Performance of a Remote Instrumentation Program

Michael Kupfer

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

ABSTRACT

One application of distributed computing is remote system instrumentation. Such instrumentation programs require good response with low overhead to provide timely results without disturbing the system being measured. A remote procedure call system, such as the Circus system developed at Berkeley, allows programmers to write distributed programs with little more effort than is required to write local programs. This paper compares a Circus-based implementation of a Berkeley UNIX† tool (vmstat) with one based on the byte-stream protocol TCP. The Circus version makes for much cleaner code, but it requires more start-up time and higher CPU overhead than the TCP version. We conclude that the present incarnation of Circus is not acceptable for our work, but that future versions of Circus should prove valuable.

1. Introduction

One application of distributed computing is remote instrumentation, which allows a user on one machine to monitor the performance of a different machine without logging on to that machine. Such a program consists of at least two processes: a data-gathering server process on the remote machine, and a data-displaying client process on the local machine. If only one server process is used, it multiplexes connections to all its clients. An alternative is to give each client process its own server process.[1] In either case the communication system seen by the client and server clearly must guarantee the accuracy of a delivered message. In addition, we feel that the communication system should guarantee message delivery. A dropped message affects the client as though the remote machine had suddenly slowed to a crawl. Thus dropped messages would unnecessarily annoy users and possibly confuse analysis programs. This problem would be tolerable if messages were infrequently lost. Unfortunately, casual instrumentation of Ethernet interfaces has shown input error rates (hence, dropped packets) of 50-100 or more per hour, which is simply too high to ignore.

Furthermore, we want a communication system that is general enough that we can easily write distributed versions of existing tools or write new distributed tools from scratch. However, the communication system should also provide adequate performance in at least two areas: program initialization and system overhead. Program initialization should be fast because we

† UNIX is a trademark of Bell Laboratories.

This work was sponsored in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under contract No. N00039-C-0235. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Goverment. Additional support was provided by the State of California MICRO program, grant number 532422 - 19900.

[1] Either technique allows the client the option of talking to multiple servers.

sometimes want the monitoring tools to tell us what is happening *now*, not what is happening 10 seconds from now. Long-term system overhead should be low enough that the tools can provide relatively long traces without disturbing the system being measured (e.g., less than 1% of the CPU should be used for the instrumentation program, and it should not cause any significant change to the system's swapping or paging behavior). For the same reason, short-term overhead—which we will not consider in this paper—should also be low. However, short-term overhead will be somewhat higher than long-term overhead because of initialization costs and because new UNIX processes usually get higher priorities than older ones.

One reliable and general communication mechanism is the remote procedure call (RPC), which by and large allows the application programmer to ignore the distributed aspects of the program. Eric Cooper's *Circus* [Cooper 84a, Cooper 84b], which is based on Xerox's *Lupine* system [Birrell 83], is a remote procedure call system that runs under Berkeley UNIX 4.2BSD. Circus differs from an earlier Berkeley UNIX RPC system [Larus 83] in that it is based on datagram service rather than on virtual circuits.

This paper evaluates the performance of a Circus-based version of *vmstat*[2] by comparing it with an implementation based on the byte-stream protocol TCP [TCP 81]. In the following section we present a brief introduction to what a programmer works with when using Circus and TCP on a 4.2BSD system. Section three describes the performance tests that were used. The results of these tests are described in section four, and an analysis of the results is in section five. In section six we present our conclusions, in section seven we suggest additional research, and in section eight we summarize the paper. Appendices A and B contain sample code from the Circus- and TCP-based programs (both client and server), with an emphasis on the differences between the two approaches.

2. Programmer's View

Circus provides the UNIX programmer with a set of facilities that are like Lupine's, except that some changes were necessary for compatibility with the Berkeley UNIX environment. First the programmer defines an interface of types, global variables, and procedure headings using a Mesa-like language derived from Xerox's *Courier* specifications [Mitchell 79] [Courier 81]. From this interface the *rig* compiler generates C code [Kernighan 78] for the server and client stubs, as well as a header file that contains C definitions for the types and variables specified in the interface. The programmer codes two programs, one for the client and one for the server. Taken together, these two parts differ little from a modular non-distributed version of the program. Most of the differences are embodied in a small amount of code that manages such chores as binding the client to the server. A run-time library and the client and server stubs handle communication between the client, the *ringmaster* binding process (which corresponds to Grapevine in the Xerox world), and the server. A programmer using Circus also has the opportunity to programmatically type-check the client/server interface with the UNIX program *lint*. Relevant portions of the Circus-based *vmstat* are in Appendix A.

