

Converting Single, Concurrent Server Software for a Cluster Server Architecture

-Abstract-

This paper presents the MSI, short for our Manifold Server Interface, an application programmer interface that greatly simplifies the conversion of single system, concurrent server software for use on a multicomputer, cluster architecture. The MSI requires only minor source code changes to existing server software. It eliminates all concern for inter-processor communications and other details associated with a distributed environment so new server software can be designed from a high-level application standpoint. We present the MSI software architecture in detail, an example single system server conversion, and performance measurements from our test bed prototype. While time-to-market is a critical concern for newly developed products, we draw attention to the need for rapid conversion tools that focus on keeping up with ever-increasing customer demands. At some point in its development cycle, a server at its peak performance can fail to meet the needs of its clientele. When a simple hardware upgrade isn't the answer, and a complete rewrite of the server software is ill advised, we suggest using an API that encapsulates the distributed system's details, leaving you to focus on the application.

Introduction

We present the MSI, or Manifold Server Interface, an API that allows the rapid conversion of single-system server software for use on a multicomputer, cluster server architecture. The use of this API requires only minor changes to the single-system source code. It may also be used as a kernel for the design of new server software, allowing the designers to focus on the application instead of the distributed system and all of its inter-processor communication issues.

While rapid software development is in the forefront of discussions about product introductions, rapid advancements in client device technology is bringing to light a need for rapid software product *improvements*. Conversion systems are becoming necessary just to keep up with ever-increasing client demands. We believe our MSI represents a distributed software architecture trend in support of these kinds of efforts.

The concept of an API to encapsulate all the sophistication to support multicomputer server architectures began as a casual conversation about the nature of client-server computing.

Client-server computing began when the interactive terminal user's processing was broken (somewhat arbitrarily) into two major components, dubbed the client and the server. (The name *server* was applied in the context of a *data* server because it executed on the machine where the data resides.) As client computers increased in processing power, their level of responsibility rose accordingly. The server functionality was transferred little by little to the client side until the server did only what was absolutely necessary to service a data request. This allowed the server to maximize its potential concurrent client load.

The recent flood of underpowered client computers in the form of hand held personal digital assistants and cellular telephones, however, suggested that this trend was about to reverse. Given an increasing number of clients with only limited capabilities, we

reasoned, the servers are going to have to take on the lion's share of the loading. Moving to a scalable, distributed cluster computer architecture should give the server room for long-term growth, but at the cost of rewriting single-system server software to run in a distributed environment. That's when our idea for a generalized conversion API took shape.

We initially designed a software architecture with a reentrant *replica* of the client running on the server. The remote client interacted with its locally resident clone, dynamically adapting to the rapidly changing loads by agreeing to shift the point of remote procedure call as necessary, to rebalance the load [1].

Post-presentation feedback led us to discuss the usefulness of not a pair of client clones, as we had developed, but N replicated servers, on a multi-computer architecture. Discussions led to a proposal to design, implement, and study the effectiveness of an Application Program Interface (API), used as a reduced-cost-and-complexity approach to supporting a high availability server on a multi-computer architecture¹.

This paper presents the software architecture of the MSI, an example system that we converted from a single to a multi-computer architecture, and performance measurements on a prototype running on our laboratory network.

System Architecture Overview

Our MSI system software has two major components: the API and the inter-processor *Liaison* task. All programming was done in the C language, running in a Linux operating system 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 the 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. Examples of such environments include:

- Real-time, conversational digital voice communication processors where lost or reacquired packets may cause confusion.
- High volume, globally distributed transaction processors where multi-phase commitments to database updates require long data transmission times.
- Battlefield telemetry processors where real-time, life-critical data are combined for central situational awareness display.
- Audio and video data stream processors where transmission interruptions might interfere with the correct interpretation of the material being presented.

In addition to the MSI and the *Liaison* process, we designed a client transaction generator and performance display monitor, both of which run on processors external to the cluster.

