Jipc Timings

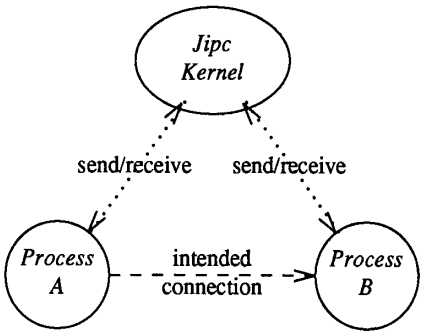
David W. Hankinson

Introduction

This paper presents empirical data on the performance of Jade's inter-process communication facility (Jipc) and compares this with the performance of the Unix† TCP protocol. Its primary use is in the evaluation of Jipc performance compared with another protocol residing on the same implementation environment. This paper is of interest because it presents the user with data that compares the two distinct systems' performance with respect to each other, both when they are within their respective design criteria and when those inherent design limits have been purposefully transgressed.

The Jipc Message Passing Protocol

The Jipc kernel is based on a *blocking send* message passing protocol. This means that when process A sends a message to process B, A is in a blocked state until B responds to the message A initially sent.



The diagram above depicts the flow of messages to and from the Jipc device driver running in the Unix kernel (labeled *Jipc Kernel*). The two circles represent two processes that are to communicate with each other through Jipc. Sending a message runs clockwise. Replying to a message, and to get a reply, runs counter-clockwise. Thus to send a message from process A to process B, a total of 4 copies of the message buffer can occur. These are from A to the Kernel, from Kernel to B, from B back to the Kernel and finally from the Kernel back to A, in this order. Furthermore, with each transmission of a message from A to B, an *implied connect* occurs. This means that A is not directly connected to B in the Unix kernel, though A would have B's address by way of the initialization routines. When the message is transferred, connection is established. After B receives A's message, if A is to continue to execute, a reply must occur which effectively breaks the implied connection between the two processes.

The above only applies when Jipc is running in the Unix domain. For the Jade 68000-based workstation, there is no Unix. The multi-tasking kernel itself runs Jipc. The overhead may not be the same as in the Unix kernel due to a totally different implementation.

---

† UNIX is a Trademark of Bell Laboratories.

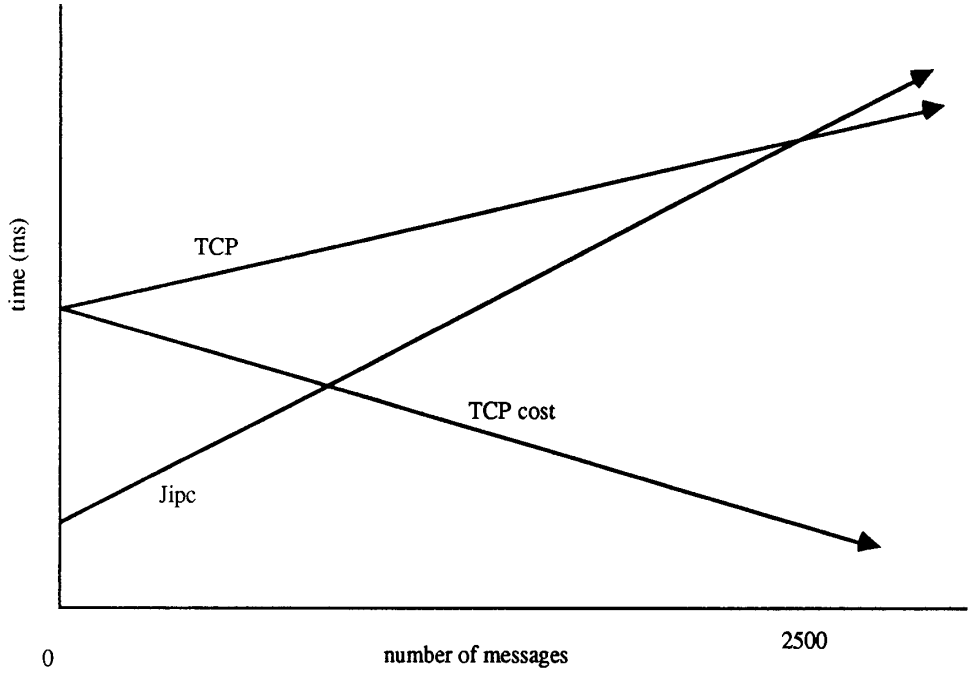
The TCP Message Passing Protocol

TCP is implemented on top of IP, which is the lowest level of networking in the Unix kernel. Communication between processes is established using system calls. TCP's protocol is based on streams. Hence, after communication is established, large amounts of data may be reliably transferred to the waiting process through standard system calls. Internally, within TCP, the only backwards communication between the two co-operating processes is the acknowledgment of the last broadcast packet. This is required as a reliability measure for TCP. TCP hides this from the user, as it is a higher level functioning layer. Within the Unix kernel a stream of TCP data is broken down into packets and transmitted. The TCP layer takes care of reforming the packets into the stream at the receiving end. TCP preforms all reliability checks as it is built over top of IP which does not have any transmission checking built in.