As part of its Interprocess Communication (IPC) facilities [Leffler 83], 4.2BSD provides the UNIX programmer with TCP service. In contrast with using Circus, a TCP-based program requires no extra paraphernalia such as *rig* (the stub compiler). The price is that the programmer must do more work, such as explicitly opening a connection between the client and server and managing I/O errors. To handle multiple clients simultaneously, the server must either multiplex its connections or *fork* off a new server process to handle each new client. If there is a server process for each client, then the client bears the additional burden of telling its server to exit when it (the client) is ready to quit. At best, this additional work is merely an annoyance; at worst, it provides ample opportunity for programming mistakes. An additional problem with using TCP is that there is no way to verify the type-correctness of the client and server communication routines, other than checking the individual *read* and *write* statements by hand (which is also liable to mistakes). Relevant portions of the TCP-based *vmstat* are in Appendix B.

[2] vmstat produces statistics about the virtual memory subsystem in Berkeley UNIX.

Thus, the most immediate advantage of Circus is its ease of use. Another expected advantage results from Circus' use of datagram communication: we expect lower start-up overhead from using Circus than we do from using a byte-stream protocol like TCP. An expected disadvantage of Circus is that its generality may make communication slower. For example, Circus allows transparent communication between different machine types, which may lead to unnecessary message-copying or format conversion in the case where both machines are of the same type. A realistic TCP-based implementation would also have to deal with this heterogeneity problem, but it may be possible to hand-tune the communication code to obtain greater efficiency than is possible with Circus.

In short, according to our introductory criteria for a distributed monitoring tool, we expect that Circus would make an excellent tool for writing remote instrumentation programs if we could obtain adequate performance from it.

3. The Tests

We propose two types of performance tests: one test measures the elapsed start-up time required by a program; the other test measures the long-term CPU utilization of a program. The point of the start-up test is that any useful instrumentation utility must provide quick service without high initialization costs. The point of the utilization test is that any useful instrumentation utility must not significantly disturb the system it is measuring.

Our first test consisted of running a program that invoked *vmstat* 300 times and recorded the accumulated execution time. We performed this test on a VAX[3] 750 with 2 megabytes of physical memory running in single-user mode. In one case we performed the test 10 times with no competing load, and in a second case we performed the test 10 times while competing with a load of seven simulated "active" users.[4]

The second test consisted of causing *vmstat* to iterate (display one line of statistics) 10,000 times at 5-second intervals. When the test finished, both the client and the server recorded information such as their elapsed times and CPU usage. We ran this test 7 times on a VAX 780 with 4 megabytes of memory, at various hours of the day and night, without attention to machine load. We also ran a similar test—using 20,000 iterations instead of 10,000—to verify that we could extrapolate our results to times longer than a day. We picked 5 seconds as the interval length because the Berkeley UNIX kernel updates its virtual memory statistics at 1- and 5-second intervals. We did not repeat the tests using a 1-second interval because, as we shall see in the next section, the CPU utilization at the 5-second refresh rate was high enough that additional tests seemed pointless.

4. Results

The results of the first test are summarized in Table 1. Each number represents an average start-up time in seconds. We also repeated the start-up tests 3 times with the original (single-process) version of *vmstat* for rough comparison purposes.

Table 1: Start-up times for TCP- and Circus-based versions

version	with load	no load
Circus	4.63	1.36
TCP	1.14	0.708
original	10.2	2.04

Table 2 gives the results for the second test. We obtained these numbers by compiling code into the *vmstat* client and server so that each program recorded its elapsed ("wall-clock") running

[3] VAX is a trademark of Digital Equipment Corporation.

[4] Each user was simulated by a shell script that repeatedly did tasks such as compilation, editing, and file copying.

time and its system and user CPU requirements. We calculated the "percent of system used" as the sum of the CPU time used divided by the elapsed time. Notice, however, that we ignore the requirements of *ringmaster* (the binding process) for the Circus version, and we ignore certain one-time start-up costs for both versions. Again, we also repeated the test a few times with the original *vmstat* for rough comparison purposes.

Table 2: Long-term CPU utilization

	system time (sec)	user time (sec)	% of system used
Circus client	332.4	190.6	1.01
Circus server	437.4	123.6	1.09
Circus (total)			2.10
TCP client	50.3	108.7	0.32
TCP server	176.6	47.0	0.44
TCP (total)			0.76
original (total)	97	117	0.44

5. Analysis

Having seen these results, we now must interpret them.

