NULLPROC* RPC (sending no data, but indicating a response) before the first RPC in each frame to make sure the PC is ready.

Even over a local-area network, compressing the frame buffer (with a typical compression ratio of 20:1 for 400K images, resulting from 3D rendering) saves around 6 seconds per frame, real time, between a PC server and a Sun 4-110. Over a wide-area network the savings are more substantial. As a feasibility study, a program, using the video client library, which renders a 3D object (a torus) has been sent to a Sun 4-110 at Duke in North Carolina, where a sequence of frames has been generated, compressed, and then sent over the wide-area network. A bandwidth of roughly 6K per second was observed. It took about 70 minutes to send over 50 of uncompressed 400K images, as opposed to only 12 minutes for 50 compressed versions (20K) of the same images.

5. CONCLUSIONS

The server portion of Sun RPC was ported to the PC. The two items that made this possible are the Microsoft C run-time library, and the Excelan Ethernet controller board and associated BSD-like socket software. There are now several such software packages, for example Sun's PC-NFS Programmer's Toolkit. Not many changes were necessary to the RPC code; the main change necessary was substituting Excelan syntax for Unix syntax socket calls.

Porting the server to the PC allows the use of RPC's to control the video workstation, accruing significant advantages. Instead of the three-step process described in section 2 of generating a metafile on the front end, transferring the metafile over the network, and then reading the metafile and generating the graphics calls on the PC, a one-step process occurs in which the data is automatically sent across the network and displayed and recorded. RPC's hide the details of the socket level and the video workstation from the user. It is relatively easy to build a high-level interface on top of RPC's which hides the details of RPC's as well.

The client can handle a variety of applications at the lowest level; it only expects a raster image in a specified form, and does not care how that image is arrived at. Scan converted 2D and 3D graphics resulting from scientific visualization, PostScript interpreted files, and images in various other formats have been converted to TARGA form, and have been sent by clients to the video workstation.

To allow time consuming processes on the PC to be overlapped with computation on the client, the display and recording RPC's on the video workstation return and send an acknowledgement back to the client before commencing the display and/or recording processes. The preroll-record-postroll on the VTR takes about 20 seconds (it was set this high to reduce the number of glitched frames). Thus this ''asynchronous'' acknowledgement provides the client more than 20 seconds real time per frame with the VTR, and several seconds per frame with the videodisk, if the process of generating an image on the client is lengthy. For images that are generated and compressed rapidly, use of the video optical disk is advantageous. For example, the generation of a 2D particle image often takes only a few seconds, after which the client has to wait if a VTR is used. With a 20-second record sequence, it still takes at least 30 minutes to record 100 images. Using the videodisk, with its roughly 0.5 second recording time, the same number of frames can be recorded in a matter of minutes.

The distributed graphics system software has been given to a number of supercomputer centers. It provides one solution to the problem of interpreting data produced at remote sites, a typical one for a user of a remote supercomputer center. Many types of scientific modelling generate huge amounts of floating-point data, which are incomprehensible (and unmanageable) unless translated into visual form [11]. Sending uncompressed raster images across a wide-area network is impractical even today, with high-bandwidth backbones. As mentioned in section 4.2, it can still be more than five times faster to send compressed, as opposed to uncompressed, images across a wide-area network. The video recording server, controlled by RPC's from the remote supercomputer, provides an economical, time-saving, and easy way to automatically display and record images generated from modelling at remote sites.

