

1990

# Dynamic server selection in a multithreaded network computing environment

Joseph F. Stapleton  
*Iowa State University*

Follow this and additional works at: <https://lib.dr.iastate.edu/rtd>

 Part of the [Computer Engineering Commons](#)

## Recommended Citation

Stapleton, Joseph F., "Dynamic server selection in a multithreaded network computing environment" (1990). *Retrospective Theses and Dissertations*. 254.  
<https://lib.dr.iastate.edu/rtd/254>

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact [digirep@iastate.edu](mailto:digirep@iastate.edu).

Dynamic server selection in a multithreaded network computing environment

by

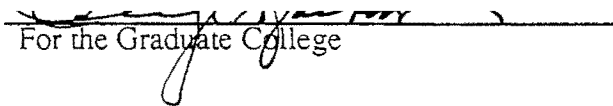
Joseph Francis Stapleton

A Thesis Submitted to the  
Graduate Faculty in Partial Fulfillment of the  
Requirements for the Degree of  
MASTER OF SCIENCE

Department: Electrical Engineering and Computer Engineering  
Major: Computer Engineering

Approved:

Signature redacted for privacy

  
For the Graduate College

Iowa State University  
Ames, Iowa

1990

## TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION	1
Problem Description	1
Proposed Solution	2
Definitions	4
Evaluation Plan	5
Document Structure	5
CHAPTER 2. BACKGROUND	7
Current CNDE Network Computing Environment	7
Comparison to Selected Other Distributed Computing Environments	17
CHAPTER 3. DESIGN CONSIDERATIONS	23
Assumptions	23
High Level Design	25
The Server Selection Algorithm	29
The Chore Queue	31
Fault Handling	33
Cross Language Considerations	34
Variable Argument Lists	35
CHAPTER 4. IMPLEMENTATION	36
Status and Error Logging Utility Functions	36
The Processor Loading Daemon	36
The Server Structure	40
The Client Structure	41
Test Programs	44
Extensibility	47
CHAPTER 5. RESULTS	48
RPC overhead measurements in the CNDE environment	48
Linpack Performance	51
Mandelbrot Performance	54
Problems Encountered	58
CHAPTER 6. CONCLUSIONS	60
Summary	60

Future Work	62
BIBLIOGRAPHY	64
ACKNOWLEDGEMENTS	66

**LIST OF FIGURES**

Figure 1. CNDE Network Topology	9
Figure 2. NCS Protocol Layers	10
Figure 3. Client and Server Program Generation Process	15
Figure 4. General Distributed Processing Configuration	26
Figure 5. Program Structure	28
Figure 6. Server Selection Scenario	32
Figure 7. LLB Entry Record Annotation Field Encoding	38
Figure 8. Chore Distribution Psuedo-Code	43
Figure 9. LINPACK Psuedo-Code	45
Figure 10. Nominal Remote Procedure Call Overhead	49
Figure 11. RPC Overhead Comparison for DDS and IP	50
Figure 12. ZAXPY Single Processor Performance	52
Figure 13. ZAXPY Remote Procedure Call Performance	53
Figure 14. Mandelbrot Execution Times	55

## LIST OF TABLES

Table 1. CNDE Apollo Node Configuration Summary

7

## CHAPTER 1. INTRODUCTION

### Problem Description

Research has been conducted at the Iowa State University Center for Nondestructive Evaluation (CNDE) to create a structure in which existing numerical modeling programs can be converted to execute in a network computing environment. This research task is to include the development of an extensible architecture which accommodates the timely integration of new processing capabilities and requirements. The research was motivated by many needs within the CNDE to reduce the predicted run times associated with the current and future modeling programs.

The project is to demonstrate the feasibility of adapting existing Fortran programs to the Network Computing Architecture (NCA). The primary goal is to create an application layer architecture with a limited set of external interfaces which exploits the opportunities for parallel processing within the existing CNDE computing environment. Parallel processing in the native CNDE environment is complicated by the fact that neither Fortran, i.e., f77 nor the NCA itself has any constructs for expressing program parallelism. Further, the NCA facilities for identifying potential computation servers provide insufficient information to evaluate candidates on the basis of expected throughput. A dedicated low performance computation server may have better throughput than a fully loaded high performance server. Additional logic is needed to estimate performance based on processing power and the current computation load on the prospective server. A secondary goal of this new architecture is to create a higher level of functional abstraction which shields the CNDE software developers from some of the details of the underlying networking issues such as node addressing.

This thesis briefly discusses the analysis process for the existing programs. The objective of the analysis is to identify which program regions to implement in parallel in order to leverage the most performance gain. The major thrust of this thesis is directed toward the

design objectives and implementation of an application layer architecture which supports parallel processing in the CNDE computing environment. Toward this end, system issues such as fault tolerance, software partitioning, and scheduling considerations are addressed in subsequent sections.

### **Proposed Solution**

The proposed solution for creating the application layer architecture is to augment standard commercial packages with local enhancements to provide the necessary degree of robustness, adaptability, and extensibility. The standard commercial packages include the Network Computing System (NCS) and utilities for evaluating current serial program behavior. Robustness is achieved through the implementation of a uniform program fault handling strategy and exploiting host operating system features to control multiple computation servers from one master process. Adaptability is provided so that the system can respond to changing conditions on the network hosts such that high throughput servers are favored over heavily loaded servers and unresponsive servers are automatically eliminated from consideration for future tasks. Extensibility is a gray area which means that relatively few software modifications are required to integrate new functions into this architecture.

The system model for the proposed solution is a master/ multiple slave paradigm to implement a divide and conquer strategy. This strategy implies that the problem can be totally partitioned into sets of independent calculations which may be performed in parallel. Borrowing a term from Jordan, each set of calculations will be called a *chore* [13]. For example, consider a Fortran DO loop in which each iteration of the loop body does not destructively interfere with any other iteration. The overall problem becomes a many to many mapping of the chores to the number of servers. When the number of chores is less than the number of active servers, the system attempts to get the earliest possible completion time at the expense of processor resource utilization. This means that if a server becomes idle before the



entire job has completed but after all of the chores have been assigned to servers, the idle server is redundantly assigned the same chore as an already active server. When any server completes this chore, all other redundant servers working on the same chore have their operation cancelled. When the chore is aborted, partial results are discarded thus wasting the aborted server's processor resources.

An overview of the run-time processing scenario is now described. A computation server program is started on several network nodes. Each instance of the server operates independently and typically runs as a background daemon. A multiprocessor node may have more than one instance of a particular server program. The client may either use all available servers to perform a given function or limit the set of active servers to be the most capable ones. Each active server independently and concurrently computes operations as requested by the client. When all chores have been computed, the client resumes normal serial processing until another opportunity for parallelism arises.

The enhancements developed for this project exist at three levels: UNIX processes, object libraries, and source code. The process is a daemon which runs on all participating nodes to monitor the load on its host and report when the load changes significantly. The object library contains utility functions such as server utilization accounting functions and server comparison functions for use in sorting routines. The NIDL and C language source code will be expanded as new functions are added. Expansions may be required to accommodate new interface definitions and provide proper processing of the new function argument lists.

The process of adapting existing software to run in the network computing environment begins with an analysis of the current application behavior. The objective of the analysis is to identify regions of the program which may be safely implemented in parallel. The software conversion process entails defining the client and server processing requirements, defining the

interface between the client and server(s), and implementing any special handling routines that are required as a result of parallel computation.

### Definitions

Some definitions are required in order to establish the proper context for the remainder of this document. The term *network computing* means a computational system in which the hardware and software components are distributed on a local area network. It is an extension to conventional distributed processing in that this network computing model supports multiple active servers operating in a coordinated fashion. It is also a form of loosely coupled parallel processing since each processing element has its own processor and memory resources and in this case, each has its own copy of the operating system. This contrasts to a special purpose multiprocessor hardware architecture which is composed of tightly coupled high performance processors such as a Connection Machine. A *Network Supercomputer* refers to a collection of high performance workstations which are interconnected via a local area network. It is characterized by a large aggregate computation capacity, large distributed memory, and a very large, perhaps variable communication latency. By this definition, the CNDE computing environment is a network supercomputer.

The term *grain* size arises in the discussion of parallel processing systems. The grain size reflects the minimum size of the program executed on each of the processors which is sufficient to overcome the increased overhead of coordinating the parallel processes. In general, multiprocessors are categorized as fine grain systems meaning that the overhead is relatively low. A network computing systems is categorized as a medium to coarse grain architecture.

A *client* or *master* is a program which controls or consumes computational resources. It is used in tandem with a *server* or *slave* program which provides the resources. Typically in this discussion, the server is strictly a software entity though in some places it may refer to the

host on which the program resides. The meaning is clear from the context. For this project, all clients and servers communicate via the NCS *Remote Procedure Call (RPC)* mechanism.

All software for this project was developed on Apollo Computer Inc. workstations. The operating system for these workstations is a proprietary product called *Domain/OS*. Domain/OS has a built in capability which allows the creation of multiple independent *threads* of execution within one process. Each thread is called a *task*; the original thread is called the distinguished task (DT). Tasks can be created with much less overhead than a process creation. Also, since all of the tasks exist within one virtual address space, inter-task communication is more efficient than inter-process communication. A task is the Domain/OS implementation of a light weight process.

### **Evaluation Plan**

The software developed for this project is a subset of a much larger CNDE development effort. Virtually all of the software developed for this project is written in C. The structure and algorithms presented herein are to be tested for two sample applications. The first is a C language program in which each scanline of a mandelbrot image is computed independently. The second application is a Fortran program which solves a set of linear equations in complex variables. The actual integration of this project software into the larger CNDE project is for future development.

### **Document Structure**

This thesis is organized into chapters. Chapter 2 provides some general background on the current CNDE computing environment and discusses the underlying network computing architecture in detail. It compares this environment to other distributed computing models currently being developed elsewhere. Chapter 3 discusses some of the analysis techniques and design decisions for implementing a distributed program. Architecture implementation details

are discussed in Chapter 4. Chapter 5 reports the results obtained from RPC overhead measurements and timing the sample programs. Chapter 6 contains a summary and discussion. By convention, all UNIX commands, file names, and library function names referenced in the text will appear in the `Courier` font.

## CHAPTER 2. BACKGROUND

There are many varieties of Network Computing Systems in the commercial and research communities. It is an attractive technology for two main reasons: 1) high performance servers can be easily accessed from low performance workstations, and 2) spreading the computation load among under-utilized nodes has great potential for increasing system throughput. The NCA objectives and implementation are described in the next section. Following that, other systems are described and compared to the NCA.

### Current CNDE Network Computing Environment

The current CNDE network computing environment is composed of various models of workstations manufactured by HP/Apollo Computer Corp. At the low end of the computation power spectrum, there are three model DN2500 CISC microprocessor based workstations and at the high end, there is one model DN10040 which contains four proprietary RISC processors. In total, there are 13 HP/Apollo nodes; their configuration is shown in Table 1.

Table 1. CNDE Apollo Node Configuration Summary

Model	Quantity	CPU	RAM (MB)	Monitor	Disk (MB)	MIPS
DN2500	3	M68030	16	M 1280x1024	210	4
DN3500	1	M68030	8	C 1024x800	(2) 380	5
DN4500	1	M68030	16	C 1280x1024	380	8
DN4500	7	M68030	16	C 1280x1024	760	8
DN10040	1	(4) PRISM	64	C 1024x800	(2) 697	22 each
<b>Totals:</b>	13	N/A	246	N/A	8484	155

All of the workstations are inter-connected via an intra-building thin wire Ethernet local area network. The Ethernet cable plant is physically configured as a star topology with a set of active repeaters at the center of the star. Valid packets which appear on one segment of the star are rebroadcast on all other segments. Thus, a logical bus topology is created. One of the spokes from the star leads to a bridge which filters traffic to and from the ISU main campus.

A consequence of this configuration is that the CNDE Ethernet is free of interference as noted by the absence of packet checksum errors. Also, statistics from the bridge indicate that the network on the CNDE side of the bridge has a very low nominal loading. Figure 1 is a diagram of the network topology.

The current version of the workstation operating system is Domain/OS Release 10.2. Both BSD and SysV variants of UNIX are layered on top of Domain/OS. The Domain/OS Concurrent Programming Support (CPS) package facilities for maintaining multiple threads of execution within one process are thread creation, termination, and synchronization.

Each Apollo workstation communicates via both DDS and TCP/IP communication protocols over Ethernet. The DDS is an Apollo proprietary protocol which provides services for all Domain/OS internode communication. The typical TCP/IP services are name service, routing, telnet, ftp, and electronic mail. NCS Applications may employ either or both protocols.

### **The Network Computing Architecture**

The Network Computing Architecture (NCA) is an architecture for distributing software applications across heterogeneous computers and networks [3, 15]. The detailed architecture specifications are found in [2, 24]. The HP/Apollo implementation of NCA is called the Network Computing System (NCS). NCS defines a request - response protocol and the packet formats to create a layer of reliable communication on top of a network layer which provides unreliable datagram services. A connectionless network protocol was selected to minimize RPC overhead and make NCS applications viable on hosts which do not support connection oriented protocols. The Berkeley socket abstraction is used to access the network layer. Figure 2 shows the NCS protocol layers with a cross reference to the ISO/OSI communication model layers. The principal NCS components are the RPC, the Network Interface Definition Language (NIDL) compiler, the location brokers, and the task broker.

