

The Manifold Server Interface

Documentation and User's Guide

- Abstract -

This document describes an application program interface (API), which may be used as a reduced-cost-and-complexity approach to converting a single-system server into a multi-computer architecture, with fault tolerance through high availability hardware and software support.

I. Introduction

A general discussion of the *load-balancing* problem revealed to us that client-server software could be viewed as a single process that was somewhat arbitrarily split across a pair of networked processors. That is, processing that must be performed on the server became part of the server software module, and processing that must be performed on the client became part of the client software module. Processing that could be done on either side of the inter-process communication boundary (the network) was arbitrarily assigned to one side or the other.

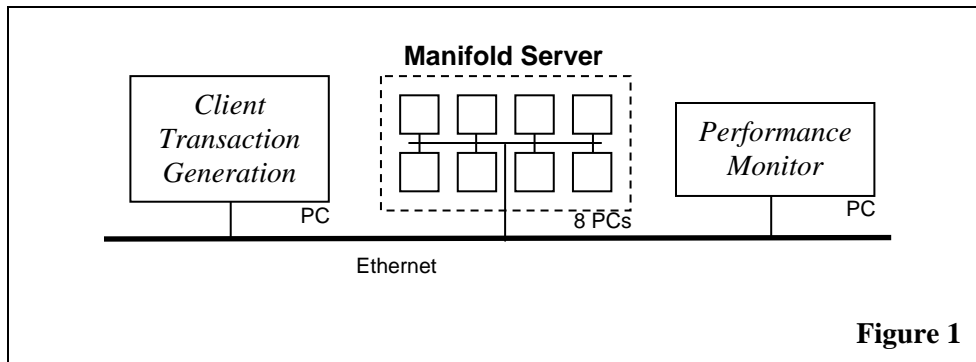
Over time, as the server's concurrent client support requirement grew, and client processors became more powerful, the arbitrary portion of the processing load was shifted away from the server and onto the client. With the proliferation of extremely low powered clients (the Personal Digital Assistant or PDA, and the Internet-ready cellular telephone, for example) this shift has started to reverse.

The embedded processor-as-client world of wireless networking communications led us to rethink exactly how much of this arbitrary load might be shifted back onto the server, especially when the server loading happened to be light. This led us to the following research proposition:

*If the point at which a client issued a remote procedure call (RPC) could be dynamically repositioned over a broad enough range, the client-side software might negotiate that point with the server, effectively shifting a portion of the resource utilization from one processor to the other and back again, according to some set of rules: **adaptive load balancing**.*

We modeled this paradigm as a pair of C-language programs running across two networked processors, then collected, analyzed and presented our preliminary results. What arose from the feedback we got was the potential for such a paradigm in *cluster* computing: distributed applications on transaction processing servers would get the same load balancing benefit as did our client-server software, plus an added high availability (HA) benefit through software and hardware fault tolerance. We extended our concept of replicated clients to that of a replicated, reentrant client-server process on each cluster node. We enhanced our paradigm to choose one of the remote processor nodes at each RPC point, based on local and remote resource utilization (global state) tables, discussed further, below.

We call the resulting cluster architecture, a network of loosely coupled computers, a **Manifold Server**: a collection of computers, each running an exact copy of the original server process, but presenting a single-machine server image to the client (Figure 1). The server process incorporates an API, called the Manifold Server Interface (MSI), designed to provide a low-impact conversion from a single threaded, single system server software into a distributed system. This document describes the MSI, the software conversion process, and some results from our behavioral testing.



A generalized implementation must allow calls and returns from any remote process, to any other remote process. A leader process must receive incoming queries and make the first function call, however, due to the nature of an IP-based network. The leader process must also be the target of the final call, sending the response back to the client. Any function calls in between, however, should be directed to the least busy remote process in order to achieve proper load balancing in a cluster; ORB-based systems can and do provide this.

Proponents of an ORB-based system focus on the B-word: the broker. “You no longer have to concern yourself with where a function (method) is,” they say, “because the broker will find it for you, and will even instantiate one for you if necessary.” Opponents of an ORB-based system focus on the B-word too, emphasizing what any practitioner will confirm: it is the inter-process communication – the network traffic – that limits the performance of a distributed system.

Beowulf systems, Networks of Workstations (NOWs), PC Farms, Distributed Shared Virtual Memory Systems, etc., each discovered that performance came down to a single metric: the ratio of the time it takes to process a packet of data (or execute a packet of instructions), to the time it takes to deliver it to (and from) some remote node. Faster delivery times meant better performance, in every case. In an age of cheap, fast memory chips and comparatively slow external I/O pathways, it remains a mystery why anyone would design a system as heavily dependent upon inter-process communications as an ORB!

The requirements for the new architecture are the same as those of the old. You are expected to have already developed a concurrent server process that is designed to run on a single system server. The interface is the same too, with expanded responsibilities for MSI_init that include a global data area. There is one new function named MSI_send that is called with a response for the client. MSI_init is still charged with building a table of your internal functions. It will also build a global data area, a structure that contains data common to all of your functions as they prepare a response to the client’s query. MSI_send ensures that the final return from your functions is to the leader process. You still consider the MSI to be a function library that is compiled along with your (monolithic) concurrent server process, with its compiler options set to distributed system mode, using calls to MSI_call that access your functions indirectly.

II. Requirements

Later sections of this document reference these overall system requirements:

1. Scalability

The purpose of this system interface shall be to facilitate the conversion of a single-system server into that of a multi-computer architecture. System performance shall be proportional to

the number of operational nodes in the system, up to some optimum number of nodes at the point of diminishing returns. Our goal is to maximize this optimum number of nodes.

