

October 1992

Parcels with TAGS

J.L. Sloan
jsloan@ncar.ucar.edu

SCIENTIFIC COMPUTING DIVISION

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
BOULDER, COLORADO

Abstract

The NCAR Text And Graphics System is a graphics and text batch output production facility. It is distributed across several networked computers which run the UNIX operating system. The control function of TAGS is implemented as multiple processes which communicate via message passing. The TAGS message passing scheme allows processes to exchange a simple form of associative memory, called a parcel, in an architecture neutral fashion. The associative memory is implemented as a binary tree containing pairs of keyword and value strings ordered by collating sequence. The underlying interprocess communication mechanism is based on Berkeley internet stream sockets. This message passing scheme provides information hiding inside the message, and allows new parameters to be added to messages without recompiling or relinking intermediate modules which handle the messages. The software as described is available from NCAR via anonymous FTP at no cost under a non-commercial license.

1. Introduction

This document describes the interprocess communication (IPC) mechanism developed as part of the NCAR Text And Graphics Systems (TAGS).

TAGS consists of approximately 100,000 lines of C code, divided between programs for graphical rendering and for system control. The system control portion is implemented as multiple processes which are distributed across several SUN-4 machines (some of which are multiprocessor systems) running the SunOS flavor of UNIX. The processes communicate by passing messages containing a simple form of associative memory. This allows processes to easily exchange job and process state information.

The TAGS IPC mechanism was built in several layers. First, we will describe a family of data structures and their associated functions for storing and manipulating simple ASCII-based associative memories. These structures and functions have proven useful beyond TAGS or even the IPC package for which they were implemented. Next, we will describe the two layers of the interprocess communication mechanism. The layers are not specific to TAGS, and have been used in other projects. Finally, we will discuss some problems imposed by the Berkeley UNIX implementation of stream sockets and the Internet Protocol that were discovered during the implementation of TAGS, and propose a solution.

Practitioners who want a quick look at TAGS IPC in use can skip to the examples near the end of this paper.

1.1 Overview of TAGS

Software is frequently better understood if it is discussed in the context of an actual application. We will spend a few moments here to describe the Text And Graphics System. Examples drawn from TAGS will be used throughout this paper.

TAGS is a batch output production facility which services several supercomputers located at the National Center for Atmospheric Research [Ruff 1991]. TAGS typically produces 500,000 frames of graphics and text output per month. Output media includes 35mm microfilm, 35mm color slides, and 105mm microfiche, generated using Dicommed film cameras, VHS, S-VHS, Umatic, and Umatic-SP video tape, and monochrome and color paper.

Jobs are submitted to TAGS over the NCAR network. Job submissions may include the input data, or request an input file from the forty-three terabyte NCAR Mass Storage System. Input data may consist of either computer graphic metafiles, several formats of raster files, or PostScript. The first release of the TAGS software went into production in May of 1990; the second and most recent release went into production in December of 1990.

2. Data Types

Users specify many parameters when submitting jobs to TAGS. These parameters are semantically pairs of keyword and value strings, and are often syntactically represented as *keyword=value*. User parameters include the choice of output media, how images are scaled and proportioned to fit in the graphic viewport, whether vector or raster output is optimal, as well as more mundane information such as account number and the location to which the hardcopy output is sent. Some example TAGS user parameters are shown below.

```
NAME=JDOE
SCI=9999
PROJ=12345678
TITL="Example Plot"
REQ=PLOT
MACR=VIEWERCL
FILM=E635
QUAL=BEST
RES=SCREEN
VIEW=8117:5362:16534:22043
WIND=0:0:32768:32768
MSFN=/jdoe/model/ExampleData
```

During the life of each TAGS job there is a considerable amount of internal information describing the state of the job which must be maintained by the system. These system parameters can also be described in the form of keyword=value pairs. Some example TAGS system parameters are shown below.

```
came=COLOR1
jid=98765
data=/s1/D98765
copy=0
state=ACTIVE
spri=18
timestamp="92/02/29 00:01:25"
size=75816
qpri=9
frame=144
```

The storage, addition, deletion, alteration and examination of the user and system parameters in a job's context under program control must be reasonably time and space efficient. Furthermore, this dynamic information must be capable of being passed in messages.

