Real-Time IPC on a client / server model: Multiple OS performances benchmark

Dominique Fober, Yann Orlarey, Stéphane Letz

▶ **To cite this version:**

Dominique Fober, Yann Orlarey, Stéphane Letz. Real-Time IPC on a client / server model: Multiple OS performances benchmark. [Technical Report] GRAME. 2001. hal-02158791

HAL Id: hal-02158791
https://hal.science/hal-02158791

Submitted on 18 Jun 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

GRAME - Computer Music Research Lab.
Technical Report - 01-08-03

Real-Time IPC on a client / server model. Multiple OS performances benchmark

D. Fober, Y. Orlarey, S. Letz
August 2001
[fober, orlarey, letz]@grame.fr

Abstract

This paper presents inter processus communication (IPC) real-time performances measured on different operating systems, including GNU/Linux, Windows 98, 2000, NT 4.0 and MacOS X. The adopted point of view is based on a client / server model. The operating systems behavior and message transmission latency times are evaluated in different contexts: with one to ten clients for the server, with systems more or less busy with alternate tasks. As we wanted to measure real world performances, the benchmarks have been applied to operating systems running standard default configurations. Each time it was possible, we compared the different systems on the base of local Unix sockets communication way. But above all, we choose the most efficient communication way per system to evaluate the overall best performances that one can expect in a client / server model.

1 Introduction

This work took its roots first in the requirements of the musical applications domain. However, the results could be equally applied to any software running in a client/server model with IPC time constraints.
In the musical domain, real-time capabilities are critical from different points of view:

- some applications are strongly dependent on hardware such as digital audio cards and need to react quickly to external events,
- musical applications need accurate and sophisticated mechanisms for time management and in particular, for scheduling.

As the corresponding pieces of software are particularly tricky to design, they are generally implemented once for all by servers which may be included as part of a specific software architecture (like JMAX[1]) or opened to external clients (LAAGA[2] or MidiShare[3]). This client/server model requires then efficient real-time communication capabilities in order to provide the server services to its clients. The design of such an architecture has been frequently approached at systems low level layers: operating systems generally provide built-in specific services for digital audio, they also allow to extend the kernel capabilities at low level layer (kernel modules for Linux, kernel extensions for MacOSX or device driver for Windows). However, such implementation of real-time IPC services is particularly system dependant and non-portable. The goal of these benchmarks is to evaluate the cost of more portable implementations, based on user level system services.
The next section presents the client / server model used for these benchmarks. Section 3 details the implementation on the different platforms. Section 4 presents the benchmark results and section 5 summarizes these results. The source code of the Linux implementation is provided in appendix.

2 The client / server model

The server application creates a real-time priority thread which task consists in sending a time stamped message to all its clients at a regular time interval. The time stamp is collected once at the sending loop entrance. Pseudo code of the server task is in figure 1. For some system dependant implementations, the time stamp transmission is implemented using shared memory. Unless specified, the message is used to transmit the current time.

[1] JMAX : http://www.ircam.fr/equipes/temps-reel/jmax/
[2] LAAGA: http://www.eca.cx/laaga/
[3] MidiShare: http://www.grame.fr/MidiShare

```
Server Task
    do forever:
        put the current time in the message
        for each client
            send the message
        sleep 10 ms
```

figure 1: the server task pseudo code

The client application also creates a real-time priority thread which task consists in reading incoming messages and computing the corresponding transmission times. It stores these transmission times in a statically allocated array and quit when there is no space left. At exit, the application flushs all the latency values on its standard output and print a transmission summary (minimum, maximum and average times). Pseudo code of the client task is in figure 2.

```
Client Task
    while array not full
        read incoming message
        get the current time
        compute the transmission latency
        store at next array location
```

figure 2: the client task pseudo code

3 Implementations

Performances have been measured on 5 different stations running 7 different operating systems. On 3 stations, 2 different operating systems was installed, which makes possible to compare their results on the same hardware base. Table 1 summarizes the hardware and software configurations. The symbolic name will be further used to refer to the corresponding machine.

processor	memory	OS	symbolic name
AMD Duron 700 Mhz	128 Mo	Linux 2.2.15 Windows 98	P700
Pentium II 350 Mhz	128 Mo	Linux 2.4.3 Windows 2000	P350
Pentium II 400 Mhz	128 Mo	Windows NT 4.0	P400
PowerPc G4 350 Mhz	192 Mo	MacOSX - Darwin 1.3 Linux PPC 2.2.15	Mac350

table 1: hardware and software configurations

Threads management and messaging system are platform dependant. Their implementation is detailled below.

3.1 Threads management

On Linux stations and on Darwin, threads are implemented using the *pthread* library. On Windows operating systems, we used the Windows threading system. For both systems, in order to minimize context switches, the server sender thread priority was higher than the client reader thread priority. Table 2 summarizes the threads priority.

thread system	pthread	windows
sender thread	SCHED_RR priority 99	THREAD_PRIORITY_TIME_CRITICAL HIGH_PRIORITY_CLASS
reader threads	SCHED_RR priority 98	THREAD_PRIORITY_TIME_CRITICAL NORMAL_PRIORITY_CLASS

table 2: threads priorities

We also tried to use SCHED_FIFO as thread policy without noticeable differences.

3.2 Communication systems

Communication using local Unix sockets has been used on Linux and Darwin. On Windows, we used the Windows messaging system (PostThreadMessage). Due to the bad performances measured on Darwin using sockets, we replace them with the Mach 3 kernel IPC system (mach_msg).

Sockets was opened in datagram mode and the server transmission socket was used in non-blocking mode.

3.3 Time measurement

On Linux stations and on Darwin, times are collected using *gettimeofday*. The additionnal cost of the time collection has been measured using the LMBench[4] tools. On Windows, times are collected using *QueryPerformanceCounter*.

3.4 Linux module implementation

In order to compare socket communication with the most possible efficient implementation, on linux x86, the client / server model has also been implemented using a kernel module for the server part. In this case, there is no use of IPC system: the server runs a system time task and the message sending procedure is replaced by a client threads wake up procedure (using *wake_up_interruptible*). The current server time is stored using a shared memory block. On the client side, the real-time thread collects the wake up event (equivalent to a message passing), then the elapsed time and returns to sleep state.

4 Measurements

For each system, performances have been measured with a variable number of clients: from 1 to 5, then with 7 and 10 clients. During all these benchmarks, the system was idle. Then we measured the effect of different tasks on a single client performance. Three type of tasks was performed:

- launching an application: netscape communicator on Linux, the Chess application on MacOS X and Internet Explorer on Windows.
- connecting to a local ftp server and getting a 1 Mb file.
- moving a window.

At the end of the session, each client generates the list of the transmission times and a summary report which includes the minimum, maximum and average latency times.

4.1 Single client performances

4.1.1 Linux module vs socket

We compared the performances of the module and socket implementations on P350, the station running Linux 2.4.3. The first benchmark presents a single client with no alternate task (figure 3), the following figures (4, 5 and 6) presents benchmarks in a busy context. Timings are in microseconds.

