Distributed Shared Memory for Real Time Hardware In the Loop Simulation

by

Timothy Elmer Kalvaitis

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Bachelor of Science

and

Master of Science in Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

© Timothy Elmer Kalvaitis, MCMXCIV. All rights reserved.

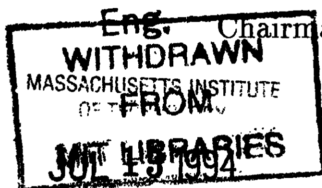
The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Signature of Author ....
Department of Electrical Engineering and Computer Science
May 6, 1994

Approved by ....
Dave Hauger
Technical Supervisor
The Charles Stark Draper Laboratory

Certified by ....
R. John Hansman Jr.
Thesis Supervisor
Associate Professor of Aeronautics and Astronautics

Accepted by ....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students


Eng.
WITHDRAWN
MASSACHUSETTS INSTITUTE
FROM
MIT LIBRARIES
JUL 13 1994

Distributed Shared Memory for Real Time Hardware In the Loop Simulation

by

Timothy Elmer Kalvaitis

Submitted to the Department of Electrical Engineering and Computer Science
on May 6, 1994, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science
and
Master of Science in Electrical Engineering

Abstract

This thesis describes a distributed shared memory implementation in the context of a framework designed to simulate six degree of freedom vehicle dynamics and control systems with hardware in the loop in real time. Distributed shared memory creates the illusion of a common address space between simulation processes located on different networked computers.

Direct memory access techniques in the form of memory mapped data structures are applied to the development of an input/output system for the simulation framework motivating the use of direct memory access as a paradigm for communication. This concept is extended to shared memory for interprocess communication. Mutual exclusion semaphores and read/write locking are discussed as a solution to the problem of concurrent memory access associated with shared memory communication.

Direct memory access is extended to network communication in the form of distributed shared memory. The primary/secondary copy distributed database is used as a model for the implementation of distributed shared memory. Concurrency control and memory consistency are discussed. Read/write locking is used to insure local memory consistency and an additional lock type is proposed as a means of enhancing performance by relaxing the inter-host memory consistency constraint. Time step synchronization of simulations communicating via distributed shared memory is discussed as a method of enforcing inter-host memory consistency when required.

Two examples of the use of distributed shared memory for simulation are provided. Performance of the implementation is evaluated and discussed in terms of both general simulation requirements and the two examples. Two other communication techniques are compared to the direct memory access model for performance, scalability, and extensibility from interprocess to network communication.

Technical Supervisor: Dave Hauger
Title: Principal Member of Technical Staff, Charles Stark Draper Laboratory

Thesis Supervisor: R. John Hansman Jr.
Title: Associate Professor of Aeronautics and Astronautics

Acknowledgments

First and foremost I would like to thank my wife, Caroline, for what seems like an endless supply of understanding. Without her love and support, I would have been unable to complete my studies at MIT. I look forward to many more wonderful years with her, our daughter Mary Beth, and our son Ricky.

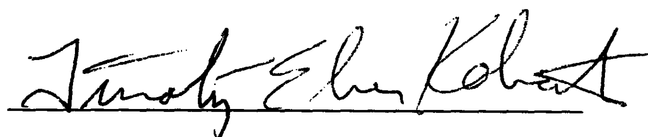
Thanks to all the Sim Lab members at Draper. In particular, I would like to thank Mario Santarelli for giving me the opportunity to pursue this work unencumbered by red tape and beauracracy. Dave Hauger for his sound technical advice and common sense understanding of computing. Bill McKinney for his patience in using (testing) the latest versions of my code and his willingness to listen to my ramblings. John Danis for his conservative approach to problem solving. Thank you to Chris Winters for teaching me the value of detail. To Dave and Chris I am especially indebted. Dave's original VAX simulation framework and Chris' subsequent UNIX design were both truly unique creations. Without the simulation framework this thesis would not have been possible. More important than the professional support I have received from these people has been the personal friendship we have developed over the past three years. For that I will always be grateful.

Finally, I would like to thank Professor Hansman for his candor in reviewing my work as well as his good advice during the writing of this thesis.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under Internal Research and Development Project #15104, "Precision Vehicle Positioning and Autonomous Work Systems".

Publication of this thesis does not constitute approval by Draper of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

I hereby assign my copyright of this thesis to The Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.



Permission is hereby granted by The Charles Stark Draper Laboratory, Inc., to the Massachusetts Institute of Technology to reproduce any or all of this thesis.

Table of Contents

1 Introduction **13**

1.1 Simulation Framework 13

1.2 Input/Ouput Requirements 14

1.3 Direct Memory Access for Device Communication 15

1.4 Direct Memory Access for the Simulation Framework 17

2 Implementation **23**

2.1 Simulation Framework Database Processor 23

2.2 Structure Maps 25

2.3 Shared Memory 27

2.4 Concurrent Memory Access 32

2.5 Distributed Shared Memory 35

2.5.1 Design Considerations 37

2.5.2 Primary/Secondary Copy Distributed Databases 39

2.5.3 Concurrency Control 42

2.5.4 Updates 45

2.5.5 Synchronization 47

3 Examples and Results **53**

3.1 Example 1 – Gantry Limit Switches 53

3.2 Example 2 – Simulation of an Autonomous Underwater Vehicle . . . 61

3.3 Distributed Shared Memory Network Performance 67

4 Comparison of Other Communication Methods **73**

4.1 Interprocess Communication 74

4.2 Network Communication 79

5 Conclusion **87**

A Performance Data **91**

References **96**

List of Figures

1-1 Structure maps and hardware access 16
1-2 Structure maps and shared memory 18
1-3 Distributed Shared Memory 19
1-4 Possible configuration using structure maps, shared memory, and distributed shared memory. Hosts 1 and 3 run the simulation. Host 2 communicates with attached hardware. Hosts 4 and 5 handle graphics intensive display of both hardware and simulation data using simulation framework tools. 21
2-1 Simulation framework processing, representation, and organization. . 24
2-2 Structure maps in the framework. 26
2-3 Structure map to shared memory. 28
2-4 A structure and its representation in shared memory. 31
2-5 Address references from shared memory to process memory. 36
2-6 A structure and its representation in distributed shared memory. . . . 38
2-7 Primary/Secondary organization for distributed shared memory. . . . 41
2-8 Primary and secondary site lock procedures. 44
2-9 Synchronization of simulation time steps. 49
2-10 Synchronization of simulation time steps using distributed shared memory. 51
3-1 Gantry hardware configuration. 55

3-2 Specification of register map for dicrete input board. 56
3-3 Specification for motor discretes. 57
3-4 Network and process configuration for unmanned undersea vehicle test. 62
3-5 Specification of vehicle state information for display. 64
3-6 Specification for calculated viewing option. 65
3-7 Simulation frequency as a funtion of data size for example two. 66
3-8 Maximum simulation frequency as a funtion of data size. 69

4-1 Point to point communication. 74
4-2 Client/Server communication. 75
4-3 Centralized shared memory communication. 76
4-4 Distributed shared memory communication. 83

List of Tables

3.1 Execution times for example one without hardware access and without distributed shared memory. 58

3.2 Execution times for example one with hardware access but without distributed shared memory. 59

3.3 Execution times for example one with both hardware access and distributed shared memory. 60

A.1 Execution time of 6000 time steps measured for varying data size in distributed shared memory for example two. 93

A.2 Execution times measured for varying data size in distributed shared memory. 94

Chapter 1

Introduction

This thesis describes a distributed shared memory system as an extension of concepts applied during the development of a generic input/output system for a simulation framework. This chapter motivates the development of distributed shared memory in the context of this framework. Chapter 2 details the implementation of the distributed shared memory system. Chapter 3 relates two examples of its use and discusses the network performance of distributed shared memory. Two other communication techniques are compared to shared memory and distributed for performance, scalability, and extensibility from interprocess to network communication in Chapter 4.

1.1 Simulation Framework

The development of the distributed shared memory system described in this thesis took place as part of a more general effort to develop a framework for real time hardware in the loop simulation [10]. This framework was designed to support the simulation of six degree of freedom vehicle dynamics and their associated hardware components. To date the framework has been used to simulate the dynamics of air, land, and undersea vehicles and the dynamics of vehicle-truss interaction in space. In addition, simulation of guidance, navigation and control systems is supported as

is the development of realistic instrumentation for the simulated vehicles.

The framework's existing capability, prior to this thesis, was based on a database preprocessor used to define simulation data structures and generate run time information describing those data structures. This information provided the basis for analysis tools such as plotting, logging, and profiling, as well as programming and debugging tools used to monitor and manipulate the simulation's data structures at run time. Hierarchical organization of data was used to manage the complexity of large scale simulations. Absent before this thesis was the capability to access hardware in real time.

To meet the goal of hardware in the loop simulation, an input/ouput (I/O) system, fully integrated with the existing framework, was required. During the course of this development, the idea of direct memory access as a means of communication was applied to hardware access and then extended to a more general mechanism for interprocess and network communication within the simulation framework. The remainder of this chapter discusses the requirements for the I/O system, and provides an overview of the development of direct memory access for device, interprocess, and network communication.

1.2 Input/Ouput Requirements

In order to interface with vehicle hardware systems, the following four I/O requirements were imposed:

1. Real time operation. This requirement was two-fold. First, real time operation required synchronizing simulation time steps with real time rates. Second, real time operation imposed a speed constraint — the I/O system needed to be as efficient as possible in order to access hardware fast enough to sustain both simulation and device rates. Real time frame rates on the order of 100 Hz or greater were required to support existing simulations. Microsecond access times were required to support a wide variety of hardware.

2. User level hardware communication. In order to avoid the overhead of operating system calls and long interrupt latencies associated with device drivers, communication with the hardware at the user process level was required.

3. Support for modular functional models transparent to the simulation. Modularity required the ability to interchange models and hardware at any time. Transparency required access to real hardware or models of hardware be indistinguishable from the simulation's point of view. In this way, a simulation can communicate with hardware or models of hardware without the need for special code.

4. Tight integration with the existing simulation framework. The simulation framework already provided several tools uniquely suited for monitoring variables and data structures in a simulation. In order to bring these tools to bear on the I/O problem, tight integration with the framework was required.

These requirements were met by applying the idea of direct memory access as a means to communicate between the simulation framework and hardware devices.

1.3 Direct Memory Access for Device Communication

Direct memory access in the context of this thesis is the communication of data from one process to another through common address space[1]. The mechanism used to support this communication for the I/O system was the *structure map*. Structure maps are data structures that have been relocated to arbitrary address space in a way that is transparent to simulation code.

For hardware communication, direct access means communicating data from a process on the host to a device attached to a bus occupying some of the host's address space [3, 6]. Figure 1-1 illustrates this type of communication. The simulation is a process on the host that has access to the memory locations used to communicate with the bus. The device occupies a predefined set of addresses on the bus. In order

[1] This is distinct from the concept of direct memory access (DMA) as a means of asynchronous communication between a device and its device driver in the host's operating system.

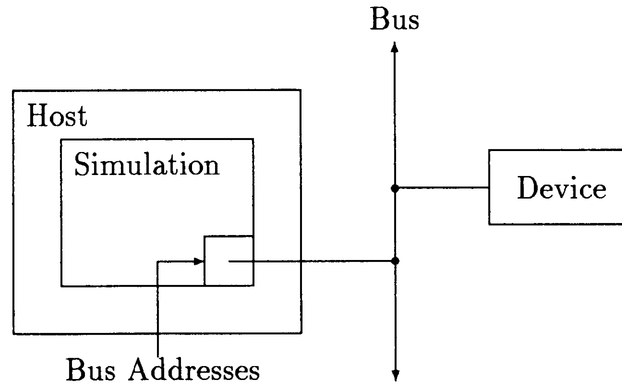


Figure 1-1: Structure maps and hardware access

to communicate with the device, the simulation simply references the appropriate bus locations via a structure mapped to those locations. Use of the framework to generate the structure map allows the framework tools to directly access the device as well.

Direct memory access through structure mapping satisfied the I/O system requirements as follows:

1. Direct memory access is efficient. Direct bus access is one of the most efficient ways in which a host can communicate with a device. This efficiency more than satisfied real time speed requirements as bus access times are on the order of microseconds.

2. Direct memory access can be applied from user level programs. Bus addresses mapped to user process memory are accessible without the need to involve the operating system.

3. Direct memory access enabled support for modular, transparent functional models of hardware. The hardware processes of Figures 1-2 and 1-3 could either be communicating with devices attached to the bus, or simulating those devices. Separating the functionality of the simulation and the hardware at the process level provided a means of modularization. Transparency was obtained because communication occurs through simulation data structures that are mapped to either bus addresses or other

parts of memory. The framework performs the mapping operation without the need to change simulation code.

4. Direct memory access enabled tight integration of the I/O system with the existing framework. The framework maintains run time information for all simulation data structures. Direct memory access allowed that information to be extended to bus relocated structure maps. As a result, the framework modeling, analysis, and program debugging tools built on this information were able to be applied to the I/O system.

Application of direct memory access to device communication was the key to satisfying the I/O system requirements. During the course of this application, however, it was found that direct memory access as a paradigm for communication was also able to be applied to other areas of the simulation framework.

1.4 Direct Memory Access for the Simulation Framework

The simulation framework's capabilities were extended by applying direct memory access to interprocess and network communication. As with device communication, this concept relies on communicating data from one process to another through common address space. Interprocess communication was implemented through shared memory [1]. Network communication was implemented by extending the concept of shared memory to distributed shared memory.

Figure 1-2 depicts a possible configuration for interprocess communication. There are two processes on the host: one carries out the simulation, the other communicates with the hardware. As before, device access is accomplished via structure map to bus addresses. In similar style, communication between the simulation and hardware processes is accomplished via structure map to shared memory. More generally, structure maps in shared memory allow communication of data structures between arbitrary processes on the same host.

Communicating data structures between processes on different hosts is the goal of

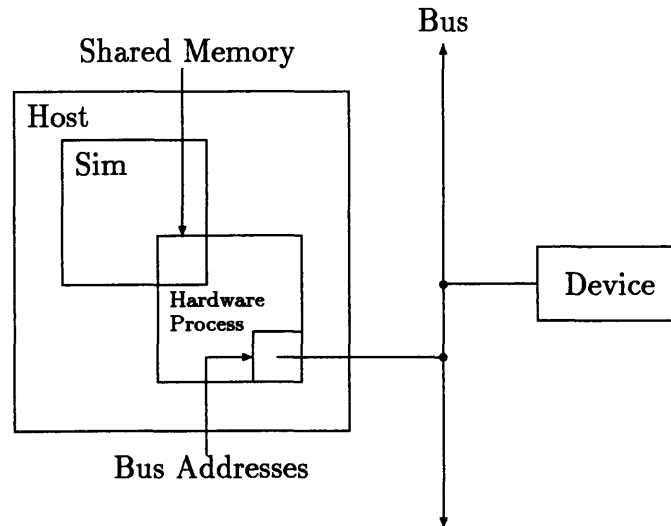


Figure 1-2: Structure maps and shared memory

distributed shared memory. Mapping structures to shared memory, and maintaining the consistency of copies of that memory over several networked hosts creates the illusion of shared memory over the network. This extension of direct memory access is depicted in Figure 1-3. In this example, the simulation resides on host 1 and, as before, the hardware processes communicates with the device via bus addresses on host 2. Communication of data between the hardware process and the simulation, from the point of view of each process, is identical to shared memory communication. The framework, using its run time representation of simulation data structures, carries out the network communication for each process and insures that the distributed shared memory is consistent between the connected hosts.

The use of direct memory access communication extended the simulation framework's capabilities by providing a simple, transparent means of interprocess and network communication. Shared memory provided general support for transparent communication between processes built with the framework. The hardware process of

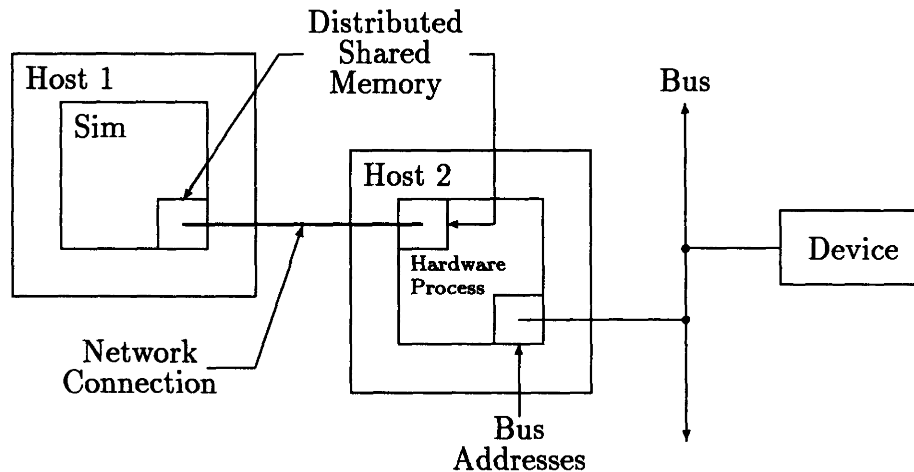


Figure 1-3: Distributed Shared Memory

Figure 1-2 could easily be another simulation communicating any data structure defined using the framework. Distributed shared memory extended this transparent communication across a network. The hardware process of Figure 1-3 could easily be another simulation sharing information over a network. Communication is simple because the framework handles the communication details — the simulation merely accesses memory.

Transparent network communication through distributed shared memory provided greater access to computational resources. This significantly enhanced the framework's capability for simulation. Access to remote hosts and specialized computing equipment is now possible allowing simulation tasks to be assigned to computing equipment best suited for the task. Graphics intensive display can be executed on machines with more graphics capability while another machine, connected to hardware, can devote its resources to I/O. Still other machines can be devoted to model computation without loading the I/O or graphics computers.