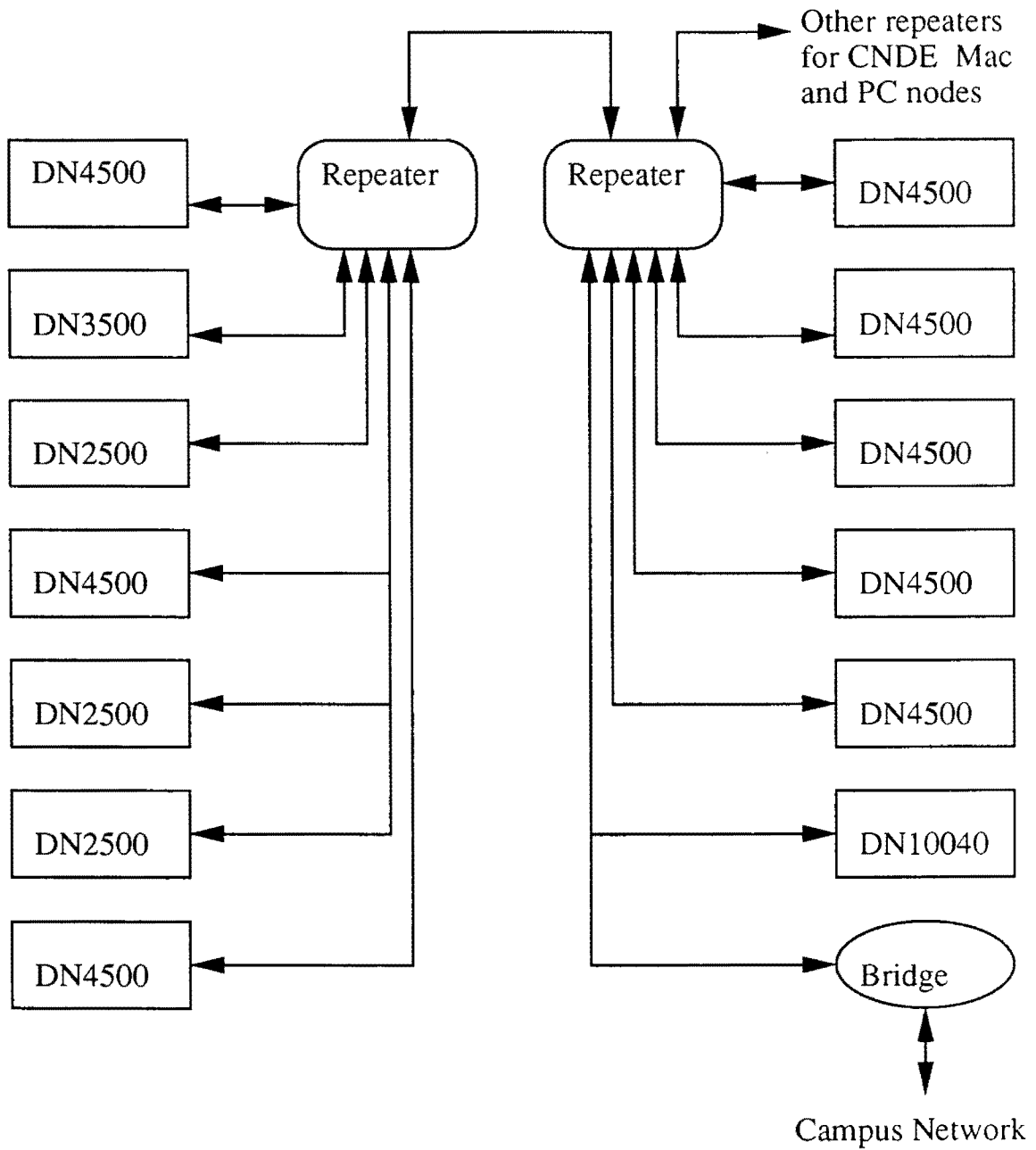
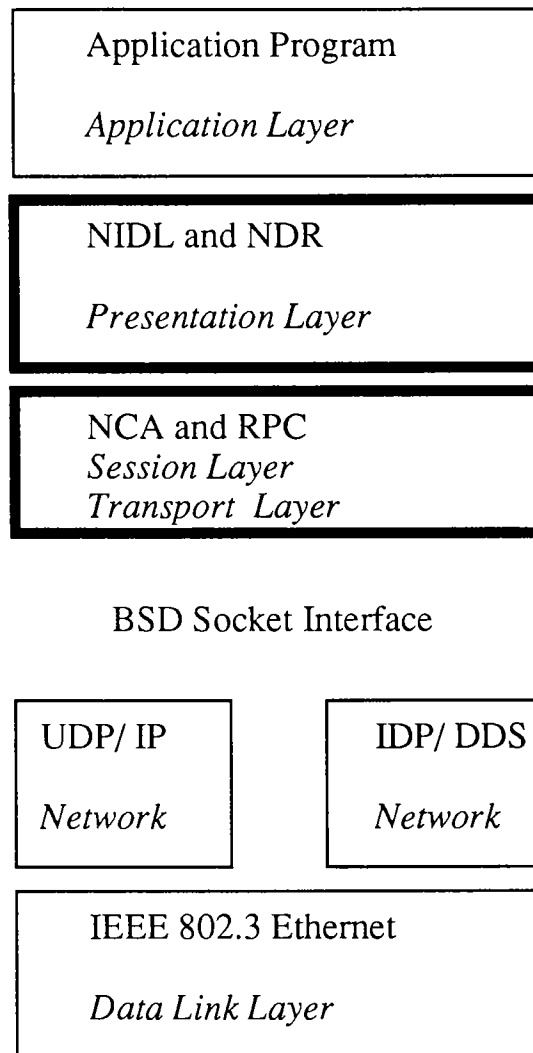


Figure 1. CNDE Network Topology



**Legend:** *ISO/OSI terminology*

UDP: datagram protocol for the Internet Protocol

IDP: datagram protocol for the Apollo DDS protocol

NCS components enclosed in heavy boxes

Figure 2. NCS Protocol Layers



NCS is used extensively in the Domain/OS daemons which manage user authorization, printing services, and file system backup. NCS is also the foundation for the software architecture developed and described herein.

The NCS Computation Model The NCS computation model is object based. Resources are characterized as objects. For example, a set of functions which manipulate matrices could be an object. Objects in turn are categorized by type and manipulated through a set of operations. An interface is composed of a set of related operations. Servers are constructed to as collections of objects. A server is said to *export* all interfaces associated with its object types. A client places a RPC to a server which is known to export the desired interface.

The RPC is the basic element of the NCS computation model. The RPC mechanism allows a client program on one host to contact a server program resident on another host through an interface which appears to be a local procedure call. In reality, the client function call transfers program control to the client stub routine. The stub routine accesses the RPC run-time library to assemble the function arguments into network packets and transmit them to the remote host. There, the server stub routine accepts the data, unloads the network packets, and invokes the intended server function. Results are returned through an inverse process. On the Apollo, this model can be extended through the use of CPS functions. CPS can be applied to NCS such that a client may define multiple tasks; each task may concurrently initiate a RPC. Concurrent RPC is the mechanism which supports parallel processing in this environment. The CPS functions were used extensively in the developed software.

Though not exercised for this project, the NCS supports communication with heterogeneous vendor platforms via the Network Data Representation (NDR). If any data

representations are not consistent on a pair of communicating hosts, the receiver<sup>1</sup> has enough information to translate received packets into the correct local format. Packets are always transmitted using "native" data formats. The receiver performs all data translations but only if they are necessary. This contrasts with the SUN implementation of RPC data representation. In that case, the sender and receiver always incur processing overhead as the data elements are converted to and from a neutral transmission representation even if the sender and receiver have the same native data representation [21]. NCS does not support explicit data typing on transmitted packets; only the actual value of the data are transmitted. All NCS data typing occurs when the interface is defined.

RPC Details The basic single-threaded RPC mechanism is mature technology. Recent commercial implementations are based on the RPC framework presented by Birrell and Nelson [6]. A goal of any RPC system is to make distributed computing easy for the implementor. The ease of use stems from the appearance of making a normal local procedure call when in fact the remote server is actually performing the intended operation.

A remote procedure call is intended to have the same behavior as a local procedure call. This dictates that the RPCs are blocking; program control is not returned until the server completes the request and returns the result. The RPC will reflect server run-time errors, for example a floating point exception, back to the client. In addition, the run-time library monitors the progress of a call so that it can detect and report host or network failures. The NCS mechanism for this function is a periodic ping and acknowledge packet exchange between the client and server. The ping frequency is adjusted with a binary exponential backoff scheme to a maximum interval of 1024 seconds [2]. The client and server are modeled as finite state

---

<sup>1</sup>Note the distinction between client/server pairs and transmitter/receiver pairs. The client is the transmitter and the server is the receiver during the function activation phase. The client and server roles are reversed during the function return phase.

machines in the run-time library. The state machines handle possible network computing anomalies such as packet retransmissions and timeouts.

Arguments to RPC functions must eventually be passed by value since a remote host cannot translate passed addresses in the proper context. The RPC interface definition may include parameters which appear to be passed by reference. In this case, the stub routines will perform automatic de-referencing. A NCS RPC occurs in a presumed trusted environment. Nothing is encrypted and there are no passwords nor any other security checks associated with the call itself. Servers and clients are assumed to be started by authorized users. In comparison, the SUN RPC protocol includes selectable security such that the designer can choose from no authentication, UNIX authentication, or DES authentication [21].

The NIDL Compiler The NCS NIDL compiler takes an interface definition as its input and generates the C language stub routines for the client and server. The NIDL defines the syntax of an interface definition. An interface definition is composed of the interface name, a universal unique identifier (UUID) for the interface, and a list of functions which are exported through the interface. A UUID is a 16 byte binary string which encodes the hardware node id on which the UUID was created and a timestamp. Each listed interface function itemizes the data type and direction of each of its parameters. By convention, direction may be in to the server, out from the server or both. Simple data types e.g., integers and characters as well as aggregate data types, e.g., structures and arrays are supported.

The stubs emitted from the NIDL compiler contain the software which redirects the local invocations to network transmissions. The stub routines also contain the software functions which marshall and unmarshall the RPC parameters. Marshalling is the process of packing the RPC parameters into network packets. The client stub is compiled and linked with the client application software. The server stub is compiled and linked to the server application. The NIDL compiler also generates a client switch stub file which allows for proper function name

translation in the case of a replicated application where the client may access some of the interfaces which it exports. Figure 3 depicts a typical compile and link process for clients and servers which are produced from Fortran, C, and NIDL source files.

NCS Location Brokers The Location Broker daemons act as repositories for server registration information and as forwarding agents for client connection requests. The daemons have been implemented in two varieties: the Local Location Broker (LLB) and the Global Location Broker (GLB). A LLB services all server registration requests and client lookup requests for the local node. The GLB is used in conjunction with the LLB to resolve addresses which are not local to the requesting host. Typically, each node runs the LLB and just a few nodes run the GLB daemon. The GLB database is replicated among all network GLB daemons in a highly available, weakly consistent fashion. The `/etc/ncs/lb_admin` utility provided with the NCS release allows a user to display and/ or manually modify the contents of either the LLBD or the GLBD data base.

A location broker entry is composed of the interface UUID, an object instance UUID, an object type UUID, a global/ local flag, a free form text annotation field, a socket address length and a complete socket address which includes an address format identifier. The socket address format also implies the data communication protocol. An IP address format means IP protocol and a DDS format means DDS protocol. A server may have many location broker entries, one for each interface that it exports.

If a server address is initially completely unknown to the client, as is usually the case in NCS, the client formulates a lookup request which encapsulates the desired server specification. The location brokers search their database for entries which match the request

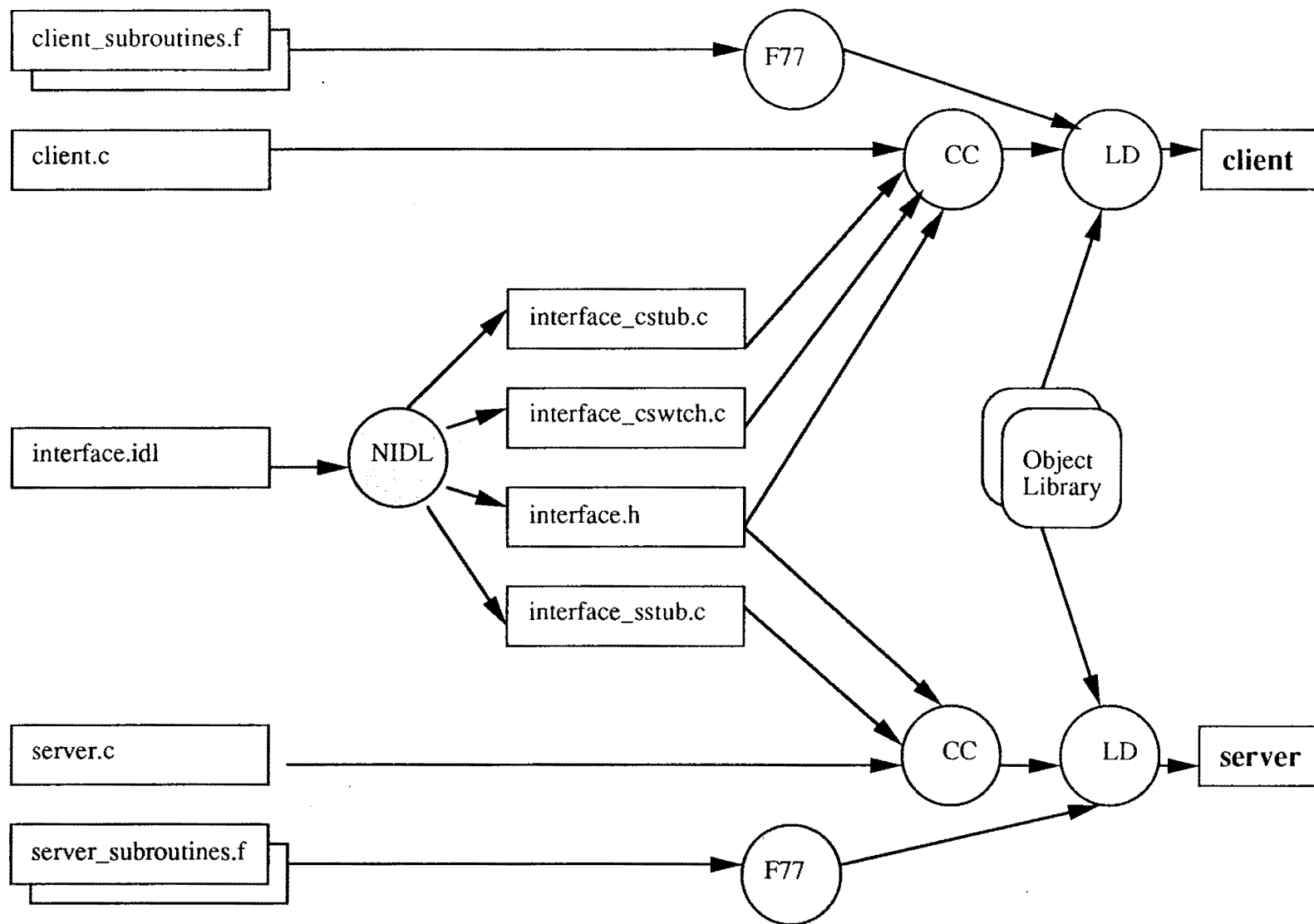


Figure 3. Client and Server Program Generation Process

specification. All entries which match the specification are returned to the requestor. The entries returned to the requestor are in no particular order. Single threaded clients would typically typically resort to using the first entry in the list. This is not a desirable characteristic if the objective is to minimize execution time and the some other entry in the list actually contain the location of a higher performance server. This deficiency is addressed in the project and discussed in Chapters three and four.

If the client has partial information on the location of the intended server, i.e., it has only the host address but not the port, the RPC may be placed to the LLBD well known port on the remote host. In this case, the LLBD acts as a forwarding agent and control is transferred to the intended server. When the RPC returns, the client will have the fully specified address for use in subsequent calls to that same server.

Task Broker The Task Broker is an additional HP/Apollo layered product which is intended to provide some measure of network host load balancing. It is oriented toward batch programs which require no user input and do strictly file input /output [12]. A system manager must ensure that the programs are available on each prospective server and configure the Task Broker with information such as program processing requirements, the expected network activity, filenames, etc. When a user submits a program start request to the Task Broker, it queries potential servers for bids and selects the highest bidder. If no bids are received, the request is queued locally until a server becomes available. After a server successfully completes, all output files are copied back to the submitting host. This technology was deemed unacceptable in the CNDE environment because its batch orientation makes it quite inflexible and it requires the system manager to have a-priori knowledge of the processing requirements. Task broker has no provision for balancing computation loads which are data dependent.

## Comparison to Selected Other Distributed Computing Environments

### Project Athena

Project Athena is a seven year old cooperative effort between MIT, Digital Equipment Corp., and IBM to develop a large scale heterogeneous computer system composed of networked workstations [8]. The individual workstations are merely the distributed hardware components of a larger system in which all resource allocation, security considerations, and access to services is handled not at the node but at the system and or network level. The Athena environment is essentially a layer of distributed services built on top of Berkeley Unix and the Network File System (NFS) from Sun Microsystems. Athena has dedicated server programs to handle user authorization and authentication, name service, system management service, file service, window management, etc. Some of the Athena service daemons are replicated to ensure high availability.