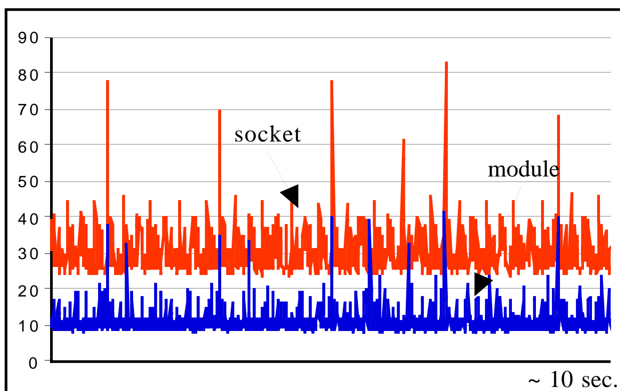

figure 3: a single client benchmark

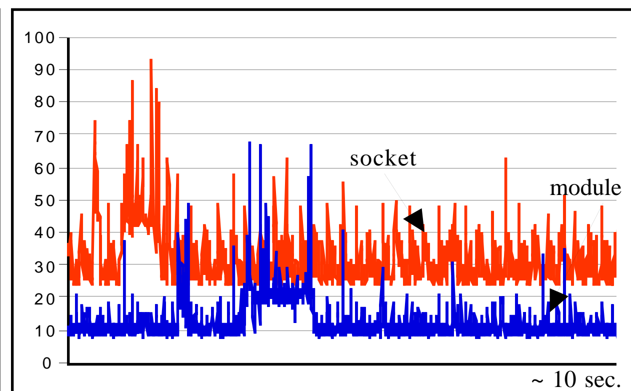

figure 4: a single client with ftp session

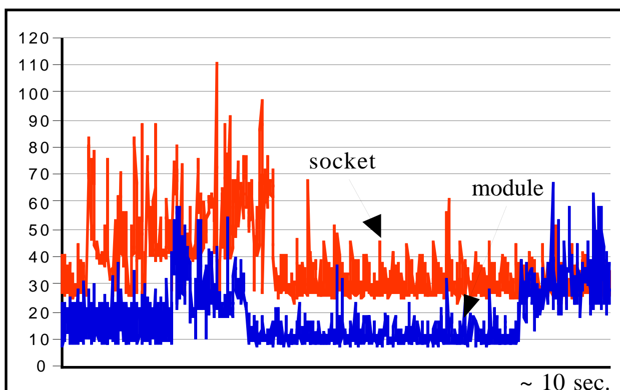

figure 5: a single client while lauching an application

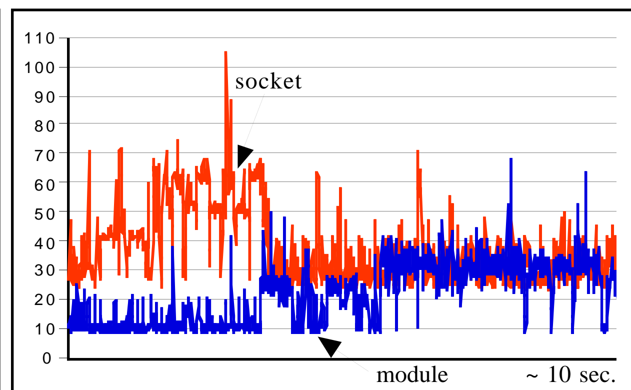

figure 6: a single client while moving a window

[4] LMBENCHS: http://www.bitmover.com/lmbench/

On figure 3, it appears clearly that the communication latency fits globally in a constant range with more or less frequent peaks. We'll further refer to this range as the *latency range*. It is characterized by its width and its medium value. The width is computed as 2 times the standard deviation and the medium value represents the arithmetic mean. The *latency peaks* are characterized by their amplitude and their frequency. To simplify the behavior description, the frequency will be qualified as low, medium or high. Tables 1 and 2 presents the latency range and the latency peaks characterizations corresponding to the figure 3.

latency range	width	medium value
module	8	12
socket	12	31

Table 1: single client latency range (fig. 3)

latency peaks	amplitude	frequency
module	from 32 to 40	low
socket	from 60 to 80	low

Table 2: single client latency peaks (fig. 3)

When the system is busy with other tasks, it also appears clearly (fig. 4 to 6) that the latency range increases in width and medium value, and that the latency peaks increase in amplitude and frequency. Table 3 summarizes the system disturbance for the figures 4 to 6.

	max range	max peak
module	from 25 to 40	71
socket	from 40 to 80	111

Table 3: busy system characterization

As expected, the module implementation is between 2 and 3 times more efficient than the socket implementation for a single client.

4.1.2 Linux socket vs Windows 2000

We compared the performances of Linux socket implementation and the Windows 2000 communication system. The benchmark has been made on the same station, P350, running both Windows 2000 and a Linux kernel 2.4.3. Timings in figure 8 to 10 are represented in log base 2.

