SPECIFYING A DOMAIN SPECIFIC LANGUAGE

FOR COOPERATIVE ROBOTICS

by

CURTIS RAY WELBORN, B.B.A., M.S.

A DISSERTATION

IN

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

DOCTOR OF PHILOSOPHY

Approved

Dan Cooke
Chairperson of the Committee

Nelson Rushton

Eric Sinzinger

Richard Watson

Accepted

John Borrelli
Dean of the Graduate School

December, 2005

ACKNOWLEDGEMENTS

I would like to first thank the people most responsible for me getting my Ph.D.: my family. My wife and children uprooted their lives to allow me to pursue a 14-year-old dream. I hope I have always kept happiness and sweetness in the family. To Dr. Daniel Cooke, thank you for getting me through this process and for always finding a way to keep me funded. I want to thank my dissertation committee: Dr. Eric Sinzinger, Dr. Nelson Ruston, and Dr. Richard Watson. I would also like to think all the professors at Texas Tech that contributed to my education, in particular Dr. Noe Lopez-Benitez, Dr. Larry Pyeatt, and Dr. Michael Gelfond. To Michael Helm, thank you for always being a friend. I am sorry I will not be around during your journey. And to my good friend Frank Roessler, "It's all your fault."

TABLE OF CONTENTS

ABSTRACT

The field of cooperative robotics focuses on research issues related to getting a team of distributed robots to work together to solve a problem. This research specifies a programming language that addresses complexity issues inherent in developing a cooperative robotic application by easing the burden of implementing a distributed application and by helping the developer organize the application such that tractable communication is maintained as the number of cooperative robots grows. The burden of developing a distributed application is eased by hiding within the programming language's execution model the code that performs interprocess communication, which is normally part of the programmer's obligation to design and implement. An application is organized using a well-known strategy[1] in operating systems--one that is used to help ensure modular system designs. Using this strategy, the programming language imposes a system architecture on an application that maintains tractable communication between robots by eliminating the need for a complex distributed communication strategy.

[1] Maintaining separation between mechanism and policy.

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Some problems are better solved by a team than by an individual. A team may be better than an individual for a variety of reasons. Sometimes coordinating the activities of simple, specialized agents may seem more straightforward than having a single, highly complicated agent performing all aspects of a problem solution. Multi-agent systems may prove more robust than a single agent system because the multi-agent system may be able to overcome the loss of an agent, whereas the single agent system cannot. Whatever the motivation may be, when multiple agents are to be coordinated, the complexity of the solution and the performance of the solution are pushed into the coordinating processes and their supporting communication. The field of cooperative robotics focuses on research issues related to getting teams of robots to work together to solve problems. To support the next generation of cooperative robotic applications, alternative models of how to program these applications must be devised.

Communication vs. Cooperation

Communication complexity for a distributed system is computed by counting the number of messages exchanged to complete a task. The communication complexity for N distributed agents to share information with each other is bounded at N^2 (Klavins, 2002). Yet, N^2 only sets an upper limit on the number of messages needed for N agents

to communicate with each other. This does not bound the number of messages that are required to get the same N agents to cooperate to complete a task. As the number of members in the group increases, each member of the group must allocate more and more time just to read and respond to the incoming messages. At some point, regardless of the communication bandwidth or raw processing power of each individual, the sheer volume of messages will overwhelm them. In cooperative robotics, broadcasting messages to all members of a team is a widely used method when there is no leader. The difference between communicating and cooperating will be illustrated using the Dining Out Problem.

Dining Out Problem

Five friends currently located in five different locations want to have dinner together, but they have yet to select a restaurant. All the friends have an equal standing in the group and as such no single person is the group leader. Because there is no leader among the friends, they all send emails to each other (broadcast) with their restaurant choice. As luck would have it, each friend chooses a different restaurant. At this point, each friend has received four messages, and a total of twenty messages have been sent (20 messages = 5 friends * 4 messages sent per friend). After the twenty messages have been exchanged, each friend now has full knowledge of the intended action (where to eat dinner) of every other friend, but as a group the friends are no closer to making a common decision than they were before the emails were sent. So, even though the friends are communicating, they are not cooperating to select a restaurant.