2. Low Impact Software Conversion

An important goal is to minimize the impact of software conversion. ORB technology often requires software developers to completely redesign their systems employing object-oriented techniques and semi-standard function calls. We assume the original concurrent single system server to be based on an ordinary monolithic architecture. The original server software must be modified to (1) inform the MSI of the nature of each of its local functions and its persistent global data, and (2) call the MSI with reference to each function at the point where the function is to be called. Generalized return from RPC requires a new MSI_send function to ensure the final response packet is sent from the lead system, the one with the IP address known to its external clients. Recompile the modified server software along with the MSI to affect a distributed version of the server, a Manifold Server.

The MSI, aware of each local function, will decide at each point of call whether to continue with a local function call, or to remotely call the same function on one of the other nodes in the cluster via a light-weight RPC. Generalized return from RPC allows a similar decision to be made at the time of return: control is passed to the least-busy node in the cluster.

3. Minimized Operating System Dependence

We will minimize the impact of a new OS release to our system by using standard function and system calls. Changes in any new release of the OS should at most require a recompilation of our system's source code.

III. High-Level Design

All programming was done in the C language, running in a Linux operating system (OS) environment. Our laboratory multi-computer comprises eight nearly identical PCs, connected via an Ethernet network. The software is instrumented for near real-time performance monitoring under varying arrival rate distributions and transaction load levels. We discuss all this in more detail below.

One of our goals is to investigate MSI's potential as a lightweight replacement for an ORB-based infrastructure on an embedded multiprocessor architecture where maximum throughput with a minimum of overhead is of the utmost importance.

In addition to the MSI, we designed a client transaction generator and performance display monitor, each of which runs on a processor that is external to the cluster.

The transaction generator may be configured to simulate clients sending randomly spaced queries in an exponentially distributed stream, in a *sweep*, or *pulse* pattern.

The performance monitor displays all eight cluster processors' resource utilization in near real-time. (See Figure 1.)

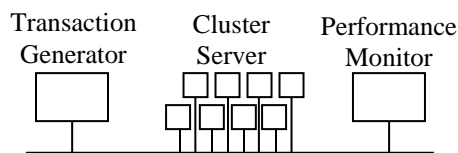


Figure 1: Cluster Server Test Bed

Each of the processor nodes runs a replica of the single-system server software, modified and recompiled with the MSI for inter-processor system communications support. Each processor has a number of performance monitor subtasks associated with it, for CPU and memory utilization. Each processor task keeps its own local statistics and passes them along with every outgoing message. It keeps a global view of the cluster nodes from remote data contained in incoming messages. It keeps a local as well as a global state table of resource utilization for every remote processor in the cluster.

Test Application:

We designed a single system, concurrent server for demonstration purposes. A sophisticated concurrent server might accept an incoming query, spawn a task to service it, and the internal processing might involve several function calls: decryption, decompression, error correction, translation, etc., before sending a response. Our simple system employs a client that generates random 3-word strings representing a mathematical expression, like “three plus two”. The first and last words are always numbers and must range from zero to nine. The middle word is an operation code and must be chosen from {plus, minus, times, by}. The string is processed by five function calls:

- | | | |
|---|-----------|---|
| 1 | parse() | breaks the incoming string into its three component words, |
| 2 | numbers() | converts the first number word to an integer, |
| 3 | opcodes() | converts an operation word into a symbol, |
| 4 | numbers() | converts the second number word into an integer, |
| 5 | solve() | returns the integer result of the operation on the two numbers. |

The result is *printed* into a buffer and sent back to the client, completing the service.

IV. Detailed Design

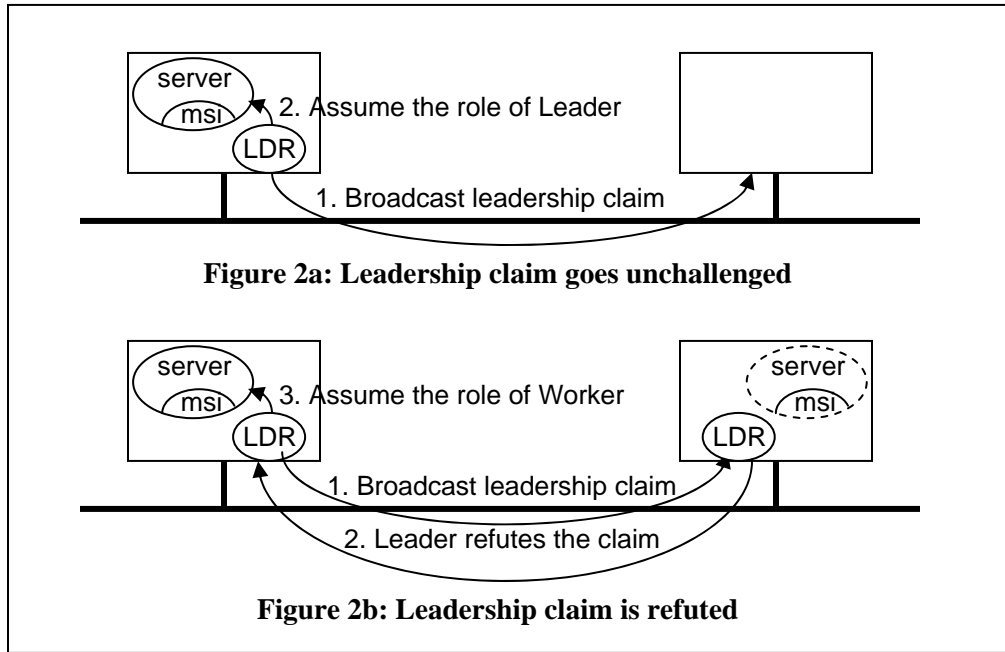
Initialization:

MSI_init is called during initialization, as many times as the server has local functions. Each call to MSI_init allows it to establish the name, local address, and passing parameter format of each local function. MSI_init is called again, for as many times as there are persistent global data elements, in support of generalized return from RPC. A final call to MSI_init passes a NULL parameter to signal an end to the server’s initialization.