New parameters may be defined to the system as the need arises. These new parameters may only be applicable, for example, to processes dealing with a new output device or a new input data format. This change should only require recompilation of the modules which deal directly with the new parameters. Modules which simply pass along messages containing these new parameters should not require recompilation or relinking. Furthermore, parameters passed in messages should be hidden from intermediate processes which have no need to reference them.

This criteria led to a family of data structures for storing and manipulating collections of keyword=value pairs: *parcels*, *packages*, *piles*, *parmfiles*, and *packets*. Together with the functions to manipulate them, they form a related group of data types that serve as a data storage backbone for the control portion of TAGS.

2.1 Parcels

A parcel is a data structure which provides a simple associative memory consisting of ASCII keyword=value pairs. A parameter value string can be extracted from the structure by providing its corresponding keyword string.

The TAGS control processes use parcels extensively. Each TAGS job has associated with it a parcel which contains all of the user parameters submitted with the job, system parameters generated as the job is accepted, and parameters describing the job's current state as it moves through the system. TAGS Spooler processes maintain their start-up configuration parameters and their internal state in parcels.

The parcel structure is implemented as a binary tree where each node contains pointers to the left and right branches, and pointers to the two character arrays containing the keyword and value strings of the parameter stored in that node. Nodes are ordered in the tree by ASCII collating sequence of the keyword.

Whenever another structure is built from a parcel, the tree is traversed depth-first in pre-order, visiting the root first followed by the right and left branches. This avoids the degenerate case by ordering parameters so that if the resulting structure is in turn used to build another parcel, the nodes are inserted in non-collating sequence order.

Functions are provided to initialize parcels, to insert, delete and extract parameters by keyword, to merge two parcels, and to apply a function to all nodes in a parcel. When inserting a keyword=value pair into a parcel, the node structure is dynamically allocated, and the two strings are copied to dynamically allocated character arrays; when a node is deleted, the node and the storage used by the keyword and value strings are deallocated. This allows string constants or the contents of automatic variables to be used as arguments to the insert functions. (By now the reader is concerned with storage management; this is discussed in some detail later.)

Since parcels are referenced by a single fundamental data type (a pointer), they can easily be passed as function parameters.

A binary tree provides a mechanism for storing and accessing parameters in a reasonably time and space efficient manner. It implements a simple associative memory where access to a particular parameter is limited to code which "knows" the corresponding keyword, and in which parameters can be dynamically added or deleted without changing the definition of the structure. A linear list of string pointers with a hashing function was also considered. The variable size of the linear list, versus the fixed size of the tree node structure, made storage management more complex. It was also felt that a hashing function might place unnecessary restrictions on the name space of the keyword. Even though a hashing function might be faster, a binary tree was deemed "good enough". In any event, the parcel data type is sufficiently abstract that a new underlying implementation could be put in place without impact to application code.

The code fragment below illustrates the ease with which parcels and their parameters can be manipulated.

```

PARCEL first, second;
char *value;

initparcel(&first);           /* Initialize two parcels */
initparcel(&second);
bundle("Keyword", "Value1", &first); /* Insert parameter */
rebundle(first, &second);      /* Merge parcels */
value=unbundle("Keyword", second); /* Extract parameter */
bundle("Keyword", "Value2", &second); /* Alter parameter value */
bundle("Keyword", (char *)0, &second); /* Delete parameter */

```

2.2 Packages

A package is a character string containing zero or more keyword=value pairs. A keyword=value pair are separated by an equal sign, and successive pairs are delimited by *csh*-style white space which is unquoted or unescaped. A package is terminated by the ASCII nul character (`\0`), following the usual C conventions.

For example, the string

```
nickname="John 'Baby' Doe" SCI=9999 NAME=JDOE sep=\t
```

forms a package containing four keyword=value pairs. The words which constitute a keyword or a value conform to a subset of C-shell syntax. Words may contain white space or encoded non-printable characters. Functions are provided to extract keyword=value pairs from a package and insert them into a parcel, and to translate the contents of a parcel into a package.

