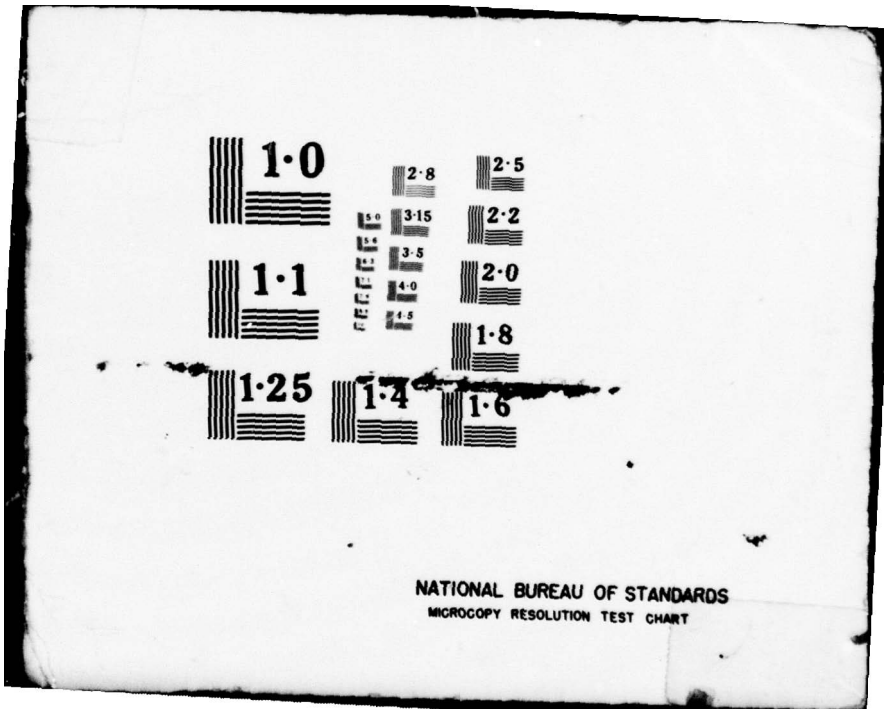
AD-A068 230 STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE F/G 9/2
A FRAMEWORK FOR PROBLEM SOLVING IN A DISTRIBUTED PROCESSING ENV--ETC(U)
DEC 78 R G SMITH MDA903-77-C-0322
UNCLASSIFIED STAN-CS-78-700 NL

1 OF 2
ADA
068230




1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
5·0
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

LEVEL I

Stanford Heuristic Programming Project
Memo HPP-78-28

December 1978

Computer Science Department
Report No. STAN-CS-78-700

AD A068230

A FRAMEWORK FOR PROBLEM SOLVING IN A DISTRIBUTED PROCESSING ENVIRONMENT

by

Reid Garfield Smith

DDC FILE COPY

DDC RECEIVED MAY 4 1979 C

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

This document has been approved for public release and sale; its distribution is unlimited.



79 05 03 123

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(14) REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: HPP-78-28 (STAN-CS-78-700, HPP-78-28)

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER (9)

4. TITLE (and Subtitle) (6): A FRAMEWORK FOR PROBLEM SOLVING IN A DISTRIBUTED PROCESSING ENVIRONMENT

5. TYPE OF REPORT & PERIOD COVERED: Technical rept.,

6. PERFORMING ORG. REPORT NUMBER: HPP-78-28 (STAN-CS-78-700)

7. AUTHOR(s) (10): Reid Garfield Smith

8. CONTRACT OR GRANT NUMBER(s) (15): MDA 903-77-C-0322

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Computer Science Department
Stanford University
Stanford, California 94305

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (12): 160 p.

11. CONTROLLING OFFICE NAME AND ADDRESS:
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Ave., Arlington, Va. 22209

12. REPORT DATE (11): Dec [redacted]78

13. NUMBER OF PAGES: 150

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
Mr. Philip Surra, Resident Representative
Office of Naval Research
Durand 165, Stanford University

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Reproduction in whole or in part is permitted for any purpose of the U.S. Government.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The concept of ***distributed problem solving***, or the cooperative solution of problems by a decentralized and loosely-coupled collection of knowledge-sources that operates in a distributed processor architecture is presented. Such architectures offer high-speed, reliable computation at low cost and are an effective way to utilize the new LSI processors and the developments of the recent synthesis of computer and communications technology. → next Page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 094 120

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

79 05 03 123

A conceptual framework called the ***contract net*** framework that specifies communications, control, and knowledge organization has been developed. Task distribution is viewed as an interactive process, a *discussion* carried on between a node with a task to be executed and a group of nodes that may be able to execute the task. This is the origin of the contract metaphor for control, where task distribution corresponds to contract negotiation.

The types of knowledge used in such a problem solver are discussed, together with the ways that the knowledge is indexed within an individual node and distributed among the collection of nodes. The use of two primary types of knowledge (referred to as ***task-centered*** and ***knowledge-source centered***) is shown.

We illustrate the kinds of information that must be passed between nodes in the distributed processor in order to carry out task and data distribution. We suggest that a common internode language is required and that the task-specific *expertise* required by a processor node can be obtained by internode transfer of procedures and data.

The use of the contract net framework is demonstrated with two implemented examples: search in the context of the N Queens problem and area surveillance by a Distributed Sensing System. Consideration is also given to the implementation of a number of familiar Artificial Intelligence problem solvers.

Features of the framework applicable to problem solving in general are abstracted from the results of this preliminary study. Comparisons with PLANNER, HEARSAY-II, and PUP6 are used to demonstrate that *negotiation*, the two-way transfer of information combined with mutual selection prior to invocation, is a natural extension to the control mechanisms used in earlier problem-solving systems.

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. SPECIAL
A

Heuristic Programming Project
Memo HPP-78-28

December 1978

Computer Science Department
Report No. STAN-CS-78-700

A FRAMEWORK FOR PROBLEM SOLVING IN A DISTRIBUTED PROCESSING ENVIRONMENT

By

Reid Garfield Smith

ABSTRACT

The concept of ***distributed problem solving***, or the cooperative solution of problems by a decentralized and loosely-coupled collection of knowledge-sources that operates in a distributed processor architecture is presented. Such architectures offer high-speed, reliable computation at low cost and are an effective way to utilize the new LSI processors and the developments of the recent synthesis of computer and communications technology.

A conceptual framework called the ***contract net*** framework that specifies communications, control, and knowledge organization has been developed. Task distribution is viewed as an interactive process, a *discussion* carried on between a node with a task to be executed and a group of nodes that may be able to execute the task. This is the origin of the contract metaphor for control, where task distribution corresponds to contract negotiation.

The types of knowledge used in such a problem solver are discussed, together with the ways that the knowledge is indexed within an individual node and distributed among the collection of nodes. The use of two primary types of knowledge (referred to as ***task-centered*** and ***knowledge-source centered***) is shown.

We illustrate the kinds of information that must be passed between nodes in the distributed processor in order to carry out task and data distribution. We suggest that a common internode language is required and that the task-specific *expertise* required by a processor node can be obtained by internode transfer of procedures and data.

The use of the contract net framework is demonstrated with two implemented examples: search in the context of the N Queens problem and area surveillance by a Distributed Sensing System. Consideration is also given to the implementation of a number of familiar Artificial Intelligence problem solvers.

Features of the framework applicable to problem solving in general are abstracted from the results of this preliminary study. Comparisons with **PLANNER**, **HEARSAY-II**, and **PUP6** are used to demonstrate that *negotiation*, the two-way transfer of information combined with mutual selection prior to invocation, is a natural extension to the control mechanisms used in earlier problem-solving systems.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under contract MDA 903-77-C-0322 and by the National Insitiutes of Health under contract RR-00785. The author was supported by the Department of National Defence of Canada.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or any agency of the U. S. Government.

Reproduced in the U.S.A. Available from the National Technical Information Service, Springfield, Virginia 22161.

Available from University Microfilm, P. O. Box 1346, Ann Arbor, Michigan 48106.

This thesis was submitted to the Department of Electrical Engineering and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

The author can now be reached at:

Defence Research Establishment Atlantic,
Box 1012,
Dartmouth, Nova Scotia,
Canada, B2Y 3Z7.
Phone: 902-426-3100.

© Copyright 1979

by

Reid Garfield Smith

Acknowledgments

I am appreciative of the support given me by the staff of the Heuristic Programming Project at Stanford, most notably Bruce Buchanan and Ed Feigenbaum. Bruce acted as my supervisor and provided personal encouragement and several helpful suggestions for strengthening the presentation of the research. He was also a great help in weaving through the bureaucratic morass prior to my oral examination. Ed Feigenbaum acted as a kind of *ex officio* advisor, continually forcing me to sharpen both the research and its presentation.

Much of the research presented in this dissertation is a result of a great deal of encouragement from Randy Davis. I am extremely indebted to him for both the interest he has shown and the care with which he has examined and refined the work over the past year.

I am grateful to Gio Wiederhold for his professional support as a member of my committee and for his personal support as a friend. I am also grateful to Susan Owicki who stepped in on short notice to act as chairman of my orals committee, and to Gene Franklin, who served as fourth reader of the dissertation.

I have received considerable personal and professional support from Jan Aikins and Tom Mitchell during the past three years.

I would like to express my thanks to a number of people who commented on various portions of this dissertation. They include Penny Nii and Jim Nourse, of Stanford, Earl Sacerdoti and Nils Nilsson, of SRI International, and Victor Lesser of the University of Massachusetts. Helen Tognetti also made several helpful comments that improved the style of the dissertation.

I feel very lucky to have had the use of excellent computing facilities, botn at SUMEX-AIM, where the research was carried out, and at the Stanford Artificial Intelligence Laboratory, where the dissertation was produced. I am grateful to the staffs of both centers.

Bernard Widrow was also instrumental in enabling me to study at Stanford and provided computation facilities for the first few months of my studies.

I would like to thank Derek Schofield, Dick Kendall, John Moldon, and the many others associated with the Department of National Defence of Canada who expended a large amount of time and effort in enabling me to come to Stanford for further studies.

Don Chiasson helped keep me in touch with reality, although only at a distance, during my time away from the Defence Research Establishment Atlantic.

Thanks are also due to Don George and Dave Coll at Carleton University, Ottawa, who whetted my appetite for research and in no small way forced me to carry on with my studies.

Finally I would like to congratulate my children for surviving what has been an especially demanding time. I have a very special debt owing to Michael and Holly.

Table of Contents

Chapter — Section — Page

Acknowledgments . . . iii

1. Distributed Problem Solving: Overview . . . 1

1.1 Introduction . . . 1

1.2 A Layered Approach . . . 3

1.3 A Cooperating Experts Metaphor . . . 4

1.4 A Framework For Distributed Problem Solving . . . 5

1.4.1 Overview . . . 5

1.4.2 Summary . . . 8

1.5 Examples . . . 8

1.5.1 Search . . . 8

1.5.2 Distributed Sensing . . . 8

1.5.3 Other Examples . . . 9

1.6 Organization . . . 9

2. Distributed Problem Solving: Motivation . . . 11

2.1 Goals Of A Distributed Approach . . . 11

2.1.1 Computational Goals . . . 11

2.1.2 Problem-solving Goals . . . 14

2.2 Architectural Considerations . . . 15

2.2.1 Multiple Processor Architectures 15

2.2.2 Processor Node Interconnection In MIMD Architectures 16

2.2.3 Loose-Coupling: Analysis. 17

2.2.4 Architectural Model 19

3. Examples 21

3.1 Example 1: Search 21

3.1.1 Distributed Search: Overview 22

3.1.2 The N Queens Problem 24

3.1.3 Contract Net Implementation. 25

3.1.4 Summary 30

3.2 Example 2: Distributed Sensing. 30

3.2.1 Hardware 31

3.2.2 Data And Task Hierarchy. 32

3.2.3 Contract Net Implementation. 34

3.2.4 Summary 50

4. A Framework For Distributed Problem Solving. 53

4.1 Communications And Control 53

4.1.1 Problem-Solving Protocols 53

4.1.2 The Contract Net Framework: Communications And Control 56

4.1.3 The Common Internode Language 58

4.1.4 Summary And Evaluation 60

4.2 Knowledge Organization 62

4.2.1 Retrieval Of Knowledge 63

4.2.2 Distribution Of Knowledge 66

5. Relations To Problem Solving In General 69
5.1 Other Systems 69
5.1.1 PLANNER And ACTORS 69
5.1.2 HEARSAY-II 70
5.1.3 PUP6 71
5.2 A Progression In Mechanisms For Transfer Of Control 71
5.2.1 Terminology 71
5.2.2 The Basic Questions And Fundamental Differences 72
5.2.3 The Comparison 72
5.3 Use Of Knowledge 75

6. The Contract Net - Systems Considerations 77
6.1 Contract Node Architecture 77
6.2 Contract Specification 78
6.3 Processing States 80
6.3.1 Commentary: Local Queuing 82
6.4 Contract Passage Through A Node 82
6.4.1 Commentary: Kernel-Size 83
6.5 Contract Net Protocol Specification 83
6.5.1 Low-Level Message Structure 85
6.5.2 Contract Messages 85
6.5.3 Request Messages 88
6.5.4 Information Messages 88
6.5.5 Commentary: Action vs. Information 88
6.6 Common Internode Language Specification 89

6.7 Message Processing Procedures 90
6.7.1 Task Announcement Processing 91
6.7.2 Bid Processing 92
6.7.3 Award Processing 93
6.7.4 Acknowledgment Processing 93
6.7.5 Report Processing 93
6.7.6 Termination Processing 94
6.7.7 Node Availability Announcement Processing 94
6.7.8 Request Processing 94
6.7.9 Information Message Processing 94
6.8 Task Structure 95

7. Summary And Conclusions 97
7.1 Distributed Problem Solving 97
7.2 The Contract Net Framework 98
7.2.1 Summary 98
7.2.2 Suitable Applications 99
7.2.3 Programming In The Framework 100
7.2.4 Limitations And Caveats 100
7.2.5 Future Perspective 101
7.3 What Have We Learned About Problem Solving? 101

Appendix A

Distributed Search Analysis 103
A.1 Searching Regular Trees: Speedup 103
A.1.1 Uniprocessor Search 105

A.1.2 Multiprocessor Search 106

A.1.3 Compositions Of Regular Trees 107

A.1.4 Speedup 107

A.2 Searching Regular Trees: Processor Requirement 111

A.3 The Maximum Speedup 115

Appendix B

Speculative Examples 117

B.1 GPS 117

B.2 STRIPS And ABSTRIPS 118

B.3 MYCIN-like Rule-Based System 119

B.4 Parsing 119

Appendix C

CNET: The Experimental Contract Net System 121

C.1 Introduction 121

C.2 Common Internode Language 121

C.3 A Sample Interaction 122

Appendix D

Glossary 135

References 141

Author Index 150

Chapter 1

Distributed Problem Solving: Overview

How can we build powerful problem solvers that effectively utilize the characteristics of distributed processor architectures? This is the central question addressed in this dissertation.

Consider, for example, a symbolic reasoning problem. Such problems generally involve search spaces that are large enough so that even the most powerful present-day computers cannot explore them exhaustively. Even when heuristics are used to prune the search space, for many problems of practical interest the computation time required to perform the search is extremely large (see, for example, discussions of **Meta-Dendral [Buchanan, 1978]**, and **CONGEN [Carhart, 1976]**).

A distributed architecture offers the potential of applying a large collection of processors to the solution of such problems. Many questions must be answered first, however:

1) Is the original problem suitable for execution in a distributed processor architecture?

2) If so, then how is the problem to be partitioned so as to take maximum advantage of concurrent computation?

3) How is internode communication to be handled?

4) What is a suitable way of effecting control over the actions and interactions of the processor nodes?

5) How can attention be focused so that execution of the problem proceeds in an efficient manner?

6) How are results to be collected and integrated?

7) What search strategies are reasonable in the new processing environment?

Some of these questions are specific to a symbolic reasoning problem, but the majority of them arise independent of the specific problem and are due to the distributed processing environment.

1.1 Introduction

Over the past several years, multiple processor architectures have come under increasing scrutiny, due both to advances in Large Scale Integrated circuit (LSI) technology **[Noyce, 1977]** and the synthesis of advanced computer and communications technology that

has resulted in networks of resource-sharing computers [Kahn, 1972] [Kimbleton, 1975]. Such architectures offer several computational advantages, including speed, reliability, and extensibility [Baer, 1973].

Low-cost small-scale LSI processors are now commonplace, and medium-scale processors are expected in the near future, with large-scale processors to follow soon after [Noyce, 1976]. Random Access Memory size is expected to increase at a similar rate. These expectations are based on two major trends: the size of an economically viable silicon die (the basis of an LSI circuit), which continues to increase year by year, and component dimensions, which continue to decrease. Thus, greater and greater numbers of components can be placed on a single slice of silicon, and fundamental physical limits appear to be at least two orders of magnitude away (see, for example, [Noyce, 1976], [Bloch, 1978], and [Faggin, 1978]).

Techniques for efficient communication between processors have been developed in the context of computer-communications networks. Nodes in such networks use the *message* as a basic unit of communication. The main results of research in this area have been communications protocols that enable reliable and efficient communication of messages between computers and efficient algorithms for routing and flow control (see, for example, [Kimbleton, 1975], [Chu, 1976], and [Green, 1975]).

The focus in this dissertation is problem solving in distributed processor architectures. A *distributed* architecture is one in which the individual processor nodes include memory as well as a processor. Communication in such architectures is generally more expensive than computation.

We will specialize our efforts towards distributed architectures in which control is decentralized, the nodes are loosely coupled (i.e., they spend a far greater percentage of time in computation than in communication with other nodes), communicate via messages, and all cooperate in the solution of a single overall problem. Distributed processor architectures with these characteristics are usefully categorized as *nearly decomposable systems* [Simon, 1969], or systems in which interactions among the components are weak but not negligible. The short-term behavior of each component is relatively independent of the behavior of the others, and the long-term behavior of each depends only in an aggregate way on the behavior of the others.

Nearly decomposable systems offer a number of problem-solving advantages, such as enabling use of the well-known *divide-and-conquer* strategy, in which a complex problem is divided into a multiplicity of smaller problems relatively independent of each other. In addition, as a result of their modularity, they offer conceptual clarity and simplicity of design. In order to gain these advantages at the problem-solving level, however, we must do more than simply interconnect a number of inexpensive processors and use the communications mechanisms that lead to distributed processor architectures. We must develop suitable techniques to control and coordinate the actions of the individual nodes so that they can cooperate to solve complex problems. We must also develop suitable techniques for partitioning problems and organizing the knowledge necessary to solve problems within a distributed architecture. In short, we must develop *problem-solving tools* that are matched to the distributed processing environment.

1.2 A Layered Approach

In the design of a distributed problem solver, the major concerns can be usefully partitioned into three layers (Figure 1.1).

At the lowest layer the focus is the underlying *architecture*. The main issues at this layer are the design of the individual nodes and the communications system that connects them. The components of an individual node must be selected (e.g., processor(s) and memory), and appropriate low-level interconnection methods must be chosen (i.e., is a single broadcast channel to be used, is each node to be connected to every other node, or are the nodes to be connected to neighbors in a regular lattice, etc.).

The middle layer focuses on the *systems* aspects of the problem solver. The concern at this level is the design of suitable communications and control mechanisms that allow coordination of the nodes. Suitable communications protocols that allow bit streams to be communicated between nodes must be designed. In addition, a local operating system must be designed to manage the resources of individual nodes.

Problem solving is the focus at the top layer. The central issue is the design of methods for partitioning and solving complex problems in a distributed manner.

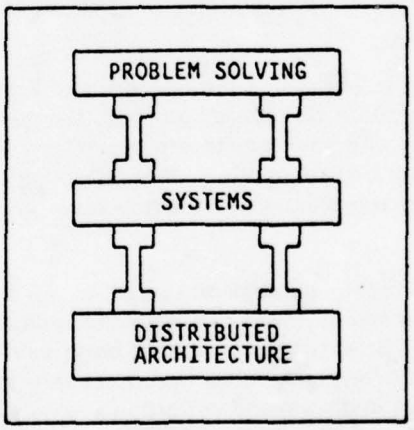


Figure 1.1. A Layered Approach To Distributed Problem Solving

How does *distributed problem solving* differ from *distributed processing*? To answer this question, let us consider the emphasis of research that has been carried out in distributed processing.

The bulk of research has been restricted to the the systems level and the distributed architecture level described above. It has generally been assumed that a well-defined and a priori partitioned problem exists, and the major concerns lie in an optimal static distribution of

subtasks, optimal methods for interconnecting processor nodes, resource allocation, and prevention of deadlock.[1] Complete knowledge of the problem has also been generally assumed (i.e., explicit knowledge of timing and precedence relations between tasks) and the major reason for distribution has been assumed to be load-balancing (e.g., **[Baer, 1973]**, **[Bowdon, 1972]**).

To construct distributed problem solvers, as opposed to distributed processors, we must maintain a focus on the problem-solving level. That is not to say that the lower levels are not important or interesting but simply that not enough research has been aimed above those levels. We now know how to design distributed architectures and how to communicate bit streams reliably between the nodes in such architectures, but we are still faced with the task of developing problem-solving methods for the distributed processing environment.

1.3 A Cooperating Experts Metaphor

A familiar metaphor for a problem solver operating in a distributed processor architecture is a group of human experts experienced at working together to complete a large task.[2] In such a situation we might see each expert spending most of his time working alone on various subtasks that have been partitioned from the main task, pausing occasionally to interact with other members of the group. These interactions generally involve requests for assistance on subtasks or the exchange of results and other information.

Of primary interest to us in examining the operation of a group of human experts is the way in which they interact to solve the overall problem, the manner in which the workload is distributed among the experts and how results are integrated for communication outside the group. No one expert is in total control of the others (although one expert may be ultimately responsible for communicating the solution of the top-level problem to the *customer* outside the group).

When an expert encounters a subtask too large to handle alone, he partitions it into manageable sub-subtasks and makes them known to the group. If he encounters a subtask for which he has no expertise, he attempts to pass it on to another more appropriate expert. If the expert knows which other experts have the necessary expertise, he can notify them directly. If the expert does not know anyone in particular who may be able to assist him or if the new subtask requires no special expertise (and he was other more pressing subtasks to be executed), then he can simply describe the subtask to the entire group.

If another expert chooses to carry out the subtask, he requests details from the original expert and the two engage in further direct communication for the duration of the subtask. In order to distribute the workload in a group of experts, then, those with subtasks

---

[1] Implicit procedure calls of the sort used in **PLANNER [Hewitt, 1971]** and other *AI* languages, that involve nondeterminism at runtime, have not been considered.

[2] This metaphor has also been used as a starting point by **[Lesser, 1975a]**, **[Lenat, 1975b]** and **[Hewitt, 1977a]**, but has resulted in systems with different characteristics than that considered here. The different approaches are compared in Chapter 5.

to be executed must find other experts capable of executing their subtasks. At the same time, it is the job of idle experts to find suitable subtasks on which to work. Those with subtasks to be executed and those capable of executing the subtasks thus engage in a kind of negotiation to distribute the workload. They become linked together in subgroups of varying sizes by agreements or informal *contracts*, forming and breaking up dynamically during the course of work.

Subgroups of this type offer two advantages. First, communication among the members does not needlessly distract the entire group. Second, the subgroup members may be able to communicate with each other in a language that is more efficient for their purposes than the language in use by the whole group.

1.4 A Framework For Distributed Problem Solving

The central result of the research reported here is a framework for problem solving in a distributed processing environment. This framework specifies mechanisms for communications, control, and knowledge organization.

1.4.1 Overview

A framework in this context has three major conceptual components: a *control* component that specifies the modes of interaction possible between processor nodes; a *knowledge organization* component that specifies how knowledge is organized in individual nodes and distributed throughout the collection of nodes; and a *communications* component that links the other components together and provides the base that enables nodes to interact.

A key problem that must be addressed by the framework is how nodes with tasks to be executed find other nodes capable of executing those tasks. We will call this the *connection problem*. (In centralized problem solvers it is called the *invocation problem*; that is, which *knowledge-source* (KS) to invoke at any given time for the execution of a task.) Because *AI* applications do not generally have well-defined algorithms for their solution, *AI* problem solvers need considerable heuristic knowledge to guide them, making the connection problem crucial.

As noted in Section 1.3, the connection problem has two parts: Nodes with tasks to be executed seek other nodes capable of their execution, and nodes not engaged at any given time seek tasks that they can execute. In fact, both sets of nodes can proceed simultaneously, with the nodes engaging each other in a process of negotiation to solve the connection problem. It is precisely this notion of negotiation that is the basis for the control component of the framework. Task distribution is a process of contract negotiation, or mutual selection between the nodes that have tasks to be executed and the nodes suited to the execution of those tasks. This process gives rise to the name--the ***contract net framework***. The collection of nodes is referred to as a contract net. A node with a task to be executed is called a ***manager***, and a node capable of executing a task is called a ***contractor***. Manager and contractor are roles taken on dynamically by nodes during the course of problem solving.

Because communications and control are so closely linked, they will be discussed together. These components are jointly composed of (*i*) a *problem-solving protocol*, called the *contract net protocol*, an extrapolation to the problem-solving level of the standard network communications protocol; (*ii*) a set of *message-processing procedures* for dealing with the messages of the protocol; and (*iii*) a *contract structure* that handles the bookkeeping required for task distribution and node interaction (Figure 1.2).

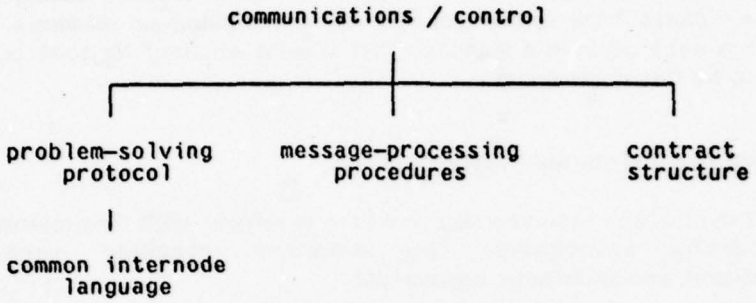


Figure 1.2. The Communications And Control Components Of The Framework.

The protocol is a good example of intertwined control and communications. It encodes task-independent information that specifies the possible actions and interactions of individual nodes of the problem solver and also includes slots for the task-dependent information necessary for the decisions that guide the control. The task-dependent information in the framework is encoded in a formal *common internode language*, understandable to all nodes. The use of a formal language, as opposed to a more *ad hoc* approach, simplifies both the transfer of expertise between nodes and the dynamic addition of new nodes to the problem solver (see Section 4.1.3).

The message-processing procedures carry out the actions specified by the messages of the protocol, as indicated by the task-independent information in those messages. For example, a ***task announcement*** message is used to advertise tasks waiting to be executed. The corresponding message-processing procedure for this announcement handles the basic actions of a node that receives such a message (ultimately such a node must decide whether or not to submit a bid on the task). The message-processing procedures interact closely with user-procedures (i.e., task-dependent procedures that are written by a user of the framework) by providing these procedures with the appropriate task-dependent information from the protocol messages, together with additional task history that helps maintain the focus of the problem solver. For example, one of the questions that must be answered by a node that receives a task announcement message is *Do I wish to submit a bid on this task?* The answer is task-dependent and so cannot be answered directly by the message-processing procedure. That procedure can, however, supply a user-procedure with the necessary task-dependent information from the task announcement called a ***task abstraction***, together with a list of other recently received task abstractions, so that a decision can be made. Thus the message-processing procedures provide structured information and structured questions to the user-procedures.

The message-processing procedures use the third part of the communications and control components: the contract structure. This structure organizes the required bookkeeping information--e.g., information about the execution of tasks, relations between tasks, and the names of nodes that are executing tasks. This information is accessed and updated by the message-processing procedures.

The knowledge organization component of the framework specifies both the manner in which knowledge in the distributed problem solver is to be retrieved by a node, either from within its own local knowledge base or from the knowledge bases of remote nodes (*retrieval*), and how it is to be distributed among the nodes (*distribution*) (Figure 1.3).

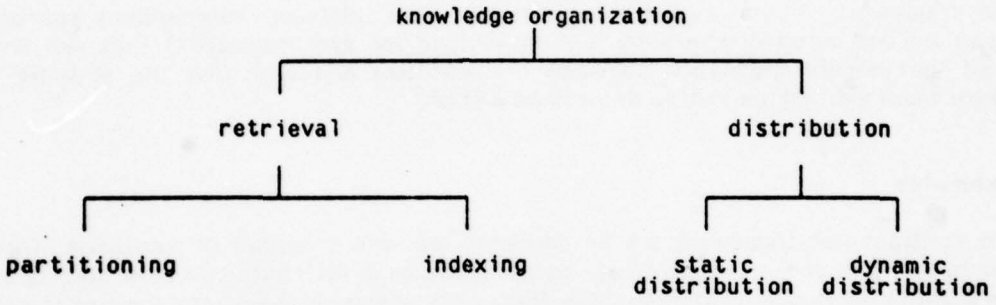


Figure 1.3. The Knowledge Organization Component Of The Framework.

As shown in the figure, retrieval can be further broken down into two parts: *partitioning* and *indexing*. Partitioning indicates the ways in which the knowledge is broken up into modules; indexing indicates the handles placed on the knowledge modules so that they can be accessed. Knowledge in the contract net framework is partitioned so as to minimize internode communication and to simplify the aquisition of knowledge by an individual node and the distribution of knowledge among the collection of nodes. Two major types of knowledge are recognized for indexing: *task-centered knowledge* and *knowledge-source-centered knowledge* (KS-centered knowledge). Task-centered knowledge is useful for finding nodes capable of executing tasks and KS-centered knowledge is useful for finding tasks to be executed by a node.

Distribution also has two aspects: *static distribution*, how knowledge is initially loaded into the nodes, and *dynamic distribution*, how knowledge is transferred between nodes as work on the overall problem proceeds. Among the kinds of knowledge that can be distributed dynamically as well as statically is knowledge about how to execute tasks. The contract net protocol and message-processing procedures enable both kinds of knowledge distribution.

1.4.2 Summary

The contract net framework enables problem solving to be carried out in the distributed processing environment in such a way as to ensure that the computational benefits of distributed processing are retained at the problem-solving level. It encourages concurrency at the task level. Relatively large code segments, rather than single operations, are assumed to execute concurrently. Connection (or invocation) is handled as a local contract negotiation--a mutual selection process in which both managers and contractors participate. This enables the nodes to make better invocation decisions because nodes participating in the negotiation have available more information about each other than is traditionally the case in *AI* problem solvers that do not use negotiation.

The framework is well suited to applications with relatively independent subtasks (which can be executed concurrently with little need for synchronization) that are large enough so that careful invocation decisions are important and such that the specific KS required for each subtask cannot be determined a priori.

1.5 Examples

The contract net framework will be demonstrated with a number of examples. These examples have been chosen to draw out underlying issues in distributed problem solving and illustrate a range of design features of the framework across different problems of varying levels of complexity. The first two examples have been implemented in an experimental contract net simulation, while the remaining examples have been examined as a pencil and paper exercise.

1.5.1 Search

The first example is the N Queens problem, where the goal is to place N queens on an $N \times N$ chessboard in such a way that no two are on the same row, column, or diagonal (i.e., none of the queens can capture any of the others). Since search is one of the major paradigms for *AI* problem solving, it is useful to demonstrate the use of the new framework on such a problem. The particular search problem considered is simple enough to demonstrate the essence of the contract net approach without requiring the full complexity of the design.

1.5.2 Distributed Sensing

The second example demonstrates the use of the contract net framework in the solution of a problem in area surveillance, such as might be encountered in ship or air traffic control. The operation of a Distributed Sensing System (DSS) is considered, that is, a network of nodes, each of which has sensing or processing capabilities, and are spread throughout a relatively large geographic area. The primary aim of the network is formation of a dynamic map of traffic in the area showing vehicle locations, classifications, courses, and speeds.

The DSS example is intended to show the utility of the contract net protocol in a more complex application, along with the types of information that are transferred between nodes during problem execution. It also offers some insight into the organization and use of knowledge in a distributed problem solver.

1.5.3 Other Examples

The other examples discussed in the dissertation are hypothesized and more speculative. They demonstrate the ways in which familiar *AI* problem solvers could be (but have not been) implemented in the contract net framework. The examples include GPS [Ernst, 1969], STRIPS [Fikes, 1971], and MYCIN [Shortliffe, 1976], among others. Each shows the advantages offered by distributed problem solving and the utility of task distribution as a negotiation process, together with the utility of task-centered and KS-centered knowledge.

1.6 Organization

The research presented in the following chapters is based on the premise that distributed processor architectures offer significant computational and problem-solving advantages over single processor architectures. In Chapter 2 we therefore step back to list and examine these advantages, considering some of the lower level questions that affect the performance of distributed problem solvers.

Chapter 3 and Chapter 4 discuss the individual components of the framework together with their instantiation in detailed examples.

Chapter 5 compares the contract net with approaches taken in earlier *AI* problem-solving systems, such as PLANNER [Hewitt, 1972], HEARSAY-II [Erman, 1975], and PUP6 [Lenat, 1975b]. This comparison shows the contract net to be a natural extension of the earlier approaches.

Details of the framework are presented in Chapter 6. This chapter lays out the specification of the protocol, the common internode language, the message-processing procedures, and the contract structure.

Chapter 7 summarizes the central themes of the dissertation. It reviews the distinguishing features of distributed problem solving and the contract net framework. It also discusses both limitations on the framework and guidelines for its use. Finally, it draws conclusions about features of the framework that are of value to problem solving in general, as well as in distributed problem solving.

Appendix A presents bounds on the time required for a distributed search of a regular tree. Also derived are bounds on the number of processor nodes required to search a regular tree.

Appendix B presents a number of brief, speculative examples of *AI* problem solving in the contract net framework.

Appendix C describes CNET, the experimental contract net system. This system enables a user to write programs that are executed on a simulated distributed architecture. User programming requirements are discussed and a sample interaction is shown.

Appendix D is a glossary.

Chapter 2

Distributed Problem Solving: Motivation

In this chapter, the assumptions behind the distributed problem solving approach are reviewed. The goals of a distributed approach are discussed and the characteristics of a problem that make it suitable for a distributed approach are noted. We also consider briefly the design of a distributed architecture, emphasizing processor node interconnection and the advantages of a loosely coupled design.

2.1 Goals Of A Distributed Approach

The goals of a distributed approach can be characterized along two primary dimensions: *computational* goals and *problem-solving* goals. Computational goals are related to actual execution of the subtasks of a problem; problem-solving goals are related to the methods and strategies used to partition a problem into subtasks and to guide the actions of the problem solver as execution of the subtasks proceeds.

2.1.1 Computational Goals

2.1.1.1 Speed

The traditional means of increasing speed have been to develop faster primitive logic elements and to utilize concurrency (i.e., take advantage of the fact that many computations can be configured so as to allow several parts of the computation to be done concurrently). The first method results in the construction of larger, faster central processors and memories, which unfortunately suffer from problems of cost, complexity, and reliability. The recent availability of simple and inexpensive LSI processors has made the second method increasingly attractive.

Concurrency can be introduced in a computer system at the levels of individual bits, single operations, or entire tasks.[1] These successive levels involve increasing complexity in the concurrent functions. *Bit-level concurrency* is common to all contemporary computers--groups of bits are processed together as words in elementary operations like addition and subtraction. *Operation-level concurrency* involves more complex functions, such as floating point arithmetic operations which may be overlapped with the execution of the main program sequence. *Task-level concurrency* involves still greater complexity in the functions, such as input/output (e.g., files can be listed while main program execution continues).

LSI processors make task-level concurrency increasingly attractive because such processors are inexpensive and have the capability to carry out complex functions. This

[1] This breakdown roughly follows that of [Kuck, 1975].

type of concurrency is central to the distributed approach to computation. As a result, distributed processing has been suggested as an alternative mechanism for obtaining high-speed computational power [Baer, 1973], [Faggin, 1978].

The potential impact of this type of processing power is especially evident in *AI* systems because they often search large spaces attempting to solve problems, as in chess [Newell, 1963] and speech understanding [Reddy, 1975]. Such problems can readily exhaust the computational resources of present-day single processor architectures.

Because the distributed approach exploits task-level concurrency, it is best applied to problems that lend themselves to decomposition into a set of relatively independent tasks with little need for global information or synchronization. A problem that meets these requirements allows maximum task-level concurrency to be achieved, since individual tasks can be assigned to separate processor nodes and these nodes can execute the tasks with little communication with other nodes.[1]

If the problem is such that its individual subtasks are generated uniformly over the course of problem execution, this leads to the most efficient and effective use of concurrency because it allows a uniform number of processor nodes to be used for the duration of the problem. If uniform generation of subtasks is not possible, problems in which most of the subtasks are generated early on in the execution lead to a greater potential for concurrency (and thus higher speed) than problems where most of the subtasks are generated late. This consideration is especially important when the number of subtasks generated exceeds the number of available processors. If the bulk of independent subtasks are generated early on, then it is possible to queue some of them for later execution without an increase in execution time for the overall problem. If they are generated late in the problem execution, however, such queuing will generally result in an increase in execution time for the overall problem.

If communication is expensive relative to computation then problems must be carefully partitioned in order to minimize communication between processor nodes.[2] Minimization of overall communication is a primary design goal for distributed problem solvers. In addition, if tasks are distributed to processor nodes dynamically, then they should be amenable to succinct description in order to further reduce the amount of communication that must be carried out between nodes.

It is also preferable for all of the subtasks to be homogeneous in form (i.e., a uniform modular problem decomposition). Subtasks in such a decomposition can be executed by any available processor node. There is thus a requirement for a minimum of functionally specialized nodes (i.e., nodes that are suitable for processing only particular types of subtask), and the processing load can be uniformly shared by a number of identical and interchangeable processor nodes. This is beneficial because specialized nodes can lead to bottleneck problems.

---

[1] Problems that do not meet this requirement will not benefit significantly from a distributed approach (see, for example [Minsky, 1971]).

[2] This is generally the case because the communications channels that link processor nodes operate at much lower speeds than do the nodes themselves. A more detailed analysis is presented in Section 2.2.2.

AI search problems are particularly amenable to the use of task-level concurrency, since a typical strategy is the recursive decomposition of problems into subsidiary problems which themselves may be suitable for decomposition [Nilsson, 1971]. Large numerical problems, such as those found in image-processing applications, are also suitable because they involve a large number of subtasks that are almost independent [Reddy, 1973], [Barrow, 1976].