Each user has access to the power of the workstation at which they are seated. The Athena system model does not include the concept of dedicated computation servers though the architecture accommodates heterogeneous workstations with disparate computation power. There is no direct support within Athena for parallel processing nor any automatic service to migrate compute intensive jobs to the most suitable computation engine. A user must be aware of the components of the network and submit jobs on capable computation servers which have been configured to accept requests from this user.

### Enterprise

The Enterprise system for distributed task scheduling was developed at the Xerox Palo Alto Research Center [17]. It is intended to run on top of a remote process communication mechanism like RPC. The system scheduling is based on the concept of an agoric computing environment where servers "bid" for available work much like an auction in an open market.

Each server independently computes its bid. Bidding is based on server processor capacity, speed, network load, job characteristics, current location of data and related tasks. Enterprise implements a Distributed Scheduling Protocol (DSP) which specifies the sequence and contents of bidding messages. The typical message sequence is *Announcement, Bid, Award*. A client broadcasts the Announcement which includes a description of the job and its priority. Idle servers respond immediately with a bid; busy servers enqueue the request and submit a bid when they become idle. An idle server selects queued requests based on the job priority in a FIFO fashion. A bid is essentially the server's estimate of the job completion time.

Multiple server bids may arrive at the client. The client always awards the job to the first server from which it receives a bid. If another server submits a later bid which is significantly better than the bid received from the first server, the job will be resubmitted to this better server and aborted on the original server.

Clients periodically request status from their active servers. If no response is obtained, the job is restarted elsewhere. Also, if the server does not get pinged periodically, it will autonomously abort the job with the assumption that the client has crashed. Enterprise has an estimation error tolerance parameter which is used to encourage reasonable estimated job completion times. A job which exceeds the estimated time by more than the error tolerance is aborted and restarted elsewhere.

Enterprise has been implemented at Xerox in LISP. Testing has shown that the estimates of processing time need not be very accurate. Estimation errors of up to  $\pm 100\%$  resulted in little performance degradation. As expected, dramatic performance improvements were noted in a test case in which the network was lightly loaded and the processors were moderately to heavily loaded. Even so, there was a steeply diminishing return on the benefit of increasing the size of the server pool beyond a fairly low limit of eight to ten servers. Communication delays



were found to be intolerable for lightly loaded processors as the delay became a significant percentage of the actual processing time.

The server selection scheme for Enterprise is considerably different than that of NCS. There is a notion of expected server performance in Enterprise. The broadcast mechanism and the number of steps required to select the server detract from the elegance.

### **Emerald System**

The Emerald System is a current research project at the University of Washington [14]. It is composed of a language and a run-time environment. Emerald supports distributed programs via objects which can transparently move between network nodes. Objects may either be static data structures or live processes. One of its goals is to achieve performance which is comparable to traditional RPC performance without adversely impacting the local operation performance. Emerald offers essentially the same set of advantages as does NCS: load balancing, simplified data movement, potential for enhanced run-time performance. Further, Emerald can reduce the interprocessor communication load by moving the communicating processes to the same node. In a traditional RPC environment, the caller is blocked while the server is executing the call. Within Emerald, the entire process moves to the remote node and continues execution.

An Emerald object associates unique name with a data representation specification, a set of operations which may be performed on the object, and an optional process. The Emerald compiler is context sensitive such that it will produce different object implementations depending how an object is used; not all objects are assumed to be global nor mobile. The roving objects are located using a forwarding address. Each node has an access table which maps objects to residency. An access table entry is created for each object that has a remote reference. If an object moves, the source and destination nodes update their access table

forwarding address field for that object. No other nodes take any action. When other nodes require access to a remote object, they traverse the tree of access table forwarding addresses.

The constraints for implementing the Emerald style mobility are more severe than for NCS. Emerald requires homogeneous nodes, i.e., trusted nodes with the same instruction set. In its current implementation, all Emerald processes resident on a single node are mapped into the same virtual address space. These restrictions make the Emerald approach unsuitable for the CNDE environment.

### **PAX-1**

The PAX-1 product from VXM Technologies, Inc strives to create a network supercomputer in the VAX-VMS environment [16, 23]. PAX is an implementation of Linda-C. Linda-C is an extension to ANSI C which provides constructs for writing explicitly parallel programs. Carriero and Gelernter provide an excellent summary of the Linda model in their 1989 paper [7]. In comparison to NCS, PAX is also suitable for medium to coarse grain parallel processing applications though it is not based on a RPC mechanism nor on a ubiquitous upper layer transport protocol such as TCP. It removes much of the overhead of interprocessor communication by directly utilizing the Ethernet data link layer.

PAX-1 processes interact via *tuples* stored in a distributed shared memory called *tuple space*. Tuples are ordered sets of parameters which may be either active data structures, i.e., processes or they may be passive data structures. Only passive tuples are currently supported in PAX. Tuples may contain wildcards which specify data types but not actual values. Tuple space is accessible from all nodes in a PAX-1 network. Tuple operations are: out, to insert an entry into the tuple space; in, to retrieve a tuple from tuple space; rd, to read a tuple but the tuple is not removed from tuple space. Out operations are non-blocking. If no matching tuple exists when an in is attempted, the requestor will block until a match appears. If more than one match exists, the one returned is chosen non-deterministically. If more than one server is

blocked on an attempt to in the same tuple, their eventual service ordering is non-deterministic. This implies that the algorithm used to award the tuples to the servers is not “fair”; some server may never be activated.

Servers will typically in a tuple which matches their template. If a match is found, it is removed from the tuple space, operated upon, and a new tuple is inserted. The client posts several entries into the tuple space requesting some service and waits for the results to be asynchronously posted back into the tuple space by a set of servers. The client has no knowledge of the server locations or even the number of servers. Additional servers may be added to the network to dynamically increase the system throughput. In a future version of PAX, it will handle heterogeneous vendor platforms, incorporate support for UNIX, and offer transparent data translation between systems. One shortcoming in the implementation is an inability to detect a server crash after it has removed a tuple but before it has posted the resulting tuple. A developer must explicitly handle this case with a timer and a signal handler function to prevent the client from blocking on an in which will never occur because the server has crashed.

## **ISIS**

The ISIS program is a current research and commercial project at the Cornell University [5]. It is a library and development environment for distributed applications. A client may reference servers via an opaque data structure known as a process group name. The client need have no knowledge of the number of group members nor their location. The group membership can grow and shrink dynamically and a process may belong to arbitrarily many groups. ISIS requires lightweight tasking to implement some functions. In this context, an ISIS task looks like any other C function but with the distinguishing feature that the ISIS task can be invoked in response to events such as the receipt of a message. Multiple ISIS tasks may be executing concurrently. The task modules must be coded in a fashion to protect the global

variables from unintentional corruption by another executing task. ISIS guarantees that all process group members receive message events in the same order. Since the events appear in the same order, the group members are said to be *virtually synchronous*. This greatly simplifies the design of distributed applications.

ISIS is well suited for controlling parallel processing applications which employ a divide and conquer strategy since one message can be simultaneously received by all servers which are members of a particular process group. Like NCS, ISIS also lacks a deterministic server selection mechanism. Future versions of ISIS will address noted problems with sluggishness and scaling difficulty.

## CHAPTER 3. DESIGN CONSIDERATIONS

The preceding chapter illustrated some of the differences in the capabilities of some network computing architectures. Some of the differences are viewed as deficiencies while others are desirable characteristics. In this chapter, I provide the framework for an architecture which is layered on NCS yet has some of the benefits of the other systems as well. The first section describes the assumption under which this design is intended to operate. Subsequent sections discuss the major components.

### Assumptions

A divide and conquer strategy is not particularly applicable to some of the existing CNDE model software. Thus, a network computing solution is not appropriate for all of the CNDE computing needs. Such programs are not under consideration for this project. Often, sequential programs have sections which can be adapted to perform multiple independent calculations. These sections of code must be analyzed so that data dependencies are identified and removed so that the code within a section can be executed in parallel. Presently, this is largely a manual process. As a starting point for this process, commercial utilities are available which identify which routines should be modified to leverage the improvement offered by parallel processing. The UNIX utility for this purpose is `prof`. It displays the percentage of time that an application spends in its subroutines. This is exactly the type of information required but it doesn't work well on the Apollo. Instead, the Apollo utilities for program analysis are a layered product called the Domain Performance Analysis Kit (DPAK). DPAK is composed of the Histogram Program Counter (HPC) tool and the Domain Performance Analysis Tool (DPAT). HPC periodically samples the program counter while a program is executing. After the program completes, a histogram is displayed which indicates the frequency of the PC being in specified address regions. The DPAT operates by periodically sampling the program call/return stack while a program is running. DPAT also records

parameters such as I/O activity and page faults. When the analysis is complete, the relative amount of time spent in each of the program functions is displayed. One can infer likely processing bottlenecks from this information. DPAK tools do not provide any advice on how amenable the functions are to parallel processing nor how to convert them.

Much has been written about tools to analyze and convert scientific Fortran programs to multiprocessor machines. Ottenstein gives a good survey of the techniques for detecting parallelism and he includes an extensive list of references in his 1985 paper [20]. The Kuck Analyzer Package (KAP) developed by Kuck and Associates, Inc. is a Fortran preprocessor which does a thorough dependency analysis to identify program regions which may be safely executed in parallel. Some versions of the KAP allow the user to specify a minimum amount of parallel activity which must be present in order to invoke the parallel code [22]. This is intended to be used to force sequential execution of loops which *could* be done in parallel but should not because the overhead cost exceeds the gain of the parallel computation.

A related design issue that must be addressed in a network computing environment is the proper selection of the client and server functional partitioning. The partitioning must reflect the architecture grain size. The overhead cost of doing a RPC is orders of magnitude greater than the cost of doing a local procedure call. This implies that to be effective, the client should make relatively few RPCs with relatively lengthy computations on the server such that the communication overhead becomes a small percentage of the overall execution time. In the Apollo environment, a local procedure call can take on the order of tenths of microseconds while a trivial RPC can take on the order of milliseconds. Both were measured by repeatedly performing null function calls in a tight loop.

The client /server communication delay is variable since the underlying technology is Ethernet. Ethernet is known to perform best on lightly loaded networks. Some recent performance measurements by the Digital Western Research Laboratory have shown that

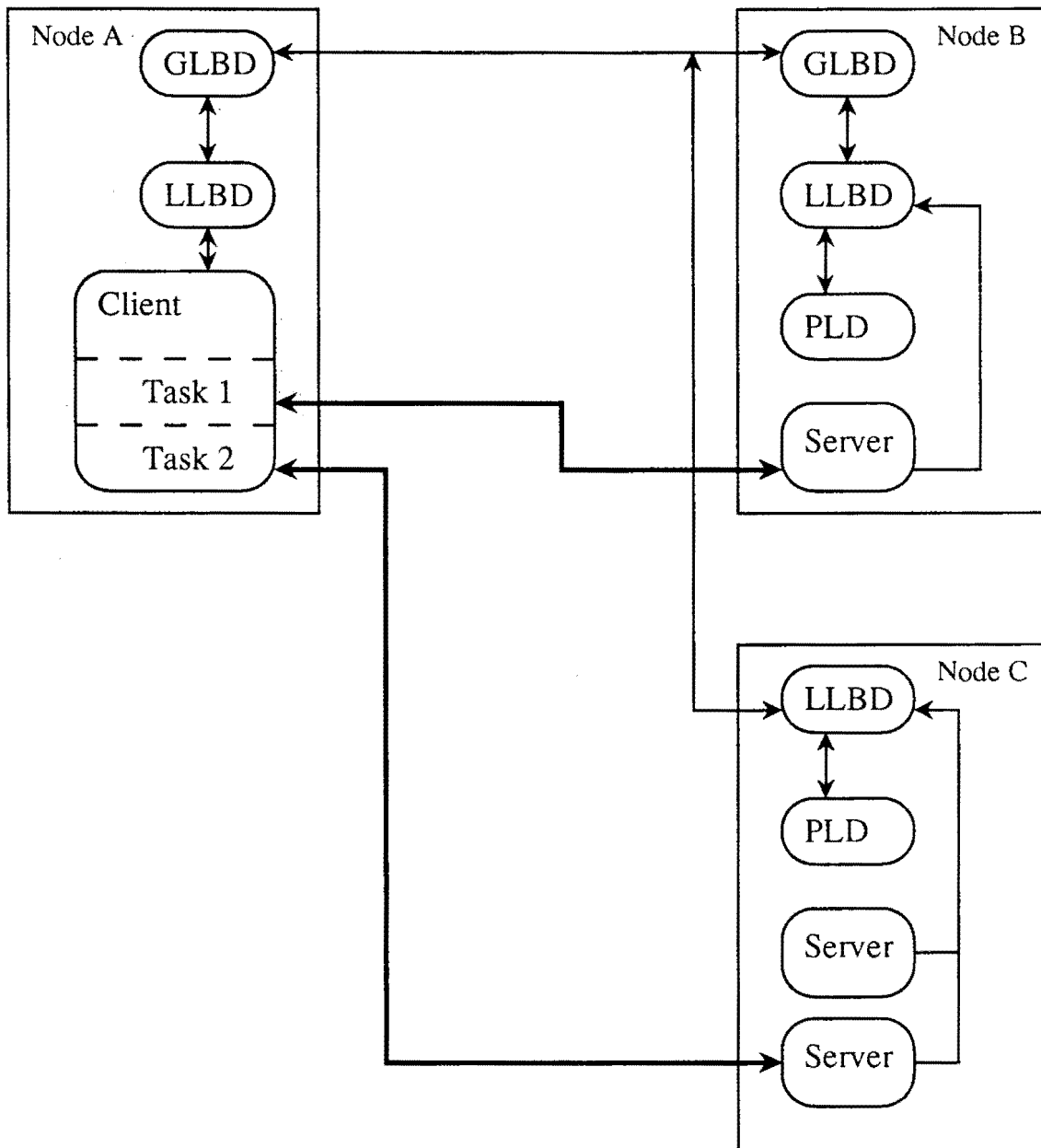
Ethernet performs quite well on more heavily loaded networks as long as the packet size varies [23]. Periodic samples of statistics collected from the CNDE Ethernet bridge have shown that under current conditions, the CNDE network is very lightly loaded. For problems of the appropriate grain size for this architecture, client/ server data communication latency relative to the server execution time is expected to be small.

Finally, I assume that there is no requirement to augment the security of NCS transactions. Packet checksums are computed by the RPC run-time library to ensure the integrity of a received packet. Servers do not maintain lists of acceptable clients; all valid service requests are accepted. A server is designed to handle at most one active client. There is no requirement for logic to prevent interference from multiple concurrent server tasks within a server process.

### **High Level Design**

Figure 4 depicts the run-time configuration of a generic application which consists of a multi-threaded client and three servers distributed over three nodes. Each server program is started at boot time. The client program is started on demand much like the entire single threaded program is currently started. Both server nodes are running the Processor Loading Daemon (PLD) and the LLBD. A GLBD is present on the client node and one of the server nodes. The client program created two tasks. Each task communicates with a server via a RPC.

During initialization, the server program queries the network nameserver to determine hardware information for the host on which it resides. The name server *HINFO* record for each host has been encoded with a processor performance metric. Currently, the metric in use is the host MIPS rating as supplied by the vendor specification data sheets. The server formulates a location broker registration entry which encapsulates the performance metric. After registering with the LLBD, the server enters a quiescent state awaiting RPC requests.