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

The performance monitor displays all eight cluster processor's resource utilization in near real-time. A block diagram of our test bed is shown in Figure 1.

¹ This research is funded by the Consortium for Embedded and Internetworking Technologies. Further details may be found through their link at <http://www.asu.edu> home page.

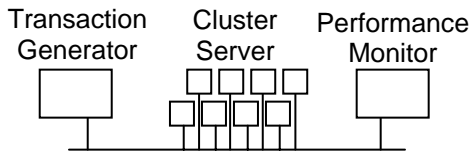


Figure 1: Cluster Server Test Bed

Each of the processor nodes runs a replica of the single-system server software, modified and recompiled with the MSI, along with a Liaison task for inter-processor system communications support. Each Liaison task has a number of performance monitor subtasks associated with it, for CPU and memory utilization, and disk and network I/O rates. Each Liaison 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. Liaison keeps a local as well as a global state table of resource utilization for every remote processor in the cluster. (See Figure 2.)

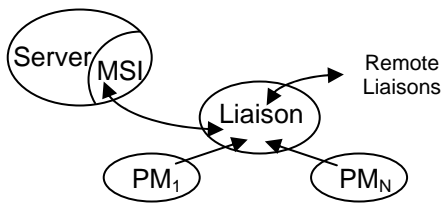


Figure 2: A Representative Node

Required Source Code Changes

The MSI appears to be a pair of functions to the application, the original single-system *concurrent* server. (A concurrent server is one that spawns a dedicated subtask to service each incoming query [2].) 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 [3]. 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(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 signal 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 a final call to MSI_init must be made 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.

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

- cc -c msi.c
- cc -DMSI -odapp msi.o app.c

The distributed application server will be in the executable named *dapp*, as indicated by

the -o flag in the compiler command above. Note that the executable is named *dapp* (for distributed application) so that it may be distinguished from an executable named *app*, compiled without the MSI variable:

- cc -c msi.c
- cc -oapp msi.o app.c

This simplifies testing by keeping the single-system executable separate from the cluster version.

Assuming the path to the executable *dapp* is accessible to each of your cluster processor nodes, through the network file system perhaps, and that one of these node machines is assigned the well-known IP address of your server, start *dapp* 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 *dapp* 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 the external clients.)

MSI Initialization

MSI_init, having been called twice with the names and characteristics of local functions in our example above, has built a local function table. (This table has an entry for each local function.) Each entry contains the local address, within the address space of the application, and the character string name of the function, as well as information about its passing parameters as inferred from the characteristics strings defined above.

MSI_init starts the Liaison subtask, which starts local performance monitoring subtasks (more about these later) and declares itself to be the *leader*. (It does so by broadcasting a "?" to every other node on the network and waiting for a few seconds. If there is no response, it assumes the role of leader. If it receives a "!" then it assumes the role of *worker*. In either case, the Liaison task signals back its role to the MSI.

The leader MSI returns to the application main task so it can begin receiving incoming queries and calling `MSI_call` as part of its query servicing. The worker MSI stays in the `MSI_init` function, in a loop that awaits work from its Liaison subtask. All the Liaison subtasks communicate amongst themselves, as well as with their local MSIs. A diagram of this interaction is shown in Figure 3.

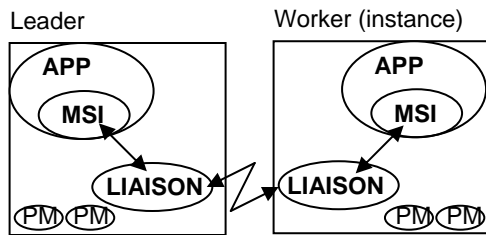


Figure 3: Leadership vs. Worker Nodes.

Remote Procedure Call

Instead of a direct call to a local function, the application calls `MSI_call` with the return and calling parameters. `MSI_call` puts each parameter into a packet, according to information it finds in the corresponding local function table. (The string name is used to locate the proper entry and is set as the first entry in the parameter packet. The string name is passed along to the remote system so it can locate its associated function table entry, giving the local address of that same function on that remote worker node.) This packet is passed from `MSI_call` to the leader node's Liaison subtask, which places it inside a larger packet, containing local performance statistics, and sends it to its counterpart Liaison on a worker node.