The framework's run time information of simulation data structures can also be mapped to distributed shared memory allowing one simulation to monitor another simulation without the need for recompilation of either program or *a priori* knowledge

of the other simulation's data structures. This simplifies source code control (common header files between simulations are not necessary) as well as inter-simulation communication. In short, any simulation can communicate its variables to any other simulation provided they are both built using the framework. Figure 1-4 shows a possible networked configuration utilizing all of the aspects of direct memory access available in the simulation framework.

The remainder of this thesis details the implementation of direct memory access for device, interprocess, and network communication in the context of the simulation framework and provides examples of its use in simulation.

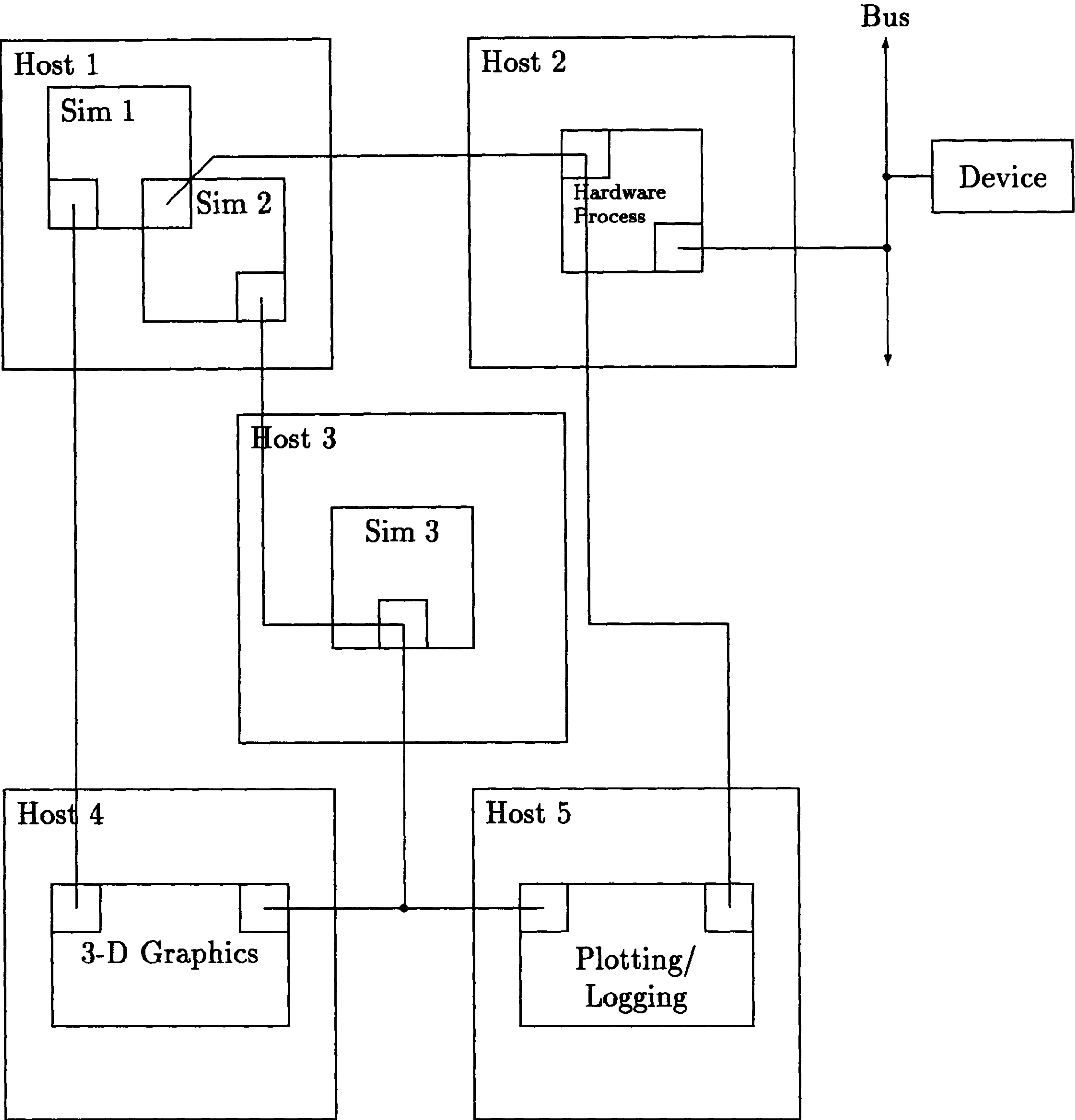


Figure 1-4: Possible configuration using structure maps, shared memory, and distributed shared memory. Hosts 1 and 3 run the simulation. Host 2 communicates with attached hardware. Hosts 4 and 5 handle graphics intensive display of both hardware and simulation data using simulation framework tools.

Chapter 2

Implementation

2.1 Simulation Framework Database Processor

The simulation framework is designed to provide access to data structures used in a simulation implemented using the C programming language. The framework includes a compile time structure preprocessor and a run time simulation environment. Figure 2-1 illustrates the compile time processing and run time organization used by the framework. Data structures are defined in specification (spec) files using a syntax resembling C. A spec file preprocessor called the database processor, or dbp for short, generates three C source files. In Figure 2-1, the file foo.spec defines a data structure called `foo_ref` and declares the global variable `foo` to be an instance of that type. The database processor generates a header file, a definition file and a file containing a run time representation of `foo`. The header file contains the C structure definition for `foo_ref` and the declaration of the global variable `foo`. Simulation source files include this header file and access the elements of `foo` by standard C programming means. The definition file defines the global variable `foo` and initializes `foo` with values specified in the spec file.

The run time representation of foo (called `foo_rep` in Figure 2-1) is contained in the third file. The representation includes information about the location of `foo`, the

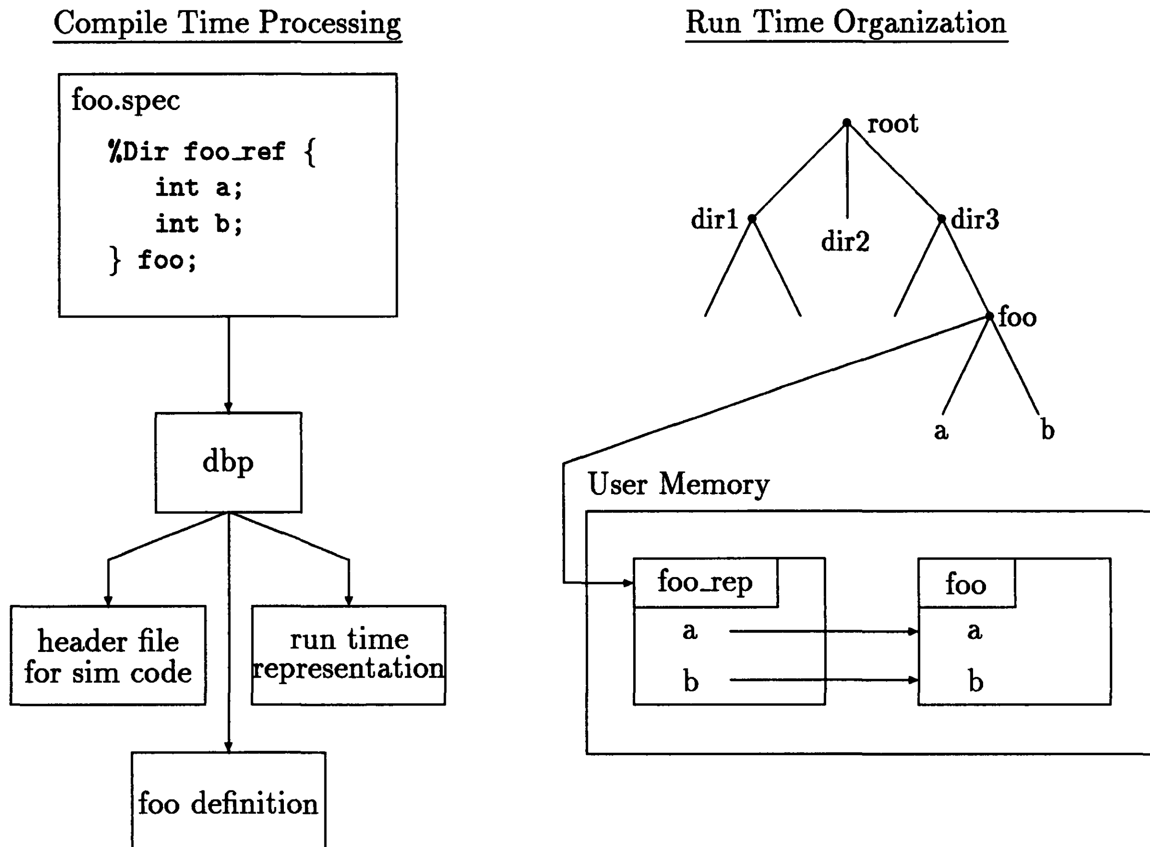


Figure 2-1: Simulation framework processing, representation, and organization.

memory layout, name, type, and size of each of the elements of type `foo_ref`, and the position of `foo` within the run time hierarchy of data structures. The run time organization illustrated in Figure 2-1 shows the conceptual relationship between `foo` and `foo_rep`. The run time hierarchy contains a pointer to `foo_rep` which holds the names and locations of the elements of `foo` (namely the variables `a` and `b`).

The run time simulation environment of the framework is built on the information generated by the database processor. A command interface controls the environment. Access to simulation data structures provides the ability to constantly monitor, log,

plot and change any variable used in the simulation at run time. This access is accomplished by looking up the name of the global structure in a hierarchy name table and then looking up the name of an element of the structure in its run time representation. For example, access to element `a` in the structure `foo` (referred to as `foo.a` in simulation source code) is accomplished by first locating `foo_rep` by the name "`foo`" in the hierarchy. Once found, `foo_rep` provides the address, type, and size associated with element `a` which is enough information to access the value of `foo.a`.

2.2 Structure Maps

The fact that data structures and their representations are stored separately allows those data structures to be relocated without affecting the framework's run time tools. This fact was exploited in the development of the I/O system in the form of structure maps. A structure map is a data structure that is memory mapped to another location. Device communication is accomplished by reading and writing into a structure that defines the register interface of the device. This structure is then mapped to the bus address of that device[1]. Figure 2-2 illustrates the relationship between a bus relocated structure map and the run time organization of the framework. In the example, the structure `foo`, defined in a spec file, has been mapped to a bus location where `a` and `b` refer to registers on the device. The run time representation of `foo` has been altered to reflect the new location of `a` and `b`. The structure hierarchy is unchanged.

Structure maps provided efficient accesses to hardware devices from the user process level on the same host as the simulation. Structure maps also provided the opportunity to model a device at the register level. Communication between a hardware model and a simulation could occur by monitoring a register definition structure that

[1]The bus address of a device is generally set by switches or jumpers located on the device.

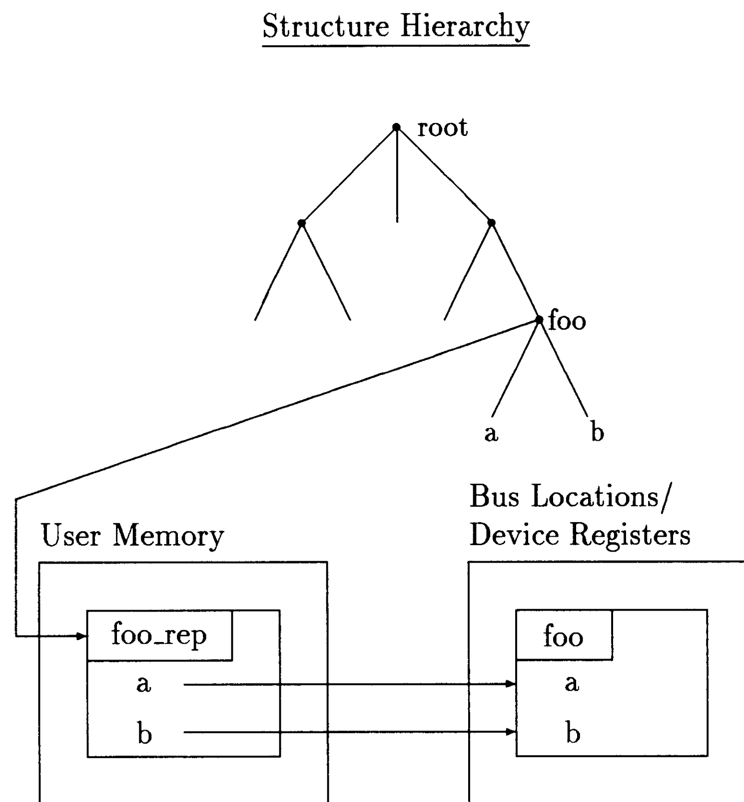


Figure 2-2: Structure maps in the framework.

has *not* been relocated to bus addresses. However, without the ability to relocate the register definition structures to address space accessible by separate processes, both the hardware model and the simulation needed to be part of the same process. This reduced the transparency of the model. Given that real devices operate in parallel with the simulation, a transparent model of those devices should also operate in parallel (at least conceptually) in order to be transparent to the simulation. This shortcoming lead to the development of structure maps in shared memory.

2.3 Shared Memory

The most straightforward way to provide communication between two simulations using structure maps was to locate the structures in shared memory. Figure 2-3 shows a possible configuration. As with device communication, `foo` is defined in a spec file and relocated to a new address. With interprocess communication, however, the new address is a shared memory address, accessible from both process 1 and process 2. Both processes have their own run time representations of `foo` that have been altered to reflect `foo`'s new shared memory location. Modification of `a` or `b` by either processes will immediately be seen by the other process.

Structure maps in shared memory provided the capability to develop hardware models that communicate with a simulation using the same register level interface as the real device. As a separate process, these models are conceptually distinct from the simulation in the same manner that the real devices are conceptually distinct. This allowed a model of hardware to be substituted without changing the code used by the simulation to communicate with the device.

As a general mechanism, structure maps in shared memory allowed devices to be modeled at a level higher than the register interface. By defining the hardware to simulation interface at a level higher than the device registers, the simulation could communicate with hardware without requiring device specific information. As an example, suppose an analog to digital converter was attached to the bus of the

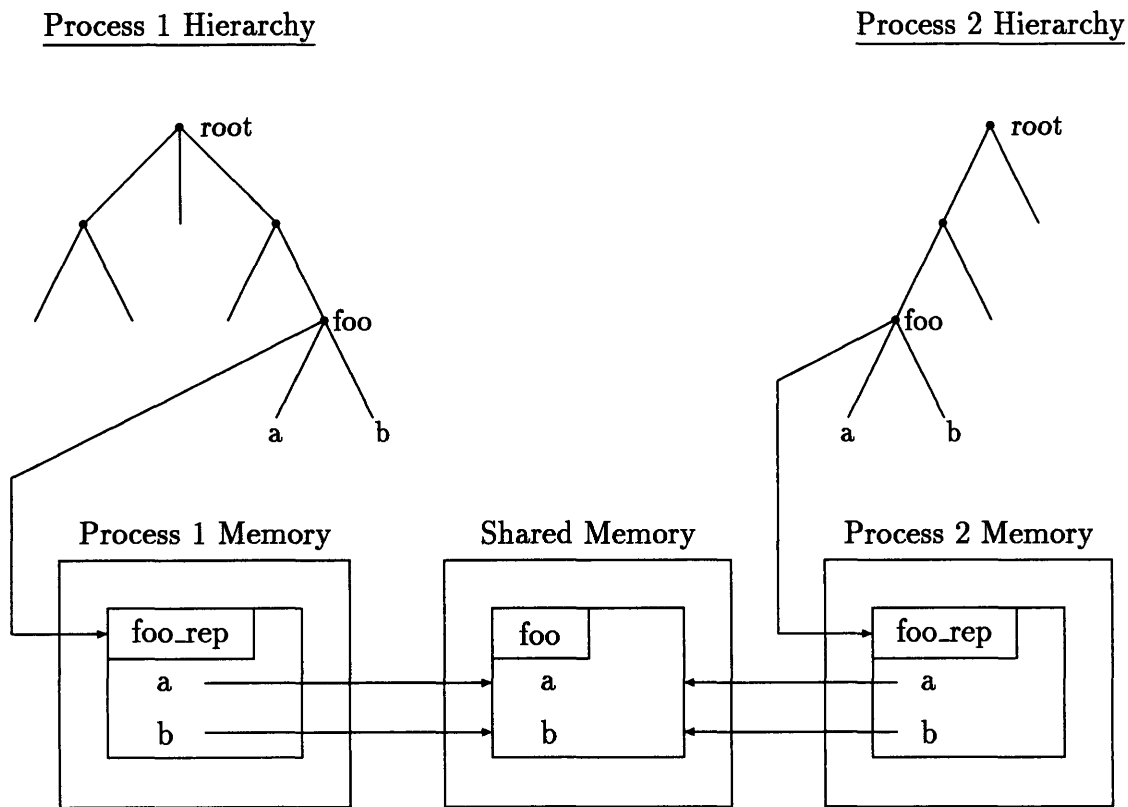


Figure 2-3: Structure map to shared memory.

simulation host. To communicate with this device, a spec file containing register definitions that control the device and provide the digital voltage values could be defined and mapped to the appropriate bus location. Suppose the digital values on the device are 12 bit integer quantities. To use these values as voltages, they must be converted from 12 bit integers into a floating point value indicating the analog voltage read by the device. If this conversion is done by the simulation, there is no possibility of using a different analog to digital converter. In order to use a more accurate converter (like a 16 bit converter), the simulation code would have to be changed. In addition, any model generating the voltage information would have to be a model of the analog to digital converter — it could not be a model of the analog signal itself as the simulation is expecting a 12 bit integer in a register of the analog to digital converter structure. It would be far more efficient to just generate the floating point voltage directly.