6. REFERENCES

1. Birrell, Andrew D. and Bruce Jay Nelson. Implementing Remote Procedure Calls. CSL-83-7, October 1983 [P83-00008] Xerox Parc Publication.
2. Campbell, G., T. A. DeFanti, J. Frederiksen, S. A. Joyce, L. A. Leske, J. A. Lindberg, and D. J. Sandin. Two Bit/Pixel Full Color Encoding. SIGGRAPH 1986 Proc. 20, 4 (Aug. 1986), 215-223.
3. Comer, D. Operating System Design. Volume II, Internetworking with XINU. Prentice Hall, Inc., Englewood Cliffs, NJ. 1987.
4. External Data Representation Protocol Specification. Sun 3.0 Documents, Revision G of 17 February 1986. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043.
5. Heckbert, P. Color Image Quantization for Frame Buffer Display. SIGGRAPH 1982 Proc. 16, 3 (July 1982), 297-307.
6. Johnston, W. E., D. E. Hall, F. Renema, and D. Robertson. Principles and Techniques for Low Cost Computer Generated Movies. LBL-22330. University of California. 1986.
7. Johnston, W. E., D. E. Hall, F. Renema, and D. Robertson. Low Cost Scientific Video Movie Making. Computer Physics Communications 45 (1987), 479-484. North Holland, Amsterdam.
8. Johnston, W. E., D. E. Hall, J. Huang, M. Rible and D. W. Robertson. Distributed Scientific Video Movie Making. Proceedings of the Supercomputing Conference 1988 (The Computer Society of the IEEE).
9. Leffler, Samuel J., Robert S. Febry, William N. Joy, Phil Lapsley, Steve Miller and Chris Torek. An Advanced 4.3 BSD Interprocess Communication Tutorial. 4.3 Berkeley Software Distribution, Virtual VAX-11 Version. University of California, Berkeley, CA. April, 1986.
10. Lynch, T. J. Data Compression: Techniques and Applications. Wadsworth, Inc., London. 1985.
11. McCormick, B. H., T. A. DeFanti, and M. D. Brown, eds. Visualization in Scientific Computing. Special Issue on Visualization in Scientific Computing. Computer Graphics 21, 6 (Nov. 1987).
12. Microsoft C Compiler for the MS-DOS Operating System. Run-Time Library Reference. Version 5.0. 1987. Microsoft Corporation, 16011 NE 36th Way, Box 97017, Redmond, WA.
13. Network Software for IBM Personal Computers Running DOS. Reference Manual. Revision A, May 1986. Excelan, Inc., 2180 Fortune Drive, San Jose, CA.
14. Remote Procedure Call Programming Guide. Sun 3.0 Documents, Revision G of 17 February 1986. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043.
15. Robertson, D. W. Use of a Distributed Movie Making System for Presentation of Fluid Flow Data. San Francisco State University, San Francisco, CA, (Masters Thesis - available as LBL-25274 from Lawrence Berkeley Laboratory), 1988.
16. Robertson, D. W., W. E. Johnston, T.-J. Chua, James Huang, F. Renema, M. Rible, N. Texier, and B. J. Wishinsky. Scry: A Distributed Image Handling System. LBL-27696. University of California, Lawrence Berkeley Laboratory, Berkeley, CA. 1989.
17. Sechrest, S. An Introductory 4.3 BSD Interprocess Communication Tutorial. 4.3 Berkeley Software Distribution, Virtual VAX-11 Version. University of California, Berkeley, CA. April, 1986.
18. Sethian, J. A., and A. F. Ghoniem. A Validation Study of Vortex Methods. J. Computational Physics 74, 2 (Feb. 1988), 283-317.

19. Sun RPC 1.1 Software Distribution. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043.

20. Tan, See-Mong. Personal communication.

21. TARGA Software Tools Notebook. Version 4.0. 1988. Truevision Inc., 7351 Shadeland Station, Suite 100, Indianapolis, IN.

22. Texier, N., W. E. Johnston and D. W. Robertson. Encoding Synthetic Animated Pictures. LBL-24236, University of California, Lawrence Berkeley Laboratory, Berkeley, CA. 1987.

23. Unix 4.3 Berkeley Software Distribution, Virtual VAX-11 Version. University of California, Berkeley, CA. April, 1986.

Appendix A

Availability of Code

The code for sample programs, the underlying client, the server, and the modified Sun RPC library is available as public domain software, called Scry [16]. Scry is provided as a professional academic contribution for joint exchange. Thus it is experimental, is provided "as is", with no warranties of any kind whatsoever, no support, promise of updates, or printed documentation.

The latest version of Scry is available by anonymous ftp (login: "anonymous", password: user e-mail address) from george.lbl.gov (128.3.196.93) in *pub/scry.tar.Z* (a compressed tar file, so don't forget to set binary mode in ftp). Be aware that the compressed file is about 4.7 megabytes. Once on your machine, run uncompress on *scry.tar.Z*, and extract the files using "tar xvf scry.tar scry" The resulting files total roughly 7 megabytes, which is mostly sample data.

Appendix B

Remote Procedure Calls and 4BSD Unix Sockets Tutorial

A summary of Sun's implementation is given here to facilitate describing how the port was achieved, and how the client uses RPC's to control the video workstation. Knowledge of Sun's implementation also would help in understanding how a port might be achieved with a different Ethernet controller installed on the PC.

A user interface to Sun RPC exists at three levels. At the highest level the user doesn't have to be aware that a remote procedure call is being used. The call looks like a regular library call [14]. In the intermediate layer, the programmer uses the *callrpc* call to make the RPC. At this level the programmer has to know the address of the server, the program, version, and procedure number of the desired procedure on the server, and the XDR routines necessary to encode/decode the arguments (more on XDR below). On the server side *registerrpc* registers the procedure number with the RPC package and associates a port with the program number if the port mapper is being used [14].