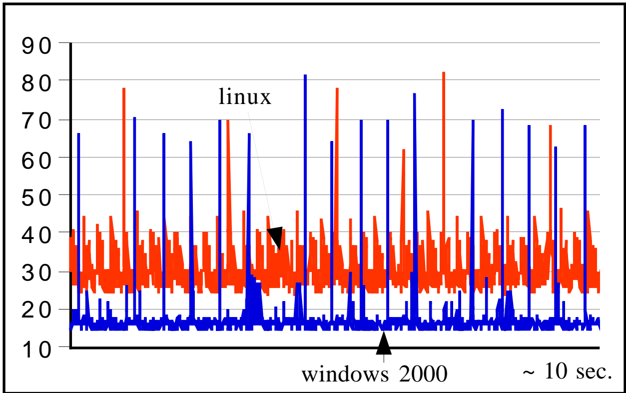

figure 7: a single client benchmark

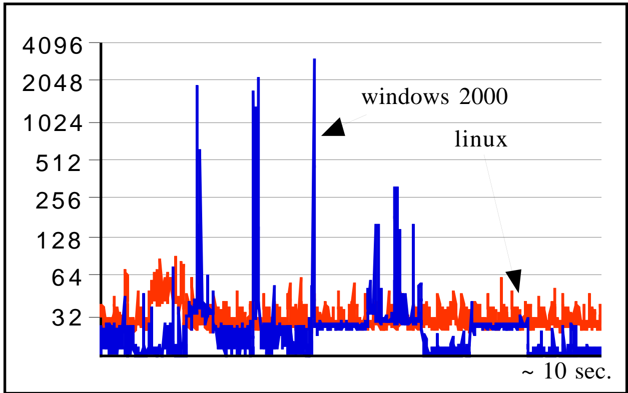

figure 8: a single client with ftp session

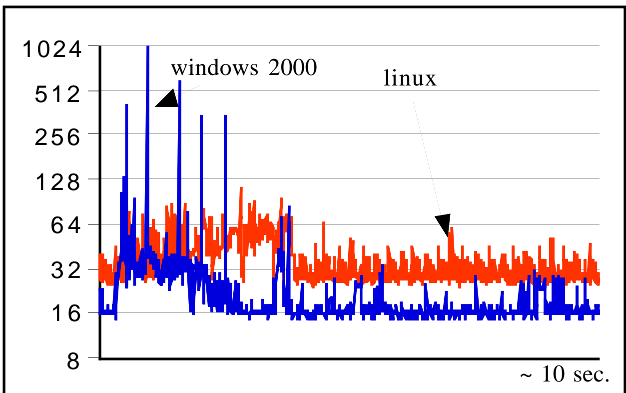

figure 9: a single client while lauching an application

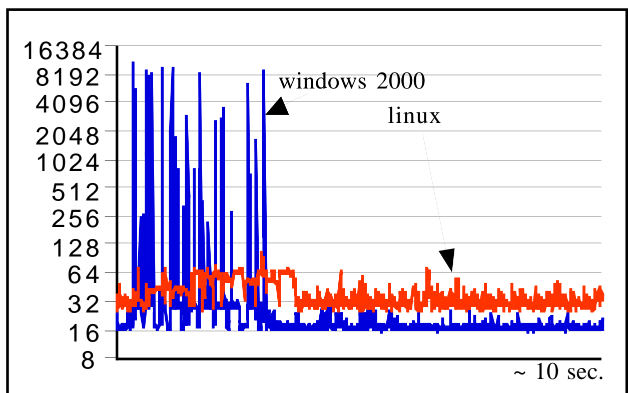

figure 10: a single client while moving a window

Tables 4 and 5 presents the latency characterization corresponding to the figure 7. Table 6 summarizes the system disturbance for the figures 8 to 10.

latency range	width	medium value
linux	12	31
windows 2000	14	17

Table 4: single client latency range (fig. 7)

latency peaks	amplitude	frequency
linux	from 60 to 80	low
windows 2000	from 60 to 80	medium

Table 5: single client latency peaks (fig. 7)

	max range	max peak
linux	from 40 to 80	111
windows 2000	na	> 8000

Table 6: busy system characterization

It appears that when the system is idle, Windows 2000 performances are better than Linux. However, it also appears clearly that Linux is really more stable than Windows 2000 in a busy environment. In this case, Linux is better than Windows to a large extent.

4.1.3 Linux socket vs Windows 98

We compared the performances of Linux socket implementation and the Windows 98 communication system. The benchmark has been made on the same station, P700, running Linux 2.2.15. As previously, timings in figure 12 to 14 are represented in log base 2.

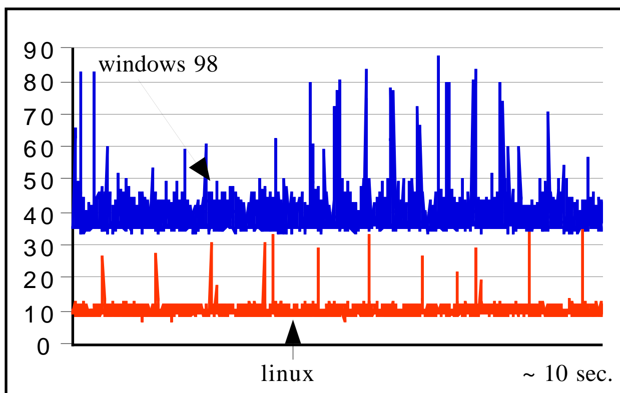

figure 11: a single client benchmark

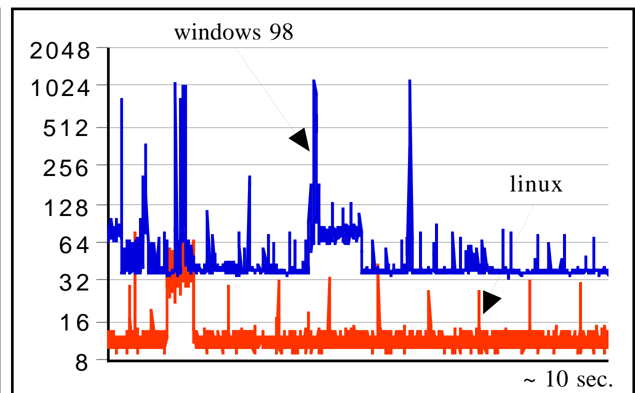

figure 12: a single client with ftp session

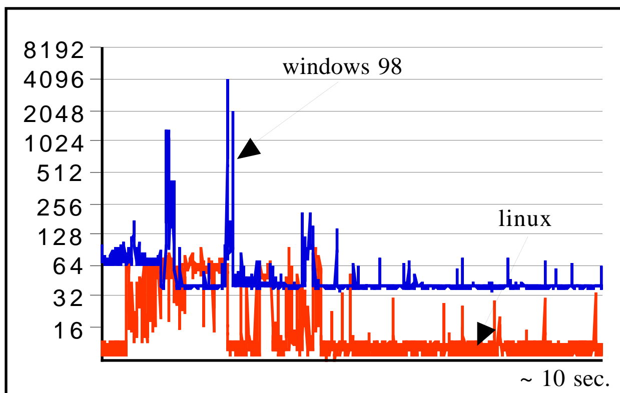

figure 13: a single client while lauching an application

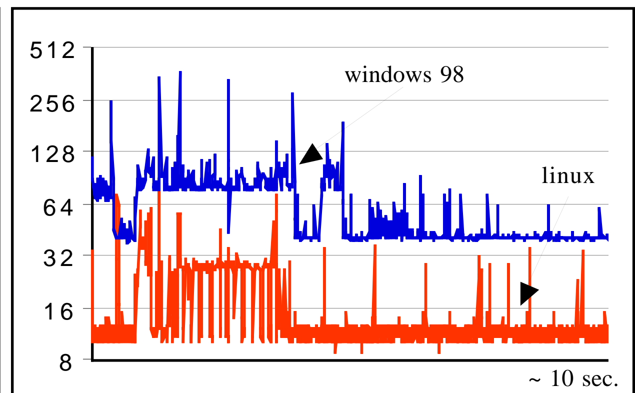

figure 14: a single client while moving a window

Tables 7 and 8 presents the latency characterization corresponding to the figure 11. Table 9 summarizes the system disturbance for the figures 12 to 14.

latency range	width	medium value
linux	5	11
windows 98	15	40

Table 7: single client latency range (fig. 11)

latency peaks	amplitude	frequency
linux	from 25 to 35	low
windows 98	from 55 to 90	medium

Table 8: single client latency peaks (fig. 11)

	max range	max peak
linux	from 25 to 30	96
windows 98	from 80 to 120	> 4000

Table 9: busy system characterization

It's not a surprise: Linux is really better and more stable than Windows 95.

4.1.4 Linux socket vs Darwin sockets and Mach 3.0 IPC

We compared the performances of socket implementation on Linux PPC 2.2.15, Darwin 1.3 and Mach messaging system. The benchmark has been made on the station Mac350.