A distributed approach also has the potential to achieve high speed in another way--by improving real-time response in applications that have a characteristic natural distribution due, for example, to spatial separation in their areas of activity. The improvement can be achieved through relocating of critical processing functions so that response time is not necessarily restricted by low speed communications channels or overburdened central computing sites. This has been the major use of distributed processing systems to date, for example in the banking industry [Foster, 1976].

For this benefit to be achieved, the problem must exhibit a large amount of *computational locality*; that is, there must be a high probability that relevant knowledge is resident at the node that needs it, so that communication between nodes can be minimized. As an example, consider a user who wishes to query a database that is stored in pieces at a number of processor nodes. If there is a processor close by the user and if the query can be answered by that processor on the basis of information held in its own local portion of the database, then there is no need for communication with remote processor nodes.

2.1.1.2 Cost

A related computational goal is achievement of high-speed computation at a lower cost than that of a single processor architecture of comparable speed. This goal now appears attainable mainly because of the advent of LSI processors. It may now be cost-effective to achieve high-speed performance by interconnecting many low-speed machines (see, for example [Jensen, 1975]).

A distributed approach to problems that have a natural decomposition can also lead to a decrease in the cost of communications channels. This is due to the fact that the distribution of processing resources can lower the requirement for high bandwidth channels, which are considerably more expensive than low bandwidth channels.

Finally, a distributed approach can be used to share computing resources, as has been done in the ARPAnet [Kahn, 1972]. One of the resulting benefits can be balancing of the computing load at a number of geographically separated sites [Bowdon, 1972]. Such load-balancing boosts throughput, since processors that might otherwise stand idle for want of locally generated tasks can be used for execution of remotely generated tasks. It can also reduce costs by sharing expensive, specialized resources (e.g., ILLIAC IV) over a larger community [Roberts, 1970].

2.1.1.3 Reliability

Distributed architectures are an effective means by which to achieve reliable computation. They offer the potential for smooth deterioration of performance in the face of individual component failures, mainly due to the minimization of shared, centralized resources. This capability is called *graceful degradation* [Baer, 1973].

2.1.1.4 Extensibility

Another goal that appears attainable through a distributed approach is extensibility--tuning the potential processing power to meet the demands of a particular task by incremental addition or deletion of processor nodes or communications channels [Widdoes, 1976]. This capability allows a smooth transition to be made in the *size* of the computer system when there is a need to handle larger problems. It also makes possible to alter the configuration of processor nodes and communications channels of a distributed architecture in response to changing problem demands [Baer, 1973].

2.1.2 Problem-solving Goals

The problem-solving goals of a distributed approach are not in general as well understood as the computational goals, but a few principles can be identified.

For example, there is a well-known problem-solving strategy of breaking up large problems into smaller, independent subproblems, and then solving the subproblems, called *divide and conquer* [Wicklegren, 1974]. This problem-decomposition strategy offers greater conceptual clarity from the point of view of the programmer. It also enables better evaluation of overall system performance and assessment of the contributions and interactions of individual knowledge-sources. This is true because the overall system can be tested with varying numbers and configurations of knowledge-sources (see, for example [Fennell, 1975]).

Large uniprocessor software systems are typically very difficult to debug and to prove correct, resulting from lack of modularity. Distributed systems, on the other hand, enforce a modular approach, thus leading to systems that are more comprehensible and easier to debug [Connell, 1976]. This type of modularity has proved useful in the construction of *AI* learning systems that use debugging as the primary means of improving their performance [Sussman, 1973].

The use of a distributed approach can lead to a method of problem solution that is natural for a given problem, as in problems that involve inherent geographical separation in their information bases, or areas of activity. Acquisition of signal data from multiple sensors and the reduction of this data to symbolic form is an example of such a problem [Nii, 1978]. More generally, this is true of problems that have either a spatial distribution or a large degree of functional specialization. Understanding continuous speech is an *AI* problem with a large degree of functional specialization; information from many different knowledge-sources (e.g., signal processors, parsers, etc.) must be combined to solve the problem. Such naturally

distributed problems are such that a single knowledge-source or node lacks an overall perspective.

Distributed processing systems can increase the amount of search that can be practically attempted to solve a problem. The capability to do more search increases the power of a problem solver in two ways. First, it reduces the number of paths that must be pruned, reducing the possibility that a valid path is falsely pruned. Second, it makes the problem solver less sensitive to path ordering, since multiple paths can be explored at the same time.

2.2 Architectural Considerations

Because the focus of this dissertation is problem solving in a distributed architecture and not the design of the architecture, we will not dwell at length on the last topic. It is useful, however, to discuss briefly the architectural considerations as they relate to the problem-solving aspects of distributed systems.

2.2.1 Multiple Processor Architectures

There are two major classes of multiple processor architecture: *Single Instruction Stream Multiple Data Stream* (SIMD), and *Multiple Instruction Stream Multiple Data Stream* (MIMD).[1]

A SIMD architecture contains a control processor that drives several task processors (and their associated memories) in a *lockstep* manner; that is, the control processor issues a single instruction stream to all task processors which execute it in synchronism for their own individual data streams [Stone, 1975]. The SIMD architecture is best-suited to computations that can be cast as vector computations, since all task processors use the same instruction stream.

The applications of interest here, however, cannot generally be cast as vector computations (e.g., *AI* search problems). They therefore require a more flexible architecture, of the MIMD variety.

A MIMD architecture is composed of a collection of processors and their associated memories, with the processors interconnected to provide a means for communicating [Stone, 1975]. The MIMD architecture is more flexible since each processor has the capability to execute an independent instruction stream. Successful application of MIMD architectures depends strongly on the way that tasks are partitioned for execution and on the ease with which communication can be carried out between processors.

---

[1] This terminology is due to [Flynn, 1972]. A *stream* is a sequence of instructions or data that is executed by or operated on by a processor.

2.2.2 Processor Node Interconnection In MIMD Architectures

A major problem that arises in the design of MIMD architectures is the interconnection of nodes [Anderson, 1975]. The recent alteration in the relative costs of computation/storage and communication caused by LSI technology advances has placed a premium on efficient processor interconnection. It is important that the high performance/price ratios of current LSI processors and memories not be degraded by costly or inefficient interconnection mechanisms.

Complete interconnection (so that a node can communicate directly over a private channel to every other node) is extremely expensive because it entails a number of channels proportional to the square of the number of nodes. In addition, it is expensive (and difficult) to add a new node to a completely interconnected network, since each such node requires the addition of n new cables to connect it to the existing nodes.

At the other extreme lies the use of a single broadcast communications channel that is shared by demand-multiplexing; that is, a node acquires the channel on demand (usually after checking to make sure that the channel is not in use), and transmits a message. In this case, addition of a new node is a relatively simple procedure, involving a single connection to the shared channel. Unfortunately, a broadcast channel can be a source of contention and delay when the number of processor nodes is large [Metcalfe, 1976].

There are many other interconnection mechanisms (see, for example [Anderson, 1975]) and several trade-offs between them, such as cost of adding a new node, reliability, and speed. For each of these mechanisms, however, it is generally the case that internode communication is expensive and slow relative to computation within a node. It is therefore desirable to minimize the requirement for internode communication if a large number of processor nodes is to function together effectively. Such systems are called *loosely coupled* [Lesser, 1975b].

Loose-coupling can be effected by careful partitioning of the top-level problem into tasks that are relatively independent of each other and that require large processing times with respect to the time required for internode communication. We will see that the control formalism used to coordinate the actions of the individual nodes (Section 4.1.2) and the knowledge organization of the system (Section 4.2) also play a role in effecting loose-coupling.

Loosely coupled systems are therefore desirable for two primary reasons. First, such systems are highly modular and, hence, offer conceptual clarity and simplicity. Second, and equally important from our perspective, such systems require (by definition) a minimum of internode communication.[1]

---

[1] It is also of interest to note that current evidence [Galbraith, 1974] suggests that effective human organizations also operate in a loosely coupled manner, minimizing unnecessary communications among the members. It has also been noted [Brooks, 1975] that one of the purposes of organization is to reduce the amount of communication and that this can be done by division of labor and specialization of function. The problem is to perform effective task partitioning while minimizing the amount of communications overhead.

2.2.3 Loose-Coupling: Analysis

As a demonstration of the importance of loose-coupling in a distributed system, we examine briefly the volume of communications traffic that is required in such a system. We will consider only the communication that is concerned with the actual problem-solving effort and will ignore underlying control messages, such as those that must be periodically transmitted to determine the operational status of individual nodes. The analysis is very rudimentary but is adequate to indicate the importance of loose-coupling.[1]

We assume the existence of a distributed architecture with a single node type. The nodes are interconnected via a single broadcast communications channel that is accessed by all nodes in a demand-multiplexed fashion, although we will not deal explicitly with the effects of contention for the channel. We further assume that all nodes execute tasks that require the same time for execution and that the tasks give rise to the same amount of communication by each node; that is, the communications and processing load is uniformly distributed among the nodes. Finally, we assume that transmission and receipt of messages by a node can be carried out concurrently with task execution.

Under these assumptions, we can determine the required bandwidth for the broadcast channel. The following parameters are used in the discussion:

W_t: the average communications channel bandwidth required to support problem solving in the distributed architecture (bits/sec).

N: the total number of processor nodes.

S: the average speed of a processor node in the system (instructions/sec).

P: the average *processing-power* of a node in the system (standard-instructions/instruction).[2]

T: the time required by a node to execute an individual task (sec).

M: the average length of the messages that must be communicated by a node during the execution of a single task (bits).

K: the average *kernel-size* of a task that is executed by a node (standard-instructions). The kernel-size of a task is the average number of standard-instructions that must be executed to complete the task.

The time required to execute an individual task by a node is given by,

$$T = P^{-1} \cdot K \cdot S^{-1} . \qquad (1)$$

---

[1] Part of this analysis has been presented in [Knutsen, 1977].

[2] The *standard-instruction* is used as a device to allow different systems to be compared. It indicates the power of the instruction set of a reference processor, for example an 8080.

During this time M bits must be transmitted by each node. Hence, ignoring the effects of contention for the broadcast channel, the total broadcast channel bandwidth requirement is given by,

$$W_t = N \cdot M \cdot T^{-1}, \quad (2)$$

or

$$W_t = N \cdot M \cdot P \cdot K^{-1} \cdot S. \quad (3)$$

$M \cdot K^{-1}$ is a measure of the coupling between the nodes. It is strictly dependent on the nature of the tasks executed by the nodes but is independent of the particular architecture; that is, it is independent of the processing-power of the nodes and the bandwidth of the communications channels that interconnect the nodes.[1] We will refer to this ratio as the average *task-coupling-factor*, C_t:

$$C_t = M \cdot K^{-1}. \quad (4)$$

We can now restate the bandwidth requirement equation, *(3)*, in terms of the task-coupling-factor.

$$W_t = N \cdot P \cdot S \cdot C_t. \quad (5)$$

Consider now the bandwidth requirement for two classes of architecture: one with nodes that correspond to typical microprocessors and one with nodes that correspond to typical large-scale processors. For the microprocessor nodes we will assume $S = 5 \cdot 10^5$, and $P = 1$. For the large-scale processor nodes we will assume $S = 5 \cdot 10^6$, and $P = 20$.

The bandwidth requirement for the two architectures is shown under conditions of varying numbers of nodes and task-coupling-factors in Figure 2.1. Results for the microprocessor node system are shown in the top line in each box (designated by "m"), and results for the large-scale processor node system are shown in the bottom line (designated by "M"). We are interested in loosely coupled systems, and the range of task-coupling-factors in the figure has been chosen to reflect this emphasis.

It is clear even from this simple analysis that loose-coupling of processor nodes is critical if the overall bandwidth requirement for a single broadcast channel is to remain realizable. Loose-coupling is especially

These results give a rudimentary guide to selection of appropriate problems: The kernel-size of the tasks into which the problem is decomposed must be large enough (as defined by the characteristics of the architecture) so that the required internode message traffic does not saturate the available communications channels. A mismatch between problem decomposition and architecture may result in no gains in speed at all. important for architectures comprised of a large number of powerful (and fast) nodes.

---

[1] Several other coupling measures can be defined. One, for example, would relate the actual time spent by a processor node in communication with other nodes to the time spent in computation to execute a task. This measure would be dependent on the characteristics of the architecture. See [Garcia-Molina, 1978] for a discussion of other measures of coupling in distributed databases.

Task Coupling Factor		Number Of Nodes			
		10^1	10^2	10^3	10^4
10^{-1}	m	$5 \cdot 10^5$	$5 \cdot 10^6$	$5 \cdot 10^7$	$5 \cdot 10^8$
	M	10^8	10^9	10^{10}	10^{11}
10^{-2}	m	$5 \cdot 10^4$	$5 \cdot 10^5$	$5 \cdot 10^6$	$5 \cdot 10^7$
	M	10^7	10^8	10^9	10^{10}
10^{-3}	m	$5 \cdot 10^3$	$5 \cdot 10^4$	$5 \cdot 10^5$	$5 \cdot 10^6$
	M	10^6	10^7	10^8	10^9
10^{-4}	m	$5 \cdot 10^2$	$5 \cdot 10^3$	$5 \cdot 10^4$	$5 \cdot 10^5$
	M	10^5	10^6	10^7	10^8

Figure 2.1. Broadcast Channel Bandwidth Requirement (bits/sec).

The effect of kernel-size on a multiple processor architecture has been examined experimentally for C.mmp, a 16-node architecture with interconnection via a crossbar switch [Wulf, 1972], [Fuller, 1976]. It has been found that as kernel-size decreases (i.e., an attempt is made to have more concurrent processes), performance can be seriously degraded. It is therefore important to find the minimum kernel-size at which a particular architecture can exhibit parallelism.[1]

2.2.4 Architectural Model

The architecture is assumed to be a network of loosely coupled, asynchronous nodes. Each node has a local memory; no memory is shared by all nodes. There is no central controller for the architecture. The nodes are interconnected via one (or more) broadcast communications channels, and internode communication is effected by message-passing. The channels are demand-multiplexed.[2] The individual nodes consist of one or more LSI processors, local memory, and communications interface.

There are a variety of architectural questions and design choices. These include, for example, determination of the required memory size at a node and the minimum processor sophistication required for effective problem-solving performance. The appropriate number and bandwidth of communications channels for such an architecture must also be examined. We can address problem-solving questions independently, however, and will therefore not

[1] The effect of coupling on distributed tree search is demonstrated in Appendix A.

[2] If a set of channels is used, then one of them may be allocated for net-wide communications and the remaining channels dynamically allocated to subnets of nodes working on related tasks.

further address architectural issues here. They are sufficiently complex to warrant devoted study.

Chapter 3

Examples

In this chapter we present two implemented examples of distributed problem solving in the contract net framework. These examples have been chosen to show that the framework is applicable both to very simple problems and to problems of considerable complexity. The first example demonstrates an approach to distributed search. The second demonstrates an approach to a naturally distributed problem, that of area surveillance with a distributed sensing system.

Each of the examples demonstrates two main issues in distributed problem solving: how to partition a problem for a distributed approach and how to use the contract net framework to solve a partitioned problem. In Appendix B we consider a number of speculative examples. These brief examples demonstrate ways in which familiar *AI* problem solvers could be implemented within the contract net framework.

In Chapter 4 we will discuss the contract net framework in more detail and review the use of its features in the examples of this chapter.

3.1 Example 1: Search

Search is one of the major paradigms of problem solving in *AI*. Examples range from programs that play games, such as checkers **[Samuel, 1967]** or chess **[Berliner, 1973]**, to programs that attempt to uncover the theory underlying the fragmentation of molecules in a mass spectrometer **[Buchanan, 1978]**. Search algorithms for uniprocessors have been widely studied and several theorems have been developed.[1]

Search problems are attractive as applications of the distributed approach for three major reasons. First, the spaces involved in search problems are often quite large; that is, exploration of a search space of the size commonly encountered in *AI* applications consumes a large amount of computing time. Thus, the capability to do more search that is offered by the distributed approach makes it attractive. Second, search problems are often modular in form. Numerous relatively independent subtasks are created during the course of a search. These subtasks are ideal candidates for distribution to individual nodes. Finally, as noted above, search is one of the major paradigms of problem solving that is recognized in *AI* research. It is therefore important to develop tools for applying the new LSI processor technology to such problems.

---

[1] Note that the *problem-solving* search considered here differs from *list* search (see, for example **[Knuth, 1975]**) for two reasons: *(i)* the tree structure for problem-solving search exists only implicitly because the search spaces are generally so large that a priori construction is prohibitive, and *(ii)* in list search there is a well-defined algorithm for finding the goal node, whereas in problem-solving search there are generally several operators that are applicable to expand any node in the tree (and backtracking must usually be used).

3.1.1 Distributed Search: Overview

Consider the exhaustive search of a tree in a distributed processor architecture. We assume that the basic task for each processor is expansion of a node in the tree; that is, generation of its successors. As soon as a processor generates a successor node, it distributes the node to another processor for further expansion. We assume that expansion of any node requires the same processing time and that enough processors exist so that the expansion of a node can be commenced by one processor as soon as the node has been generated by another. This permits us to see the flow of the search process, but it obscures the normal problem of *selecting* the most appropriate nodes for expansion at any given instant in time. We will return to this problem later in the section. We also assume that both the distribution of a node to a processor and the reporting of results require a negligible amount of time compared to the time to expand the node.

The flow of the search process is shown in Figure 3.1 for a regular tree of branching factor *2* and depth *3*. The subfigures represent the state of the search at successive units of time. Nodes are shown as black circles at the time unit at which they are generated, and then shown as white circles with the time unit of generation inside for successive time units. Nodes that have not yet been generated are shown as empty white circles.

In the first subfigure (representing time *1*), the root node is shown to have been generated at time *0* and one successor node is shown in black. This successor is distributed (we assume instantaneously) to another processor, so that at time *2*, two successors are generated. The number of processors involved in the search peaks at *4* for time unit *3*, and finally decreases to *1* at time unit *5*. We can see that the search proceeds as an expanding wedge through the tree.[1]

Large speedups are associated with large trees; so, problems that entail a large amount of exhaustive search are the most amenable to speedup. In addition, trees comprised of *OR* nodes lend themselves more readily to concurrent exploration than do those comprised of *AND* nodes. This is because there is less processor node synchronization required for their exploration. Trees with *Ordered-AND* nodes[2] are the least amenable to concurrent exploration because they require the greatest amount of synchronization (which inevitably means that some nodes will stand idle waiting on results being generated by other nodes).

---

[1] In Appendix A we consider both the time and the number of processors required to search trees of various sizes, together with the effect of communications cost (which was ignored here).

[2] An *Ordered-AND* node is an *AND* node whose successors must be expanded in a particular order.

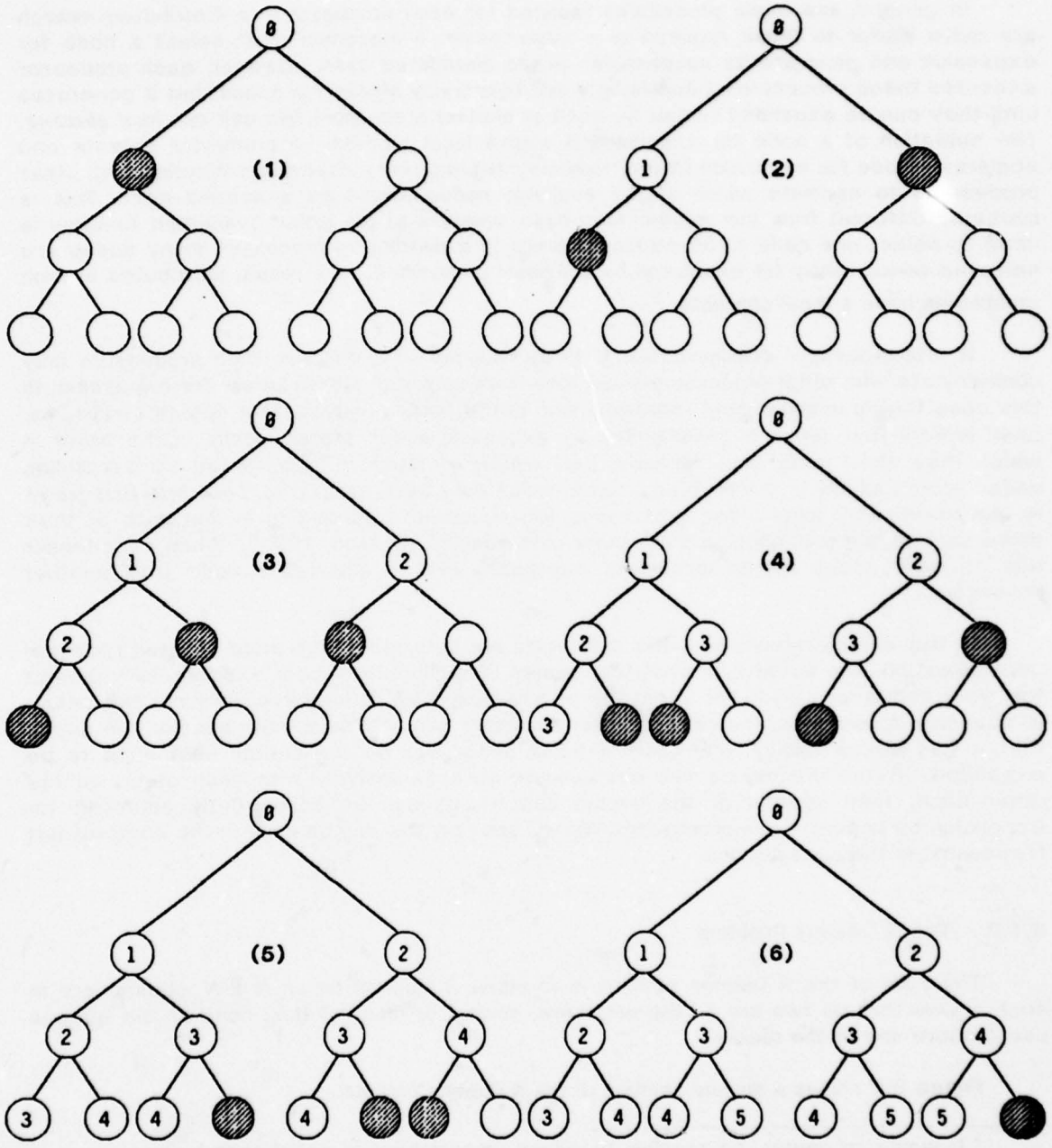


Figure 3.1. Distributed Search Of A Regular Tree. [Each subfigure depicts the progress of the search at one time unit (shown inside the subfigure). Newly generated nodes are black; nodes already generated are white, enclosing the time unit at which they were generated; and nodes not yet generated are white.]

In general, the basic procedures required for each processor in a distributed search are quite similar to those required in a uniprocessor. A processor must select a node for expansion and generate its successors. In the distributed case, however, each processor executes these procedures autonomously and temporarily stores the nodes that it generates until they can be expanded (either by itself or another processor). We call this *local queuing*. The selection of a node for expansion is also a local process. A processor selects and acquires a node for expansion (often from another processor) after communicating with other processors to estimate which of the available nodes should be expanded next. This is markedly different from the uniprocessor case, where a single global evaluation function is used to select one node to be expanded next. In a distributed processor, many nodes are selected concurrently for expansion by individual processors. As a result, distributed search strategies have a local character.[1]

If interprocessor communication is to be reduced to a minimum, then processors only communicate with other processors when they have no nodes stored locally for expansion. In this case they can make local approximations to the familiar uniprocessor search strategies. *Local breadth-first search* is implemented by expanding nodes stored locally in the order in which they were generated. Similarly, *local depth-first search* is implemented by expanding nodes stored locally in the reverse order in which they were generated. *Local best-first search* is implemented by local ordering of nodes for expansion according to an estimate of their distance from the goal node, or some other cost measure [Nilsson, 1971]. When a processor has no more nodes stored locally for expansion, then it acquires a node from another processor.

Better approximations to global strategies are obtained at the price of interprocessor communication. The intent of a best-first search in a uniprocessor, for example, is to select the most appropriate node for expansion at any given time. If interprocessor communication is severely constrained, then an individual processor can only select the best of the nodes that it has stored locally; and none of these nodes may be the overall best node to be expanded. If the processors can communicate more extensively with each other, on the other hand, then several of the overall best nodes can be concurrently selected for expansion by separate idle processors. We will see how this can be done in the contract net framework, in the next section.

3.1.2 The N Queens Problem

The goal of the N Queens problem is to place N queens on an $N \times N$ chessboard in such a way that no two are on the same row, column, or diagonal (i.e., none of the queens can capture any of the others).

Figure 3.2 shows a sample solution to the 4 Queens problem.

---

[1] It would, of course, be possible to impose a centralized or global search strategy on a distributed processor by requiring that all nodes ready for expansion be transmitted to a central repository. A central evaluation function would then order the nodes to implement the desired search strategy, and idle processors would remove nodes in order from the repository. This could result in bottleneck and reliability problems, however, as well as telling us little about distributed problem solving.

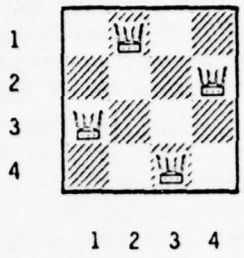


Figure 3.2. A Four Queens Problem Solution.

This problem serves as a simple introductory example of the application of the contract net framework. It is a vehicle for demonstration and not a problem that would obviously benefit from a distributed approach. It does, however, give rise to a few insights about features required in a framework for distributed problem solving.

3.1.3 Contract Net Implementation

We have implemented this example so that a distributed search is carried out for a pre-specified number of solutions. When any one node has compiled the required number of solutions, then the search is terminated. In addition, three different search strategies have been implemented: local-breadth-first, local-depth-first, and random ordering of nodes. The method of **[Floyd, 1967]** for extending a partial board was used in the implementation.[1]

The basic task for each processor node in this problem is the placement of one queen on an empty column of the chessboard in such a way that the no-capture constraint is not violated. Given a partial solution to the problem (i.e., a board with fewer than N queens placed--a *partial board*), the subtrees formed by all valid placements of the next queen can be explored independently.

The processor node at which the problem is started, called the *top-level* node, begins with an empty board. It generates N subtasks, each of which corresponds to a partial board with 1 queen in the first column and in a different row for each subtask. These subtasks are distributed to other nodes in the net through a process of contract negotiation. Successful bidders become contractors for the task of extending the partial boards to complete board descriptions. The top-level node becomes the manager for this task. (It is now free to become a contractor for future subtasks.)

This process is continued recursively for each column of the board; that is, the nodes (contractors) trying to extend partial boards (here, with 1 queen already placed) generate independent subtasks by placing a queen in the next column (here, the second column), under the no-capture constraint. They then distribute the subtasks (and take on the role of manager for them) and the process continues.

---

[1] A trace of the CNET computation for the 4 Queens problem is shown in Appendix C.

The task for any node is the completion of the partial board that forms the initial state description when the task is commenced. Each node attempts to place one queen on the next empty column of the board. Each time it is able to place a queen, it attempts to award the task of extending the resultant partial board to another node in the net for independent extension. There is thus only one type of task for all nodes--extension of a partial board. When a processor node places the N^{th} queen, and thus has a complete board description corresponding to a solution to the problem, it reports to its manager. Further reports ripple upward to the top level and the search terminates when some pre-specified number of solutions has been compiled at any one node; that is, when any manager has compiled the required number of solutions, it terminates any outstanding subtasks and reports to its own manager. This manager in turn terminates outstanding subtasks, and so on. Ultimately, the top-level node reports the solutions to the user.[1]

The general procedure for task distribution and coordination, is for task announcements to be made by nodes as they generate new partial boards. Bids are submitted to these nodes by other idle nodes. Successful bidders are awarded contracts. They then periodically report results to the nodes that awarded the contracts.

A task announcement contains three forms of information of interest here:[2] an ***eligibility specification***, or list of criteria that a node must meet to be eligible to submit a bid; a ***task abstraction***, or brief description of the task to be executed; and a ***bid specification***, which details the expected form of a bid for the task. Of primary interest here is the task abstraction. For this example it indicates the type of task to be executed (an *extend-board* task) and the present *state* of the task, relative to the goal state (in this case, the number of queens that have already been placed on the partial board). The present state indicator, number of queens, gives a potential contractor a method of ranking announced tasks at any given time in order to select a task for execution. It is used by nodes in this example to effect the desired (local) search strategy. A local breadth-first strategy, for example, is implemented by ranking boards that have a small number of queens placed, higher than those that have a larger number of queens placed. Bids are therefore submitted first for these boards, and they are therefore generally executed before the others. The effect is to approximate a breadth-first search, in which all nodes at one ply of the search tree are expanded before nodes at the next (or deeper) ply are expanded. In a similar fashion, a local depth-first strategy is effected by ranking announced boards with a large number of queens placed, higher than those with a smaller number of queens placed. Finally, a random strategy is implemented by replacing the indication of the state of the task by a random integer.[3]

Each node in a contract net maintains a list of tasks that have been recently announced (the details are discussed in Chapter 6). When a node goes idle, it selects,

---

[1] Note that this way of implementing the search is quite similar to a backtracking method in a single processor architecture.

[2] Samples of the message traffic are shown in the next section, and a complete description of the information transmitted in contract net messages is presented in Section 4.1.2.

[3] It is of course possible to execute different search strategies at different nodes. We assume here that all nodes execute the same strategy for simplicity.

according to its own criteria, the current optimum task for which to submit a bid from among the tasks contained in its list. Each node therefore has a kind of *window* through which to view the currently available tasks. This window lends a more global character to the search strategy at the expense of local storage and communication.[1]

One of two eligibility specifications are used. In the first case, a null specification is used. The assumption here is that all nodes have the necessary procedures for executing the *extend-board* task. In this case a bid simply indicates that a processor is willing to execute the announced task. In the second case, the eligibility specification can indicate that a bidder either *must have* the named procedures (see the sample messages) before submitting a bid, or it can simply name the required procedures. The assumption here is that not all processors are pre-loaded with the necessary procedures. A potential contractor can either acquire the necessary procedures via a ***request*** message or can indicate in its bid that it needs the procedures to execute the task (unless prohibited from bidding by the *must have* restriction). In this case the contract is awarded to the first node that has the procedures, or, in the absence of any such bidders, to the first node that bids. In general, then, contracts are awarded to the first bidder. No more complex strategy is required because any processor with the procedures has the capability to execute the task.

The award message is the message sent to the successful bidder. It includes the specification of the partial board that is to be extended.

The report is a message sent by a contractor to a manager that simply indicates that a partial board has (or has not) been successfully extended. It includes a specification of the completed board(s) or indicates that the partial board cannot be extended. The example has been implemented so that a manager receiving such a report can either forward it to its own manager, and so on, until the top-level node receives it; or wait until all of its subcontractors have reported, and then forward all successful reports; or wait until a specified number of successful reports have been received, forward them, and terminate the remaining outstanding subcontracts.

Figure 3.3 shows the search tree for the 4 Queens problem. Nodes in the tree correspond to partial board descriptions. The row indices of queens placed in successive columns of the board are listed from left to right beside the nodes to which they correspond. Nodes that cannot be extended through placement of queens that do not violate the no-capture constraint are marked with "F," and nodes that correspond to solutions to the problem are marked with "S."

[1] Local storage is required for the list of tasks. Communication is increased over that required if a node were simply to select tasks from among those it generated itself, seeking remotely generated tasks only after it had completed its own.

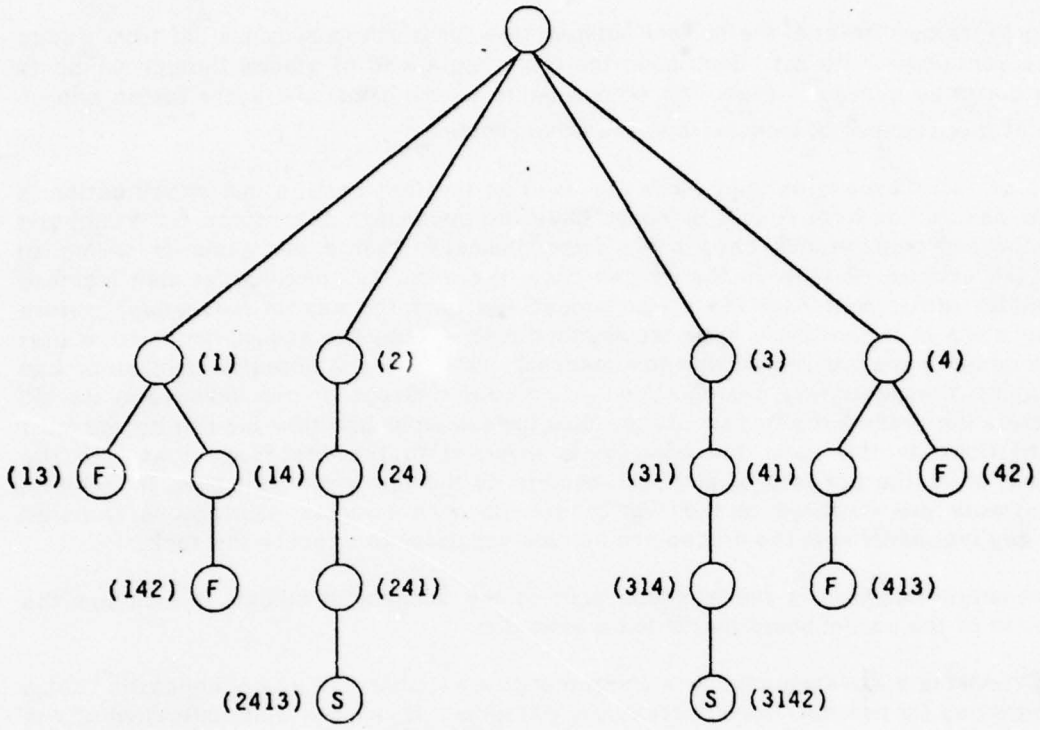


Figure 3.3. Four Queens Problem Search Tree.

3.1.3.1 Sample Messages

This section contains selected messages that are transmitted during execution of the N Queens problem. For brevity, the messages shown contain only the information mentioned in Section 4.1.2. Terms written in upper case are included in the core common internode language (discussed in Section 4.1.3), while terms written in lower case are specific to the N Queens problem. Italicized statements are commentary about the content and sequence of messages.

<The node given responsibility for the top-level task issues messages of the following form as it generates the first subtasks.>

To: *
<"*" *indicates a broadcast message.*>
From: node-1
Type: task announcement
Contract: 1
Message:

<Needed - extension of a partial board with 1 queen placed.>

Task Abstraction: TASK TYPE extend-board
board queens 1
Eligibility Specification: [MUST-HAVE] PROCEDURE NAME extend-board
Bid Specification: NIL

<If the MUST-HAVE term is present, then only nodes that are in possession of the named procedure are eligible to bid. If it is absent, then a node may make a bid which includes a request for the procedure as shown below. This demonstrates how an eligibility specification can be used to tightly constrain the number of nodes that can bid on a task or to loosen the constraints in order to foster distribution of expertise.>

<Idle nodes respond as follows.>

To: node-1
From: node-i
Type: bid
Contract: 1
Message:

<Will extend board.>

Node Abstraction: [REQUIRE PROCEDURE NAME extend-board] *<If required.>*

<The first bidding node is sent the following message.>

To: node-i
From: node-1
Type: award
Contract: 1
Message:

<Extend board with description (...).>

Task Specification: board specification (...)
[PROCEDURE NAME extend-board CODE (...)] *<If required.>*

<Eventually, messages of the following form are transmitted.>

To: node-k
From: node-x
Type: report
Contract: k
Message:

<Failed to extend board.>

Result Description: FAILURE

To: node-k
From: node-y
Type: report
Contract: k
Message:

<Succeeded in extension of board. Final description(s) (...).>

Result Description: SUCCESS
board specification (...) *<There could be several of these.>*

<If the required number of solutions has been accumulated by any node (ultimately node-1), then messages of the following form will flow throughout the net as they are received by managers that send further termination messages to their subcontractors.>

To: node-p
From: node-m
Type: termination
Contract: m
Message: NIL *<Terminate execution of contract.>*

3.1.4 Summary

We have shown the use of the contract net framework in the solution of a simple search problem. Only the basics of the framework are required for this problem and the contract negotiation process is particularly simple: so only a small amount of information needs to be transferred between nodes. Consequently, only a degree of the power of the approach is demonstrated here. The main use of the framework in this example is to demonstrate how links are set up between nodes for distribution of the processing load and communication of results. Nodes are used efficiently because they can take on multiple roles: A node that has generated all *1*-queen extensions to the current board and distributed them to other nodes (contractors) need only deal with reports occasionally (in its role as manager). It is therefore free to act as a contractor for other tasks. The result is that no nodes remain idle as long as there are tasks available to be executed. The explicit manager-contractor links assist in rapid local pruning of the search space (via termination messages) when a sufficient number of solutions has been found. Each manager can directly terminate the execution of subtasks being executed by its contractors as soon as it becomes aware that the results are no longer required. Thus, the net does not have to wait for reports to reach the top-level node before subtasks are terminated.

3.2 Example 2: Distributed Sensing

In this section we demonstrate the use of the contract net framework in the solution of a problem in area surveillance, such as is encountered in ship or air traffic control. The example helps to demonstrate the utility of the contract negotiation and knowledge organization aspects of the framework.

We consider the operation of a network of nodes, each having either sensing or processing capabilities and all spread throughout a relatively large geographic area. Such a network is called a ***Distributed Sensing System*** (DSS).

The primary aim of the system is rapid, reliable, accurate, and low-cost analysis of the traffic in a designated area. This analysis involves detection, classification, and tracking of vehicles; that is, the solution to the problem is a *dynamic map of traffic in the area.* Construction and maintenance of such a map requires the interpretation and integration of a large quantity of sensory information received by the collection of sensor elements.

There are many trade-offs involved in the design of a DSS architecture. We present only one possible approach that considers a limited number of these trade-offs.[1] The primary intent of this example is to demonstrate the contract net approach, and we therefore focus on the initialization and communications aspects of the DSS. For a discussion of other aspects of this problem see [Nii, 1978].

3.2.1 Hardware

All communication in the DSS is assumed to take place over a broadcast channel (using for example, packet radio techniques [Kahn, 1975]). The nodes are assumed to be in fixed positions known to themselves but not known a priori to other nodes in the net. Each node has one of two capabilities: sensing or processing. The sensing capability includes low-level signal analysis and feature extraction.[2] We assume that a variety of sensor types exist in the DSS, that the sensors are widely spaced, and that there is some overlap in sensor area coverage. Nodes with processing capability supply the computational power necessary to effect the high-level analysis and control in the net. They are not necessarily near the sensors whose data they process.