The worker Liaison accepts the incoming packet, updates its global performance statistics table from the incoming packet, and passes the packet of parameters to the local `MSI_init`. `MSI_init`, which was waiting for work, uses the function's string name to find the corresponding entry in its local function table, and issues a local call to that function with the parameter values from the

packet. The local function returns; `MSI_init` passes the packet of results back to its local Liaison, which sends these results back to the leader Liaison. It updates the performance statistics in its global table, and passes the results back to `MSI_call`, which returns to the application.

The original server was designed for concurrency, remember, so every incoming client query causes it to spawn itself as a subtask to handle just that query. Now, we get a separate copy of the MSI along with the server, and it may interact with one of the worker nodes, independently of any other copies, exploiting parallelism needed to gain an advantage from a cluster of nodes.

Note that the local performance statistics are added to every packet before it is sent out. Note too, that the remote performance statistics from incoming packets are used to update a global table at each node. It is these performance statistics that the leader uses to determine which remote worker to choose when it comes time for a RPC.

Performance Statistics

The Linux operating system provides for *proc* pseudo-files, which may be read and interpreted just like any text file. There are several of these available to the application programmer, but if the performance data sought is not available, one may write a new *proc* module in support of whatever metric must be supported [4]. Our MSI system runs under Linux and keeps track of four performance metrics: processor and real memory utilization, and disk and network rates (reads and write operations per second).

The Liaison subtask on each node starts four subtasks, one for each metric. The subtasks run asynchronously, updating their isolated performance data area in a memory space that is shared with the Liaison subtask, which captures these data regularly, updating its local performance statistics data

based on a running average of these individually calculated values. The times between Liaison's data captures, as well as the number of elements considered in the running average, are configurable.

Every time Liaison sends out a packet of parameters, it includes its local performance statistics. Each time a Liaison receives a packet, it updates the sender's entry in its global table from that packet's statistical data.

When the leader Liaison must choose a worker for a RPC it inspects these global statistical table entries. The first idle worker is chosen, if there is one. If no worker is idle, choice is based on a formula that quantifies the *busyness* B of each worker.

$$B = (P + M + D + N) / 4$$

where

P = percent processor utilization

M = percent real memory utilization

D = (100 * disk access rate) / max

N = (100 * network access rate) / max

Note that the *max* disk and network rates are hardware dependent and must be empirically determined. These max values effectively determine the percent utilization of the disk and network devices. (We are experimenting with more complex interpretations of B.)

Fault Tolerance

Each time the Liaison subtask captures the performance data from the shared memory, it resets each value to -1. If it captures a -1 for any metric it assumes that the performance monitor associated with that metric has failed (or is hung). It terminates the (possibly hung) subtask and restarts it, up to some configurable maximum number of restart attempts. This provides software

fault tolerance at the level of performance monitoring subtasks.

Each time the Liaison subtask is started it looks for a special temporary file with a shared memory id. If it finds the file, it reads the id and attaches the existing shared memory area. And as it captures performance data from that area, it may have to restart the performance monitors associated with each isolated metric data area.

If there is no such file, the Liaison subtask assumes that it is starting afresh, creates the shared memory area, and stores its id in the newly created id file. It proceeds to start each of the performance monitor subtasks and saves each process id in the shared memory area. This provides software fault tolerance at the level of the Liaison subtask and its shared memory, performance data area.

The application is restarted automatically through the addition of a RESPAWN entry in an OS system file [5]. At this writing we are investigating ways to elect a new leader should workers discover (or suspect) that the leader is no longer effective [6].