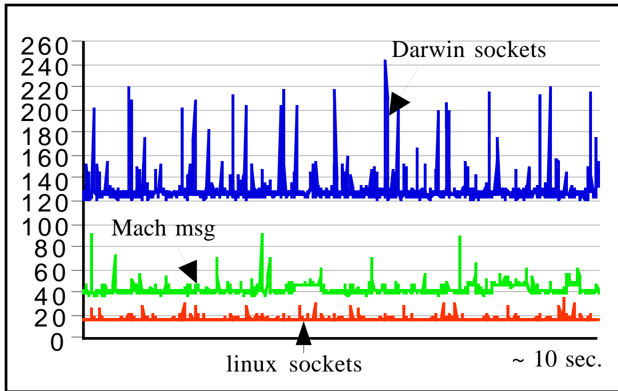

figure 15: a single client benchmark

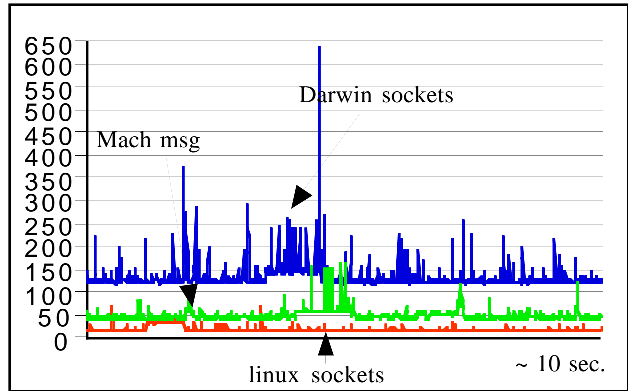

figure 16: a single client with ftp session

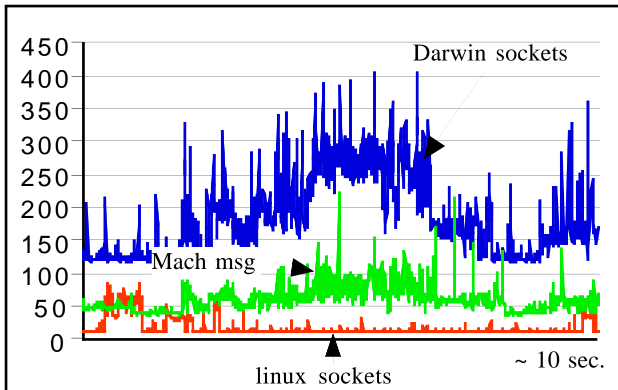

figure 17: a single client while lauching an application

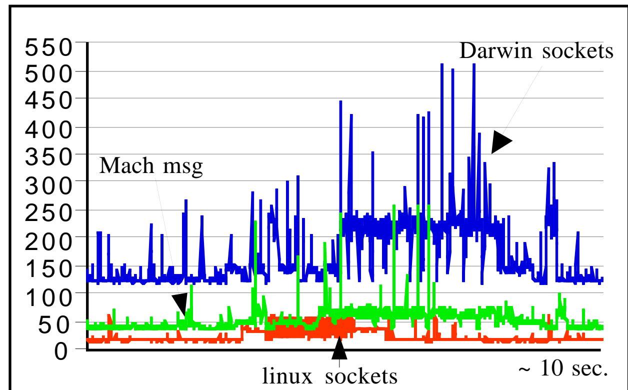

figure 18: a single client while moving a window

Tables 10 and 11 presents the latency characterization corresponding to the figure 15. Table 12 summarizes the system disturbance for the figures 16 to 18.

latency range	width	medium value
Darwin socket	29	131
Linux socket	3	16
Mach msg	11	41

Table 10: single client latency range (fig. 15)

latency peaks	amplitude	frequency
Darwin socket	from 175 to 245	frequent
Linux socket	from 25 to 35	low
Mach msg	from 60 to 90	low

Table 11: single client latency peaks (fig. 15)

	max range	max peak
Darwin socket	from 240 to 300	639
Linux socket	from 40 to 65	90
Mach msg	from 50 to 100	261

Table 12: busy system characterization

It appears that Linux sockets outperform Darwin sockets by a wide margin. Compared to the Mach messaging system, Linux sockets remains 2 times more efficient for a single client.

4.1.5 Windows 98, 2000 and NT 4.0

Here are the compared performances of Windows communication system on different Windows systems. The measurements for 98, 2000 and NT have been respectively made on P700, P350 and P400. The results have been reduced to the clock speed of P700 ie they have been multiplied by 350/700 for P350 and by 400/700 for P400. Timings in figure 20 to 22 are represented in log base 2.

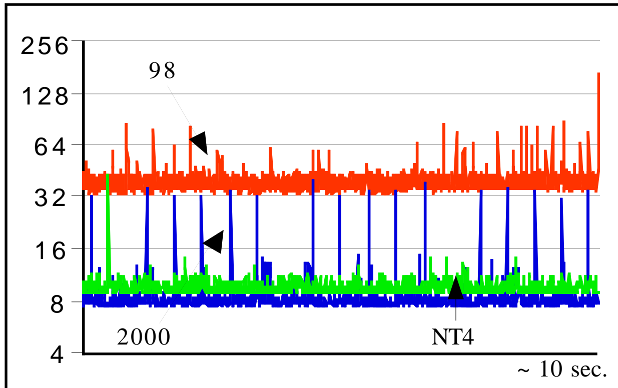

figure 19: a single client benchmark

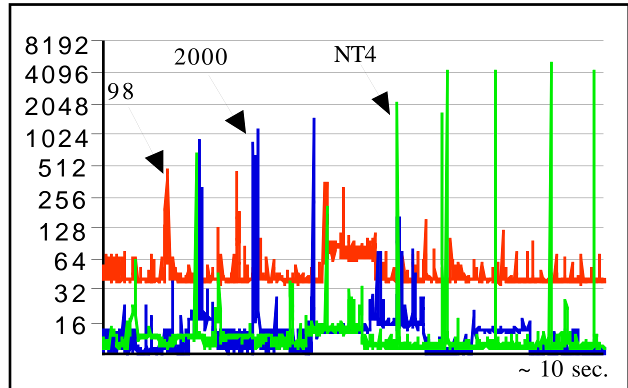

figure 20: a single client with ftp session

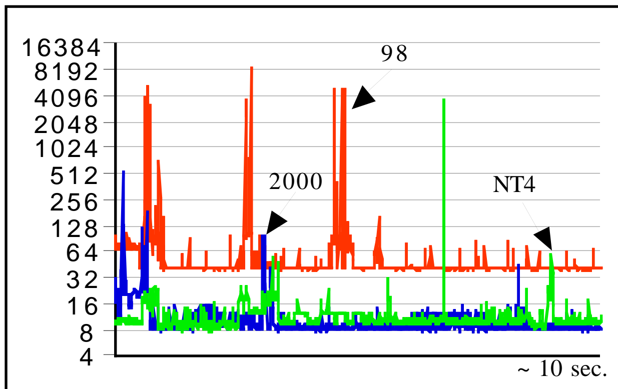

figure 21: a single client while lauching an application

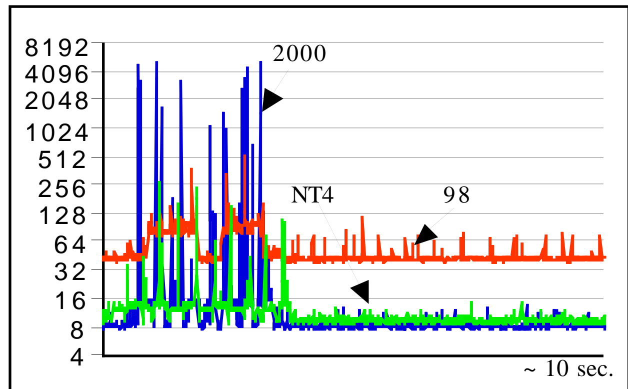

figure 22: a single client while moving a window

The results show equivalent performances for 2000 and NT4. They confirm that the system is not stable in busy environments.