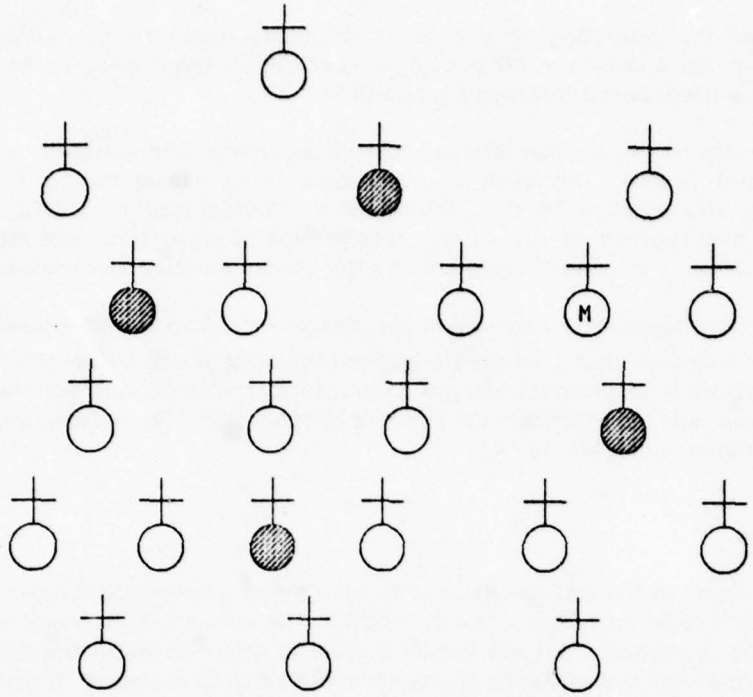
A DSS may have several functions, ranging from analysis of vehicle data in the overall area of coverage to control over the courses and speeds of those vehicles. We consider here the analysis function. In this case the overall area map must be integrated at one node in the system (it could also be integrated by an agent outside the system, like a monitoring aircraft). We therefore distinguish one processor node as the monitor node. Its function is to begin the initialization of the DSS and to integrate the overall area map for communication to an agent outside the DSS. We will see that it does not correspond to a central controller.

Figure 3.4 is a schematic representation of a DSS. Processing nodes are shown in black, and sensing nodes in white. The monitor node is shown in white with an "M" in the middle.

---

[1] Further discussion of the background issues inherent in DSS design is presented in [Smith, 1978a].

[2] In a real DSS, it is likely that sensors and low-level signal analysis devices would not be considered as statically connected parts, as it is often the case that many different types of analysis are applied to the output of a single sensor, or to that of groups of sensors taken together.



Legend:

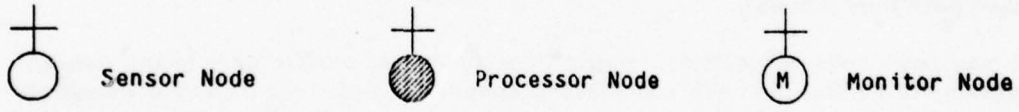


Figure 3.4. A Distributed Sensing System.

3.2.2 Data And Task Hierarchy

The DSS must integrate a large quantity of signal data, reducing it and transforming it into a symbolic form meaningful to and useful to a human decision maker. We view this process as occurring in several stages, which together form a data hierarchy (Figure 3.5). The hierarchy offers an overview of DSS functions and suggests a task partitioning suitable for a contract net approach. A particular node in the DSS handles data at only one level of the data hierarchy at any given moment and communicates with nodes at other levels of the hierarchy.

overall area map

area map

vehicle

signal group

signal

Figure 3.5. Data Hierarchy.

For purposes of this example, the only form of signal processing we consider is narrow band spectral analysis. The *signal* has the following features: frequency, time of detection, strength, characteristics (e.g., increasing signal strength), name and position of the detecting node, and the name, type, and orientation of the detecting sensor.[1]

Signals are formed into *signal groups* at the second level of the data hierarchy. A signal group is a collection of related signals.[2] For this example, the signal groups have the following features: the fundamental frequency of the group, the time of group formation, and the features of the signals in the group (as above).

The next level of the hierarchy is the description of the *vehicle*. It has one or more signal groups associated with it and is further specified by position, speed, course, and type.[3] Position can be established by triangulation, using matching groups detected by several sensors with different positions and orientations. Speed and course must be established over time by tracking.

The *area map* forms the next level of the data hierarchy. This map incorporates information about the vehicle traffic in an area. It is an integration of the vehicle level data. There will be several such maps for the DSS, corresponding to areas in the span of coverage of the net.

---

[1] The frequency spectrum of noise radiated by a vehicle typically contains narrow band signal components that are caused by rotating machinery associated with the vehicle (e.g., engines or generators). The frequencies of such signals are correlated with the type of rotating machine and its speed of rotation. They are indicators of the classification of the vehicle. Narrowband signals also undergo shifts in frequency, due to Doppler effect, or instability and change in the speed of rotation of the associated machine. Alterations in signal strength also occur as a result of propagation conditions and variations in the distance between the vehicle and the sensor.

[2] A signal group that is often used to integrate narrow band signal data, for example, is the harmonic set, a group of signals that are harmonically related (i.e., the frequency of each signal in the group is an integral multiple of the lowest, or fundamental frequency). A single rotating machine often gives rise to several narrow band signals that form a harmonic set.

[3] For simplicity, we have ignored a level in the hierarchy that can be called *component sources of signal*, as in [Nii, 1978]. At this level, a DSS would normally try to attribute signals and signal groups to particular pieces of machinery associated with a vehicle.

The final level is the complete or *overall area map*. In this example, the map is integrated from the individual area maps by the monitor node.

As indicated above, the hierarchy of tasks follows directly from the data hierarchy. The monitor node manages several *area* contractors. These contractors are responsible for the formation of traffic maps in their immediate areas. Each area contractor, in turn, manages several *group* contractors that provide it with signal groups for its area (Figure 3.6). Each group contractor integrates raw signal data from *signal* contractors that have sensing capabilities.

The area contractors also manage several *vehicle* contractors that are responsible for integration of information associated with individual vehicles. Each of these contractors manages: a *classification* contractor that determines vehicle type; a *localization* contractor, that determines vehicle position; and a *tracking* contractor, that tracks the vehicle as it passes through the area.[1]

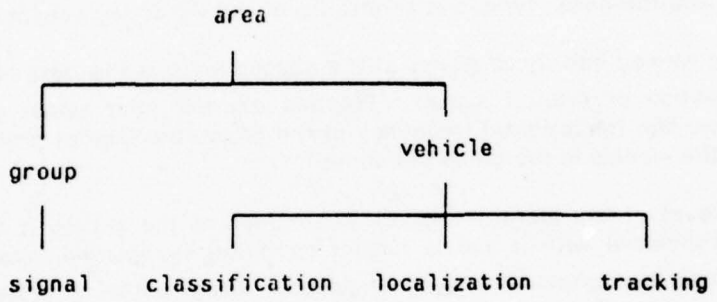


Figure 3.6. Area Task Partitioning.

Note that this particular partitioning of tasks is only one of many possibilities that might be specified by the system designer.

3.2.3 Contract Net Implementation

This section reviews in qualitative terms how the DSS problem can be attacked using the contract net approach and illustrates several of the ideas central to its operation. Section 3.2.3.4 gives specific examples of the message traffic that is described here, and the reader may find it helpful to refer to those messages during the following discussion.

---

[1] In a real solution to the DSS problem, it is possible that not all of these tasks would be large enough to justify the overhead of contracting; that is, some of them might be done in a single node. Note also that some of the tasks in the hierarchy are *continuing* tasks (e.g., the area task), while others are *one-time* tasks (e.g., the localization task).

3.2.3.1 Initialization

The monitor node is responsible for initialization of the DSS and for formation of the overall map. It must first select nodes to be area contractors and partition the system's span of coverage into areas, based on the positions of those selected nodes. For purposes of illustration we assume that the monitor node knows the names of nodes that are potential area contractors, but it must establish their positions in order to partition the overall span of coverage. Hence, it begins by announcing contracts for formation of area maps of the traffic. Because the monitor node knows the names of potential area contractors, it can avoid a general broadcast and can instead use a focused addressing scheme. The announcement contains the three components described in Section 4.1.2: a task abstraction, an eligibility specification, and a bid specification. The task abstraction is simply the task type, and the eligibility specification is blank (because the monitor node knows which other nodes are potential contractors and addresses them directly). The bid specification is of primary interest for this task. It informs a prospective area contractor to respond with its position. Remember that the purpose of a bid specification is to enable a manager to select, from all of the bidders, the most appropriate nodes to execute a contract. Node position is the information required by the monitor node to make that selection. Given that information, the monitor node can partition the overall span of coverage into approximately equal-sized areas and can select a subset of the bidders to be area contractors. Having decided upon a partitioning, the monitor node broadcasts an ***information message*** to the other nodes in the system. This message defines the names and specifications (in terms of latitude and longitude ranges) of the individual areas. Each selected area contractor is then informed of its area of responsibility in an award message.[1]

The area contractors' purpose is to integrate vehicle data into area maps. They must first establish the existence of vehicles on the basis of signal group data. To do this, each area contractor solicits other nodes to provide signal group data. In the absence of any information about which nodes are suitable, each area contractor announces the task using a general broadcast. The task abstraction in these announcements is the type of task. The eligibility specification is the area for which the individual area contractor is responsible; that is, a node is only eligible to bid on this task if it is in the same area as the announcing area contractor. This restriction helps to prevent a case in which a signal group contractor is so far away from its manager that reliable communication is difficult to achieve. The bid specification is again node position. Potential group contractors respond with their respective positions, and, based on this information, the area contractors award signal group contracts to nodes in their areas of responsibility.

The signal group contractors' task is to integrate signal data from sensor nodes into signal groups. Therefore, they must first find nodes that will provide raw signal data. This is done with signal task announcements. The eligibility specification in these announcements indicates that only those nodes located in the same area as the announcer and having

---

[1] The full announcement-bid-award sequence is necessary (rather than a directed contract) because the monitor node needs to know the positions of all of the potential area contractors in order to partition the overall span of coverage of the DSS into manageable areas. Note that this means that the DSS can adjust to a change in the number or position of potential area contractors.

sensing capabilities should bid on this task. The task abstraction indicates the position of an individual signal group contractor. This information assists potential signal contractors in determining the group contractors to which they should respond.[1]

The potential signal contractors listen to the task announcements from the various group contractors. They respond to the nearest group contractor with a bid that supplies their position and a description of their sensors. The group contractors use this information to select a set of bidders that covers their immediate vicinity with a suitable variety of sensors, and then award signal contracts on this basis. The awards specify the sensors that each signal contractor is to use to provide raw data to its managing group contractor. Figure 3.7 depicts the exchange between one group contractor (the black node), and several potential signal contractors (the white nodes). Successful bidders are connected by solid lines to the group contractor and unsuccessful bidders are connected by dashed lines.

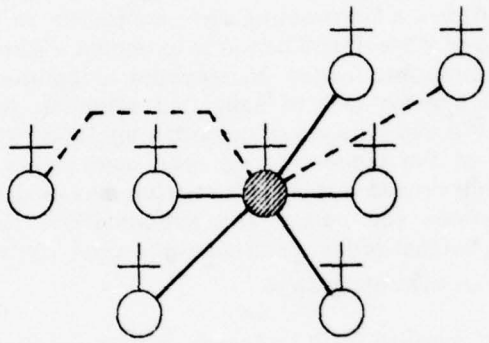


Figure 3.7. Signal Contract Negotiation.

There are some potential problems of asynchrony in the receipt of announcements from group contractors. A potential contractor for signal tasks must determine the group contractor that is closest to it by listening to several signal task announcements. The potential signal contractors use the ***expiration times*** that are included in task announcements (i.e., the times after which no further bids will be accepted) as a guide to the length of time during which they can listen to announcements before submitting a bid. In the best case, the expiration times are long enough to allow announcements from group contractors to reach all local potential signal contractors so that optimum partitioning can be achieved. In the worst case, however, a potential signal contractor may submit a bid to a group contractor that is not the closest one to it (because the task announcement of the closest group contractor is not received until after a bid has already been submitted to another group contractor). The result is a sub-optimal partitioning.

[1] The signal task is to detect signals and report them to a signal group contractor in a particular position. Hence the position of the group contractor is reasonable as part of the abstraction for the task.

The signal contract is a good example of the contract negotiation process, illustrating how the matching of contractors to managers is an interactive process. It involves a mutual decision based on local processing by both the group contractors and the potential signal contractors. The potential signal contractors base their decision on a distance metric and respond to the closest manager. The group contractors use the number of sensors and distribution of sensor types observed in the bids to select a set of signal contractors that ensures that every area is covered by every kind of sensor. Thus each party to the contract evaluates the proposals made by the other, using a different evaluation function, and a task distribution agreement is completed via mutual selection.

Reviewing the status of the DSS, we have a single monitor node that manages several area contractors. Each area contractor manages several group contractors, and each group contractor manages several signal contractors. The data initially flows from the bottom to the top of this hierarchy. The signal contractors supply raw signal data; each group contractor integrates the raw data from several signal contractors to form a signal group, and these groups are passed along to the area contractors, which eventually form area maps by integrating information based on the data from several group contractors. All the area maps are then passed to the monitor which forms the final traffic map.

As we have noted, in this example one area contractor manages several group contractors and each group contractor in turn manages several signal contractors. It is possible, however, that a single group contractor could supply information to several area contractors, and a single signal contractor could supply information to several group contractors. It may be useful, for instance, to have a particular group contractor near an area boundary report to the area contractors on both sides of the boundary. This is easily accommodated within our framework.

3.2.3.2 Comments On The DSS Organization

We have taken a top-down, distributed approach to initializing the DSS. An alternative approach might involve acquisition of the positions of all nodes at a very early stage, followed by area definition, award of area contracts, and so on. This would involve a more global approach to the problem of initialization, using a single node that initialized the net by gathering together all the necessary data. We have not pursued this approach for several reasons, primarily because it would tell us little about solving problems in a distributed manner. An underlying theme of this research is a search for ways in which to effect distributed problem solving--rather than ways to do traditional problem solving in a distributed architecture.

Moreover, there are two practical difficulties with the global approach. First, it concentrates a large amount of message traffic and processing at a single node (say the monitor node) because such a node would be responsible for accepting position messages from every other node in the net. Second, it may not be possible for any single node to communicate directly with all other nodes in a widely separated collection. This would mean that either indirect routing of messages would be required for communication or that each of the nodes would require powerful transmitters. This is one of the advantages of the distributed and dynamically defined organization we have adopted: Only pairs of nodes that are close together enough to communicate directly are linked together with contracts.

3.2.3.3 Operation

We now consider the activities of the system as it commences operation.

When a signal is detected or when a change occurs in the features of a known signal, the detecting signal contractor reports this fact to its manager (a group contractor) (Figure 3.8). This node, in turn, attempts to integrate the information into an existing signal group or to form a new signal group.

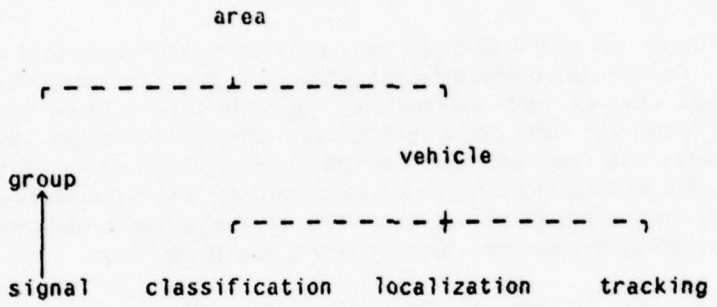


Figure 3.8. Signal Contract Reporting.

A group contractor reports the existence of a new signal group to its manager (an area contractor) (Figure 3.9).

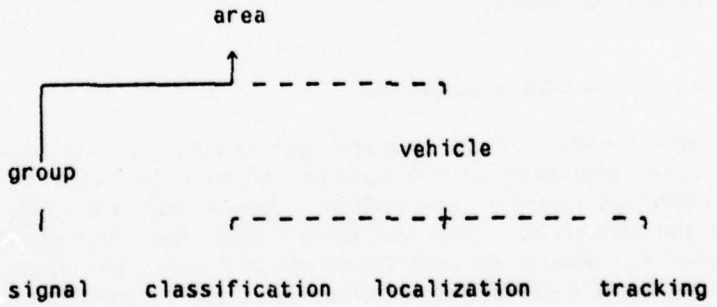


Figure 3.9. Group Contract Reporting.

Whenever a new group is detected, the managing area contractor attempts to find a node to execute a vehicle contract (Figure 3.10). The task of a vehicle contractor is to classify, localize, and track the vehicle associated with the signal group. Since a newly detected signal group may be attributable to a known vehicle, the area contractor first requests from the existing collection of vehicle contractors a measure of confidence in the fact that the new group is attributable to one of the known vehicles. Based on these

responses, the area contractor either starts up a new vehicle contractor or augments the existing contract of the appropriate existing vehicle contractor, with the task of making certain that the new group corresponds to a known vehicle. This may entail, for example, the gathering of new data via the adjustment of sensors or contracts to new sensor nodes.

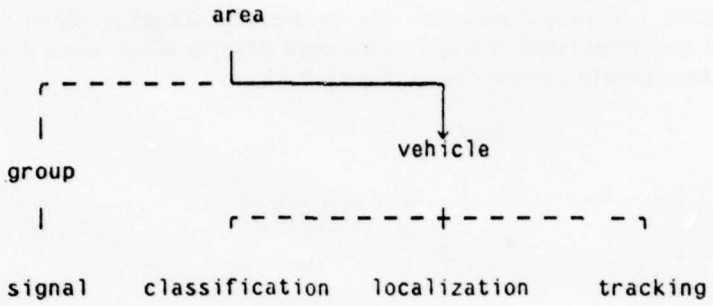


Figure 3.10. Vehicle Contract Initiation.

The form of the signal group confirmation request demonstrates a trade-off that arises in many distributed problem-solving applications--a trade-off between communication and local processing. The area contractor has the option of transmitting to the existing vehicle contractors either the complete signal group or an abstraction of it (e.g., its fundamental frequency). In either case, the response to the request is a measure of the individual vehicle contractor's confidence that the group corresponds to a vehicle it knows about. If a vehicle contractor returns a high confidence measure in its response, then in the first case (complete signal group announced) the area contractor has the information it wants. In the second case, however, the area contractor must still transmit the complete signal group description and await a further report.

The first approach has the disadvantage of using up more local processing time in each of the vehicle contractors than the second: in the former, the original request message is more complex. The question of interest is, *Under what conditions should a complete signal group description be announced instead of an abstraction?* The answer appears to depend on the quality of the abstraction. If the abstraction is good enough that the vehicle contractors are able to make definitive statements on the basis of its use, then the second approach is likely better than the first, in that it minimizes local processing time. On the other hand, if the abstraction does not allow definitive statements to be made, then its use will result in increased message traffic and local processing time, since the area contractor will not be certain as to the best course of action as a result of uncertain responses from the vehicle contractors.

The vehicle contractor then makes two task announcements: vehicle classification and vehicle localization. The task abstraction of the classification task announcement is a list of the fundamental frequencies of the signal groups currently associated with the vehicle. This information may help a potential classification contractor select an appropriate task (a contractor may, for example, already be familiar with vehicles that have signal groups with the announced fundamental frequencies). The eligibility specification is blank. The bid

specification indicates that a bidder should respond with a tentative classification and an associated confidence measure. This measure is used to select a classification contractor--the bidder with the highest confidence is chosen. The award is the complete current description of the vehicle.

A classification contractor may be able to classify directly, given the signal group information; or, on the other hand, it may require more data, in which case it can communicate directly with the appropriate sensor nodes (Figure 3.11).

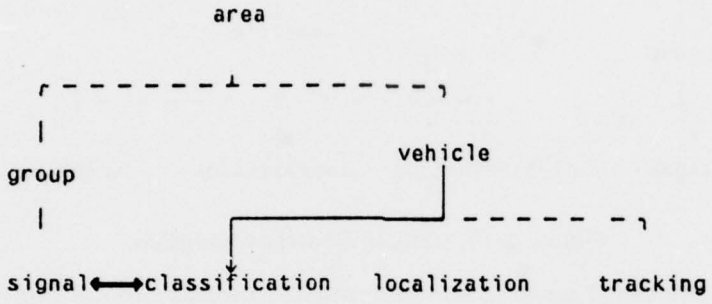


Figure 3.11. Classification Contract Communication.

The task abstraction for a localization task announcement (Figure 3.12) is a list of positions of the nodes that have detected a vehicle. The eligibility specification and bid specification are blank. The bid is simply an affirmative response to the announcement and the contract is awarded to the first bidder, which does the required triangulation to obtain the position of the vehicle.

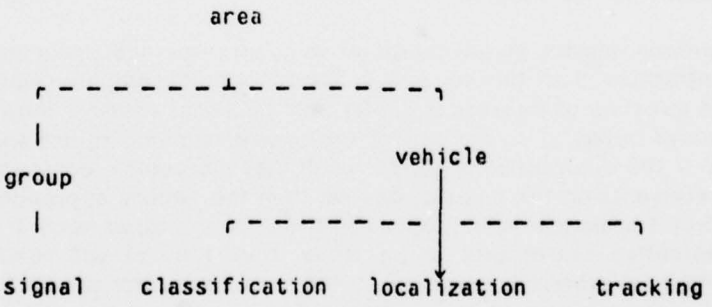


Figure 3.12. Localization Contract Initiation.

Once the vehicle has been localized, it must be tracked. This is handled by the vehicle contractor by entering into follow-up localization contracts from time to time and using the results to update its vehicle description (Figure 3.13). As an alternative, the area contractor could award separate tracking contracts. The decision as to which method to use depends

on loading and communications. If, for example, the area contractor is very busy with integration of data from many group contractors, then it seems more appropriate to isolate it from the additional load of tracking contracts. If, on the other hand, the area contractor is not overly busy, then we can let it handle updated vehicle contracts, taking advantage of the fact that it is in the best position to integrate the results and coordinate the efforts of multiple tracking contractors. In this example, we assume that the management load would be too large for the area contractor.

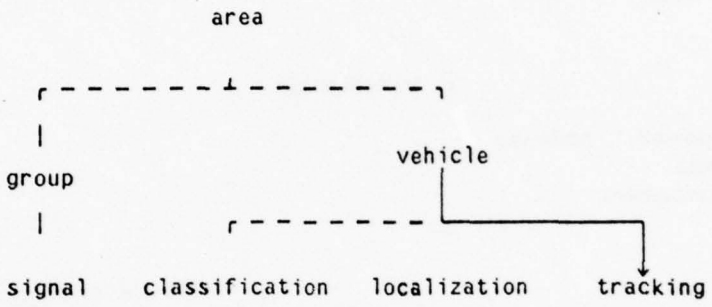


Figure 3.13. Localization Contract Initiation.

There are a variety of other issues that must be considered in the design and operation of a real distributed sensing system. Most of these are quite specific to the DSS application, and would take us away from our primary concern with the use of the contract net framework. One issue, however, presents an interesting example of the utility of the use of contracting relative to the use of information messages.

Consider the case of a vehicle that is moving from the area of one area contractor to that of another. How is responsibility for tracking the vehicle to be transferred? There are two possibilities. First, the area contractor that is currently responsible for the vehicle could send out information messages to neighboring area contractors. These messages would serve to set up expectations in the neighboring area contractors that a vehicle was about to pass into one of their areas. The processing is fully decentralized in this approach.

Second, the area contractor currently responsible for the vehicle could send a report to its manager (the monitor node) that contains the same information. The monitor node could then award a new contract to one of the neighboring area contractors to handle the vehicle when it passes out of the original area. This is a hierarchical control approach to the problem. It entails more (centralized) processing by the monitor node.

With the first approach there is no guarantee that another area contractor will pick up on the information message and immediately take responsibility for the vehicle, where the second approach does provide such a guarantee. The lack of guarantee is not generally serious but may result in excess processing because the vehicle must be redetected in the new area, reclassified, relocalized, and so on. The trade-off is thus more processing by the monitor node against (possibly) more processing by nodes in the new area. In the simulation,

we have used the hierarchical control approach because of its guarantee of transfer of responsibility.

3.2.3.4 Sample Messages

This section contains selected messages that are transmitted during initialization and operation of the DSS.

Initialization

To: node-m1, node-m2,..., node-mp
From: monitor node
Type: task announcement
Contract: m
Message:

<Needed: maps of the traffic in areas. Respond with position.>

Task Abstraction: TASK TYPE area
Eligibility Specification: NIL
Bid Specification: NODE NAME SELF POSITION

<Bids such as the following are received by the monitor node, which then uses them to partition the overall span of coverage of the DSS into suitable areas.>

To: monitor node
From: node-m1
Type: bid
Contract: m
Message:
Node Abstraction: NODE NAME m1 POSITION p

<The monitor node broadcasts the area definitions.>

To: *
From: monitor node
Type: information
Contract: m
Message:
Information Specification: area NAME A specification (...)
area NAME B specification (...) *<For each of the areas.>*

<Several similar awards are made for each of the areas defined by the monitor node.>

To: node-m1
From: monitor node
Type: award
Contract: m
Message: *<Map traffic in area A.>*
Task Specification: **area NAME A**

<Announcements of the following form are made by the various area contractors.>

To: *
From: node-m1
Type: task announcement
Contract: g
Message:
<Needed: signal groups for traffic. Must be in area A. Respond with position.>
Task Abstraction: TASK TYPE group
Eligibility Specification: NODE NAME SELF POSITION **area A**
Bid Specification: NODE NAME SELF POSITION

To: node-m1
From: node-sg1
Type: bid
Contract: g
Message:
Node Abstraction: NODE NAME sg1 POSITION p1

<Several bids are submitted to the area contractors.>

To: node-sg1
From: node-m1
Type: award
Contract: g
Message:
<Report signal groups.>
Task Specification: NIL

<Several similar awards are made to nodes in the different areas.>

<Announcements of the following form are transmitted by the various group contractors.>

To: *
From: node-sg1
Type: task announcement
Contract: s
Message:

<Needed: signal data for traffic. My position is p. If in possession of sensors, and located in area A, respond with position, and type and number of sensors.>

Task Abstraction: TASK TYPE signal
NODE NAME sg1 POSITION p
Eligibility Specification: MUST-HAVE DEVICE TYPE sensor
MUST-HAVE NODE NAME SELF POSITION area A
Bid Specification: NODE NAME SELF POSITION
EVERY DEVICE TYPE sensor TYPE NUMBER

<Nodes with sensors respond to the nearest group contractor.>

To: node-sg1
From: node-s1
Type: bid
Contract: s
Message:
Node Abstraction: NODE NAME s1 POSITION q
sensor TYPE S NUMBER 3
sensor TYPE T NUMBER 1

<Several similar awards are transmitted.>

To: node-s1
From: node-sg1
Type: award
Contract: s
Message:

<Report signals. Use sensors S1 and S2.>

Task Specification: sensor NAME S1
sensor NAME S2

<The DSS has now been initialized and commences operation.>

Operation

<As signals are detected, messages of the following form are transmitted.>

To: node-sg1
From: node-s1
Type: report
Contract: s
Message:

<Detected signal: frequency f0, time-of-detection t0, strength s0, characteristics (...), detecting-node s1, position q, sensor S1, type S, orientation a.>

Result Description: signal NAME S1
frequency f0
time-of-detection t0
strength s0
characteristics (...)
NODE NAME s1
POSITION q
sensor NAME S1
TYPE S
orientation a

<After several such messages have been received, signal groups are formed.>

To: node-m1
From: node-sg1
Type: report
Contract: g
Message:

<Formed signal group: fundamental fh0, time-of-formation t1, signals (...).>

Result Description: group NAME SG1
fundamental fh0
time-of-formation t1
signals (...)

<In order to avoid starting up a new vehicle contractor if it is not necessary, the area contractor first checks to see if the new signal group corresponds to a known vehicle.>

To: node-v1, node-v2, ..., node-vn
From: node-m1
Type: request
Contract: v
Message:

<Signal group - fundamental fh0. Respond with confidence that it corresponds to a known vehicle.>

Request Specification: group fundamental fh0
RESPOND vehicle NAME confidence

<Vehicle contractors respond with messages of the following form.>

To: node-m1
From: node-vi
Type: information
Contract: v
Message:
Information Specification: vehicle NAME V1 confidence 0.9

<We see that the ***information message*** *is used both as a general data transfer message and as a response to a request.>*

<If such a positive message is received, then the area contractor sends the following message.>

To: node-vi
From: node-m1
Type: request
Contract: v
Message:

<Confirm correspondence of vehicle V1 with signal group SG1 that has characteristics (...).>

Request Specification: vehicle NAME V1
group NAME SG1 ... *<detailed specification of the signal group>*

<If no such messages are received, then the area contractor transmits a message of the following form.>

To: *
From: node-m1
Type: task announcement
Contract: v
Message:

<Needed: node to process vehicle associated with a signal group that has fundamental fh0. The task requires the process-vehicle procedure (and nodes must already have this procedure to be eligible to submit a bid).>

Task Abstraction: TASK TYPE vehicle
vehicle group fundamental fh0
Eligibility Specification: [MUST-HAVE] PROCEDURE NAME process-vehicle
Bid Specification: NIL

To: node-m1
From: node-v1
Type: bid
Contract: v
Message:
Node Abstraction: [REQUIRE PROCEDURE NAME process-vehicle]. *<If required.>*

<Several such bids may be received.>

To: node-v1
From: node-m1
Type: award
Contract: v
Message:

<Process vehicle designated V1.
Signal group (...) ... >

Task Specification: vehicle NAME V1
group NAME SG1 *<There could be several of these.>*
fundamental fh0
time-of-formation t0
signals (...)
[PROCEDURE NAME process-vehicle CODE (...)]. *<If required.>*

<The vehicle contractor now tries to find nodes to classify and localize the vehicle.>

To: *
From: node-v1
Type: task announcement
Contract: c
Message:

<Needed: classification of vehicle with signal group that has fundamental fh0. Respond with a tentative classification and confidence measure.>

Task Abstraction: TASK TYPE classification
group fundamental fh0
Eligibility Specification: NIL
Bid Specification: vehicle classification confidence

To: node-v1
From: node-c1
Type: bid
Contract: c
Message:
Node Abstraction: vehicle classification ore-carrier confidence 0.7

<Several such bids may be received.>

To: node-c1
From: node-v1
Type: award
Contract: c
Message:

<Classify vehicle designated V1 with signal groups (...). Forward reports to node-m1.>

Task Specification: vehicle NAME V1 group (...) group (...) ...
REPORT-RECIPIENT NODE NAME m1

<Note that direct communication is set up between node-c1 and node-m1.>

<Eventually,>

To: node-v1, node-m1
From: node-c1
Type: final report
Contract: c
Message:
Result Description: vehicle NAME V1 classification ore-carrier

---

<Once the vehicle has been detected at several sensor node positions, the vehicle contractor tries to localize the vehicle, as follows.>

To: *
From: node-v1
Type: task announcement
Contract: p
Message:

<Needed: position for vehicle detected at sensor positions p2, p3, and p4.>

Task Abstraction: TASK TYPE localization
vehicle detected-position p2
vehicle detected-position p3
vehicle detected-position p4
Eligibility Specification: NIL
Bid Specification: NIL

To: node-v1
From: node-l1
Type: bid
Contract: p
Message:
Node Abstraction: NIL *<Will localize vehicle.>*

To: node-l1
From: node-v1
Type: award
Contract: p
Message:

<Localize vehicle V1. Detecting sensor information (...). Forward reports to node-m1.>

Task Specification: vehicle NAME V1 group (...) group (...) ...
REPORT-RECIPIENT NODE NAME m1

To: node-v1, node-m1
From: node-l1
Type: report
Contract: p
Message:
Result Description: vehicle NAME V1 POSITION p6

<Periodically, a vehicle contractor sends messages of the following form.>

To: node-m1
From: node-vi
Type: report
Contract: v
Message:
Result Description: vehicle NAME Vj
POSITION pj
speed vsj
course vcj
classification ore-carrier

<Periodically, an area contractor sends messages of the following form.>

To: monitor node
From: node-mi
Type: report
Contract: m
Message:
Result Description: area-map vehicle (...) vehicle (...) ...

<And so on, ...>

3.2.4 Summary

The use of the contract net framework in the DSS enables the implementation of a dynamic configuration, depending on the actual positions of sensor and processor nodes and the ease with which communication can be established. Such a configuration offers a significant operational improvement over a static a priori configuration; specifically, it ensures that nodes that must cooperate for the solution of the area surveillance problem are able to communicate with each other. This avoids the necessity for either indirect routing of messages or powerful transmitters for all nodes.

As in the first example, the distributed control enabled by the framework enhances both reliability and equalization of the processing load. It is also possible to recover from the failure of nodes, because there are explicit links between nodes that share responsibility for execution of tasks (managers and contractors). The failure of a signal contractor, for example, can be detected by its manager, and the contract for which it was responsible can be re-announced and awarded to another node.

The framework is also advantageous for this problem because it enables addition of new nodes to the net, even after operation has commenced. This is possible for two reasons.

First, the nodes communicate in a common protocol. This protocol enables a new node to interpret the task-independent portions of messages (e.g., that a particular message specifies a task to be executed). Second, the protocol is augmented by a common internode language. This enables a new node to identify, for example, the specific information that it must have to execute a particular task. The protocol and language also enable a new node to request this information from other nodes in the net.

The contract negotiation process is considerably more complex for the DSS example than for the N Queens example. We have shown the effectiveness of interactive mutual decisions (by those with tasks to be executed and those in a position to execute the tasks) in distributing tasks throughout the system, as well as in preventing excessive internode communication. This feature was used, for example, in setting up signal contracts. The potential signal contractors used the information in the signal task announcements to select the closest managers to which to respond. The managers, in turn, used the number of sensors and the distribution of sensors in the returned bids to make a final selection of signal contractors. This point will be revisited and explored in detail in Section 5.2, where we compare the contract net framework with other systems for AI problem solving.

Chapter 4

A Framework For Distributed Problem Solving

In this chapter we present an overview of the three components of our framework for distributed problem solving: communications, control, and knowledge organization. A detailed discussion of the systems issues and implementation of the framework is presented in Chapter 6. Chapter 5 compares the contract net approach to those taken in earlier *AI* problem-solving systems.

4.1 Communications And Control

As we noted in Chapter 1, communications and control are closely linked and will therefore be discussed together. To review, the central parts of these two components of the framework are shown in Figure 4.1.

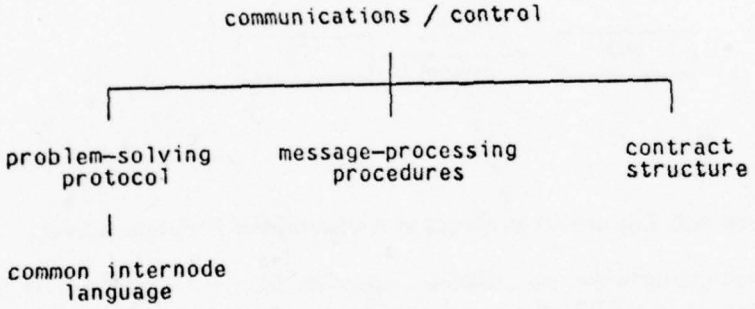


Figure 4.1. The Communications And Control Components Of The Framework.

The *problem-solving protocol* is a set of messages that enables nodes to communicate and interact to solve problems. A *common internode language* is used to encode the necessary task-dependent information in slots of the messages. The *message-processing procedures* control the actions of a node in response to messages of the protocol. They are the main interface with the user-procedures that make up a program. The *contract structure* handles the bookkeeping required for task distribution and node interaction.

4.1.1 Problem-Solving Protocols

The use of *communications protocols* in networks of resource-sharing computers, such as the ARPAnet, is by now quite familiar. These protocols have as their primary function reliable and efficient communication between computers [Green, 1975]. The layers of protocol in the

PRECEDING PAGE BLANK-NOT FILMED

ARPAnet, for example, serve to connect IMPs to IMPs (the subnet communications devices), hosts to hosts (the nodes of the network), and processes executing in the various hosts to other such processes (Figure 4.2). The use of messages in a distributed architecture is also advantageous **[Wecker, 1974]** because it reduces the problem of shared-memory access and the corresponding problem of locking. It also buffers individual nodes against the actions of faulty nodes.

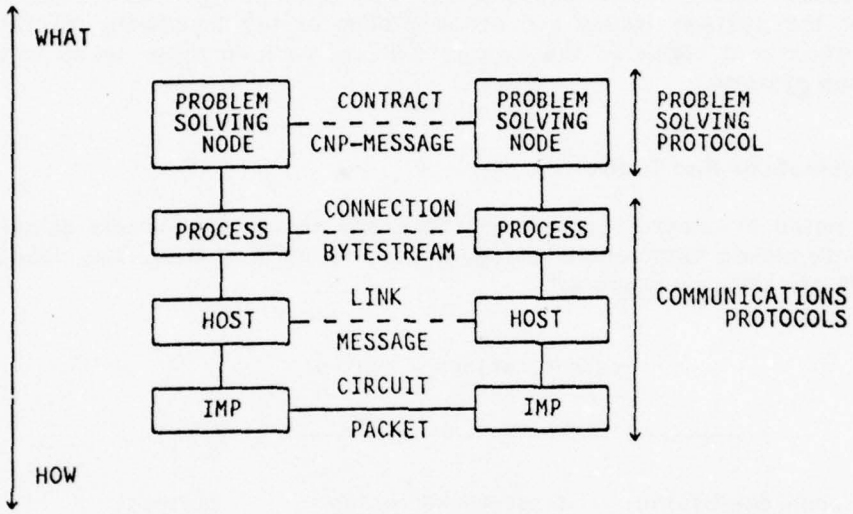


Figure 4.2. Layers Of Protocol In A Distributed Problem Solver.[1]

We have a rather different perspective, however, than the designers of the **ARPAnet**. The host computers of the **ARPAnet** have quite disparate functions and only cooperate in small numbers (primarily in pairs) on a limited basis, typically for statically defined tasks (e.g., file or message transfer). In our distributed problem solver, on the other hand, the nodes may cooperate extensively. All of the nodes may be involved in the solution of a single overall problem. In addition, the type of cooperation may be dynamic; that is, the functions fulfilled by individual nodes may change over the course of solving the problem.