To solve this problem, the simulation could communicate with a process that represents the hardware. The simulation could read floating point voltages from a structure mapped to shared memory while the hardware process communicated with the analog to digital converter via a register definition structure mapped to bus locations. The hardware process would handle the conversion to a floating point voltage value and place the result in shared memory. This would isolate the simulation from hardware changes (like an upgrade to a better converter) as well as provide a mechanism for efficiently modeling the signal coming from the converter — the hardware process could just send floating point voltages without doing any conversion.

In addition to device access and hardware models, structure maps in shared memory provided a general mechanism for communicating data between processes developed using the framework since any structure defined in a spec file could be relocated to shared memory. However, while the basic idea of structure maps in shared memory was sound, the implementation was limiting. In order to communicate, two processes had to be compiled with a common spec file that defined the communication interface.

This implicitly limited the structures that *could* be communicated between the two processes to only those known at compile time. While this was not a severe restriction for well defined interfaces like device register specifications, it did not maximize the flexibility available from the framework's run time tools.

As an example, suppose that a hardware process and a simulation process were running on the same machine communicating through shared memory. It would be possible to plot the variables of process 1 using the framework tools from process 1 and it would be possible to plot the variables of process 2 from process 2. It would also be possible to plot the variables in shared memory on these same plots from either process 1 or process 2. However, these would be the *only* variables in either process for which this was true. It would not be possible to plot a variable from process 2 on a plot of variables from process 1 unless those variables had been defined at compile time as shared variables.

Placing the run time representation of a structure in shared memory along with the structure map itself eliminated this compile time restriction. Figure 2-4 illustrates the resulting organization. In this example, both `foo` and `foo_rep` are placed in shared memory, and the run time hierarchy of both processes are updated to reflect this fact. The effect is to have a part of the hierarchy shared by both processes via shared memory. To set up this configuration, the spec file for `foo` is compiled into process 1. At run time (via command execution), process 1 places `foo` and `foo_rep` in a predetermined shared memory block. To gain access to `foo` from process 2, the process maps the known shared memory block into its address space and links the root of the hierarchy (in the case `foo_rep`) into its own hierarchy. Once linked, process 2 can apply any of the framework tools to `foo`.

Placing both the structure map and the run time representation in shared memory provided the ability to communicate any data structure defined in spec files between any number of processes on the same host developed using the framework. Distributed shared memory, to be described in Section 2.5, extended this capability to processes on

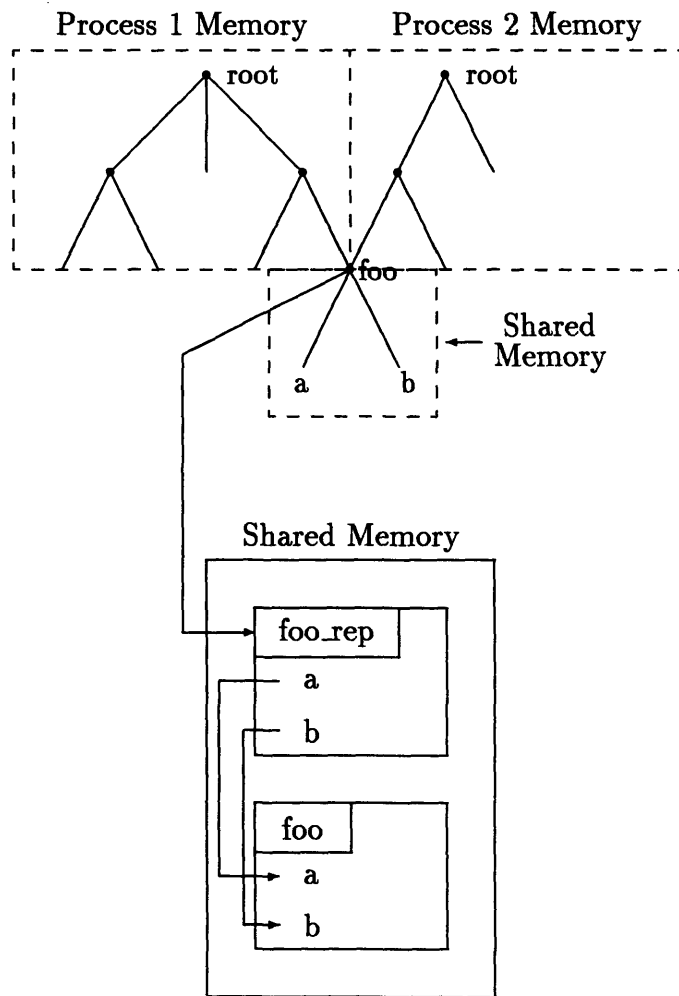


Figure 2-4: A structure and its representation in shared memory.

different hosts. Before discussing the implementation of distributed shared memory, however, it is necessary to point out some of the implementation details of placing structure maps and their representations in address space accessible by concurrent processes. These details apply to both shared and distributed shared memory, and will lay the foundation for much of the discussion of distributed shared memory.

2.4 Concurrent Memory Access

Concurrent processes that access a common address space require concurrency control to prevent simultaneous access to that address space. As an example, recall Figure 2-4. Suppose process 1 always reads `foo` and process 2 always writes `foo`. If both of these operations happen concurrently, without control, it is possible for process 1 to read the value of `a` and process 2 to then write both `a` and `b` before process 1 has a chance to read `b`. If `a` and `b` are dependent variables, (such as part of a vehicle state vector), then process 1 will have an inconsistent set of values for `a` and `b`. This is particularly a problem for simulation where the set of values in a structure correspond to values for one slice of simulation time. Reading half of the values from one time slice and the other half from another time slice usually causes serious problems in the simulation.

To solve this problem, concurrent access must be controlled. A simple mechanism to accomplish this is a system semaphore [1]. A system semaphore is managed by the operating system to control access to shared resources. To access the shared resource, a process attempts to acquire the semaphore. If it is available, the process proceeds to use the resource. If it is unavailable, the operating system stops the process and queues it to be run when the semaphore becomes available. Once a process is done using the shared resource, the semaphore is released and the resource is made available to other processes.

A typical implementation of a system semaphore is a counter in the operating system. A process acquires the semaphore by decrementing the counter and releases

the semaphore by incrementing the counter. As long as the counter is nonnegative, the process proceeds. If the process decrements the counter to a negative value, the operating system suspends the process until the counter is again nonnegative (i.e. the semaphore is released).

Initializing the counter to a value of one provides mutually exclusive access to a shared resource by several processes. Using a mutual exclusion semaphore is a simple locking strategy. Applied to shared memory, this insures that only one process at a time can either read from or write to that area of memory. This solves the access problem described in the above example. Suppose the semaphore is initialized to one. Process 1 acquires the semaphore by decrementing the counter to zero. Since the value is nonnegative, process 1 proceeds to read `a` from `foo`. After process 1 reads `a`, process 2 attempts to acquire the semaphore protecting `foo` by decrementing the counter. The resulting value is -1 so the process is suspended. Process 1 then reads `b` from `foo` and releases the semaphore by incrementing the counter. This action causes process 2 to wake and write both `a` and `b`. Process 2 then releases the semaphore by incrementing the counter. The resulting counter value is one, and the shared memory has been protected.

While a mutual exclusion semaphore can suffice for shared memory access, a more complex scheme was required for the framework in order to take advantage of the type of communication found in simulation. The problem with a mutual exclusion semaphore for simulation is that it is unnecessarily restrictive. Concurrent reading processes must be suspended even if data is not changing in shared memory. As an example, suppose there are three processes. Process 1 and process 2 read from, and process 3 writes to shared memory. If all processes are required to acquire the semaphore before accessing the shared memory, then it is possible, for process 1 to be suspended waiting for process 2 to complete its reading even though process 3 is not waiting to write into the area. It would be more efficient if process 1 and process 2 were both allowed to read concurrently as long as process 3 was not writing into

shared memory. This leads to the notion of read and write locks [2, 8].

A read lock is acquired when a process wishes to read from shared memory. A write lock is acquired when a process wishes to write to shared memory. The rules for locking are as follows:

- There are as many read locks as there are processes wishing to access shared memory.
- Only one write lock exists for a shared area.
- Acquiring the write lock prevents read locks from being acquired (as well as other write locks since there is only one write lock).
- Acquiring a read lock prevents a write lock but does not prevent another read lock from being acquired.

Implementation of this form of locking can be done with a system semaphore implemented as a counter. The counter's initial value is equal to the maximum number of processes allowed to simultaneously read the shared area. A process acquires a read lock by decrementing the counter by one and releases the lock by incrementing the counter by one. A process acquires a write lock by decrementing the counter by its initial value and releases the lock by incrementing it by that value. For example, if the counter is initialized to ten, a write lock is acquired by decrementing the counter by ten and released by incrementing by ten. As with the mutual exclusion semaphore, a process is suspended and queued if the counter becomes negative.

Read/write locking is more efficient for processes that access shared memory such that there is one writer and multiple readers by allowing multiple reads to execute concurrently. Since this sort of communication was typical for simulations built with the simulation framework, read/write locks were used for concurrency control for shared memory.

In addition to concurrency control, placing structure maps and their representations in shared memory required that all pointer references located in the shared

block were references into that same block and not into process specific address space. Figure 2-5 demonstrates the problem. Again, `foo_rep` and `foo` are located in shared memory. Recall that the framework generates type information as well as the location of each element of `foo`. Figure 2-5 shows the entry in `foo_rep` for element `a`. The `location` field is a pointer to element `a` in `foo`. Since this address is located in the shared block, both processes can successfully dereference the address to obtain the value contained in `a`.

Suppose, however, that `foo` originated in process 1 and was relocated to shared memory with *only* the `location` fields of `foo_rep` altered to point into shared memory. In this case, the `type` field for `a` in `foo_rep` would contain a pointer reference to an address in process 1's address space (location 0x5000 in this case). If process 2 attempts to use this information by dereferencing the pointer, either the operating system will signal an error because the address dereferenced does not exist in process 2's address space, or, as shown in Figure 2-5, the address *does* exist in process 2's address space, but some random information (almost certainly not the correct information) is located at that address. In this case, unpredictable results will occur.

To prevent this kind of addressing problem, it is essential that all pointers with the potential to be dereferenced from several processes hold addresses of locations in shared memory. For the framework this implied moving all supporting structures (such as type and unit information) into shared memory. This insured that any processes that had access to the shared area could dereference any address contained in that area.

2.5 Distributed Shared Memory

Distributed shared memory in the context of the simulation framework is an extension of the shared memory implementation to network communication. Figure 2-6 illustrates the basic concept for a two host configuration. In this example, the pro-

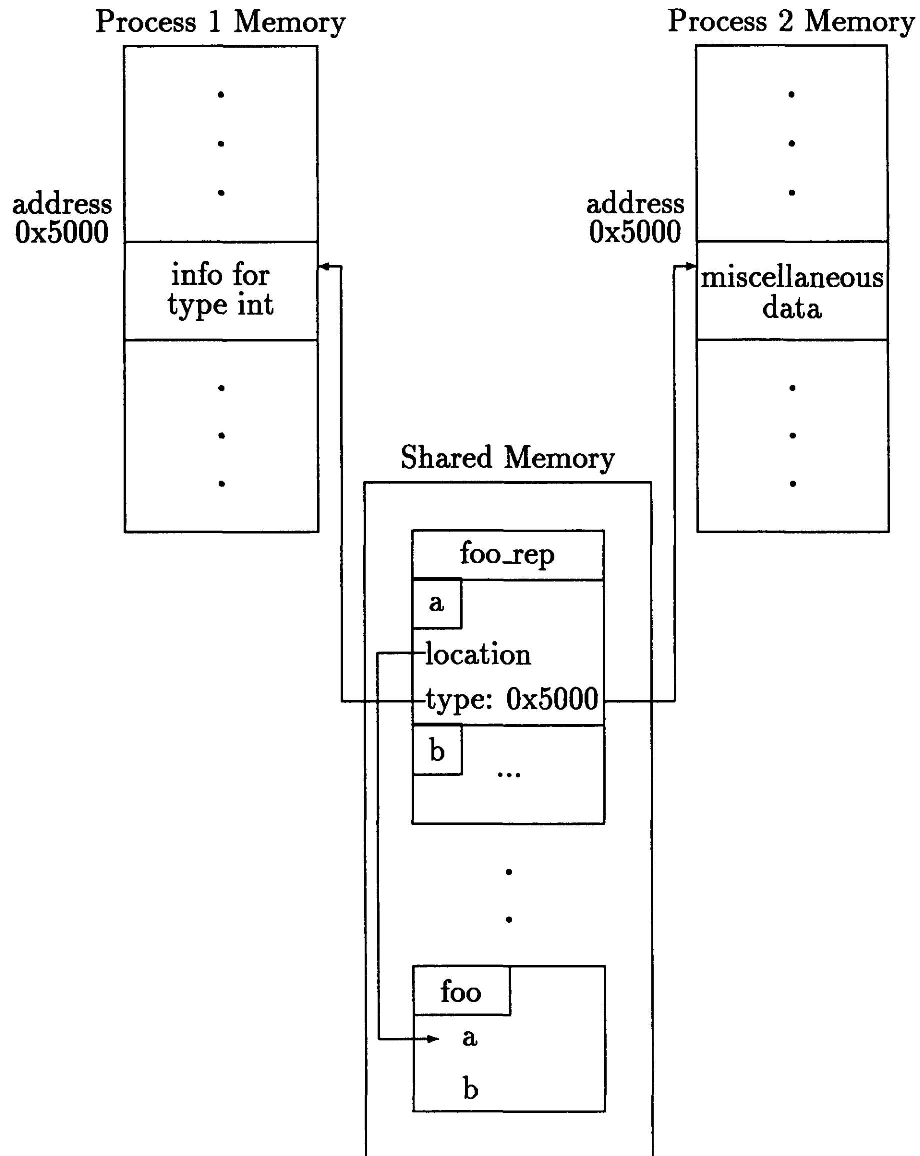


Figure 2-5: Address references from shared memory to process memory.

cesses of Figure 2-4 are located on two machines connected by a network line. On both machines, a block of shared memory exists that contains the structure `foo` and its framework representation `foo_rep`. As with the shared memory implementation, the hierarchy of both processes is altered to reflect the new location of `foo` in shared memory. The distributed shared memory implementation, however, has the added task of insuring that the shared memory on host 1 is consistent with the shared memory on host 2. In this way, access to `foo` from the perspective of either process 1 or process 2 is identical to the shared memory access previously described. The following sections describe the design and implementation of distributed shared memory.

2.5.1 Design Considerations

The simulation framework was designed to support real time hardware in the loop simulation as defined in chapter 1. In order to be a practical piece of the framework, the implementation of distributed shared memory needed to be efficient and interchangeable with the shared memory interface. Network update times needed to be as efficient as possible to meet the 100 Hz frame rates supported by the framework. Similarly, network overhead for concurrency control needed to be minimized, especially given the relative infrequency of concurrency conflicts for the kind of simulation supported by the framework. As an extension of shared memory, distributed shared memory needed to have the same programmatic interface as shared memory so that the communicating processes could be assigned to a single host or multiple hosts without recompilation.