4.2 Multiple clients performances

Figure 23 shows performances measured on linux with 5 clients. It appears that the system behavior when several clients are running, results generally in a stack of latency ranges. Each range may be characterized as previously by its width and its medium value. We'll talk of consecutive clients when the latency ranges of these clients are consecutives. The latency time of the first client is generally spent by the server into the sending messages loop plus a context switch from the server to this first client and the reading of the message. We'll further refer to this time as the *first latency time*.

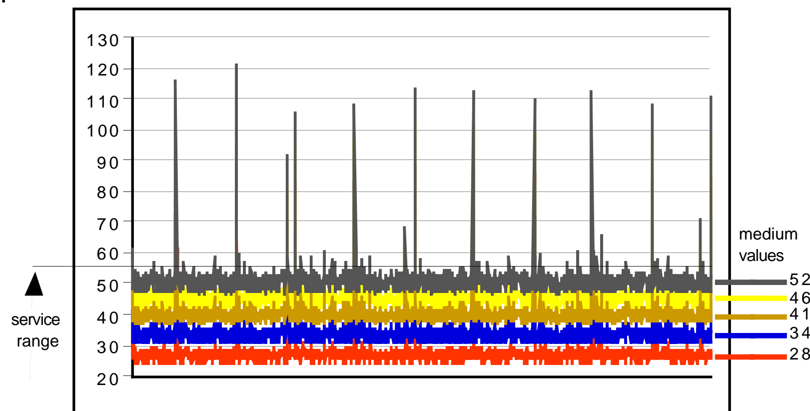

figure 23: a 5 clients benchmark

The time elapsed between 2 consecutive clients is generally spent into the election of a new thread for execution plus a context switch from one client to the next and the reading of the message. We'll further refer to this time as the *next latency time*.
The system behavior when several clients are running may be characterized by the *first* and the *next* latency times. Unless notified, the latency peaks occured according to those reported for a single client. We'll also refer to the *service time* to characterize the last served client: it represents the sum of its medium value and its standard deviation. It may be viewed as the latency range required to serve all the clients.
For the following results, we focused on comparisons between Linux sockets and other systems. Therefore, NT is not mentionned in multiple clients section because it didn't run together with another system.

4.2.1 Linux socket vs Windows 2000

Figure 24 presents the first latency times measured on Linux and Windows 2000 for 1 to 10 concurrent clients. Service times are in figure 25 and next latency times in table 13. As in 4.1.2, measurements have been made on P350.
Windows 2000 is faster than Linux to serve the first client but slower to switch from one client to the next.

	linux	windows 2000
next latency time	12	15

Table 13: *next* latency times

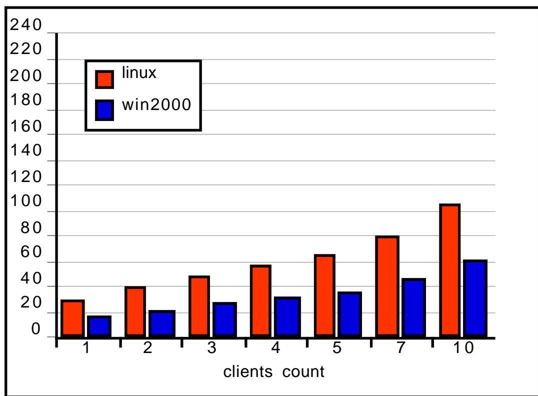

figure 24: first latency times

figure 25: service times

4.2.2 Linux socket vs Windows 98

Figure 26 presents the first latency times measured on Linux and Windows 2000 for 1 to 10 concurrent clients. Service times are in figure 27 and next latency times in table 14. As in 4.1.3, measurements have been made on P700.

	linux	windows 98
next latency time	6	40

Table 14: *next* latency times

Windows 98 bad performances are confirmed with the multi-clients benchs: it takes up to 1/2 millisecond to serve 10 concurrent clients.

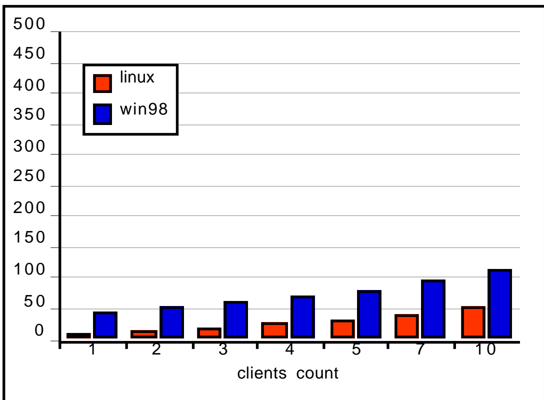

figure 26: first latency times

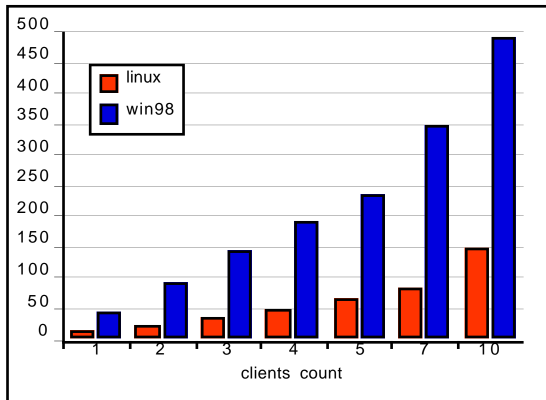

figure 27: service times

4.2.3 Linux socket vs Darwin sockets and Mach 3.0 IPC

Figure 28 presents the first latency times measured for Linux PPC, Darwin and Mach messaging system for 1 to 10 concurrent clients. Service times are in figure 29 and next latency times in table 15. As in 4.1.4, measurements have been made on Mac350.

	linux ppc	darwin	mach
next latency time	10	36	23

Table 15: *next* latency times

Darwin sockets bad performances are widely confirmed when several clients are running.

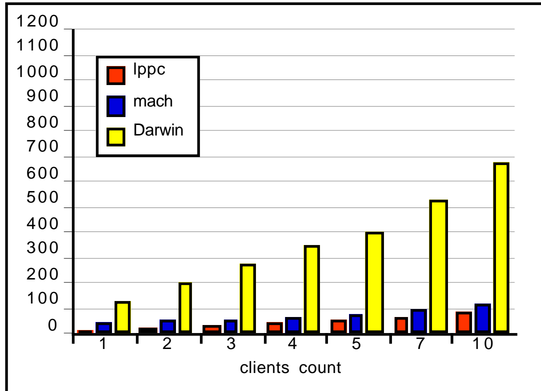

figure 28: first latency times

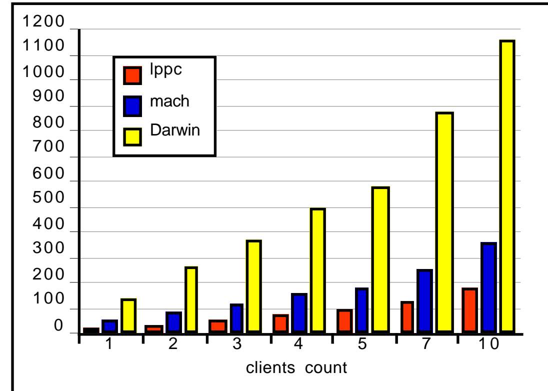

figure 29: service times

Note that as shown in figure 30, the standard deviation of the last client increases significantly for Darwin sockets and Mach messages when the clients count increases.