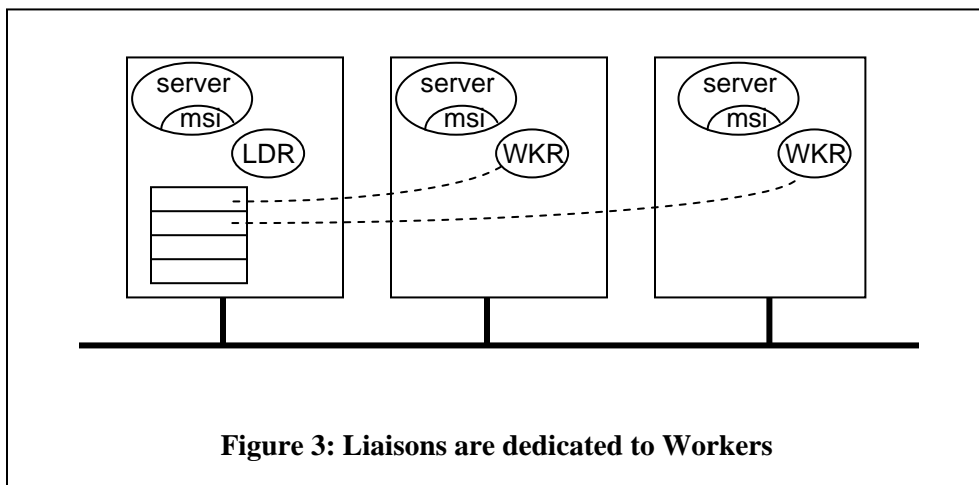
When MSI_init is called for the last time, it builds a table of function names, addresses, and parameter characteristics, as it did in the preliminary version. It also builds a common data structure with each of the members that you defined in the sequence of calls. Finally, it starts a manager subtask that broadcasts a claim to Leadership mode to any other Leadership task that may be running in the same cluster.

- If this is the first Leadership task to execute on the cluster its claim will go unchallenged and it will signal the parent task that it is in fact the Leader.
- If there already is a Leadership task running, it will reply to this claim, and the claimant will signal the parent task, the local server, to run in Worker mode.

Figure 2a diagrams the case where a Leadership claim goes unchallenged and Figure 2b shows the case where there is a response.



As each subsequent node in the cluster initializes, its server's claim to Leadership will be refuted, and it will revert to Worker mode. Note that as these claims are received by the Leader (the server on the first node to initialize) it updates its table of Worker nodes, as shown in Figure 3.



Operation:

As each client query comes in to the Lead server, it spawns a subtask to receive it. (This is how a concurrent server handles incoming query packets, by definition.) MSI_call knows that it is in Leadership mode. It builds the global data area and marshals passing parameters needed to call

your first internal function, sends it to the least busy remote Worker node, according to its table of remote Workers, and waits for a response. All packet data transfer is via UDP.

Remote Worker tasks read incoming packets directly into a shared memory area where their local MSI process can find it. Once detected, the MSI spawns a subtask that initializes the global data area and makes a local call to the specified internal function. On return, it updates the global data area, marshals passing parameters needed to call your next internal function, and sends it to the least busy remote Worker node, according to a local copy of the table of remote Workers, and the MSI subtask exits. The local Worker task waits for another incoming packet while this local call is being processed. (The table of remote Workers represents distributed, shared, global data. Its management will be addressed later in this report.) Figure 4 shows a Worker node receiving a remote procedure call (RPC) packet, processing it via the MSI, and sending a subsequent request to another remote Worker.

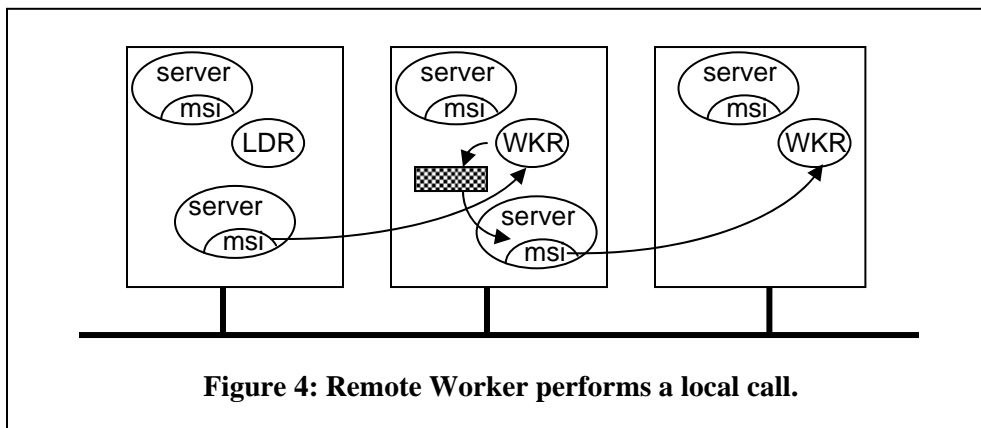


Figure 4: Remote Worker performs a local call.

The last call is always to MSI_send. (This happens when the original query is processed and a response is ready to be sent back to the client.) It sends the result not to the client, but back to the Lead server subtask that is still waiting for it. That subtask sends the result to the client and exits.

We expect that in the unlikely event a packet is created and sent to some remote server each time the next function is called the performance will approximate that of an ORB-based system. We expect that the remote Worker node will discover that it is the least busy node most of the time, resulting in an overall fewer number of RPC requests on average, allowing the MSI to outperform an ORB-based system.