5.1. Start-up Delay

Table 1 shows that the TCP version of the program consistently starts up faster than the Circus version. This result may seem surprising, as byte-stream protocols have a reputation for high overhead in establishing connections. However, the protocol-related activities may only be a small part of all the program's activities. Using the *gprof* profiler [Graham 82], we see that for one run of 100 iterations, the TCP-based client spends 30 ms (1% of its total CPU time) establishing a connection; the Circus-based client requires less than 10 ms (0.2%) of CPU time to connect to the server. However, to find a server, the Circus-based client must send a message to *ringmaster* and then wait for a reply, which is entirely transparent to *gprof* and is presumably slow. Contrast this with the TCP-based version, which spends 100 ms looking up the server's Internet address in a well-known file. Thus, we hypothesize that the Circus-based client process requires less CPU time than the TCP-based client, but it requires more elapsed time because of client-server binding.

Both versions are much faster than the original version of *vmstat*. We obtain this savings because the original version does an *nlist,* which tells where the interesting numbers live in kernel memory, each time it is invoked. Both of the experimental versions do only one nlist, when the server is started up, and they re-use that information when a new client executes. Thus the comparison between the original and experimental versions is biased, but it points out an advantage of using the client/server paradigm for UNIX instrumentation programs.

5.2. CPU Utilization and Steady-State Delay

As with the first test, the Circus version performs worse than the TCP version: it uses about 3 times as much of the CPU as the TCP version does. There are many causes for this difference, some of which are inherent to an RPC system, some of which result from Circus's design, and some of which result from Circus's implementation, which is untuned and entirely at the user level. One inherent problem of the RPC-based system is that it must send a message to the server for every information message that the server sends back. In the TCP-based system, the server just keeps sending information until the client sends one message ending the link. *Gprof* analysis suggests that the Circus-based client can spend up to 14% of its time (ie., 0.14% of the CPU) just sending these request messages. Also, the Circus-based version incurs extra byte-copying costs (compared with the TCP-based version) in parameter passing. This copying comes from moving whole structures around; the TCP-based version just moves pointers.

There is another problem that is not inherent to RPC systems in general but which results from Circus's charter as a *reliable* remote procedure call system. Circus allows more than one process to export a given service. When a client makes a remote call, the client stub sends requests to all servers that export that call and uses a voting scheme to determine the result that it returns. Circus does provide a mechanism that allows the client to specify which server (or servers) to use. However, the techniques necessary to handle communication in the general case (multiple servers) can have appreciable cost even when only one server is called. Thus remote instrumentation programs, which do not need this replication mechanism, must bear this added cost when using Circus.

The lack of tuning in Circus leads to problems such as unnecessary *malloc* (memory allocation) calls, expensive queueing operations, and unnecessary copying. The mallocs are done at each call, when the client stub allocates and returns buffer space. The stub could avoid these problems by maintaining its own pool of buffers. The queueing operations, which support communication, could probably be made less expensive by using register variables [Kernighan 78]. Although any general-purpose communication mechanism must provide machine independence, it seems reasonable for the stubs to recognize that they are running on compatible architectures and agree to use that architecture's data format, rather than wasting cycles converting to and from some general-purpose format.[5]

There are two good reasons for putting Circus's reliable, procedure-oriented communication protocol in the kernel.[6] Because Circus runs entirely in user space, it must implement timeouts using the *alarm* library routine, which means that Circus preempts *SIGALRM* signals. The first problem is that this preemption forces users who want an alarm-clock function to use an inefficient kludge. The second problem is one of performance. When the stub sends off a request, it must make at least four system calls: the first call sends the request, the second call sets the alarm, the third call (*select*) waits for a reply from any of the servers, and the fourth call finally reads in the result. Each additional server requires two additional system calls. A kernel-based implementation of Circus would avoid both of these problems.

6. Conclusions

The purpose of the preceding tests was to evaluate a Circus-based remote instrumentation program. These tests lead us to conclude that Circus's current incarnation is not ready for use in production programs. The problem is not the slower elapsed start-up time of the Circus-based version, which is negligible (and certainly faster than the original version of *vmstat)*. A more serious problem is the CPU overhead that Circus requires, which is twice our 1% guideline. Fortunately, tuning the performance of Circus, removing the replication mechanisms, and moving the communication code into the kernel should solve this problem. We predict that once this task has been done, Circus will be an excellent tool for distributed monitoring programs. In the meantime, programmers will have to balance their desperation for such a distributed program against the pain of writing a program based on TCP.

7. Additional Research

While this paper provides generally encouraging results, additional work should be done to confirm our optimism. In particular, we would like to repeat these tests using the non-replicated, kernel-based version of Circus being developed by Karen White [White 85], after it has been as thoroughly tuned as the Berkeley UNIX TCP implementation.

Furthermore, there are metrics other than the ones that we have chosen. The most obvious candidate for additional testing is memory usage. A server with a large working set size (e.g.,

[5] The design for a new version of Circus includes this stub-to-stub handshake and a fix for the buffer space problem.