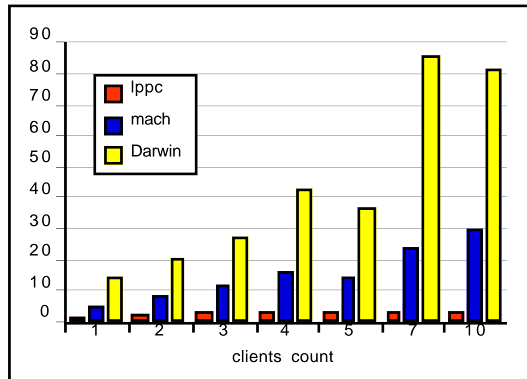

Figure 30: standard deviations for the last client

4.2.4 Linux socket vs kernel module

The Linux kernel module implementation behaves differently than the socket implementation when several clients are running.

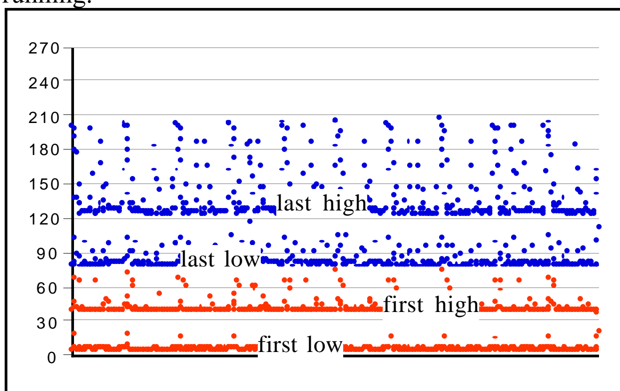

figure 31: 10 clients module first and last latency times

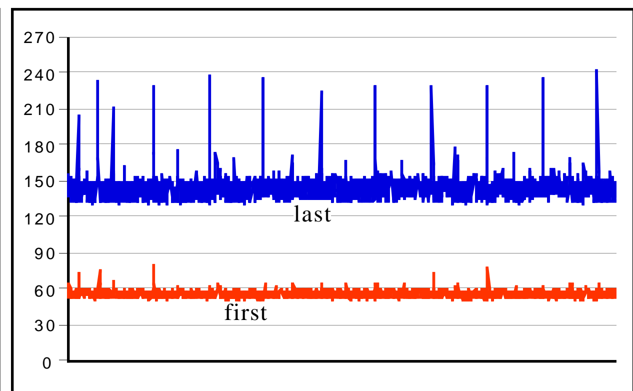

figure 32: 10 clients sockets first and last latency times

As shown in figure 31, the main difference is that the first and last latency times switch between two different levels while in the socket version (figure 32), they remains stable at a unique level. This behavior is unexplained, it may be related to the fact that the different clients are not served each time in the same order, which also prevents us to compute the next latency time. These benchmarks have been made on the same station, P700, running Linux 2.2.15. However, computation of the service times remains correct and comparative performances of socket vs module implementation are presented if figure 33.

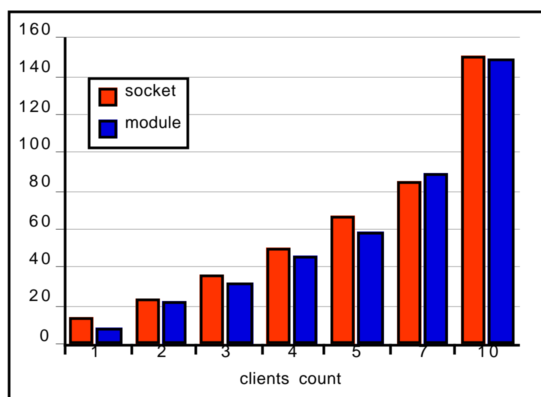

figure 33: service times

While the module implementation is better than the socket implementation for a single client, the difference tends to disappear when the number of clients increases.

5 Results synthesis

The results collected above may be interpreted in term of efficiency. We'll consider a theoritical case, where the server is running a time task every millisecond and needs to communicate with its clients at every tick. This rate corresponds to a realistic requirement of musical applications:

- a sound server using a sampling rate of 44100 Hz and a buffer size of 64 frames should provide samples to its clients every 1,45 ms,
- a 1 ms resolution is required for events based musical applications such as MIDI applications.

The efficiency of the communication system may be evaluated as the ratio of the service time and the tick duration. It represents in percentage, the CPU time used only to dispatch the messages. Figure 34, 35 and 36 presents these results viewed as CPU usage percentage and measured respectively on P700, P350 and Mac350.

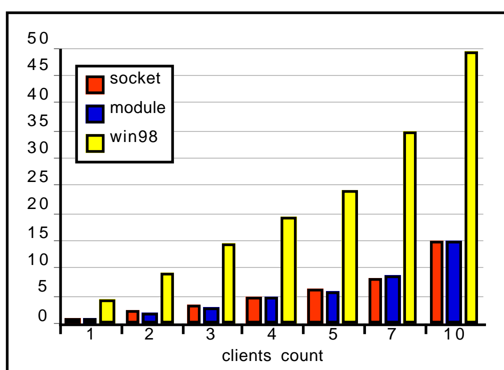

figure 34: compared efficiency of linux sockets, linux module and Windows 98

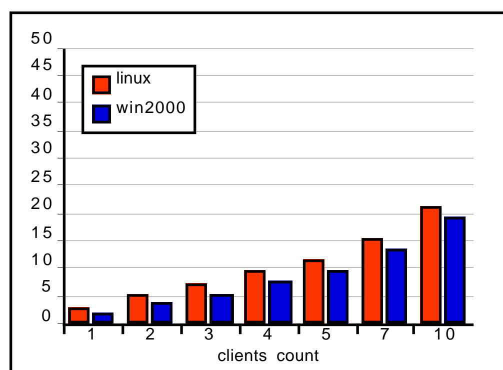

figure 35: compared efficiency of linux sockets and Windows 2000

1. It appears clearly that Linux socket is the best solution, compared to other IPC systems. Windows 2000 (and NT 4) are also very efficient but are too sensitive to perturbations and showed dramatic latency peaks in busy environments. Darwin sockets appear to be unusable in a real-time context: they consume more than 100% of the CPU to handle 10 concurrent clients. Although less efficient than Linux PPC sockets, the Mach messaging system may provide a good solution, considering that it is not very sensitive to busy environments.