Termination:

This is a test bed system. We built in a special query packet that signals the end of testing. The Lead node forwards this request to each of the remote Worker nodes, cleans up its subtasks and shared memory areas, and exits gracefully. Each Worker node, upon receiving a request to terminate, does likewise.

Table of Remote Workers:

This table is subject to update over time, as the state and loading status of each remote Worker node changes. Updates to a distributed table of shared global data must be handled in exactly the

same way as updates to the copies of a distributed database. Each copy of the data is called a replica, and when the data in a replica is altered, that change must be propagated to each of the other known copies of that database. The exact way in which these updates are applied is called a Replication Control Algorithm (RCA). We have access to earlier research where several of these RCA methods were analyzed and compared to one another. We will build an appropriate choice into our final product.

Summary:

We built and analyzed a preliminary test bed system and learned some lessons from it. We apply what we learned to this new design and expect the results to be as we originally projected, that the MSI will outperform an ORB-based system under similar conditions. Our next is to implement the new design and test it under various loads. The versatile client software, the remote performance monitoring system, and much of the source code from the original system can be reused, and we have gained experience from their development. The time to develop and test this new design should therefore be considerably shorter than that of the original.

V. Users Guide

The assumption is that a C language program, a single system, *concurrent* server, already exists. (A concurrent server is one that spawns a dedicated subtask to service each incoming query [1].) It is also assumed that you have access to a small, Linux-based PC cluster on a local area network. Our example shows 5 PCs, but even three will give a sense of distributed processing.

The MSI requires that your C program be modified in three ways: (1) the two MSI function calls must be given prototype statements, (2) the MSI initialization must be called once for each of the original program's functions, and then once as a *final* call, and (3) each of the original program functions must be called indirectly, through the MSI calling function. Examples follow.

The MSI appears to be a pair of functions to the application, the original single-system *concurrent* server. The application calls the first function, named `MSI_init`, as many times as necessary to inform the MSI of all its local functions. `MSI_init` adds each function's name, address, and parameter characteristics to an internal table. The application then calls those local functions indirectly, through the second MSI function, `MSI_call`. `MSI_call` marshals the parameters into a packet and sends it to the Liaison task for a remote procedure call, or RPC. The simplicity of the required source code changes should become evident through the following examples.

Perhaps the best way to handle any required code changes is to use the C language *ifdef* along with a variable that can be defined by compiler flags [2]. Then, if the flag is present, all the code changes are compiled; if not, then the application is compiled without the changes. This greatly simplifies testing and provides an easy-to-use mechanism to withdraw any changes related to the distributed environment.

First, one must provide prototypes for the two MSI functions:

```
#ifdef MSI
void MSI_init();
void MSI_call();
```

```
#endif
```

Second, provide a call to `MSI_init` for each local function, ending with a *final* call:

```
#ifndef MSI
    MSI_init(fun1, "fun1", "0,*80,4");
    MSI_init(fun2, "fun2", "4,*80");
    MSI_init(mystruct.value1, "mystruct.value1", 4);
    MSI_init(mystruct.value2, "mystruct.value2", 4);
    MSI_init(mystruct.string, "mystruct.string", *80);
    MSI_init(NULL);
#endif
```

Note that the application is planning to make calls to a local function named *fun1*, and another named *fun2*. Both the addresses and the character string names of these functions are passed to the MSI via an `MSI_init` call.

The first function has a void return as signified by the first zero in the third passing parameter, the parameter characteristics string. (The first position in the parameter characteristics string is the only position where a zero length is allowed!) The first passing parameter to *fun1* is a pointer (note the “*”) to an area of unknown type, which may be up to 80 bytes long. This typically represents a character string or a structure, but the MSI does not need to know the exact nature of this parameter. The second parameter is a 4-byte value, perhaps an integer but again, this information does not need to be known.

The second function has a 4-byte return value and a single parameter, a pointer to an area of up to 80 bytes.

The NULL function address is a signals to `MSI_init` to finalize its local function table and to continue its initialization processing.

All calls to `MSI_init` must occur prior to any call to `MSI_call` and you must make a final call to `MSI_init` with a NULL address parameter.

`MSI_call` is used instead of a direct call to the local functions:

```
#ifdef MSI
    MSI_call("fun1", NULL, &packet);
#else
    fun1(&packet);
#endif
    . . .

#ifdef MSI
    MSI_call("fun2", &retval, &packet);
#else
    retval = fun2(&packet);
#endif
```

In the above example, *retval* is an integer and *packet* is a local structure. Note the use of *fun2*'s returned value in the parameter list to `MSI_call`, rather than as an assignment in the direct call. Note also the use of an ampersand (i.e., `&retval`) to allow return of the changed value to the calling application.

Finally, call the new MSI function to send a response back to the client. This is similar to the rules used to describe the use of `MSI_call` in prior documentation:

```
#ifdef MSI
```

```

    MSI_send(&packet, sizeof(struct packet));
#else
    send(fd, &packet, sizeof(struct packet), 0);
#endif

```

Once compiled in distributed system mode, your server will call `MSI_init` to define each of your internal functions, each of the members of your internal data structure, and one final time, with a NULL pointer, to signal the end to the sequence of calls.

Compile the MSI (`msi.c` in our example) into an object module, or a function library. Make changes to the source code (`appser.c`) and recompile it with the defined compiler variable, named `MSI` in our example:

```

➤ cc -c msi.c
➤ cc -DMSI -odappser msi.o appser.c

```

The distributed application server will be in the executable named *dappser*, as indicated by the `-o` flag in the compiler command above. Note that the executable is prefixed with a *d* so that it may be distinguished from the single-system executable named *appser*, which is compiled without the `MSI` variable defined:

```

➤ cc -c msi.c
➤ cc -oapp msi.o appser.c

```

This simplifies testing by keeping the single-system executable *appser* separate from *dappser*, the distributed version of the server.