[6] "reliable" as in "guaranteed delivery of uncorrupted data," not as in "replicated."

from buffer requirements) will certainly disturb the system which it is trying to measure, even if its measured CPU utilization is low.

8. Summary

Having identified an interesting class of distributed computing programs (remote instrumentation), we have decided on certain performance requirements and a possible technique for writing programs that belong to that class (remote procedure calls). We have built a realistic example program using this technique (by modifying vmstat), and we have obtained encouraging results by comparing this example program with a version based on a different technique (TCP). We expect that as Circus, a UNIX implementation of this technique, is refined, it will compare favorably with its competitors.

9. Acknowledgements

Thanks are owed to Eric Cooper, Circus' creator, for making Circus available and answering many questions. Thanks are also owed to Edward Hunter, Bart Miller, and assorted others for their comments and suggestions.

10. References

[Birrell 83]
A. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems,* Volume 2, No. 1, February 1984.

[Cooper 84a]
E.C. Cooper. Circus: A Replicated Procedure Call Facility. *Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems,* October 1984.

[Cooper 84b]
E.C. Cooper. Mechanisms for Constructing Reliable Distributed Programs. Ph.D. dissertation, Computer Science Division, University of California, Berkeley. In preparation.

[Courier 81]
Xerox Corporation. *Courier: The Remote Procedure Call Protocol.* Xerox System Integration Standard 038112, December 1981.

[Graham 82]
S.L. Graham, P.B. Kessler, and M.K. McKusick. Gprof: A Call Graph Execution Profiler. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices,* Vol. 17, No. 6, pp. 120-126, June 1892.

[Kernighan 78]
B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, New Jersey, 1978.

[Larus 83]
J. Larus. On the Performance of Courier Remote Procedure Calls Under 4.1c BSD. Report No. UCB/CSD 83/123, University of California at Berkeley, August 1983.

[Leffler 83]
S.J. Leffler, R.S. Fabry, W.N. Joy. A 4.2bsd Interprocess Communication Primer, draft of July 27, 1983. Computer Systems Research Group, University of California, Berkeley, 1983.

[Mitchell 79]
J.G. Mitchell, W. Maybury, and R. Sweet. Mesa language manual (Version 5.0). *Technical Report CSL-79-3,* Xerox Palo Alto Research Center, 1979.

[TCP 81]
Transmission Control Protocol. RFC 793, Information Sciences Institute Marina del Rey, California, September 1981.

[White 85]
K. White. Untitled M.S. report, Computer Science Division, University of California, Berkeley. In preparation.