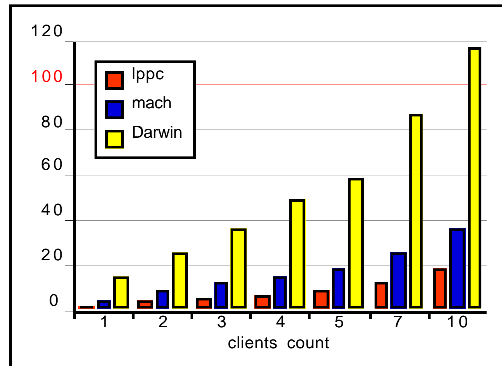

figure 36: compared efficiency of linux ppc sockets, Darwin sockets and Mach messages

Note about the selected results.
The benchmarks were made several times for each system. The collected results represents more than 3000 files. The selected results are the most representative of each system behavior. As previously mentionned, in order to reflect real world behavior, the different systems was running standard configurations and in particular, network was up. However, it appears that the measurements are very sensitive to the system context: for example, being connected to a samba server while running the benchmark may significantly change the results. Although the standard behavior remains globally constant, these changes may affect dramatically the peak latency values which may then reach 1 to several milliseconds.

Threads management - File: threads.c

```
#include <errno.h>
#include <pthread.h>
#include <stdio.h>

#include "ipc_bench.h"

#ifdef FIFO
#define RTSCHED SCHED_FIFO
#else
#define RTSCHED SCHED_RR
#endif


//_____________________________________________________________
pthread_t create_thread (int priority, threadProcPtr proc)
{
	pthread_t thread;
	int  ret = pthread_create(&thread, NULL, proc, 0);
	if (!ret) {
		struct sched_param param;
		param.sched_priority = priority;
		if (pthread_setschedparam(thread, RTSCHED,  &param))
			fprintf (stderr, "no real-time thread\n");
		return thread;
	}
	else fprintf (stderr, "server pthread_create failed: (%s)\n", strerror (errno));
	return 0;
}

//_____________________________________________________________
void set_cancel ()
{
	int old;
	pthread_setcancelstate (PTHREAD_CANCEL_ENABLE, &old);
	pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS, &old);
}

//_____________________________________________________________
void stopThread (pthread_t thread)
{
	void *threadRet;
	if (thread) {
		pthread_cancel (thread);
		pthread_join (thread, &threadRet);
	}
}
```

Server implementation : server.c

```
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/un.h>

#include "ipc_bench.h"

#define non_blocking_send
typedef struct client Client, *ClientPtr;
struct client {
	ClientPtr next;
	ClientAddr adr;
};

ClientPtr gClientList = 0;
int gSock = 0;
int gSnd = 0;
#ifdef non_blocking_send
#define sndPath		"/tmp/tsockSend"
#endif

//___________________________________________________________
static void fatalerror (char *msg)
{
	fprintf (stderr, "Fatal error: %s\n", msg);
	exit (1);
}

//___________________________________________________________
static void unlink_path (int status, void *arg)
{
	char *path = (char *) arg;
	if (unlink (path)) perror ("unlink");
	free (arg);
}

//___________________________________________________________
static int create_socket (char *path, int block)
{
	ClientAddr addr;
	int i, s;
	char msg[512];

	s = socket (AF_UNIX, SOCK_DGRAM, 0);
	if (s < 0) fatalerror ("cannot initialize socket");

	addr.sun_family = AF_UNIX;
	sprintf (addr.sun_path, path);

	on_exit (unlink_path, (void *) strdup (addr.sun_path));

	if (bind (s, (struct sockaddr *) &addr, sizeof (addr)) < 0) {
```

```
		sprintf (msg, "cannot bind server to socket (%s)", strerror (errno));
		close (s);
		fatalerror  (msg);
	}
	chmod(path, S_IRUSR+S_IWUSR+S_IRGRP+S_IWGRP+S_IROTH+S_IWOTH);
	if (block) {
		if (fcntl (s, F_SETFL, O_NONBLOCK) == -1)
			perror ("set non-blocking");
	}
	return s;
}

//_____________________________________________________________
static ClientPtr find_client (ClientAddr *addr, ClientPtr * prev)
{
	ClientPtr c = gClientList;
	*prev = 0;
	while (c) {
		if (!strcmp(addr->sun_path, c->adr.sun_path)) {
			return c;
		}
		*prev = c;
		c = c->next;
	}
	return 0;
}

//_____________________________________________________________
static void open_client (ClientAddr *addr)
{
	ClientPtr prev, c = find_client (addr, &prev);
	if (c) {
		fprintf (stderr, "server: open_client: still opened\n");
	}
	else {
		c = (ClientPtr)malloc (sizeof(Client));
		if (!c) {
			fprintf (stderr, "server: open_client: malloc failed\n");
			return;
		}
		fprintf (stdout, "   open client %s\n", addr->sun_path);
		bcopy (addr, &c->adr, sizeof(ClientAddr));
		c->next = gClientList;
		gClientList = c;
	}
}

//_____________________________________________________________
static void close_client (ClientAddr *addr)
{
	ClientPtr prev, c = find_client (addr, &prev);
	if (c) {
		fprintf (stdout, "   close client %s\n", addr->sun_path);
		if (prev) prev->next = c->next;
		else gClientList = c->next;
		free (c);
	}
}
```

```
//______________________________________________________________
static void rcvmsg (int n, TimeMsg * msg, ClientAddr *addr)
{
	switch (msg->type) {
		case kOpenType:		open_client (addr);
			break;
		case kCloseType:	close_client (addr);
			break;
		default:
			fprintf (stderr, "unknown msg received: (%d)\n", msg->type);
	}
}

//______________________________________________________________
static void * sok_listen (void * ptr)
{
	ClientAddr from; short len;
	TimeMsg msg;
	int n, addr_len;

	set_cancel ();
	while (1) {
		addr_len = sizeof(from);
		n = recvfrom (gSock, &msg, sizeof(msg), 0, (struct sockaddr *)&from, &addr_len);
		if (n == -1) {
			fprintf (stderr, "error rcvfrom: (%s)\n", strerror (errno));
			break;
		}
		else rcvmsg (n, &msg, &from);
	}
	return 0;
}

//______________________________________________________________
static void * sok_send (void * ptr)
{
	ClientAddr from; short len;
	TimeMsg msg;
	int n, addr_len;

	set_cancel ();
	while (1) {
		ClientPtr next, c = gClientList;
		msg.type = kTimeType;
		gettimeofday (&msg.t, 0);
		while (c) {
			n = sendto(gSnd, &msg, sizeof(msg), 0, (struct sockaddr *)&c->adr,
				sizeof(ClientAddr));
			next = c->next;
			if (n == -1) {
				close_client (&c->adr);
			}
			c = next;
		}
		usleep (10000);
	}
	return 0;
}
```

```
//____________________________________________________________
static void wait_first ()
{
	fprintf (stdout, "   waiting for first client\n");
	while (gClientList == 0)
		usleep (100000);
}

//____________________________________________________________
static void wait_end ()
{
	fprintf (stdout, "   server is running\n");
	while (gClientList)
		usleep (100000);
}

//____________________________________________________________
main (int argc, char *argv[])
{
	pthread_t threadIn, threadOut;

	fprintf (stdout, "Server socket bench test\n");
	gSock = create_socket (sockPath, 0);
#ifdef non_blocking_send
	gSnd = create_socket (sndPath, 1);
#else
	gSnd = gSock;
#endif
	threadIn = create_thread (97, sok_listen);
	if (threadIn) wait_first ();
	threadOut = create_thread (99, sok_send);
	if (threadOut) wait_end ();
	if (gSock) close(gSock);
	stopThread (threadIn);
	stopThread (threadOut);
	fprintf (stdout, "done\n");
	return 0;
}
```