Timing and Instrumentation

The motivation to convert a simple, single system concurrent server into one that runs on a cluster architecture is often based on the expectation of improved performance. In the case of a transaction server, we interpret this improvement to mean an increase in the number of transactions per second that are successfully processed. (We wrote the server under test as well as the client.) Each query goes through a complex set of processing steps at the server, but the client can calculate what result to expect in advance and compare it to what the server returns. Both correct and incorrect responses are counted, and a percentage of correct answers is used as a threshold to determine the maximum sustainable client transaction rate.

Furthermore, the percentage of accurate responses may be continually displayed by each client as an indication that accuracy may be falling off, perhaps due to excessive server load.

In an effort to improve the performance of the server, we time each phase of the RPC, and investigate those timings that appear to be excessive.

Tanenbaum, in [6], suggests breaking the RPC process into several phases, timing each of them during a stream of queries, and analyzing them graphically, as a bar graph. We defined fourteen phases. (See Figure 4.)

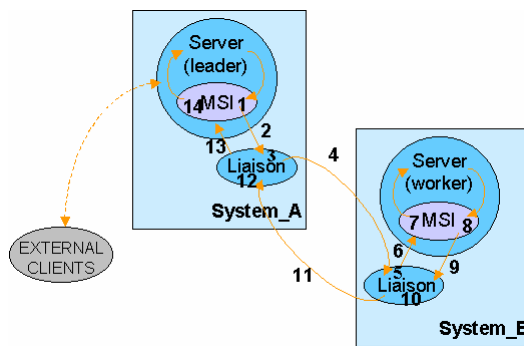


Figure 4: The Fourteen MSI Phases

Note that phases are inside the MSI, but not inside the application code. We decided to focus on tuning the MSI independently of the application. Note also that there is a classic distributed time-of-day clock issue when measuring the 4th and 11th phases. We save the time a packet leaves the leader Liaison task (enters phase 4), and subtract that from the time it reenters (leaves phase 11). The remote Liaison task similarly saves the time that the packet enters and subtracts that from the time it leaves, entering the difference in the packet itself. The leader Liaison subtracts the remote time from the difference it calculates to get the packet's round-trip time on the network. It reports half this time for both phases 4 and 11. A bar graph of our test bed's performance (before any tuning) is shown in Figure 5.

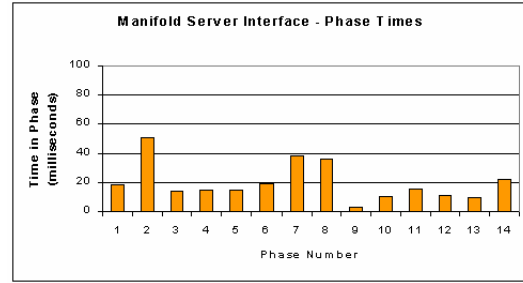


Figure 5: Test Bed MSI Phase Times

Message Logging

The usual way to document errors or critical state changes is through print messages. As this system is a collection of subtasks with complex interactions, we created a pair of log files for the MSI and the Liaison on each node. When one of the log file pair fills to some configurable maximum number of lines, a second log file is opened. The files are in the local node's temporary disk area, named msi0.log and msi1.log. When msi0 fills msi1 is reused; when msi1 fills, msi0 is reused, saving a large amount of potentially critical information during a debug run.

The Example Application

We chose an application requiring several steps to process a query, but wanted it to be simple enough to explain in a paragraph. Our focus here is on the API.

The client sends a simple math problem as words in a character string. It gets the result back in a numeric string. For example, if the client sends “**three times four**” the result is “**12**” from the server. (We said simple.) This gives us two opportunities: First, the client can pre-calculate the result as it generates its stream of random queries, and can check the server's response for accuracy. (In a very heavy loading situation errors are expected.) Second, the server calls five functions to process each string - good exercise for our Manifold Server Interface.

Here are the five calls:

1. Function *parse* breaks the above query string into its three components: “three”, “times” and “four”,
2. *numbers* converts the string “three” returning the integer 3,
3. *opcodes* converts the string “times” into the single character “*”,
4. *numbers* converts “four” to integer 4,
5. *solve* takes the two numbers and the operation code, returning “12”.