The lowest user-level RPC call, upon which the two previous levels are built, has three hidden layers that handle transmitted and received data. The topmost layer handles the details of the RPC protocol, e.g. finding the remote program and procedure, sending and receiving arguments and replies, and establishing whether the RPC was successful. The next lowest layer converts the procedure arguments and results into a machine-independent representation and handles buffering of input and output. The lowest layer, the socket level, performs the actual transfer of the data.

The low-level user interface is difficult to use because, while the data transfer calls themselves are invisible, some setup is necessary which requires the user to know about sockets. To underscore the difficulties with using low-level primitives for communications, Sun documentation notes that: "This [user] level should be avoided if possible" [14].

7. THE SOCKET LEVEL

Sockets provide the mechanism of mapping addresses of target systems and processes into the Unix file system I/O mechanism. *read* and *write* (for stream I/O), or *sendto* and *recvfrom* (for datagram I/O) provide the methods of data transfer between processes on the same or different machines in Unix. Operations on sockets can be classified as to which domain or name space the address of a socket is taken from, and what type of communications protocol is used [17].

Sun RPC uses the Internet mode of addressing. In this mode the address of a socket is composed of a pair of numbers: the address of the machine on the network, and a receiving process identifier, the port. The two protocols of communication which are used by Sun RPC are UDP (User Datagram Protocol) and TCP (Transmission Control Protocol). TCP is a stream, or connection oriented protocol, and data transmitted by a single RPC using TCP can be of any length up to $2^{31} - 1$ bytes (in Sun RPC) [5]. UDP is a datagram protocol. The data transferred by a single RPC is limited by the maximum packet length. This length is system dependent, but modern Unix systems, including Sun, typically allow 8K [14]. UDP transmission is not error-free, while TCP is guaranteed to be reliable, at the cost of error-checking overhead [17]. (It was noticed during the course of this study, that while up to 8K can be sent in a UDP packet, performance suffered at larger packet sizes. A formal analysis was not made, but it appears that 2K is closer to the optimum amount sent by a Sun 3 communicating with the PC Ethernet board.)

The type of protocol to be used is specified when a socket is created (using the *socket* call). On the server side, the internet address is bound to the socket with the *bind* call; on the client side binding occurs when a *sendto* or *connect* call is made (see below). These two activities are performed internally by a Sun routine at the lowest user level. Before binding occurs, the port number must either be specified or automatically assigned by the system. If one specifies the port number, it must be converted to network byte order by the routine *htons* (host to network, short). Conversion to network byte order is necessary to ensure machine-independent representation of 16-bit and 32-bit integers [17]. The routines RPC uses for transmitting and receiving messages through a socket using UDP are *sendto* and *recvfrom*, and *read* and *write* using TCP [19].

Under the UDP protocol, the destination address is included with each packet; with the TCP protocol, a explicit connection exists between the client and server sockets. This connection is set up internally in Sun RPC with the *connect* call on the client, and the *listen* and *accept* calls on the server. After a socket is created on the server, the *listen* call is made to indicate that the socket can accept one of a specified-length queue of incoming connection requests. *accept* accepts one of these requests and returns a socket with the same characteristics as the original bound socket [17,23]. When the connection is closed, this new socket is destroyed. If the *listen* and *accept* calls have been made on the server, a *connect* call by the client will establish the connection [17]. More information will be provided on the socket-level calls in Section 11.

8. THE XDR LEVEL

The transmitted data is sent and received unaltered by *sendto*, etc. Since data representation on different machines differs, data is converted to XDR (eXternal Data Representation) before sending it over the network. (This process is also called serializing or encoding.) Machine dependencies may extend beyond different byte ordering, for example, for floating point data types. Once it reaches the other machine, RPC uses XDR routines to convert from the XDR representation to that particular machine's representation (also called deserializing or decoding) [14]. Each of the data type primitives used (e.g. short, long, bool, enum) is converted to one or more groups of 4 bytes externally (XDR form) [14]. At this point all that is left is to convert these four-byte groups to network byte ordering. The very lowest level routines, which are used in all the other XDR routines, are the in-line routines *IXDR_GET_LONG* and *IXDR_PUT_LONG*, or the routines *x_getlong* and *x_putlong*. To give the flavor, the in-line routines are repeated below [19]

```
long *buf ;

#define IXDR_GET_LONG (buf)      ntohl (*buf++)
#define IXDR_PUT_LONG (buf,v)     (*buf++ = htonl (v))
```

ntohl and *htonl* are macros that convert to and from network byte order for longs [9]. If the machine byte order is the same as the network byte order, as on the Sun, the macros do nothing.

It should be noted that the encoding or decoding done inside an XDR routine is machine dependent. For example, the conversion to or from the XDR representation for floating point numbers will be different for every differing machine representation.