GLBD: NCS Global Location Broker  
 LLBD: NCS Local Location Broker  
 PLD: CNDE Processor Loading Daemon  
 Client: CNDE Model User Interface (typical)  
 Server: CNDE Model Computation (typical)

Dark Line: Remote Procedure Call path  
 Light Line: Daemon data flow path

Figure 4. General Distributed Processing Configuration



The PLD regularly measures the processing load on its host node. To reduce processing overhead, not every load calculation results in a LLBD update. The algorithm to update the LLBD is adapted from the algorithm used to propagate routing table updates in the ARPANET [18]. The LLBD is updated only if the load is "significantly different" from the loading the last LLBD update. The phrase "significantly different" means that the absolute value of the change in the PLD load is greater than some threshold. The threshold is a generally decreasing function of time which gets reset to its maximum value whenever a LLBD update occurs. After every PLD measurement interval, the threshold value is decreased. Eventually, the threshold could reach zero in which case an update will occur and the threshold will be reset regardless of the load change. This algorithm was selected because large changes in the load are reported quickly and persistent smaller changes are reported eventually. If the load is significantly different from the previously reported load, the PLD will retrieve all LLBD entries which match its template type. If any entries are found, they are updated with the new load information and reinserted in the LLBD database.

A user may manually update the LLBD entry annotation field to mark a server "off-line" via the `/etc/ncs/lb_admin` utility. If a server is marked off-line, the server will continue to process chores from the current client but future clients will not use this server. The use of this capability is limited to special situations such as a case where the machine is scheduled to go down for maintenance and an orderly server shutdown is desired.

When a client program requests server location information from the LLBD, it automatically receives the processor performance metric and the processor loading information. No separate query is needed to determine the load. Also, note that the client does not use a broadcast mechanism to initiate the server selection process.

The organization of existing CNDE modeling programs differs significantly from the organization which is required for this environment. The top section of Figure 5 shows a

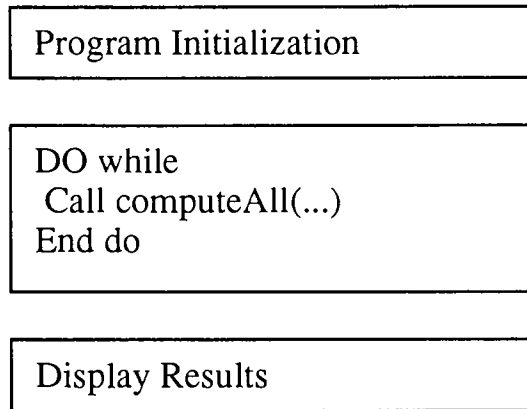
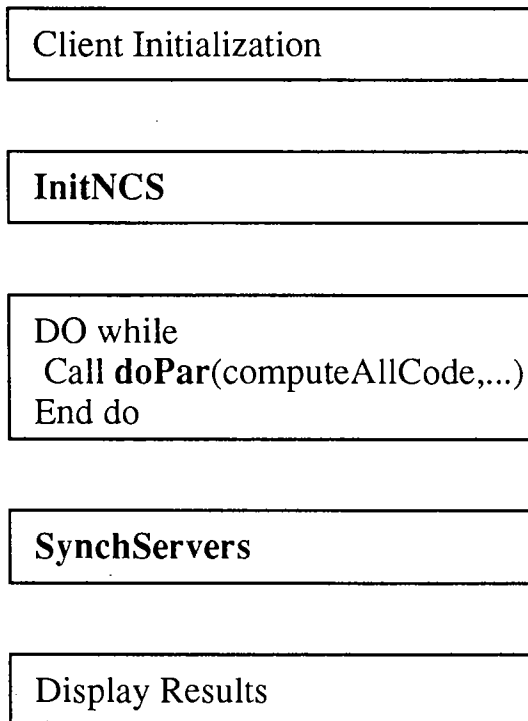
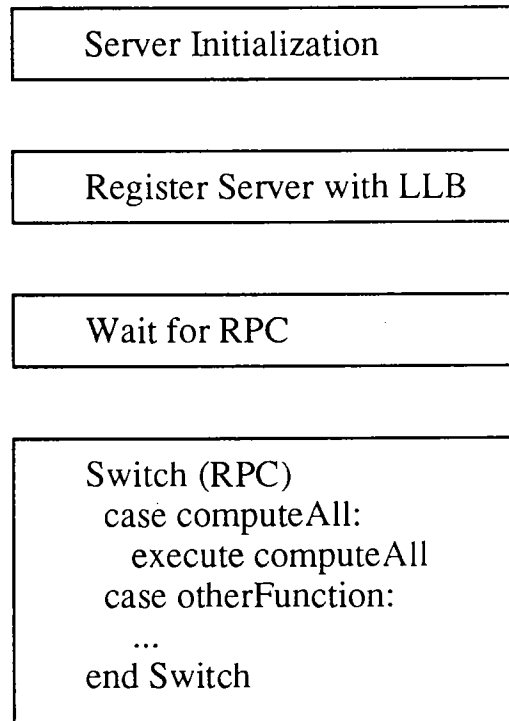
*Old Program Structure**New Client Structure**New Server Structure*

Figure 5. Program Structure

simple program which has an organization similar to current CNDE programs. It contains initialization functions, a main processing loop and a set of functions to display the results of the computation. In contrast, the lower section of the figure shows the high level organization of the server and client programs. The server program does its own initialization and waits for a RPC to arrive. When it does arrive, the RPC run-time library invokes the intended function and then returns the results to the client. The client program performs initialization and display functions. The computation load is shifted to the server with the three functions shown in boldface in the figure.

The three callable functions are intended to easily identify and isolate the parallel regions. They also serve to mask the details of multi-threaded execution from the client application code. A call to the function initNCS identifies the desired functional interface and the requested number of servers. initNCS performs the location broker lookup, initial server selection, and starts a client task for each available server. A call to the doPar function identifies a set of parameters for a chore to be executed. The function doPar queues the chore and returns immediately without waiting for the chore to be processed by some server. A call to the syncServers function causes the client Distinguished Task to block while waiting for all client sub-tasks to complete. The sequence initNCS, doPar, and syncServers may be executed more than once within one program.

### **The Server Selection Algorithm**

The server selection algorithm has two modes of operation: one during parallel processing initialization, i.e., during initNCS, and the other while a client is distributing chores to a set of active servers. In the first case, a server is thought to be superior if its performance metric divided by the current load is greater than the corresponding ratio for another server. A LLBD query returns a set of server records. The server state, performance metric, and the current load information are extracted from entries encoded in the LLBD record annotation field

for each server. If the server state is marked off-line, that server is eliminated from further consideration. The ratio of performance metric to current load is computed for each potential server and the results are inverse sorted such that the highest ratio servers are listed first followed in decreasing order by lower performance workstations. This mode is used as an estimate of which servers will perform better. If the client call to initNCS limits the requested number of active servers, the LLBD list is trimmed on this basis.

The second mode of server selection is actually a technique known as self-scheduling. In this case, the total client job to be processed in parallel is divided into a number of chores. The number of chores is initially large relative to the number of processing elements. Each server obtains a chore and when finished with that computation, it obtains and performs the next unassigned chore. Self-scheduling was selected because it automatically adapts to the run-time server response conditions and does not rely on potentially stale or irrelevant load information. The load information could be stale if a significant amount of time had elapsed since the initNCS function was invoked. In other words, current performance data is preferred over a guess based on the location broker information. The self-scheduling technique is more flexible than a pre-scheduled technique because the programmer does not need to manually try to balance the computation load. The pre-scheduling or a-priori load balancing technique was not used because it is not a viable method when the execution time for each chore can vary sharply because data dependencies may cause different conditional branch paths to be executed. Pre-scheduling is also difficult to apply when the processing elements do not have uniform computation speed, as in this network.

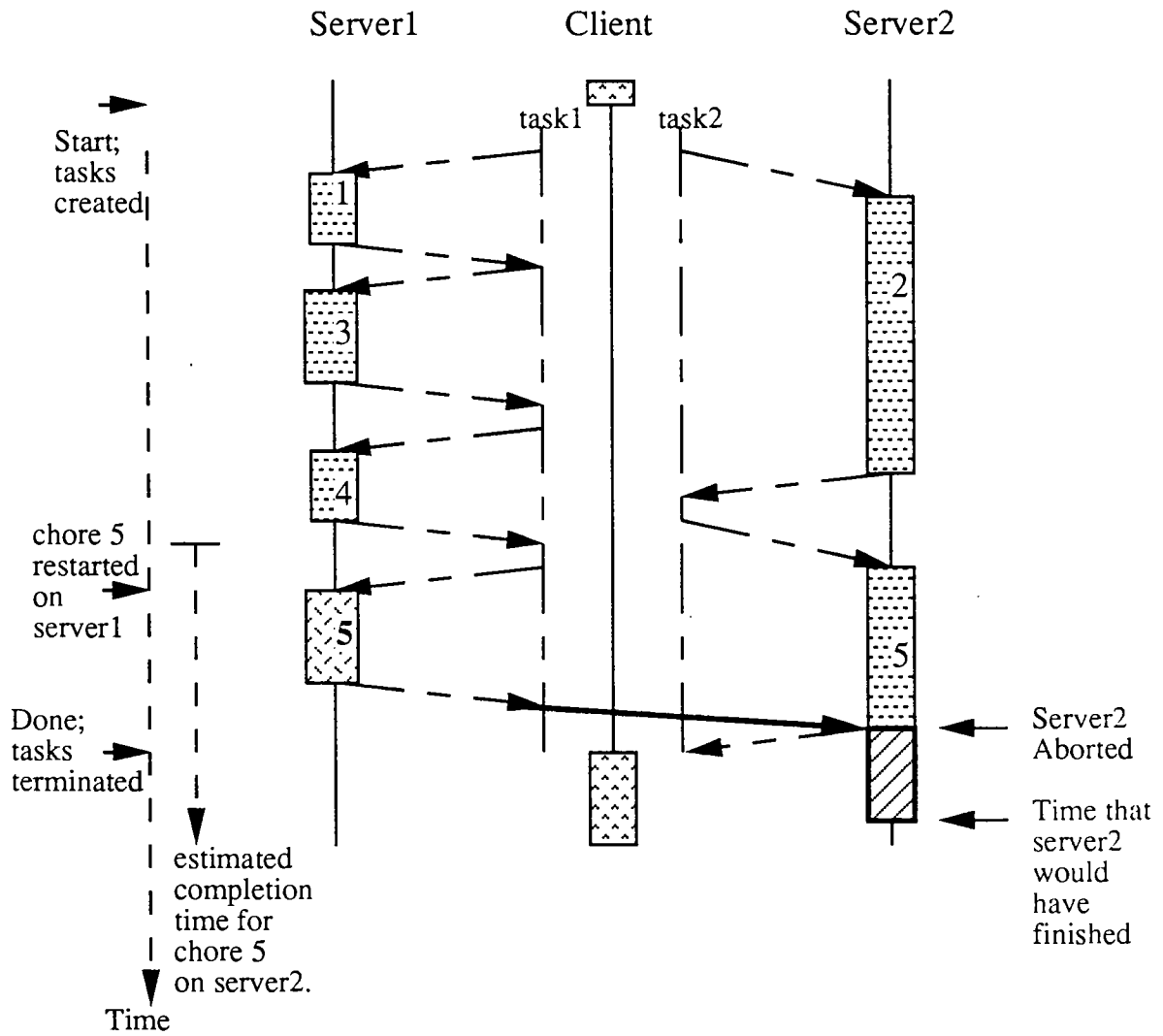
The server selection algorithm must include special logic to continue to dispense chores until all chores have been successfully completed. This means that near the end of processing, some servers will be assigned redundant chores. In other words, a presently active chore will be assigned to an idle server. All redundant servers are aborted when any server completes the

chore. This is necessary to prevent a client deadlock in the event that all servers except one have terminated normally and the last active server crashed while processing the last chore. The chore selected for redundant assignment is the chore presently assigned to an active server with the latest estimated completion time. The estimated completion time for each server is computed as the worst case chore processing time minus the time elapsed since the server started the current chore. This logic requires the client to keep a small amount of accounting history for each active server. Since the accounting information must be accessed by each client task, a critical section is declared to protect the accounting information from corruption by multiple writers.

Figure 6 illustrates a server selection scenario which includes redundant chore assignment and a server abort. The server on the left processes chores much more quickly than the server on the right. Note that the gaps between server chores have been enlarged for diagram clarity. The server utilization would typically be much higher.

### **The Chore Queue**

A temporary, dynamically allocated queue is used to spool the function arguments of doPar requests. The purpose is to allow the client DT to request service without causing it to block while waiting for the server to complete the request. The request is placed on the queue and control returns to the DT before a server has completed execution. In addition to pointers to the previous and next queue entries, the queue entry contains a pointer to a structure which contains all of the necessary arguments to invoke the RPC. Queue entries may also be inserted during client task fault handling to ensure that the request is eventually serviced even though a particular server may have crashed before completing a request that it had previously removed from the queue. Queue entries are removed when a server becomes available to service another RPC as in the self-scheduling discussion above.



The client has 5 chores to distribute among two servers. Chores are assigned on a first server available basis. After chore number 5 has been assigned, server1 completes chore 4. The client estimates the completion time for all other active chores. Server1 restarts chore 5 and completes it before server2. The client issues an abort to server2. The job is now complete; client tasks are terminated.

Figure 6. Server Selection Scenario

Within a DO loop, calls to subroutines may have more than one argument which depends on the loop index value. Either the complete set of function arguments can be temporarily stored and accessed when a chore is to be assigned or all arguments can be computed from the index value as the chore is being assigned. The queue mechanism was selected to facilitate easier integration into existing code. There is a cost for maintaining the queue which would be avoided if the alternate approach were taken.

### **Fault Handling**

All fault handling for this project is implemented with the Portable Process Fault Manager (PFM) library routines. PFM is a builtin package for Domain/OS. On other platforms, the PFM is a subset of the NCA product distribution. The PFM is divided into two fault management mechanisms: cleanup handlers and fault handlers. The major difference between the two is that fault handlers can return to the point at which the fault occurred and cleanup handlers cannot. A cleanup handler will resume execution at the first instruction following the cleanup handler code in the source file. Normally a cleanup handler would be placed at the beginning of a function so that if a fault does occur, and it is deemed non-fatal, all of the statements in the function would be re-executed thereby resetting local variables. The handlers can be chained such that the most recently declared handler will execute first, followed if appropriate, by the next most recently declared, and so on. If the default system supplied fault handler is invoked, the entire process will terminate. The cleanup handler logic must be carefully crafted such that asynchronous signals such as SIGKILL or SIGQUIT still have the desired effect on the program.

The project implementation exclusively uses cleanup handlers to simplify the re-initialization process after a fault has occurred. Each client and server program declares a cleanup handler. Further, each client task also defines its own cleanup handler. The RPC run-

time faults are managed within the task. Unexpected faults, i.e., faults that cannot be handled are passed to the next fault handler in the chain.

The client task cleanup handlers have been implemented to allow two types of faults. The first is an intentional server abort and the other is a NCS communications failure. A server will be intentionally aborted if the RPC results have already been returned by another server. NCS communication failures are treated as transient failures and the call is tried again. If a particular server gets too many communication failures, it is marked as “dead” and is no longer used for the current set of chores. The server process cleanup handler is mainly used to unregister the server from the location broker data base before the process terminates. This ensures that subsequent LLBD lookup requests will return only active servers.