In the case we are considering, then, communications protocols are only a start--a prerequisite for distributed problem solving. They enable bit streams to be passed between processors but do not consider the *semantics* of the information being passed. We need to build upon the work of network and communications protocol designers and focus on *what* to

[1] The horizontal lines should be read as follows. An entity on one side is *connected* to the corresponding entity on the other side via the means named above the line, and *communicates* whatever is named below the line. For examples, an IMP is connected to another IMP via a *circuit* and communicates a *packet*. The solid lines indicate *actual* communication, and the dashed lines indicate *virtual* communication; that is, communication via a software protocol.

say, as opposed to *how* to say it. In ARPAnet terms, we must move above the process-to-process protocol to add yet another layer--a problem-solving protocol (Figure 4.2). In Section 4.1.2 we present one such protocol, where individual nodes are connected by *contracts* and communicate *contract net protocol messages*.

A problem-solving protocol must convey two types of information: task-independent information and task-dependent information. Task-independent information is essentially control information. It specifies what actions are to be taken on receipt of a particular message. This type of information is used for *any* problems to which the problem solver is applied. The task-dependent information corresponds to the *contents* of a message. It is used to guide the actions of a node for a particular problem.

4.1.1.1 Design Goals

The general design goals for a problem-solving protocol are as follows:

1) The protocol should be well-suited to systems that are loosely coupled. As noted in Section 2.2.3, it is important to minimize communication since communications channel capacity is expensive. While careful task partitioning has the greatest potential impact on the amount of communication required, the problem-solving protocol also plays a role. Therefore, the protocol should be efficient in terms of its use of communications resources (i.e., terse).[1]

2) The protocol should foster *distribution* of control and data in order to insure that advantage is taken of the gains in speed and reliability that can be achieved through the use of a distributed architecture. Centralized control could create an artificial bottleneck (slowing the system down) and make it difficult for the system to recover from failure of critical components [Gonzalez, 1972] [Farber, 1972]. Distributed control also helps to create a flexible system; that is, a number of different (potentially dynamic) approaches to problem solving can be implemented. For example, some nodes can proceed in a *data-directed* fashion while others proceed in a *model-directed* fashion [Buchanan, 1978].

3) The protocol should aid in maintaining the focus of the problem solver, to combat the combinatorial explosion that besets almost all *AI* programs. For a uniprocessor, focus involves selection of the most appropriate task to be executed at each instant in time so as to minimize the total number of KS executions (and overall processing time) [Hayes-Roth, 1977]. For a distributed problem solver, focus can be reformulated as connection of the most appropriate tasks to be executed with KSs capable of their execution (the connection problem discussed in Section 1.4).

---

[1] Current evidence [Galbraith, 1974] suggests that effective human organizations operate in an analogous manner, minimizing unnecessary communications among the members.

4.1.2 The Contract Net Framework: Communications And Control

The contract net framework [Smith, 1977] [Smith, 1978b] uses an announcement-bid-award sequence of *contract negotiation* to solve the connection problem. It views task distribution as an interactive process, one that entails a discussion between a node with a task to be executed and nodes that may be able to execute the task. The protocol defines a set of messages that has so far proven adequate for both control and data distribution. We present a simplified description of the contract net approach in this section.[1]

A contract net is a collection of interconnected processor nodes whose interactions are governed by a problem-solving protocol based on the contract metaphor. Each node in the net operates asynchronously and with relative autonomy. The execution of an individual task is handled as a contract. A node that generates a task advertises existence of that task to the other nodes in the net with a ***task announcement***, then acts as the ***manager*** of that task for its duration. In the absence of any information about the specific capabilities of the other nodes in the net, the manager is forced to issue a *general broadcast* to all nodes. If, however, the manager possesses some knowledge about which of the other nodes in the net are likely candidates, then it can issue a *limited broadcast* to just those candidates. Finally, if the manager knows exactly which of the other nodes is appropriate, then it can issue a *point-to-point* announcement.[2] As work on the problem progresses, many such task announcements will be made by various managers.

Nodes in the net have been listening to the task announcements and have been evaluating their own level of interest in each task with respect to their specialized hardware and software resources. When a task is found to be of sufficient interest, a node submits a ***bid***. A bid indicates the capabilities of the bidder that are relevant to execution of the announced task. A manager may receive several such bids in response to a single task announcement; based on the information in the bids, it selects one (or several) node(s) for execution of the task. The selection is communicated to the successful bidder(s) through an ***award*** message. These selected nodes assume responsibility for execution of the task, and each is called a ***contractor*** for that task.

A contract is thus an explicit agreement between a node that generates a task (the manager) and a node that executes the task (the contractor). Note that establishing a contract is a process of mutual selection. Available contractors evaluate task announcements made by several managers until they find one of interest; the managers then evaluate the bids received from potential contractors and select one they determine to be most appropriate. Both parties to the agreement have evaluated the information supplied by the other and a mutual selection has been made.

---

[1] The basic idea of contracting is not new. A rudimentary bidding scheme, for example, was used for resource allocation in the Distributed Computing System [Farber, 1972]. The contract net protocol, however, is the first to use the idea for *AI* problem solving. It therefore pays more attention to the type of information that must be transferred between nodes to solve the connection problem than do earlier protocols.

[2] Restricting the set of addressees of an announcement (which we call ***focused addressing***) is typically a heuristic process, since the information upon which it is based may not be exact (e.g., it may be inferred from prior responses to announcements).

The contract negotiation process is expedited by three forms of task-dependent information contained in a task announcement. An ***eligibility specification*** lists the criteria that a node must meet to be eligible to submit a bid. This specification reduces message traffic by pruning nodes whose bids would be clearly unacceptable. A ***task abstraction*** is a brief description of the task to be executed and allows a potential contractor to evaluate its level of interest in executing this task relative to others that are available. An abstraction is used rather than a complete description in order to reduce the length of the message (and hence message traffic).[1] Finally, a ***bid specification*** details the expected form of a bid for that task. It enables a potential contractor to transmit a bid that contains only a brief specification of its capabilities that are relevant to the task (this specification is called a ***node abstraction***), rather than a complete description. This both simplifies the task of the manager in evaluating bids and further reduces message traffic.[2]

Contracts are queued locally by the node that generates them until they can be awarded. If no bids are received for a contract by the time an ***expiration time*** (included in the task announcement) has passed, then the contract is re-announced. This process is repeated until the contract can be awarded.[3]

The award message contains a ***task specification***, which includes the complete specification of the task to be executed. After the task has been completed, the contractor sends a ***report*** to its manager. This message includes a ***result description***, which communicates the results that have been achieved during execution of the task.[4]

The manager may terminate contracts as necessary (with a ***termination*** message), and further (sub)contracts may be let in turn as required by the size of a contract or by a requirement for special expertise or data that the contractor does not have.

It is important to note that individual nodes are not designated a priori as managers or contractors. These are only roles, and any node can take on either role. During the course of problem solving, a particular node normally takes on both roles (perhaps even simultaneously for different contracts). This leads to more efficient utilization of nodes, as compared, for example, to schemes that do not allow nodes that have contracted out subtasks to take on other tasks while they are waiting for results **[Harris, 1977]**. Individual nodes, then, are not statically tied to the control hierarchy.

Note also that the idea of transfer of expertise between nodes (via transfer of

---

[1] One component of the abstraction is the *task type*, or generic classification of the task. See Section 6.8 for details.

[2] The information that makes up the eligibility specification, task abstraction, and bid specification for any given example must be supplied by the applications programmer. We discuss one method for encoding this information in Section 4.1.3. We saw examples of this type of information in Chapter 3.

[3] This is a simplified version of the actual process (and does not work in the case of a task that cannot be executed due, for example, to lack of sufficient data); see Chapter 6 for the complete description.

[4] The encoding of the task specification and the result description is discussed in Section 4.1.3.

procedures or data) can be readily handled by the protocol. It can be handled as a standard contract between a node that announces (in effect) *I need the code for <procedure-description>*, and a node that bids on the task by indicating that it has the required information.

To review, the normal method of negotiating a contract is for a node (called the manager for a task) to issue a task announcement. Many such announcements are made over the course of time. Other nodes are listening and submit bids on those announcements for which they are suited. The managers evaluate the bids and award contracts to the most suitable nodes (which are then called contractors for the awarded tasks).

The normal contract negotiation process can be simplified in some instances, with a resulting enhancement in the efficiency of the protocol. If a manager knows exactly which node is appropriate for execution of a task, a ***directed contract*** can be awarded. This differs from the ***announced contract*** in that no announcement is made and no bids are submitted. Instead, an award is made directly. In such cases, nodes awarded contracts must acknowledge receipt and have the option of refusal.[1]

The protocol has also been designed to allow a reversal of the normal negotiation process. When the processing load on the net is high, most task announcements will not be answered with bids because all nodes will be already busy. Hence, the protocol includes a ***node availability announcement*** message. Such a message can be issued by an idle node. It is an invitation for managers to send task announcements or directed contracts to that node. We will discuss this aspect of contract acquisition further in Chapter 6.

Finally, for tasks that amount to simple requests for information, a contract may not be appropriate. In such cases, a request-response sequence can be used without further embellishment. Such messages (that aid in the distribution of data as opposed to control) are implemented as ***request*** and ***information*** messages. The request message is used to encode straightforward requests for information for which contracting is unnecessary. The information message is used both as a response to a request message and as a general data transfer message.

4.1.3 The Common Internode Language

We have noted that the problem-solving protocol directly encodes task-independent information and provides slots for task-dependent information. In this section we discuss the advantages of using a formal language, which we call the *common internode language*, to

---

[1] Note that a directed contract is not the same as a standard contract in which the announcement is made to only one node (via a point-to-point message). The directed contract is used by a manager that knows that the addressed node can execute the task. A task announcement, on the other hand, can be addressed to a single node if the manager is not sure that the addressed node can execute the task, and must wait for the return bid before awarding the contract. The focused addressing in both cases is used in an attempt to reduce message processing overhead. The directed contract can further reduce communications and message processing overhead by eliminating the announcement and bid messages.

encode task-dependent information. We compare this with less-structured means for encoding such information.

The common internode language is the means by which task-dependent information is communicated between nodes in the contract net framework. The language provides the primitive elements with which such items as the task abstraction, eligibility specification, and bid specification are encoded. It is the medium by which nodes discuss tasks and KSs, as well as pose the questions that arise during the contract negotiation process about eligibility to bid on tasks, rank ordering of tasks, and control of task distribution. The language has a grammar and a core vocabulary of terms that appear to be of use in a wide range of applications. The vocabulary is intended to be extensible; that is, new items can be added for specific applications (we saw examples of such items in Chapter 3; e.g., *signal*, *group*, *vehicle*, etc.).

The common internode language permits a new node to isolate the information it must have to execute a particular task. For example, a statement like *MUST-HAVE PROCEDURE NAME extend-board* (from the N Queens example) can be parsed by all nodes and translated into a search of their local knowledge bases for the named procedure. A node that does not have the procedure knows from the unsuccessful search that if it wants to execute the task for which this statement is part of the eligibility specification, then it must acquire the procedure. This mechanism works in general, first, because all nodes understand the (task-independent) contract net protocol, which enables them to identify the slots that contain task-dependent information (e.g., the eligibility specification of a task announcement); and second, because they know how to parse the common internode language, which enables them to identify the information they do not possess. The language is therefore an aid to dynamic distribution of knowledge (discussed in Section 4.2.2), since a node can determine what information it needs and specify that information to other nodes in terms of a request.[1]

It might appear that a common internode language could profitably be avoided by encoding all knowledge required for contract negotiation in task-dependent procedures. Such procedures would need to be transmitted once for each type of task. Henceforth all messages for which the procedures were applicable would be shorter than they would be if the information were encoded in a common language in each message. This procedural approach would therefore seem to offer more efficient operation, due to shorter messages and simpler message processing.

If we examine the procedural approach carefully, however, it becomes clear that it does not in fact eliminate the need for a common language. Each of the messages in the contract net protocol that includes slots for task-dependent information causes local knowledge base searches to be initiated. In the case of the task announcement, for example, a potential contractor must search its local knowledge base to see if it meets the eligibility specification of the announcement. Regardless of whether a common language or a task-dependent procedure is used to interpret the specification, it is the same set of items for which the local knowledge base must be searched; that is, both the manager and the

[1] The constrained format of a language provides similar advantages with respect to machine readability as those offered by rules in a production system [Davis, 1977a].

potential contractor must have the same interpretation for the items in the specification, and this implies the need for a common language.

In addition, a multiplicity of procedures would be required, one to specify the interpretation for every kind of message that might be transmitted for each type of task executed in the net. This would restrict the style of communication, especially in regard to requests and information messages.

Finally, the use of a language enables local processing by a node to evaluate task announcements and bids from its own point of view. A node is responsible for searching its own knowledge base to determine, for example, its eligibility to bid on a task. The language specifies the necessary objects, but not how a knowledge base is to be searched to find them. Because one node (a manager) does not specify a procedure to be executed by another node (a potential contractor) to determine eligibility, local knowledge bases that have somewhat different structures (but a common interpretation of objects) can be accommodated. It is not clear that this offers greater power than the procedural approach, but the perspective (i.e., of making a node responsible for examining its own knowledge base, as opposed to allowing other nodes to have more direct access) is more in keeping with the metaphor of cooperating, but autonomous, experts.

A common internode language is therefore desirable, although the efficiency of procedural communication can still be retained for specialized communication. Two nodes that are linked via a contract, for example, can adopt a more compact form of communication for their messages, since no other nodes need interpret the messages. This compact form of communication can be viewed as a specialized language that the nodes use to communicate with other nodes that share their expertise (a familiar mode of operation for cooperating experts). In the DSS example, once the area and vehicle contractors have established communication through the contract negotiation process, they might alter the language in which they communicate in order to reduce the length of messages and simplify message processing.

4.1.4 Summary And Evaluation

The contract net protocol contains message types that appear to be useful for a wide range of problem-solving interactions. Each message contains slots for task-dependent information germane to carrying out the intended function of the message. In a task announcement, for example, slots are provided for task-dependent information that enables a node receiving the message to decide whether or not it is able to execute the task (eligibility specification), to rank the announced task relative to other announced tasks (task abstraction), to formulate an appropriate bid on the task (bid specification), and to know the time period during which the announcement is valid (expiration time). In the signal task announcement of the DSS example, the eligibility specification indicated that only nodes located in the same area as the manager and having sensing capabilities should bid on the task; the task abstraction indicated the type of task and the position of the manager; and the bid specification indicated that a bidder should specify its position in its bid.

The power of the protocol is most visible when tasks require specialized KSs (and not

just processing power), when the appropriate KSs for a given task are not known a priori (i.e., there is potential nondeterminism at runtime), and when the tasks are large enough to justify a more substantial transfer of information before invocation than is generally allowed in problem solvers (consider again, for example, the signal task in the DSS example). The protocol, however, has been designed to be useful for distributed problem solving at different levels of complexity. The N Queens problem, for example, required only simple forms of the messages of the protocol. Many of the message types were not needed (e.g., requests and information messages). In addition, a complex contract negotiation process was not needed. This reduced the length of the messages (e.g., a minimal task abstraction was required) and reduced message-processing overhead (e.g., contracts were awarded to the first bidder).

We can now consider how well the contract net protocol meets the design goals specified earlier for a problem-solving protocol.

1) *<The protocol should be well-suited to systems that are loosely coupled.>*
The protocol is well suited to loosely coupled systems in two respects. First, it provides a very general form of guidance in determining appropriate partitioning of problems: the notion of tasks executed under contracts is appropriate for a kernel-size larger than that typically used in problem-solving systems. Second, the protocol is efficient with respect to its use of communications channels. Such efficiency helps to preserve whatever loose-coupling character is already present in the system as a result of problem partitioning. The information in task announcements, for instance, helps minimize the amount of channel capacity consumed by communications overhead. Consider the increased communications overhead that would result if for each of the signal tasks of the DSS example, all sensor nodes were to respond to each of the managers for the tasks. Much of this traffic is eliminated by the eligibility specifications of the task announcements (i.e., potential bidders must be in the same area as the managers, as well as have sensing capabilities).

2) *<The protocol should foster distribution of control and data.>*
The use of autonomous nodes interacting through a process of contract negotiation fosters distribution of control and data throughout the system. Decentralized control and two-way links between managers and contractors enhance system reliability, because they enable recovery from individual component failure. The failure of a contractor, for example, is not fatal, since its manager can re-announce the appropriate contract and recover from the failure. This strategy allows the system to recover from any node failure except that of the node that holds the original top-level problem.[1]

3) *<The protocol should aid in maintaining the focus of the problem solver.>*
Maintenance of focus is perhaps the most difficult of the design goals to meet, and we do not yet have a full understanding of the underlying issues involved. We have tried to enable a node to make the most appropriate selection of tasks by providing slots for task-

[1] At the top level, contracting can distribute control *almost* completely, hence removing the bottlenecks that centralized controllers create. There still remains, however, the reliability problem inherent in having only a single node responsible for the top-level problem. Since this cannot be handled directly by the manager-contractor links, standard sorts of redundancy are required.

dependent information in the messages (e.g., the task abstraction and bid specification). The protocol, however, cannot do more than this. The ultimate responsibility for task (and KS) selection lies with the task announcement and bid evaluation procedures.

We noted in Chapter 3 that each node in a contract net maintains a list of the best recent task announcements it has seen--a kind of window on the tasks at hand for the net as a whole. This window enables the nodes to compare announcements, to assist in making the best task selections over time. In a similar fashion, managers are able to compare bids from different nodes. This type of mutual selection was used, for example, in setting up the signal contracts of the DSS.[1]

4.2 Knowledge Organization

We now consider the last major component of the framework--the knowledge organization. We view the contract negotiation process from a different perspective, examining the kinds of knowledge that are used, the ways in which the knowledge is retrieved, and the ways in which it is distributed among the nodes.

There are two major concerns for knowledge organization (Figure 4.3): retrieval and distribution.

Retrieval indicates the ways in which knowledge is accessed by a node, either from its local knowledge base or from that of a remote node. As shown in the figure, retrieval itself has two major concerns: *partitioning and indexing.*

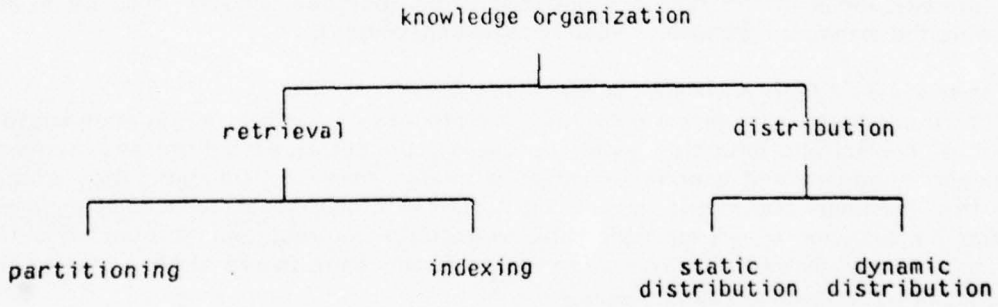


Figure 4.3. The Knowledge Organization Component Of The Framework.

Partitioning indicates the manner in which the knowledge is broken up or modularized to simplify its acquisition, alteration, and distribution. *Indexing* indicates the *handles* placed on the knowledge modules so that they can be accessed. The main issue in indexing is how the knowledge is used (e.g., to select suitable KSs for a particular task, or to select suitable tasks for a particular KS).

---

[1] We present a detailed discussion in Chapter 6.

Distribution indicates where the knowledge resides, that is, in which nodes. There are two aspects to distribution of knowledge: *static distribution*, dealing with the question of how knowledge is pre-loaded into the individual nodes (i.e., a priori distribution), and *dynamic distribution*, dealing with acquisition of knowledge by a node as work on a problem progresses.

The various concerns in knowledge organization are interdependent. Partitioning, for example, generally affects both the indexing to be used and the static distribution of knowledge among the processor nodes. In addition, the knowledge organization is closely tied to communication. The common internode language, for example, encodes task-dependent information that is used by the message-processing procedures to store and retrieve information from the local knowledge bases of individual nodes. The structure of a local knowledge base therefore determines in large measure the structure of the common internode language used to access it.

4.2.1 Retrieval Of Knowledge

In the contract net framework, the main issue in retrieval of knowledge is connecting nodes with tasks to be executed to nodes with KSs that are appropriate to execute those tasks. Indexing therefore plays a central role. Partitioning, on the other hand, is not essentially different for a distributed problem solver than it is for a uniprocessor problem solver, and we only discuss it briefly.

4.2.1.1 Partitioning

In a distributed problem solver there is some task-independent *systems* knowledge that must exist in all nodes. In our framework this is the knowledge necessary to handle contracts, to process the messages of the contract net protocol, and to process statements expressed in the common internode language. The knowledge of interest for partitioning, on the other hand, is primarily task-dependent. We will briefly consider two aspects of partitioning: formation of knowledge-sources and storage of data in a local knowledge base.

It is common in *AI* problem solvers to partition expertise into domain-specific *knowledge-sources*, each of which is expert in a particular part of the overall problem domain. KSs are typically formed empirically, based on examination of different types of knowledge that can be brought to bear on a particular problem. In a speech-understanding problem, for example, knowledge is available from the speech signal itself, from the syntax of the utterances, and from their semantics **[Erman, 1975]**. The decisions about which KSs are to be formed is often made in concert with the formation of a data hierarchy for a problem. KSs are typically chosen to handle data at one level of abstraction or to bridge two levels (see, for example **[Erman, 1975]**, **[Engelmore, 1977]**, and **[Nii, 1978]**).

In a distributed processor there is only one additional consideration to be factored into decisions about formation of KSs; that is, the KSs themselves will likely be distributed to different nodes, and their interactions will therefore often be more expensive than in a uniprocessor. As a result, the kernel-size of the KSs must be chosen carefully, based on the characteristics of the distributed processor. As we have seen in Section 2.2.3, kernel-size is one of the major factors that determines the degree of coupling between nodes.

The second aspect in partitioning is how knowledge is to be stored in a local knowledge base. In the individual nodes of a contract net (as implemented in CNET), knowledge is partitioned about *conceptual objects* [Dahl, 1968], [Bobrow, 1977], such as *nodes*, *tasks*, *procedures*, and *contracts*. Objects are linked together by *attributes*, which in turn have *values* (which may be other objects). A *contract* object, for example, has a *manager* attribute whose value is a *node* object.[1]

A local knowledge base is initialized with a predetermined core structure of objects with associated attributes and values.[2] The core structure is task-independent and can be augmented in an individual node with task-dependent objects, attributes, and values as required by the problem at hand. In the DSS, for example, *signal* and *signal-group* are objects that must be added for that particular application.

4.2.1.2 Indexing

The aim of indexing knowledge is to insure that it can be retrieved for application to particular tasks (e.g., by invocation of KSs). In the contract net framework, KSs are normally invoked through a mutual selection process of contract negotiation. Hence, in the following discussion of indexing, we consider *(i)* the questions that must be answered by nodes (managers and potential contractors) during the contract negotiation process; and *(ii)* the types of knowledge that are used by the manager and potential contractors to effect this negotiation. We will discuss implementation issues in Chapter 6.

A manager has two questions to answer during the contract negotiation process. First,

(1) ***To whom do I address my task announcement?***

The responses to this question come in the form of bids on the announced task from eligible candidate contractors. Having received these bids, a manager must answer the question,

(2) ***How can I select the best candidates from among the potential contractors for my task?***

A node that receives an announcement (i.e., a potential contractor) must answer two complementary questions during the negotiation process. First,

(3) ***Am I relevant to this task and is it appropriate for me to consider making a bid?***

The answer to questions of this form enable a node to find candidate tasks. In addition, it must also answer the question,

(4) ***Do I wish to submit a bid on this task?***

---

[1] A complete specification of the objects and attributes that have been implemented in CNET is presented in Chapter 6.

[2] See Section 6.6 for a detailed description of the core structure.

Both managers and contractors then, attempt to find eligible candidates (either nodes or tasks) and rank them from their individual points of view.

In order to facilitate the contract negotiation process, we find it convenient to specify knowledge as being either *task-centered* or *knowledge-source-centered* (KS-centered).

Task-centered knowledge is indexed from the point of view of a particular task and provides information about KSs with respect to that task. It is used by a manager to assist in answering questions (1) and (2) during contract negotiation. Two major forms can be imagined:

(a) ***IF I have a task of the form [...] to be executed, THEN KSs of the form [...] are potentially useful.***

and,

(b) ***IF I have a task of the form [...] to be executed, THEN KSs of the form [...] are more useful than KS's of the form [...].***

KS-centered knowledge, on the other hand, is indexed from the point of view of a particular KS, providing information about tasks with respect to that KS. It is used by a potential contractor to assist in answering questions (3) and (4) above. Again, two major forms can be imagined:

(c) ***IF my knowledge base contains information of the form [...], THEN tasks of the form [...] are appropriate for me.***

or,

(d) ***IF my knowledge base contains information of the form [...], THEN tasks of the form [...] are more appropriate for me than tasks of the form [...].***

Both kinds of knowledge are used during the contract negotiation process. Task-centered knowledge is used first by a manager to determine the subset of nodes to which to address a task announcement; that is (a) provides the answer to question (1). This type of knowledge reduces message traffic and message processing overhead because it enables focused addressing, as in the DSS example, where the monitor node used task-centered knowledge to directly address the potential area contractors in the area task announcements.

Task-centered knowledge is also used by a manager to determine the best course of action once responses are received; that is (b) provides the answer to question (2). This type of task-centered knowledge was used by the area contractors in the DSS example to decide whether or not to start up new vehicle contractors in response to receipt of new signal group information.

The bid evaluation procedures that use task-centered knowledge to select the next KS to invoke are thus an appropriate location for strategy information that guides the

operation of the problem solver (i.e., information about which KS to use when more than one is applicable). This is the type of knowledge encoded as *meta-rules* by [Davis, 1977c]. We will return to a more general discussion of this type of knowledge in Section 5.3.

KS-centered knowledge is used by a potential contractor that receives an announcement, first to determine that it is relevant to the announced task; that is, (c) provides the answer to question (3). This type of KS-centered knowledge was used by nodes in the DSS example to determine that they were eligible to bid on signal tasks.

Associating such knowledge with individual KSs allows concurrency in a distributed problem solver because many KSs can simultaneously determine their relevance to a task; that is, each KS carries information allowing it to determine the range of tasks to which it is relevant. If individual KSs were unable to do this (i.e., if no KS-centered knowledge were used), then a manager would be forced to query each potential contractor serially.

KS-centered knowledge is also used by a node to select the task it wishes to execute next; that is, (d) provides the answer to question (4). In the DSS example, a potential signal contractor used this type of KS-centered knowledge to select the particular signal task on which to submit a bid. The task announcement evaluation procedures that use KS-centered knowledge are thus another appropriate location for encoding strategies. (We will discuss this further in Section 5.3.)

To summarize, indexing of knowledge according to its utility for selecting KSs or tasks is useful in the contract net framework because *(i)* it helps to minimize message traffic and message processing overhead, *(ii)* it allows concurrency in the connection of managers with contractors, and *(iii)* it specifies useful locations for encoding strategy information.

4.2.2 Distribution Of Knowledge

We noted at the beginning of this section that distribution of knowledge has two aspects--static and dynamic. Static distribution is largely task-specific. KSs that have expertise for particular tasks are generally placed in different nodes. The criteria for a good static distribution of knowledge are minimization of message traffic and avoidance of critical function nodes that could reduce processing speed and create reliability problems. In the DSS, for example, KSs with expertise at a particular level of the data hierarchy were distributed in different nodes. This enabled nodes to carry on simultaneously with computation specific to those levels of the data hierarchy that concerned them (without the need for frequent interactions with nodes operating at different levels of the hierarchy).

Dynamic distribution of knowledge is the means by which nodes can acquire and transfer information and expertise as the problem solving progresses. Such distribution enables more effective use of available computational resources: A node that is standing idle because it lacks information required to execute a previously announced task can acquire the procedures necessary to execute that task. Dynamic knowledge distribution also facilitates the addition of a new node to an existing net; the node can dynamically acquire the procedures and data necessary to allow it to participate in the operation of the net.

Nodes therefore do not have to be functionally defined a priori; that is, any node can acquire the procedures necessary to execute any task that its physical attributes (e.g., memory, peripherals, etc.) will support. The alternatives to dynamic distribution of knowledge are either to force the node to remain idle until it receives a task announcement for which it already has the necessary procedures, or to pre-load each node in the net with all the procedures that will ever be needed for the overall problem. The first alternative reduces the potential processing speed and efficiency of the net by forcing nodes to stand idle, and the second alternative places costly memory requirements on each node.[1]

In the contract net framework, knowledge can be transferred between nodes in three ways. First, a node can transmit a request directly to another node for the transfer of the required knowledge. The response is the knowledge requested (e.g., the code for a procedure). Second, a node can broadcast a task announcement in which the task is a transfer of knowledge. A bid on the task indicates that another node has the knowledge and is willing to transmit it. Finally, a node can note in its bid on a task that it requires particular knowledge in order to execute the task. The manager can then send the required knowledge in the contract award if the bid is accepted.

The first mechanism is used for brief and straightforward requests, where the requesting node knows the address of another node that has the required knowledge. The second mechanism is used when a node does not know the address of a node that has the required knowledge, or cannot completely specify what is required. In this case, more complex processing will be required by a node with the knowledge, and the complexity of the contract negotiation process is justified.[2] The third mechanism is useful when the managing node already has the relevant knowledge but wants to work on some other aspect of the task. In addition, it allows the manager to award a contract to a node that would otherwise stand idle.

---

[1] If the distributed problem solver is only intended to be applied to a single well-understood problem, of course, suitable numbers of specialist nodes can be defined, obviating the need for dynamic distribution of knowledge.

[2] This is also the case when the transfer requires a significant amount of communication.

Chapter 5

Relations To Problem Solving In General

The contract net draws upon a variety of ideas from the *AI* literature. In this chapter we relate the framework to other systems. We also focus on a comparison of the approach to transfer of control provided by the contract net framework with that provided by previous problem-solving formalisms. This comparison makes clear the ways in which the contract net view is unique and offers a natural next step in a progression of mechanisms for effecting transfer of control. We consider as points of comparison the techniques used in subroutine calls, production systems **[Davis, 1977a]**, **PLANNER [Hewitt, 1972]**, **CONNIVER [McDermott, 1974]**, **HEARSAY-II [Erman, 1975]**, a task agenda system (like **AM [Lenat, 1976]**), and the **PUP6** system **[Lenat, 1975b]**. Finally, we compare the use of knowledge in the contract net framework to its use in other systems.

5.1 Other Systems

5.1.1 PLANNER And ACTORS

The **PLANNER [Hewitt, 1972]** goal specification provides a mechanism for advertising a task to a group of KSs instead of invoking a specific KS by name. Theorems in a **PLANNER** knowledge base are given a *pattern* to be matched in a goal specification. A sample pattern from the blocks world might be *(BLOCK1 IN BOX2)*, where the goal to be achieved is to place a block in a box. Theorems that have expertise for putting blocks inside of boxes contain corresponding patterns, say *(?block IN ?box)*, that match the pattern in the goal specification. Such theorems are retrieved by a search of the knowledge base. This is called *pattern-directed invocation*.

The contract net task announcement performs a similar function to that of the **PLANNER** goal specification.[1] Rather than a pattern, however, a node in a contract net is given a series of common internode language statements in the eligibility specification and task abstraction slots of the task announcement message. A node can therefore perform more general local processing to determine its relevance to a particular task. In the DSS, for example, potential signal contractors used the positions of group contractors (part of the task abstraction) to find the closest group contractor to which to submit a bid.

The **ACTOR** model of computation that succeeded **PLANNER** is based on the concept of a group of experts that communicate by passing point-to-point messages **[Hewitt, 1977a]**, **[Hewitt, 1977b]**. In contract net messages, by contrast, a variety of addressing modes is used (general broadcast, limited broadcast, and point-to-point). These different modes serve

[1] It should be noted that in **PLANNER** a goal specification (in its simplest form) is effectively a *broadcast* message to all theorems in the system. We will return to this point in Section 5.2.3.

PRECEDING PAGE BLANK-NOT FILMED

to reduce message traffic and message-processing overhead. More important, the contract net assumes a loose-coupling of tasks, whereas the ACTOR model does not. This assumption implies a difference in the kernel-sizes of tasks into which a problem is decomposed for the two formalisms (large for contractors and small for actors), which results from the different motivations of their designers. Whereas actors have been used as a means of studying fundamental issues involving the nature of computation, control, and program correctness, the contract net is designed as a mechanism for distributed problem solving and hence views its primitive operations in terms of comparatively large, domain-specific tasks.[1]

5.1.2 HEARSAY-II

The concept of a group of cooperating KSs has been used to advantage in the HEARSAY-II speech understanding system [Erman, 1975]. In HEARSAY-II, the KSs were separate, independent, and anonymous modules (i.e., they could be addressed by capability rather than by name). They communicated through a common blackboard, which was used to record the emerging sentence hypothesis. KS independence was enforced to enable the system to operate in the absence of some of the KSs and to allow assessment of individual KS effectiveness. By *KS independence* we mean that each KS made no assumptions about the existence of other KSs. Each was therefore able to function independently, in the absence of any other KSs. This is especially useful in speech understanding applications because of the errorful nature of the speech signal and the difficulty with predicting which of the individual KSs will be able to contribute effectively to the interpretation of any particular utterance.

The contract net draws upon this model with respect to the modularity and independence of KSs. HEARSAY-II provided empirical evidence of the utility of loose-coupling from a problem-solving point of view. As we have seen in Section 2.2.3, loose-coupling is also important in a distributed system for reducing message traffic. HEARSAY-II also demonstrated the need for a common method for communication between KSs [Lesser, 1977]. All KSs knew the structure of the blackboard (i.e., the structure of a hypothesis and the levels of data abstraction). In the contract net framework, nodes can interact because they all know the contract net protocol and the common internode language.

In the distributed environment, a global data structure like a blackboard can cause bottleneck and reliability problems. To avoid these, the contract net framework is designed so that individual nodes communicate in a direct and local manner.

KSs in the HEARSAY model were seen primarily as information-gathering and information-dispensing processes [Reddy, 1975]. Hierarchical control was deemed unnecessary, and this caused some difficulty because there was no way to specify the

---

[1] While both use the word *contract*, the intent of the word in each case is very different. In actor-based systems [Hewitt, 1975], a contract is an expression of what is supposed to be accomplished by an actor and is intended to be an agreement between the (human) implementer of a module and its users. A contract in a contract net, on the other hand, is a method of connecting nodes that have tasks to be executed with nodes that are able to execute those tasks. The agreement is thus between two processor nodes.

processing that had already been applied to a hypothesis and no way to formulate the kind of processing that might yet be applied [Lesser, 1977]. The contract net, on the other hand, is well suited to hierarchical control because it uses direct communication between nodes and a negotiation process for task distribution.

5.1.3 PUP6

The model of a group of human experts cooperating to solve a large problem was also used effectively in PUP6, a system used to write programs for concept formation, grammatical inference, and data retrieval [Lenat, 1975a], [Lenat, 1975b], based on informal specifications.

KSs in PUP6 normally determined for themselves that they were applicable to particular tasks by examining a task agenda and then making requests for control to a central scheduler. This procedure amounted to use of what we have called KS-centered knowledge for invocation. However, some interaction between modules was allowed to effect transfer of control. While this interaction in PUP6 was accomplished by pattern matching, the contract net expands on this through the use of a contract negotiation process, based on a common internode language. Both task-centered knowledge and KS-centered knowledge are used in negotiation.

PUP6 made no allowance for acquired expertise, since each module in the system had a standard set of parts that did not vary over time. Contract nodes, on the other hand, have a standard core structure but, in addition, have a common internode language that enables them to acquire expertise via transfer of procedures and data.

5.2 A Progression In Mechanisms For Transfer Of Control

5.2.1 Terminology

In a uniprocessor system, when a process decomposes a problem it is working on and selects another process to work on a selected subtask, it yields (perhaps temporary) control. In a distributed system, however, when one processor decomposes a problem it is working on and hands one of the resulting subtasks to another processor, both processors continue working on their respective tasks. We therefore use the term *task distribution* for what is more traditionally referred to as *transfer of control*.

Since all of the systems we wish to use for comparison were designed for uniprocessors, we will adopt this terminology and make the comparison on the basis of transfer of control. This decision provides a common language for comparison without losing sight of the important issues. It also serves to demonstrate that the issues we deal with in this section are fundamental issues of KS invocation and problem solving, independent of distributed processing.

5.2.2 The Basic Questions And Fundamental Differences

To make clear the place of the contract net in the sequence of invocation mechanisms that have been created, we consider the process of transfer of control from the perspective of both the caller and the respondent. We focus in particular on the aspects of selection and consider what opportunities a calling process has for selecting an appropriate respondent and what opportunities a potential respondent has for selecting the task on which to work. In each case we consider two basic questions that either the caller or the respondent might ask:

What is the character of the choice available? (i.e., at runtime, what choice of respondents does the caller have and what choice of callers does the respondent have?)

On what kind of information is that choice based? (e.g., are potential respondents given, say, a pattern to match, and is the caller given information on how the match was made by potential respondents?)

The answers to these questions will demonstrate how the contract net view of control transfer differs from that of the earlier formalisms with respect to:

(a) *Information transfer*: The announcement-bid-award sequence means that there is more information, and more complex information, transferred in both directions (between caller and respondent) before invocation occurs.

(b) *Local selection*: The computation devoted to the selection process, based on the information transfer noted above, is more extensive and more complex than that used in traditional approaches. It is *local* in the sense that selection is associated with, and specific to, an individual KS (rather than embodied in a global evaluation function).

(c) *Mutual selection*: The local selection process is symmetric, in the sense that the caller evaluates potential respondents from its perspective (via the bid evaluation procedure) and the respondents evaluate the available tasks from their perspective (via the task evaluation procedures).

5.2.3 The Comparison

5.2.3.1 Other Systems

Subroutine invocation represents a degenerate case, since all the selection is done ahead of time by the programmer and is *hardwired* into the code. The possible respondents are named explicitly at compile-time and there is no opportunity for choice or nondeterminism at runtime.

A degree of nondeterminism for the caller is evident in traditional production rule systems, since a number of rules may be retrieved at once. A range of selection criteria have been used (see [Davis, 1977a]), but these have typically been implemented with a single syntactic criterion hardwired into the interpreter.

PLANNER's pattern-directed invocation provides a facility at the programming language level for nondeterministic KS retrieval and offers, in the *recommendation list*, a specific mechanism for encoding selection information. The THUSE construct provides a way of specifying which KSs (theorems) to try in which order, while the theorem base filter (THTBF) construct offers a way of invoking a predicate function of one argument (the name of the next theorem whose pattern has matched the goal) that can *veto* the use of that theorem.