TAGS uses packages for displaying parcels in debugging output, for translating parameters in command line or interactive input into parcels, and for nesting parcels by embedding the contents of a parcel in the form of a package as a value of a parameter inside another parcel. This last use is of particular value inside start-up configuration files and, as will be seen, in message passing.

The code fragment below illustrates the use of packages.

```

PACKAGE *string;

string=wrap(first);           /* Extract package from parcel */
printf("%s\n", string);
unwrap(string, &second);      /* Merge package into parcel */
bundle("First", string, &second); /* Insert package as parameter */

```

To implement a consistent look and feel, a set of parsing tools were developed and used consistently throughout the TAGS system. Uses include the parsing of packages. *Nextparser* is an efficient, table-driven push-down automaton which parses *csh*-style syntax, including nested single and double quotes, escaped embedded quotes, escaped newlines for line continuation, and comments. It has two entry points: *nextfrombuffer*, which returns the next "word" from a character array, and *nextfromfile*, which returns the next "word" from a file. In each case, a "word" is a character string token in the *csh* sense, including quoted or escaped white space, etc.

The code fragment below illustrates how the parser can be used to display all words in a buffer or a file.

```
char *buffer, *here, word[128];
FILE *file;
int rc;

/* Scan for all words in a buffer */
here=buffer;
while ( (here=nextfrombuffer(here,word)) !=NULL )
    printf("%s ",word);
putchar('\n');

/* Scan for all words in a file */
while ( (rc=nextfromfile(file,word)) >= 0)
    if (rc>0)
        printf("%s ",word);
    else
        putchar('\n');
```

The *esclapse* and *escpand* functions collapse and expand C-style escape sequences embedded inside character strings. This allows escape sequences such as $\backslash t$ (tab), $\backslash n$ (new-line), or arbitrary octal characters such as $\backslash 014$ (form feed) to be easily embedded in symbolic form inside parameter values, or special characters to be recoded as printable characters for display.

2.3 Piles

Even though it is simple to traverse all nodes in a parcel, occasionally it is convenient to step through all parameters in a linear fashion. A pile is similar to a C argument vector, except that it contains consecutive *pairs* of character pointers instead of single pointers. Functions are provided which derive a pile from a parcel, and which build a parcel from a pile.

When a pile is generated from a parcel, each pair of pointers in the pile point back to keyword and value strings in the parcel. The pile does not contain character strings of its own. Hence, changes in the original parcel may yield unexpected results in the pile. When a parcel is built from a pile, the insert function is used so that the keyword and value strings in the parcel have their own storage allocated.

TAGS frequently uses piles as an intermediate form to generate argument vectors or environment vectors when child processes are created.

The code fragment below illustrates the use of piles.

```
PILE list;
char *p;

list=pile(first);          /* Generate a pile from parcel */
for (p=list; *p!=NULL; p+=2)
    printf("%s=%s\n",*p,*(p+1));
unpile(list,&second);      /* Merge contents of pile into parcel */
```

2.4 Parmfiles

Parcels can be translated into parmfiles and vice versa. Parmfiles are editable character files which contain a single keyword=value pair on each line.

TAGS control processes maintain state internally using parcels, and periodically dump these parcels to disk as parmfiles. The parameters for each TAGS job is kept in memory in a parcel, and this information is mirrored in a parmfile for each job. This aids in debugging after a software failure, and allows TAGS to automatically recover its prior state after a system crash. Parmfiles also serve as start-up configuration files. Finally, since parcels are exported to parmfiles using a file descriptor specified by the caller, exporting parcels to the standard error file descriptor is a convenient way to display the contents of a parcel for debugging.

The code fragment below illustrates the use of parmfiles.

```
int fd;                                /* Open file descriptor */

exportparcel(fd,first);                 /* Dump parcel to parmfile */
importparcel(fd,&second);               /* Merge parmfile into parcel */
```

2.5 Packets

The first release of the TAGS software used packages as the mechanism to transmit parcels through message passing. This worked okay, but the overhead in reparsing the package on the receiving end was deemed unnecessary. A packet is a simplified data structure which in TAGS is used only in the message passing code. A packet is created by traversing the parcel and copying the keyword and value strings, including the nul terminators, into consecutive bytes in a contiguous memory buffer. This simplifies rebuilding the parcel on the receiving end since only the nul terminators need to be found. In addition, the total length of the packet in bytes is kept, in network byte order, in the first four bytes of the packet. This simplifies the management of message buffers in the IPC code.