An important design consideration in meeting these two goals was to take advantage of the uniqueness of simulation communication as it existed for the framework. These simulations generally communicate in one direction. Most applications require one writer and multiple readers (for example, the simulation as a writer, and several graphics processes as readers). Exploiting this fact both simplified the design and resulted in high efficiency for the most common configuration. Also important was

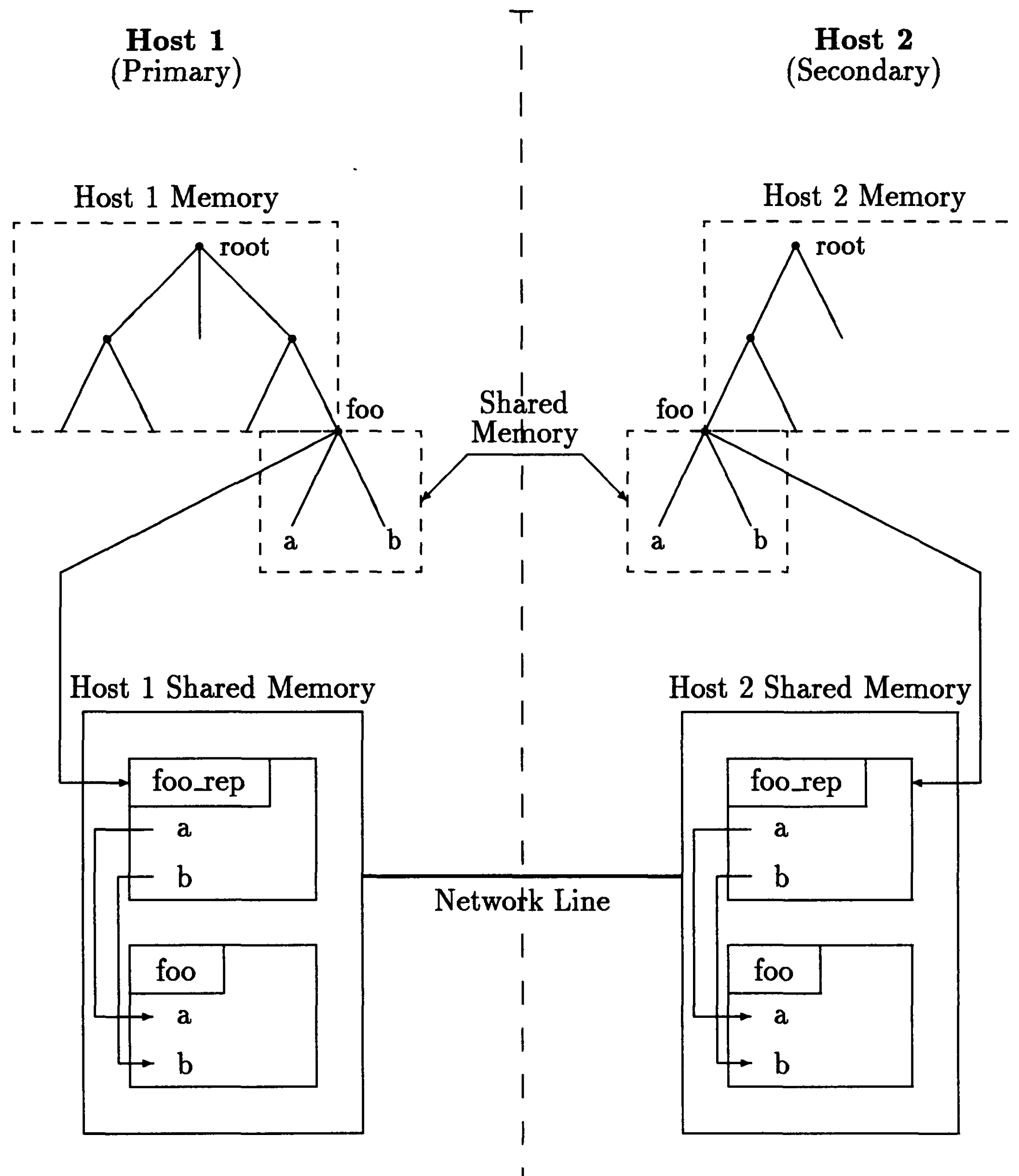


Figure 2-6: A structure and its representation in distributed shared memory.

recognizing that distributed shared memory in the context of the simulation framework need not be concerned with the types of issues related to operating system support for distributed shared memory. At the operating system level, very little is known about the kind of data stored in a shared area and the kinds of access patterns that data will have. As a result, no assumptions can be made that might increase the efficiency of the implementation. Conversely, the framework, as a tool for a particular class of simulation problems holds a lot of information that was able to be applied to the design of distributed shared memory.

2.5.2 Primary/Secondary Copy Distributed Databases

A useful perspective in the design of distributed shared memory was to view the simulation framework as a database of data structures. Structure representations in a hierarchical organization amounted to a form of database organization with each structure comprising a record in the database. Given this perspective, distributed shared memory could be viewed as a type of distributed database problem. As such, this perspective suggested a solution from distributed database technology. Specifically, a form of distributed database implementation that utilizes a primary copy of the database and several secondary copies [8]. In this type of implementation, the primary copy, by definition, is always up to date and holds the current version of the data in the database. The secondary copies are duplicates of the primary copy. Only the primary copy is allowed to be modified. Once modified, the secondary copies are sent updates to insure their consistency with the primary site. The secondary sites modify the database by first sending a lock request to the primary site, then sending the modification. Once received at the primary site, the modification is disemminated to the remaining secondary sites.

There are advantages and disadvantages associated with the primary/secondary implementation versus alternative implementations. The main advantage is locking efficiency [8]. Since all transactions go through the primary site, a lock request from

a secondary site requires communication with only the primary site[2]. Locking is even more efficient if the lock originates at the primary site since no network traffic is required to obtain the lock. If the processes at the secondary site are generally reading the database, and only the processes at the primary site are writing, there is very little network overhead for concurrency control. This fit the common configuration for simulations using the framework.

Fortunately, the disadvantages of the primary/secondary copy implementation versus other distributed database designs did not apply to the simulation framework. The main problem with the primary/secondary implementation from a database perspective is that it is vulnerable to primary site failure [8]. For database applications that require consistency over a long period of time, this can mean possible data loss. Loss of data in simulation, however, is not a problem. Simulation variables are not persistent, and simulations are designed to be repeatable. As a result, all simulation data placed in distributed shared memory can be reproduced by simply rerunning the simulation. If the primary site fails, it is generally a simple matter to restart the simulation using a different primary site.

Given the efficiency and lack of applicable disadvantages with respect to the alternatives, the primary/secondary site distributed database became the paradigm for the distributed shared memory design in the framework. Figure 2-7 illustrates a typical configuration. One host acts as the primary site with a shared memory block on that site comprising the primary copy of the area. Other hosts connected to the area are secondary sites each holding a copy of the primary area in shared memory. A process at the primary site satisfies secondary lock requests and sends network updates to the secondary sites. A process on each secondary host receives updates from the primary site and places them in shared memory to insure consistency with the primary copy. Details of the locking protocol for concurrency control and the update procedure are contained in the following two sections.

[2]Other, more complex locking protocols require communication between several sites.

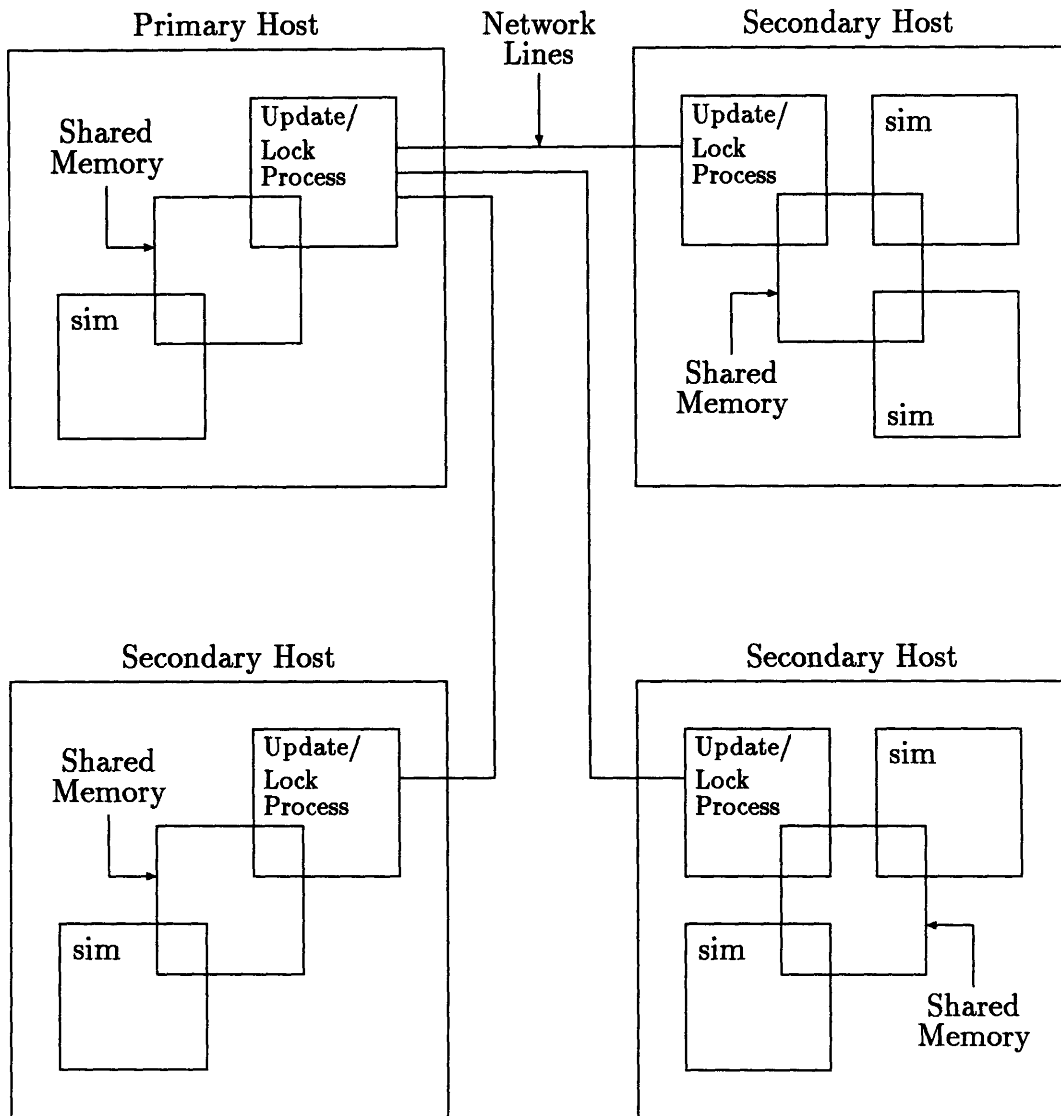


Figure 2-7: Primary/Secondary organization for distributed shared memory.

2.5.3 Concurrency Control

The read/write locks described in Section 2.4 for shared memory required modification when applied to distributed shared memory. The primary/secondary organization for a distributed database requires that lock requests be made through the primary site. Making requests to the primary site will insure that the secondary sites are always consistent with the primary. This form of read/write locking, however, is potentially inefficient, especially for processes that can relax the consistency constraint.

To understand the inefficiency, recall the configuration of Figure 2-7. If read locks must be acquired from the primary host, then a write lock on the primary host will cause all secondary host processes requesting a read lock to suspend operation until the write lock at the primary site is released. This is inefficient for two reasons. First, a read lock request requires network communication to the primary site which can be slow. In addition, requiring consistency between the primary and secondary copies at all times is not generally required for simulation framework applications.

Recall that for shared memory the goal of concurrency control is to insure that the data contained in a shared block always comprise a self consistent set. Extending this constraint to distributed shared memory implies that each copy of the shared area always comprise a self consistent set and that they are all the same set. This is an unnecessarily tight constraint for simulation. The reason for this is that synchronization of network processes with each other is not usually necessary as each process requires only one set of data in distributed shared memory to operate.

For example, suppose the primary host is executing a vehicle simulation, and a secondary host is displaying the vehicle. The only data required to display the vehicle would be the state vector of the vehicle. It is not necessary that the display always have the state vector for the same time step as the simulation. It is only necessary that the data in distributed shared memory be a complete state vector for a given time step. For this reason it is possible to relax the consistency constraint by requiring that each copy of the shared area must always comprise a self consistent set *locally.*

The relaxed constraint, however, does not require that each copy be the same at all times[3].

In order to take advantage of the fact that the consistency constraint could be relaxed for most applications, a third type of lock was added to the locking mechanism — the network lock. The network lock is a lock at the primary site that is requested by a secondary site before requesting a write lock. It is used to signify that a write to a secondary copy is about to take place and that an update from the secondary site will require a write lock at the primary site in the future. The network lock rules added to the read/write lock rules are:

- There is one network lock and it resides at the primary site.
- Acquiring the write lock at the primary site requires acquiring the network lock first (whether or not the request comes from the primary or secondary site).
- Acquiring a network lock from a secondary site prevents acquisition of a write lock at the primary site, and prevents acquisition of the network lock by another secondary site, but does not prevent read locks at any of the sites.
- Obtaining a write lock at the primary site prevents acquisition of the network lock and prevents read locks at the primary site, but does not prevent read locks at secondary sites.

The network lock is the only lock that is specific to the primary site. Each site has its own local read and write locks to protect the shared memory blocks. Figure 2-8 illustrates the procedure for obtaining read and write locks at both primary and secondary sites. At both primary and secondary sites a read lock is acquired by acquiring the local read lock. A write lock request from a process at the primary site is obtained by first acquiring the network lock then the local write lock. Once data is written in the primary copy, updates are sent to the secondary sites.

[3]Section 2.5.5 describes support for applications that cannot relax the consistency constraint.

Primary Site Lock Requests	Secondary Site Lock Requests
Read Lock:	Read Lock:
Acquire local read lock.	Acquire local read lock.
Read data.	Read data.
Release local read lock.	Release local read lock.
Write lock:	Write lock:
Acquire network lock.	Acquire primary network lock.
Acquire local write lock.	Acquire local write lock.
Write data.	Write data.
Update secondaries.	Update primary site.
Release local write lock.	Release local write lock.
Release network lock.	Release primary network lock.

Figure 2-8: Primary and secondary site lock procedures.

At a secondary site, a write lock is obtained by first requesting the network lock from the primary site, then acquiring the local (secondary) write lock. Once the data is written to the secondary copy, it is sent back to the primary site where it is placed in shared memory by obtaining the primary site write lock, writing the data, and updating the other secondary copies.

The efficiency of the read/write/network lock strategy comes from the fact read locks at one site do not affect other sites and do not require network communication. A write lock at the primary site prevents read locks at that site and prevents network locks, but does not prevent read locks at the secondary sites. This allows secondary site processes requiring only read locks to continue processing during a primary site write lock. Similarly, the network lock does not prevent read locks at the primary site which allows primary site processes to continue execution without waiting for the pending update from the secondary holding the network lock.

The cost of this strategy is the fact the the primary and secondary copies are not *guaranteed* to be identical to each other at all times. This is a direct result of relaxing the consistency constraint. It is possible for a secondary site to read its copy while the primary site changes its copy. This is slightly different behavior than simple shared memory. For shared memory, it is not possible for any other process to perform a read while a write is being performed (as long as read/write locking is used). If this were allowed, the reading process might get inconsistent data from the shared block. For simulation this usually means reading the shared block and getting half of the data from one time step and half from another.

Since the secondary sites in distributed shared memory actually have physically different copies, however, this is not a problem. As long as local read/write locks insure the self-consistency of each shared block, the penalty for changing the primary site in parallel with reading a secondary site is a delay in receiving the information at the secondary site. This is a penalty only if the application requires that distributed shared memory act *exactly* like shared memory. In this case, allowing a read while the primary site was being written would be viewed as reading a bad value since, in simple shared memory, once the write has occurred, it is not possible to read the old value.

For the simulations built with the framework, however, it was sufficient to insure that local shared areas were self consistent without requiring that they be identical to the primary copy before allowing a read. For the framework implementation, read/write locking was used to insure local self-consistency, and the network lock in conjunction with the primary site write lock procedure was used to insure that the updates reached all sites.

2.5.4 Updates

The primary/secondary copy implementation required a procedure to update secondary copies with primary copy changes. There were several possibilities. One was

to send an entire copy of the shared area to each secondary when a primary site write lock was released. While this is a simple solution, it carries a significant efficiency cost as large amounts of data, most of which already exists at the secondary sites, need to be communicated over the network every simulation time step.

Another alternative was to create a mechanism that detected the changes made to the shared area after releasing a write lock and send only the changes in updates to the secondary sites. While this is more efficient than sending the entire area every time step, it is still inefficient because there is no way for a general facility like the framework to predict the changes made to a shared area as they differ from simulation to simulation. As a result, only a memory comparison function executed every simulation time step would suffice. If the shared area is relatively large, this is a very expensive operation.

The solution used for the distributed shared memory implementation was to require writing processes to specify the changes made to the shared area. Once specified, the framework sends the updates to the secondary sites. This solution has the disadvantage of requiring extra bookkeeping on the part of the user, however it has several advantages that need to be weighed against this disadvantage.

The primary advantage is efficiency. The user is in the best position to decide what changes were made between the acquisition and release of a write lock since the user must specify those changes to begin with. The user can generally identify the difference from time step to time step without doing a memory comparison on the entire area and without sending the entire area over the network.

Similarly, the user can decide *when* to update the data. The user can take advantage of the relaxed consistency constraint for the shared area at the primary site without immediately sending an update. This minimizes network traffic by reducing the update frequency to only that required by the user. It also reduces memory requirements because the user can use the shared memory area without keeping a local working copy.

Finally, the framework run time data structure representations minimize the bookkeeping overhead required on the part of the user. Since there is both size and location information of every data structure located in the shared area, all that is required for update bookkeeping is a pointer to the framework representation of each data structure needing an update[4]. To perform an update, only the pointers of the representations need to be passed to the framework. The framework then uses that information to send the actual structure and its identifier to the secondary sites. At the secondary site the identifier is used to locate the local framework representation. Once found, the size and location of the incoming data is known, and the update is performed at the secondary site. The network update overhead per data structure is only the one word used as the identifier — the rest of the update is modified data.

2.5.5 Synchronization

Synchronization in the context of the simulation framework is the process of linking one simulation's time steps with another. This is necessary in the case of distributed simulation. Suppose two processes are responsible for part of the computation of a simulation time step. Suppose further that process 2 needs to wait for process 1 to finish (so that it can get some data from that process) but process 1 does not need to wait for data from process 2. If the data is communicated through shared memory then process 1 is a writer and process 2 is a reader.

Figure 2-9 illustrates three possible synchronization states for the two processes. Each plot shows process 1 on the top and process 2 on the bottom. Process 1 writes and process 2 reads are shown versus time. Also shown are the simulation time steps (labeled t_0, t_1, etc). Note that reads are generally performed at the beginning of the time step while writes are done at the end of the time step. The top plot shows process 1 and process 2 running asynchronously. One problem shown is that process 1

[4]It is also possible to keep pointers to individual fields within a data structure, but this is somewhat less efficient in practice.