Note that there is a degree of selection possible here, selection that may involve a considerable amount of computation (by the theorem base filter), and selection that is local, in the sense that filters may be goal-specific. However, the selection is also limited in several ways. First, in the standard invocation mechanism, the information available to the caller is at best the name of the next potential respondent; in effect, a one-bit answer of the form *Yes, I match that pattern*. The caller does not receive any additional information from the potential respondent (such as, for instance, exactly how it matched the pattern), nor is there any easy way to provide for information transfer in that direction. Second, the choice is, as noted, a simple veto based on just that single KS. That is, since final judgement is passed on each potential KS in turn, it is not possible to make comparisons between potential KSs or to pass judgment on the whole group and choose the one that looks (by some measure) the best. Both of these shortcomings could be overcome if we were willing to create a superstructure on top of the existing invocation mechanism, but this would be functionally identical to the announcement-bid-award mechanism described above. The point is simply that the standard PLANNER invocation mechanism has no such facility, and the built-in depth-first search with backtracking makes it expensive to implement.

CONNIVER [McDermott, 1974] represents a useful advance in offering nondeterministic KS invocation, since the result of a pattern-directed call is a *possibilities list* containing *all* the KSs that match the pattern. While there is no explicit mechanism parallel to PLANNER's recommendation list, the possibilities list is accessible as a data structure and can be modified to reflect any judgments the caller might make concerning the relative utility of the KSs retrieved. Also, paired with each KS on the possibilities list is an association-list of pattern variables and bindings, which makes possible a determination of how the calling pattern was matched by each KS. This mechanism offers the caller some information about each respondent that can be useful in making the judgments noted above. It does not, however, allow the respondent to perform local processing to determine its relevance.

The HEARSAY-II system illustrates a number of similar facilities in an data-directed system. In particular, the focus of attention mechanism has a pointer available to all the KSs that are ready to be invoked (so it can make comparative decisions), as well as information (in the *response frame*) for estimating the potential contribution of each of the KSs. The system can effect some degree of selection regarding the KSs ready for invocation and has available to it a body of knowledge about each KS on which to base its selection. The response frame thus provides information transfer from respondent to caller that, while fixed in format, is more extensive than previous mechanisms. There is also a fair amount of computation devoted to the selection process, but note that the selection is not local, since there is a single, global strategy used for every selection.

There are several things to note about the systems reviewed thus far. First, we see an increase in the amount and variety of information that is transferred (before invocation)

from caller to respondent (e.g., from explicit naming in subroutines, to patterns in PLANNER) and from respondent to caller (e.g., from no response in subroutines to the response frames of HEARSAY-II). Note, however, that in no case do we have available a general information transmission mechanism. In all cases, the mechanisms have been designed to carry one particular sort of information and are not easily modified. Second, we see a progression from the retrieval of a single KS to the explicit collection of the entire set of potentially useful KSs, providing the opportunity for more complex varieties of selection. Finally, note that all the selection so far is from one perspective; the selection of respondents by the caller. In none of these systems do the respondents have any choice in the matter.

A system where respondents perform the selection is a task agenda system (like AM, [Lenat, 1976]) in which there is a central *task blackboard* that contains an unordered list of tasks that need to be performed. As a KS works on its current task, it may discover new (sub)tasks that require execution and add them to the blackboard. When a KS finishes its current task, it looks at the blackboard, evaluates the lists of tasks there, and decides which one it wants to execute. Note that in this system the respondents have all the selection capability; that is, rather than having a caller announce a task and evaluate the set of KSs that respond, we have the KSs examining the list of tasks and selecting the one they wish to work on.

PUP6 was the first system to view transfer of control as a *discussion* between the caller and potential respondents. If, in response to a task broadcast, a KS receives more than one offer to execute a task, it may ask questions of the respondents to determine which of them ought to be used. While this interchange is highly stylized and not very flexible, it does represent an attempt to implement explicit two-way communication.

5.2.3.2 The Contract Net Framework

The contract net differs from these approaches in several ways. First, from the point of view of the caller, the standard task broadcast-and-response interchange has been improved by making possible a more informative response. That is, instead of the traditional tools that allow the caller to receive simply a list of potential respondents, the contract net has available a mechanism that makes it possible for the caller to receive an extensive description of potential utility from each respondent. This may help to minimize backtracking, since greater care can be taken in making decisions about invocation.

Second, the contract net emphasizes the utility of local selection. An explicit place in the framework has been provided for mechanisms in which both the caller (in the bid evaluation procedure) and the respondents (in the task evaluation procedure) can invest computational effort in selecting KSs for invocation or in selecting tasks to work on, respectively. These selection procedures are also *local* in the sense that they are associated with and written from the perspective of the individual KS (as opposed to, say, HEARSAY-II's global focus of attention function). While we have labelled this process *selection*, it might more appropriately be labelled *deliberation*. This would clarify that its purpose is, for example, for the caller to decide in general what to do with the bids received and not merely which of them to accept. Note that one possible decision is that *none* of the bids is adequate and thus none of the potential respondents would be invoked (instead, the task

may be re-announced later). This choice is not typically available in other problem-solving systems and hence emphasizes the wider perspective taken by the contract net on the transfer of control issue.

Finally, and perhaps most important, there appears to be a novel symmetry in the transfer of control process. Recall that PLANNER, CONNIVER, and HEARSAY-II all offered the caller some ability to select from among the respondents, while AM allowed the respondents to select from among the tasks. The contract net, however, relies on the notion of contract negotiation as a metaphor and emphasizes an interactive, *mutual selection* process where task distribution is the result of a discussion between processors. As a result of the information exchanged in this discussion, the caller can select from among potential respondents (with its bid evaluation procedure) while the KSs can select from among potential tasks (with their task evaluation procedures).

5.3 Use Of Knowledge

We have paid particular attention to partitioning and indexing of knowledge because it is our belief that appropriate partitioning and indexing enables the programmer to make explicit statements about strategy. This, in turn, helps discourage the use of methods that achieve the intentions of the programmer via side-effects.

As an example of control by side-effect, consider the multiple priority level task agenda (from [Davis, 1977b]). Suppose that a programmer wants to insure a particular partial ordering of tasks on the agenda so that tasks in some set A are executed before tasks in another set B. If he has no direct way of stating the order in which tasks are to be executed, he is forced to use an indirect mechanism. One such mechanism might be to assign a higher priority to tasks in A and a lower priority to tasks in B. Unfortunately, this type of indirection causes several problems. First, the code is difficult to understand and debug because the priority adjustments are not explicitly related to the desired effect; once the priorities have been set there is no record of why they were set in this particular way. Second, it is difficult to change such a mechanism in response to new demands on the program or to new knowledge because the priorities may interact in ways that are difficult to foresee. For example, adjustment of the priority assigned to tasks in A may require corresponding adjustment of that assigned to tasks in B.

Systems with explicit statements of strategy and control, on the other hand, tend to be more comprehensible and flexible, in the sense that it is relatively easy to add or modify KSs, and to employ dynamic control structures (e.g., control structures which move between the data-driven and model-driven extremes as operation proceeds).

The contract net framework indexes explicit statements about strategy and control in two ways: according to tasks (task-centered knowledge) and according to KSs (KS-centered knowledge). We discuss below the utility of the two indexing methods.

It is useful to index knowledge according to tasks when the tasks are well-defined but the specific application of KSs is difficult to predict in advance (e.g., MYCIN [Shortliffe, 1976]). In MYCIN, task-centered knowledge has been used to encode strategies in the form

of explicit meta-rules [Davis, 1977c]. These rules indicate which of the task-dependent rules in the medical knowledge base should be applied in a particular situation, and in which order.

In the contract net framework, the task-centered knowledge used by a manager, together with the hierarchical control afforded by the manager-contractor structure itself is useful for coordination of multiple KSs (contractors). The KSs themselves are not in a position to coordinate their efforts, but the manager can fill this role. A system that uses only KS-centered knowledge, on the other hand, has difficulty with KS-coordination, as for example, in PUP6. A common solution to this problem is introduction of a central scheduler (as in HEARSAY-II) with access to information about *all* of the KSs, so that it can coordinate their actions.

By contrast, it is useful to index knowledge according to KSs in applications where the exact set of tasks is unknown a priori but where there exists a known set of tools. As an example, consider HEARSAY-II. Each KS has an associated *stimulus-frame* [Lesser, 1977]. These frames indicate to the scheduler that a KS is relevant to extending the current best hypothesis in response to a change on the blackboard.

In a distributed system it is also more efficient to associate knowledge about individual KS utility and characteristics with the KS itself (i.e., the node that embodies the KS) than to associate the knowledge with many tasks (spread across many nodes). This approach minimizes the number of nodes that must store and access such information. It also means that new types of KSs can be more readily added to the net without having to reprogram existing nodes. This feature is particularly useful in a DSS, for example, where sensor nodes often have limited lifetimes, making replacement of existing nodes a frequent requirement.

Both task-centered knowledge and KS-centered knowledge have been commonly used in earlier *AI* problem-solving systems. The contract net framework, however, appears to be the first in which the two types are used together. We have already seen the advantages offered by this approach in terms of transfer of control as compared to earlier *AI* systems.

Chapter 6

The Contract Net - Systems Considerations

6.1 Contract Node Architecture

We can view a node as having three major functional components: a *contract processor* that handles the structures required for distributed processing (e.g., processing of protocol messages, management of node resources, etc.); a *task processor* that carries out the applications computation to complete the task of a contract; and a *communications processor* that handles low-level communication with other nodes (Figure 6.1). These components share a *local knowledge base* that contains the procedures and data required to effect contract handling, task processing, and internode communication. It is important to note that the components shown are strictly functional in nature. They may be implemented in a number of different ways, ranging from a cluster of processors within a node as shown in Figure 6.1, to a cluster of processes within a single processor.[1]

In this chapter we are concerned with contract handling. We will only briefly touch on task processing and low-level communication.

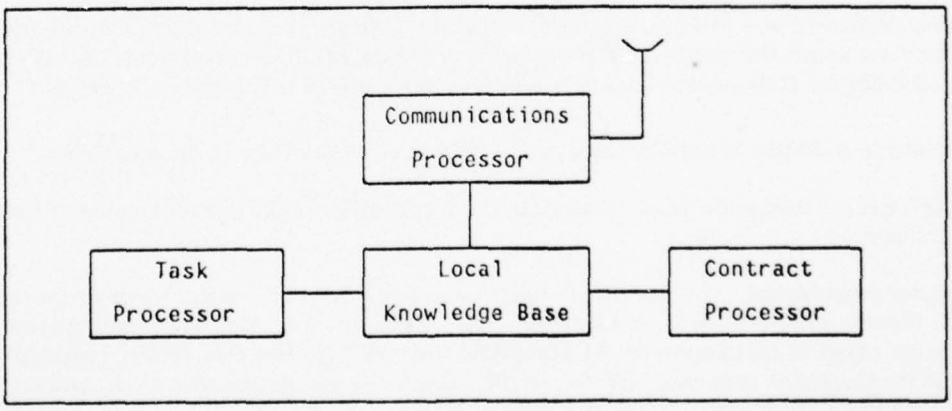


Figure 6.1. Processor Node Architecture.

[1] As an example, DCS [Farber, 1972] was implemented so that messages were passed between processes in a node in essentially the same way that they were passed between nodes.

6.2 Contract Specification

The contract is the basic structure used by a node to organize the knowledge pertinent to management and execution of tasks. The slots of a contract are shown in Figure 6.2. As in all such specifications in this chapter, a variant of BNF is used with the following conventions. Nonterminal symbols are enclosed by "< >", terminal symbols are written without delimiters, and nonterminals whose specific expansion is not germane to the discussion are enclosed by "[ ]". The slots followed by "*" are optional in a basic implementation.

A node maintains a contract structure of this form for each task it has responsibility for executing (i.e., each task for which it has been awarded a contract and is the contractor). In the following sections we will often use the phrases *execute a contract*, and *process a contract* to mean execute or process the task associated with a contract.

```
<contract> -> [name]
              [manager]
              [report-recipients] *
              [related-contractors] *
              <task>
              <results>
              <subcontract-list>
```

Figure 6.2. A Contract.

The following is a brief description of the slots shown in Figure 6.2. Some of these slots are specified when the contract is awarded and others are filled in dynamically as execution of the associated task proceeds. Later sections will discuss this process in detail.

name is a unique identifier used as a reference for the task to be executed.[1]

manager is the node that generated the contract and that is responsible for monitoring its execution.

report-recipients are the nodes to which reports for the contract are to be sent. The default report recipient is the manager. The utility of this slot was suggested by the *continuation* used in messages by ACTORS [Hewitt, 1973]. The two differ, however, in that the ***report-recipient*** receives only *information* about the execution of a task, whereas *control* is passed to the *continuation* in actor-based systems.

related-contractors are the nodes that are working on related contracts (e.g., subcontracts of the same contract). This slot enables horizontal communication between contractors working on tasks at one level of a control hierarchy. All messages from one contractor to another related contractor do not, therefore, need to proceed up to another level of the hierarchy (e.g., to a common manager) and back down to the related contractor.

---

[1] The specific algorithm used in CNET to ensure uniqueness is described in Appendix C.

The ***report-recipients*** and ***related-contractors*** slots are not required in a basic implementation. They do, however, enable greater flexibility in the communication and cooperation that can exist in a contract net. They give a node an explicit indication of the other nodes in the net with which it may be useful to communicate, (e.g., to enable use of focused addressing in task announcements).

task is the structure that contains the complete specification of the task to be executed, together with information that is used in handling the task as a contract for distributed processing. The information in a task structure is discussed in Section 6.8.

results are the outcome of executing a task. They are reported to the report recipients of the contract.

subcontract-list is the collection of contracts generated from the initial contract. The node in which the initial contract structure exists has the responsibility of monitoring the execution of these subcontracts (i.e., it is their manager).

There is no conceptual difference between a ***contract*** and a ***subcontract***, but we distinguish between them to simplify the discussion in this chapter. A contract holds the information required by the contractor for the associated task. A subcontract holds the information required by the manager for the associated task.

The structure of a subcontract is shown in Figure 6.3.

<subcontract> -> [name]
[contractor]
*<task> **
*<results> **
*[predecessors] **
*[successors] **

Figure 6.3 A Subcontract.

name is a unique identifier for the task to be executed.

contractor is the node that is executing the subcontract.

task is the complete specification of the task.

results are the outcome of executing the subcontract.

predecessors are the names of subcontracts whose tasks are preconditions for the task of this subcontract (after [Sacerdoti, 1975]).

successors are the names of subcontracts for which the task of this subcontract is a precondition (after [Sacerdoti, 1975]).

Only the ***name*** and ***contractor*** slots need be maintained by the manager in a basic implementation. Maintainance of the ***task*** and ***results*** slots, however, facilitates recovery if a contractor fails. The ***results*** structure is also useful for keeping track of which contractor obtained a particular result. ***Predecessors*** and ***successors*** allow handling of ordered subcontracts (e.g., successors of an *Ordered-AND* node in a search application).

6.3 Processing States

Contracts are processed according to the state transition diagram shown in Figure 6.4, which uses the terminology of operating systems (see, for example **[Brinch Hansen, 1973]**).

A contract that is awarded to a node enters the ***ready*** state, in which it waits until the task processor is available. At that time, the contract enters the ***executing*** state. If execution results in the generation of subcontracts, then these enter the ***announced*** state, shown by the double dashed line in the figure, in which they remain until they have been awarded. A subcontract may either be awarded to another node, in which case it passes out of this node (shown in the figure by a solid line), or it may be awarded to this node, in which case it passes to the ***ready*** state (shown by a dashed line).

If all possible subcontracts have been generated and execution cannot continue on the task associated with the contract until subcontract results have been received, then the contract enters the ***suspended*** state; and the node begins processing another contract in the ***ready*** state. Upon receipt of a report of a subcontract result, the contract is transferred to the ***ready*** state, where it awaits further processing.

When execution of the contract has been completed, it is transferred to the ***terminated*** state; a report is generated and sent to the report recipients for the contract, and the node begins processing another contract in the ***ready*** state.

If a contract in the ***executing*** state is transferred to either the ***suspended*** or ***terminated*** state and if no contracts are available for execution in the ***ready*** state, then the node attempts to acquire a new contract--either by making a bid on a recent task announcement or by transmitting a node availability announcement (see Section 6.5).

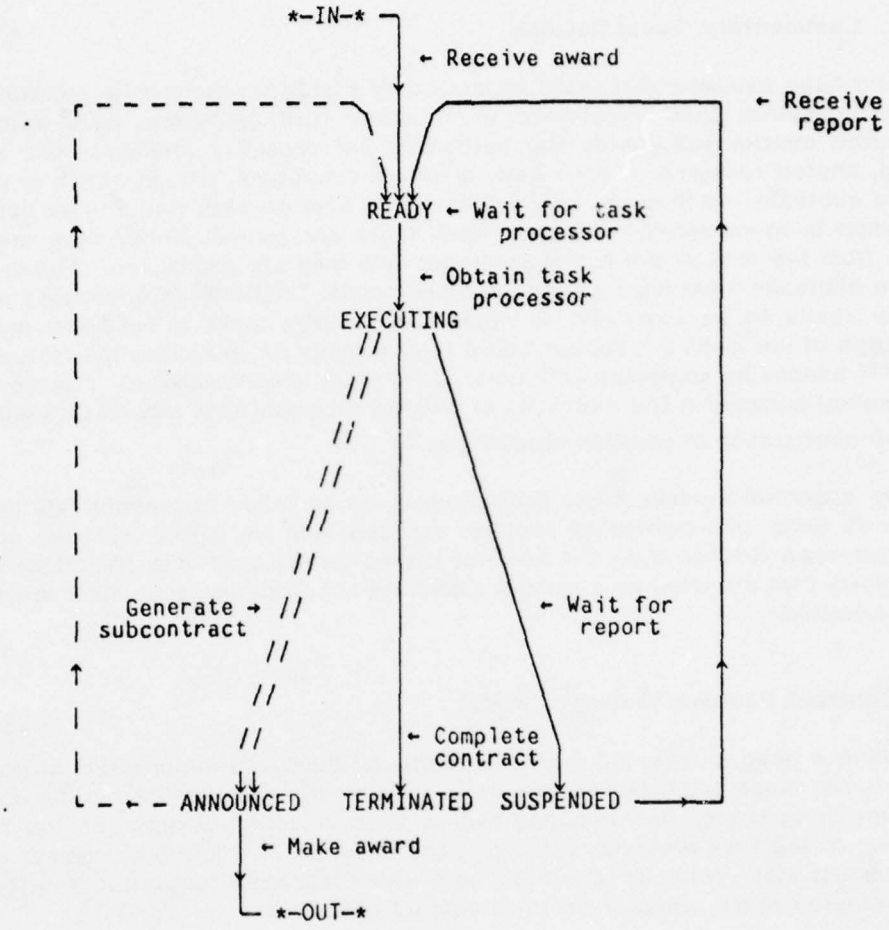


Figure 6.4. Contract-processing States.[1]

[1] A node requires a scheduler to allocate task processor time to respond to the events shown in the figure. The scheduler used in CNET (Appendix C) is non-preemptive and gives priority to a contract whose execution can continue due to reports, over a contract whose execution has not yet been started.

6.3.1 Commentary: Local Queuing

We have assumed that tasks generated by a node are queued for execution locally, at that node, rather than transferred to a central task repository. Local queuing fosters distributed control and avoids the bottleneck and reliability problems that accompany a central, shared resource. It does have a further advantage, though, which is evident when ordered subtasks are involved (i.e., subtasks that must be executed in a particular order, as is common in robot problem solving). When tasks are queued locally they are effectively hidden from the rest of the nodes in the net until they are announced. This is useful when ordered subtasks have been generated by one node. They are not announced until they are actually ready to be executed. In other words, hiding tasks in individual nodes has the advantage of not making it appear that a task is ready for execution by another node when in fact it cannot be executed until other tasks have been completed. If there indeed is no concurrency possible in the execution of ordered subtasks, then local task queuing does not give the appearance of possible concurrency.[1]

On a central agenda, extra precautions must be taken to prevent this situation. It is sometimes done with multi-level priorities (an approach we dislike because of its indirect character--see Section 5.3), but could of course be effected with (functionally) the same mechanisms that are used by a node in a contract net to hide a task until it is actually ready to be executed.

6.4 Contract Passage Through A Node

When a node is awarded a new contract, it initializes an information structure for the contract. The ***name*** and ***task*** slots are initialized from the specifications in the announcement and award messages; the ***manager*** slot is filled in with the name of the manager. As execution of the task proceeds (according to the states of Figure 6.4), results are compiled in the ***results*** slot. When execution of the contract has been completed, it is transferred to the ***terminated*** state, where it will eventually be deleted (see Section 6.7).

If a subtask is generated during the execution of the contract, a new subcontract structure is created for the subtask and the ***name*** and ***task*** slots are filled in. The subcontract is placed in the ***announced*** state with pointers in the ***subcontract-list*** slot of the contract that generated it. The contract processor forms and transmits a task announcement message for the subcontract (see Section 6.5), to try to find a node to execute it.

One other type of information is linked to a subcontract in the ***announced*** state: ***active-bids***. Active bids are the bids (together with the names of the bidders) that have been received for an announced subcontract that has not yet been awarded.

When a subcontract is awarded, the name of its contractor is noted in the ***contractor*** slot and it is removed from the ***announced*** state. The only record of the subcontract that is

---

[1] Note that local queuing does not offer any assistance if there are time dependencies between subtasks that have been generated by different nodes.

held by the manager at this point is the information in the ***subcontract-list*** slot of the contract that generated it.[1]

6.4.1 Commentary: Kernel-Size

Each node must have a mechanism for deciding when to attempt to find another node to execute a subtask, as opposed to executing the task itself. Even if a task requires no specialized expertise it may be distributed to another node to obtain a speedup (as in the N Queens problem). In this case the overhead involved in contract negotiation (which depends on the characteristics of the system (see Section 2.2.3)) establishes a minimum kernel-size for a task that can be usefully distributed. Tasks of kernel-size less than this minimum will be more effectively executed by the node that generates them; tasks of larger kernel-size can be usefully distributed to obtain a speedup.

In most cases the minimum kernel-size for distribution will be determined by the programmer (probably as a result of experimentation--as is presently done to determine a useful working-set size to prevent thrashing). We can also speculate, however, that a distributed problem solver could learn to make this type of decision itself, as a result of experience. For example, if the programmer sets an absolute minimum kernel-size for task distribution, then the problem solver could gather statistics both on the time required for the distribution and for the execution of particular types of task and determine suitable groupings of minimum kernel-size tasks that can be effectively distributed. For example, assume that some of the absolute minimum kernel-size task types are designated A, B, and C by the programmer. If the problem solver determines on the basis of experience that A, B, and C require more time for distribution than they require for execution, then it could form a new task D for distribution, where D = A∧B∧C. As long as D does not require significantly more communication for its distribution than A, B, and C, it will have the effect of reducing coupling.

6.5 Contract Net Protocol Specification

The need for each of the message types has been determined empirically. The collection presented here is adequate for the examples presented in the dissertation. Further message types may, however, be required for more complex behavior.

The messages of the protocol are shown below (Figure 6.5).[2] Options are separated by "|" or denoted as separate productions. Message types that need not be included in a basic implementation are followed by "*".

---

[1] Thus, for all contracts a node has links up one level to the manager that is responsible for monitoring its execution and links down one level to subcontractors that are executing subcontracts that it has generated.

[2] Slots that contain information encoded in the common internode language are enclosed in "{}". We have seen examples in Section 3.1.3.1 and Section 3.2.3.4.

```
<message> -> <header> <addressee> <originator> <text> <trailer>

<header> -> [line-control-header] [time]

<trailer> -> [error-control-trailer] [line-control-trailer]

<addressee> -> [contract-net-address] | [subnet-address] | [node-address]

<originator> -> [node-address]

<text> -> [communications-control-message] | <contract-message> |
          <request-message> | <information-message>

<contract-message> -> <task-announcement> | <bid> | <announced-award> |
                      <directed-award> | <acknowledgment> | <report> |
                      <termination> | <node-availability-announcement>

<task-announcement> -> TASK-ANNOUNCEMENT [name]
                       {eligibility-specification} {task-abstraction}
                       {bid-specification} [expiration-time]

<bid> -> BID [name] {node-abstraction}

<announced-award> -> ANNOUNCED-AWARD [name] {task-specification}

<directed-award> -> DIRECTED-AWARD [name] {eligibility-specification}      *
                    {task-abstraction} {task-specification}

<acknowledgment> -> ACCEPTANCE [name]                                      *
                 -> REFUSAL [name] {refusal-justification}

<report> -> INTERIM-REPORT [name] {result-description}
         -> FINAL-REPORT [name] {result-description}

<termination> -> TERMINATION [name]                                        *

<node-availability-announcement> -> NODE-AVAILABILITY-ANNOUNCEMENT         *
                                    {eligibility-specification}
                                    {node-abstraction} [expiration-time]

<request-message> -> REQUEST [name] {request-specification}                *

<information-message> -> INFORMATION [name] {information-specification}    *
```

Figure 6.5. Contract Net Protocol Specification.

6.5.1 Low-Level Message Structure

Low-level message components are shown for completeness. As noted previously, they are not our major concern and we therefore only briefly describe them. (See, for example [Carr, 1970], [Metcalfe, 1976], [Sunshine, 1976] for details.) *[line-control-header]* delimits the beginning of a message. *[time]* specifies the time at which the message was transmitted. *[error-control-trailer]* is used by an addressee of the message to determine that it has been correctly received (a cyclic redundancy check polynomial is commonly used for this purpose). *[line-control-trailer]* delimits the end of the message.

<addressee> can take three different forms: *[contract-net-address]* for general broadcast messages, *[subnet-address]* for limited broadcast messages, or *[node-address]* for point-to-point messages.

The *[communications-control-message]* is a low-level message designed to maintain communications subnet information, such as routing tables. Messages of the form *Hello*, and *I heard you*, as exchanged periodically by IMPs in the ARPAnet as status checks, fall into this category [Heart, 1972].

6.5.2 Contract Messages

The *<contract-message>* forms the core of the protocol. Each contract message includes the *[name]* of the contract to which it refers.

6.5.2.1 Task Announcement

The *<task-announcement>* is used to advertise the existence of tasks to be executed.

The *{eligibility-specification}* lists the criteria that a node must meet to be eligible to submit a bid. This specification reduces message traffic by pruning nodes whose bids would be clearly unacceptable.

The *{task-abstraction}* is a brief description of the task to be executed that allows a potential contractor to evaluate its level of interest in executing a task relative to others currently available. An abstraction is used rather than a complete specification in order to reduce message traffic.

The *{bid-specification}* details the expected form of a bid for the task. It enables a potential contractor to submit a bid that contains only the brief specification of its capabilities relevant to the task, rather than a complete description. This both simplifies the task of the manager in evaluating bids and further reduces message traffic.

The *[expiration-time]* specifies the time interval during which bids will be accepted

for the task.[1] The task will be awarded to the most acceptable bidder at the end of this time interval, or it will be re-announced if no suitable bids have been received (we will later see special cases).

6.5.2.2 Bid

The ***<bid>*** is a response to a task announcement. It contains a ***{node-abstraction}***, which is a brief specification of the capabilities of the node that are relevant to the announced task (see above). It is written in the form specified by the bid specification of the corresponding task announcement.

6.5.2.3 Award

There are two varieties of contract award: the ***<announced-award>*** and the ***<directed-award>***. The former is used to award a contract to a node after an announcement-bid sequence of contract negotiation; the latter is used to award a contract directly to a node without negotiation. Each type of award contains a ***{task-specification}***; that is, a complete description of the task to be executed.

6.5.2.4 Acknowledgment

The ***<acknowledgment>*** is used to acknowledge acceptance or rejection of a directed award. An ***acceptance*** simply names the contract. A ***refusal***, on the other hand, contains a ***{refusal-justification}***, which details the reasons for the refusal.

Where message-passing systems require acknowledgment of messages at the distributed-architecture level, there is a further requirement in the case of the directed award at the problem-solving level: A node must be able to refuse or give a negative acknowledgment to a directed award if a *master-slave* control regime is to be avoided. That is, there must be the capacity for the negative acknowledgment at the problem-solving level of the form *I can't execute this contract because ...*, as opposed to the negative acknowledgment at the distributed architecture level which is of the form *I didn't receive your message correctly*.

6.5.2.5 Report

The ***<report>*** is used by a contractor to inform the manager (and other report recipients, if any) that a task has been partially executed (an ***interim-report***) or completed (a ***final-report***). The report contains a ***{result-description}*** that specifies the results of the execution.

---

[1] Note that global synchronization is assumed--but only to an accuracy commensurate with the time intervals specified in such messages.

6.5.2.6 Termination

The *<termination>* is used by a manager to indicate to a contractor that execution of a task is no longer required. The effect on the contractor is analogous to that of an interrupt: Execution of the task is terminated.

6.5.2.7 Node Availability Announcement

The *<node-availability-announcement>* is used to advertise the availability of a node for task execution. A node that transmits such a message is idle and searching for a task to execute. Interested nodes with suitable tasks do not submit bids for such an idle node; they transmit task announcements or directed awards to it. The message has slots that are analogous to those of the task announcement.

In this case, the *{eligibility-specification}* lists the criteria that the node will look for in a task announcement or award. The *{node-abstraction}* is a brief specification of the capabilities of the node that are of interest to prospective managers, and the *[expiration-time]* specifies the time interval during which the announcement is valid. The node availability announcement, like the task announcement, will be re-announced at the end of the expiration time if a suitable contract has not been acquired.

A node can thus acquire a contract in one of two ways: It can wait for a suitable task announcement and submit a bid, or it can advertise its availability with a node availability announcement. The decision as to which method to use is net-load dependent. If the net is not heavily loaded, for example, then the use of the task announcement is warranted in all cases, since the availability of a task to be executed is the event of primary importance. If, on the other hand, the net is heavily loaded (i.e., most nodes have a number of tasks queued for execution) then no bids will be received on most task announcements. Unlimited use of these announcements would serve only to saturate the available communications channels. In this case, the availability of an idle node is the event of primary importance and the use of node availability announcements is preferred.

The decision to advertise availability or wait for a task announcement can be made by an individual node, based on its own experience with task announcements. It can do this by keeping track of the number of re-announcements that it has made for tasks that it has tried to distribute (over a time window in the recent past). If several re-announcements are the norm, then it can stop making task announcements, and switch to the node availability announcement when it goes idle. The process can then be restarted; that is, after the node has acquired a new contract, it can switch back to task announcement mode for new subcontracts until it again becomes evident that the net is saturated.[1]

---

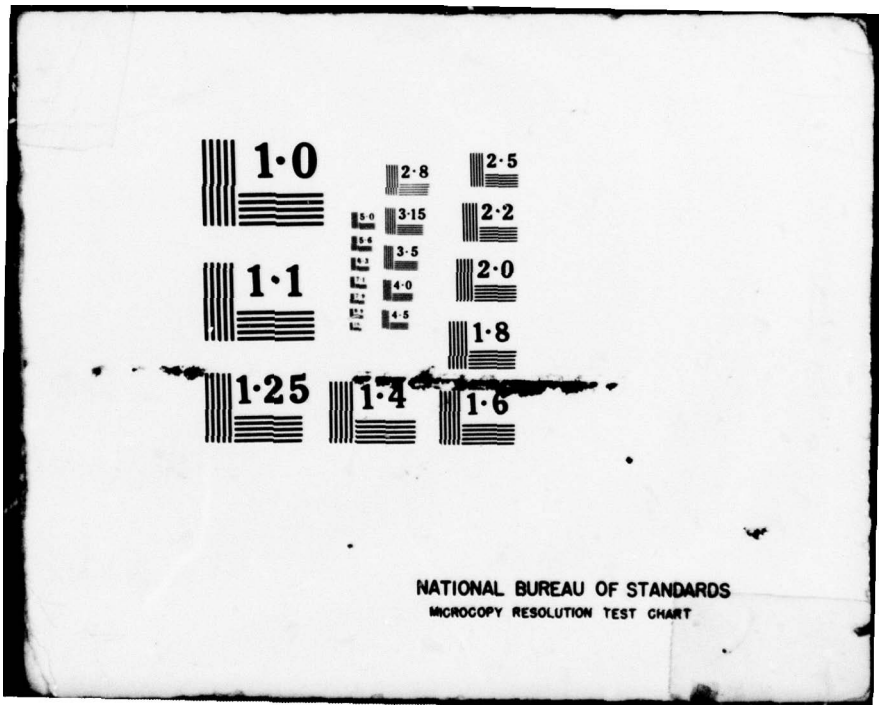
[1] Possible deadlocks caused by different views of net-loading by different nodes are solved with timeouts. Nodes that are idle and estimate that the net is lightly loaded will await task announcements. They may not receive any because other nodes with tasks to be executed estimate the net to be heavily loaded, and hence do not announce their tasks. After a time, however, both sets of nodes will adjust their estimates and remove the deadlock.

AD-A068 230 STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE F/G 9/2
A FRAMEWORK FOR PROBLEM SOLVING IN A DISTRIBUTED PROCESSING ENV--ETC(U)
DEC 78 R G SMITH MDA903-77-C-0322
UNCLASSIFIED STAN-CS-78-700 NL

2 OF 2
ADA
068230



END
DATE
FILMED
6-79
DDC


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

6.5.3 Request Messages

The *<request-message>* is used to communicate simple requests between nodes. It contains the *[name]* of the contract that gave rise to the message and a *{request-specification}*, which details the request. A request might be, for example, *Send me the code for procedure x.*

6.5.4 Information Messages

The *<information-message>* is used both as a response to a request message and as a general data transfer message. It contains an *{information-specification}*, which details the information of interest.

When the information message is a response to a request message, it includes the name of the contract that gave rise to the request. On the other hand, when it is used as a vehicle for communicating unsolicited results or other data, then it includes the name of the contract that gave rise to the information message itself. We will see the implications of this in Section 6.7.9.

6.5.5 Commentary: Action vs. Information

We can distinguish two classes of messages in the contract net protocol: the action message and the information message. This classification is at once an indication of the content of a message and the expectations of the addressee held by the originator. The classification is not necessary in the protocol but provides some useful insight into the styles of problem solving that can be achieved within the contract net framework.

An action message is directed from one node to another for action. It therefore carries an implied obligation or responsibility on the part of the addressee; that is, the originator expects the addressee to respond to the message through some ensuing action. Contract messages and request messages are action messages. They are used to cause events to happen in the net (e.g., the execution of tasks). In this sense, they act as interrupts (we noted, for example, that the termination message has this flavor).

An action message may indicate to the addressee that the originator expects a return message of some sort and that it will itself be forced to take further action if such a message does not appear. The task announcement, for example, will be re-issued if no bids have been received before the expiration time has passed. Action messages thus imply a *two-way* communication.

An information message, on the other hand, is directed from one node to another to convey information only. It carries no obligation on the part of the addressee. It is used both as a response to a request message and as a means of communicating unsolicited results or data throughout the net (where the utility of such information is judged by the individual nodes that receive such messages). The originator does not expect a message in return.

There is thus no implied two-way communication for such messages.[1]

Action messages are used to effect hierarchical control in the contract net framework. Information messages, on the other hand, are used to effect a style of control that has been called *cooperative distributed processing* [Lesser, 1978]. We noted the difference between the two styles of control for the DSS (Section 3.2) in the situation where a vehicle is passing out of one area and into a neighboring area.

6.6 Common Internode Language Specification

We now describe the common internode language that has been implemented in CNET (see Appendix C). The design is rudimentary but adequate for the examples presented in Chapter 3. Samples of its use were presented in Section 3.1.3.1 and Section 3.2.3.4. Many designs are possible. In general, a user will want to specify a language to meet the needs of his own application. See Section 4.1.3 for a discussion of why a language is preferred over a less-structured means of encoding task-dependent information.

CNET uses a *semantic-grammar* parser (similar to that of [Bonnet, 1978]). Several grammars are used, depending on the purpose of the parse. For example, eligibility specifications are encoded according to the grammar shown in Figure 6.6. Optional symbols are enclosed by "()". Nonterminal symbols that expand to words are enclosed by "[]", and other nonterminals are enclosed by "<>". Values are quoted as shown.[2] The grammar accepts sentences that specify an object by its attributes and values. When the value refers to another object (found in the slot specified by the preceding attribute), then successive attribute-value specifications refer to that object. Sample sentences are *MUST-HAVE NODE NAME 'SELF POSITION area 'A*, and *PROCEDURE NAME 'extend-board* (where words in the core language are shown in upper case, and domain-specific words are shown in lower case).

<eligibility-specification> -> (MUST-HAVE) [object] <att>

<att> -> [attribute] ([attribute]) '[value] (<att>)

Figure 6.6. Eligibility Specification Grammar.

---

[1] A similar distinction has proved useful in military message-passing systems, although the designations *action* and *information* refer only to the addressee; the *action-addressee* is the addressee required by the originator to take the necessary action, and the *information-addressee* is the addressee considered by the originator to require simply notification of the contents of the message.

[2] The quotes were not shown in earlier sections for simplicity.

OBJECTS

CONTRACT
DEVICE
NODE
POSITION
PROCEDURE
TASK
TASK-TEMPLATE

ATTRIBUTES

ANNOUNCEMENT-PROCEDURE	ANNOUNCEMENT-RANKING-PROCEDURE
AWARD-PROCEDURE	BID-CONSTRUCTION-PROCEDURE
BID-RANKING-PROCEDURE	CODE
EXECUTION-PROCEDURE	INFORMATION-ACCEPTANCE-PROCEDURE
LATITUDE	LONGITUDE
MANAGER	NAME
NUMBER	POSITION
PREDECESSOR	REFUSAL-PROCEDURE
REFUSAL-PROCESSING-PROCEDURE	RELATED-CONTRACTOR
REPORT-ACCEPTANCE-PROCEDURE	REPORT-RECIPIENT
RESULT	SUBCONTRACT
SUCCESSOR	SPECIFICATION
TYPE	

Figure 6.7. Common Internode Language - Core Objects and Attributes.

The objects and attributes in the core vocabulary (Figure 6.7) are domain-independent and known a priori to all nodes. The language is extensible in the sense that new objects and attributes can be added for particular domains (e.g., *signal* and *fundamental* in the DSS). Some of the attributes are related to the processing of tasks (e.g., EXECUTION-PROCEDURE). These are discussed in Section 6.7 and Section 6.8.