### **Cross Language Considerations**

The existing CNDE numerical model software is written entirely in Fortran. The software developed for this project was written in the C language for compatibility with the NIDL generated stub files which are also C source files. The two languages have some data representation incompatibilities. C has no native definition of complex variables; Fortran does. The simplest solution in C is to explicitly define a new type which is a structure composed of a floating point real component and a floating point imaginary component. The type double complex is defined similarly except that both structure elements are double precision.

Fortran subroutine calls always pass parameters by reference. The C language supports parameter passing by value and by reference. All C functions which call Fortran subroutines must restrict argument passing to conform to Fortran conventions. By default, Fortran and C access array elements in a different order. Fortran is column major, i.e., complete columns are stored sequentially in memory. C array storage is row major. For this project, the storage arrangement is inconsequential since the C routines do not operate on the arrays passed to and from the Fortran subroutines. The consistent specification of an array starting address and the



number of elements is sufficient to allow proper communication from the client Fortran to C stub, then across the network and finally from the server C stub to server Fortran.

In UNIX, the /bin/f77 Fortran compiler will append an underscore to any external name. This is important in the context of Fortran making a subroutine call to a C function. In order for the linker to resolve the function name, the C routine must have an explicit underscore appended to the function name since the C compiler does not do this automatically. For example, the Fortran statement “CALL initNCS(…)” invokes a function which must be named `initNCS_` in the C source file. Alternatively, on the Apollo the DOMAIN Fortran compiler (/com/ftn), does not append the underscore thus the C routines must not have it. There are no function naming incompatibilities for the reverse case of a C function calling a Fortran subroutine.

### Variable Argument Lists

Variable argument lists are used to implement a consistent interface to the *doPar* function. The number, order, and the type of arguments to two distinct chore processing functions may be completely different yet it makes sense to have one function which handles all chore request queueing. The *doPar* function handles differences in calling semantics with a variable argument list declaration. It will accept any number of arguments of any type. The only restriction is that the first argument be an integer function code so that the rest of the argument list can be properly popped off the call/return stack. The arguments must be pulled off the stack manually since the compiler has no knowledge of the programmer’s intentions. In this application, the function code is the controlling variable in a switch construct. Within the switch, cases are defined for each function code. In each case, the arguments are known and they can be retrieved from the argument list and stored in a temporary structure. The address of the argument structure is copied to the queue element data field and the queue element is inserted on the chore queue.

## CHAPTER 4. IMPLEMENTATION

This chapter contains the detailed description of the software implemented for this project. First, the application layer support software is described. The support software includes the status and error logging utility functions, the processor loading daemon, the server initialization functions, and the client code to perform server selection and multi-tasking. The structure of the test programs is also described.

### Status and Error Logging Utility Functions

The status and error logging functions were written to capture the output from programs which typically run in the background, i.e., the daemon and the servers. Both `stdout` and `stderr` file descriptors are redirected to program specific files in the `/usr/tmp` directory. The files are opened in the append mode so that the information from prior executions is retained. The `errorLog` function accepts a character string and writes it to the `stderr` with a timestamp. It is possible to follow the call to `errorLog` with a call to the C formatted output function `fprintf` to record additional information such as parameter values or trace text. All fault handlers implemented for this project use both mechanisms to record the time and the fault status code. The log message timestamp facilitates tracing a sequence of event messages on multiple nodes<sup>1</sup>; this has proven to be an invaluable debugging aid. The log files may be accessed while the associated program is running via the UNIX `tail` command.

### The Processor Loading Daemon

The processor loading calculations are performed in a distributed autonomous manner. Each participating node executes its own instance of the daemon. The processor load information is periodically attained by spawning a shell which executes the BSD `/bin/csh` command `uptime`. The `uptime` command produces a string which contains the processor

---

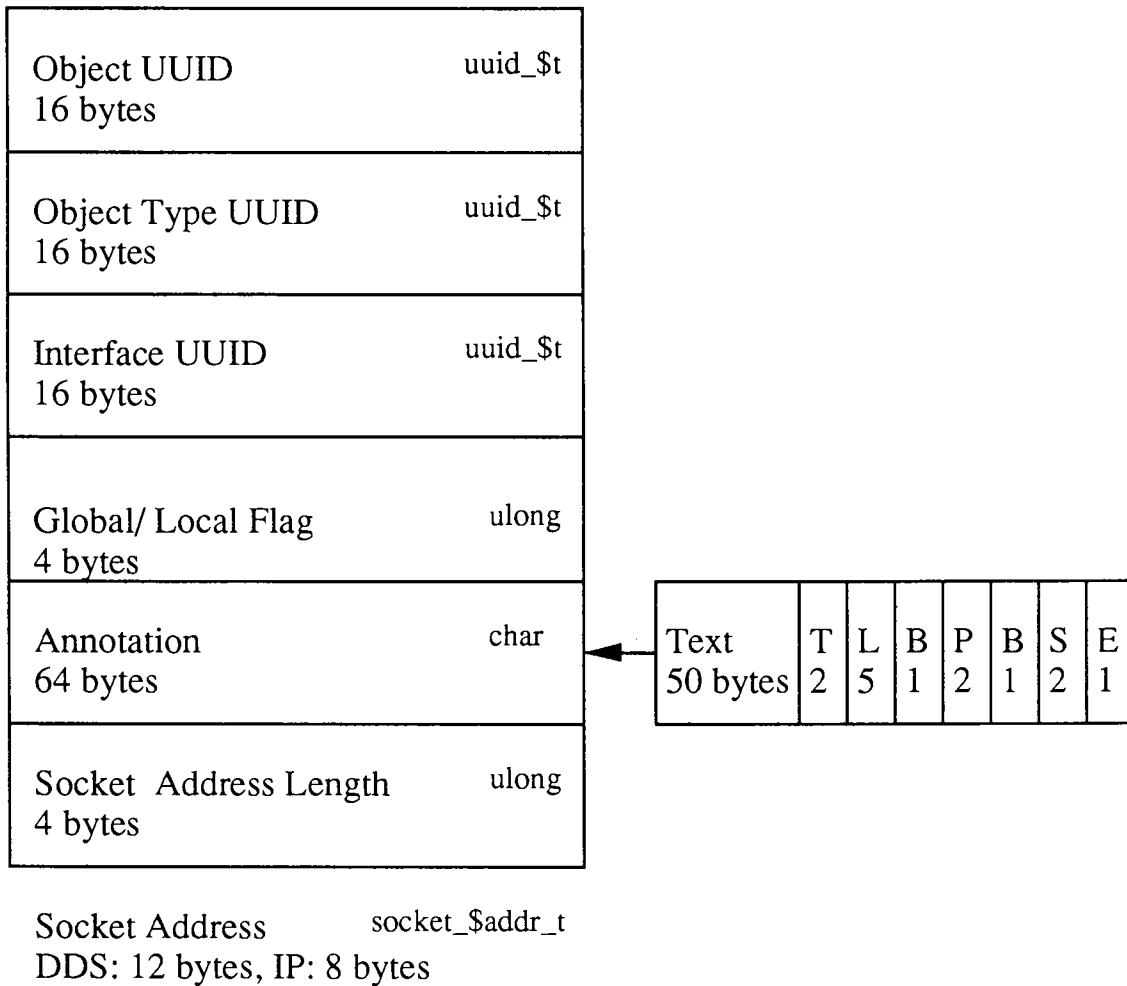
<sup>1</sup>This works to the extent that the individual node clocks are synchronized. All CNDE Apollo nodes run the UNIX time daemon (`timed`) for this purpose.

uptime, i.e., the elapsed time since the system was booted and the load averages for the preceding one, five, and ten minute intervals. The shell output is piped into the PLD which parses the string to obtain the one minute load average. This number represents a sliding average of the number of UNIX processes which were in the operating system run queue during the last minute. For this project, I assume that all users processes are running at the same priority since the UNIX priority mechanism is not well supported in Domain/OS.

The load information is maintained by the host operating system. A more direct path to acquire the information from a bona-fide UNIX system is to read it from the UNIX system table via the pseudo-device `/dev/kmem` instead of spawning a shell process [9]. This device is not available in Domain/OS. An undocumented and unsupported alternative on the Apollo is the `proc1_$get_loadav` system call which returns the required information but it is subject to change without notice. Thus, the awkward shell mechanism was selected for implementation because it is the only supported means to acquire the desired information on the Apollo.

The *cndeType* has been defined as a particular static UUID. When a server implemented in this project registers with the LLBD, it must do so with the object type field set to the *cndeType*. The PLD formulates a single LLBD query for all *cndeType* entries to obtain records for all relevant servers and exclude those LLBD records which are not maintained by the PLD. From a purely organizational point of view, it may desirable at some point in the future to declare and process additional LLBD object types. At present, one object type is adequate for the test application programs.

The 64 character LLBD entry annotation field is partitioned into five text sub-fields for this project. The length and organization of the sub-fields is shown in Figure 7. The PLD information is inserted into the load sub-field for each record received from the LLBD. Each LLBD entry is re-registered to cause a location broker database update.



Legend:  
**T:** Token 2 bytes  
**L:** Load 5 bytes  
**B:** Blank 1 byte  
**P:** Performance 2 bytes  
**S:** State 2 bytes  
**E:** Terminator 1 byte

Figure 7. LLB Entry Record Annotation Field Encoding

The current PLD update algorithm implementation specifies a load sample interval of 30 seconds and the maximum threshold value value is one job. After a measurement period, the threshold is decremented by an amount that ensures that the threshold value will reach zero after 300 seconds have elapsed. This means that at least one update will occur every five minutes. In comparison, the ARPANET routing table update algorithm implementation has a ten second measurement interval and at least one update will occur every minute [18]. The values were selected as a first guess at reasonable parameters for a quasi-static system. The parameter values which were selected may be adjusted as the run-time environment becomes known; this tuning process remains for future development.

An additional duty imposed on the PLD is to detect and remove LLBD entries which have become invalid. When the PLD initializes itself and about once per day, it verifies that the entries retrieved from the LLBD are valid by performing a NCS `rrpc_$are_you_there()` query to the server address listed in the LLBD entry. If the server does not respond within the NCS timeout period, the entry is deleted from the location broker database. Each server developed for this project declares a cleanup handler which will remove the server's entries from the location broker database when the program terminates. This handler may not get an opportunity to run if there is a catastrophic node failure such as a shutdown induced by a local power outage. The PLD ensures that the old LLBD entries are removed when the node and the PLD are restarted.

The PLD has been implemented with a selectable level of detail recorded in the program log messages. When enabled, the messages are written to `stderr` which is directed to a file as described above. There are three levels of logging. Level zero indicates that only fault information and no status information is to be written to the log file. Level one means basic information is recorded and level two means detailed traces are to be recorded. By default, level zero is enabled. In the spirit of the BIND server selectable logging mechanism, the levels

can be adjusted while the program is running by delivering UNIX SIGUSR1 and/ or SIGUSR2 asynchronous signals to the PLD. The SIGUSR1 signal causes a level increase and SIGUSR2 resets the level to zero.

### The Server Structure

The server program entry point is the initialization routine developed for this project. During server initialization, a BIND server resource record query packet is assembled to encapsulate a request for the HINFO record pertaining to the server host. The server sends the query packet to the nameserver and awaits a response. The HINFO resource record contains one field for the CPU identification and one field for the operating system identification [10, 19]. Both fields are set by the system manager when the node is configured as a network member. The processor performance metric has been encoded in the HINFO record by appending the metric to the CPU field. The buffer returned from the nameserver is parsed to extract the metric from the CPU field. The metric is inserted into a sub-field which has been allocated in the LLBD entry annotation field as shown in Figure 7.

The server continues the LLBD entry initialization by setting the processor loading sub-field to one and setting the current state sub-field to "UP". It generates a new UUID and loads the UUID into the object instance field. The LLBD type field is set to the *cndeType*. Then the server initializes all of the RPC function vectors and registers each interface<sup>1</sup> with the LLBD. If for any reason the LLBD record cannot be properly initialized, the server will terminate to prevent the PLD from parsing malformed LLBD annotation fields.

Each interface exported by a server must have a cleanup handler and an abort function declared. The abort function is necessary to support chore abort requests from a client. The abort function delivers a CPS signal to the server task which is actively processing a chore.

---

<sup>1</sup>Recall that in the NCS context, an interface refers to a collection of related functions; each RPC function exported by a server does not require its own entry.

The abort function returns to the caller and the active server task enters the cleanup handler. The cleanup handler validates the chore abort signal, terminates the task, and passes the abort status back to the client task which had initiated the chore RPC. The server then enters a quiescent state awaiting another RPC from a client.

A more elegant implementation of the server abort function would be an abort capability built into the RPC run time library which would be callable from the client. In fact, such a function exists, but it exists in name only. An invocation of the `rrpc_$remote_shutdown()` function returns a status code which is translated to mean "function not yet implemented".

### The Client Structure

The majority of the software written for this project is a collection of client support functions. This is expected because the client does all of the coordination and book-keeping for parallel processing. The client program entry point is located in the application program `per se`; it is not the `initNCS` function. The functions discussed in this section are organized in the hierarchy that they are used to implement the `initNCS`, `doPar`, and `synchServers` functions.

The two arguments passed to `initNCS` are the function code and the requested number of servers. The function code is used as the control variable to a switch construct. Within the switch, cases are declared for each valid function code. For each case, the object interface UUID is determined and the abort function pointer is set. The function code was used instead of the interface UUID directly because the UUIDs cannot be compared for equality in the switch. The abort function is also interface specific; it must be set for each case.

Potential servers are identified by sending an object interface query packet to the LLBD. Each server record received from the LLBD has the server state, load, and the performance metric encoded in the annotation field as described in the PLD and server sections. The client reads the state sub-field of each LLBD entry to verify that it is marked "UP". If it is not, the

entry is discarded. Then, the client sorts the set of remaining entries using the BSD `qsort` utility. Entries are sorted based on their processor performance to processor loading ratio. The highest ratio servers appear first in the list returned from the `qsort` operation. A RPC handle is created for each server up to the lesser of the number of servers requested and the number of servers available.

The server accounting table is cleared and one client task is created for each server. Each task begins executing immediately and establishes its own cleanup handler. Each task enters a self-scheduling loop to retrieve an entry from the chore queue and process it. The loop is exited when there are no more chores to compute. Then, the task sets its completion status, releases the cleanup handler and exits.

The `doPar` function accepts a variable argument list. Once again, a switch based on the function code is entered. Each case of the switch allocates the required amount of temporary storage for the function arguments. The function arguments are copied from the variable argument list to the temporary storage structure. Next, the chore queue is locked, the address of the temporary storage is inserted on the queue with the BSD `insque` utility and the queue is unlocked. The `doPar` function returns to the caller without waiting for the chore to be computed.

The client application calls `synchServers` to establish a rendezvous after all calls to `doPar` have been completed. The `synchServers` function sets a global flag which indicates that no more new chores are to be enqueued and waits for the client tasks to complete processing. As each task sets its completion code, the `synchServers` function releases the task. When all tasks have terminated, the `synchServers` call returns and the client program continues processing with a single thread of execution.