Sun RPC makes use of structures whose members are pointers to functions. One place such structures are used is in the implementation of XDR streams, which are the manner of buffering data for transmission/receipt through sockets after the data has been converted to an external representation. The relevant data structure is the XDR handle, reproduced with Sun's comments below: [19]

```
/*
 * The XDR handle.
 * Contains operation which is being applied to the stream,
 * an operations vector for the particular implementation (e.g. see x_mem.c),
 * and two private fields for the use of the particular implementation.
 */
typedef struct {
        enum xdr_op     x_op;                   /* operation; fast additional param */
        struct xdr_ops {
                bool_t  (*x_getlong)();         /* get a long from underlying stream */
                bool_t  (*x_putlong)();         /* put a long to " */
                bool_t  (*x_getbytes)();        /* get some bytes from " */
                bool_t  (*x_putbytes)();        /* put some bytes to " */
                u_int   (*x_getpostn)();        /* returns bytes off from beginning */
                bool_t  (*x_setpostn)();        /* lets you reposition the stream */
                long *  (*x_inline)();          /* buf quick ptr to buffered data */
                void    (*x_destroy)();         /* free privates of this xdr_stream */
        } *x_ops;
        caddr_t         x_public;       /* users' data */
        caddr_t         x_private;      /* pointer to private data */
        caddr_t         x_base;         /* private used for position info */
        int             x_handy;        /* extra private word */
} XDR;
```

x_op contains the type of operation being performed, that is, encoding, decoding, or freeing memory that was allocated by XDR (e.g. to provide space for an array being passed) [5].

Sun provides three kinds of streams. These are: (1) streams for use with standard I/O; (2) memory streams, which send data to (in serializing) or get data from (in deserializing) a buffer in memory; (3) and record streams, which allow for data up to $2^{31} - 1$ bytes in length to be sent, a special marker indicating the end of the data. Memory streams are used with the UDP protocol, where the entire buffer that has been built up is sent at once. Record streams are used with the TCP protocol, where data is transmitted and received once the fixed-size buffer has been filled, or the special marker indicating the end of data is reached [5].

When a stream is initialized, the structure *x_ops* is set to the functions used by that particular stream. For example, the routine corresponding to *x_getlong* for the memory stream is *xdrmem_getlong*. In the case of the memory stream, initialization is done by *xdrmem_create*. The routines "filling the place of" *x_getpostn*, *x_setpostn*, *x_inline*, and *x_destroy* are used to control positioning in and destruction of the memory buffer [19]. Since the memory stream uses UDP, the size of the buffer is limited, with the actual limit being machine dependent (up to 8K bytes [14]). *x_private* contains the address of the buffer. *x_base* contains the current position in the buffer and *x_handy* contains the current size of the buffer. *x_public* is provided for user data when XDR is accessed directly without the mediation of the RPC package [5].

In the case of the record stream, *x_private* contains the address of the *RECSTREAM* data structure. *RECSTREAM* contains, among other things, the addresses of the ingoing and outgoing buffers, the next input and output position, the addresses of the ends of the incoming and outgoing buffers, and the pointers to the routines performing the actual data transfer. *x_handy* and *x_base* are not used with record streams.

Sending a more complicated data structure to a stream, such as a structure or array, involves writing an XDR routine for that structure which decomposes the data structure into its component data types and uses the appropriate XDR routine for each primitive data type. That is, the XDR routine for the structure contains as many XDR routines for primitives as there are components. Identical XDR routines (on client and server) are used for decoding and encoding; thus the same routine will on decoding build the structure up out of its components [5]. The XDR handle must be passed to every XDR routine since it contains the operation to be performed, the pointers to the functions that will be used for that particular stream, and the buffer from or to which the components will be sent [19].

As an example of the use of XDR routines, the *svc_getargs* macro (which decodes the arguments for the server) is passed an argument that is to hold the decoded data, a pointer to the XDR routine that decodes it, and the service transport handle (more on this below) [14]. To get an integer argument, the call would be *svc_getargs (transp,xdr_int,&int_arg).* (In the Sun RPC package the name of every XDR routine begins with *xdr_*). Another place where pointers to functions are used is in the server routines. For example, *svc_getargs* expands to:

```
(*(xprt)->xp_ops->xp_getargs) ((xprt), (xargs), (argsp))  [19]
```