A Comparison between TCP and Jipc

The most dramatic difference in the two IPC systems is the protocol and the way that Jipc must send acknowledgements to its sender if that process is to continue execution. Typically then, the performance of the systems is quite different, as shown in the performance graph below.

Performance Graph



On the ordinate one has the number of messages transferred. For Jipc this means a single *j_send/j_receive/j_reply* sequence. For TCP this translates approximately into the system call *write*. Both of these are kernel entries, although the only real communication Jipc has with the Unix kernel is through the *j_ioctl* call to the Jipc device driver, residing in the Unix kernel. Time is on the abscissa. Time units are not relevant, though they are linear.

Jipc messages vary in size (from 0 to 350 bytes). The message transmission mechanism is optimized for the range of message sizes available. Therefore the performance line for Jipc is really a family of lines but one may assume that they are one. TCP also has an optimum size for transmission of data (somewhere in the range of 1k, or multiples of 1k). For the purposes of this paper, I have selected the message size to be that of a single 32-bit integer (4 bytes). This is uniform for both protocols. The number of messages transferred between the processes is 10000.

When Jipc sends a message, there is an implied connection established between the sender and the receiver. If a null size message is sent then the time to send this is roughly the overhead for a Jipc message. A formula may be derived that represents the time it takes for Jipc to send a message.

$$t = n\ (c_1 + c_2) \tag{1}$$

This linear equation, expresses the time for n Jipc messages to reach their destination. The constant c_2 is the overhead time for Jipc, derived through null-sized message transfers. The constant c_1 is the byte transfer overhead for differing size messages. The variable n is the number of messages that are sent. Empirically, c_2 was found to be in the order of 20ms, for the monitored version of Jipc. When using the unmonitored library 10ms is the norm.

TCP has a different formula.

$$t = c_1 n + c_{2m} \tag{2}$$

Formula (2) is the time it takes for TCP to send for n messages. The constant m is the number of connects initiated. The constants c_1 and c_2 have the same meaning for both ipc systems, as mentioned above.

The performance graph clearly shows the performance lines for Jipc crossing with the one for TCP. This point is where TCP becomes more efficient in sending larger amounts of data. Recall that TCP has a much higher initial overhead. The third line on the performance graph, TCP's cost curve, demonstrates the increase in efficiency over time with the larger number of messages sent. Once the TCP connection has been established, a larger amount of data can be moved over a given time frame, as opposed to Jipc. This performance line begins where TCP's performance line originates and runs downward, linearly to the point directly below the crossover between Jipc and TCP then asymptotically to the x-axis.

The crossover point is where TCP will beat Jipc in the number of messages transferred over a given time frame. For this experiment, this number is roughly 2500 messages (monitored). For the unmonitored version, expect twice that (5000 messages). This was calculated by the average time in which TCP sent 10000 messages and Jipc doing the same. The times were extrapolated and then a message transfer crossover point was calculated by the average time difference.

Results

Test conditions are:

1) Unloaded machine.
2) 10000 message transfers.
3) Average of the results obtained from 3 trials of each test.

Jipc Timing Results					
From	To	Jipc	Jipc•	TCP	TCP†
vax	vax*	145.0	83.5	190.0	880.0
vax	vax	488.4	302.5	186.7	866.3
vax	sun	442.7	310.4	149.0	894.0
sun	vax	457.0	322.8	32.0	680.0
sun	sun*	158.8	60.5	82.0	716.0
vax	corvus	357.5	225.5	-	-
sun	corvus	367.2	244.6	-	-
corvus	sun	497.0	279.0	-	-
corvus	vax	499.2	301.0	-	-
corvus	corvus*	208.0	113.0	-	-
corvus	corvus	503.9	341.2	-	-

Legend

Note: All times are in seconds.

† denotes TCP emulating Jipc (where $nP = m$ *in formula (2)).*

• denotes unmonitored Jipc timings.