The logic used to dispense chores is shown in Figure 8. The basic flow is described here. If there is a chore queue entry, remove it with the BSD `remque` utility. If there is no



Begin:

```

lock choreQ
if (!first_time_for_server) {
    update ServerTable.worst_case_time
    free(memory used for previous set of call arguments)
}

if (current_server_state == SERVER_RESTART) {
    /* must be the first one finished this chore */
    abort redundant servers, set their state to SERVER_ABORT
}

current_server_state = SERVER_IDLE

if (choreQ has an entry) {
    remove; load pointer to arg structure into ServerTable
    update ServerTable.startTime, state, numberServiced
    unlock choreQ
    return (VALID)
} else if (!synchronizingServers) { /* more chores expected */
    unlock choreQ
    return(WAIT_TRY_AGAIN)
}

/* must redundantly start a currently active chore */
scan server table for chore with the latest estimated completion time
if (no servers are active) {
    set the global_done flag
    unlock choreQ
    return (DONE)
}

set ServerTable.state to SERVER_RESTART in current and worst
case server.
update ServerTable.startTime, numberServiced for current server
unlock choreQ
return (VALID)

```

End:

Figure 6. Chore Distribution Psuedo-Code

queue entry but more are expected, return a function code which instructs the requesting task to wait and try again later. Otherwise, the redundant chore logic is activated. Note that a mutual exclusion lock is required to prevent the asynchronous tasks from corrupting the global chore accounting data structure. Since BSD does not support semaphores, the lock is set and cleared with Domain/OS system service calls `mutex_$lock` and `mutex_$unlock` [4]. The chore accounting data structure is a table which indicates the address of the current argument set, the server state, the worst case processing time, and the current chore start time for each known server.

## **Test Programs**

### **Mandelbrot**

The client program for the mandelbrot application is composed of three major parts. The first part creates a display window on the workstation and loads a color map. The second part performs the `initNCS`, loads the chore queue via calls to `doPar`, and invokes tasks. The third part is the task function itself which controls a server, gets chores, and draws each scanline on the monitor. Each server program is structured as a single block of code which registers its interfaces and waits for a RPC to compute the scanline pixel values. To analyze the behavior of the system when the server computation time is very large relative to the RPC data transfer time, large RPC processing time can be simulated by artificially increasing the number of times that the scanline is computed for each RPC.

### **Linpack**

The LINPACK function selected for evaluation in this architecture is the ZGECO subroutine which factors a double precision complex matrix and estimates the condition of the matrix. The psuedo-code for the ZGECO subroutine and its subroutine calling hierarchy are shown in Figure 9. This section discusses the analysis process for the existing software and

**ZGECO Structure**

Begin:

```

Call ZGEFA           !factor matrix
Loop:
  Call ZDSCAL        !scale vector by double precision scalar
EndLoop:
Loop:
  Call ZDOTC         !Complex dot product
  Call ZDSCAL        !scale vector by double precision scalar
EndLoop:
Loop:
  Call ZAXPY         !constant *vector + vector
  Call XDSCAL        !scale vector by double precision scalar
EndLoop:
Call ZDSCAL          !scale vector by double precision scalar

Loop:
  Call ZDSCAL        !scale vector by double precision scalar
  Call ZAXPY         !constant *vector + vector
EndLoop:

```

End:

**ZGEFA Structure**

Begin:

```

Loop:
  IZAMAX             !Get index of element with max value
  Call ZSCAL         !scale vector by complex constant
  Loop:
    Call ZAXPY       !constant *vector + vector
  EndLoop:
EndLoop:

```

End:

Figure 9. LINPACK Psuedo-Code

identifies likely candidate functions for parallel execution in multiple servers. A `prof` analysis indicates that the ZGECO execution time is dominated by the time spent in the ZGEFA function which factors the matrix. ZGEFA in turn makes repeated calls to the ZAXPY subroutine to scale a vector and add it to another vector. For each ZGECO invocation, there is only one call to the ZGEFA function. Clearly, this cannot be parallelized. Within ZGEFA however, ZAXPY is called within a loop. Each ZAXPY call can be safely executed in parallel.

The execution time for ZAXPY is expected to be a linear function of the number of vector elements since the computation for each element in the resultant vector requires exactly four multiply and four addition operations. To determine if the ZAXPY routine should be implemented as a RPC function, some single processor execution time measurements were collected for various size vectors on several node types. On the DN4500, the ZAXPY execution time for a 100 element vector is 2.5 msec. On the DN10040, the same computation requires 250  $\mu$ sec. The other consideration in evaluating potential RPC candidates is the size of the interface or the number of bytes which must be transferred in each direction during the RPC. Let  $E$  be the number of vector elements. Then the size of the data which must be shipped to the server is given by  $S = (2 * E + 1) * \text{sizeof}(d\_complex) + 3 * \text{sizeof}(\text{int})$ . The amount of data which is returned to the client from the server is given by  $C = E * \text{sizeof}(d\_complex)$ . The total data transferred is  $T = S + C \simeq 3 * E * \text{sizeof}(d\_complex)$ . On the Apollo, the size of a double precision complex number is 16 bytes thus the total data transferred for a 100 element ZAXPY operation is approximately 4800 bytes. A ZAXPY prototype has been implemented as a single threaded client and server. The client has a double do loop organization. The outer loop controls the number of vector elements and the inner loop controls the number of ZAXPY operations for each vector size.

Intuitively, the ZAXPY RPCs have relatively large data transfer requirements and relatively short processing time. No performance improvement is expected if ZAXPY were to

be implemented in NCS. Test results which validate this analysis are reported in Chapter 6. The overhead costs are simply too high; it will be seen that the performance actually degrades substantially.

### **Extensibility**

At the coarsest level, this architecture can be extended through the incorporation of additional server nodes. The additional server nodes need not be binary compatible with the existing CNDE nodes; the minimum requirements for a new node are that it supports TCP/IP and NCS. If the node is to run the client, the CPS multi-tasking capability is required as well. As the node is configured to be a member of the CNDE network, a properly formatted HINFO record must be created and inserted in the BIND server database.

If additional functions are added to an interface which is already supported in this architecture, then the only files that must be updated are the NIDL source files and the application specific client and server routines. If a completely new interface is to be integrated into this architecture, then in addition to the modifications for the previous case, source files provided in this project must be updated to provide a new case in the initNCS, doPar, and the task function. The utility functions are contained in object libraries and do not require modification except for maintenance.

## CHAPTER 5. RESULTS

### RPC overhead measurements in the CNDE environment

To establish the baseline from which analyses can be made regarding the RPC data transfer time, performance measurements were made in the CNDE network. The overhead measurement methodology was adapted from Francisco and La Bossiere [11]. To measure the overhead time, single threaded null RPCs were placed between client and server pairs running on several combinations of node types. The RPCs transferred variable length arrays to the server and from the server in both idempotent and non-idempotent (at most once semantics) modes. The test results for each size vector were averaged over three trials in each direction for both modes. The vector size ranged from zero to 10,000 bytes. The performance of the DDS protocol versus the IP protocol was also measured in this manner. The nominal RPC performance of several client / server configurations is plotted in Figure 10. Figure 11 compares DDS and IP performance for two cases.

The performance plots indicate that the overhead time is a nearly linear function of the argument list size or the number of bytes transferred. The best case results occur when the client and server are co-located on the DN10040. The cost is essentially a memory to memory move. The DDS protocol performs slightly better than the IP. The difference is narrowed on the DN10040 since the TCP daemon is not competing with the client or server for the processor. One effect noticed during testing was that the NCS protocol performs best when the client and server processor speeds are closely matched. If they are not, timeouts and the associated recovery mechanisms degrade throughput.

Note that since this is an Ethernet environment, the transfer rates collected are to be regarded as “nominal” rates. The actual rates could be much worse depending on the network load. The testing was performed in a quiescent though not pristine environment. No other users were logged in during the testing. No action was taken to specifically limit the other

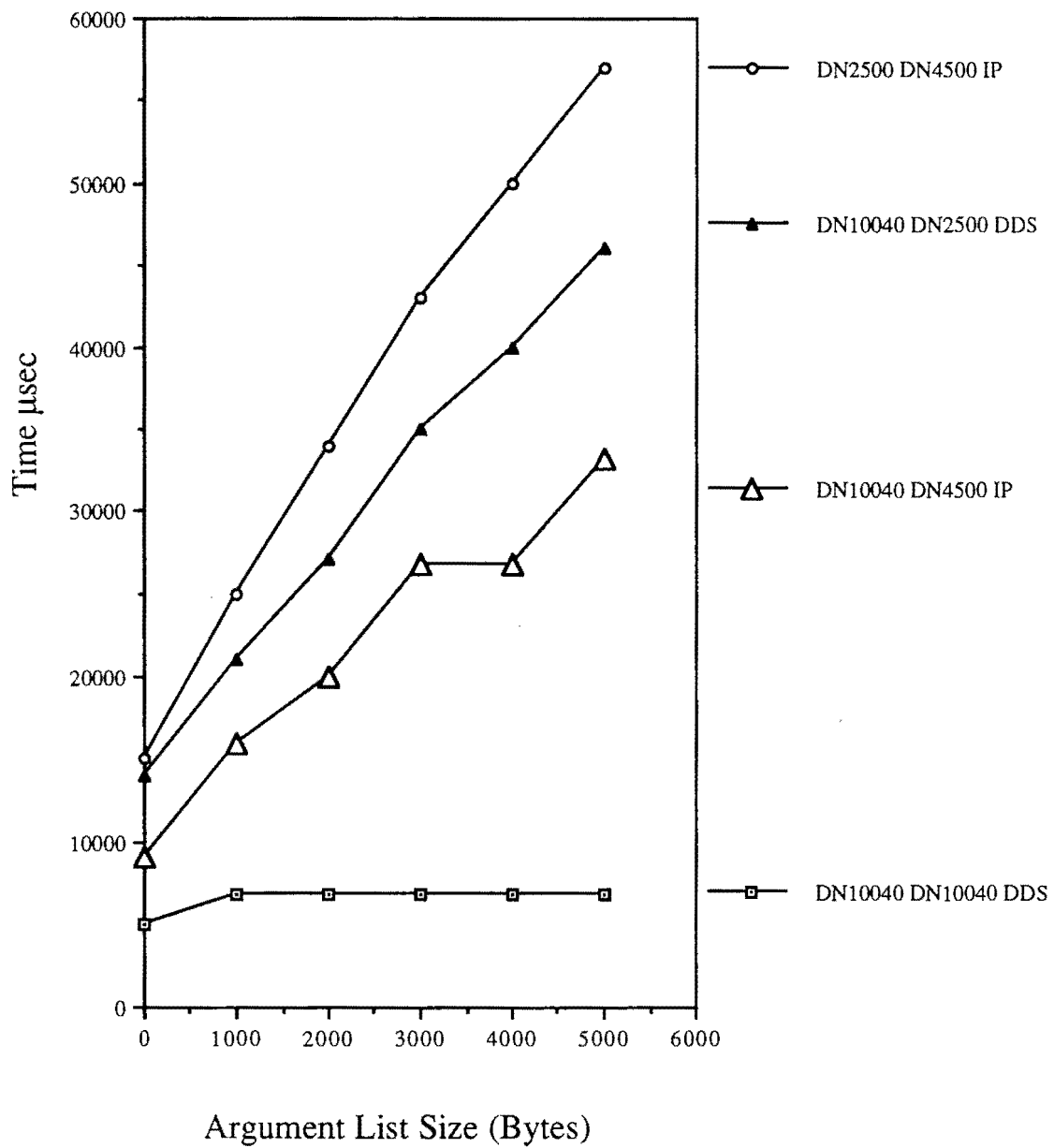


Figure 10. Nominal Remote Procedure Call Overhead

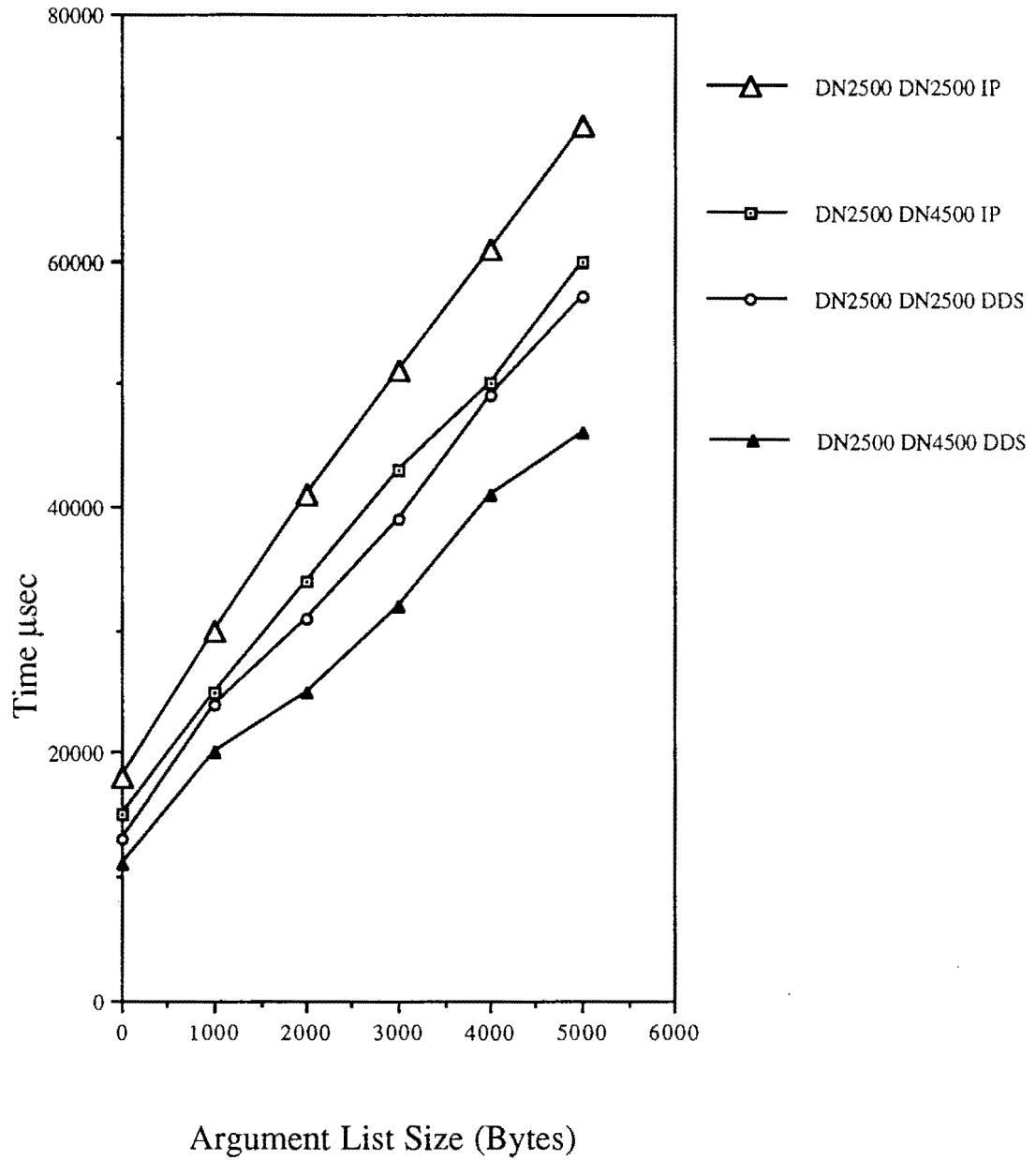


Figure 11. RPC Overhead Comparison for DDS and IP



background traffic which may have been present on the network. No NCS communication failures were reported and the hardware network adapter device error counts remained constant through the tests. The RPCs were performed with application level verification enabled such that both the client and server explicitly computed checksums on the transmitted and received data buffers.