The relevant data structure is the *SVCXPRT* handle, reproduced below from [19] with Sun's comments.

```
/*
 * Server side transport handle
 */
typedef struct {
        int                 xp_sock;
        u_short             xp_port;                /* associated port number */
        struct xp_ops {
          bool_t            (*xp_recv)();           /* receive incomming requests */
          enum xprt_stat (*xp_stat)();              /* get transport status */
          bool_t            (*xp_getargs)();        /* get arguments */
          bool_t            (*xp_reply)();          /* send reply */
          bool_t            (*xp_freeargs)();       /* free mem allocated for args */
          void                      (*xp_destroy)(); /* destroy this struct */
    } *xp_ops;
        int                         xp_addrlen;     /* length of remote address */
        struct sockaddr_in          xp_raddr;       /* remote address */
        struct opaque_auth          xp_verf;        /* raw response verifier */
        caddr_t                     xp_p1;          /* private */
        caddr_t                     xp_p2;          /* private */
} SVCXPRT;
```

Different routines will be called depending on the transport protocol, that is, TCP or UDP. For UDP the routine *svcudp_getargs* would be called for the macro above. The function pointers in the *xp_ops* structure are initialized when the service transport handle is created with *svcudp_create*, which also creates the appropriate stream. *xp_p2* is the address of the structure *svcudp_data* in the case of UDP. It contains among other things the address of the XDR handle and the size of the memory buffer. For the memory stream the address of the buffer is kept in *xp_p1* (as mentioned above, it is also kept in the *x_private* field in the XDR structure) *svcudp_create* also creates the socket through which data is transferred. The number identifying the socket is kept in *xp_sock*. In the case of TCP, *xp_p2* is unused. *xp_p1* contains the address of a structure *tcp_rendezvous* during the period of time after a socket has been created but before a TCP connection has been established. *tcp_rendezvous* contains the size of the sending and receiving buffers. After a connection has been established, *xp_p1* contains the address of a structure *tcp_conn*, which has among other fields the XDR handle [19]. These two states are explained in more detail in Section 11.

The analogous data structure for the client side is the client handle *CLIENT*. It is reproduced below with Sun's comments [19].

```
/*
 * Client rpc handle.
 * Created by individual implementations, see e.g. rpc_udp.c.
 * Client is responsible for initializing auth, see e.g. auth_none.c.
 */
typedef struct {
        AUTH    *cl_auth;                                   /* authenticator */
        struct clnt_ops {
                enum clnt_stat    (*cl_call)();             /* call remote procedure */
                void              (*cl_abort)();            /* abort a call */
                void              (*cl_geterr)();           /* get specific error code */
                bool_t            (*cl_freeres)();  /* frees results */
                void              (*cl_destroy)();  /* destroy this structure */
        } *cl_ops;
        caddr_t                   cl_private;       /* private stuff */
} CLIENT;
```

The default authentication, i.e. none, was chosen. If security is important it is really up to the user to provide it, because with the authentication RPC provides, "It is easy to impersonate a user" [14].

As above, the pointers to the functions will be set when *cl_ops* is initialized in the client handle creation routine. In the case of UDP, this is *clntudp_create*. As above, in this routine the appropriate stream is created, as well as the socket to transmit data and receive a reply. *cl_private* is the address of the structure *cu_data* in the case of UDP, and *ct_data* in the case of TCP. Both contain among other things the socket number, the address of the destination, and the address of the XDR handle. [19].

9. THE RPC PROTOCOL LEVEL

Sun RPC requires that the client and server communicate additional information besides the actual arguments of the remote procedure call and the reply sent back. This information is kept in the RPC message data structure, which is reproduced below with an expansion of its constituent data structures, along with Sun's comments [19]. The data in the fields of this structure is inserted before the arguments or the reply in the particular stream used (i.e. memory or record).

```
/*
 * The rpc message
 */
struct rpc_msg {
        u_long                  rm_xid;
        enum msg_type           rm_direction;
        union {
                struct call_body RM_cmb;
                struct reply_body RM_rmb;
        } ru;
};
```

The rpc message contains the transaction id *rm_xid*, to make sure the reply is received from the process that the arguments were sent to and to make sure that the appropriate data structures haven't been corrupted. *rm_direction* gives the direction, i.e. call or reply. Depending on the direction, the union *ru* contains the structure *call_body* or *reply_body* [19].

```
/*
 * Body of an rpc request call.
 */
struct call_body {
        u_long cb_rpcvers;          /* must be equal to two */
        u_long cb_prog;
        u_long cb_vers;
        u_long cb_proc;
        struct opaque_auth cb_cred;
        struct opaque_auth cb_verf;        /* protocol specific - */
                                           /* provided by client  */
};
```

call_body contains the information necessary to identify the desired procedure on the server, along with some authentication fields, which are unused in this implementation. *cb_prog* contains the program number, *cb_vers* the version of the program, and *cb_proc* the desired procedure number [14,19].

```
/*
 * Body of a reply to an rpc request.
 */
struct reply_body {
        enum reply_stat rp_stat;
        union {
                struct accepted_reply RP_ar;
                struct rejected_reply RP_dr;
        } ru;
};
```