Appendix A: portions of the Circus-based vmstat

```
--          @(#)vmstat.rig        1.2        10/18/84

vmstat == begin

-- The following block is from <sys/vmmeter.h>:
--
        vmmeter: type == record [
                v_swtch: long cardinal,
                v_trap: long cardinal,
                v_syscall: long cardinal,
                v_intr: long cardinal,
                v_soft: long cardinal,
                v_pdma: long cardinal,
                v_pswpin: long cardinal,
                v_pswpout: long cardinal,
                v_pgin: long cardinal,
                v_pgout: long cardinal,
                v_pgpgin: long cardinal,
                v_pgpgout: long cardinal,
                v_intrans: long cardinal,
                v_pgrec: long cardinal,
                v_xsfrec: long cardinal,
                v_xifrec: long cardinal,
                v_exfod: long cardinal,
                v_zfod: long cardinal,
                v_vrfod: long cardinal,
                v_nexfod: long cardinal,
                v_nzfod: long cardinal,
                v_nvrfod: long cardinal,
                v_pgfrec: long cardinal,
                v_faults: long cardinal,
                v_scan: long cardinal,
                v_rev: long cardinal,
                v_seqfree: long cardinal,
                v_dfree: long cardinal,
                v_fastpgrec: long cardinal,
                v_swpin: long cardinal,
                v_swpout: long cardinal
        ];

        vmtotal: type == record [
                t_rq: integer,
                t_dw: integer,
                t_pw: integer,
                t_sl: integer,
                t_sw: integer,
                t_vm: long integer,
                t_avm: long integer,
                t_rm: integer,
                t_arm: integer,
                t_vmtxt: long integer,
                t_avmtxt: long integer,
                t_rmtxt: integer,
                t_armtxt: integer,
                t_free: integer
        ];
--
-- (end of <sys/vmmeter.h>)

-- The following block is from <sys/vmsystm.h>:
--
        forkstat: type == record [
                cntfork: long integer,
                cntvfork: long integer,
```

```
            sizfork: long integer,
            sizvfork: long integer
    ];
--
-- (end of <sys/vmsystm.h>)

    doubleFloat: type = array 2 of long integer;
    CPUSTATES: integer = 4;                 -- from <sys/dk.h>
    DK_NDRIVE: integer = 4;                 -- from <sys/dk.h>

    VMSTATS: type = record [                -- package of virtual memory stats
            busy: long integer,
            time: array CPUSTATES of long integer,
            xfer: array DK_NDRIVE of long integer,
            Rate: vmmeter,
            Total: vmtotal,
            Sum: vmmeter,
            Forkstat: forkstat,
            rectime: long cardinal,
            pgintime: long cardinal
    ];

    disk_drive: type = record [             -- info for disk drives
            name: string,
            unit: integer
    ];

    Time_t: type = long integer;            -- seconds since 1jan70

    vm_init: type = record [                -- initial message to client
            drive: array DK_NDRIVE of disk_drive,
            hz: long integer,               -- clock rate
            phz: long integer               -- profiling clock rate?
    ];

    vm_info: type = record [                -- regular VM stats info
            time: Time_t,                   -- time that numbers were gotten
            s: VMSTATS,
            deficit: long integer,          -- anticipated memory deficit
            etime: doubleFloat,
            nintv: long integer             -- stats collection interval
    ];

    vmstat_info: procedure [firstCall: boolean] returns
            [infoPkt: vm_info];

    vmstat_init: procedure returns [initPkt: vm_init];

end.
```

```
#ifndef lint
static   char *sccsid2 = "@(#)vmstat.c        2.7.1.2 (kupfer/test) 4/14/84";
#endif

#include <sys/param.h>
#include <stdio.h>
#include <strings.h>
#include <sys/vm.h>
#include <sys/dk.h>
#include "vmstat_defs.h"
#define  HOSTNSIZE 255              /* host name size */
#define  YES 1
#define  NO 0

int      HZ;
char     host[HOSTNSIZE];           /* name of the host we want to talk to */
unsigned stime;                     /* Sleep time between refreshes */
vm_init  initPkt;                   /* init packet */
vm_info  infoPkt;                   /* info packet for one call */

#define  INTS(x)   ((x) - (hz + phz))


/*
 * Print VM statistics, using a remote server to collect the data.
 * Uses Eric Cooper's Circus for RPC.
 */
main(argc, argv)                                                        main
        int argc;
        char **argv;
{
        int lines;                  /* count lines for headering */
        int iter;                   /* number of iterations to make */
        extern char _sobuf[];
        boolean   firstCall;        /* flag for 1st call to server */

        if (argc > 1 && strcmp(argv[1], "-t") == 0) {
                set_trace_flags(argv[2]);
                argc -= 2;
                argv += 2;
        }

        setbuf(stdout, _sobuf);

        stime = 5;                  /* default sleep time */
        (void) gethostname(host, HOSTNSIZE);/* default host: us */
        argc--, argv++;

        /*
         * Figure out how many iterations to make and how long for each
         * refresh.  If no numbers were given, only iterate 1 time.  If
         * only a refresh interval was given, iterate forever.  Otherwise,
         * the user will tell us how many times to iterate.
         */
        if (argc < 1)
                iter = 1;
        else {
                stime = atoi(argv[0]);
                if (argc == 1)
                        iter = 0;           /* close enough to infinity... */
                else
                        iter = atoi(argv[1]);
        }
```

```
        /*
         * Import the vmstat interface and get 1-time info about the clock
         * rate and drives.
         */
        set_troupe_list(1, host);
        if (!import_vmstat()) {
                fprintf(stderr, "can't import vmstat\n");
                exit(1);
        }

        initPkt = vmstat_init();
        HZ = initPkt.phz ? initPkt.phz : initPkt.hz;
        firstCall = YES;

reprint:
        lines = 20;
printf("\
 procs      memory   %-18.18s page          disk    faults          cpu\n\
 r b w   avm  fre   re at  pi  po  fr  de  sr %c%d %c%d %c%d %c%d  in  sy  cs us sy id\n",
 host,
 initPkt.drive[0].name[0], initPkt.drive[0].unit,
 initPkt.drive[1].name[0], initPkt.drive[1].unit,
 initPkt.drive[2].name[0], initPkt.drive[2].unit,
 initPkt.drive[3].name[0], initPkt.drive[3].unit);

loop:
        infoPkt = vmstat_info(firstCall);
        firstCall = NO;
        displayinfo(&infoPkt);
        if (--iter == 0) {
                exit(0);
        }
        go_to_sleep(stime, 0);          /* (Zero microseconds) */
        if (--lines <= 0)
                goto reprint;
        goto loop;
}
```

```
#ifndef lint
static char sccsid[] = "@(#)vmstatd.c          3.8.1.2 (kupfer/test) 4/14/84";
#endif

#include <stdio.h>
#include <strings.h>
#include <sys/ioctl.h>
#include <sys/param.h>
#include <sys/file.h>
#include <sys/vm.h>
#include <sys/dk.h>
#include <nlist.h>
#include <sys/buf.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
#ifdef vax
#include <vaxuba/ubavar.h>
#include <vaxmba/mbavar.h>
#endif
#include "vmstat_defs.h"
#define YES     1
#define NO      0

/*
 * This program simply provides subroutines which the client calls via an
 * RPC mechanism.  Vmstat_init is called once (per client), so that we
 * don't waste time retransmitting repetitious information (eg., clock
 * rate).  After that, vmstat_info is called every time a client wants new
 * information.
 */

extern int errno;
unsigned stime=1;               /* interval between refreshes */
time_t   boottime;
vm_info  avgInfo;               /* boottime to now averages */
vm_info  runningInfo;           /* running rates */
vm_init  initPkt;               /* packet of init info */
vm_info  infoPkt;               /* packet of regular info */
time_t   lastRefresh=0;         /* time of last refresh */
time_t   now;                   /* (the current time) */
```