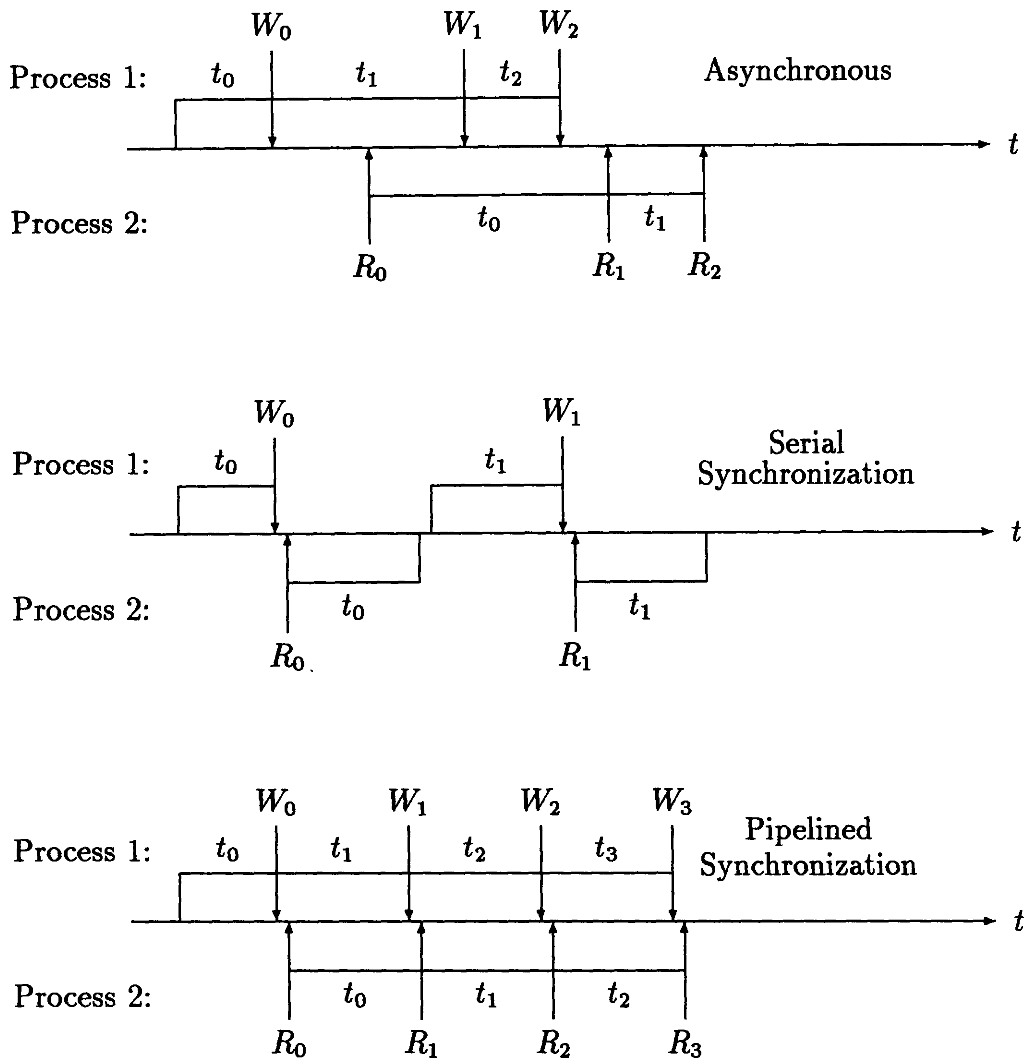
can perform two writes (W_1 and W_2) in between two process 2 reads (R_0 and R_1). Synchronous operation requires that there be one read for each write.

The second plot shows one solution to this problem. Process one is synchronized with process two such that they are running serially. This means that process 1 executes time step t_0 while process 2 waits. Then process 2 executes its part of time step t_0 while process 1 waits. In this way there is one read for each write and the simulation time steps are synchronized. This is not a particularly efficient solution. Since process 1 does not rely on data from process 2 there is no reason for it to wait for process 2 to complete its time step before performing the next one (although we still do not want it to execute two time steps before process 2 completes one). As a result, it is possible for both processes to operate synchronously, but in parallel.

This solution is a form of pipelining commonly used in microprocessors to parallelize instruction execution [9]. Plot three illustrates the result. Process 1 performs its part of time step t_0 while process 2 waits. Once complete, process 2 starts its part of time step t_0. At the same time, process 2 begins execution of time step t_1. From then on, both processes perform their part of a time step in parallel. Note that there is still only one read for every write, and that the two processes are synchronized.

Figure 2-10 illustrates the same problem using distributed shared memory for interprocess communication. In this case, process 1 is on the primary host and process two is on a secondary host. The plots again indicate reads, writes, and time steps for each process. However, unlike simple shared memory, there is the added difference that a write at the primary site does not reach the secondary site immediately. As a result, a write operation has an associated update operation that occurs some time after the write.

The top plot shows the two processes running asynchronously. As before, there is still the problem of two writes (W_1 and W_2) in between two process 2 reads (this time R_1 and R_2). There is also an added problem — a process 2 read (R_0) between a process 1 write and its update (W_0 and U_0). This is a problem because process 2 is



t_n : time step n

W_n : Write to shared memory for time step n

R_n : Read from shared memory for time step n

Figure 2-9: Synchronization of simulation time steps.

essentially reading an old value with respect to the primary site from its copy of the shared area. This is a direct result of relaxing the consistency constraint. If process 2 and process 1 are really asynchronous (as in the case of a simulation and a hardware process) this is not an issue. If, however, the goal is to have only one read for each write then reading between a write and its update must be prevented.

Fortunately, synchronizing the two processes solves this problem. Plot two shows process 1 and process 2 utilizing pipelined synchronization. The key to the synchronization is the fact that process 2 waits for an update from the primary site before proceeding. Process 1 executes time step t_0 while process 2 waits for an update. Once done, process 1 sends the update. Once received, process 2 executes its part of time step t_0 while process 1 begins time step t_1 in parallel.

Note that because process 2 waits on the *updates* from the primary site, the consistency constraint is no longer relaxed and both the primary and secondary sites are effectively the same from the perspective of reads and writes. The effect of relaxing the consistency constraint was to allow a read to be performed between a write and its update. From the point of view of shared memory, this is not possible since reads and updates are identical operations. If this is prevented by allowing reads to occur only after an update, however, then the effects of distributed shared memory and shared memory are the same.

The implementation of synchronization in the framework was done in the update process at the secondary site. The processes interested in synchronization register their process identifiers with the update process and suspend themselves. When an update is received at the secondary site by the update process, all waiting processes are awakened and the update process waits until they are finished before accepting another primary site update. Through this mechanism, both synchronization and inter-host consistency are insured.

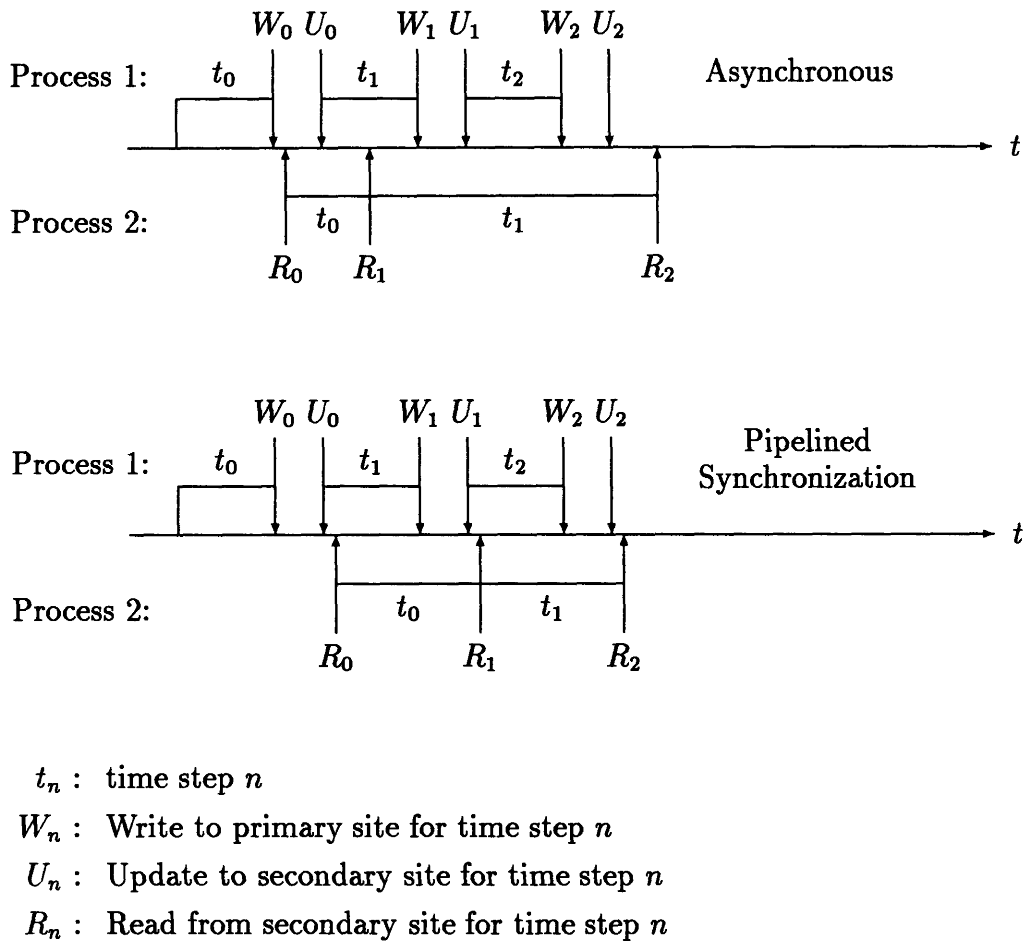


Figure 2-10: Synchronization of simulation time steps using distributed shared memory.

Chapter 3

Examples and Results

This chapter provides two examples and a performance evaluation of direct memory access communication. The first example uses direct memory access for hardware communication and distributed shared memory for network communication in a hardware in the loop simulation. The second example uses shared memory and distributed shared memory to communicate information to local and remote display processes in a simulation hosted on several machines. The chapter closes with an evaluation of the general network performance of distributed shared memory.

3.1 Example 1 – Gantry Limit Switches

This example demonstrates the use of direct memory access techniques for device and network communication in a hardware in the loop simulation developed under the Proximity Operations research and development project at Draper Laboratory. The goal of the project is to develop image processing techniques to enable an autonomous underwater vehicle to locate objects on the ocean floor. The project includes a simulation of an underwater vehicle and environment that is used to drive a six degree of freedom gantry. The six degrees of freedom consist of three translational (x, y, and z) and three rotational (yaw, pitch, and roll) axes. Attached to the gantry is a camera

that is used to generate images from a scale model of the underwater terrain. The simulation models the dynamics of the underwater vehicle and computes the position and orientation of a camera attached to the simulated vehicle. Once the simulated camera state for each time step is computed, the gantry positions the real camera to the equivalent point in the gantry's range of motion to match that state. Images taken by the camera on the gantry are used by the simulation to estimate of the likelihood of a target in the camera's field of view. An estimate of the target's range is also produced. The simulation control software uses this information to guide the simulated vehicle to a location more suitable for identification. The new position is again realized by the gantry and another image is taken and evaluated.

The simulation was developed using the simulation framework described in Chapter 2. The hardware interfaced to the simulation includes the 6 gantry motors and their controllers (one for each degree of freedom), 6 serial ports for communication with the controllers, 2 analog inputs for position feedback, 48 discrete inputs to and 46 discrete outputs from the controllers, a timer board used as a real time clock, a CCD camera, and a video frame grabber. The host computer is a Silicon Graphics IRIS 4D/440GTX with 4 processors.

This example will focus on monitoring the state of the gantry limit switches using distributed shared memory. The gantry has 14 limit switches each used to indicate a position on one of the gantry's axes. Each of the three translational axes has three switches — one at the positive limit, one at the negative limit, and one at the center or zero position. The rotational switches consist of one limit switch indicating the zero position for the yaw axis and a pair of switches indicating the clockwise and counterclockwise limits for pitch and roll. When the camera on the gantry is oriented such that the the gantry travels to a limit switch position, the switch state is toggled.

Figure 3-1 shows a subset of the hardware configuration used for this example. The limit switches for each axis are connected to the controller for that axis. The controller can be programmed to shut its motor down when one of the limit switches

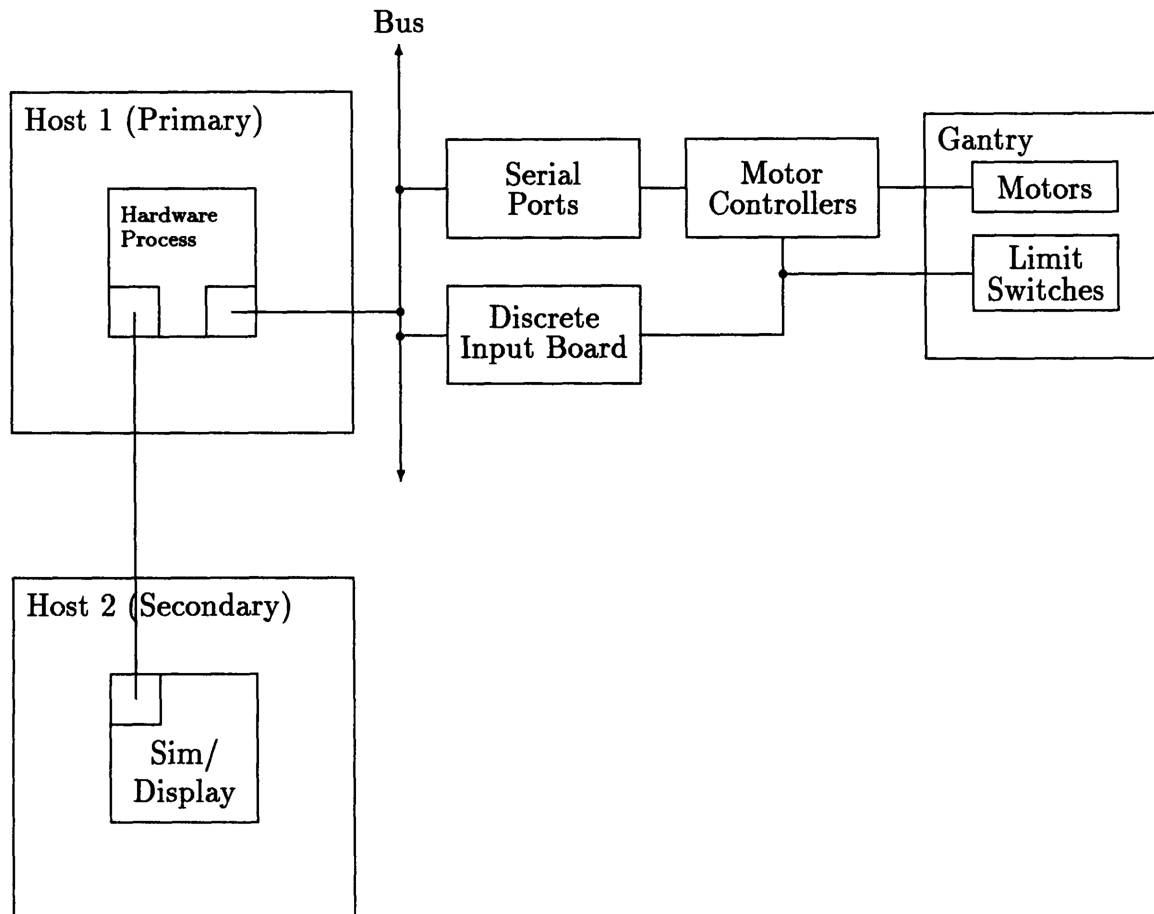


Figure 3-1: Gantry hardware configuration.

changes state. The switches are also connected to a discrete input board that relays the limit switch state to the simulation on the host computer through a structure map locates at the bus address of the input board. The controllers are connected to the host via serial ports which are also controlled using structure maps. While it was possible to read the state of the limit switches through serial port commands to the controllers, the state could be read in much less time directly from the input discrete board. This allowed the limits to be continuously monitored without interfering with host to controller communication.

For this example, direct memory access in the form of a structure map was used to communicate with the input discrete board. Figure 3-2 shows the framework's

```
%Dir discrete_in_ref
{
 short   idu        hex    0     : Board ID Register;
 short   csr        hex    0     : Control/Status Register;
 short   unused[6]  hex    0     : Unused;
 short   dr[8]      hex    0     : Data Register (Discrete Inputs);
} discrete_in;
```

Figure 3-2: Specification of register map for dicrete input board.

structure map defining the register organization of the board. This structure was relocated to the input board's bus address in order to communicate between the hardware process and the input board. Communication with the board was done by writing the appropriate bits in the Control/Status register (the `csr` field of the `discrete_in` structure) to command the board to read the inputs. The value of each discrete is placed in the Data Registers (the `dr[8]` fields) by the input board. Each bit of the Data Registers corresponds to one discrete input.

Figure 3-3 shows the structure used by the simulation to monitor the input discretes. The hardware process of Figure 3-1 continuously reads the input discretes, through the `discrete_in` structure map, rearranges the bits of the Data Registers, and places the results in the `mtr_discretes` structure. The bits are rearranged such that each element of the `din` field in `mtr_discretes` corresponds to one motor's set of discrete inputs. The discrete outputs for each motor are stored in the `dout` fields. Finally, a composite of the inputs and outputs are stored in the `discretes` field for each motor.

For this example, the `mtr_discretes` structure was used for network communication by placing it and its framework representation in distributed shared memory as described in Chapter 2. Communication takes place when the hardware process on host 1 write locks the distributed shared memory area, writes its rearranged version of the input discretes (read from the input discrete board) into the `mtr_discretes` structure residing in the shared area, and unlocks the area causing an update to be

```
%Dir mtr_discretes_ref
{
 int din[NUM_MOTORS]       hex  {0}     : I/O discrete ins;
 int dout[NUM_MOTORS]      hex  {0}     : I/O discrete outs;
 int discretes[NUM_MOTORS] hex  {0}     : I/O discretes;
} mtr_discretes;
```

Figure 3-3: Specification for motor discretes.

sent to the simulation on host 2.