Assuming the path to the executable *dappser* is accessible to each of your cluster processor nodes, through the network file system perhaps, and that one of these node machines has a well-known server IP address, start *dappser* on the machine with the server's IP address first. After a few seconds, start it on each of the other nodes. (The first instance of *dappser* declares itself to be in *leadership* mode, as opposed to *worker* mode, to be described in detail below; it must run on the node with the IP address known to all external clients.)

Demo System:

Let's assume you have access to a small network of five PCs (Figure 5), one to run the transaction generator and the performance monitor, and another four nodes operating together as a cluster server.

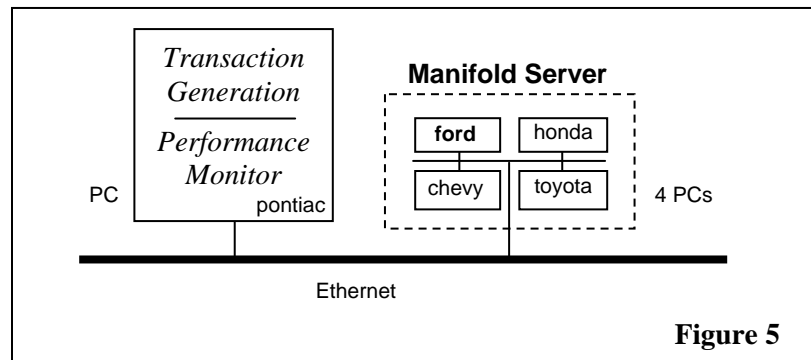


Figure 5

The keyboard and monitor are connected to the network node named *pontiac*, which is also the Network File System (NFS) server for your user's home directory. Start this machine first. Login to *pontiac* and start the other four systems in any order.

Once logged in, you should see a single X window. Open four more X windows and login remotely to one of the other nodes in each of these windows. For example:

➤ `ssh ford`

At this time you will have five windows open on the single display that is physically connected to the node named *pontiac*, one window logged into each of the other four nodes. Start the distributed application server named *dappser* on the multi-computer node named *ford*:

➤ `dappser`

Wait five seconds for it to initialize. Once it discovers its leadership role, start *dappser* on each of the other three nodes. (They will each discover that they are to assume worker roles.) Wait until all four nodes are *ready*.

Start the transaction generator in the original window that was open on *pontiac*:

➤ `appli ford`

Generate a transaction stream by entering a coded message at the *appli* input prompt:

`25 e`

This instructs the application client (*appli*) to generate a stream of 25 query packets, each delayed randomly with an exponentially distributed time delay pattern. Any integer number of packets may be followed by a letter *e* (exponential), *p* (pulse), or *s* (sweep) for the various supported time delay distributions. Note that pulse and sweep are based on a 50-packet stream, described below, and will be most effective when the requested number of packets is some multiple of 50. Here are sample results from *appli*:

```
1, 0.001196, 0.157816
2, 0.001194, 0.310893
. . .
25, 0.001197, 3.08802
```

There are three numbers: the first number is the packet sequence number, the second is the response time for just that packet, and the third number is the elapsed time up to receiving that packet in the stream.

Terminate *appli* by typing `^C` (control-C). Terminate *dappser* on each of the secondary nodes (the workers) in the same way, but issue a command to terminate the *dappser* process:

- `ps aux | grep dappser` [find the process id of your *dappser* process]
- `kill -9 731` [assuming the process id was found to be 731]
- `ipcs` [find the identities of any shared memory areas]
- `ipcrm -m <shmid>` [removes the shared memory area with id *shmid*]

Similarly, the leader node requires you to issue a kill command for every *dappser*, *liaison*, and status monitor before removing shared memory areas. The method of ending these processes is still crude at this writing. This procedure will improve as fault tolerant software features are added and refined.

In each of the remote node processor windows issue *halt* and wait for a “System halted” message before turning off the power to each of the four machines. Close their windows and issue a halt command in the original window opened to pontiac. Turn off its power after the halt message.

Comments:

The most troubling implementation issue we’ve discovered to date is the seemingly unwarranted release-to-release changes in the Linux *proc* files. (These are near real-time performance statistics used by our performance monitoring routines to adjust the amount of loading between the client and server.) There are three files; a simple list (the Linux *cat* command) should reveal any differences between what they contain and what our performance routines *think* they contain:

> **cat /proc/stat**

```
cpu 1020 1 957 30621
cpu0 1020 1 957 30621
page 89795 26492
swap 1 0
intr 45780 32599 149 0 0 3 0 0 0 1 0 0 1947 570 0 10479 32
disk_io: (3,0):(10447,8092,178314,2355,52960) (3,1):(33,33,195,0,0)
(8,0):(451,451,458,0,0)
ctxt 59915
btime 1219252391
processes 4715
```

The first line is used for CPU utilization: The system clock interrupts Linux 100 times per second so it can look at the *state* of the CPU (central processing unit, or processor). It may be in one of four states: user (ordinary), nice (see the Linux *nice* command), system (operating system), or idle mode. One of these four counters is incremented each time an interrupt occurs, according to the CPU state at that time. We get all four counters, once each second. The difference between any two sets is the exact *percentage* of time that the CPU was in user, nice, system state, or idle, respectively, for that 1-second time period. If your stat file is different, you may have to change *MSI/stat/cpuStat.c*.