* denotes that the sending/receiving machines are the same.

Interpretation of the Results

To properly interpret the results, the protocols where discussed in the above sections. The timings here present three areas for comparison:

1) Jipc communication between various machines under Jipc.

2) TCP communication between Unix machines (m in equation (2) is 1).

3) TCP communication as above, however, emulating the *implied connect* as Jipc does (m equals n as above).

Each set of data represents the time it takes to send 10000 messages from one machine to the next. In certain cases the target machine is itself. Each has the *Real time* given in seconds. For Unix and the Jade workstation, these times are subjective since events such as context switches (involuntary) and page faults are not taken into account. All timings were done when each machine was as unloaded as possible.

The above results show a wide range of deviation from each other. Clearly the Corvus workstation timings may not be used in comparison with TCP, simply because no Unix kernel is present within. Those timings are included as a performance indicator.

Conclusions

Each ipc system is designed for different needs. For TCP the results show that it works well when a large number of bytes are to be transmitted between processes. TCP, it seems, does not work well when trying to emulate Jipc. TCP is good for large data transfers to a single process. Jipc is not as time efficient when a large number of messages are sent. Expect an order of magnitude in difference between TCP and Jipc when a large amount of data is transferred. The design of Jipc though, was not as a TCP type protocol, but rather in the use of distributed systems where very few block transfers are done. For a small number of messages, Jipc is clearly superior. In contrast to TCP, where the initial overhead is high, Jipc's overhead is much smaller when very little is sent. Jipc is also very good for sending a large number of small messages to different processes. Really, each system is designed for different needs.

Appendix 1 - TCP Benchmarking programs

File tcpserve.c

```
 1
 2  /* tcpserve - Server program for TCP domain timings. */
 3
 4  #include <sys/ioctl.h>
 5  #include <sys/file.h>
 6  #include <sys/types.h>
 7  #include <sys/socket.h>
 8  #include <netinet/in.h>
 9  #include <netdb.h>
10  #include <stdio.h>
11  #include <errno.h>
12
13  #define MYPORT 12345
14  #define die(s) (perror(s),exit(-1))
15
16  extern errno;
17
18  /* Accept stuff along descriptor f. */
19
20  doit(f)
21  { auto x; register n;
22    while ((n=read(f,(char *)&x,sizeof x))>0);
23    close(f);
24  }
25
26  main()
27  { struct sockaddr_in name,addr; int addrlen,ns,s;
28    if ((s=socket(AF_INET,SOCK_STREAM,0))<0)
29      die("tcpserve (socket)");
30    name.sin_family=AF_INET;
31    name.sin_addr.s_addr=INADDR_ANY;
32    name.sin_port=htons(MYPORT);
33    if ((bind(s,(char *)&name,sizeof(name)))<0)
34      die("tcpserve (bind)");
35    if (listen(s,5)<0)
36      die("tcpserve (backlog)");
37    for (;;)
38    { addrlen=sizeof(addr);
39      if ((ns=accept(s,(char *)&addr,&addrlen,0))<0)
40      { if (errno!=EINTR)
41          die("tcpserve (accept)");
42        continue;
43      }
44      doit(ns);
45    }
46  }
47
```

File tcptalk.c

```
 1
 2  /* tcptalk - Talk program for TCP domain timings. */
 3
 4  #include <sys/types.h>
 5  #include <sys/socket.h>
 6  #include <netinet/in.h>
 7  #include <netdb.h>
 8  #include <stdio.h>
 9
10  #define MYPORT 12345
11  #define die(s) (perror(s),exit(-1))
12
13  quit(a,b)
14  { _doprnt(a,&b,_iob+2);
15    exit(-1);
16  }
17
18  main(argc,argv)
19    char **argv;
20  { struct sockaddr_in sin; struct hostent *machine; char *msin; int s;
21    if (argc<3)
22      quit("Usage: %s host transfers\n",*argv);
23    msin= *++argv;
24    if ((s=socket(AF_INET,SOCK_STREAM,0))<0)
25      die("tcptalk (socket)");
26    if (!(machine=(struct hostent *)gethostbyname(msin)))
27      quit("Unknown host (%s)\n",msin);
28    bzero((char *)&sin,sizeof(sin));
29    bcopy(machine->h_addr,(char *)&sin.sin_addr,machine->h_length);
30    sin.sin_family=AF_INET;
31    sin.sin_port=htons(MYPORT);
32    if (connect(s,&sin,sizeof(sin))<0)
33      die("tcptalk (connect)");
34    { register i; auto x;
35      i=atoi(*++argv);
36      do
37        (void)write(s,(char *)&x,sizeof x);
38      while (--i>0);
39    }
40    close(s);
41    return 0;
42  }
43
```