main

```
main(argc, argv)
int argc;
char *argv[];
{
        int i;
        time_t initialize();

        if (argc > 1)
                stime = atoi(argv[1]);
        boottime = initialize();        /* get clock, disk drive info */
        refresh();                      /* read 1st set of values */

        if (!export_vmstat()) {
                fprintf(stderr, "can't export vmstat\n");
                exit(1);
        }

        /*
         * Disassociate ourselves from our parent.  This is especially
         * needed if you use rsh to start up the server.
         */
#ifndef noOrphan
```

```
        /* ... */
#endif

        server_loop();
}

/*
 * Get the namelist for the kernel and do any one-time reading of kernel
 * memory.  Return the system boot time, and set "now" to be the current
 * time.
 */
time_t
initialize()
{
/* ... */
}
```

initialize

```
/*
 * Send initial data to the client: clock rate and disk info.
 * Be sure to update our buffers if we haven't been called in a long
 * time.
 */
vm_init
vmstat_init()
{
        if (time(&now) - lastRefresh >= stime) {
                lastRefresh = now;
                refresh();
        }
        return(initPkt);
}
```

vmstat_init

```
/*
 * Send a message with the vmstat info in it.  The argument specifies
 * whether this is the client's first call.  If it is, the server should
 * give average numbers (averaged since system boot).  Otherwise,
 * the server should give the going rate.
 */
vm_info
vmstat_info(firstcall)
boolean firstcall;
{
        if (time(&now) - lastRefresh >= stime) {
                lastRefresh = now;
                refresh();
        }

        if (firstcall)
                bcopy(&avgInfo, &infoPkt, sizeof(vm_info));
        else
                bcopy(&runningInfo, &infoPkt, sizeof(vm_info));

        return(infoPkt);
}
```

vmstat_info

```
/*
 * Refresh the "avgInfo" and "runningInfo" buffers.  Uses the global
 * current time ("now").
 */
refresh()
{
        time_t interval;

        interval = now - boottime;
        getinfo(interval, &avgInfo, &runningInfo);
}
```