The code fragment below illustrates the use of packets.

```
PACKET *pgram;
int length;

pgram=pack(first);                     /* Generate packet from parcel */
length=packetlength(pgram);            /* Get packet length */
unpack(pgram,&second);                  /* Merge packet into parcel */
```

3. Interprocess Communication

Rendering graphics can require considerable CPU horsepower. Real-time control of output devices can require a nearly dedicated CPU. The ability to economically expand processor power by incrementally adding more supermicrocomputer-class machines is attractive. Hence, TAGS is implemented as a collection of cooperating processes which can be distributed across multiple processors. Message passing was chosen as the model for interprocess communication.

TAGS currently runs on several SUN-4 machines. However, we did not want to limit TAGS to using a single vendor, hardware architecture, or UNIX platform. Also, it seemed to be a good idea for the TAGS tools and message passing software be reusable in other projects. Hence, the message passing mechanism needed to be architecture neutral, and be built on a widely available underlying IPC mechanism.

The TAGS interprocess communication code was implemented in two layers: the *IPC* layer and the *messages* layer. These correspond roughly to the OSI session and presentation layers respectively.

3.1 IPC Layer

The IPC layer provides a simplified interface to TCP/IP using Berkeley internet stream sockets. Server or client sockets are established using a single function call. Remote processes are identified by host and port strings. The host string can be the host or domain name, or the IP host number in the familiar dot notation. The port string can be the service name or the port number.

Functions are provided in the IPC layer to establish connections, to send and receive streams of bytes, to receive with a timeout interval specified by the application, and to poll sockets with or without timeout. (Although not used in TAGS, IPC functions to make use of internet datagram sockets were also implemented.) A mechanism to automatically manage large collections of sockets using the *select(2)* system call was implemented. It prevents sockets from being starved by scheduling I/O service round-robin.

Even without the message passing layer described below, the IPC layer offers a much simpler interface to the world of Berkeley UNIX interprocess communication than using the native function and kernel calls. The code fragment below illustrates how to establish a server socket, wait for a socket to become ready, accept connection requests, and receive data.

```
mainsock=ipcserver(ipcaddress("ServerHost"));
readysock=ipcready();
if (readysock==mainsock) {
    iosock=ipcaccept(readysock);
    (void)ipcrecv(iosock,buffer,length);
}
```

The code fragment below shows how to connect to a server process and transmit data.

```
iosock=ipcclient(ipcaddress("ServerHost"),ipcport("ServerPort"));
(void)ipcsend(iosock,buffer,length);
```

3.2 Messages Layer

The messages layer imposes a message passing scheme on an IPC stream connection that allows processes to exchange parcels in an architecture neutral fashion. It uses the parcel and packet abstract data types and the IPC layer to implement a new abstract data type called a message. Messages are dynamically allocated and freed as needed. Simple functions are provided to establish a server socket to listen for connection requests, to create client sockets connected to server sockets, to send and receive messages, and to insert, delete and

extract parameters from the parcel inside a message. When a message is sent to another process, the embedded parcel is automatically converted into a packet, transmitted by the IPC layer, and reconstructed back into a parcel at the receiving end.

It was recognized early on that passing simple requests and responses between processes by parcel was frequently unnecessary. Hence, each message has associated with it not only a parcel, but a fixed size header containing four four-byte fields (long words on most machines) and an eight byte character field. These fields are labeled function, status, id, sequence and name respectively. Although their names are suggestive, neither the message passing nor IPC code makes any reference to them other than to translate the four binary fields into network byte order for transmission, and back into host byte order on the receiving end. Hence, the application is free to make use of the fields in the fixed header in any way, including ignoring them altogether.

A next message pointer field is included in the message data structure. This field is not transmitted when a message is sent, but provides a convenient place to hang a link so that simple FIFO queues of message structures can be easily maintained.