Each time the original server uses *MSI_call* to indirectly call a server functions, a remote procedure call occurs. The RPC executes on the remote processor that is least *busy*, per our definition above. The parallelism inherent at the query level (when the queries are coming faster than a single server can handle them) is exploited by this simple system. We expect the performance will excel under loads so heavy that the inter-arrival times are shorter than the service time, but not so heavy that too many queries are lost.

Performance Results

Our initial system level performance data acquisition resulted in a plot of completion times, the time to process a fixed number of packets with an exponential inter-arrival rate [7] against the number of worker nodes. See Figure 6.

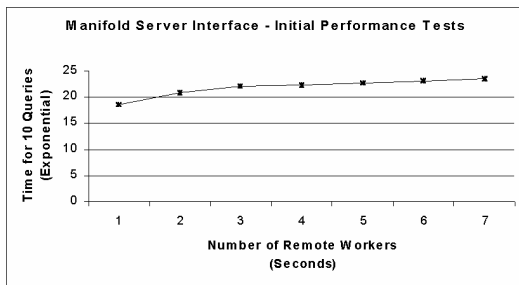


Figure 6: Initial Performance Data

We call this the Fred Brooks Effect because the completion time increases as workers are added [8]. This clearly indicates a need for tuning. A look at the phase times (Figure 6) suggests the link between the MSI and the leader Liaison, as well as the marshaling and un-marshaling of parameters in the worker.

There is also a possibility of a design flaw in the software architecture that interferes with exploitation of parallelism. We disproved this hypothesis by forcing longer execution times and rerunning the system performance completion times against the number of workers (Figure 7).

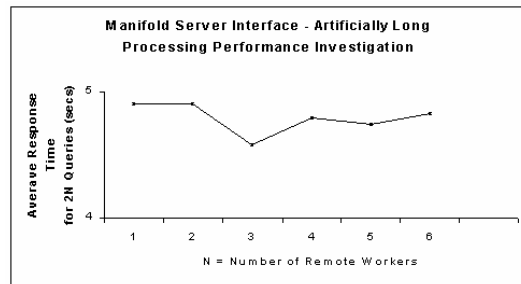


Figure 7: Performance Test Rerun with Artificially Long Processing Times

We note that once the data processing time is significantly longer than system overhead toward cluster support, there is an indication that performance improves as workers are added, up to some point of diminishing returns. This is the expected result.

Now that we know the logic of the software architecture is basically sound, we can look for guidance from our phase time diagram in Figure 5. Our first change toward improving the efficiency of this system will be to replace the TCP/IP link between the MSI and the leader Liaison with shared memory. The next change will be to optimize the code in the worker Liaison that handles passing parameters in the RPC packets.

Conclusions

We set out to develop an alternative to ORB-based exploitation of parallelism by designing an API, one that encapsulates all the sophistication required to manage a cluster system architecture, so developers need not concern themselves with those details, focusing on their application.

While our early performance results indicate some changes are necessary in our architecture, we believe this effort sets a direction for future single-system to parallel-system conversions.

References

1. A. Vrenios, "Straddled Clients and Adaptive RPC" 3rd IEEE Workshop on Mobile Computing Systems, December 8, 2000.
2. *Unix Distributed Programming*, C. Brown, Prentice Hall, 1994.
3. *The C Programming Language*, 2nd Edition, B. Kernighan and D. Ritchie, Prentice Hall, 1988.
4. *Linux Programming White Papers*, D. Rusling, et al., Coriolis Press, 1998.
5. Linux man pages for Unix file *inittab*.
6. *Distributed Operating Systems*, A. Tanenbaum, Prentice Hall, 1995.
7. *Statistical Distributions*, 2nd Edition, M. Evans, N. Hastings and B. Peacock, John Wiley & Sons, 1993.
8. *The Mythical Man-Month*, Fred P. Brooks Jr., Addison-Wesley, 1975.