The line beginning *disk_io:* is used to calculate disk input-output activity. The first device, labeled (3, 0), is associated with a parenthesized list of five numbers. The first is the sum of the second and fourth. The second number is the number of *read* input-output operations, and the fourth is the number of *writes*. (The third and fifth numbers are the number of *blocks* of data read and written, respectively. We use operations rather than data blocks: see */MSI/stat/dskStat.c*.)

> **cat /proc/meminfo**

```
total:    used:    free:    shared: buffers:  cached:
Mem:  517865472 197529600 320335872          0 13848576 81739776
Swap: 534118400          0 534118400
MemTotal:          505728 kB
MemFree:           312828 kB
MemShared:         0 kB
Buffers:           13524 kB
```

```

Cached:          79824 kB
SwapCached:      0 kB
Active:          152704 kB
ActiveAnon:      66404 kB
ActiveCache:     86300 kB
Inact_dirty:     612 kB
Inact_laundry:   0 kB
Inact_clean:     6436 kB
Inact_target:    31948 kB
HighTotal:       0 kB
HighFree:        0 kB
LowTotal:        505728 kB
LowFree:         312828 kB
SwapTotal:       521600 kB
SwapFree:        521600 kB

```

Only the second line is used. (The column headers are skipped.) Dividing the memory *used* by the *total* memory available (times 100%) yields the percentage of real (physical) memory in use. See /MSI/stat/memStat.c.

> **cat /proc/net/dev**

```

Inter-|   Receive                               |   Transmit
face |bytes  packets errs drop fifo frame compressed multicast|bytes  packets
errs drop fifo colls carrier compressed
   lo:  70511    1027    0    0    0    0    0    0    0    70511    1027
0      0      0      0      0    0    0
   eth0:  2523     39    0    0    0    0    0    0    0    2993     41
0      0      0      0      0    0

```

The line beginning *eth0* is of interest; specifically, the first and ninth numeric values, which represent the number of *received* and *transmitted* packets, respectively. Reading two sets of these values, one second apart, allows us to calculate the number of packet received and transmitted per second during that time period, by the network interface card. See /MSI/stat/netStat.c.

If you cannot easily interpret the format of your files, try looking at the kernel source files under

<linux_kernel_source>/fs/proc/ Where <linux_kernel_source> is often /usr/src/linux/.

Finally, this leader-plus-servers software architecture is less than what we originally proposed. This simpler version, referred to as the “rudimentary” API above, is intended to be a *growth medium* for the intended system, allowing us to shake out the cluster hardware and software, and to develop some ancillary utilities, the test load generator, for example. Allowing a function to be called from a remote machine (what we have now) is easy compared to allowing that called function to return to the same software running on some third machine (what we want).

References:

- 1 *Unix Distributed Programming*, C. Brown, Prentice Hall, 1994.
- 2 *The C Programming Language*, 2nd Ed., B. Kernighan and D. Ritchie, Prentice Hall, 1988

VI. Performance Statistics

There are four processors in the *demo* cluster, so there may be 1, 2 or 3 worker nodes servicing the leader node's requests. We sent a stream of ten packets to the system in each of its three configurations, with the following results (Figure 4):

Note that three streams of ten exponentially distributed query packets were sent to the leader, and that this graph represents the overall processing times for each stream, with one, two, and three operational worker nodes, respectively. This graph suggests an improvement in the overall response times as more workers are added to the multi-computer cluster. Improvement cannot continue indefinitely, but the data suggest that an optimum number of workers is greater than or equal to three on this system. Our off-campus lab system has more processors, but they are slower. We expect a better response curve but our tests gave us the following results (Figure 5):

We suspect that the cumulative software overhead in managing the cluster is masking the transaction processing time performance, that the sum of the transaction processing times and the management overhead is growing as more workers are added to the system, giving us greater overall completion times, though the transaction processing time itself is decreasing. We could separate the transaction processing times and produce a better looking graph, but it is the overall transaction processing times that we must focus on if we are to improve the product.

The next logical step in our analysis is to discover where the time is being spent as a single transaction makes its way through the system. We have devised a plan to measure fourteen separate events along this path. (See Figure 6.)