### **Linpack Performance**

The best case RPC overhead for a null argument list is approximately 5 msec. This implies that the client could make a maximum of 200 calls per second. A more realistic figure for the overhead when a total of 5000 bytes are transferred is 35 msec; or 29 calls per second. Recall from Chapter 5 that the worst case ZAXPY computation time on a DN4500 was 2.5 msec for 100 vector elements; on the DN10040, the computation time was 250  $\mu$ sec. One hundred ZAXPY operations on the DN4500 require 250 msec. Single processor ZAXPY execution times for various length vectors are shown in Figure 12.

Testing the ZAXPY operation for a single threaded RPC has yielded some rather surprising results. The performance for several cases are plotted in Figure 13. Once again, the best case is the client and server co-located on the DN10040. For 100 element vectors, the execution time is 12.8 msec. DPAT analysis shows that neither the client nor the server CPU were fully utilized suggesting delay due to memory contention. The execution time for the client and server running on separate DN4500s yields better results than the client on a DN4500 and the server on the DN10040. This means that the NCS error recovery mechanisms for flow control errors between the client and the server cost more than the actual vector computation.

These figures are now compared to the expected time for 100 ZAXPY operations performed in parallel on the DN4500 and the DN10040. Note that during either the null RPC call or the single processor ZAXPY performance measurements, the entire CPU was dedicated

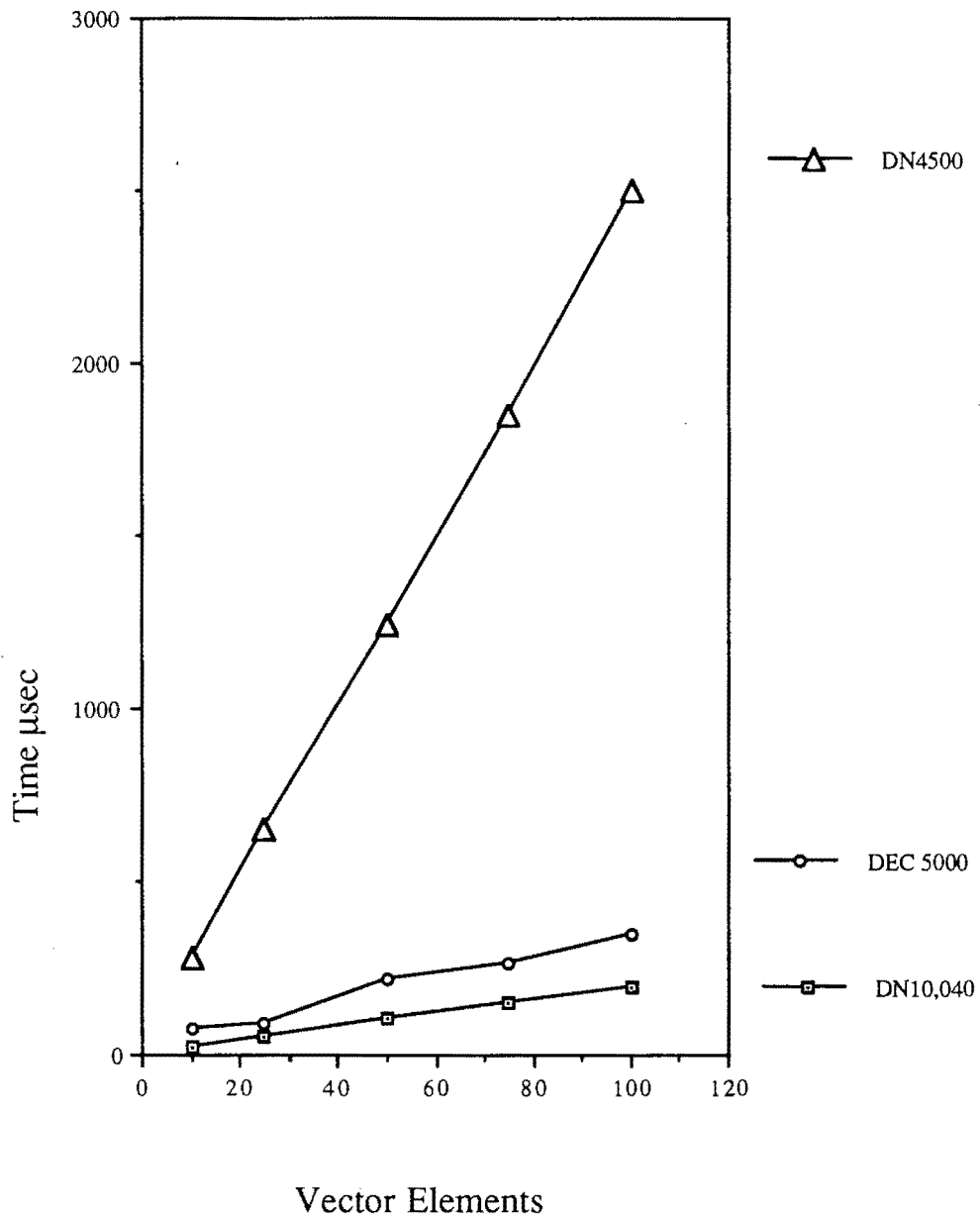


Figure 12. ZAXPY Single Processor Performance

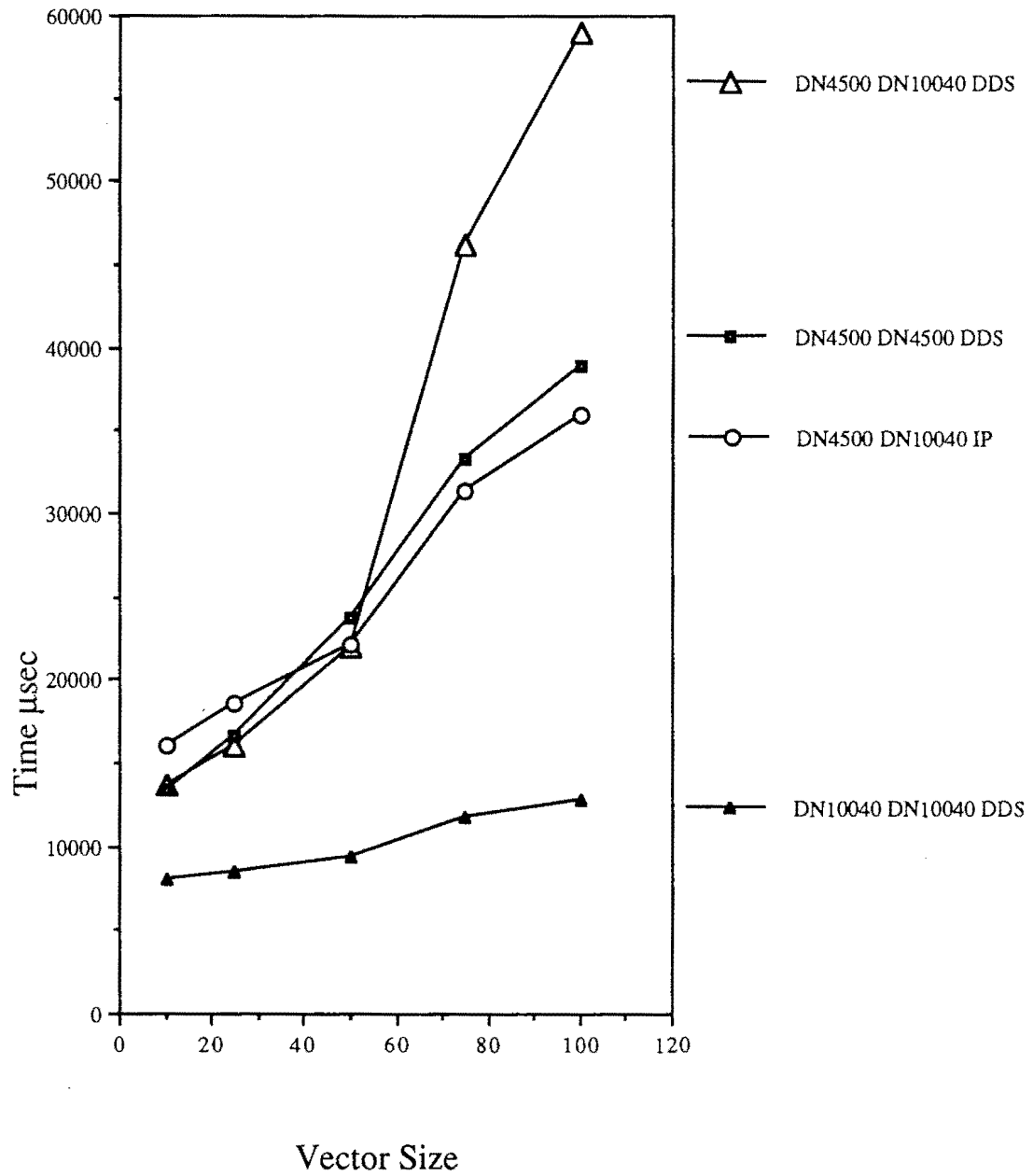


Figure 13. ZAXPY Remote Procedure Call Performance

to that one job. In this case, two tasks would be competing for the same processor on the DN4500. If the task time slice algorithm is fair, then the times listed above for data transfer and computation on the DN4500 can be expected to double. Thus, the RPC overhead from the DN4500 to the DN10040 becomes 70 msec and the execution time on the DN4500 jumps to 5 msec. The execution time on the DN10040 remains at 250  $\mu$ sec. In the first 70 msec of the parallel processing interval, 14 elements have been computed on the DN4500 and the data transfer overhead time for one element has elapsed. At 70.25 msec, 15 elements have been computed. At 140.5 msec, there have been 30 elements computed, at 210.75 there are 45 elements and so on up to all 100 elements at 471.5 msec. Note that this is nearly double the 250 msec required on the DN4500 alone. Doubling the execution time by increasing the number of processors is clearly unacceptable. This problem does not map well to this architecture because the transfer time is much greater than the computation time on either processor.

### **Mandelbrot Performance**

Most of the testing for this project was done with the mandelbrot application. The basic functions of the server selection and fault handler mechanisms were demonstrated by exercising the client and server programs and artificially inducing faults or marking a server off-line with the `/etc/ncs/lb_admin` utility. At the end of chore processing, the client displayed statistics about the number of chores processed by each server, their worst case time, etc.

Mandelbrot image generation times for a fixed set of 400 scanlines computed with several client/ server configurations were measured and are shown in Figure 14. There are several features of Figure 14 worth noting. First, the execution time on a single DN4500 workstation is an average of 454 seconds. Moving the client to the DN10040 causes a decrease of 29 seconds or 6%. Augmenting this configuration with additional DN4500 servers

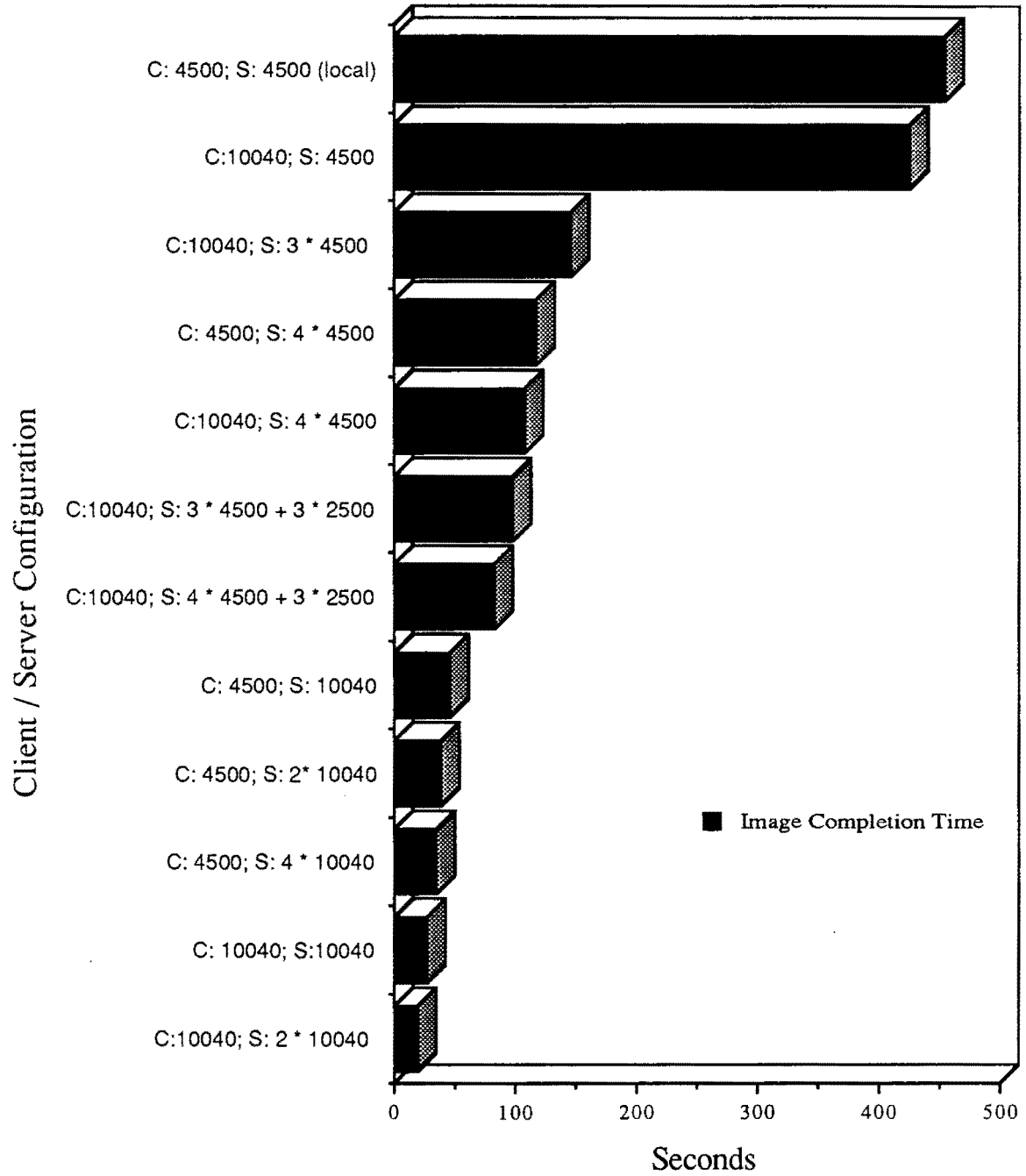


Figure 14. Mandelbrot Execution Times

scales almost linearly: three servers produce an image in 148 seconds; four servers complete in 108 seconds. Further improvements are noted if three DN2500 workstations are added to the test configuration. In this case, the total time is an average of 82 seconds for a total speedup of 82%. Note that the aggregate performance metric for the seven processors is 44. The metric for a single DN10040 processor is 22. But the execution time for the DN10040 server is 27 seconds. This number is slightly distorted by the fact that both the client and server are located on the DN10040 and the intra-processor data transfer time is much less than the data transfer time over the network as shown in the RPC overhead results section.