The message structure is shown below.

```
struct message {
    struct message *    msg_next;
    PARCEL              msg_parcel;
    long                msg_function;
    long                msg_status;
    long                msg_id;
    long                msg_sequence;
    char                msg_name[8];
};
```

```
typedef struct message MESSAGE;
```

A hidden field containing the *userid* of the sending process is passed (in network byte order) along with each message. This field is not directly accessible by the sending or receiving application, but can be read by the receiver by calling the appropriate function. This feature is used, along with the IP address of the sender as provided by the Berkeley code, in TAGS to implement a simple security mechanism using access control lists. This can be spoofed rather easily, and in no way takes the place of more robust network security schemes.

This layer implements a message passing mechanism that is *sometimes synchronous*. The sending process may block on its send until the receiving process issues its corresponding receive, depending on how much of the message can be buffered inside the kernel. If the entire message cannot fit inside the kernel buffers (as is sometimes the case in TAGS when large parcels are being transmitted), the sender and receiver must rendezvous, and the sender waits until the kernel can consume the remainder of the message. (This behavior and its implications is discussed in some detail later).

Using the messages layer, creating a server port from which to accept connections, waiting for a socket to become ready, and accepting a connection request, is a matter of

```

mainsock=mserver("ServiceName");
readysock=mready();
if (readysock==mainsock)
    iosock=maccept(readysock);

```

while establishing a connection to a server port from a client requires

```

iosock=mclient("ServiceHost","ServiceName");

```

Inserting parameters into a message and sending it looks like

```

minsert(msg,"NAME","JDOE");      /* NAME=JDOE */
minsert(msg,"req","remove");     /* req=remove */
msend(iosock,msg);

```

while receiving the message and extracting the parameters is coded as

```

mrecv(iosock,&msg);
printf("NAME=%s req=%s0,mextract(msg,"NAME"),mextract(msg,"req"));

```

Complete client and server programs which demonstrate the TAGS message passing scheme are shown below in *Program 1* and *Program 2* respectively.

One of the design goals of the message passing mechanism was to make the underlying IPC mechanism sufficiently abstract so that it could be replaced with a different communication service with few changes to the application code. For example, a strategy we considered but did not choose was to use System V asynchronous message queues, and use a socket-based surrogate process to transfer System V messages between machines. In this case the service name might identify a particular message queue rather than an IP port, since the surrogate process would always listen at a well known port.

3.3 Remote Procedure Calls

Parcel-based message passing allows a variable number of parameters in a message to be treated as a single object. Although we chose message passing as the most suitable IPC mechanism for the control portions of TAGS, we realized early on that a remote procedure call (RPC) mechanism would simplify writing user interface programs which must communicate with TAGS. We also realized somewhat later that the parcel-based system gave us the means to trivially implement an RPC-like programmatic interface to TAGS.

This RPC-like mechanism was used by the TAGS operator interface, and permitted the operator interface code to communicate with the TAGS control processes using function calls. A similar RPC-like interface for other applications that use the message passing facility could easily be provided the same way.

The function *tagsetl* (TAGS Control) builds a message from its input arguments, sends the message to TAGS, waits for a reply, and places the contents of the response message into output arguments if they were supplied. C preprocessor macros are defined using *tagsetl* for all permitted operator functions. Each macro is defined using only those input and output arguments that are germane to the function it implements; all unnecessary arguments to *tagsetl* are set to appropriate defaults. This implements a very clean, readable function-call

interface to TAGS.

The code fragment below shows some example remote procedure calls to TAGS. The status code that is returned by *tagscall* is omitted for readability.

```
char *SpoolerId, *JobId;
PARCEL parms;

TagsRestart(SpoolerId);           /* Restart spooler */
JobStatus(SpoolerId, JobId, &parms); /* Get job parms */
JobHold(SpoolerId, JobId);        /* Hold a job */
JobHold(SpoolerId, USE_ALL);      /* Hold all jobs on device */
JobHold(USE_ALL, USE_ALL);       /* Hold all jobs everywhere */
JobHold(USE_COMPUTE, JobId);     /* Hold job wherever it is */
QueEnaEnq(SpoolerId);           /* Enable job enqueue */
DvcClear(SpoolerId);            /* Clear driver state */
RunSuspend(USE_ALL, USE_IMPLICIT); /* Suspend all running jobs */
```