Client implementation: client.c

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/un.h>

#include "ipc_bench.h"


#define kMaxValues   1100
#define kIgnoreValues 100

int gSock = 0;
long gCount=0;
long gTable[kMaxValues];

//_______________________________________________________________
static void fatalerror (char *msg)
{
	fprintf (stderr, "Fatal error: %s\n", msg);
	exit (1);
}

//_______________________________________________________________
static void unlink_path (int status, void *arg)
{
	char *path = (char *) arg;
	unlink (path);
	free (arg);
}

//_______________________________________________________________
static void create_socket ()
{
	struct sockaddr_un addr;
	int i, s;
	char msg[512];

	s = socket (AF_UNIX, SOCK_DGRAM, 0);
	if (s < 0) fatalerror ("cannot initialize socket");

	addr.sun_family = AF_UNIX;
	for (i = 0; i < 999; i++) {
		snprintf (addr.sun_path, sizeof (addr.sun_path) - 1, "/tmp/tclient_%d", i);
		if (access (addr.sun_path, F_OK) != 0) {
			break;
		}
	}

	if (i == 999) {
		close (s);
		fatalerror ("all possible server socket names in use!!!");
	}

	on_exit (unlink_path, (void *) strdup (addr.sun_path));
```

```
	if (bind (s, (struct sockaddr *) &addr, sizeof (addr)) < 0) {
		sprintf (msg, "cannot bind server to socket (%s)", strerror (errno));
		close (s);
		fatalerror (msg);
	}
	gSock = s;
}

//_____________________________________________________________
static long elapsed (struct timeval *t1, struct timeval *t2)
{
	long sec = t1->tv_sec - t2->tv_sec;
	long usec = t1->tv_usec - t2->tv_usec;
	return sec ? (sec * 1000000) + usec : usec;
}

//_____________________________________________________________
static void rcvmsg (int n, TimeMsg * msg)
{
	struct timeval t; long d;

	gettimeofday (&t, 0);
	d = elapsed(&t, &msg->t);
	if (gCount < kMaxValues) {
		gTable[gCount] = d;
		gCount++;
	}
}

//_____________________________________________________________
static int print_result ()
{
	long i, d, min=0xffff, max=0, sum=0, total;
	if (!gCount) return 0;
	for (i=kIgnoreValues; i < gCount; i++) {
		d = gTable[i];
		if (d < min) min = d;
		else if (d > max) max = d;
		sum += d;
		fprintf (stdout, "%ld\n", d);
	}
	total = gCount - kIgnoreValues;
	fprintf (stderr, "Transmission summary:\n");
	fprintf (stderr, "    total msg received: %d\n", total);
	fprintf (stderr, "    min time: %d\n", min);
	fprintf (stderr, "    max time: %d\n", max);
	fprintf (stderr, "    average: %ld\n", sum / total);
	return 1;
}

//_____________________________________________________________
static void * sok_loop (void * ptr)
{
	ClientAddr from;
	TimeMsg msg;
	int n, addr_len = sizeof(from);

	set_cancel ();
```

```
	while (gCount < kMaxValues) {
		addr_len = sizeof(struct sockaddr);
		n = recvfrom (gSock, &msg, sizeof(msg), 0, (struct sockaddr *)&from, &addr_len);
		if (n == -1) {
			fprintf (stderr, "error rcvfrom: (%s)\n", strerror (errno));
			break;
		}
		else if (msg.type == kTimeType) rcvmsg (n, &msg);
		else fprintf (stderr, "unexpected msg (type %d)\n", msg.type);
	}
	return 0;
}

//____________________________________________________________
static int send_msg (int type)
{
	TimeMsg msg;
	struct sockaddr_un adr;
	int n;

	msg.type = type;
	adr.sun_family = AF_UNIX;
	sprintf (adr.sun_path, sockPath);
	n = sendto(gSock, &msg, sizeof(msg), 0, (struct sockaddr *)&adr, sizeof(struct sockaddr_un));
	if (n == -1) {
		perror ("sendto");
		return 0;
	}
	return 1;
}

//____________________________________________________________
main (int argc, char *argv[])
{
	void *threadRet; pthread_t thread;

	fprintf (stderr, "Client socket bench\n");
	create_socket ();
	if (!send_msg (kOpenType)) {
		close(gSock);
		return 1;
	}
	thread = create_thread (98, sok_loop);
	if (thread) {
		fprintf (stderr, "   receiving messages...\n");
		pthread_join (thread, &threadRet);
	}
	send_msg (kCloseType);
	close(gSock);
	if (!print_result()) fprintf (stderr, "   done\n");
	return 0;
}
```

Source code: misc

Header file: ipc_bench.h

```
#ifndef __ipc_bench__
#define __ipc_bench__

#include <pthread.h>
#include <sys/time.h>

#define sockPath        "/tmp/tsockServer"
enum { kOpenType=1, kCloseType, kTimeType };

typedef struct sockaddr_un ClientAddr, *ClientAddrPtr;
typedef struct msg {
        int                     type;
        struct timeval t;
} TimeMsg;

typedef void * ( * threadProcPtr) (void * ptr);

pthread_t create_thread (int priority, threadProcPtr proc);
void stopThread (pthread_t thread);
void set_cancel ();

#endif
```

Makefile

```
LIB = -lpthread
OBJS = server.o threads.o
OBJC = client.o threads.o

all : client server

client : $(OBJC)  ipc_bench.h
        gcc $(OBJC) $(LIB) -o client

server : $(OBJS)  ipc_bench.h
        gcc $(OBJS) $(LIB) -o server

clean :
         rm -f *.o client server
```

Source code: misc

Bench script

```
#!/bin/sh
#
# bench script
#

if [ $# != 1 ]
then
	echo "usage: bench <dir name>"
	echo "		where 'dir name' is the results output directory"
	exit 1
fi

dir=$1
[ -d $dir ] || mkdir $dir

function makebench () {
	local count=$1
	local a=1
	local out=$3/$count-tasks
	[ -d $out ] || mkdir $out
	server > /dev/null &
	echo "  $count concurrent clients"
	while [ $a -lt $count ]
	do
		$2 >$out/$a.out 2>$out/$a.sum &
		a=$(($a + 1))
	done
	$2 >$out/$a.out 2>$out/$a.sum
	sleep 2
}

function makemulti () {
	local n=$2
	local a=$1
	while [ $a -le $n ]
	do
		makebench $a $3 $4
		a=$(($a + 1))
	done
}

function makebusy () {
	server  > /dev/null &
	$1 >$out/$3.out 2>$out/$3.sum &
	sleep 3
	bbusy $3
	echo -n "hit return when done"
	read
}

makebench 1 client $dir
makemulti 2 5 client $dir
makebench 7 client $dir
makebench 10 client $dir
```