To test the network dependency, the client was moved to a DN4500 and tested against a single DN10040 server and also tested against a set of four DN4500 servers. In the first case, the execution time is 47 seconds; in the second, it is 118 seconds. Once again the DN10040 performs well above a set of lower performance servers. Looked at another way, four DN4500 servers driving a client on the DN10040 produce an image in 108 seconds and the same four servers driving a client on a DN4500 produces an image in 118 seconds. The best time obtained for a client on the DN4500 was provided by four servers running on the DN10040. However, the difference between four DN10040 servers and two DN10040 servers was only two seconds. This is expected from a DPAK analysis since one DN10040 server causes the DN4500 client to consume more than 50% of the host CPU and two DN10040 servers cause the client to consume more than 90%. Above two DN10040 servers, the DN4500 client was clearly saturated and could not keep up with the chores returned by the fast servers. If the client were on the DN10040, the saturation problem still exists but it is not as severe since the client processor capacity is much greater.

A version of the application was created to replicate the calculations within each server to simulate compute intensive RPC calls. The data collected from these runs was used to determine if the RPC overhead can be amortized over the computation periods to show even

more significant speedup from the parallel processing. The relative performance of the same test configurations showed no significant differences. One notable difference is the effect of the slow server abort function in configurations which mixed a DN10040 processor with some DN2500 and DN4500 processors. In almost every case where redundant chores were started, the DN10040 processor finished first and the slow speed server was aborted. In a few cases, the slow server got enough of a “head start” to finish before the DN10040 server.

The PLD performance is included in this section because most of its evaluation and analysis pertains to the Mandelbrot application testing. The log files indicate that the program only occasionally encounters an error while trying to determine the load information from the shell. The fault handler gets activated and the program recovers. In most cases, the load threshold drops to zero causing an automatic LLBD update. While not specifically tested, the CPU time charged to the processor loading daemon is on the order of 60 seconds of CPU time per day. This figure was obtained by sampling the processor status while a PLD instance was running. The processing time will vary with the number of *cndeType* entries in the location broker database and the fluctuation in each processor loading.

To determine if the time for maintaining the chore queue is significant, tests were run with and without these functions enabled. The cost for the chore queue management is approximately two seconds on the DN4500 and one second on the DN10040 which was deemed as slightly high but still acceptable. The server selection mechanism was further demonstrated by utilizing a set of lightly loaded workstations to out perform a high performance workstation which was moderately to heavily loaded. Four external jobs were started on the DN10040. Then one mandelbrot server was started on the DN10040 and the client was run on the DN4500. The total time for this processor and job configuration was 168 seconds. In this case, the loaded DN10040 performed slower than a set of four idle DN4500 servers which finished in 118 seconds.

For this application, the number of chores serviced correlates well with the performance metric divided by the processor loading for server configurations which include DN4500 and DN2500 workstations. The DN10040 processor completed more chores than would have been expected using this method. Even so, the method is useful because the ratio for the DN10040 is the highest and servers located on this processor do perform the best.

### **Problems Encountered**

There were several problems noted in the Apollo development environment. The problem with the most impact was that not all NCS functions are implemented in the RPC run time library as pointed out in Chapter four. This caused the redundant server abort logic to greatly increase in complexity. It also has a ripple effect which makes the integration of new functional interfaces more difficult because an explicit chore abort function must be defined.

The NCS run-time library is not entirely bug-free. During the overhead performance measurement testing, a few cases were encountered in which the client and server deadlocked: both sides were active but neither made any progress on the call. The RPC should have aborted due to either the packet retry count or ping count values exceeding their maximum values. Also, NCS flow control mechanisms are not effectively implemented. When a fast server and a slow client communicate or vice-versa, there is a significant amount of pinging and packet retries. This does not occur to the same extent when the client and server processing speeds are evenly matched.

The NIDL syntax is deficient in its ability to handle either more than one variable length array or two dimensional arrays in a RPC interface definition. This problem was first noted by Francisco [11]. The impact for this project was that the interface defined for the LINPACK RPC tests forced the two variable length ZAXPY source vectors to be concatenated into one larger array by the client and unpacked at the server.



Another annoying aspect of the development environment was that the UNIX `lint` utility cannot be effectively used on NCS applications. `Lint` fails with a segmentation fault while processing one of the required NCS include files. If the suspicious header files are excluded from the `lint` analysis, too many error messages are displayed for the utility to be useful.

Some bugs were also noted in the implementation of the CPS and the PFM packages. There were intermittent failures in the delivery of CPS inter-task signals which caused problems in the server abort logic. Most signaling failures simply displayed a generic run-time error message and left no traceback or core dump to assist in isolating the true cause of the problem. Also, the CPS function used by a task to give up control of the processor does not behave as described in the release notes. This has the effect of causing all chores to be queued before any chores are be assigned to servers. The PFM cleanup handlers occasionally fail to execute for no externally apparent reason. This problem was detected when the servers were stopped and their entries were not removed from the LLBD database even though a handler had been successfully established to remove them. This led to increased complexity in the PLD to periodically verify that the LLBD entries do indeed represent functional servers.

Another problem encountered but not directly addressed is the issue of portability. The multi-threaded client RPC code developed on the Apollo is not directly portable to other workstations which do not support CPS. Lack of an equivalent mechanism elsewhere would result in single threaded applications which would at best attain close to the performance of the highest compute power server in the network. In retrospect, the tasking mechanism was a poor choice because of the portability considerations. The root of the problem however is that the blocking RPC semantics are not inherently well suited for parallel processing.

## CHAPTER 6. CONCLUSIONS

### Summary

This thesis describes a new application layer architecture for use in a dynamic network computing environment. A client/ multiple server model is used to implement medium grain parallel processing. Multi-threaded clients and servers communicate via the NCS RPC facilities. The major design issues addressed are run time server selection, fault handling, extensibility, and performance. A few cases of applications were analyzed and their suitability for use in this new architecture is discussed.

The underlying Network Computing Architecture is described and compared to other current research and commercial network computing environments. The enhancements developed for use in this project were inspired by the solutions presented in these other distributed computing architectures. In particular, the concept of evaluating servers based on their expected throughput was imported from the Enterprise project. Also, the chore queue can be thought of as a sort of tuple space which is accessible to all client tasks. In comparison to Athena, this architecture has better support for parallel processing through the use of CPS. Server "bids" are accepted with much less traffic than that required in Enterprise since no broadcast messages are involved and the client can make its server selection decisions based on one LLBD query. This architecture is also much more flexible than the Emerald system since all participating programs running on a node are not mapped into the same virtual address space though one client and all of its constituent tasks are at present mapped to the same virtual address space. More than one client may be active with its own address space. The biggest advantage the architecture offers over PAX is better crash detection and error recovery. In this case, the NCS protocol will detect a communication failure due to lack of ping responses; in PAX, a client may simply deadlock while waiting for a tuple to arrive from a server which has long since crashed. ISIS offers the virtual synchronicity feature which ought to make the

design and implementation of a distributed processing system simpler. One big disadvantage of this architecture is the fact that the interfaces to access NCS and parallel processing are not as clean as they might be. A developer must do more than make a few library calls to invoke the power available in the network. One shortcoming present in all of these systems is the lack of decent development tools which would guide a developer in making decisions on program partitioning, interface sizing, server placement, etc.

This project met its goal of creating an extensible application architecture which adapts to run time conditions. This architecture has been shown to be a good environment for some classes of separable problems though as we have seen, certainly not all.

It is true that a single threaded RPC is relatively easy to implement and understand. Its major deficiency is that it has high overhead costs. For many single threaded applications, the entire program should be moved to the server and perform all local procedure calls to avoid the RPC cost. Difficulty arises when considering multithreaded RPC because there must be some mechanism to circumvent the RPC blocking semantics. Additionally, the job must be partitioned over a server pool which may have vastly different computation speeds. In this architecture, the Domain/OS CPS package was used to define multiple client threads. Each thread initiated a RPC to its own server. The disparate server computation capacities was addressed by implementing a self scheduling algorithm so that the servers can be assigned new work as they become ready.

Not all compute intensive problems can be solved with a network computing model. Message passing models like RPC appear to be entirely the wrong approach for LINPACK because of the data dependencies and the data communication requirements. A shared memory multiprocessor model is much more suitable for the current LINPACK algorithms and other functions which have short processing time and long argument lists.

This project did not set out to design new algorithms and sometimes this is exactly the approach that should be taken. The idea of plugging existing programs into a parallel architecture is often impractical; the original design may not accommodate any method other than sequential processing. There is no substitute for better algorithms; any architecture cannot exploit parallelism which simply does not exist.

### **Future Work**

Additional research is needed to develop a software tool set which facilitates distributed system development. Jordan points out the need for automated analyzers which can perform global algorithm analysis instead of limiting the scope to a subroutine [13]. His concept must be expanded from a multi-processor parallel processing environment to be applicable in a network computing architecture.

Within the realm of the architecture developed for this project, there is ample opportunity for continued development. Work remains to be done in the area of pre-loading parameter values which would be retained in the server over a set of chores thereby reducing the data communication requirements. A pre-compiler to automate the initNCS and doPar code modifications which are required to integrate new functions into the architecture should be developed. Also, the vendor MIPS ratings may not accurately reflect the anticipated application computation profile. Thus the performance metric should be based on the execution time required for representative "real" CNDE application programs. Suitable CNDE numerical modeling programs should be fully integrate into this architecture and tested.

Outside the Apollo domain, this application architecture can be extended to include clients and servers on heterogeneous platforms. Preliminary investigation performed with DEC workstations in the CNDE environment demonstrated that production release NIDL and the NCS run-time library from the two vendors are not entirely compatible. Interoperability should continue to be investigated as future production releases become available. One area in

particular which needs attention is a CPS like mechanism to support multiple servers from a client running in a standard UNIX environment.

Beyond the application architecture developed here, some representative CNDE applications should be implemented in some of the other network programming paradigms such as ISIS to compare ease of integration, flexibility, and performance.

## BIBLIOGRAPHY

- [1] Apollo Computer, Inc. Concurrent Programming Support (CPS) Reference. 1<sup>st</sup> ed. Chelmsford, MA: Apollo Computer, Inc., 1987.
- [2] Apollo Computer, Inc. Network Computing Architecture (NCA) Protocol Specifications. 1<sup>st</sup> ed. Chelmsford, MA: Apollo Computer, Inc., 1989.
- [3] Apollo Computer, Inc. Network Interface Definition Language (NIDL); Apollo Binary Release Document. Software Release 1.5.1. Chelmsford, MA: Apollo Computer, Inc., 1990.
- [4] Apollo Computer, Inc. Programming with Domain/OS Calls. Revision A00. Chelmsford, MA: Apollo Computer, Inc., 1988.
- [5] Birman, K., Cooper, R., and Marzullo, K.. "ISIS and META Projects: Progress Report." Technical Report TR90-1103. Ithaca, NY: Department of Computer Science; Cornell University, February 22, 1990.
- [6] Birrell, A. D., and Nelson, B. J. "Implementing Remote Procedure Calls." ACM Transactions on Computer Systems 2 (February 1984): 39-59.
- [7] Carriero, N., and Gelernter, D. "Linda In Context." Communications of the ACM 32 (April 1989): 444-458.
- [8] Champine, G.A., Geer, Jr., D.E., and Run, W. N. "Project Athena as a Distributed Computer System." Computer 23 (September 1990): 40-50.
- [9] Curry, D.A. Using C on the UNIX System: A Guide to System Programming. Sebastopol, CA: O'Reilly & Associates, 1989.
- [10] Dunlap, K. J., and Karels, M.J. "Name Server Operations Guide for BIND Release 4.8." Berkeley, CA: Computer Systems Research Group; Computer Science Division; Department of Electrical and Computer Sciences, University of California., 1988.
- [11] Francisco, C. R., and LaBossiere, D. "A Method for Estimating Application Performance in a Network Computing Environment Using the LINPACK Benchmark as an Example." Chelmsford, MA: Apollo Computer, Inc., May 17, 1988.
- [12] Hewlett-Packard, Inc. "Task Broker for Networked Environments based on the UNIX Operating System". Technical Data Sheet 5952-1192. Rolling Meadows, IL: Hewlett-Packard Co., 1989.
- [13] Jordan, H. F. "HEP Architecture, Programming and Performance." In Parallel MIMD Computation: The HEP Supercomputer and its Applications edited by J. S. Kowalik, 1-40. Cambridge, MA: The MIT Press, 1985.

- [14] Jul, E., Levy, H., Hutchinson, N., and Black, A. "Fine-Grained Mobility in the Emerald System." ACM Transactions on Computer Systems 6 (February 1988): 109-133.
- [15] Kong, M., Dineen, T. H., Leach, P. J., Martin, E. A., Mishkin, N. W., Pato, J. N., and Wyant, G. L. Network Computing System Reference Manual. 1<sup>st</sup> ed. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1990.
- [16] Leichter, J. VAX Linda-C Users's Guide. Order Number VLN-UG-101. Boston, MA: VXM Technologies, Inc., 1990.
- [17] Malone, T. W., Fikes, R. E., Grant, K. R., and Howard, M. T. "Enterprise: A Market-Like Task Scheduler for Distributed Computing Environments." In The Ecology of Computation, edited by B. A. Huberman, 177-206. Amsterdam, The Netherlands: Elsevier Science Publishers B.V., 1988.
- [18] McQuillan, J. M., Richer, I., and Rosen, E. C. "The New Routing Algorithm for the ARPANET." IEEE Transactions on Communications COM-28 (May 1980): 771-719.
- [19] Mockapetris, P. "Domain Names - Implementation and Specification." Internet Request for Comment 1035. Network Information Center, SRI International, Menlo Park, CA, November 1987.
- [20] Ottenstein, K.J. "A Brief Survey of Implicit Parallelism Detection." In Parallel MIMD Computation: The HEP Supercomputer and its Applications edited by J. S. Kowalik, 93-122. Cambridge, MA: The MIT Press, 1985.
- [21] Stevens, W. R. UNIX Network Programming. 1<sup>st</sup> ed. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1990.
- [22] van der Leeden, R. "First Experiences with Series 10000 Concurrent Fortran (HP-CF)" The Apollo Systems Division SE Newsletter 6 (July 9,1990) Chelmsford, MA: Apollo Computer, Inc., 1990.
- [23] VXM Technologies, Inc. PAX-1 "Creating a Network Supercomputer". Product Announcement Bulletin. Boston, MA: VXM Technologies, Inc., 1989.
- [24] Zahn, L., Dineen, T. H., Leach, P. J., Martin, E. A., Mishkin, N. W., Pato, J. N., and Wyant, G. L. Network Computing Architecture. 1<sup>st</sup> ed. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1990.

## ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my major professor Dr. Doug Jacobson for his encouragement, patience, and understanding. Program of Study Committee members Dr. Lester Schmerr and Dr. Johnny Wong deserve thanks for their contributions of time and guidance. I would also like to thank Dr. Schmerr for giving me the opportunity to conduct research at the Center for Nondestructive Evaluation.

I would like to thank my colleagues and fellow graduate students at the Center for Nondestructive Evaluation for their friendship and our many enlightened conversations. Steve Nugen provided many useful references and valuable insight into a variety of computing issues.

This thesis would not have been possible without the support of my family. I wish to thank my parents for teaching me the value of setting and pursuing one's goals. I also want to thank my wife, Susan for her tolerance and support especially during the trying times of the final few months.

This research was supported by the Iowa State University Center for Nondestructive Evaluation under NIST contract number 704 25 25.