We chose not to use the standard RPC mechanism that is available under SunOS and a variety of other UNIX variants. If we had to make the decision again, we might chose differently. However, even so, we might well makes use of parcels and packages when using RPCs. Using packages -- which are after all just nul-terminated ASCII strings -- as (perhaps the only) arguments to remote procedure calls would allow us to add new parameters and remove old ones, just as we did with messages, without necessarily recompiling modules which handle the parameters intermediately.

4. Storage Management

The basic approach to storage management of the parcel and message code is never to throw anything away. Fixed-sized structures such as parcel nodes and messages are allocated using *malloc(3)* only if none are already available from a pool maintained in the allocating function. These structures are returned to the appropriate pools when they are no longer needed.

String management is more complicated, but a similar approach is used. Pools of string buffers of various lengths, such as 2, 4, 8, 16, 32, 64, 128, 256 and 512 bytes, are kept. Requests for storage less than or equal to the length of the largest string buffer are only *malloc*-ed if the appropriate pool is empty, and even then only the selected fixed sizes are allocated. All freed strings are returned to the appropriate pool. If a string request is made for a string exceeding that of the largest available string buffer (this would have be a pretty large string), the string is *malloc*-ed in the normal fashion and *free*-d when no longer needed. The length of each string buffer is stored in the long word ahead of the string. The integrity of the length field is obviously at risk, although many storage management schemes have this same flaw. The alternative would have been to maintain a database of string addresses and lengths inside the string manager. Since the existence of the length field is not divulged to the application, a new management scheme could be implemented without impacting the application code.

As an application runs, the string and structure pools grow, stabilizing at the point at which the peak allocation can be met. This results in fast allocation of storage while avoiding heap fragmentation, at some expense of unused slack bytes in the string buffers. It may

appear that this results in a larger than necessary process data space, but most *malloc* algorithms expand the heap to meet the peak storage demand, and never return any of the freed storage to the system. It could be argued that this storage strategy reserves excess space at the expense of code which does not make use of the structure and string pools, but in TAGS such code is virtually non-existent.

A more formal C-based garbage collector was considered, and some provisions for such a collector were made in the TAGS storage management code, but in practice it proved unnecessary.

5. Discussion

Binary trees are hardly anything new, nor are Berkeley UNIX sockets. The use of parcel-like data structures, and of *flattener* and *fluffer* functions to convert between the structures and I/O streams, has mainly appeared in unpublished, often commercial, code. Data hiding and abstract data types (where the only access to a data structure is through a supplied set of functions) are commonly found in object oriented languages. However, the use of these concepts in a successful production distributed system, and the availability of the code, may be of interest to the practitioner.

The MACH operating system [Baron et al. 1988, Tanenbaum 1992] implements messages with a variable number of parameters. However, it allows parameters of many data types, and types each parameter with a prefix indicating how it should be interpreted. TAGS messages simplify the data conversion problem by passing parameters in ASCII using C string conventions, and by using a header in which thirty-two bit binary words are passed in network byte order.

The latest release of the software handles architectures which do not have thirty-two bit longs; however, there is an implicit assumption that binary numbers are stored in the typical two's complement form. Use of architecture neutral formats such as Sun Microsystems' XDR (eXternal Data Representation) [Sun 1987], which also encodes integers as thirty-two bit two's complement quantities, would overcome this limitation.

[Liskov et al. 1986] discusses the penalty in performance when using synchronous communication due to the need for both partners to rendezvous. UNIX internet stream socket implementations are *sometimes synchronous*, depending on how much of a message can be buffered inside the kernel. If an entire message cannot fit in the kernel buffers, the sender blocks on the send until the receiver issues its corresponding receive. The amount of message buffering is dependent upon kernel configuration. Tests using a typically configured SPARCstation, running SunOS 4.1.1, show that as large as an eight kilobyte message can be buffered, allowing the sender to continue without rendezvous with the receiver. The buffering of socket data inside the kernel, using chains of *mbufs*, is described in detail by [Leffler 1989].