refresh

Appendix B: portions of the TCP-based vmstat

```
/* @(#)vmstat.h    2.2    3/12/84 */

#define VMSTAT_EXIT    'X'          /* message: kill the current connection */
#define ERRLOG "vmstatd.errlog"     /* error log for the server daemon */

typedef struct
{
        int       busy;
        long      time[CPUSTATES];
        long      xfer[DK_NDRIVE];
        struct    vmmeter  Rate;
        struct    vmtotal  Total;
        struct    vmmeter  Sum;
        struct    forkstat Forkstat;
        unsigned  rectime;
        unsigned  pgintime;
} VMSTATS;

/*
 * The variables in the following block are all sent to the client at
 * one time or another.
 */
char dr_name[DK_NDRIVE][10];
char dr_unit[DK_NDRIVE];
int       phz;
int       hz;
VMSTATS s;
time_t    now;                      /* time that we read from /dev/kmem */
int       deficit;
double    etime;
unsigned  stime;                    /* sleep time as specified by command line */
int       nintv;                    /* now - boottime (1st pass only)*/

#define   rate          s.Rate
#define   total         s.Total
#define   sum           s.Sum
#define   forkstat      s.Forkstat

/*
 * INITBUFSIZE is the num of bytes needed to buffer the initialization data:
 *         dr_name, dr_unit, phz, and hz.
 * MESGBUFSIZE is the num of bytes needed to buffer one message:
 *         now, s, deficit, etime, and nintv.
 */
#define INITBUFSIZE (10*DK_NDRIVE*sizeof(char) + DK_NDRIVE*sizeof(char) \
        + sizeof(int) + sizeof(int))
#define MESGBUFSIZE (sizeof(time_t) + sizeof(VMSTATS) + sizeof(int) \
        + sizeof(double) + sizeof(int))

char      vms_initbuf[INITBUFSIZE];
char      vms_mesgbuf[MESGBUFSIZE];
```

```
#ifndef lint
static  char *sccsid2 = "@(#)vmstat.c        2.8 (kupfer/test) 4/12/84";
#endif

#include <sys/param.h>
#include <signal.h>
#include <netdb.h>
#include <stdio.h>
#include <strings.h>
#include <sys/vm.h>
#include <sys/dk.h>
#include <sys/socket.h>
#include "vmstat.h"
#define HOSTNSIZE 100                   /* host name size */
#define YES 1
#define NO 0

int     HZ;
char    host[HOSTNSIZE];                /* name of the host we want to talk to */
int     sock;                           /* socket for talking to the server */

#define INTS(x)  ((x) - (hz + phz))


/*
 * Print VM statistics, using a remote server to collect the data.
 */
main(argc, argv)                                                          main
        int argc;
        char **argv;
{
        int lines;                      /* count lines for headering */
        int iter;                       /* number of iterations to make */
        extern char _sobuf[];
        struct hostent *hp = NULL;              /* points to host description */
        int sockopen();                 /* opens a socket connection to the server */
        int quit();

        (void) signal(SIGINT, quit);            /* otherwise server dumps core */
        (void) signal(SIGHUP, quit);
        setbuf(stdout, _sobuf);

        stime = 5;
        (void) gethostname(host, HOSTNSIZE);
        host[HOSTNSIZE-1] = '\0';
        hp = gethostbyname(host);               /* (default = host we're on) */
        argc--, argv++;

        /*
         * Figure out how many iterations to make and how long for each
         * refresh.  If no numbers were given, only iterate 1 time.  If
         * only a refresh interval was given, iterate forever.  Otherwise,
         * the user will tell us how many times to iterate.
         */

        if (argc < 1)
                iter = 1;
        else {
                stime = atoi(argv[0]);
                if (argc == 1)
                        iter = 0;               /* close enough to infinity... */
                else
                        iter = atoi(argv[1]);
        }

        sock = sockopen(hp, "vmstat");
```

```
        /*
         * Tell the server how long to sleep and get initial (1-time)
         * info.
         */
        write(sock, (char *)&stime, sizeof stime);
        recv_init(sock);
        HZ = phz ? phz : hz;

reprint:
        lines = 20;
printf("\
 procs     memory     %-18.18s page         disk    faults          cpu\n\
 r b w    avm  fre    re  at   pi   po   fr   de   sr %c%d %c%d %c%d %c%d   in   sy   cs us sy id\n",
 host,
 dr_name[0][0], dr_unit[0], dr_name[1][0], dr_unit[1],
 dr_name[2][0], dr_unit[2], dr_name[3][0], dr_unit[3]);

loop:
        recvinfo(sock);
        displayinfo();
        if (--iter == 0) {
                cleanup(sock);
                exit(0);
        }
        if (--lines <= 0)
                goto reprint;
        goto loop;
}

/*
 * Catch a SIGINT and quit.
 */
quit()                                                                  quit
{
        cleanup(sock);
        exit(0);
}


/*
 * Tell the server that we are done; flush any data remaining in
 * the connection.
 */
char quitnow[] = VMSTAT_EXIT;       /* message to close the connection */
                                                                        cleanup
cleanup(asock)
int asock;                          /* the socket we've been using*/
{
        char fbuf[4096];            /* buffer for flushing the connection */

        if (send(asock, quitnow, sizeof(quitnow), MSG_OOB) < 0) {
                perror("vmstat: sending quitnow");
                exit(1);
        }
        while (read(asock, fbuf, sizeof fbuf) > 0)
                ;
        (void) close(asock);
}
```

```
#ifndef lint
static char sccsid[] = "@(#)vmstatd.c        3.9 (kupfer/test) 4/12/84";
#endif

#include <signal.h>
#include <sys/time.h>
#include <sys/file.h>
#include <netdb.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/param.h>
#include <netinet/in.h>
#include <errno.h>
#include <sys/socket.h>
#include <sys/vm.h>
#include <sys/dk.h>
#include <nlist.h>
#include <sys/buf.h>
#include <sys/wait.h>
#include <sys/resource.h>
#ifdef vax
#include <vaxuba/ubavar.h>
#include <vaxmba/mbavar.h>
#endif
#include <strings.h>
#include "vmstat.h"
#define YES 1
#define NO 0

/*
 * This program creates a socket, does some initial reading of the
 * kernel memory, and then begins an infinite loop.
 * Each time through the loop it accepts a connection and forks off
 * a child to manage it.  The child simply sends vmstat numbers thru it.
 * When the client tells the child to quit, the child closes the
 * old connection and exits.
 */

int msgsock;                        /* the socket connecting with the client */
extern int errno;
```