A graphical display was used for this example to indicate the state of the discretes by representing each bit as a light on an LED panel. The LED panel was updated at a 60 Hz rate. This panel is also used in the full simulation as a means of monitoring the state of the gantry. As an exercise of both the hardware access and distributed shared memory, three tests were performed:

- Host 1 displayed the LED panel and was interfaced to the hardware via bus relocated structure map. Limit switches were manually toggled while monitoring the LED panel. Host 2 was not involved.

- The LED panel was moved to host 2 and the `mtr_discretes` structure was placed in distributed shared memory. Host 1 continued to interact with the hardware. Again limit switched were manually toggled while the LED panel on host 2 was monitored.

- Limit switches were simulated by disabling the hardware and driving the `mtr_discretes` structure directly in distributed shared memory from a simulation on host 1. Again, the switch states were monitored via the LED panel on host 2.

Both the hardware access mechanism and distributed shared memory were successful in all three tests verifying that the two mechanisms functioned as expected.

Number of time steps	Execution time (seconds)	Frequency (Hz)
500000	7.09	70,520
500000	7.12	70,220
500000	7.14	70,030
500000	7.15	69,930
Average Frequency		70,175

Table 3.1: Execution times for example one without hardware access and without distributed shared memory.

Once verified, the performance of both hardware access and distributed shared memory were tested.

To test the performance of the hardware access, only the hardware process of Figure 3-1 was used. Two test were done. The first test measured the speed of the hardware process without any hardware access[1]. Access to hardware was disabled by leaving the `discrete_in` structure in user memory rather than relocating it to a bus address. Table 3.1 indicates the results of running 500,000 steps of the hardware process. As indicated, the average frequency was about 70 kHz. The second test measured the speed of the hardware process with hardware access enabled. The test was performed by mapping the `discrete_in` structure to its bus address to re-enable hardware access. Table 3.2 indicates the results of running several tests with this configuration. The average frequency obtained was 12 kHz.

The time to access a bus location using structure maps can be computed from the two test results. The average time for one step of the hardware process without bus access was 1/70175 or 14.25 microseconds. The average step time with bus access was 1/12100 or 82.64 microseconds. This means hardware access accounted for 72.39 microseconds per step. For this example, there were five words read from the bus each step resulting in an average time of 14.48 microseconds for each bus access.

[1]The framework provides support for measuring the execution time of a process.

Number of time steps	Execution time (seconds)	Frequency (Hz)
80000	6.61	12,100
80000	6.64	12,050
80000	6.65	12,030
160000	13.18	12,140
160000	13.22	12,100
160000	13.23	12,090
320000	26.30	12,170
320000	26.32	12,160
320000	26.43	12,100
Average Frequency		12,100

Table 3.2: Execution times for example one with hardware access but without distributed shared memory.

To test the performance of distributed shared memory for example one, the simulation on host 2 was run with the LED panel disabled (this was necessary because the panel is synchronized with the scan rate of the monitor which would fix the simulation rate at 60 Hz). The host 2 simulation included only reads from distributed shared memory — model computation was disabled to isolate the performance of distributed shared memory. The hardware process was again run on host 1 with hardware access enabled. The hardware process communicated the `mtr_discretes` structure to host 2 using distributed shared memory. Both processes were synchronized using the pipelined synchronization technique described in Chapter 2. Table 3.3 shows the results of several tests. The average frequency of this configuration was about 390 Hz. This represents the frequency limit of distributed shared memory for the two machines involved in the test.

This example demonstrated that direct memory access applied to hardware communication is very efficient. Recall from Chapter 1 that the frequency requirement for simulations built using the framework was 100 Hz. The above results indicate that this rate is easily obtainable for a hardware in the loop simulation residing on

Number of time steps	Execution time (seconds)	Frequency (Hz)
4000	10.32	387.6
4000	10.49	381.3
4000	10.51	380.6
8000	20.52	389.9
8000	20.89	382.9
8000	20.70	386.5
16000	40.80	392.2
16000	40.89	391.3
16000	40.92	391.0
Average Frequency		387.0

Table 3.3: Execution times for example one with both hardware access and distributed shared memory.

the same machine as the attached hardware. Since bus access times are on the order of microseconds, hardware communication is a very small part of the 10 millisecond time step of a 100 Hz simulation. Similarly, these results indicate that a hardware in the loop simulation using direct memory access techniques can keep up with hardware operating at rates in the several kilohertz range.

This example also demonstrated that it is possible to run a hardware in the loop simulation on a machine *not* connected directly to the hardware using distributed shared memory. Since the maximum frequency of 390 Hz for this example exceeded the 100 Hz requirement by nearly a factor of four, it is possible to monitor hardware remotely, communicate the results using distributed shared memory, and still meet the 100 Hz requirement. Section 3.3 will show that by using faster machines the maximum frequency can be pushed back into the kilohertz range which can further reduce the difference between local and remote computation. Section 3.3 will also show that as long as the communicated data size remains below about 1 Kbyte, the maximum simulation frequency will be nearly independent of the amount of data communicated.

3.2 Example 2 – Simulation of an Autonomous Underwater Vehicle

This example demonstrates the use of distributed shared memory as a tool for communicating between simulations hosted on several machines. Figure 3-4 shows the configuration used for the second example. Two types of processes are involved — a vehicle simulation and a display process. Host 1 runs a simulation of Draper Laboratory's Unmanned Underwater Vehicle (UUV). The simulation models the underwater environment, the vehicle's motion in that environment, and on board guidance, navigation, and control. Also on host 1 is the display process that provides a three dimensional perspective view of the UUV and its environment. For this example, a copy of the display process was also run on host 2 and host 3. All three hosts communicated using distributed shared memory. Since the display process on host 1 was on the same machine as the simulation, communication was through the shared memory block on host 1. The other two hosts received updates from host 1. Note that while the display processes were identical executables, only the vehicle state was shared with the simulation. Since each display process controlled its own viewing perspective, each host was able to display an independent view of the vehicle simultaneously.

Figure 3-5 and Figure 3-6 define the structures used to communicate between the simulation and the display processes. The structure `uuvdraw_buf` of Figure 3-5 contains the state of the vehicle for simulated time step `t`. This information, generated by the simulation, includes the position and orientation of the vehicle in the environment, as well as the state of the fins and main propeller[2]. Figure 3-6 shows the `calc_view` structure that defines the eye position of the viewer. Each display process decides whether or not to use the `calc_view` structure for its viewing position. If all of the processes use the structure, they all display the vehicle from the

[2]The other fields in the structure are used for different displays.

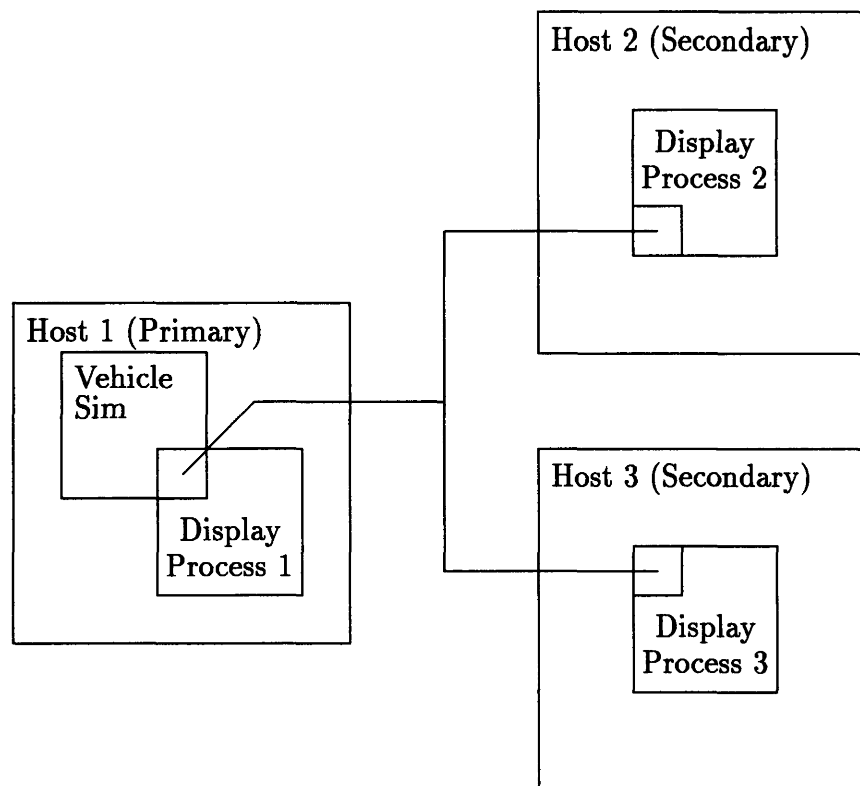


Figure 3-4: Network and process configuration for unmanned undersea vehicle test.

same perspective. Display processes not using this structure define their eye positions independently.

Communication between the vehicle simulation and display processes occurs for each time step of the vehicle simulation. Communication for a time step is initiated with a write lock on distributed shared memory on the part of the vehicle simulation. Both the `uuvdraw_buf` and `calc_view` structures are updated in the shared area on host 1 with the state of the vehicle at the current simulation time step. Once the simulation releases the write lock on the area, an update is sent to hosts 2 and 3. Each display process then accesses the `uuvdraw_buf` and `calc_view` structures in the shared area of the corresponding host and redraws the display to reflect the new vehicle state.

This example was run on three Silicon Graphics workstations connected on an Ethernet network. The simulation was run at 60 time steps per second. The simulation time step used was 0.05 seconds so that simulation time elapsed three times faster than real time. Each display process had a 60 Hz update rate. During execution, commands were issued to the vehicle control system while the viewing perspective of each display was held constant. Once it was verified that all three displays were functioning properly, the viewing perspective of each display was changed independently. In all cases, the displays reflected a consistent vehicle state from different viewing perspectives.

Once the configuration was functioning, the performance of distributed shared memory for this application was tested. Recall that the performance goal of example one was to communicate a fixed amount of data between machines on the network at as high a rate as possible. For example two, however, the frequency of the display process was fixed at 60 Hz which is the scan rate of the monitors used. There was no point in exceeding this frequency since it is not possible to display any more than 60 frames per second on the display hardware. The performance goal for this example, was to be able to utilize as much information as possible from the simulation on host 1

```
%Dir uuvdraw_buf_ref
{
 int    update_ctr      na      : buffer counter;

 double t               sec     : sim time;
 int    crash           sw      : TRUE=vehicle has crashed;

 float  x               ft      : vehicle position;
 float  y               ft      : vehicle position;
 float  z               ft      : vehicle position;
 float  phi             deg     : vehicle orientation;
 float  theta           deg     : vehicle orientation;
 float  psi             deg     : vehicle orientation;

 float  alt             ft      : vehicle altitude;
 float  zdot            ft/sec  : depth rate;
 float  uf              ft/sec  : vehicle fwd speed w/rt fluid;
 float  psidot          deg/sec : vehicle heading rate;

 float  df_top          deg     : top fin angle;
 float  df_botm         deg     : botm fin angle;
 float  df_port         deg     : port fin angle;
 float  df_stbd         deg     : stbd fin angle;

 float  da              deg     : composite aileron;
 float  dr              deg     : composite rudder;
 float  ds              deg     : composite sternplane;

 float  prop_rpm        rpm     : main prop rpm;
 float  prop_angle      deg     : main prop angle;
} uuvdraw_buf;
```

Figure 3-5: Specification of vehicle state information for display.

```
%Dir calc_view_ref
{
 double t            na    : sim time;
 int    view_mode    na    : calculated view mode;
 int    view_index   na    : calculated view index;
 float  view_x       ft    : calculated view x;
 float  view_y       ft    : calculated view y;
 float  view_z       ft    : calculated view z;
 float  view_roll    deg   : calculated view roll;
 float  view_tilt    deg   : calculated view pitch;
 float  view_pan     deg   : calculated view yaw;
} calc_view;
```

Figure 3-6: Specification for calculated viewing option.

in the display processes on the remote hosts.

To measure the performance of distributed shared memory for this configuration, several test were made. For each test, 6000 time steps of the vehicle simulation were run at 60 Hz real time. The data size communicated using distributed shared memory was varied for each test. The execution time was measured and the resulting display frequency computed[3]. Figure 3-7 plots the resulting frequency versus communicated data size for these tests. The plot shows that distributed shared memory was able to accommodate data sizes of up to 16 Kbytes for each time step at 60 Hz. This means that roughly 120 times more data than was used in the original example could be sent providing much more information about the simulation running on host 1. Figure 3-7 also indicates that for data sizes greater than 16 Kbytes, the 60 Hz frequency could not be maintained. Section 3.3 will show that for large data sizes performance decreases roughly linearly with increasing data size due to network throughput limits.

This example demonstrated that distributed shared memory could be used to distribute a simulation and its display over several networked machines. In addition, it demonstrated that distributed shared memory could be added to an existing simula-

[3]See Appendix A for complete test data.

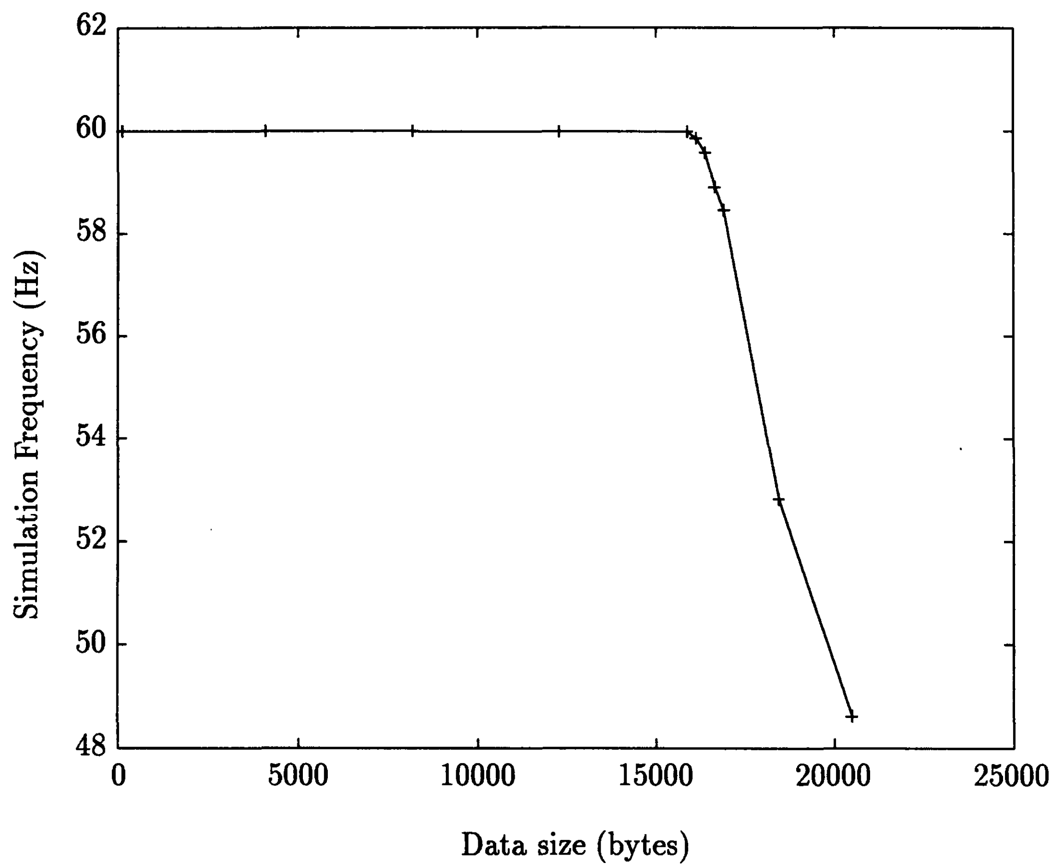


Figure 3-7: Simulation frequency as a funtion of data size for example two.

tion in order to improve its capability. Both the simulation and the display processes used in this example existed before the development of distributed shared memory for the simulation framework. Both programs received minor modifications to use distributed shared memory. As a result, a simulation that previously provided only one perspective of the vehicle was able to provide several different views simultaneously with the addition of distributed shared memory. The fact that the views could reside on different hosts meant that more screen space was available for display. Furthermore, the performance of distributed shared memory implies that the display process can be upgraded to display much more information about the vehicle being simulated. The fact that the display process can be executed on a machine other than the simulation host means that it need not contend with the simulation for access to the same computational resources.

3.3 Distributed Shared Memory Network Performance

The previous two sections discussed the performance of distributed shared memory in the context of particular simulations. This section isolates the network communication component of distributed shared memory in order to measure peak performance. Two Silicon Graphics Indigo 2 workstations connected via Ethernet on an isolated network were used to communicate data between a simulation on each host containing only distributed shared memory operations and no model computation. A discussion of the test results will relate the size of the communicated data to the maximum simulation frequency obtainable using distributed shared memory as a measure of network performance.

The examples provided in this chapter have demonstrated that the simulation framework's implementation of distributed shared memory was able to support both the data rate requirements of a hardware in the loop simulation and the data size requirements of a 60 Hz distributed simulation. In both cases, a relatively small amount of data was communicated in the initial demonstration. Example one communicated

48 bytes while example two communicated 136 bytes of data[4]. The fact that the amount of data in these examples was small, however, also demonstrates that significant capability can be gained without the need for communicating large amounts of data. This was especially true for the second example where the communication of 136 bytes of data was enough to support simultaneous multiple views of the vehicle, updated 60 times a second, that had not previously existed. It is useful to keep this perspective in mind when evaluating the general performance of distributed shared memory.