reply_body contains information about what happened as a result of the call, i.e. *rp_stat* tells whether there the call was accepted or denied. Depending on *rp_stat*, the union *ru* contains the structure *accepted_reply* or *rejected_reply* [19].

```
/*
 * Reply to an rpc request that was accepted by the server.
 * Note: there could be an error even though the request was
 * accepted.
 */
struct accepted_reply {
        struct opaque_auth          ar_verf;
        enum accept_stat ar_stat;
        union {
                struct {
                        u_long  low;
                        u_long  high;
                } AR_versions;
                struct {
                        caddr_t          where;
                        xdrproc_t        proc;
                } AR_results;
                /* and many other null cases */
        } ru;
};
```

```
/*
 * Reply to an rpc request that was rejected by the server.
 */
struct rejected_reply {
        enum reject_stat rj_stat;
        union {
                struct {
                        u_long low;
                        u_long high;
                } RJ_versions;
                enum auth_stat RJ_why;  /* why authentication did not work */
        } ru;
};
```

accepted_reply contains the authentication parameter *ar_verf*, and *ar_stat*, which tells whether there was success or some error, such as the program being unavailable on the server. The *AR_versions* structure, and analogously, the *RJ_versions* structure in *rejected_reply*, contain information on whether there was a program version mismatch. *rj_stat* in *rejected_reply* tells why the message was rejected. *where* in *accepted_reply* contains the address of where to place the results and *proc* the address of the XDR routine used for the results. These are filled in on the the client side and are used when, during the decoding of the reply, the pointer to the stream buffer advances to where the server had placed the identical fields of the *reply_accepted* structure. The appropriate XDR call is then made to decode the results of the RPC [19].

The server uses information in *call_body* (the program and version number) to find the desired procedure. This is done through the use of the service callout list. The relevant data structure is reproduced below, along with Sun's comments [19].

```
/*
 * The services list
 * Each entry represents a set of procedures (an rpc program).
 * The dispatch routine takes request structs and runs the
 * appropriate procedure.
 */
static struct svc_callout {
        struct svc_callout *sc_next;
        u_long                  sc_prog;
        u_long                  sc_vers;
        void                    (*sc_dispatch)();
} *svc_head;
```

sc_next is a pointer to the next entry in the callout list. *sc_prog* is the program number and *sc_vers* is the program version number. *sc_dispatch* is a pointer to the dispatch routine associated with the program and version numbers. The dispatch routine is the remote "program": cases within it correspond to the remote "procedures".

At the lowest user-level of Sun RPC, the service must be registered by the server with the routine *svc_register (transport handle, program #, version #, dispatch address, protocol)* to place it on the service callout list. After that the server goes into an infinite loop. The routine called inside this loop uses select to accept remote procedure calls (thus avoiding busy-waiting) and uses the RPC message to search through the callout list. If found, the dispatch routine is called, which calls the appropriate procedure and sends back the results to the caller with *svc_sendreply* [14].

10. POTENTIAL DIFFICULTIES

There are a few areas where RPC's as implemented by Sun might present some difficulties. At the intermediate and lowest levels, only one call argument and one reply argument can be sent at a time. If there are many arguments, they must be bundled up into a structure or otherwise placed in contiguous memory. This doesn't present a problem at the highest level since the bundling can take place at the intermediate level. A more serious problem is that of the XDR routines for int's and long's. If there is a standard service that takes int's as arguments and if the client does not know the identity of the server, as in the case where broadcasting of a RPC is used, it does not know whether int's correspond to long's or short's on the destination machine. This is a problem because in the XDR routine for int's, *xdr_int*, decodes an int depending on sizeof(int). If int corresponds to short, *xdr_short* is called, and if not, *xdr_long* is called. An analogous problem occurs when it is known that int's are long's on the client and short's on the server. In that case, one could send and receive all int's as long's.

11. PORT OF RPC TO THE PC

The portion of the Sun RPC package that a server is built upon was ported to an IBM PC-AT compatible, using the Excelan socket library[4], and the Microsoft C compiler and runtime library. (Hereafter ''server'' will refer either to this portion of the Sun RPC library, or the program that uses it.)

The first change made was to change file names so that they contained no more than 8 characters (DOS specifications). The next change made was to replace slash by backslash in all the include statements that used it. All the standard header files are part of the Microsoft run-time library. *sys/param.h* was replaced by *stdio.h* in *rpc_prot.c* since all that was used from *param.h* was *NULL*. The standard typedef's for unsigned types (e.g. *u_int*, *u_char*) were added to a new header file.

bcopy and *bzero*, which are used in a number of places in Sun RPC, are changed to the analogous Microsoft library routines *memcpy* and *memset*. The form of the calls is as below:

Unix (ref [23])	Microsoft (ref [12])
1. bcopy (src,dest,length) char *src, *dest ; int length ;	memcpy (dest,src,length) char *dest,*src ; unsigned length ;
2. bzero (buf,length) char *buf ; int length ;	memset (buf,0,cnt) char *buf ; unsigned cnt ;

All socket calls are changed from BSD Unix to the corresponding EXCELAN specifications. The mapping is given in the table below:

[4] Enhancement: It is possible to convert to the PC NFS toolkit.