6.7 Message Processing Procedures

In this section we describe in general the procedures that respond to messages in the contract net protocol. We ignore the details of their implementation in CNET. The required *user-procedures* are also discussed, again ignoring the specifics of their interaction with CNET (e.g., the exact form of the values they return).

6.7.1 Task Announcement Processing

A node that receives a task announcement examines the eligibility specification to determine its relevance to the announced task. This decision is based on a search of the local knowledge base of the node to determine if it has the required hardware characteristics, procedures, and data.[1]

If the node determines that it is relevant to a task, then it must establish its level of interest in the task, relative to others that have been announced. Eventually, the node must select a task for which to submit a bid. It is difficult to establish a total order over announced tasks because this would necessitate the use of a simplistic global priority measure to compare tasks of different type. Such measures typically perform too much data compression to be useful in *AI* problem solvers (see, for example **[Minsky, 1972]** and **[Berliner, 1973]**). Hence, the best a node can usually do is establish a partial order over announced tasks. A node retains a list (called ***active-task-announcements***) containing the best task announcements of each type that it has received, whose expiration times have not passed; that is, each announcement in the list corresponds to a different task *type*, and it is the best that has been received for that type. (In a more complex implementation, a node could retain all task announcements that are still valid; but this has not proved necessary for the examples in this dissertation.)

The information used by a node to determine the relative ranking of tasks is contained in the task abstraction. The node calls a task-dependent announcement ranking procedure, to compare the task abstraction of the new announcement with the task abstraction of the best task announcement of the same type received thus far.[2] The information returned by the ranking procedure is used to decide whether to replace the current best task announcement of the given type with the new task announcement or to discard the new announcement.

This announcement-ranking activity proceeds concurrently with task processing until the node goes idle, that is, completes the execution of tasks in the ***executing*** and ***ready*** states. At this point, the node checks its current list of task announcements and selects a task on which to submit a bid. If there is only one type of task (i.e., the list has a single member), the procedure is straightforward. If, on the other hand, there are a number of task types available, the node must select one of them. The current version of CNET selects the oldest task on the list. More complex decision procedures could, of course, be implemented.

The node then calls a task-dependent bid construction procedure. This procedure uses the bid specification of the task announcement to form a bid. The procedure returns a node abstraction (i.e., an abstraction of the capabilities of a node that are relevant to the

---

[1] It is assumed that such searches are short enough to be processed by the contract processor of the node in an interrupt mode, as a task announcement is received (i.e., they do not require use of the task processor).

[2] All task-dependent procedures must be supplied by the user. They are called by the contract processor to perform specific functions. The complete set of procedures is summarized in Section 6.8. Many of the procedures have default definitions in CNET so that a user need not specify all of them in every case.

execution of the announced task). The node then stores a temporary structure (called a ***pseudo-contract***) in its local knowledge base. This structure contains enough information so that processing can be started when an award is received.[1] Finally, a bid is transmitted to the manager.

We have thus far discussed the straightforward receipt of a task announcement, the selection of a task, and the submission of a bid. There is, however, one further situation with respect to this process that must be addressed. We have described the process under the assumption that a node not consider submission of a bid until it goes idle. This strategy can, however, lead to difficulty. For instance, a node that issues a task announcement may not receive any bids for one of two reasons: *(i)* there are no available nodes, as may occur for example in the N Queens problem; or, *(ii)* no node has the necessary data to execute the task, as may occur in the DSS application if the eligibility specification is too stringent. In the first case, the task announcement may be usefully reissued until a bid is obtained from an idle node. This approach is, however, inappropriate in the second case; therefore a node requires a way of determining which of the two cases caused the no-bids situation.

This is handled in the contract net by allowing a node to note in the bid specification of a task announcement that it wants bids of the form *I am eligible to execute this task, but cannot do so because I am busy*. A node receiving such a task announcement cannot deal with it in the usual way (i.e., by waiting until it is idle to submit a bid), but must instead respond immediately (if eligible) with a bid of the above form.

6.7.2 Bid Processing

When a bid is received by a node, it calls a task-dependent bid ranking procedure to compare the new bid with the list of bids thus far received for the task (the ***active-bids*** discussed earlier). This procedure must indicate that the new bid is *satisfactory* (i.e., warrants an immediate award), or sort the new bid into the list of active bids. If the *satisfactory* value is returned, then the contract is awarded immediately to the node that made the new bid. Otherwise, the node waits until the expiration time has passed and calls a task-dependent award procedure to determine the appropriate action to be taken.

In the simplest case, the node awards the contract to the node that made the best bid. As described above, however, there are other possibilities to be considered. If no bids have been received by the end of the expiration time, then the normal procedure is to reissue the task announcement. The award procedure, however, may have the node alter the task announcement before reissuing it (to loosen up the eligibility specification for example). In general, any number of different actions are possible, including the award of the contract to multiple bidders.

---

[1] This structure is deleted by the node if it does not receive an award for the contract.

6.7.3 Award Processing

An announced award is relatively straightforward, except in the case of multiple awards, where a node has made bids on several contracts and has been awarded more than one. In this case, the node initializes the necessary contract structures and moves them all to the ***ready*** state, in order of receipt.

A directed award is somewhat more complex, since it must be acknowledged. The node first checks the eligibility specification of the award. If it is able to execute the contract, then the node places the contract in the ***ready*** state and transmits an acceptance to the manager. If not, it calls a task-dependent refusal procedure to generate a refusal justification and then transmits a refusal to the manager.[1]

We have assumed that the scheduling of tasks is non-preemptive in the task processor of a node; that is, a low-priority task that is currently executing cannot be preempted by a high-priority task that is awarded at a later time. This strategy is adequate for the examples presented in the dissertation but may require alteration for other applications, notably those involving time deadlines (see, for example [Bowdon, 1972]).

6.7.4 Acknowledgment Processing

Receipt of an affirmative acknowledgment results in little processing, since it simply confirms that the desired node has agreed to commence processing. On receipt of a negative acknowledgment, on the other hand, the manager must call a task-dependent refusal processing procedure to examine the refusal justification and take an appropriate course of action. This action might be to wait and try again, or to issue a task announcement rather than to try and award the task directly, or to attempt a directed award to another node.

6.7.5 Report Processing

When a manager receives a report of a subcontract result, it copies the results into the ***results*** slot of the subcontract from which they were obtained (bound to the name of the contractor that generated them, in the case of multiple contractors) and calls a task-dependent report acceptance procedure to integrate the results in the ***results*** slot of the contract that generated that subcontract. It then moves the contract from the ***suspended*** state to the ***ready*** state in preparation for continuation of processing (unless the contract is already in the ***ready*** or ***executing*** state).

If the report is a final report, the subcontract is removed from the ***subcontract-list*** slot of the contract and from the ***predecessors*** and ***successors*** slots of other subcontracts.

---

[1] A manager that makes a directed award notes that the contractor is only tentatively assigned until it receives an acceptance.

6.7.6 Termination Processing

When a node receives a termination message, it stops processing on the named contract and moves the contract to the ***terminated*** state. It then terminates all outstanding subcontracts of the contract (using messages if necessary) and removes them from the ***successors*** slots of other subcontracts.

In the current implementation, a specified number of contracts are retained in the ***terminated*** state. After this state is full, the oldest contracts are discarded, so as not to saturate the storage capabilities of a node. In a more complex implementation, decisions might be made about integrating selected information from terminated contracts, about to be discarded, into the local knowledge base of a node. We have not addressed this issue.

6.7.7 Node Availability Announcement Processing

When a node receives a node availability announcement, it examines the eligibility specification and tries to match the criteria for a suitable task against the task abstractions of the tasks in its own ***announced*** state. For a task whose task abstraction matches the criteria, the node tries to match the eligibility specification of the task against the node abstraction of the node availability announcement. If the node that issued the node availability announcement is suitable for the task, then a directed award is transmitted to it.

6.7.8 Request Processing

The request message is intended to allow one node to obtain information from another without the overhead of contract negotiation. Such messages are designed for requests that require a trivial amount of computation and their contents are processed in an interrupt mode by the contract processor of a node. Requests that require a nontrivial amount of computation should be packaged as standard contracts.

The contract processor searches its local knowledge base for the desired information, packages it into an information specification, and transmits an information message to the requesting node.

6.7.9 Information Message Processing

Upon receipt of an information message, a node calls a task-dependent information acceptance procedure associated with the contract named in the message. This procedure copies the information specification of the message into the local knowledge base as appropriate. If the contract named in the message does not belong to the node (as will happen when the information message is being used for transmission of unsolicited data), then the node uses a default procedure to process the message (i.e., store the information in the local knowledge base). More complex procedures are of course possible, but these have not been implemented.

6.8 Task Structure

A task structure is used to maintain information specific to the processing of a particular task. Each type of task has an associated ***task-template*** with slots corresponding to the roles that task-dependent procedures play in a contract net. These roles are summarized in Figure 6.8. The *execution-procedure*, for example, actually processes the task. The *report-acceptance-procedure* integrates the results of a report message into the ***results*** slot of the contract for which they are intended. The task-template slots are filled in with the names of the procedures that actually fill these roles for tasks of the associated type.

The task structure itself has all of the *role* slots and values of its associated template plus a ***specification*** slot that contains the actual description of the task (e.g., a partial board for the N Queens problem).

announcement-ranking-procedure
bid-construction-procedure
bid-ranking-procedure
award-procedure
refusal-procedure
refusal-processing-procedure
report-acceptance-procedure
termination-procedure
information-acceptance-procedure
execution-procedure

Figure 6.8. Task-Dependent Procedures.

Chapter 7

Summary And Conclusions

The movement of the progressive societies has hitherto been a movement from status to contract.

- Sir Henry James Sumner Maine,
Ancient Law, 3rd American Ed., 1873, p. 165.

7.1 Distributed Problem Solving

Distributed Problem Solving is the cooperative solution of problems by a decentralized collection of loosely coupled knowledge-sources or KSs. *Decentralized* means that both control and data are logically, and sometimes geographically, distributed--there is neither global control nor global data storage. The KSs must share tasks and results--*cooperate* to solve problems. *Loosely coupled* means that individual KSs spend the great percentage of their time in *computation* as opposed to *communication*.

We have emphasized distributed *problem solving*, as opposed to distributed *processing*, because of our concern with the higher level problem-solving activity, as opposed to the lower level architectural and systems concerns. We have not considered the design of individual processor nodes, nor their low-level interconnection. We have assumed the existence of an underlying architecture and communications system; that is, a collection of nodes capable of executing tasks, together with a mechanism for reliable transfer of bit streams between arbitrary nodes. In the terms of Section 1.2 we have focused our efforts at the problem-solving level, keeping in mind the constraints imposed both by decisions made at the distributed-architecture level and the mechanisms required at the systems level to make the distributed problem solver work. The discussion has therefore been phrased in terms of the familiar problem-solving vocabulary: *control* and *knowledge organization*.

The distributed context has, however, forced us to pay careful attention to *communication*--generally not a problem-solving concern. This focus has been necessary both because of the practical importance of loose-coupling and because there is neither global control nor global data storage. Communication is thus the only way to coordinate the actions of individual nodes and to integrate the results that they achieve during the execution of tasks. Communication--often considered to be a lower level systems concern--is therefore usefully considered at the problem-solving level.

PRECEDING PAGE BLANK-NOT FILMED

7.2 The Contract Net Framework

The contract net framework is a set of design specifications for distributed problem solvers. It addresses three major issues: communications, control, and knowledge organization. The main emphasis is on communications and control, and the main result is a high-level *problem-solving protocol*. Only those aspects of knowledge organization that are particular to the distributed context and support of the problem-solving protocol are considered in detail.

7.2.1 Summary

All interactions between nodes in a contract net are governed by messages of the contract net protocol. This is a problem-solving protocol for communication of both task-independent and task-dependent information.

Task-independent information is essentially control information and is encoded directly in the protocol. It specifies what is to be done with the task-dependent information in the messages. The task-independent information in a message may specify, for example, that the information in the task-dependent slots describes a task to be executed (called a ***task-announcement***), or the results of the execution of a task (called a ***report***). The messages have slots for task-dependent information necessary for carrying out the intended function of the message. The role of the information in each slot is defined by the protocol--e.g., a task announcement has slots for task-dependent information that enables a node receiving the message to decide whether or not it is able to execute the task (called an ***eligibility specification***), to rank the announced task relative to other announced tasks (called a ***task abstraction***), to construct an appropriate bid on the task (called a ***bid specification***), and to know the time period during which the announcement is valid (called an ***expiration time***).

The task-dependent information in the slots is encoded in a common internode language. This language simplifies the entry of a new node into the net because: (*i*) A node can isolate the information it needs to begin to participate in the actions of the distributed problem solver, and (*ii*) a node can express a request for the transfer of required task-dependent information. This type of transfer is called dynamic distribution of knowledge.

The contract net protocol helps to reduce message traffic and message processing overhead--through the use of eligibility specifications, task abstractions, and bid specifications in task announcements; focused addressing; and specialized interactions like directed contracts and requests.

A contract negotiation process is used to solve the *connection problem*--the problem of dynamically connecting nodes that have tasks to be executed with nodes that are capable of executing those tasks. This approach is especially appropriate for a distributed problem solver in that it requires neither global control nor global data storage. System concurrency is also enhanced because both managers and contractors simultaneously seek each other out, finally achieving connections by mutual selection.

Contract negotiation enables a range of invocation styles. The full announcement-bid-

award sequence enables nondeterministic invocation. The task announcement broadcast is a distributed analog of the PLANNER goal specification. The independence and anonymity of KSs are maintained because individual nodes are not addressed by name to execute a task but, rather, are addressed by capability. Focused addressing is less nondeterministic because it permits a node to use more specific knowledge about what other nodes can assist task execution. This type of addressing is analogous to the THUSE construct in PLANNER. Finally, directed contracts allow more efficient operation when nondeterminism is not required and the required KS-invocation sequence for a problem is well-understood.

Each node in a contract net thus takes on one of two roles related to the execution of an individual task: manager or contractor. The manager is responsible for monitoring the execution of a task and processing the results of its execution. The contractor is responsible for the actual execution of the task. The contract links between nodes assist in coordination of KSs (i.e., a manager can coordinate the actions of contractors working on related subtasks). They also permit restart at any level but the top level in the case of node failure. Hence, shared responsibility for tasks enables graceful degradation of performance at the problem-solving level.

7.2.2 Suitable Applications

The framework is particularly well-matched to problems that use a hierarchy of tasks and levels of data abstraction. Any heuristic search problem is an example of the former, and applications that deal with sensed data (e.g., audio or video signals) are examples of the latter. The manager-contractor structure provides a natural way to effect hierarchical control (in the distributed case, it's actually concurrent hierarchical control), and the managers at each level in the hierarchy are an appropriate place for data integration and abstraction. It should also be noted that the control hierarchies in the contract net framework are not simple vertical hierarchies but are the more complex generalized hierarchies discussed by [Simon, 1969]. The manager-contractor links are not the only means of communication. Nodes are able to communicate horizontally with related-contractors or with any other nodes in the net, as we saw in the DSS example--where classification contractors were able to communicate directly with signal contractors.

The announcement-bid-award sequence of contract negotiation enables more information and more complex information to be transferred in both directions (between caller and respondent) before KS-invocation occurs. The computation devoted to the selection process, based on the information transfer noted above, is more extensive and more complex than that used in traditional approaches, and is *local* in the sense that selection is associated with and specific to an individual KS (rather than embodied in, say, a global evaluation function). As a result, the framework is most useful when the specific KS to be invoked at any time is not known a priori and when specific expertise is required (e.g., the signal task of the DSS).

It also follows that the framework is primarily applicable to domains where the subtasks are large (in the loose-coupling sense defined in Chapter 2) and where it is worthwhile to expend a potentially nontrivial amount of computation and communication to invoke the best KSs for each subtask, so as to maintain the focus of the problem solver.

7.2.3 Programming In The Framework

A contract net system like CNET supplies the necessary task-independent procedures to use the framework, and the applications programmer must supply the task-dependent procedures. Because of the distributed nature of the problem solver, it is convenient to write these procedures in a *message-driven* style;[1] that is, sensitive to the receipt of messages of the contract net protocol. Hence, a task-dependent procedure is needed to handle the receipt of each of the messages of the protocol that is actually used in a particular application. (In Chapter 6 we discussed the interface between the task-independent message-processing procedures and the user-procedures.) This might sound like a large number of procedures, and it can be. Two points should be noted, however. First, the task-dependent procedures are essentially the same procedures that must be written for the uniprocessor problem solver. They have simply been made explicit in the contract net framework. In any problem solver, procedures must exist to select appropriate operators, deal with the results of operator application, backtracking, integration of the results from multiple operators, and so on. The contract net framework enforces a particular structure on the code that must be written--a structure that is appropriate for the distributed processing environment. Second, as noted in Section 6.7, default procedures can be used in many cases.

7.2.4 Limitations And Caveats

There are of course a number of limitations and caveats to consider. First, much of what we have proposed is a framework for problem solving that provides some ideas about what information is useful and how that information can be organized. There is still a considerable problem involved in instantiating the framework in the context of a specific task domain. Beyond the general guidelines offered earlier, it is not obvious, for instance, exactly what information should be in a task abstraction, bid, or task evaluation procedure. Yet the successful application of the machinery described above depends strongly on the choices made. In this sense, several of the mechanisms we have proposed are similar in spirit to the concept of the recommendation list in PLANNER: The mechanism provides a site for embedding particular types of task-dependent information (e.g. an eligibility specification), but does not specify, for a particular problem, the content, nor how to instantiate it in a particular domain. The utility of such mechanisms lies in their ability to help a user structure and understand a problem: We tread the traditional thin line between too much generality that provides too little guidance, and too much structure that overly constrains the user's options.

We have also not dealt with problems arising from partial information and conflicting views of the problem that may be held by individual nodes in the distributed problem solver (see, for example **[Lesser, 1978]** for a preliminary discussion). We have taken the point of view that problems of communications, control, and knowledge organization need to be attacked first in order to establish a suitable base from which to attack these problems.

---

[1] This style of programming has been suggested by several authors, including **[Hewitt, 1977a]** and **[Feldman, 1977]**.

An important caveat in considering use of the contract net framework, which has been touched on earlier, is that the kernel-sizes of the tasks must be large enough to justify the effort expended in distributing them. It is apparent, for instance, that the communication involved in task announcements, bids, awards, etc., and the computation involved in the deliberation phase (the task and bid evaluations) may add up to a substantial amount of overhead. It would make little sense to go through an extended mutual selection process to get some simple arithmetic done or to do a simple database access. While we discussed earlier how the full protocol can be abbreviated to an appropriately terse degree of interchange (e.g., directed contacts and the request-response mechanism), many other systems are already capable of supporting this variety of behavior. The interesting contribution of our framework lies in applications to problems where the more complex interchange provides an efficient and effective basis for problem solving.

7.2.5 Future Perspective

The contract net framework must be tested in a wide variety of task domains before its ultimate utility can be assessed. Analyses of message traffic and processing time should give indications as to the utility of abstractions in messages, the utility of focused addressing, and the utility of directed contracts. It is also necessary to study the difficulties involved in writing programs that operate in the distributed environment. Better guidelines must be developed for the selection of suitable task domains and the selection of suitable task partitionings. We intend to carry out these experiments using CNET (see Appendix C) as a testbed for our ideas.

7.3 What Have We Learned About Problem Solving?

We originally set out to develop a framework for problem solving in a distributed processing environment. To this end we developed a task-independent problem-solving protocol (the contract net protocol), a task-dependent common internode language, control based on contract negotiation as a way of handling task distribution, and an associated knowledge organization.

We can now ask what aspects of the resultant framework may also be useful for problem solving in general. One aspect is mutual selection. The problem of task distribution can be phrased as one of connecting tasks to be executed with KSs appropriate for their execution. In most current *AI* problem solvers (and in most of computer science, for that matter), the connection is effected with a limited, one-way transfer of information (e.g., a pattern transferred from the caller to the respondent). In the contract net view, by contrast, the transfer is two-way, and the information is not so strictly limited (e.g., to a name or a pattern). In addition, information about the complete collection of candidate KSs is available before final selection is made. The result is that tasks and KSs can be selected with more care than has previously been possible.

A second aspect is simultaneous use of task-centered and KS-centered knowledge. In a sense this is *meta-knowledge*, closely related to the meta-rules of [Davis, 1976]. The perspective here is somewhat broader, however. Meta-rules are expressions of what we

called task-centered knowledge--knowledge about which KS to invoke when a variety of KSs is available. In the contract net view, we have made use of a complementary kind of knowledge--KS-centered knowledge, or knowledge about which task to execute when a variety of tasks is available. The contract net appears to be the first use of both kinds of knowledge at the same time.

A central observation of this work is the importance of the problem of connecting tasks to be executed with KSs suitable for their execution. The key concept is *connection*. In the traditional view, this problem has generally been seen as a one-way connection. Either the control resides mainly with the tasks (through a central task generator and controller) or it resides mainly with the KSs. The key point here is that the control can be shared by both; that is, the problem becomes one of mutual or two-way selection, not one-way selection.

Appendix A

Distributed Search Analysis

Measures of the speedup that can be expected from the application of a distributed processor[1] to search problems that involve regular trees are presented, the effect of coupling on speedup is considered, and bounds on the number of processors that are required are discussed.

The results in this appendix have been verified by simulation using CNET (described in Appendix C).

A.1 Searching Regular Trees: Speedup

In order to obtain simple measures for speedup, consider the search of a regular tree, as shown in Figure A.1.

The total number of nodes, n, in a tree structure with depth d and branching factor b is

$$n = (b^{d+1} - 1)/(b - 1) \qquad d > 0, \text{ else } n = 1. \tag{1}$$

The number of tip nodes, n_t, in such a tree is

$$n_t = b^d \ . \tag{2}$$

Other expressions of interest for regular trees are the following (following [Simon, 1976]). First, for $b >> d$, n approaches n_t from above--almost all nodes are tip nodes in trees with large branching factors. Second, in trees with only one tip node that corresponds to a goal node, an average of $n/2$ nodes will be visited before the goal node is found, in the absence of other information. This is achieved with a depth-first search to depth d. A breadth-first search will visit an average of $n - b^d/2$ nodes before reaching the goal node.

The time required for the search is the most appropriate measure of performance for a multiprocessor. The traditional uniprocessor measure of number of nodes examined is still a valid measure of the power of the search strategy, but is insufficient to capture the effect of multiple processors.

---

[1] Hereafter, the distributed processor will be referred to as a multiprocessor, for the sake of brevity.

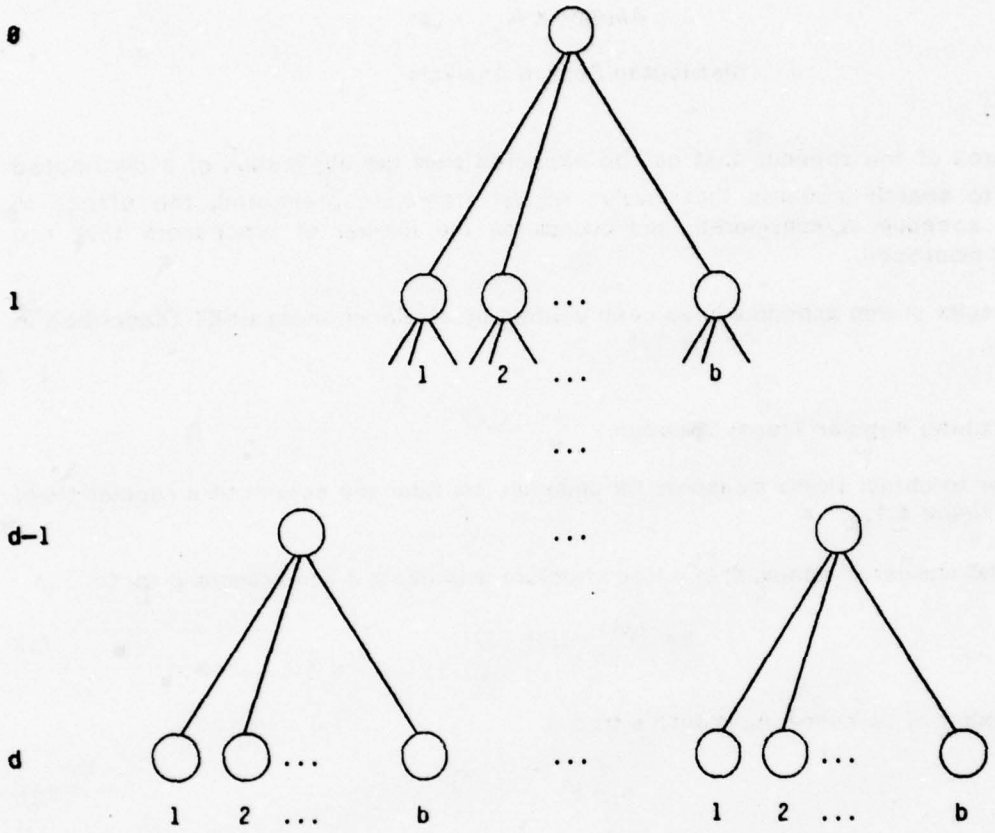


Figure A.1. A Regular Tree

We make the following simplifying assumptions in the analysis:

1) The basic task for an individual processor is the generation of a successor (of a node) in the search tree.

2) Any task requires the same processing time.

3) Enough processors exist so that a task can be commenced by one processor as soon as it has been generated by another.[1]

4) The distribution of tasks to processors and the reports of results take a negligible amount of time compared to the processing time for tasks.

5) Distribution of a task to another processor can be carried out concurrently

[1] This is, of course, unrealistic, in any practical case, but the intent here is simply to bound the search time, and the assumption is a useful simplification device.

with execution of another task (i.e., a processor can distribute one successor node to another processor while it is generating a second successor node).

6) The search terminates when a single goal node has been reached.

A.1.1 Uniprocessor Search

The search time divides into two components: the time to expand a node, t_e, (i.e., the time required to generate all successor nodes of a node), and the time to select a new node for expansion, t_s.

The time to expand a node can be rewritten in terms of the time required to generate a single successor node, t_g, as follows,

$$t_e = b \cdot t_g \ . \tag{3}$$

A.1.1.1 Minimum Time

The minimum time to find one goal node in regular tree will occur if the tree is being searched in a depth-first fashion and the search strategy is so good that no false paths are explored. Under this assumption, the search time, t^u_{min}, is

$$t^u_{min} = ((d-1) \cdot b + 1) \cdot t_g + (d-1) \cdot t_s \ . \tag{4}$$

Further assumptions here are that a node is completely expanded (i.e., all successor nodes are generated) before a new node is selected for expansion (as described by [Nilsson, 1971]) and that the goal node can be recognized as soon as it is generated.

One could consider search strategies where only some of the successors of a node are generated before a new node is selected for expansion (see, for example [Kowalski, 1970]). In the limiting case, the time could be computed that would be required to reach the goal node using a search strategy where a successor node is itself expanded immediately upon generation. This is a *pure* depth-first strategy and there is no time required for selection. While it is overly optimistic in most cases, it does result in a smaller search time in this case, say, t^u_{minp}, given by,

$$t^u_{minp} = d \cdot t_g \ . \tag{5}$$

A.1.1.2 Maximum Time

This time, t^u_{max}, results from a search strategy that is so bad that exhaustive search of the tree must be performed before the goal node is found. In this case, the search time, t^u_{max}, is

$$t^u_{max} = (n-1) \cdot t_g + (n-1-n_t) \cdot t_s \ . \tag{6}$$

Note that this is the same search time that would result if exhaustive search were required for any reason, as, for example, if the problem were to find *all* goal nodes (ignoring the fact that t_s would likely be minimal in this case, in that it would generally make no difference how nodes were selected for expansion).[1]

A.1.2 Multiprocessor Search

The search strategy is that presented in Section 3.1.1, namely, that a node is distributed for expansion by another processor as soon as it is generated; there is thus no time required for selection of nodes and the search time depends on the time to generate a successor node, which we assume is still t_g, and the time to distribute a node to another processor, t_c. We assume, in the following, that the t_c cost must be incurred any time the expansion of a node is started by a processor, even if the node was generated by that processor. This would be the case, for example, if a central task repository were used in the multiprocessor. It leads to a somewhat pessimistic estimate for search time (and therefore speedup) but simplifies the analysis considerably. We will later consider the effect of dropping this assumption.

A.1.2.1 Minimum Time

The minimum time for a multiprocessor to find a single node in a regular tree, t^m_{min}, is

$$t^m_{min} = d \cdot t_g + (d - 1) \cdot t_c \quad . \tag{7}$$

If t_c is minimal compared to t_g, then the multiprocessor requires the same minimum time as a uniprocessor that uses a *pure* depth-first search strategy.

A.1.2.2 Maximum Time

The maximum time, t^m_{max}, is given by

$$t^m_{max} = d \cdot b \cdot t_g + (d - 1) \cdot t_c \quad . \tag{8}$$

This is equivalent to the time required to expand the nodes that border the tree on one side. (The reader may wish to refer back to Section 3.1.1 (in which $t_c = 0$ was assumed) to verify this equation.)

---

[1] The estimate of (6) is slightly pessimistic in that we have assumed that the time t_s is always required for selection of a node for expansion, even for the last node at a particular depth. We could correct this by subtracting a further $(b^{d-1} - 1)/(b - 1)$ from the term multiplied by t_s in (6). We could also consider variable t_s, but this seems like an unnecessary flourish for the level of analysis considered here.

A.1.3 Compositions Of Regular Trees

Derivations of search times can readily be extended to *compositions* of regular trees. Consider, for example, a tree of depth d that is made up of a regular tree of depth d_1 ($d_1 < d$) and branching factor b_1, in which each of the tip nodes is extended by another regular tree of depth d_2 (where $d_1 + d_2 = d$), with branching factor b_2. In this case, the minimum multiprocessor search time is the same as in (7), and the maximum search time is given by the summation of the maximum time for the regular tree of depth d_1 and the maximum time for *one* of the regular trees of depth d_2; that is,

$$t^m_{max} = d_1 \cdot b_1 \cdot t_g + (d_1 - 1) \cdot t_c + d_2 \cdot b_2 \cdot t_g + (d_2 - 1) \cdot t_c \ . \qquad (9)$$

The search times for uniprocessors can be derived in the same way and are given below.

$$t^u_{min} = ((d_1 - 1) \cdot b_1 + 1) \cdot t_g + (d_1 - 1) \cdot t_s + ((d_2 - 1) \cdot b_2 + 1) \cdot t_g + (d_2 - 1) \cdot t_s \ . \qquad (10)$$

$$t^u_{minp} = d \cdot t_g \ . \qquad (11)$$

$$t^u_{max} = (n_1 - 1) \cdot t_g + (n_1 - 1 - n_{1,t}) \cdot t_s + (n_2 - 1) \cdot t_g + (n_2 - 1 - n_{2,t}) \cdot t_s \ . \qquad (12)$$

Therefore, although we will only explicitly consider regular trees in the remaining analysis, the results are easily extended to a more general class of trees--those which are compositions of regular trees.

A.1.4 Speedup

The speedup obtainable from the application of a multiprocessor to the search of a regular tree, S, is given by

$$S = t^u / t^m \ . \qquad (13)$$

The speedup obtainable from the use of a multiprocessor in searching a regular tree depends on the amount of search that is actually carried out. The speedup for minimum search, S_{min}, is unity at best (if *pure* depth-first search is used by the uniprocessor) and will be less due to communications overhead in the multiprocessor. The speedup for exhaustive, or maximum, search, S_{maxe}, is given by the following,

$$S_{maxe} = ((n - 1) \cdot t_g + (n - 1 - n_t) \cdot t_s) / (d \cdot b \cdot t_g + (d - 1) \cdot t_c) \ . \qquad (14)$$

Note that S_{maxe} is not the maximum attainable speedup for a regular tree. It is, however, a convenient measure for comparison. In Section we will derive the address of the tip node for which the maximum speedup is attained.

Note also that as the selectivity of the search strategy is augmented, thus diminishing the need for exhaustive search, the advantage of concurrent computation is also diminished.

In order to draw some simple conclusions from the S_{maxe} equation, we will assume that $t_s << t_g$. Under this assumption,

$$S_{maxe} \approx (n - 1)\cdot t_g / (d\cdot b\cdot t_g + (d - 1)\cdot t_c) \quad (15)$$

or

$$S_{maxe} \approx (n - 1)/(d\cdot b + (d - 1)\cdot(t_c / t_g)) \ . \quad (16)$$

t_c / t_g is a measure of the *coupling* between processor nodes for the distributed search problem. We will call this ratio the *processor-coupling-factor*, C_p. Note that C_p depends on both the characteristics of the task and the characteristics of the multiprocessor. It is thus not the same as the task-coupling-factor, C_t, of Section 2.2.3. Thus, rewriting the equation for S_{maxe}, we have

$$S_{maxe} \approx (n - 1)/(d\cdot b + (d - 1)\cdot C_p) \ . \quad (17)$$

Figure A.2 shows the variation in S_{maxe} as a function of C_p for a regular tree of branching factor 2 and depth 10. The cost of a mismatch between the task kernel-size and the communications characteristics of the multiprocessor is again apparent. The interpretation here is identical to that presented in Section 2.2.3; that is, that loose-coupling must be ensured by a proper match of task kernel-size to multiprocessor communications characteristics if a significant speedup is to be achieved through use of a distributed approach.

Figure A.3 shows the maximum speedups attainable for exhaustive search of three regular trees, of branching factor 2, 3, and 6, as a function of depth, under the assumption that $C_p = 0$. Also shown is a more realistic version of S_{min}, obtained by comparing the minimum time for a multiprocessor to search a regular tree, t^m_{min}, with the minimum time required for a uniprocessor, t^u_{min}, assuming that all successors of a node must be generated before a new node can be selected for expansion. It is given by the following (including C_p),

$$S_{min} \approx ((d - 1)\cdot b + 1)/(d + (d - 1)\cdot C_p) \ . \quad (18)$$

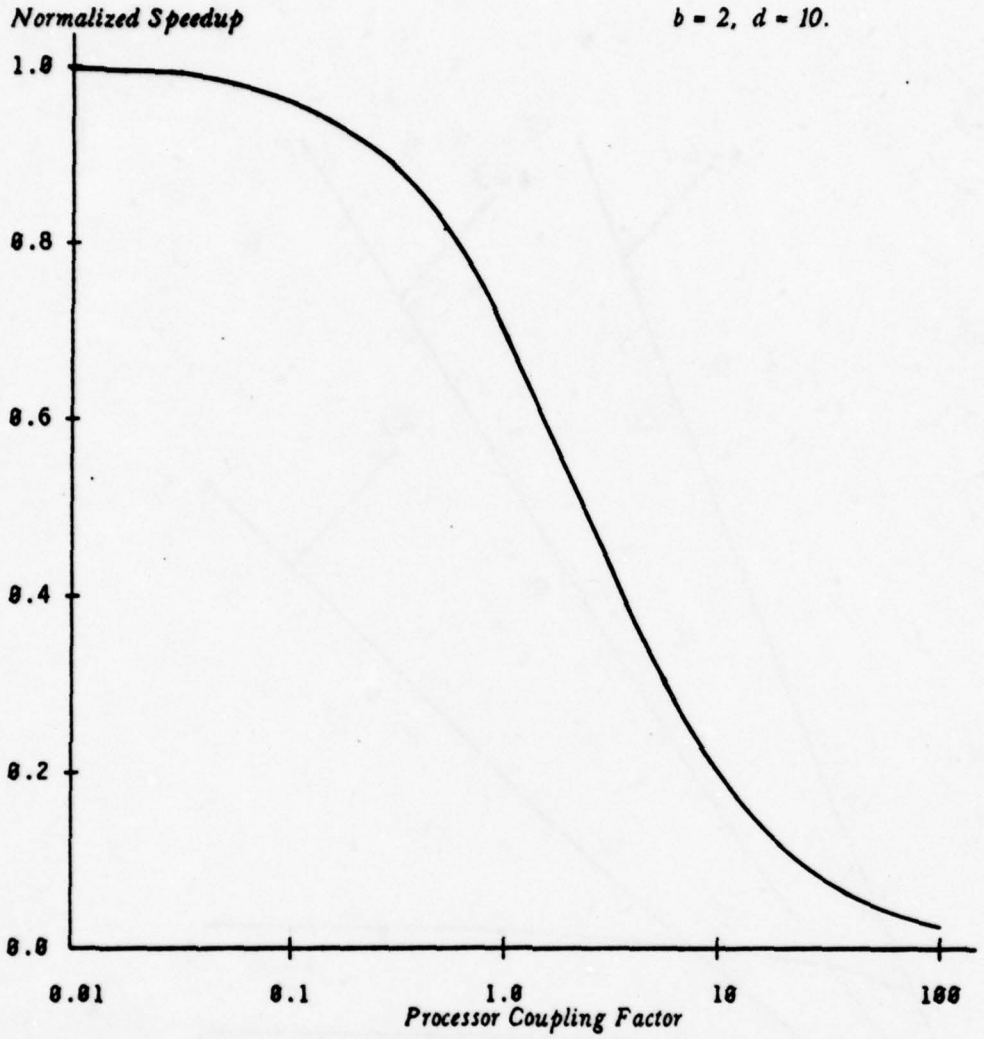


Figure A.2. The Effect Of Coupling On Speedup.