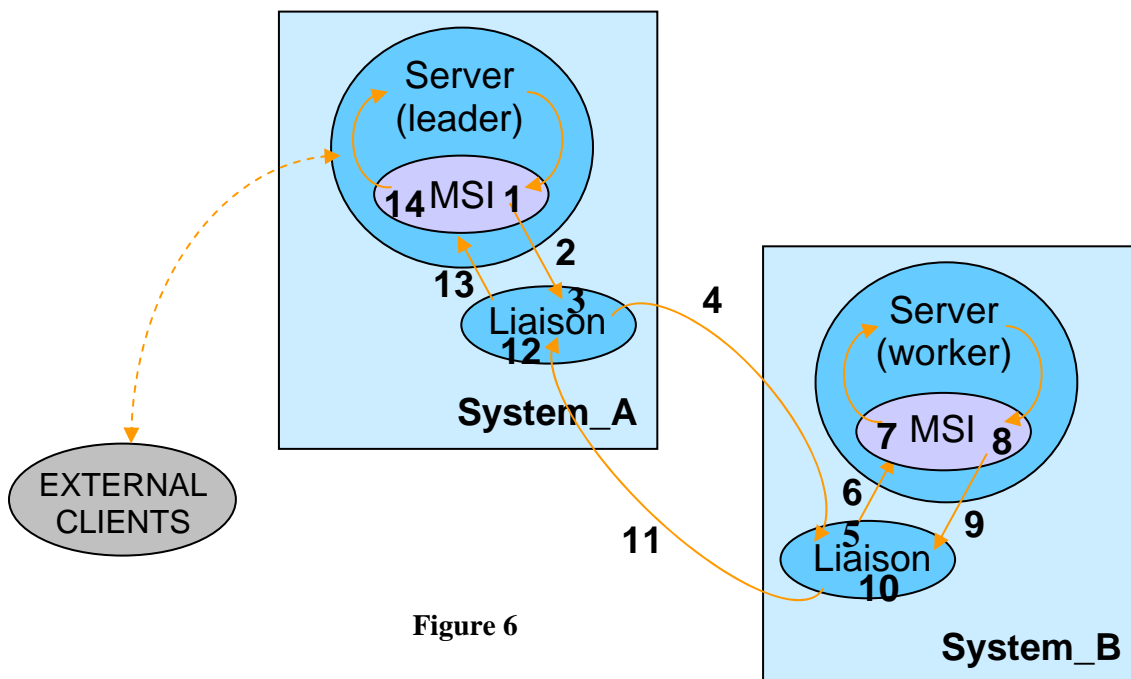
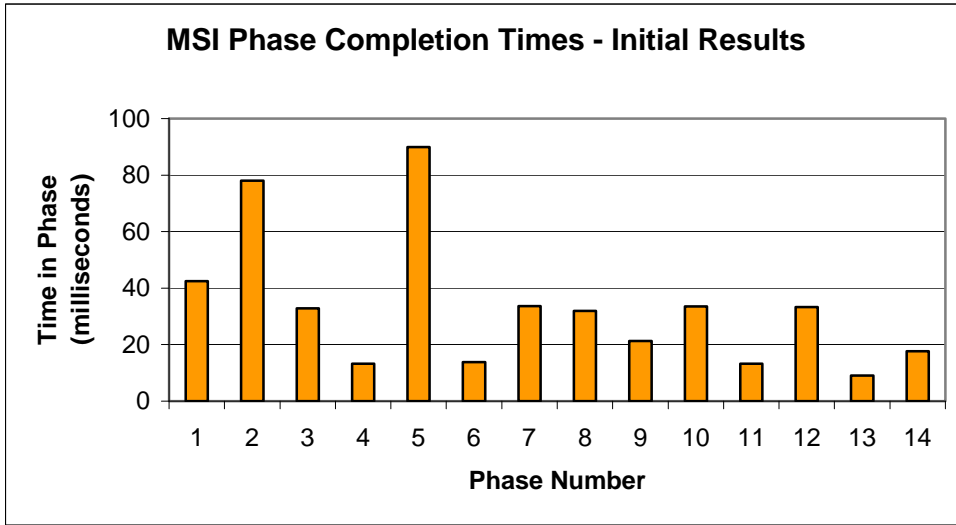


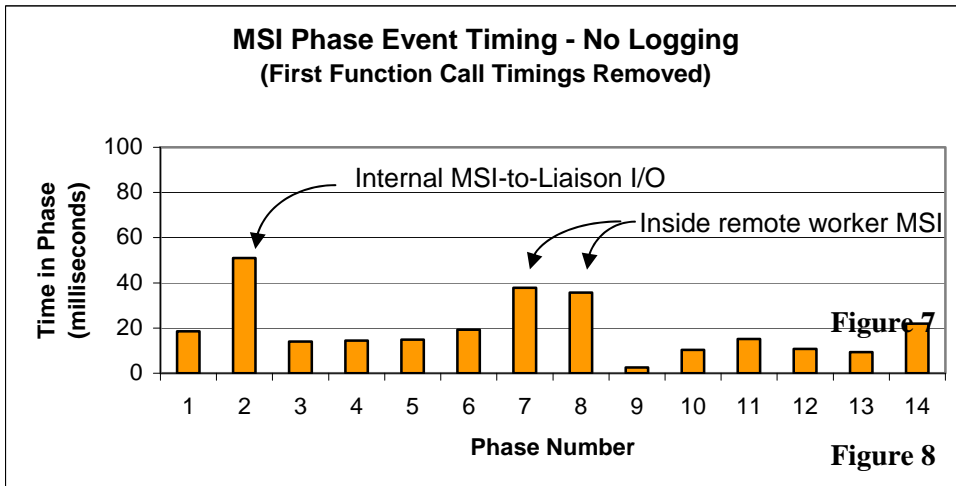
Figure 6

Figure 5

As you can see from the diagram, the actual processing time has been left off because we wish to discover and then try to reduce the overhead times. The results of our timing run are in Figure 7.



An examination of the activities in each of the highest bars showed the results were clouded by two factors, event logging for debugging purposes, and protocol initialization times during the first function calls. (We do five function calls for each incoming transaction, as explained above.) We reran the timing with only the last four functions calls, and event logging disabled, with the following results (Figure 8):



Here we have a clearer picture of where the management overhead time is being spent. There appear to be two avenues of attack regarding performance improvement:

1. The internal MSI-to-Liaison packet transfer will be handled in a shared memory space instead of an internal TCP/IP stream socket, and
2. A close look at the passing parameter packetization and depacketization code is expected to reveal inefficiencies that, when corrected, will improve the overall system performance.