	Unix (ref [23])	EXCELAN (ref [13])
1.	sock = socket (domain,type, protocol) int domain, type, protocol ;	sock = socket (type,protocol, server,options) int sock, type, options ; struct sockaddr *server ; struct sockproto *protocol ;
2.	bind (sock,&server,sizeof(server)) struct sockaddr_in server ; int sock ;	corresponding functionality is merged into the socket call
3.	hp = gethostbyname (aname) struct hostent *hp ; char *aname ;	addr = rhost (aname) unsigned long addr ; char **aname ;
4.	cc = recvfrom (sock,buf,buflen, flags,from,fromlen) int cc, sock, buflen, flags ; int *fromlen ; char *buf ; struct sockaddr *from ;	cc = soreceive (sock,from,buf,buflen) int cc, sock, buflen ; struct sockaddr *from ; char *buf ;
5.	cc = sendto (sock,buf,buflen, flags,to,tolen) int cc, sock, buflen, flags ; int tolen ; char *buf ; struct sockaddr *to ;	cc = sosend (sock,to,buf,buflen) int cc, sock, buflen ; struct sockaddr *to ; char *buf ;
6.	connect (sock, addr, addrlen) int sock ; struct sockaddr *addr ; int addrlen ;	connect (sock, addr) int sock ; struct sockaddr *addr ;
7.	listen (sock, backlog) int sock, backlog ;	corresponding functionality is merged into the socket call

```
8. ns = accept (s, from, addrlen)        accept (s, from)

   int ns, s ;                            int s ;
   struct sockaddr *from ;                struct sockaddr *from ;
   int *addrlen ;


9. n = select (width, rfds, wfds,         n = soselect (width, rfds, wfds,
              excfds, timeout)                  timeout)

   int n, width ;                         int n, width ;
   fd_set *rfds, wfds, excfds ;           long *rfds, *wfds, timeout ;
   struct timeval *timeout ;


10. cc = read (sock,buf,buflen)           cc = soread (sock,buf,buflen)

   int cc, sock, buflen ;                 int cc, sock, buflen ;
   char *buf ;                            char *buf ;


11. cc = write (sock,buf,buflen)          cc = sowrite (sock,buf,buflen)

   int cc, sock, buflen ;                 int cc, sock, buflen ;
   char *buf ;                            char *buf ;


12. close (sock)                          soclose (sock)

   int sock ;                             int sock ;
```

On the Unix side, the *socket* call creates a socket. Domain is *AF_INET* for the internet domain, and type is *SOCK_DGRAM* for the UDP protocol, and *SOCK_STREAM* for the TCP protocol. *protocol* should be assigned the value 0. *bind* binds an internet address to the socket [17,23]. On the Excelan side the *socket* call combines the Unix *socket* and *bind* calls. As before, type is *SOCK_DGRAM* for UDP and *SOCK_STREAM* for TCP. *protocol* should be assigned the pointer *(sockproto *) 0*. *server* contains the internet address of the server. *options* is obtained by or'ing together the various options desired. An example of an option is *SO_DEBUG*, which enables debugging information [13]. *gethostbyname* on the Unix side and *rhost* on the Excelan side are used to find the internet address of the server if only the symbolic host name is known [13,17].

recvfrom on the Unix side receives a datagram through a socket, while *sosend* on the Excelan side sends a datagram through a socket. This pair of calls works together, that is, data is transmitted and received without error due to mismatch between Excelan and Unix IPC implementations. *sendto* on the Unix side sends a datagram while *soreceive* receives one. This pair also works together. The *from* argument in *soreceive* is filled in with the address of the sender, which is used by RPC to send back the reply. *read* can receive data sent through a connected socket, while *sowrite* sends data through a connected socket. *write* can write data through a connected socket, while *soread* performs the corresponding read on the Excelan side. *read* and *write* are also used with file descriptors [17].

close on the Unix side (among other things) and *soclose* on the Excelan side close a socket [13,23]. Since the only way the server will stop running on the PC is through an interrupt, the break handler must close any open sockets. Otherwise the PC hangs [13].