main

```
main()
{
        int i;
        int sock;
        int reaper();               /* reaps exit'd child processes */
        time_t boottime;
        time_t initialize();
        int servsock();             /* sets up a socket for the server */

        sock = servsock("vmstat");
        boottime = initialize();
        (void) signal(SIGCHLD, reaper);

        /*
         * Disassociate ourselves from our parent.  This is especially
         * needed if you use rsh to start up the server.
         */
#ifndef noOrphan
        /* ... */
#endif

        /*
         * Start accepting connections.  The accept might fail if we get
         * interrupted by a child's exiting.  If this happens, just try
         * again.  In case of an unexpected error, we pause first before
```

```
	 * retrying.  There is a 2-fold motive for this: (1) given some
	 * breathing room, maybe the kernel will get its act together and
	 * then we won't get the same error again. (2) at least the error
	 * log file won't grow so rapidly.
	 */
	(void) listen (sock, 5);
	for (;;) {
		msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
		if (msgsock < 0) {
			if (errno != EINTR) {
				perror("vmstatd: accept");
				sleep ((unsigned) 5);
			}
			continue;
		}
		if (fork() == 0) {
			(void) close(sock);		/* (child) */
			doTheDirtyWork(boottime);
		}
		else
			(void) close(msgsock);
	}
}

/*
 * Wait until a child process exits.
 */
reaper()
{
	union wait status;

	while (wait3(&status, WNOHANG, (struct rusage *)0) > 0)
		;
}

/*
 * Get the namelist for the kernel and do any one-time reading of
 * kernel memory.  Return the system boot time.
 */
time_t
initialize()
{
/* ... */
}

/*
 * The child first reads the sleep time (ie., the sampling rate) and
 * sends initial information such as the device names, their "unit"s,
 * and some clock info.  It then recalculates how long it's been since
 * system boot (this is used once, so that we can get the long-term
 * average rates since boot time; after that, we use the current rates).
 */
doTheDirtyWork(boottime)
time_t boottime;			/* system boot time */
{
	int oob();			/* catches SIGURG */

	(void) signal(SIGURG, oob);
	{int pid = -getpid();
	(void) ioctl(msgsock, SIOCSPGRP, (char *)&pid);}

	send_init();
	read(msgsock, (char *)&stime, sizeof stime);

	(void) time(&now);
	nintv = now - boottime;
```

reaper

initialize

doTheDirtyWork

```
        for (;;) {
                getinfo();
                sendinfo();
                sleep(stime);
                nintv = 1;              /* (for getinfo's sake) */
        }
}

/*
 * Catch a SIGURG: read in and process a message from the client.
 */
oob()
{
        char abuf[1];

        (void) recv(msgsock, abuf, sizeof abuf, MSG_OOB);
        switch (abuf[0]) {
        case VMSTAT_EXIT:
                (void) close(msgsock);
                exit(0);
                break;
        default:
                fprintf(stderr, "vmstatd: unknown request: 0%o\n", abuf[0]);
                break;
        }
}

/*
 * Package the collected info in a buffer and send it off to the client.
 */
sendinfo()
{
        char *bufp = vms_mesgbuf;               /* points into the buffer */

        bcopy((char *)&now, bufp, sizeof now);
        bufp += sizeof now;
        bcopy((char *)&s, bufp, sizeof s);
        bufp += sizeof s;
        bcopy((char *)&deficit, bufp, sizeof deficit);
        bufp += sizeof deficit;
        bcopy((char *)&etime, bufp, sizeof etime);
        bufp += sizeof etime;
        bcopy((char *)&nintv, bufp, sizeof nintv);

        write(msgsock, vms_mesgbuf, MESGBUFSIZE);
}

/*
 * Move the device names and clock info into a buffer and then send it
 * all off to the client.
 */
send_init()
{
        char *bufp = vms_initbuf;               /* points into the buffer */

        bcopy(dr_unit, bufp, sizeof dr_unit);
        bufp += sizeof dr_unit;
        bcopy(&dr_name[0][0], bufp, sizeof dr_name);
        bufp += sizeof dr_name;
        bcopy((char *)&hz, bufp, sizeof hz);
        bufp += sizeof hz;
        bcopy((char *)&phz, bufp, sizeof phz);

        write(msgsock, vms_initbuf, INITBUFSIZE);
}
```

oob

sendinfo

send_init