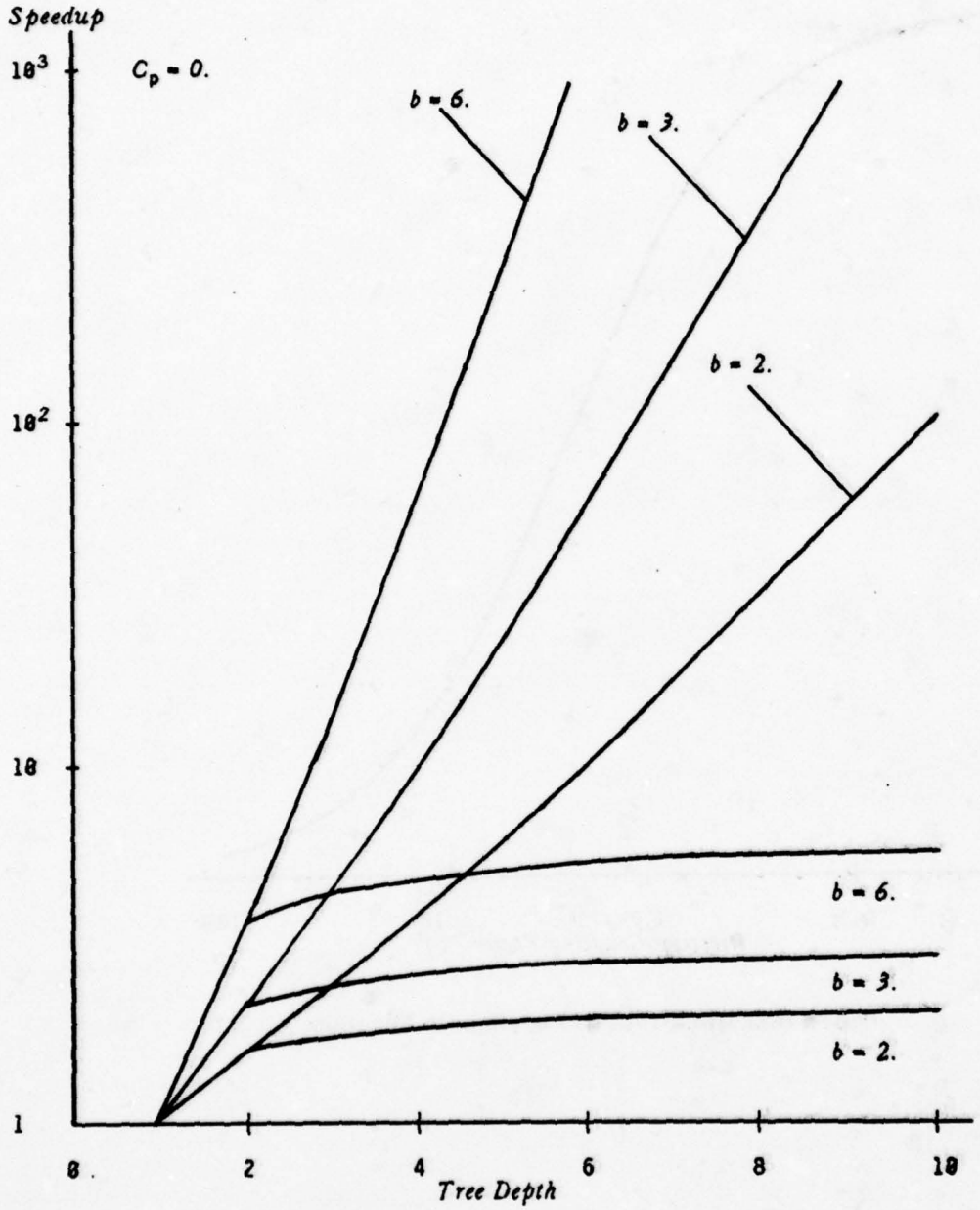


Figure A.3. Speedup For Three Regular Trees ($C_p = 0$).

A.2 Searching Regular Trees: Processor Requirement

We consider the number of processors required in a multiprocessor to obtain a speedup for the search of a regular tree. We restrict ourselves to the *exhaustive search* case and derive simple lower and upper bounds. We further assume that $t_c = 0$, and that $t_g = 1$.

As a lower bound we make an *area* approximation: We know that a uniprocessor requires t^u_{max} time units to perform an exhaustive search of a regular tree. This can be considered as the area of a rectangle that represents the amount of work that must be done to execute the search. Now, a multiprocessor requires t^m_{max} time units for the same search. This can be thought to represent the dimension of one side of the rectangle. The required number of processors, P_{max}, forms the other dimension of the rectangle, under the assumption that all processors are fully utilized throughout the period of the computation. Thus,

$$P_{max} = S_{maxe} \quad . \tag{19}$$

This assumption, however, leads to an overly optimistic estimate (i.e., underestimate) for P_{max}, since very few processors are in use near the beginning of the search. It is certainly a lower bound, however, and is intuitively reasonable. The equation simply states that we must have (at least) x processors to achieve a speedup of x.

In order to improve our estimate, let us consider the rate at which processors are pressed into service as the search continues. The number of new tasks generated at each successive time unit in the search of a tree of infinite depth and branching factor b is given by the following summation,

$$P^*_j = (P^*_{j-1} + P^*_{j-2} + \ldots + P^*_{j-b}) \quad . \tag{20}$$

$$j = 1,2,3, \ldots$$
$$P^*_j = 0, \; j < -1.$$
$$P^*_{-1}, P^*_0 = 1.$$

This corresponds to a generalized Fibonacci series (see, for example, [Vilenkin, 1971]) of order b and gives us the number of processors required for the search at each instant of time, assuming that the tree is of infinite depth (the "*" superscript is used to indicate that the series is written for a tree of infinite depth). To account for the finite depth of the trees of interest, however, the equation must be modified. The modification is straightforward once we observe that each time a processor reaches a tip node in the tree, the effect is to prune a subtree from the infinite tree. This pruning begins after d time units. We can account for the pruning of these subtrees by subtracting Fibonacci series, that start at times when processors reach tip nodes, from the original series. The number of series to be subtracted at each time instant corresponds to the number of tip nodes reached at that time instant. The number of tip nodes reached at each instant of time (starting at the d^{th} time instant, when the first tip node is reached, to the $b \cdot d^{th}$ time instant, when the search is completed) is given by,

$$k_j = C_b(d, j-d) \tag{21}$$

$$j = d, d+1, d+2, \ldots, b \cdot d.$$
$$k_j = 0, \; j < d, \; j > b \cdot d.$$

where $C_m(n, k)$ is the coefficient in the n^{th} row and the k^{th} column of the generalized Pascal's triangle of order m (sometimes called the m-arithmetic triangle) [Vilenkin, 1971] (where the initial row and column both have index 0). In general, the $C_m(n, k)$ of a generalized Pascal's triangle obey the equation

$$C_m(n, k) = C_m(n-1, k) + C_m(n-1, k-1) + \ldots + C_m(n-1, k-m+1) \quad . \tag{22}$$

$$0 \le k \le n \cdot (m-1), \quad n \ge 0.$$

$$C_m(0, 0) = 1.$$
$$C_m(1, k) = 1, \quad 0 \le k \le m-1.$$
$$C_m(1, k) = 0, \quad k \ge m.$$

Thus the actual number of processors required, P_j, is given by

$$P_j = P_j^* - k_d \cdot P_{j-d}^* - k_{d+1} \cdot P_{j-(d+1)}^* - \ldots - k_{b \cdot d} \cdot P_{j-(b \cdot d)}^* \quad . \tag{23}$$

$$j = 0, 1, 2, \ldots, b \cdot d.$$

And an upper bound on the number of processors required is given by

$$P_{max} = MAX(P_j) \quad . \tag{24}$$

$$0 \le j \le b \cdot d.$$

This estimate of the required number of processors is an upper bound because of the assumption that nodes cannot be queued for later expansion, but instead have to be expanded as soon as they are generated. This is not generally required, the result of lower demand for processors as the search nears termination.

Figure A.4 shows the two bounds for the required number of processors for exhaustive search of the same three trees as used in Figure A.3. Also shown is the actual number of processors required to achieve the maximum exhaustive search speedups (as determined by simulation).

Figure A.5 shows the normalized speedup for a tree of branching factor 3 and depth 6 as the number of available processors is varied. It is apparent that many processors are required to achieve the last epsilon of speedup. Also shown in the figure is the efficiency, E, for the search, where

$$E = S/P \quad . \tag{25}$$

and P is the number of processors involved in the search. This is a conservative estimate of efficiency, in that processors that stand idle at the beginning of the search are included in P. These processors might be applied to another top-level problem during this time in a general-purpose multiprocessor.

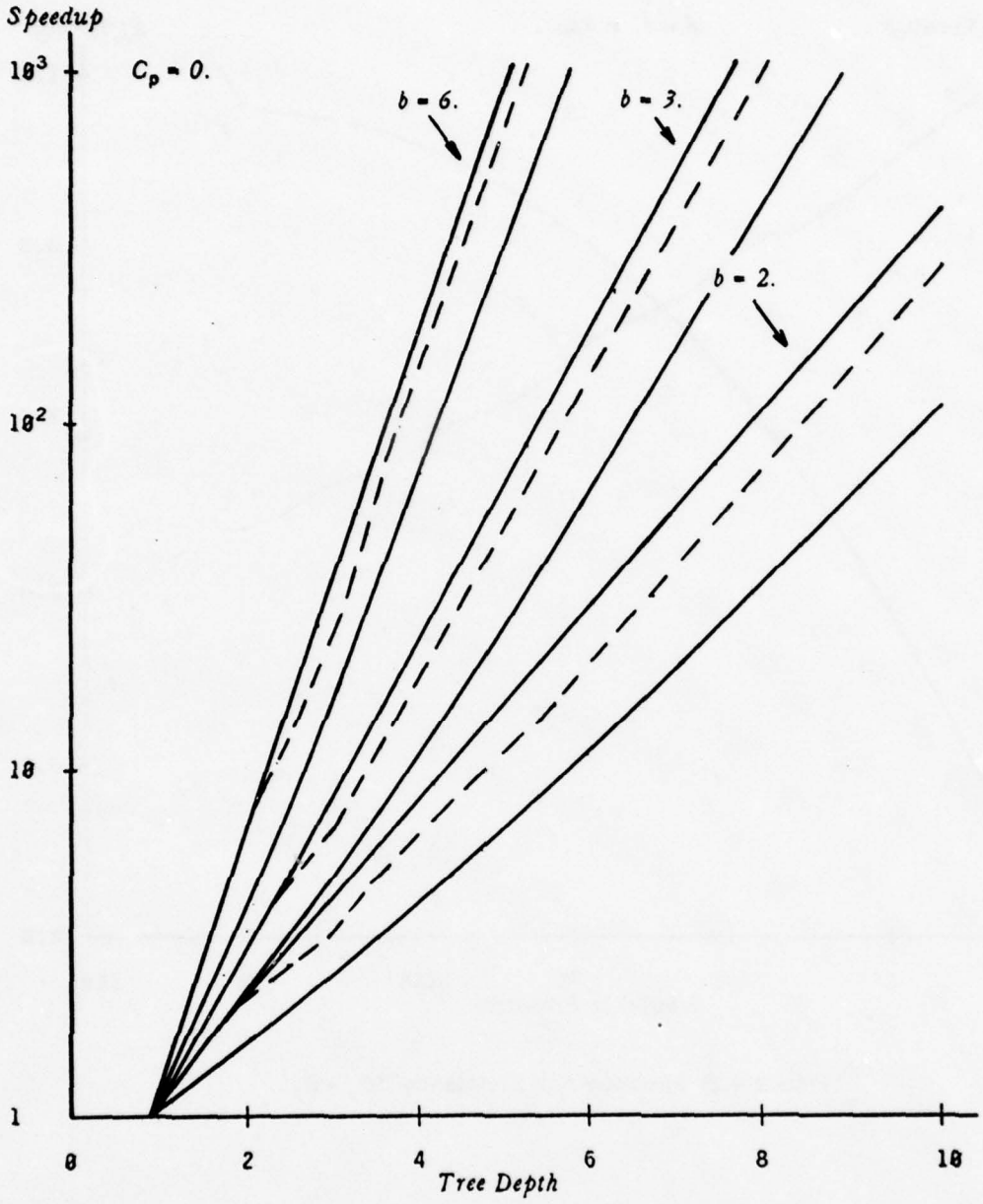


Figure A.4. Processors For Maximum Exhaustive Search Speedup ($C_p = 0$).

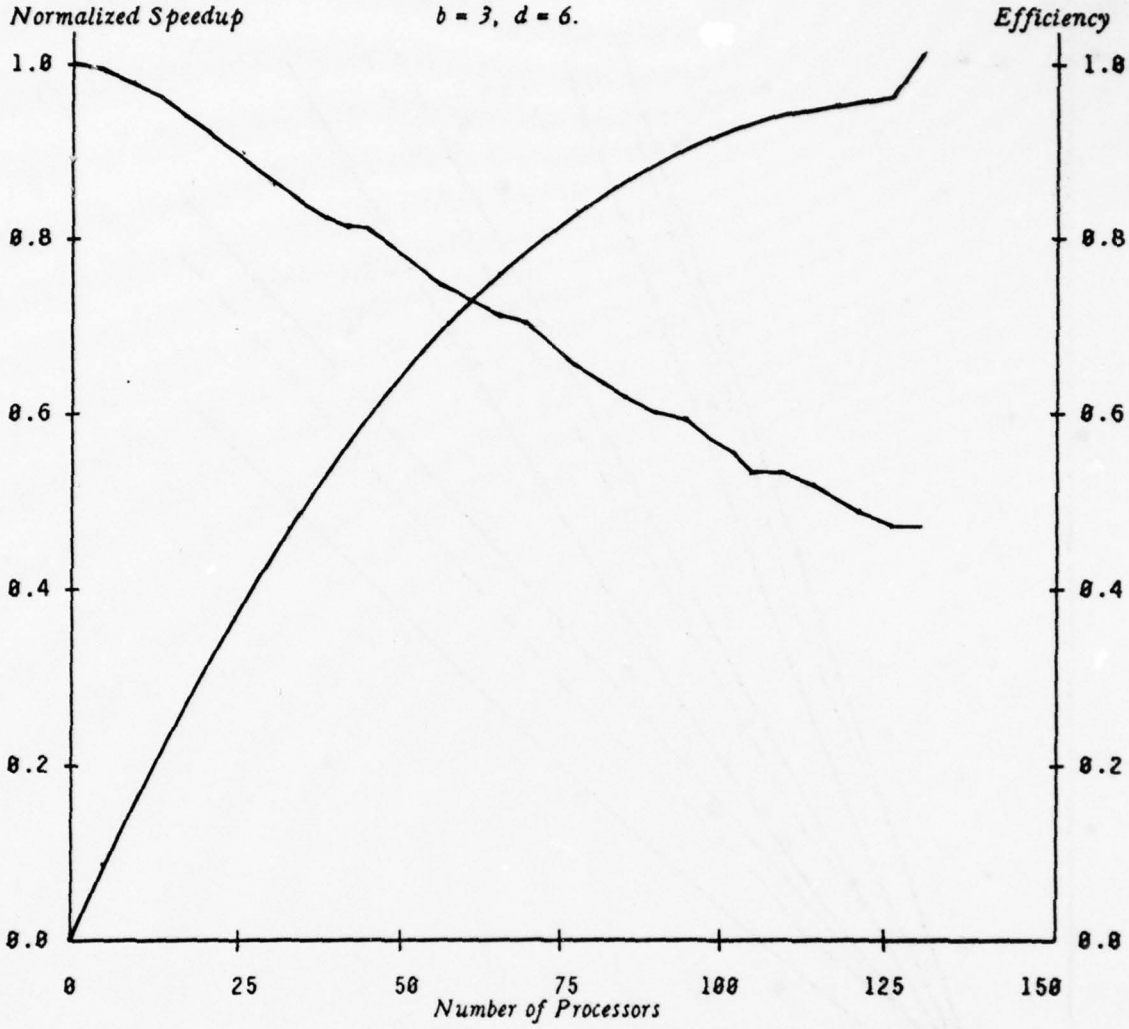


Figure A.5. Speedup And Efficiency ($C_p = 0$).

We noted earlier that our speedup estimates for multiprocessor search are slightly pessimistic, because of our assumption that the cost t_c is *always* incurred when a processor acquires a node for expansion. In Figure A.6, we consider the effects of dropping this assumption. The figure compares the possible speedups for varying numbers of processors on a tree of depth *6* and branching factor *3* for both the global queuing of nodes to be expanded and the *local* queuing of such nodes. In a local queuing strategy, a processor only acquires a node from another processor when it has none of its own to expand; that is, as a processor generates new nodes, it queues them locally for expansion and processes them alone as soon as it can. Processors that are idle and have no nodes queued for expansion must still acquire them from other processors (using, for example, the contract negotiation approach described in Section 4.1.2), at a cost of t_c.

Local queuing strategies are useful when the processor-coupling-factor, C_p, is high. In Figure A.6, $C_p = 1$. We see a small improvement for local queuing in each case.

For further comparison, *two* local strategies have been used for the figure: local breadth-first and local depth-first (refer back to Section 3.1.1 for a description of these strategies). We see that a breadth-first strategy leads to slightly larger speedups than does a depth-first strategy, mainly because tasks get distributed to idle processors earlier in the search.

A.3 The Maximum Speedup

We now derive the address of the tip node at which the maximum speedup is attained. As in the previous section, we will assume that $C_p = 0$, $t_c = 0$, and $t_g = 1$.

We can write the *address* of a tip node, a_k, as follows

$$a_k = x_k + n_t \qquad 0 \le k < n_t . \tag{26}$$

x_k is the *index* of the tip node, $0 \le x_k < n_t$. a_k is also the number of time units required by a uniprocessor to reach the tip node with that address, using a breadth-first search algorithm, under the above assumptions.

The number of time units required by a multiprocessor to reach the node with address a_k is given by

$$y_k = \Sigma_{j=0}^{d-1} \; 2^{bj} \tag{27}$$

where $b_j = \{0, 1\}$, $0 \le j \le d-1$ are the values of the bits in the binary representation of the index, x_k.

Hence, the address of the node for which the maximum speedup is attained, a_{smax}, is given by

$$a_{smax} = MAX(a_k / y_k) \tag{28}$$

which is also the maximum speedup for the tree, S_{max}, under the simplifying assumptions.

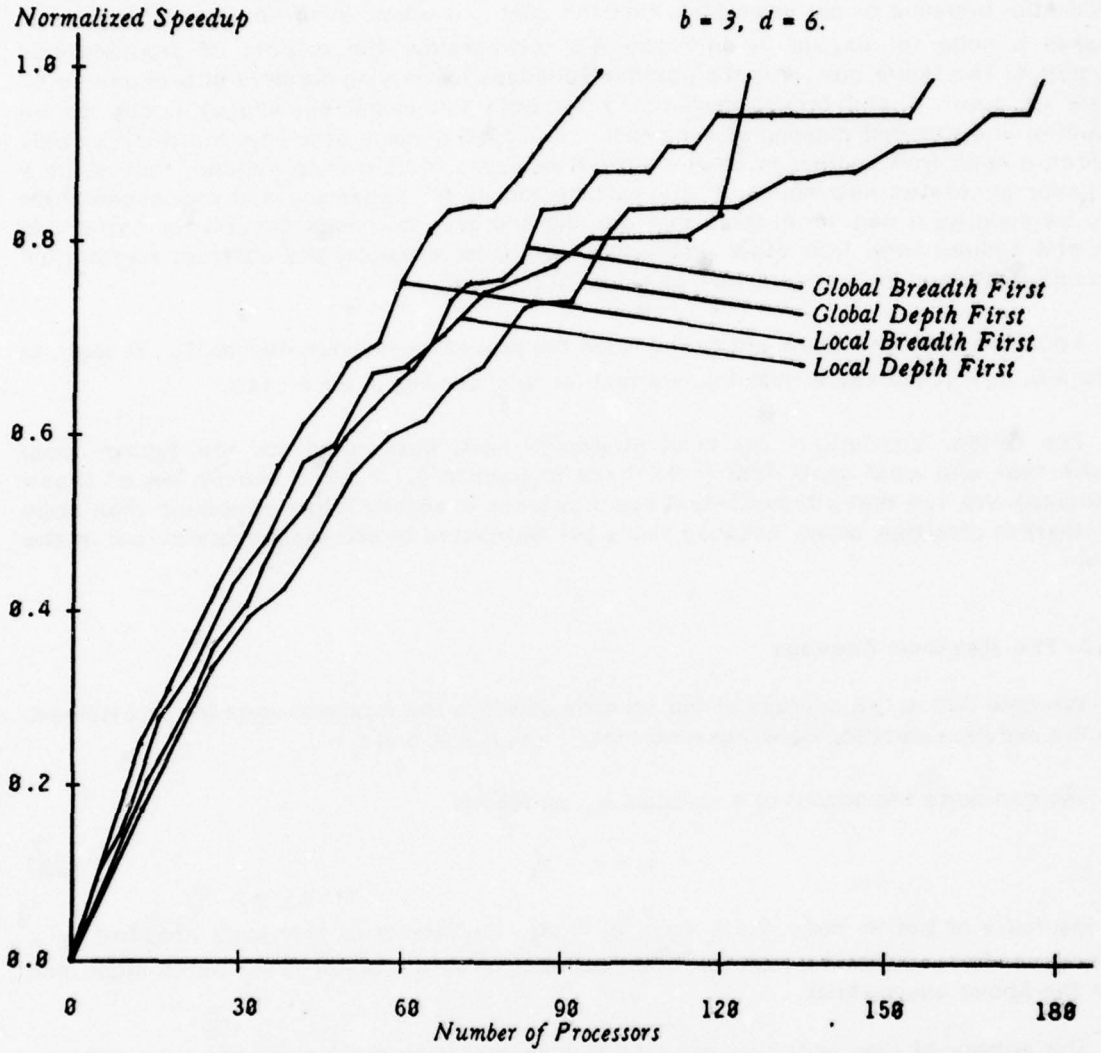


Figure A.6. The Effect Of Local And Global Queuing Strategies On Speedup ($C_p = 1$).

Appendix B

Speculative Examples

We discuss here several other possible applications of the contract net framework. The examples demonstrate the implementation of well-known *AI* problem-solving techniques in the framework. The aim of adopting a distributed approach in these examples is to increase problem-solving speed. The issues are: *(i)* how is distribution to be done; and *(ii)* how is the framework useful?

B.1 GPS

The **General Problem Solver**, or GPS [Ernst, 1969] uses a problem-solving technique called *means-ends analysis* to solve a wide variety of problems. At each stage of the problem-solving activity, a *difference* between the current state and the goal state is computed, and an *operator* is selected to reduce the difference.

In a contract net implementation, one operator is placed in each processor node. The search starts at one top-level node, as in the N Queens problem. This node computes a difference between the initial state and the goal state and attempts to find another node to reduce the difference. It transmits a task announcement in which the eligibility specification is the computed difference. Other nodes receive this announcement and determine their applicability by comparing the advertised difference with their own preconditions. They return bids that indicate the extent to which they can reduce the difference. The top-level node then awards contracts to several of the bidders who concurrently attempt to reduce the difference. The award for the task contains the descriptions of the initial state and the goal state.

This process is recursive, as nodes are forced to call in different nodes as subcontractors to assist in reducing differences. A node may at any time compute a number of differences between the current state and the goal state. It attempts to reduce these differences concurrently by making task announcements for different contracts.

The contract net framework is useful for connecting nodes with differences to be reduced and nodes that are capable of reducing those differences. The negotiation process also enables an extension in capability over the original GPS. In the original, only differences between two states were ordered. (The most difficult difference was chosen to be reduced first.) In a contract net implementation, operators for reducing differences can be similarly ordered by a manager upon receipt of bids. This enables more effective utilization of resources.

B.2 STRIPS And ABSTRIPS

STRIPS is a robot problem solver that uses means-ends analysis to enable a robot to navigate and rearrange objects in a simple world of rooms, doors, and boxes [Fikes, 1971]. In STRIPS a resolution theorem-prover is used to extract differences. The contract net implementation is similar to that used for GPS. One or more operators are placed in each processor node. In this case, however, we assume that the task of computing a difference is large enough so that it too should be distributed. Therefore theorem-provers are only placed in some of the nodes. The task of reducing a difference is distributed as in the previous example. For the task of computing a difference, the task announcement includes, as a task abstraction, brief descriptions of the two states between which the difference is to be computed.

As stated earlier, the contract net framework is useful for connecting nodes with differences to be reduced and nodes capable of reducing those differences. For STRIPS, however, the framework is also useful for connecting nodes with differences to be computed and nodes capable of computing those differences.

STRIPS has been augmented with a MACROP facility for storing useful operator sequences [Fikes, 1972]. In a contract net implementation, MACROPS are stored in individual nodes that can participate in the negotiation like any other node. When they are applicable, of course, their bids will be more attractive than those of nodes that possess only single operators, because they can reduce larger differences.

ABSTRIPS [Sacerdoti, 1974] is a system constructed on top of the original STRIPS to perform hierarchical planning. The STRIPS operator preconditions (the *add lists* and *delete lists*) are rewritten to impose a hierarchical relationship on them. The most important of the elements of these lists are collected in the highest level abstraction space; the next most important, in a second abstraction space; and so on, down to the ground space. ABSTRIPS then pushes forward an end-to-end search in each abstraction space before moving down to the next level in the hierarchy of spaces and pushing forward again, using the search from the higher levels as a skeleton plan. When a plan fails in a particular abstraction space, control is returned to the next higher level space, the node that caused the failure is removed from consideration, and the search to the goal in that space is redone.

In a contract net implementation, some nodes can be used to check the viability of plans developed in the highest level abstraction space, *while that plan is still being developed*; that is, as operators are selected in the highest level space, nodes can be allocated to check their application in successively lower level spaces while other nodes proceed with the highest level search. Thus the search proceeds in two dimensions simultaneously: end-to-end in any particular abstraction space with vertical checking in successively lower level abstraction spaces. This further reduces backtracking by noticing failures as quickly as possible.

Some of the ideas here are quite similar to those used by [Sacerdoti, 1975] in the NOAH system. Sacerdoti uses a *procedural net* of states in a *distributed world model*. A contract net as described here can be viewed as an extension of those ideas to the distributed processing environment, in which nodes in the net have their own processing capabilities. Contract negotiation, in this view, is simply a way of connecting the nodes.

B.3 MYCIN-like Rule-Based System

MYCIN is a rule-based system that assists physicians in diagnosis and treatment of blood infections [Shortliffe, 1976]. The system uses backward chaining through an extensive collection of domain-specific production rules [Davis, 1977a] to make inferences. This type of system is coming into increasing use in the *AI* community for the construction of high-performance systems [Feigenbaum, 1977], [Duda, 1978].

In a contract net implementation, the rules are placed in separate processor nodes. Each node has one rule (or a set of rules), together with a copy of the rule interpreter.[1] The task to be distributed is firing a rule, given the current problem state. Nodes attempt to match the right-hand sides of their rules (remember that MYCIN uses backward chaining) against the abstraction of the problem state included in a task announcement. A bid indicates the applicability of some of the rules in the local knowledge base of a node. The process proceeds recursively as nodes, trying to instantiate the left-hand sides of their rules, are forced to seek help from other nodes. Meta-rules [Davis, 1977c] can also be stored in separate nodes and used to order the invocation of the domain-specific rules.

The contract net framework is useful here primarily because it enables use of KS-centered knowledge for determining rule applicability. Nodes can concurrently check the applicability of their rules to announced tasks. The framework also enables the application of these rules to be ordered according to task-centered knowledge (e.g., meta-rules).

B.4 Parsing

Kaplan [Kaplan, 1973] has suggested the implementation of grammars as *asynchronous communicating parallel processes* for parsing natural languages, with the aim of reducing the exponential computation often required in linear control approaches to the problem. We will restrict ourselves to ATN grammars in the following, although Kaplan considered other possibilities.

In a contract net implementation, one node is originally charged with the task of parsing the sentence. This node becomes the top-level manager in a contract net. The node has a copy of the top-level grammar of the parser. Other nodes in the net have specialized knowledge that enables them to assist the top-level node in the parsing. Specifically, other nodes contain subgrammars (e.g., the noun phrase grammar). Several nodes may possess the same specialized knowledge. The top-level manager starts the parsing by attempting to find suitable specialists to do the processing required to make transitions between nodes in the top-level ATN. Its announcements contain, as a task abstraction, an initial fragment of the portion of the sentence to be parsed. Nodes with subgrammars return bids indicating that an initial indication of suitability has been found (i.e., the initial fragment indicates that a particular specialist may be able to assist). The manager uses the returned bids to select suitable specialists to attempt each transition. It therefore has control over scheduling of

[1] Nodes should contain sets of rules that deal with different topics to achieve maximum concurrency--at potential cost in reliability. If enough nodes exist, then each node contains exactly one rule.

the specialists (Kaplan used an agenda mechanism and a central scheduler for this [Kaplan, 1975]). The task specification is the complete sentence fragment. The manager accepts reports, integrates results, starts up other specialists to make further transitions, and so on.

Each specialist can itself become a manager by generating further subtasks as it attempts to make a transition (it could even become a manager for the top-level node by calling for the complete grammar to be used on a transition). Thus a recursive hierarchical structure can be implemented.

The contract net framework is useful for this application as a communications structure and as a method for setting up an asynchronous hierarchical control structure. The managers are suitable places for the integration of results. They can also coordinate the efforts of nodes working on lower-level tasks, through the hierarchical structure.

Appendix C

CNET: The Experimental Contract Net System

C.1 Introduction

CNET is a system of INTERLISP [Teitelman, 1975] functions that enables a user to simulate the solution of a problem using the contract net framework.[1] The system simulates the operation of a collection of processor nodes as specified in Section 6. We discuss here some aspects of interaction with CNET.[2]

CNET is an event-driven simulation. It begins by asking the user for simulation parameters (e.g., the number of processor nodes to be used, the time required to generate and transmit a task announcement, etc.). It then transfers control to an initializing function supplied by the user. This function is responsible for initializing the local knowledge bases of the nodes, and returning a list of top level tasks (each task is specified by a task type and the necessary initial data).

These tasks are assigned by CNET to nodes as contracts. CNET then starts processing those contracts. Quasi-parallelism is maintained by initializing the task execution procedures as *generators* using the INTERLISP spaghetti stack. These procedures interact with simulator procedures (e.g., to announce tasks, send reports, etc.) through a simulator-system call. Each time such a call is made, the caller is suspended and the next event on the event list is processed. This event-processing continues until all top-level tasks have been completed.

A variety of display options have been provided, so that, for example, a user can examine the message traffic, or write messages to the listing file in the correct sequence (annotated to include the name of the node from which they originate).

The user is also asked to specify a finalizing function to be called each time a top-level task is completed (to close files, write messages, perform any cleanup operations, etc.).

After all top-level tasks have been completed, a summary of message traffic and processor node utilization statistics is displayed (see below).

C.2 Common Internode Language

The core common internode language is defined by CNET, but functions are provided to allow the user to specify new objects and attributes (e.g., *signal*, in the DSS). Functions are also provided to simplify storage and retrieval of objects in the local knowledge bases of the nodes.

---

[1] The author gratefully acknowledges Bill VanMelle, Carli Scott, Jim Bennett, and the rest of the MYCIN hackers for their assistance with the INTERLISP implementation.

[2] A more complete description of CNET will follow in a later document.

C.3 A Sample Interaction

The following trace shows one interaction with CNET in solving the 4 Queens problem discussed in Section 3.1.2. For brevity, the contents of the messages have been deleted, but italicized statements are included as commentary about the trace. Actual sample messages were shown in Section 3.1.3.1.

```
----- CONTRACT NET Simulation -----

Maximum number of processor nodes in the CONTRACT NET (>0)  [ 10 ] ** 5
```
<The user is first asked for simulation parameters. Default answers are shown in "[]". Defaults can be accepted with <CR>>.

```
Task time expansion factor  [ 100 ] ** ?
```
<HELP messages are stored in a hashfile. They are displayed in response to "?".>

```
The factor by which simulation time units expended in task-related processing are
to be expanded (for purposes of timing measurements) over simulation time units
expended in contract-related processing. This factor effectively alters the
"coupling factor" of the net.

Task time expansion factor  [ 100 ] **

Terminated contracts  [ 10 ] ** ?

The number of contract structures that are retained by a node in the terminated
state. When this number is reached by a node, it deletes the oldest contract
(after calling an applications function that may, for example summarize
information that the user wants to retain).

Terminated contracts  [ 10 ] **

Default delay parameters  [ YES ] ** ?

These parameters define the number of simulation time units required to perform
various CONTRACT NET functions (e.g., task announcement).

Default delay parameters  [ YES ] ** No

Time to make a task announcement  [ 1 ] **
```
<This interaction continues until all of the delay parameters have been assigned values.>

```
Display messages  [ NO ] ** ?
```
<There are a number of similar display options.>

```
Initial Applications Function  [ $INITIALIZE ] ** ?
```
<The default procedure would give a short demonstration of CNET.>

The name of the function called by the CONTRACT NET simulator to initialize the nodes for the application and return a list of top level tasks to be executed.

```
Initial Applications Function  [ $INITIALIZE ] **  QINITIALIZE
```
<The user opts for the N Queens problem.>

```
Final Applications Function  [ $FINALIZE ] ** ?
```

The name of the function called by the CONTRACT NET simulator whenever a top level task is completed.

```
Final Applications Function  [ $FINALIZE ] ** QFINALIZE
```
<End of questions.>

```
  CONTRACT NET Simulation Parameters

     Number of Processor Nodes in Net: 5
     Applications time unit expansion: 100
     Contracts held in terminated state: 10

  CONTRACT NET Delay Parameters

     Time to make a task announcement: 1
     Time before a task is re-announced: 1000
     Time to process a task announcement: 1
     Time to make a node availability announcement: 1
     Time to process a node availability announcement: 1
     Time to make a bid: 1
     Time to process a bid: 1
     Time to make an announced award: 1
     Time to process an announced award: 1
     Time to make a directed award: 1
     Time to process a directed award: 1
     Time to acknowledge a directed award: 1
     Time to process an acknowledgment: 1
     Time to make a report to another node: 1
     Time to process a report: 1
     Time to generate a termination: 1
     Time to process a termination: 1
     Time to generate a request: 1
     Time to process a request: 1
     Time to generate an information message: 1
     Time to process an information message: 1
```
<Times are shown as multiples of one simulator time unit.>

```
::::::::::::::::::::::::::: Start of Simulation ::::::::::::::::::::::::::

Number of Queens  [ 5 ] ** 4
```
<These questions come from QINITIALIZE.>

```
Search Strategy  [ 0 ] **                    <local depth-first. The other possibilities
                                              are local breadth-first and random order.>

Report Strategy  [ 0 ] ** ?

The reporting strategy determines the action to be taken by a processor node when
it receives a report of the completion of a subtask for which it is the manager.
The possibilities are as follows:

        STRATEGY                  RESPONSE
        report interim as           0
         well as final results
        report only final           1
         results

An interim result is a report of the completion of one subtask from a collection
of subtasks generated from the same task. A final result is a report on the
collection of subtasks. It also indicates completion of processing for the
collection of subtasks.

Report Strategy  [ 0 ] **

Number of solutions  [ 1 ] ** 2          <Number of solutions to be found by any node
                                          before other paths are terminated.>

                                         <Knowledge bases are now initialized and a list
                                          of tasks is returned to CNET.>

Time: 0                                  <CNET assigns node 1 the task of extending an empty board.>

                                         <Functions are provided to allow the user to keep
                                          track of time inside task execution procedures.>

Time: 201                                <Abbreviated forms of the messages are shown below,
                                          as well as the times at which they are generated.>

To: *                                    <"*" indicates a general broadcast.>
From: 1
Type: TASK ANNOUNCEMENT
Contract: 1 1[1]
                                         <The task is to extend board (1).>

Time: 202

To: 1
From: 5
Type: BID
Contract: 1 1                            <Bids from idle nodes.>
```

[1] Contracts are assigned names by CNET as follows: Top-level contracts are assigned the name of the node in which they are processed (e.g., "1" in this case). The names of contracts generated from them are formed by concatenating the subtask sequence number to the name of the contract from which it was generated. Hence "1 1" is the name of the first contract generated from contract "1", and contract "1 3 1" is the name of the first contract that was generated from the third contract that was generated from contract "1".

```
To: 1
From: 4
Type: BID
Contract: 1 1

To: 1
From: 3
Type: BID
Contract: 1 1

To: 1
From: 2
Type: BID
Contract: 1 1

Time: 204

To: 5
From: 1
Type: ANNOUNCED AWARD
Contract: 1 1                    <Contract awarded to first bidder.>

Time: 401

To: *
From: 1
Type: TASK ANNOUNCEMENT
Contract: 2 1                    <The task is to extend board (2).>

Time: 402

To: 1
From: 2
Type: BID
Contract: 2 1

To: 1
From: 4
Type: BID
Contract: 2 1

To: 1
From: 3
Type: BID
Contract: 2 1

Time: 404

To: 2
From: 1
Type: ANNOUNCED AWARD
Contract: 2 1
```

```
Time: 601

To: *
From: 1
Type: TASK ANNOUNCEMENT
Contract: 3 1                                  <The task is to extend board (3).>

Time: 602

To: 1
From: 4
Type: BID
Contract: 3 1

To: 1
From: 3
Type: BID
Contract: 3 1

Time: 604

To: 4
From: 1
Type: ANNOUNCED AWARD
Contract: 3 1

Time: 606

To: *
From: 5
Type: TASK ANNOUNCEMENT
Contract: 1 1 1                            <The task is to extend board (1 3).>

Time: 607

To: 5
From: 3
Type: BID
Contract: 1 1 1

Time: 609

To: 3
From: 5
Type: ANNOUNCED AWARD
Contract: 1 1 1

Time: 801

To: *
From: 1
Type: TASK ANNOUNCEMENT
Contract: 4 1                                  <The task is to extend board (4).>
```

```
To: 5
From: 1
Type: BID
Contract: 1 1 1

Time: 802

To: 1
From: 1
Type: BID
Contract: 4 1

Time: 804

To: 1
From: 1
Type: ANNOUNCED AWARD
Contract: 4 1

Time: 806

To: *
From: 4
Type: TASK ANNOUNCEMENT
Contract: 1 3 1
```

<The task is to extend board (3 1).>

```
To: *
From: 5
Type: TASK ANNOUNCEMENT
Contract: 2 1 1
```

<The task is to extend board (1 4).>

```
To: 5
From: 5
Type: BID
Contract: 1 1 1

Time: 807

To: 4
From: 5
Type: BID
Contract: 1 3 1

To: 5
From: 5
Type: BID
Contract: 2 1 1

Time: 809

To: 5
From: 4
Type: ANNOUNCED AWARD
Contract: 1 3 1
```

```
To: 5
From: 5
Type: ANNOUNCED AWARD
Contract: 2 1 1

Time: 906

To: *
From: 2
Type: TASK ANNOUNCEMENT
Contract: 1 2 1
```

<The task is to extend board (2 4).>

```
To: 5
From: 2
Type: BID
Contract: 1 1 1

Time: 907

To: 2
From: 2
Type: BID
Contract: 1 2 1

Time: 909

To: 2
From: 2
Type: ANNOUNCED AWARD
Contract: 1 2 1

Time: 1006

To: *
From: 1
Type: TASK ANNOUNCEMENT
Contract: 1 4 1
```

The task is to extend board (4 1).>

```
Time: 1111

To: 5
From: 3
Type: FINAL REPORT
Contract: 1 1 1
```

<Report of failure to extend board (1 3).
The actual message is "FAILURE",
and the contract name specifies the board.>

```
To: *
From: 2
Type: TASK ANNOUNCEMENT
Contract: 1 1 2 1
```