The *connect* call made on the client side establishes a TCP connection. Since only the server was brought up on the PC, the Excelan *connect* has not been tested. As mentioned in Section 7, *listen* and *accept* prepare the server to accept a connection. In Unix, *listen* establishes a queue on which incoming connections wait until they can be accepted. *accept* takes one request, and accepts a connection on a duplicate of the original socket (*ns* in the table above). When a connected TCP socket on the client side is closed, the connection is destroyed, and the corresponding server socket is also closed [17].

When using Excelan software, specifying *ACCEPTCONN* as an option on the *socket* call readies it for an accept, as does the explicit Unix *listen*, with the difference that no queue for incoming connections is established. The Excelan *accept* accepts a connection, but does not return a new socket, since there can be only one connection accepted per open socket [13].

The Unix *select* uses a mask of *width* bits to determine which sockets are ready for reading (*readfds*) and writing (*writefds*) [13]. According to the Excelan manual (Revision A, May 1986), *soselect*, performing much the same function, only works under TCP. In practice it does not work under UDP or TCP. (This may very well have changed under later revisions.) Another port of the server portion of the RPC library using UDP substituted the Excelan *soioctl* call (with the *FIONREAD* option) to determine if there were bytes waiting in a socket's receive buffer [20].

soselect was the only Excelan call that was an obstacle to porting the portion of Sun RPC using UDP. However, the video server is only suitable for use by a single client at a time. Two or more clients attempting to access the server at once would result in images composed of groups of scan lines from different sources, and disjointed movies having, say, one frame of a mathematical surface and the next of a fluid-flow simulation. (An attempt at enforcing single-client usage is described in Section 4.1.) Thus the fix described in the previous paragraph is not necessary. The call that receives incoming data, *soreceive*, blocks until a message arriving at the single open socket is available [13].

The port of the portion of the Sun RPC server using TCP was more difficult. In Sun RPC, when a socket is created, a portion of the server handle *SVCXPRT*, *tcp_rendezvous* (see Section 8), is kept in a data structure located in an array (*xports*) at a position corresponding to the socket number. In particular, this occurs for the original socket created when the server starts running. A connection request for that original socket is treated as incoming data by *select*, and the data structure for that socket is used to call a routine that handles the acceptance of a new connection. At this point a duplicate socket is created, and the information associated with this new connection socket is placed in *xports*. When *select* shows that this new socket is ready for reading, the *xports* data structure for this new socket is used to call a routine that reads data from incoming RPC calls. After the connection is closed, and the duplicate socket destroyed, this process repeats when a new connection request is made for the original socket.

In this case, the *soioctl* call described above would have to be made to substitute for *select*'s functionality, since more than one socket is open (assuming *soioctl* would treat an incoming connection request as incoming data -- this has not been tested). Another alternative might be to use the *soioctl* call to make a socket non-blocking. However, the nonfunctional *soselect* is not the only difficulty with the Excelan routines. As mentioned above, *accept* does not create a duplicate socket. The connection must be made with the original socket, which is then closed when the connection is destroyed.

To solve this problem, *soselect* is bypassed as in UDP. The routine accepting incoming data, in this case *soread*, blocks until data is received. When the connection is closed, the next *soread* returns an error, which is used to notify the server that it must start again from the beginning. This involves the overhead of making the *socket* call again (with its implicit *bind*), and filling in some data structures. In the case of the video server, connections are fairly lengthy, especially to record a movie, so this overhead is not a problem.

The remaining major change is that the port mapper routines were bypassed[5]. The port mapper is like a server ''server'' in that is is an independent process that servers consult when they want to associate a program number with a port number. The port number is chosen by the port mapper [14]. To allow the use of the port mapper on the PC (which uses the uni-tasking operating system MS-DOS), the port mapper would have to run concurrently with the server. Thus it was bypassed. This was possible without a major rewrite of the code by ensuring that the port mapper is never called. The only routine on the server side that would cause a call to the port mapper (besides the unused port mapper routines) is *svc_register*. The *svc_register call* is made with the last argument set to zero, which causes the port mapper to be bypassed [14]. Since the port number cannot be ascertained by the client from the non-existent port mapper on the PC, it cannot be arbitrary. The port number of the socket on the PC is fixed (given by *PCPORT*). The client and the server must define *PCPORT* consistently. This is hardwiring, but as justification it can be said that the client already had to know the program, version, and procedure number for a RPC call to work.

The server and the XDR routines tested (and thus the routines that they call) seem to work with no modifications beyond those described above. As mentioned above, the way XDR encodes from or decodes to a particular machine data representation is machine dependent. For example, the XDR routine for doubles (which we do not use) would need to be modified for the PC.

[5] Enhancement: It is possible that a third-party port mapper running on a machine other than a PC could be used to assign PC port numbers.

END

DATE
FILMED
5/14/92

[illegible]