The problem this presents to the IPC programmer is that it allows the behavior of a message passing application to change depending on the size of the messages, the number of communicating processes, and the amount of other network activity on a particular system (which may be beyond the programmer's control). Hence, a distributed application may exhibit asynchronous behavior during testing, but revert to synchronous behavior under load, with potentially significant performance consequences.

There is also a potential deadlock condition which may arise only when large messages are passed. Consider two communicating processes, each of which issues a send followed by a receive, with appropriate context switching to produce the sequence shown below.

Process 1:	Process 2:
send(Process 2)	send(Process 1)
receive(Process 2)	receive(Process 1)

With a synchronous message passing scheme, the processes would always deadlock, since each would block on its send. With stream sockets, small messages may fit completely in the kernel buffers, allowing each process to complete its send and issue its receive. With large messages, each process may block on its send, again resulting in deadlock. A similar problem when using *pipes* (which in many Berkeley UNIX-based platforms are special cases of stream sockets) is described by [Rochkind 1985].

The need for rendezvous has been mitigated in TAGS independent of kernel buffering through the use of an intermediate transaction manager, the TAGS Control Program. Control Program receives all incoming messages from senders and queues them until the receivers initiate a receive on their message socket. Control Program also serves as a convenient point from which to monitor messages between processes. This proved useful during debugging.

For systems which have asynchronous I/O and the SIGIO signal, the messages layer provides an interrupt-driven message multiplexing function that can be used by a receiver to automatically manage multiple sockets, for both connection requests and incoming messages, as a "background" activity. Incoming messages are assembled by finite state machines invoked by a signal handler. Completed messages are queued in memory for later reference by the receiver. This asynchronous *message service* is experimental. Signal semantics differ between various UNIX implementations. MACH-like threads would provide a simpler solution.

Parcels can be a hard sell. The idea of consigning data to a "black box" and not keeping it in fixed length variables of known size is a foreign one to some programmers. It is also sometimes difficult to persuade programmers to use the object oriented concepts of data hiding and abstract data types in non-object oriented languages. Many programmers balk at the perceived overhead in maintaining parcels and of using them as a medium for interprocess communication.

However, parcels greatly simplified the multi-programmer development of a medium-sized production distributed system. The ability to add parameters to messages without recompiling all the various message passing modules probably paid for their use. The fact that large collections of parameters could be passed around in a program as a single fundamental data type (a pointer), and the ease and efficiency with which they could be manipulated, was a big win.

6. Interoperability and Portability

The software described here has been successfully ported to, and tested for message passing interoperability on, the following platforms.

SMI SUN-4	SunOS 4.1
SMI SUN-3	SunOS 4.0
DEC VAX	Ultrix-32 V3.0
IBM 370/4381	VM AIX/370 1.2
IBM RS6000	AIX 3.1
H-P HP9000	HP-UX A.B8.05

The VAX architecture has a different host byte order than many other machines. The IBM 370 does not support signed characters. Hence, the fact that these two platforms can exchange parcel-based messages with the others is a good test for the claimed architecture neutrality of the code. A port to an IBM 3090/110J running MVS is partially complete and tested. Some preliminary work has been done to port the code to a CRI CRAY Y-MP under UNICOS.

7. Software Availability

The sources for the tools and IPC software libraries are available from NCAR via anonymous FTP at no cost under a non-commercial license agreement. The software is distributed as compressed *tar* files in the directory `~ftp/tagslib` on the host `ftp.ucar.edu (128.117.64.4)`. Makefiles, UNIX manual pages, and some example applications programs are included in the distribution.

See the included README files for licensing information. The author should be contacted for information on commercial use. The sources were all originally developed at NCAR and contain no code licensed from other organizations. All software is copyrighted by the University Corporation for Atmospheric Research. All rights are reserved.

The file `libtools.tar.Z` contains the C-based software tools, including the parser and support for the family of data types. The file `libipc.tar.Z` contains the C-based interprocess communication software. These files contain the complete sources (including bugs) that were originally part of the production (second release) TAGS system.