File tcpslow.c

```
1
2   /* tcpslow - Talk program for TCP emulating jipc. */
3
4   #include <sys/types.h>
5   #include <sys/socket.h>
6   #include <netinet/in.h>
7   #include <netdb.h>
8   #include <stdio.h>
9
10  #define MYPORT 12345
11  #define die(s) (perror(s),exit(-1))
12
13  quit(a,b)
14  { _doprnt(a,&b,_iob+2);
15    exit(-1);
16  }
17
18  main(argc,argv)
19    char **argv;
20  { struct sockaddr_in sin; struct hostent *machine; char *msin; int s;
21    if (argc<3)
22      quit("Usage: %s host transfers\n",*argv);
23    msin= *++argv;
24    if (!(machine=(struct hostent *)gethostbyname(msin)))
25      quit("Unknown host (%s)\n",msin);
26
27    /* To emulate jipc we must open (connect) each time a transfer
28       is to occur.  This involves all the setup of the Unix
29       tcp domain sockets.
30     */
31
32    { register i=atoi(*++argv); auto x;
33      do
34      { if ((s=socket(AF_INET,SOCK_STREAM,0))<0)
35      die("tcpslow (socket)");
36        bzero((char *)&sin,sizeof(sin));
37        bcopy(machine->h_addr,(char *)&sin.sin_addr,machine->h_length);
38        sin.sin_family=AF_INET;
39        sin.sin_port=htons(MYPORT);
40        if (connect(s,&sin,sizeof(sin))<0)
41      die("tcpslow (connect)");
42        (void)write(s,&x,sizeof(x));
43        close(s);
44      } while (--i>0);
45    }
46    return 0;
47  }
48
```

Appendix 2 - Jipc Benchmarking Programs

File v_serve.c

```
 1
 2   /* vserve - vax/suna server */
 3
 4   #include <jipc.h>
 5
 6   /* Simply loop forever replying null to any messages sent this way. */
 7
 8   main()
 9   { j_enter_system(123,"server");
10     for (;;)
11       j_reply_null(j_receive_any());
12   }
13
```

File v_talk.c

```
 1
 2  /* talk- Unix talking process.
 3
 4     This process assumes that the timings come from yet another unix
 5     process (time) which will be its parent.
 6   */
 7
 8  #include <jipc.h>
 9
10  /* Find the "server" process and transfer messages of size int.
11     Machine on which to look for "server" is first argument.
12     Number of messages transferred is second argument.
13   */
14
15  main(argc,argv)
16    char **argv;
17  { j_process_id t; register i;
18    j_enter_system(123,"talker");
19    t=j_search_machine(*++argv,"server");
20    j_puti(12345);
21    i=atoi(*++argv);
22    do
23      j_send(t);
24    while (--i>0);
25    return 0;
26  }
27
```

File cv_serve.c

```
 1
 2  /* cv_serve - Server for the corvus workstation. */
 3
 4  #include <jipc.h>
 5
 6  /* Indefinite loop of j_receive_any and j_reply_null. */
 7
 8  doit()
 9  { j_initialize();
10    for (;;)
11      j_reply_null(j_receive_any());
12  }
13
14  /* Create a process whose name is "server" and whose body is
15     the above function - Required for the corvus.
16   */
17
18  main()
19  { j_process_id dude;
20    j_initialize();
21    dude=j_create_process_shared("server",doit,0);
22  }
23
```

File cv_talk.c

```
 1
 2  /* cv_talk - Corvus talk process.
 3
 4     Since the corvus does not have Unix, the ck_real_time() function
 5     is used to preform the actual timings on itself.
 6     This program simply finds the "server" process on the machine
 7     specified as the first argument, and begins transmission of
 8     4-byte messages, with the limit specified in the second argument.
 9   */
10
11  #include <jipc.h>
12
13  main(argc,argv)
14    char **argv;
15  { j_process_id t; register i,time;
16    j_initialize();
17    j_leave_system();
18    j_enter_system(123,"talker");
19    t=j_search_machine(*++argv,"server");
20    j_puti(12345);
21    i=atoi(*++argv);
22    time=ck_real_time();
23    do
24      j_send(t);
25    while (--i>0);
26    printf("elapsed real time = %d\n",(ck_real_time()-time)/100);
27    return 0;
28  }
29
```