<The task is to extend board (2 4 1).>

```
Time: 1112

To: 2
From: 3
Type: BID
Contract: 1 1 2 1

To: 2
From: 4
Type: BID
Contract: 1 1 2 1

Time: 1114

To: 3
From: 2
Type: ANNOUNCED AWARD
Contract: 1 1 2 1

Time: 1206

To: *
From: 1
Type: TASK ANNOUNCEMENT
Contract: 2 4 1                    <The task is to extend board (4 2).>

Time: 1207

To: 1
From: 4
Type: BID
Contract: 2 4 1

Time: 1209

To: 4
From: 1
Type: ANNOUNCED AWARD
Contract: 2 4 1

Time: 1311

To: *
From: 5
Type: TASK ANNOUNCEMENT
Contract: 1 1 3 1                  <The task is to extend board (3 1 4).>

Time: 1312

To: 1
From: 5
Type: INTERIM REPORT
Contract: 1 1                      <Interim report of failure to extend board (1); i.e.,
                                    a failure has occurred for one of the subtasks.>
```

```
. Time: 1407

  To: 0                                  <"0" indicates a report to the user; i.e., the function QFINALIZE.>
  From: 1
  Type: INTERIM REPORT
  Contract: 1                            <Interim report of failure to extend the empty board; i.e.,
                                          one subtask of the top-level task has resulted in failure.>

  No Solutions Found                     <This message output by QFINALIZE.>

  To: 1
  From: 1
  Type: BID
  Contract: 1 4 1

  Time: 1409

  To: 1
  From: 1
  Type: ANNOUNCED AWARD
  Contract: 1 4 1

  Time: 1515

  From: 3
  Generated Board: 2 4 1 3               <An example of a message inserted in the listing file.>

  Time: 1516

  To: 2
  From: 3
  Type: FINAL REPORT
  Contract: 1 1 2 1                      <Generated solution: (2 4 1 3).>

  Time: 1518

  To: 2
  From: 2
  Type: FINAL REPORT
  Contract: 1 2 1                        <Propagation of report of solution (2 4 1 3).>

  Time: 1520

  To: 1
  From: 2
  Type: FINAL REPORT
  Contract: 2 1                          <Propagation of report of solution (2 4 1 3).>

  Time: 1613

  To: *
  From: 5
  Type: TASK ANNOUNCEMENT
  Contract: 1 2 1 1                      <The task is to extend board (1 4 2).>
```

```
Time: 1614

To: 5
From: 3
Type: BID
Contract: 1 2 1 1

To: 5
From: 2
Type: BID
Contract: 1 2 1 1

Time: 1616

To: 3
From: 5
Type: ANNOUNCED AWARD
Contract: 1 2 1 1

Time: 1711

To: 1
From: 4
Type: FINAL REPORT
Contract: 2 4 1                    <Report of failure to extend board (4 2).>

Time: 1811

To: *
From: 1
Type: TASK ANNOUNCEMENT
Contract: 1 1 4 1                  <The task is to extend board (4 1 3).>

Time: 1812

To: 1
From: 2
Type: BID
Contract: 1 1 4 1

To: 1
From: 4
Type: BID
Contract: 1 1 4 1

Time: 1813

To: 5
From: 5
Type: BID
Contract: 1 1 3 1
```

```
Time: 1814

To: 2
From: 1
Type: ANNOUNCED AWARD
Contract: 1 1 4 1

Time: 1815

To: 5
From: 5
Type: ANNOUNCED AWARD
Contract: 1 1 3 1

Time: 1912

To: 0
From: 1
Type: INTERIM REPORT
Contract: 1                                  <Interim report of finding solution (2 4 1 3).>

Solutions Found:                             <This message output by QFINALIZE.>
 Queen-rows: 2 4 1 3

Time: 1913

To: 1
From: 1
Type: INTERIM REPORT
Contract: 4 1                                <Report of failure to extend board (4).>

Time: 1915

To: 0
From: 1
Type: INTERIM REPORT
Contract: 1                                  <Report of another failure to find a solution.>

No Solutions Found                           <This message output by QFINALIZE.>

From: 5
Generated Board: 3 1 4 2

Time: 2117

To: 5
From: 5
Type: FINAL REPORT
Contract: 1 1 3 1                            <Generated solution: (3 1 4 2).>
```

```
Time: 2118

To: 5
From: 3
Type: FINAL REPORT
Contract: 1 2 1 1                    <Report of failure to extend board (1 4 2).>

Time: 2119

To: 4
From: 5
Type: FINAL REPORT
Contract: 1 3 1                      <Propagation of report on board (3 1 4 2).>

Time: 2120

To: 5
From: 5
Type: FINAL REPORT
Contract: 2 1 1                      <Report of failure to extend board (1 4).>

Time: 2121

To: 1
From: 4
Type: FINAL REPORT
Contract: 3 1                        <Propagation of report on board (3 1 4 2).>

Time: 2122

To: 1
From: 5
Type: FINAL REPORT
Contract: 1 1                        <Report of failure to extend board (1).>

Time: 2123

To: 0
From: 1
Type: FINAL REPORT
Contract: 1                          <Report to user of finding solution (3 1 4 2) and
                                      completing the top-level task.>

Solutions Found:                     <This message output by QFINALIZE.>
 Queen-rows: 3 1 4 2

To: 1
From: 1
Type: TERMINATION
Contract: 4 1                        <Two solutions have now been found (as specified in QINITIALIZE),
                                      so processing can now be terminated on other partial boards.>
```

```
Time: 2125

To: 1
From: 1
Type: TERMINATION
Contract: 1 4 1                    <Propagation of termination message to subcontractor.>

Time: 2127

To: 2
From: 1
Type: TERMINATION
Contract: 1 1 4 1

Time: 2128                         <This time reached before all processing terminated.>

:::::::::::::::::::::::::: End of Simulation ::::::::::::::::::::::::::

Time Units to Completion: 2127
                                   <This represents a speedup of 3.6 over
                                    the comparable uniprocessor search.>

Communications Traffic Summary     <Not including messages addressed to user.>

Number of messages: 71
Number of broadcast messages: 14
Number of task announcements: 14
Number of bids: 26
Number of announced awards: 14
Number of directed awards: 0
Number of acceptances: 0
Number of refusals: 0
Number of interim reports: 2
Number of final reports: 12
Number of terminations: 3
Number of node availability announcements: 0
Number of requests: 0
Number of information messages: 0
Number of task re-announcements: 0  <Expiration time was set long enough that
                                     re-announcements were not required.>

Processor Node Utilization Statistics

     Node        Utilization
       1            .678
       2            .534
       3            .498
       4            .356
       5            .678

Mean Processor Node Utilization: .549
Standard Deviation: .135
```

Appendix D

Glossary

acceptance: an affirmative acknowledgment by a node of a directed award. This reply is a signal that the node receiving the directed award has accepted the contract for execution.

acknowledgment: a contract net protocol message used to signify that a directed award has been received. The content of the acknowledgment indicates whether or not the node that received the award has accepted or refused it. (See **acceptance** and **refusal**.)

active announcement: a task announcement for which the expiration time has not expired. Such announcements are retained by prospective bidders as a catalog of tasks on which to bid when they go idle.

active bid: a bid made on an announced task. Such bids are retained by the manager (bound to the subcontract to which they refer) pending award of the task.

announced: the processing state of a node that holds subcontracts, pending their award to other nodes.

announced award: a contract net protocol message that is sent to a successful bidder for a contract (following an announcement-bid sequence of contract negotiation) and that indicates that the bidder is now the contractor for the contract. (See **task specification**.)

award: a contract net protocol message that indicates to the recipient node that it is to execute the associated contract. (See **announced award** and **directed-award**.)

bid: a contract net protocol message sent by a node interested in executing a task to the node that announced the task. The bid indicates the capabilities of the bidder that are relevant to the execution of the task. (See **node abstraction**.)

bid specification: a slot in a task announcement that holds task-dependent information that indicates to a prospective bidder the information about its capabilities that is desired in a bid by the manager.

broadcast channel: a communications channel that permits simultaneous transmission or receipt to or from a variety of nodes.

common internode language: a formal language in which task-dependent information is encoded in messages of the contract net protocol.

communications channel: a logical communications path that connects two or more processor nodes.

communications protocol: a pre-defined set of rules that control the communication among entities (e.g., processor nodes or processes) that communicate via messages.

connection problem: the problem of linking nodes that have tasks to be executed with other nodes capable of their execution.

contract: the basic information structure of the contract net. It binds together information about the associated task, results, subtasks, and about other related nodes in the net

contract net: a collection of nodes in a distributed problem solver that interact via the contract net protocol.

contract net protocol: a problem-solving protocol used to coordinate the actions of nodes in a contract net.

contract net protocol message: a message of a node in a contract net, using the contract net protocol.

contractor: the *role* played by a processor node that has been awarded a contract and that is responsible for its execution.

coupling: a measure of the amount of interaction that exists between entities (e.g. tasks, nodes) in a distributed architecture.

die: a small (currently up to approximately 40 mm^2) slice of material, generally silicon, that forms the base of an integrated circuit.

directed award: an award that is not preceded by the normal announcement-bid sequence of contract negotiation, but which is forwarded directly to a node. (See **eligibility specification** and **task specification**.)

distributed problem solving: cooperative solution of problems by a decentralized collection of loosely coupled knowledge-sources.

distributed processing: parallel processing in which the individual processes are loosely coupled.

distributed processor architecture: collection of processor nodes (processors and memory) together with low-level communications hardware and software.

eligibility specification: a slot in a task announcement that carries task-dependent information enabling a node receiving the announcement to determine whether or not it is capable of executing the announced task. The slot is also used both in a directed award to enable the node receiving the award to determine if it can execute the awarded task and in a node availability announcement to specify the kind of task that a node is willing to execute.

expiration time: the period of time after which an announcement is no longer valid. After this period, The announced task, for example, is either be awarded or re-announced, or some other action is taken.

final report: a report made by a contractor to a manager that contains task results and indicates completion of the contract. (See **result description.**)

global: pertaining to all processor nodes in a distributed architecture.

heuristic search: search that involves dynamic construction of the search space, together with the application of heuristics or rules of good guessing, so as to order the nodes in the search space that are to be explored.

information message: a contract net protocol message to respond to a request message or to spread results or other data throughout the net. (See **information specification.**)

information specification: a slot in an information message that carries task-dependent information corresponding to the contents of the message.

interim report: a report made by a contractor on a contract to the manager for that contract while execution of the contract is proceeding. (See **result description.**)

kernel-size: the number of standard-instructions executed by a node to process a task.

local: pertaining to one processor node, or a subset of processor nodes in a distributed architecture.

local knowledge base: the knowledge base of an individual processor node.

loosely coupled processor nodes: processor nodes are defined to be loosely coupled if the ratio of the number of instructions executed per task at a node to the number of bits required for internode communication per task is large. This implies that individual nodes spend the bulk of their time in computation, and a much smaller percentage of their time in communication.

manager: the *role* played by a processor node that has generated a contract and that is responsible for monitoring its execution and processing the results.

MIMD: a multiple processor architecture composed of a collection of processors (and their associated memories), with the processors interconnected to provide a means for cooperating during a computation. Each processor has the capability to execute an independent instruction stream.

multiprocessing: simultaneous processing of two or more portions of the same program by two or more processing units.

multiprocessor: generally taken to be a parallel processor consisting of several processing elements that share a common primary memory.

multiprogramming: time and resource sharing of a computer system by two or more programs resident simultaneously in primary memory.

node: see processor node.

node abstraction: a slot in a bid that carries task-dependent information that indicates to a manager the capabilities of a bidder relevant to the execution of the task for which the bid is intended. It is also used to advertise the capabilities of an idle node in a node availability announcement.

node availability announcement: a contract net protocol message that indicates that a node is idle and looking for an appropriate task to execute. (See **eligibility specification, node abstraction** and **expiration time.**)

parallel processing: either multiprocessing, or multiprogramming, or a combination of both.

predecessors: a slot in a subcontract structure that holds the names of other subcontracts that must be executed before the subcontract can be announced.

problem-solving protocol: a pre-defined set of rules that controls the problem-solving communication among entities that communicate via messages. (See **communications protocol.**)

process: an instance of a program in execution; the code plus enough state information so that it can be executed independent of, and concurrent with, other such entities.

processing element: typically the central processor in a computer.

processor node: a processing-element/memory/communications-interface combination in a distributed architecture.

protocol: see **communications protocol** and **problem-solving protocol.**

pseudo-contract: the information saved by a node about a task upon submitting a bid on the task.

ready: the processing state of a node that holds contracts waiting to be executed.

regular tree: a search tree with a constant branching factor and a uniform depth of tip nodes.

refusal: a negative acknowledgment by a node for a directed award. It indicates that the node receiving the award is unable to execute the associated contract. (See **refusal specification.**)

refusal specification: a slot in a refusal that carries task-dependent information that indicates to a manager why a node that received a directed award is not able to execute the associated contract.

related contractors: a slot in a contract structure specifying the names of other nodes working on related contracts.

report: a contract net protocol message sent by a contractor to a manager to forward results of the execution of a contract. (See **interim report** and **final report.**)

report recipients: a slot in a contract structure specifying the names of other nodes to which reports are to be sent.

request: a contract net protocol message sent by one node to another when the first seeks to obtain relatively straightforward information from the second, and a contract relationship (which implies subtasking) is not necessary. (See **request specification.**)

request specification: a slot in a request message that carries task-dependent information specifying the information being requested.

results: a slot in a contract structure that carries the results of the execution of a subcontract. The slot may be included in a subcontract structure for the same reason.

result description: a slot in a report that carries results for the execution of a contract.

SIMD: a multiple processor architecture containing a control processor that drives several task processors (and their associated memories) in a *lockstep* manner; that is, the control processor issues a single instruction stream to all task processors, which execute it in synchronism for their own individual data streams.

standard-instruction: used as a comparison device to allow different processor types to be compared. It indicates the power of the instruction set of a reference processor (e.g., 8080).

subcontract: a contract-like structure used by a manager to hold information about subtasks that it has generated. The information includes the name of the subcontractor and may include the subtask specification, results, predecessors, and successors.

subcontract-list: a slot in a contract structure that holds a list of subtasks of the task of the contract.

successors: a slot in a subcontract structure that holds the names of other subcontracts that cannot be announced until the subcontract is completed.

suspended: processing state of a node that holds contracts whose execution cannot proceed until results from subcontract execution are received.

task: an independent segment of a sequentially organized program.

task abstraction: a slot in a task announcement message that carries task-dependent information that enables a node receiving the announcement to determine whether or not it wants to submit a bid on the announced task.

task announcement: a contract net protocol message that advertises the the existence of a task that is waiting to be executed. (See **eligibility specification, task abstraction, bid specification,** and **expiration time.**)

task specification: a slot in an award message that carries task-dependent information that specifies the task to be executed.

terminated: a processing state that holds contracts whose execution has been completed.

termination: a contract net protocol message sent by a manager to a contractor to indicate that execution of a contract is to be stopped.

References

The following abbreviations are used in the Reference section.

IJCAI2 *Proceedings of the Second International Joint Conference on Artificial Intelligence*, London, England, September 1971 (available from The British Computer Society, 29 Portland Place, London, W1N 4AP, England).

IJCAI3 *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford, Calif., August 1975 (available from SRI International Publications, 330 Ravenswood Ave., Menlo Park, Calif., 94025).

IJCAI4 *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR, September 1975 (available from MIT AI Lab, 545 Technology Square, Cambridge, Mass., 02138).

IJCAI5 *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Mass., August 1977 (available from MIT AI Lab, 545 Technology Square, Cambridge, Mass., 02138).

[Anderson, 1975]
G. A. Anderson and E. D. Jensen, Computer Interconnection Structures: Taxonomy, Characteristics, and Examples. *Computing Surveys*, Vol. 7, No. 4, December 1975, pp. 197-213.

[Baer, 1973]
J.-L. Baer, A Survey Of Some Theoretical Aspects Of Multiprocessing. *Computing Surveys*, Vol. 5, No. 1, March, 1973, pp. 31-80.

[Barrow, 1976]
H. G. Barrow and J. M. Tenenbaum, *MSYS: A System For Reasoning About Scenes*. SRI AIC TN 121, SRI International, Menlo Park, Ca., April 1976.

[Berliner, 1973]
H. J. Berliner, Some Necessary Conditions For A Master Chess Program. *IJCAI3*, 1973, pp. 77-85.

[Bloch, 1978]
E. Bloch and D. Galage, Component Progress: Its Effect On High-Speed Computer Architecture And Machine Organization. *Computer*, Vol. 11, No. 4, April 1978, pp. 64-75.

[Bobrow, 1977]
D. G. Bobrow and T. Winograd, An Overview Of KRL, A Knowledge Representation Language. *Cognitive Science*, Vol. 1, No. 1, January 1977, pp. 3-46.

[Bonnet, 1978]
A. Bonnet, *BAOBAB, A Parser For A Rule-based System Using A Semantic Grammar*. STAN-CS-78-668 (HPP-78-10), Dept. of Computer Science, Stanford University, June, 1978.

[Bowdon, 1972]
E. K. Bowdon, Sr. and W. J. Barr, Cost Effective Priority Assignment In Network Computers. *FJCC Proceedings*, Vol. 41, Montvale, N. J.: AFIPS Press, 1972. Pp. 755-763.

[Brinch Hansen, 1973]
P. Brinch Hansen, *Operating System Principles*. Englewood Cliffs, N. J.: Prentice-Hall, 1973.

[Brooks, 1975]
F. P. Brooks, Jr., *The Mythical Man-Month*. Reading, Mass.: Addison-Wesley, 1975.

[Buchanan, 1978]
B. G. Buchanan and T. M. Mitchell, Model-Directed Learning Of Production Rules. In D. A. Waterman and F. Hayes-Roth (Eds.), *Pattern-Directed Inference Systems*. New York: Academic Press, 1978. Pp.297-312.

[Carhart, 1976]
R. E. Carhart and D. H. Smith, Applications Of Artificial Intelligence For Chemical Inference XX. Intelligent Use Of Constraints In Computer-Assisted Structure Elucidation. *Computers In Chemistry*, Vol. 1, 1976, p. 79.

[Carr, 1970]
C. S. Carr, S. D. Crocker, and V. G. Cerf, Host/Host Communication Protocol In The ARPA Network. *SJCC Proceedings*, Vol. 36, Montvale, N. J.: AFIPS Press, 1970. Pp. 589-597.

[Chu, 1976]
W. W. Chu, *Advances In Computer Communications* (2nd ed). Artech House, 1976.

[Connell, 1976]
H. E. T. Connell, A Multi-Minicomputer Network for Optical Moving Target Indication. *Conference Digest, IEEE COMPCON Fall*, 1976, pp. 233-236.

[Dahl, 1968]
O-J. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA 67 - Common Base Language*. Norwegian Computing Center Pub. No. S-2, Oslo, May 1968.

[Davis, 1976]
R. Davis, *Applications Of Meta-Level Knowledge To The Construction, Maintenance And Use Of Large Knowledge Bases*. STAN-CS-76-552 (HPP-76-7), Dept. of Computer Science, Stanford University, July 1976.

[Davis, 1977a]
R. Davis and J. King, An Overview of Production Systems. In E. W. Elcock and D. Michie (Eds.), *Machine Intelligence 8*. New York: Wiley & Sons, 1977. Pp. 300-332.

[Davis, 1977b]
R. Davis, Generalized Procedure Calling And Content-Directed Invocation. *Proceedings of the Symposium on Artificial Intelligence and Programming Languages, SIGART Newsletter*, No. 64, August 1977, pp. 45-54.

[Davis, 1977c]
R. Davis and B. G. Buchanan, Meta-Level Knowledge: Overview And Applications. *IJCAI5*, 1977, pp. 920-927.

[Duda, 1978]
R. O. Duda, P. E. Hart, N. J. Nilsson, and G. L. Sutherland, Semantic Network Representations In Rule-Based Inference Systems. In D. A. Waterman and F. Hayes-Roth (Eds.), *Pattern-Directed Inference Systems*. New York: Academic Press, 1978. Pp. 203-221.

[Engelmore, 1977]
R. S. Engelmore and H. P. Nii, *A Knowledge-Based System For The Interpretation Of Protein X-Ray Crystallographic Data*. STAN-CS-77-589 (HPP-77-2), Dept. of Computer Science, Stanford University, February 1977.

[Erman, 1975]
L. D. Erman and V. R. Lesser, A Multi-level Organization For Problem Solving Using Many, Diverse, Cooperating Sources of Knowledge. *IJCAI4*, 1975, pp. 483-490.

[Ernst, 1969]
G. W. Ernst, *GPS: A Case Study In Generality And Problem Solving*. New York: Academic Press, 1969.

[Faggin, 1978]
F. Faggin, How VLSI Impacts Computer Architecture. *IEEE Spectrum*, Vol. 15, No. 5, May 1978, pp. 28-31.

[Farber, 1972]
D. J. Farber and K. C. Larson, The Structure Of The Distributed Computing System - Software. In J. Fox (Ed.), *Proceedings of the Symposium on Computer-Communications Networks And Teletraffic*. Brooklyn, N. Y.: Polytechnic Press of the Polytechnic Institute of Brooklyn, April 1972. Pp. 539-545.

[Feigenbaum, 1977]
E. A. Feigenbaum, The Art Of Artificial Intelligence: I. Themes And Case Studies Of Knowledge Engineering. *IJCAI5*, 1977, pp. 1014-1029.

[Feldman, 1977]
J. A. Feldman, *A Programming Methodology For Distributed Computing (Among Other Things)*. TR 9, Dept. of Computer Science, University of Rochester, 1977.

[Fennell, 1975]
R. D. Fennell, *Multiprocess Software Architecture For AI Problem Solving*. Dept. of Computer Science, Carnegie-Mellon University, May 1975

[Fikes, 1971]
R. E. Fikes and N. J. Nilsson, STRIPS: A New Approach To The Application Of Theorem Proving To Problem Solving. *Artificial Intelligence*, Vol. 2, 1971, pp. 189-208.

[Fikes, 1972]
R. E. Fikes, P. E. Hart, and N. J. Nilsson, Learning And Executing Generalized Robot Plans. *Artificial Intelligence*, Vol. 3, 1972, pp. 251-288.

[Floyd, 1967]
R. W. Floyd, Nondeterministic Algorithms. *JACM*, Vol. 14, No. 4, October 1967, pp. 636-644.

[Flynn, 1972]
M. J. Flynn, Some Computer Organizations and Their Effectiveness. *IEEE Trans. on Computers*, Vol. C-21, No. 9, September 1972, pp. 948-960.

[Foster, 1976]
J. D. Foster, The Development Of A Concept For Distributive Processing. *Conference Digest, IEEE COMPCON Spring*, 1976, pp. 28-30.

[Fuller, 1976]
S. H. Fuller and P. N. Oleinick, Initial Measurements Of Parallel Programs on a Multi-Mini-Processor. *Conference Digest, IEEE COMPCON Fall*, 1976, pp. 358-363.

[Galbraith, 1974]
J. R. Galbraith, Organizational Design: An Information Processing View. In D. A. Kolb, I. M. Rubin, and J. M. McIntyre (Eds.), *Organizational Psychology* (2nd ed). Englewood Cliffs, N. J.: Prentice-Hall, 1974. Pp. 313-322.

[Garcia-Molina, 1978]
H. Garcia-Molina, *Distributed Database Coupling*. HPP-78-4, Heuristic Programming Project, Dept. of Computer Science, Stanford University, March 1978.

[Gonzalez, 1972]
M. J. Gonzalez Jr. and C. V. Ramamoorthy, Parallel Task Execution In A Decentralized System. *IEEE Trans. on Computers*, Vol. C-21, No. 12, December 1972, pp. 1310-1322.

[Green, 1975]
P. E. Green, Jr., and R. W. Lucky, *Computer Communications*. New York: IEEE Press, 1975.

[Harris, 1977]
J. A. Harris and D. R. Smith, Hierarchical Multiprocessor Configurations. *Proceedings of the 4th Annual Symposium on Computer Architecture*, March 1977, pp. 41-47.

[Hayes-Roth, 1977]
F. Hayes-Roth and V. R. Lesser, Focus Of Attention In The HEARSAY-II Speech Understanding System. *IJCAI5*, 1977, pp.27-35.

[Heart, 1972]
F. E. Heart, R. E. Kahn, S. M. Ornstein, W. R. Crowther and D. C. Walden, The Interface Message Processor For The ARPA Computer Network. *SJCC Proceedings*, Vol. 40, Montvale, N. J.: AFIPS Press, 1972. Pp. 551-567.

[Hewitt, 1971]
C. Hewitt, Procedural Embedding Of Knowledge In Planner. *IJCAI2*, 1971, pp. 167-182

[Hewitt, 1972]
C. Hewitt, *Description And Theoretical Analysis (Using Schemata) Of PLANNER: A Language For Proving Theorems And Manipulating Models In A Robot*. MIT AI TR 258, MIT, April 1972

[Hewitt, 1973]
C. Hewitt, P. Bishop, and R. Steiger, A Universal Modular ACTOR Formalism for Artificial Intelligence. *IJCAI3*, 1973, pp. 235-245.

[Hewitt, 1975]
C. Hewitt and B. Smith, Towards a Programming Apprentice. *IEEE Trans. on Software Engineering*, Vol. SE-1, No. 1, March 1975, pp. 26-45.

[Hewitt, 1977a]
C. Hewitt, Viewing Control Structures As Patterns Of Passing Messages. *Artificial Intelligence*, Vol. 8, 1977, pp. 323-364.

[Hewitt, 1977b]
C. Hewitt and H. Baker, Laws For Communicating Parallel Processes. In B. Gilchrist (Ed.), *Information Processing 77*. Amsterdam: North-Holland, 1977. Pp. 987-992.

[Jensen, 1975]
E. D. Jensen, The Influence Of Microprocessors On Computer Architecture: Distributed Processing. *Proceedings of the ACM National Conference*, Minneapolis, Minn., October 1975, pp. 125-128.

[Kahn, 1972]
R. E. Kahn, Resource-Sharing Computer Communications Networks. *Proc. IEEE*, Vol. 60, No. 11, November 1972, pp. 1397-1407.

[Kahn, 1975]
R. E. Kahn, The Organization Of Computer Resources Into A Packet Radio Network. *NCC Proceedings*, Vol. 44, Montvale, N. J.: AFIPS Press, 1975. Pp. 177-186.

[Kaplan, 1973]
R. M. Kaplan, A Multi-Processing Approach To Natural Language. *NCC Proceedings*, Vol. 46, Montvale, N. J.: AFIPS Press, 1973. Pp. 435-440.

[Kaplan, 1975]
R. M. Kaplan, On Process Models For Sentence Analysis. In D. E. Rumelhart and D. A. Norman, *Explorations In Cognition*. San Francisco: W. H. Freeman, 1975. Pp. 117-135.

[Kimbleton, 1975]
S. R. Kimbleton and G. M. Schneider, Computer Communications Networks: Approaches, Objectives, and Performance Considerations. *Computing Surveys*, Vol. 7, No. 3, September 1975, pp. 129-173.

[Knuth, 1975]
D. E. Knuth, *Sorting And Searching*. Reading, Mass.: Addison-Wesley, 1975

[Knutsen, 1977]
K. G. Knutsen, *Some Issues In The Design Of Large Multi-Microprocessor Networks*. HPP-77-31 (working paper), Heuristic Programming Project, Dept. of Computer Science, Stanford University, September 1977.

[Kowalski, 1970]
R. Kowalski, Search Strategies For Theorem-Proving. In B. Meltzer and D. Michie (Eds.), *Machine Intelligence 5*. New York: American Elsevier, 1970, pp. 181-201.

[Kuck, 1975]
D. J. Kuck, Parallel Processor Architecture - A Survey. *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, Syracuse, August 1975, pp. 15-39.

[Lenat, 1975a]
D. B. Lenat, Synthesis Of Large Programs From Specific Dialogues. *Proceedings of the International Symposium on Proving and Improving Programs*, IRIA, Le Chesnay, France, July 1975, pp. 225-242.

[Lenat, 1975b]
D. B. Lenat, Beings: Knowledge As Interacting Experts. *IJCAI4*, 1975, pp. 126-133.

[Lenat, 1976]
D. B. Lenat, *AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search*. STAN-CS-76-570 (HPP-76-8), Dept. of Computer Science, Stanford University, July 1976.

[Lesser, 1975a]
V. R. Lesser, R. D. Fennell, L. D. Erman, and D. R. Reddy, Organization of the HEARSAY II Speech Understanding System. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Vol. ASSP-23, No. 1, February 1975, pp. 11-24.

[Lesser, 1975b]
V. R. Lesser, Parallel Processing In Speech Understanding Systems: A Survey Of Design Problems. In D. R. Reddy (Ed.), *Speech Recognition*. New York: Academic Press, 1975. Pp. 481-499.

[Lesser, 1977]
V. R. Lesser and L. D. Erman, A Retrospective View Of The HEARSAY-II Architecture. *IJCAI5*, 1977, pp. 790-800.

[Lesser, 1978]
V. R. Lesser, *Cooperative Distributed Processing*. COINS TR 78-7, Dept. of Computer and Information Science, University of Massachusetts, Amherst, Mass., May 1978.

[McDermott, 1974]
D. V. McDermott and G. J. Sussman, *The CONNIVER Reference Manual*. AI Memo 259a, MIT, January, 1974.

[Metcalfe, 1976]
R. M. Metcalfe and D. R. Boggs, Ethernet: Distributed Packet Switching For Local Computer Networks. *CACM*, Vol. 19, No. 7, July 1976, pp. 395-404.

[Minsky, 1971]
M. Minsky and S. Papert, On Some Associative, Parallel, and Analog Computations. In E. L. Jacks (Ed.), *Associative Information Techniques*. New York: American Elsevier, 1971. Pp. 27-47.

[Minsky, 1972]
M. Minsky and S. Papert, *Artificial Intelligence - Progress Report*. MIT AI Memo 252, MIT, January 1972.

[Newell, 1963]
A. Newell, J. C. Shaw, and H. A. Simon, Chess-Playing Programs And The Problem Of Complexity. In E. A. Feigenbaum and J. Feldman, (Eds.), *Computers And Thought*. New York: McGraw-Hill, 1963. Pp. 39-70.

[Nii, 1978]
H. P. Nii and E. A. Feigenbaum, Rule-Based Understanding Of Signals. In D. A. Waterman and F. Hayes-Roth (Eds.), *Pattern-Directed Inference Systems*. New York: Academic Press, 1978. Pp. 483-501.

[Nilsson, 1971]
N. J. Nilsson, *Problem Solving Methods In Artificial Intelligence*. New York: McGraw-Hill, 1971.

[Noyce, 1976]
R. N. Noyce, From Relays To MPU's. *Computer*, Vol. 9, No. 12, December 1976, pp. 26-29.

[Noyce, 1977]
R. N. Noyce, Microelectronics. *Scientific American*, Vol. 237, No. 3, September 1977, pp. 62-69.

[Reddy, 1973]
D. R. Reddy, Some Numerical Problems In Artificial Intelligence. In J. F. Traub (Ed.), *Complexity Of Sequential And Parallel Numerical Algorithms*. New York: Academic Press, 1973. Pp. 131-147.

[Reddy, 1975]
D. R. Reddy and L. D. Erman, Tutorial On System Organization For Speech Understanding. In D. R. Reddy (Ed.), *Speech Recognition*. New York: Wiley & Sons, 1975. Pp. 457-479.

[Roberts, 1970]
L. G. Roberts, Computer Network Development To Achieve Resource Sharing. *SJCC Proceedings*, Vol. 36, Montvale, N. J.: AFIPS Press, 1970. Pp. 543-549.

[Sacerdoti, 1974]
E. D. Sacerdoti, Planning In A Hierarchy Of Abstraction Spaces. *Artificial Intelligence* Vol. 5, 1974, pp. 115-135.

[Sacerdoti, 1975]
E. D. Sacerdoti, *A Structure For Plans And Behavior*. SRI AIC TN 109, SRI International, Menlo Park, Ca., August 1975.

[Samuel, 1967]
A. L. Samuel, Some Studies In Machine Learning Using The Game Of Checkers II - Recent Progress. *IBM Journal Of Research And Development*, Vol. 11, No. 6, November 1967, pp. 601-617.

[Shortliffe, 1976]
E. H. Shortliffe, *MYCIN: Computer-Based Medical Consultations*. New York: American-Elsevier, 1976.

[Simon, 1969]
H. A. Simon, *The Sciences Of The Artificial*. Cambridge, Mass.: MIT Press, 1969.

[Simon, 1976]
H. A. Simon and J. B. Kadane, Problems Of Computational Complexity In Artificial Intelligence. In J. F. Traub (Ed.), *Algorithms and Complexity*. New York: Academic Press, 1976. Pp. 281-298.

[Smith, 1977]
R. G. Smith, The CONTRACT NET: A Formalism For The Control Of Distributed Problem Solving. *IJCAI5*, 1977, p. 472.

[Smith, 1978a]
R. G. Smith, *Issues In Distributed Sensor Net Design*. HPP-78-2 (Working Paper), Heuristic Programming Project, Dept. of Computer Science, Stanford University, January 1978.

[Smith, 1978b]
R. G. Smith and R. Davis, Distributed Problem Solving: The Contract Net Approach. *Proceedings of the Second National Conference of the Canadian Society for Computational Studies of Intelligence*, Toronto, Canada, July 1978, pp. 278-287.

[Stone, 1975]
H. S. Stone (Ed.), *Introduction To Computer Architecture*. Chicago: Science Research Associates, Inc., 1975.

[Sunshine, 1976]
C. A. Sunshine, Factors In Interprocess Communication Protocol Efficiency For Computer Networks. *NCC Proceedings*, Vol. 45, Montvale, N. J.: AFIPS Press, 1976. Pp. 571-576.

[Sussman, 1973]
G. J. Sussman, *A Computational Model Of Skill Acquisition*. MIT-AI-TR-297, MIT, August 1973.

[Teitelman, 1975]
W. Teitelman, *INTERLISP Reference Manual*. Xerox Palo Alto Research Center, Palo Alto, Ca., December 1975.

[Vilenkin, 1971]
N. A. Vilenkin, *Combinatorics* (A. Shenitzer and S. Shenitzer, trans.). New York: Academic Press, 1971.

[Wecker, 1974]
S. Wecker, A Design For A Multiple Processor Operating Environment. *Conference Digest, IEEE COMPCON Spring*, 1974, pp. 143-146.

[Wicklegren, 1974]
W. A. Wicklegren, *How To Solve Problems*. San Francisco: W. H. Freeman, 1974.

[Widdoes, 1976]
L. C. Widdoes Jr., Architectural Considerations For General Purpose Multiprocessors. *Conference Digest, IEEE COMPCON Fall*, 1976, pp. 251-254.

[Wulf, 1972]
W. A. Wulf and C. G. Bell, C.mmp - A Multi-Mini-Processor, *FJCC Proceedings*, Vol. 41, Montvale, N. J.: AFIPS Press, 1972. Pp. 765-777.

Author Index

[Anderson, 1975] 16

[Baer, 1973] 2, 4, 12, 14
[Barrow, 1976] 13
[Berliner, 1973] 21, 91
[Bloch, 1978] 2
[Bobrow, 1977] 64
[Bonnet, 1978] 89
[Bowdon, 1972] 4, 13, 93
[Brinch Hansen, 1973] 80
[Brooks, 1975] 16
[Buchanan, 1978] 1, 21, 55

[Carhart, 1976] 1
[Carr, 1970] 85
[Chu, 1976] 2
[Connell, 1976] 14

[Dahl, 1968] 64
[Davis, 1976] 101
[Davis, 1977a] 59, 69, 72, 119
[Davis, 1977b] 75
[Davis, 1977c] 66, 76, 119
[Duda, 1978] 119

[Engelmore, 1977] 63
[Erman, 1975] 9, 63, 69, 70
[Ernst, 1969] 9, 117

[Faggin, 1978] 2, 12
[Farber, 1972] 55, 56, 77
[Feigenbaum, 1977] 119
[Feldman, 1977] 100
[Fennell, 1975] 14
[Fikes, 1971] 9, 118
[Fikes, 1972] 118
[Floyd, 1967] 25
[Flynn, 1972] 15
[Foster, 1976] 13
[Fuller, 1976] 19

[Galbraith, 1974] 16, 55
[Garcia-Molina, 1978] 18
[Gonzalez, 1972] 55

[Green, 1975] 2, 53

[Harris, 1977] 57
[Hayes-Roth, 1977] 55
[Heart, 1972] 85
[Hewitt, 1971] 4
[Hewitt, 1972] 9, 69
[Hewitt, 1973] 78
[Hewitt, 1975] 70
[Hewitt, 1977a] 4, 69, 100
[Hewitt, 1977b] 69

[Jensen, 1975] 13

[Kahn, 1972] 2, 13
[Kahn, 1975] 31
[Kaplan, 1973] 119
[Kaplan, 1975] 120
[Kimbleton, 1975] 2
[Knuth, 1975] 21
[Knutsen, 1977] 17
[Kowalski, 1970] 105
[Kuck, 1975] 11

[Lenat, 1975a] 71
[Lenat, 1975b] 4, 9, 69, 71
[Lenat, 1976] 69, 74
[Lesser, 1975a] 4
[Lesser, 1975b] 16
[Lesser, 1977] 70, 71, 76
[Lesser, 1978] 89, 100

[McDermott, 1974] 69, 73
[Metcalfe, 1976] 16, 85
[Minsky, 1971] 12
[Minsky, 1972] 91

[Newell, 1963] 12
[Nii, 1978] 14, 31, 33, 63
[Nilsson, 1971] 13, 24, 105
[Noyce, 1976] 2
[Noyce, 1977] 1

[Reddy, 1973] 13

[Reddy, 1975] 12, 70
[Roberts, 1970] 13

[Sacerdoti, 1974] 118
[Sacerdoti, 1975] 79, 118
[Samuel, 1967] 21
[Shortliffe, 1976] 9, 75, 119
[Simon, 1969] 2, 99
[Simon, 1976] 103
[Smith, 1977] 56
[Smith, 1978a] 31
[Smith, 1978b] 56
[Stone, 1975] 15
[Sunshine, 1976] 85
[Sussman, 1973] 14

[Teitelman, 1975] 121

[Vilenkin, 1971] 111, 112

[Wecker, 1974] 54
[Wicklegren, 1974] 14
[Widdoes, 1976] 14
[Wulf, 1972] 19