VII. Appendices

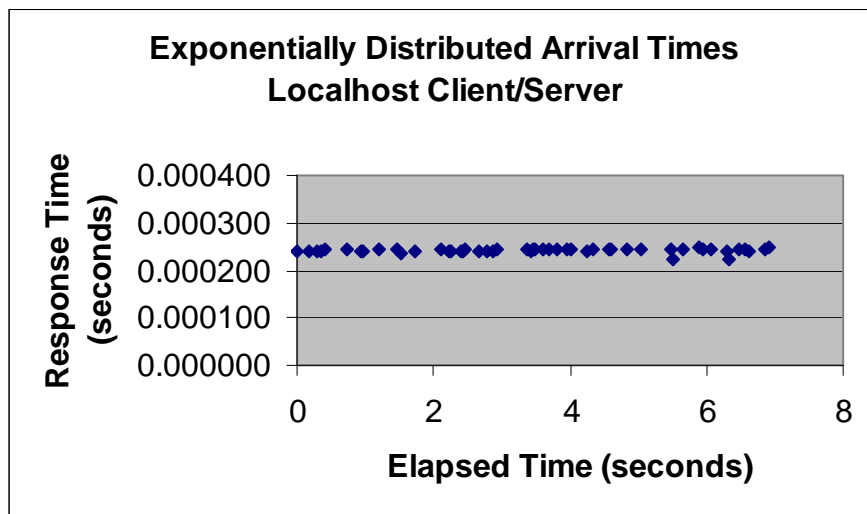
A. Test Load Generator

Performance testing is evaluated over a variety of query arrival rate distributions, which are effected through five delay time distributions. If the time delay between queries is chosen at random, over say an exponential distribution, the arrival rate pattern (at the server) is assumed to be exponential also. A *delay* function is incorporated into the client that allows one of these distributions to be used as a burst of query packets is sent to the server. Two of these distributions, are cyclic (see below) and are based on an arbitrary burst set of fifty packets. (See source module `delay.c` for further details.)

1. Exponential

Each delay time is again chosen as a number of microseconds between zero and 1000000, but the selection is exponentially distributed.

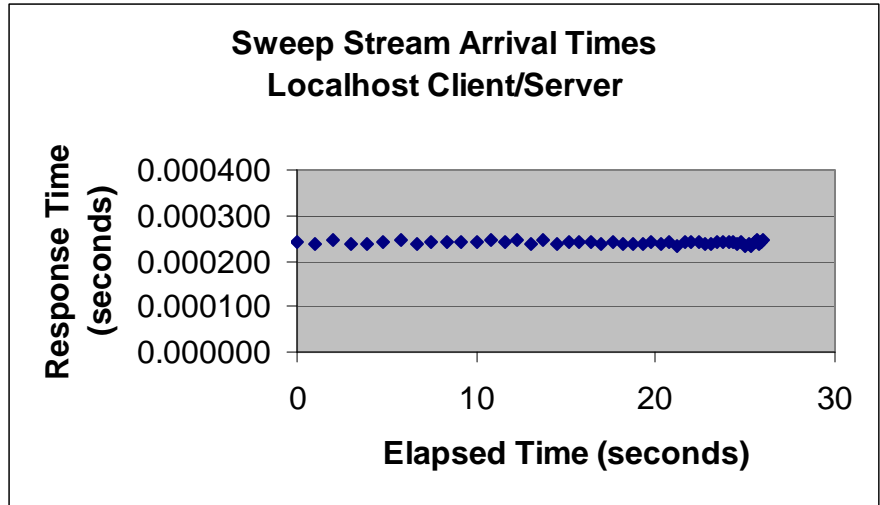
Below is a graph of the time delay distribution and response times, where the client and the server were both on the same physical machine. (These, somewhat trivial data were from early tests of the Load Generator for baseline comparisons of later results.)



Note the *bursty* nature of the data over the elapsed time period, characteristic of an exponential arrival time pattern, and a close approximation to real world network behavior. Note also the short response times associated with these initial results as described above.

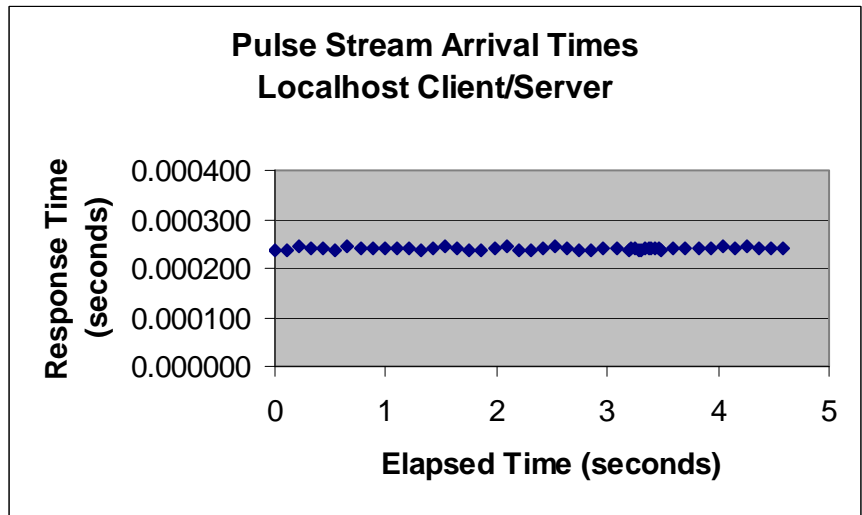
2. Sweep

The sweep distribution is one of the two cyclical distributions, operating over an arbitrary burst set of fifty query packets. That is, the initial time delay is 1000000 microseconds, or a full second. Subsequent delay times are each reduced by 20 milliseconds, for a final delay time (between the 49th and 50th packets) of 40 milliseconds. The effect on the server is to receive faster and faster queries from a particular client. (This range may be scaled downward later for stress testing.)



3. Pulse

The pulse distribution, similar to the sweep distribution described above, arbitrarily selects a 100-millisecond delay time between every query except the 30th through the 40th, which are delayed by 20 milliseconds. The effect on the server is to receive a sudden burst of high-speed queries from a particular client.



Comments:

Finally, consider the fact that Linux is a multitasking operating system and even a modest desktop PC may support many client processes, each with one of the above arrival rate distributions configured for server behavior observations under a variety of combinations of loading stress.