Two tests were tests were used to evaluate the performance of distributed shared memory in the simulation framework. The first test was designed to measure the maximum simulation frequency (the number of time steps executed per second) of the framework without model computation and network overhead. The test consisted of a baseline simulation built with the framework that executed simulation time steps that performed no model computation and no network communication. The baseline simulation was run on one machine for 50,000 time steps and the resulting frequency was computed by dividing the number of time steps by the total execution time. The resulting frequency was approximately 105 kHz.

The second test was designed to measure the effect of varying the data size communicated over the network on the simulation frequency. The baseline simulation was modified to communicate data using distributed shared memory. This new simulation was placed on both machines and synchronized using the pipelined synchronization scheme described in Chapter 2. The simulations were run for several thousand time steps with varying data size and the resulting simulation frequencies were again computed by dividing the number of time steps by the total execution time[5].

Figure 3-8 shows the resulting data along side a plot of the maximum simulation frequency possible if the theoretical Ethernet transfer rate of 10 Mbits per second

[4]Recall that these sizes reflect only the *changing* data from time step to time step.
[5]See Appendix A for a complete data set.

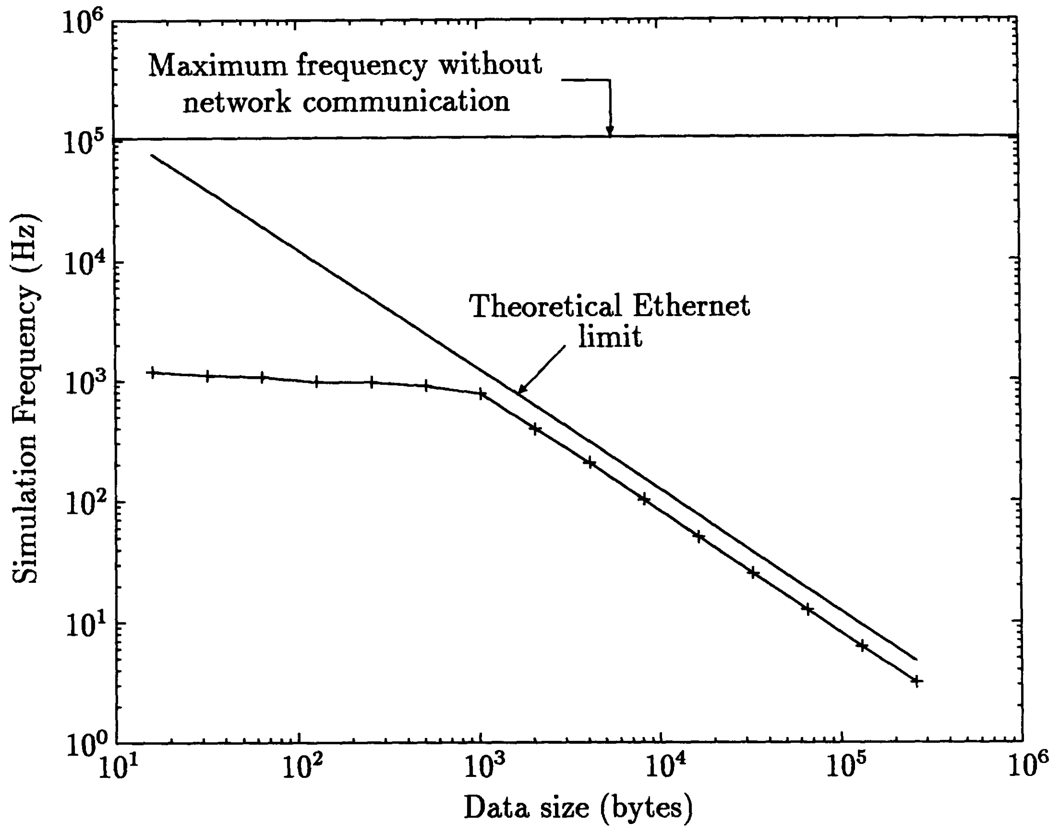


Figure 3-8: Maximum simulation frequency as a funtion of data size.

could be attained. Each data point (marked with a '+') indicates the average simulation frequency attained for a given data size. The top line of the plot shows the frequency of the baseline simulation for reference. It is important to note that data of this kind is system dependent. However, while the magnitude of the data may vary widely from system to system, the shape of the data should not.

The test results indicate that the maximum simulation frequency is nearly constant until the data size reaches about 1 Kbyte. The reason for this is that there is a fixed amount of operating system overhead associated with a network transfer. This overhead occurs from the time the primary simulation initiates the data transfer (by making an operating system call) to the time the operating system puts the data on

the Ethernet line. For small data sizes this time is more significant than the Ethernet transfer time. However, as the data size grows, the Ethernet transfer time is more significant than the operating system overhead. The data size for which the Ethernet transfer time becomes more significant than the operating system overhead for the systems used in this test is about 1 Kbyte. This means that for data sizes less than 1 Kbyte, the maximum simulation frequency is about 1 kHz and is nearly independent of data size.

For data sizes greater than 1 Kbyte, the Ethernet transfer time becomes the limiting factor. The data shows that the maximum simulation frequency decreases linearly with an increase in data size. The reason that the data points shown do not lie on the theoretical Ethernet limit is that fact that not all of the data sent over the network is simulation data — some of the data is network packet overhead used during Ethernet communication. Given that there is always some fixed packet overhead, the theoretical limit cannot be reached.

Note that adding secondary machines does not affect the transfer rate as long as the machines are on the same network. Using network broadcasting or multicasting, the primary host need only put the data on the Ethernet line once — all those listening to the broadcast or multicast address will receive the update simultaneously.

It is interesting to relate these results to the performance results from the two examples in this chapter. Example one was able to communicate 48 bytes of data at a maximum frequency of 390 Hz. The data of Figure 3-8, however, indicate that the 48 bytes sent by host 1 could be sent at a rate of about 1 kHz. The reason for this discrepancy is that the machines used for example one were slower than those used to generate the data in Figure 3-8. The machines used for example one utilized a 40 MHz MIPS R2000 processor for both host 1 and host 2. The Indigo 2 workstations used to generate the data for Figure 3-8 use a 100 MHz MIPS R4000. Comparing clock speeds we find that the machines used in example one operate at 40% of the clock rate of the machines used to generate Figure 3-8. This *roughly* corresponds

to the difference in operating system overhead between the two machines since they utilize the same operating system and therefore execute the same instructions to perform a network transfer. All other conditions being the same, one should expect the machines of example one to have a maximum frequency (for 48 bytes) of about 40% of the maximum indicated in Figure 3-8 or about 400 Hz which is roughly the result obtained from the example one tests.

This analysis implies that the performance of the host machine is more important than the performance of the network for small data sizes and that a faster machine could raise the maximum simulation frequency (although it would still be limited by the maximum Ethernet rate).

Example two indicated that up to about 16 Kbytes could be transferred at 60 Hz. This is the same result indicated by the data of Figure 3-8. In this case, it is the Ethernet throughput, not the operating system overhead that is the limiting factor. Note that the machines used for example one were also used for example two. The fact these machines are slower than the ones used to generate Figure 3-8 supports the analysis that the operating system overhead is not significant for large data sizes. Despite the fact the the overhead time for the slow machines is nearly three times that of the fast machines, the overhead for both machines is a small fraction of the total network transfer time which is dominated by the Ethernet throughput limit. Since both the fast and slow machines utilized Ethernet for their network communication, they should both have about the same frequency limit for large data size. Both the results of example two and the data for Figure 3-8 bear this out. This analysis implies that using faster machines does not effect the performance of distributed shared memory for large data sizes.

It is also interesting to see the effect of using a faster network (such as a fiber optic network) for distributed shared memory. The theoretical limit of a network that has greater throughput than Ethernet will be a line to the right of the Ethernet limit of Figure 3-8 with similar slope. Since the faster network will generally require the same

operating system overhead as Ethernet, the 1 kHz limit for small data sizes will still exist. However, the maximum data size for which this limit applies will be larger. For Ethernet, the 1 kHz limit applies to data sizes up to 1 Kbyte. If, for example, the theoretical limit goes up by a factor of 10 with the addition of a faster network, data sizes of up to 10 Kbytes will be able to be sent at the 1 kHz rate. Similarly, data sizes greater than 10 Kbytes will see an improvement over Ethernet due to the higher throughput of the fast network.

This means that applications that require large amounts of communicated data can benefit from a faster network. For example two of this chapter, where the update rate is fixed at 60 Hz, this means that more data can be sent using the faster network than is possible using Ethernet. However, for data sizes under 1 Kbyte a faster network has no effect due to the operating system overhead. The fast network has no advantage over Ethernet when applied to example one where the data size is small and the goal is to transfer that data as fast as possible. In this case, the operating system overhead limit is reached before reaching the network throughput limit. This means that both Ethernet and the fast network will have identical performance. For typical applications of the framework where data size requirements are generally small and simulation frequencies are relatively low, using Ethernet for distributed shared memory was adequate.

Chapter 4

Comparison of Other Communication Methods

This chapter compares direct memory access to two other communication techniques in terms of performance, scalability, and extensibility. Performance refers to the amount of communication overhead associated with each technique. Scalability refers to the ease with which a communication mechanism can incorporate new processes into the existing organization. Extensibility refers to the ability of each mechanism to be extended from interprocess to network communication. Associated with extensibility is the transparency of the extension. A communication mechanism is transparently extensible if it can be extended from interprocess to network communication without the need to change the programmatic interface. The communication techniques evaluated here in addition to shared memory and distributed shared memory communication described in Chapter 2, are point to point communication using message passing, and client/server communication [7]. In all cases, the goal is to provide one process access to data managed by another process.

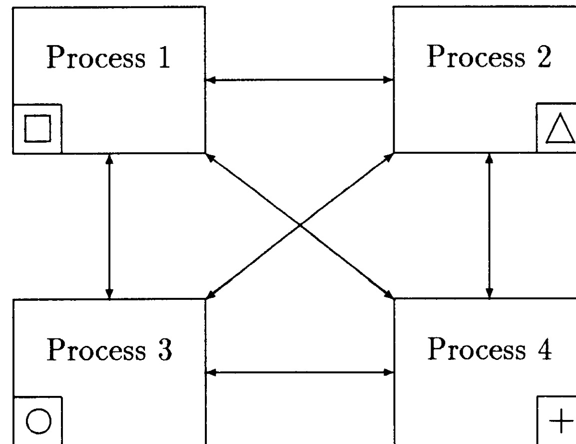


Figure 4-1: Point to point communication.

4.1 Interprocess Communication

Figure 4-1 illustrates interprocess communication with a fully connected point to point organization. Each process has access to some private memory where a piece of the data to be communicated is stored. Each process is connected to every other process in order to gain access to all of the data available for communication. This organization can be realized for interprocess communication in the UNIX operating system by using pipes for the connections [1]. Communication occurs by sending messages between processes. If, for example, process 1 wanted access to the data in process 2, it would send a request for that data directly to process 2. Process 2 would then respond with the requested value directly to process 1. The maximum connectivity of this organization provides an opportunity to parallelize communication. If process 1 requires the data from process 2, and process 3 requires the data from process 4, the two transactions can take place in parallel.

Figure 4-2 illustrates an alternative to the point to point organization — the

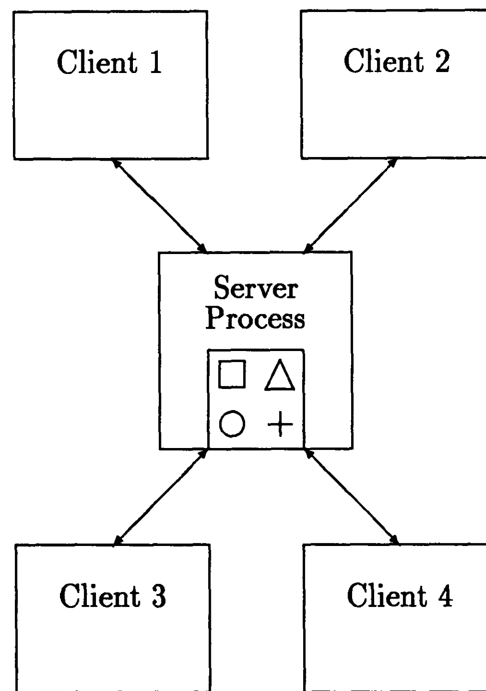


Figure 4-2: Client/Server communication.

client/server model for interprocess communication. For this model, the data to be communicated is kept in the memory of a server process. Client processes have connections to the server similar to the point to point connections previously described. When a client process requires access to the data at the server, a request is sent to the server, and the server responds with the data. Both read and write requests go through the server process. The centralized data storage of the client/server model allows other clients to be added to an existing organization with relative ease.

Figure 4-3 illustrates the organization of the centralized shared memory communication described in Section 2.3. This organization uses the centralized data storage of the client/server model in combination with the maximum connectivity of the point to point organization. As a result, the centralized shared memory communication retains the potential for parallelism of the point to point model (since each process can access the data simultaneously) and the scalability of the client/server model

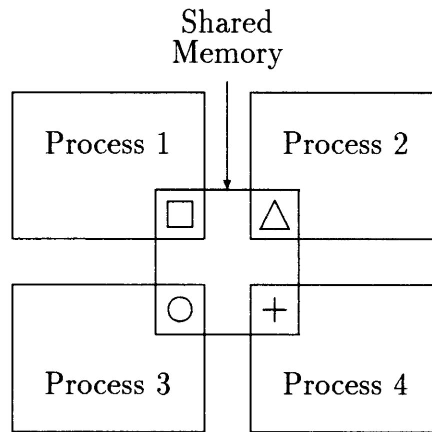


Figure 4-3: Centralized shared memory communication.

(due to the centralized data storage).

Despite borrowing qualities of the other organizations, centralized shared memory communication has advantages over the point to point and client/server models for interprocess communication in terms of both performance and scalability.

Performance: Centralized shared memory communication performs better than either point to point or client/server communication. Both the point to point and client server models involve overhead to transfer data from one process to the other. For interprocess communication using pipes, this overhead is twofold — operating system overhead, and communication overhead.

For a UNIX implementation that uses pipes to connect two processes, the operating system overhead results from requiring extra copies of the communicated data in the operating system. For example, if process 1 wishes to communicate by pipe to process 2, process 1 copies data from its memory to a pipe buffer in operating system memory. The operating system then copies the data from the pipe buffer representing the process 1 end of the pipe, to another buffer representing the process 2 end. From there, process 2 copies the data from the pipe buffer into its memory and the transaction is complete. This requires a total of four copies of the data to perform

the transaction.

Interestingly, Stevens[7] suggests that shared memory be used as an optimization of the pipe implementation. For this optimization, the pipes used to connect each process are each replaced with a shared memory buffer and a semaphore. The shared memory buffer is used as a communication buffer to transfer data from one end of the connection to the other. The semaphore is used to coordinate the communication. With this optimization, process 1 copies the data to be communicated into the shared memory block associated with the process 2 connection and signals process 2 with the semaphore. Process 2 then copies the data from shared memory into its address space and signals process 1 to complete the transaction. This use of shared memory prevents the extra copy of the data required by the operating system in the transfer from server to client.

While the operating system overhead associated with both point to point and client/server interprocess communication can be eliminated, the actual communication overhead cannot. Both of these organizations, require each process to copy data from one end of the connection to the other. In addition, two ends of a connection must communicate a request to the other end in order to initiate the communication. Optimizing the implementation of the connections eliminates neither the data copying nor the request overhead.

Centralized shared memory communication does not suffer from operating system or communication overhead. As with the shared memory optimization for pipes, operating system overhead is not a problem because the operating system is not involved in shared memory reads and writes. In addition, there is no communication overhead because no data needs to be transferred from one process's address space to another since the shared block is already in the address space of every process. The only overhead involved is the concurrency control mechanism. As described in Section 2.4, using read/write locks affords maximum opportunity for parallel access to the data and is especially effective for simultaneous shared memory reads.

Simultaneous reads are a performance problem for both point to point and client/server communication. For the client/server model, simultaneous requests to the server result in some processes waiting while the requests of another process is satisfied. This is a severe penalty in the case where all of the requests are read requests and is especially severe when they are requests for the same data. In this case, identical data is communicated to all requesting processes, but it is done *serially.* The point to point organization has more potential for parallelism, but still suffers from the same kind of problem. In the event that several processes request the same piece of data, the organization reduces to the client/server model where the process responsible for the requested data is forced to service the requests of the other processes one at a time.

Centralized shared memory, provides a greater opportunity for parallel communication than either point to point or client/server communication. In addition, both the operating system and communication overhead problems of point to point and client/server communication do not exist for a centralized shared memory organization. As a result, centralized shared memory communication can outperform either of the other methods.

Scalability: Of the three organizations discussed, centralized shared memory communication requires the least amount of overhead when additional processes are added to the organization. The point to point model does not scale well. Adding a process to an existing organization of N processes requires an additional N connections. For the non-optimized case, this introduces a large operating system overhead penalty. Even for the case optimized with shared memory communication buffers, each additional connection requires a new semaphore and another shared block. In addition, each process must have knowledge of where the data they are interested in resides in order to access that data. If there is no *a priori* location information on the part of an added process, that information must be obtained by sending a request to each of the N existing processes.

The client/server organization has better scalability. The addition of one client

requires the addition of only one connection. For the optimized version, this means another shared block and a semaphore. The centralized shared memory organization, however, requires no additional overhead when another process is added. No additional locks are required, and no additional memory or connections are required. Each process simply incorporates the shared block into its address space and accesses the data. As a result, centralized shared memory communication is more scalable than either point to point of client/server communication.