The files `newipc.tar.Z` and `newtools.tar.Z` contain the newest release of the software. This release has been tested, but has not been used in production, and hence should be considered beta test. The new code does not require that long words on communicating processors be the same size, an unfortunate assumption made in the original code. As a result, the format of the message structure differs slightly from the production libraries. It also contains some optimizations to the tree traversal algorithms, and some changes to enhance portability. The confidence in this "research" release is high.

Acknowledgements

Many thanks are due to my fellow TAGS team members, Craig Ruff and Mark Uris of SCD. They showed courage in being my beta testers, helping me to debug my code in their production programs. Thanks also to Gene Harano of SCD, who ported the code to MVS,

its first non-UNIX platform. Finally, I am grateful to Dennis Colarelli, Craig Ruff, and Professors Robert Dixon, David Hemmendinger, and Alton Sanders, who reviewed this paper and made many helpful suggestions.

Trademarks

UNIX is a trademark of UNIX Systems Laboratories.

PostScript is a trademark of Adobe Systems Incorporated.

References

- [Baron 1988] R. Baron et al., *MACH Kernel Interface Manual*, Department of Computer Science, Carnegie-Mellon University, 1988
- [Leffler 1989] S. Leffler et al., *The Design and Implementation of the 4.3BSD UNIX Operating System*, MA: Addison-Wesley, 1989, p. 296
- [Liskov 1986] B. Liskov et al., "Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing", *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, 1986
- [Rochkind 1985] M. Rochkind, *Advanced UNIX Programming*, NJ: Prentice-Hall, 1985, p. 126
- [Ruff 1991] C. Ruff, *Text and Graphics System Reference Manual*, Scientific Computing Division, National Center for Atmospheric Research, 1991
- [Sun 1987] Sun Microsystems, Inc., *XDR: External Data Representation Standard*, RFC1014, 1987
- [Tanenbaum 1992] A. Tanenbaum, *Modern Operating Systems*, NJ: Prentice Hall, 1992, p. 667

Program 1: Server

```
/*
**      SERVER
**
**      Arguments:      <service> <response>
**
*/

main(argc,argv)
int argc;
char **argv;
{
    int sock, newsock;
    MESSAGE *msg;

    /* Create server socket */
    sock=mserver(argv[1]);

    while (1) {

        /* Wait for connection request */
        newsock=mready();

        if (newsock==sock) {

            /* Accept request */
            newsock=maccept(sock);

            /* Create child to service request */
            if (fork()==0) {

                msg=NULL;
                while (1) {

                    /* Receive message */
                    if (mrecv(newsock,&msg)==0)
                        break;

                    /* Extract request */
                    printf("%s=%s\n","REQUEST",
                        mextract(msg,"REQUEST"));

                    /* Insert response */
                    minsert(msg,"RESPONSE",argv[2]);

                    /* Send reply */
                    msend(newsock,msg);

                }

                /* Release message buffer */
                mfree(&msg);
            }
        }
    }
}
```

```
/* Terminate connection */  
mclose(newsock);  
exit(0);
```

```
}  
mclose(newsock);
```

```
}
```

```
}
```

```
}
```

Program 2: Client

```
/*
**      CLIENT
**
**      Arguments: <host> <service> <request> <count>
**
*/

main(argc,argv)
int argc;
char **argv;
{
    int sock, pid, count;
    MESSAGE *msg;

    /* Open a connection to the server process */
    sock=mclient(argv[1],argv[2]);

    pid=getpid();

    /* Send as many messages as specified */
    for (count=atoi(argv[4]); count>0; count--) {

        /* Allocate and initialize a message */
        msg=message(0L,0L,(long)pid,0L,"CLIENT");

        /* Insert a request and send the message */
        minsert(msg,"REQUEST",argv[3]);
        msend(sock,msg);

        /* Receive a message and extract the response */
        mrecv(sock,&msg);
        printf("%s=%s\n","RESPONSE",
            mextract(msg,"RESPONSE"));

        /* Free the message */
        mfree(&msg);
    }

    /* Terminate the connection and exit */
    mclose(sock);
    exit(0);
}
```