Centralized shared memory communication combines the parallel operation of point to point communication with the centralized data storage of client/server communication. As a result, centralized shared memory communication has both the performance benefits associated with greater parallelism, and the scalability benefits of a centralized memory organization. This combination provides centralized shared memory communication with both performance *and* scalability advantages over point to point and client/server interprocess communication. The next section discusses these organizations when they are extended to network communication.

4.2 Network Communication

In addition to the issues of performance and scalability associated with interprocess communication the issue of transparent extensibility must be evaluated for network communication. Extensibility refers to the ability of each communication mechanism to be extended from interprocess to network communication. Transparent extensibility adds the constraint that these mechanisms be interchangeable. This provides communicating processes maximum mobility. If the interprocess and network communication mechanisms are the same, communicating processes can reside either on the same host or on different hosts without modification affording the ability to assign computational resources accordingly.

Extensibility: Both the point to point and client/server models can be extended to network communication. In a UNIX environment, the pipe connections used for

interprocess communication can be replaced with sockets for network communication [7]. The programmatic interface for sockets is similar to that of pipes and, in some implementations, the use of a socket connection between processes on the same machine is equivalent to using a pipe [1]. As a result, both point to point and client/server interprocess communication can be extended transparently to network communication.

As indicated in Section 2.5, the centralized shared memory interprocess communication can also be extended to network communication in the form of distributed shared memory. The read/write/network concurrency control mechanism is a direct extension of read/write locking and carries the same programmatic interface by design. Updates for distributed shared memory occur after releasing a write lock in a way that is transparent to the user. As a result, distributed shared memory communication has the same interface as shared memory communication and the extension to network communication is transparent.

All three interprocess mechanisms,therefore, are transparently extensible. However, while both the basic point to point and client/server models can be extended to network communication, their optimized counterparts cannot. The pipe connections used in these methods can only be replaced by shared memory connections if the processes at both ends of the connection reside on the same machine. Since there is no shared memory equivalent over the network, sockets must be used for communication. The fact that the optimizations available for point to point and client/server interprocess communication are not network extensible has serious performance implications.

Performance: Both the point to point and client/server models still suffer from the performance problems discussed in Section 4.1 when extended to network communication. In addition, these problems are magnified by the long latencies introduced by network communication. The client/server model is the worst offender. Consider the case of simultaneous reads discussed in the previous section. Suppose three clients,

each on a separate machine from the server, simultaneously request a read from the server. As before, some of the processes must wait for the others to finish. The waiting time for networked client/server communication, however, is greater than for interprocess communication because it takes much more time for the server process to communicate with a client using a network connection than using a pipe or a shared memory connection. As a result, not only are the client requests serviced serially, but there is a longer service time for each request due to the additional network latency.

This problem can be alleviated somewhat by providing parallel server processes on the server host. This solution, however, only addresses the problem of serial servicing. Unless there are separate physical network lines for each client (an unusual configuration) theses services will be serialized by the fact that communication with each process must be multiplexed on the same physical connection. The analysis of Section 3.3 also suggests that if the client requests consist of small data sizes, there is a limit on service times related to the operating system overhead of a network transfer. As a result, there is a large performance penalty for making several small requests to a server over the network that does not exist for clients residing on the same machine as the server.

Distributed shared memory, described in Chapter 2, minimizes this problem by placing the responsibility for network communication with processes that *write* into shared memory. Once the primary copy receives the write, all other hosts are updated without the need for a remote process to request an update. This is just the reverse of point to point and client/server communication. For both of these mechanisms, communication is initiated by the remote processes wishing to access data that is located elsewhere.

This is taken to the extreme in client/server communication where the client *always* initiates the communication with the server. For this case, clients are never notified when the data at the server has been modified. This is especially a problem for data that changes only infrequently. Since the data resides at the server, a client

cannot generally know if the data it needs has changed. As a result, it must request that data from the server every time it needs it. This forces the client to suffer a network penalty even though the data it is requesting is very likely to be data it already has. This same problem exists for point to point communication. Every time a value is required, a network transaction must take place forcing each process to suffer the network overhead.

Initiating communication on the part of shared memory writers also avoids the serialization suffered by the client/server model when a common physical network connection is used. Distributed shared memory can use this single line to its advantage by allowing an update from the primary site to be broadcast to all secondary hosts simultaneously. This type of parallelization is only possible when communication is initiated by a writing process since the communication originates from a single source. For network communication initiated by reading processes, such as in the point to point and client/server model, broadcasting is of no use because the requests originate from multiple sources even if they are requests for the same data.

Figure 4-4 illustrates the organization of distributed shared memory. Network overhead is reduced because processes on secondary hosts can read the data in distributed shared memory by simply accessing the secondary copy on that host. If the consistency constraint is relaxed as described in Section 2.5.3, there is no network penalty for this type of transaction.

Even if this constraint were not relaxed, however, distributed shared memory would be no less efficient than the client/server model. To enforce consistency, read locks would have to be requested from the primary site resulting in a network transaction to perform a read. This is very similar to making a client read request to the server in the client/server model. If the data requested is small, the service time for the request is dominated by operating system overhead and the performance of the two methods is about the same.

If, however, the requested data is large, distributed shared memory is more effi-

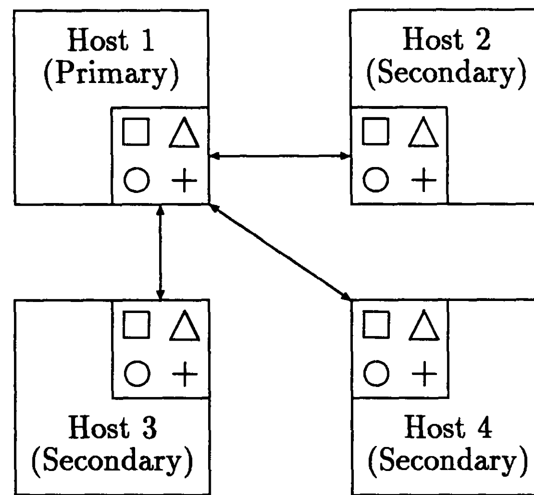


Figure 4-4: Distributed shared memory communication.

cient. Since a transfer of data is initiated at the primary site and occurs only when it is *changed* and not when it is *requested* by the secondary site as in the client/server model, distributed shared memory will perform better assuming that not *all* of the large data block changes all of the time. The network overhead for distributed shared memory from the point of view of the requesting process in this case is simply the overhead of the read lock request. Since the read lock request is a small data size transaction it will only be affected by the fixed operating system overhead associated with small data transactions. The large data size request on the part of a client in the client/server model, however, will have a penalty proportional to the amount of data requested that will *necessarily* be greater than the read lock penalty according to the data of Section 3.3. This penalty will be required even if nothing in the large data block has changed since the last time the client made a request for that data. As a result, distributed shared memory, even with the enforced consistency constraint, will out-perform the client/server model.

It is interesting to note that the point to point and client server organizations are explicit communication models. In both of these organizations, the connections between processes exist for the sole purpose of communication. Distributed shared

memory, however, can be viewed as a *memory* model. The communication in distributed shared memory exists for the purpose of maintaining the memory model. Network communication is possible because the underlying shared memory model is maintained over the network. This is the reason that the network connections exist between hosts and not between processes. The memory model requires that the shared block on each host be viewed as part of the same piece of memory. Memory is a system resource that is made available to processes on the system.

Viewed from this perspective, it is natural that the network connections supporting distributed shared memory exist between systems and not between processes on those system. In fact, if the networked machines shared a common bus and had access to the same physical piece of memory attached to that bus, network communication could be eliminated altogether without affecting the ability of processes on each machine to communicate using direct memory access. In addition, this sort of change would be entirely transparent to the processes using direct memory access in the same way that distributed shared memory is transparent.

Much of the communication performance of distributed shared memory is achieved using this point of view. By looking at distributed shared memory as an extension of each hosts memory hierarchy, access optimizations from the computer architecture literature can be applied. One such optimization is caching. The idea behind a cache is to keep a small amount of frequently used data in an area able to be accessed by a processor much more quickly than main memory. Distributed shared memory takes advantage of the fact that processes tend to read data more frequently than they change it. Similarly, processes tend to want access to the same data repeatedly. This kind of data access pattern has been applied with proven success to CPU memory caching [5]. As a result, performance can be gained by keeping a copy of this information in a location accessible to the host processor much more quickly than it would be able to be accessed over the network. In these terms, the secondary copies used in the distributed shared memory implementation can be viewed as caches of

the primary site data for the other hosts in the organization.

Caching memory for multiple machines suggests application multiprocessor caching strategies. In fact, Gupta, *et. al.* [4] specifically discuss relaxing the memory consistency constraint to improve multiprocessor cache performance. This is very similar to the relaxed constraint used to improve the performance of distributed shared memory in Section 2.5.3. The relationship between distributed shared memory and the processor memory hierarchy is an interesting area for further research.

Given the low network overhead for the data access patterns of typical applications, distributed shared memory has a performance advantage over both point to point and client/server network communication.

Scalability: Recall that the point to point organization did not scale well as more processes were added due to the large increase in the number of connections required to support the organization. The client/server model, however, scaled well with the addition of more clients. This is still the case when these models are applied to network communication. However, when the total number of connections for each model is considered, distributed shared memory scales even better than the client/server model.

For the client/server model, adding a client on a remote host requires a network link to the server machine for each client on the remote host. If, for example, two clients are on one host, and the server is on a different host, both clients require a different connection to the server.

For distributed shared memory, this is not the case. Only one connection exists for each *host* independent of the number of processes at each host. The reason for this is that each host holds an entire copy of the shared area. As a result, processes on that host only need to use shared memory to read that area — no network connection is required. Only the update process on each secondary host needs to have a connection with the primary site.

In addition, the update process on the secondary host can use its single connection

to the primary to fulfill all of the network lock requests for processes on the secondary site. Only one connection is required because the shared area at the secondary site is conceptually the same as the area on the primary site. Once a process at the secondary site obtains the lock it is not possible for any other secondary site process to obtain it since a network lock at the primary copy is viewed as a lock of the entire shared area. Therefore, the lock requests can be queued up at the secondary site and serviced one at a time requiring the use of only one connection between a primary and secondary site.

The only overhead associated with adding another process to the distributed shared memory organization is the extra network connection and shared memory block required when a new *host* is added. Processes added on hosts already in the organization require no additional connection, and can be added without *any* additional overhead in the same manner as centralized shared memory described in the previous section. As a result, additional processes can be added to a distributed shared memory organization with much less overhead than either the point to point or client/server models resulting in better scalability than either of these two methods.

This chapter has shown that compared to the point to point and client/server models, distributed shared memory has both performance and scalability advantages for interprocess and network communication. It is also as transparently extensible from interprocess communication to network communication as either of these models. When applied to simulations developed with the simulation framework, distributed shared memory represented a substantial improvement over either of the other two methods.

Chapter 5

Conclusion

This thesis has described the application of direct memory access techniques to solve the problems of device, interprocess and network communication in the context of a framework for simulation. Data structures memory mapped to address space accessible to all communicating parties was the basis for this communication. Structures mapped to bus addresses provided common memory locations for a host processor to communicate with hardware devices attached to the bus. Sharing memory locations between several processes on the same host extended direct memory access to interprocess communication. Maintaining the shared memory model across networked platforms allowed direct memory communication to be extended to network communication in the form of distributed shared memory.

As with any direct memory access implementation, concurrency control was required to protect the common address space from inconsistency due to concurrent access. Bus protocols control concurrent access to common device memory. Chapter 2 showed that read/write locking can control concurrent shared memory access in a way that enables simultaneous readers to execute in parallel for maximum performance. Read/write locking was extended as a network concurrency control mechanism for distributed shared memory.

Performance was enhanced by relaxing the memory consistency constraint between

networked hosts. As indicated by the examples of Chapter 3, this was a particularly effective optimization when applied to the type of simulations developed with the simulation framework. The data of Chapter 3 demonstrate that this optimization enabled the network performance of distributed shared memory to approach the maximum throughput of the network used to connect the communicating hosts. When applied to simulation, pipelined synchronization, described in Chapter 2, could be applied to insure that this performance could be obtained for without sacrificing global memory consistency.

Chapter 4 demonstrated the advantages of direct memory access communication with respect to two other commonly used techniques. Direct memory access was shown to provide better performance and scalability for interprocess communication than either of the other methods. In addition, direct memory access communication, in the form of distributed shared memory, was able to extend these advantages to network communication.

Viewing direct memory access as a memory model allowed network communication to be driven by the changes written to distributed shared memory rather than by read requests initiated by remote processes. This point of view resulted in less network overhead by reducing the amount of data required to be communicated between remote sites and by enabling the communicated data to be broadcast to those sites simultaneously. For simulations developed using the simulation framework, direct memory access provided a more efficient, more scalable, and more transparent alternative to either of the other two communication methods.

This thesis has demonstrated that direct memory access is an effective form of communication when applied to simulation. Much of the communication used for these applications, however is not unique to simulation. As indicated in Chapter 4, the memory access patterns of simulations are not unlike the access patterns of other applications. As a result, applying distributed shared memory as a means of supporting direct memory access between different hosts warrants further research as a

general mechanism for remote interprocess communication.

Appendix A

Performance Data

Table A.1 contains the raw data used to generate Figure 3-7 for example two of Chapter 3. Host 1 (a Silicon Graphics 440/GTX) attempted to execute the simulation at 60 Hz while communicating data to two display processes each located on a remote host (one a Silicon Graphics Personal Iris 4D/35, the other a Silicon Graphics Iris Indigo). Execution times were measured by subtracting the system time at the end of the last time step from the system time at the beginning of the first time step. Data size was varied for each test and the resulting execution times used to compute the simulation frequency by dividing the number of time steps (6,000) by the recorded execution time to obtain the number of time steps executed per second. Figure 3-7 plots the resulting frequency versus data size.

Table A.2 contains the raw data used to generate Figure 3-8. Execution times were obtained by running a simulation on each of two host machines (both Silicon Graphics Indigo 2 workstations connected via Ethernet) for several thousand time steps while transferring a specified number of bytes every time step from the primary to the secondary host via distributed shared memory. The first row of the table indicates the execution time of the baseline simulation. No network communication was performed during the baseline run. The frequency column of Table A.2 was computed as in Table A.1. Figure 3-8 plots this frequency versus data size on a

log-log plot.

Data Size (bytes)	Execution time (seconds)	Frequency (Hz)
136	100.00	60.00
136	100.00	60.00
136	100.00	60.00
4096	100.00	60.00
4096	100.00	60.00
4096	100.00	60.00
8192	100.00	60.00
8192	100.00	60.00
8192	100.00	60.00
12288	100.01	59.99
12288	100.01	59.99
12288	100.01	59.99
15872	100.02	59.99
15872	100.02	59.99
15872	100.03	59.98
16128	100.04	59.98
16128	100.19	59.88
16128	100.50	59.70
16384	100.04	59.98
16384	100.19	59.41
16384	100.50	59.33
16640	100.66	59.61
16640	101.08	59.36
16640	103.93	57.73
16896	102.36	58.61
16896	102.56	58.50
16896	103.03	58.24
18432	112.56	53.30
18432	113.04	53.08
18432	115.26	52.05
20480	122.59	48.94
20480	124.70	48.12
20480	122.97	48.79

Table A.1: Execution time of 6000 time steps measured for varying data size in distributed shared memory for example two.

Number of time steps	Data Size (bytes)	Execution time (seconds)	Frequency (Hz)
50000	0 (baseline run)	0.48	104,200
50000	16	42.50	1176
50000	16	42.87	1166
50000	16	40.18	1244
50000	16	40.76	1227
50000	16	43.15	1158
50000	32	45.22	1106
50000	32	44.59	1121
50000	64	46.26	1081
50000	64	46.02	1086
50000	64	47.53	1052
50000	128	52.03	961.0
50000	128	50.46	990.9
50000	256	51.27	975.2
50000	256	51.62	968.6
50000	256	52.72	948.4
50000	512	56.98	877.5
50000	512	55.28	904.5
50000	512	55.20	905.8
50000	1024	65.26	766.2
50000	1024	65.18	767.1
50000	1024	65.63	761.8
50000	2048	129.14	387.2
50000	2048	129.11	387.3
50000	4096	247.74	201.8
50000	4096	247.11	202.3
50000	8192	493.06	101.4
25000	16384	498.26	50.17
6250	65536	496.44	12.59
3125	131072	498.75	6.27
1563	262144	497.91	3.14

Table A.2: Execution times measured for varying data size in distributed shared memory.

References

[1] Maurice J. Bach. *The Design of the UNIX Operating System*, chapter 12. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[2] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. In Michael Stonebraker, editor, *Readings in Database Systems*, pages 230–248. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1988.

[3] Janet I. Egan and Thomas J. Teixeira. *Writing a UNIX Device Driver.* John Wiley and Sons, Inc., New York, New York, 1988.

[4] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th Annual Symposium on Computer Architecture*, pages 254–263, 1991.

[5] John L. Henessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, chapter 8. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[6] Gary Sloane and Kevin Walsh. *IRIX Device Driver Programming Guide.* Silicon Graphics, Inc., Mountain View, California, 1992.

[7] W. Richard Stevens. *UNIX Network Programming*, chapter 17. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[8] Jeffrey D. Ullman. *Principles of Database and Knowlegde-Base Systems*, volume 1. Computer Science Press, Rockville, Maryland, 1988.

[9] Stephen A. Ward and Robert H. Halstead Jr. *Computation Structures*, chapter 8. MIT Press, Cambridge, Massachusetts, 1991.

[10] Christopher J. Winters. Simulation framework users guide. Internal document at the Charles Stark Draper Laboratory, Cambridge, Massachusetts, 1992.