The Problem

This research will present work on a domain specific programming language for cooperative robotics problems. While the language, known as L, may be applicable to distributed computing and programming in general, the focus of this research is on cooperative robotics. The objective of this research is to address issues of complexity in the development of cooperative robotic applications. The specific areas of complexity to be addressed are:

- to reduce the complexity of developing a distributed application,
- to maintain tractable communication complexity among cooperating components.

The specific results of this research will be:

- a programming model,
- an execution model,
- robotic capability primitives,
- example applications written in L.

The programming model will define the syntax and semantics of the programming language, any abstractions the programming language uses, and the specific obligations or responsibilities a programmer has when programming in the language. The execution model will define how the programming language is implemented and how it differs from more conventional programming languages.

How the Problem will be Solved

The programming model introduces two constructs to an object-based[2] language, known as L, that allow the programmer to state nothing beyond a specification of the communication behavior of a system. The code that performs distributed communication, which is normally part of the programmer's obligation to design and implement, is hidden in the programming language's execution model. In other words, determining how distributed communication will be implemented is not part of the mental load of a developer. Using L, there is no difference in how a developer creates a distributed application when compared to the development of a stand-alone application. The programming model imposes a system architecture that can implicitly support distributed communication. This system architecture is based upon a well-known strategy[3] in operating systems--one that is used to help ensure modular system designs. Imposing this system architecture allows tractable communication complexity to be maintained between distributed components by eliminating the need for a complex distributed communication strategy. L employs a strategy whereby a computation is migrated to the location of data (computational mobility) rather than data being passed to the location of a distributed computation (message passing). Two challenges associated with using computational mobility (Milojičić, Douglis, and Wheeler, 1999) are:

1. the intermittent disconnect of wireless communication technology,

---

[2] An object-based language supports the creation of classes and objects but does not support inheritance.

[3] Maintaining separation between mechanism and policy.

2. the bandwidth of wireless communication technology is generally orders of magnitude less than line-based networks.

To address the first challenge, a system must be made as autonomous (Forman and Zahorjan, 1994) as possible so that it can continue to operate when it has lost access to the network. The greater the autonomy of the system, the longer the system will be able to operate effectively while disconnected from the network. The second challenge is being addressed by the continual evolution of wireless communication technology:

> …compared to today's best wireline networks, wireless networks are low-bandwidth. But compared to yesterday's networks, wireless is competitive, and it is improving steadily. …In the near future, broadband wireless networking may even supplant wired telephony and cable modems in some environments. (Milojičić et al., 1999, p. 268)

The execution model specifies target code produced by the compiler, the operation of the run-time environment for L, and overhead incurred by the run-time environment to support the programming model. A proof of concept (prototype) compiler is implemented to ensure that the execution model can be implemented and deployed on conventional computer hardware and to allow example applications to be written.

The robotics capability primitives section gives example primitives that organizes robotic capabilities to make their use more effective in L. As the primitives are but one example of how the robotic capabilities can be organized, guidelines for the continuted development of primitives are presented.

How the Solution will be Evaluated

Example applications written in L will be implemented to empirically validate concepts of the programming/execution model. The language will be used to construct the primitive models of distributed architectures: in particular, the Client-Server Model, the Multiple-Tier Server Model, and the Peer-to-Peer Model. The example programs will show that even though the programming language advertises no conventional interprocess communication constructs (e.g., sockets, RMI) for writing a distributed application, distributed applications with functionality that map to existing distributed architecture models can still be created and deployed on conventional hardware. In addition to implementing various distributed architecture model programs, this research will present how a robotic control system, based on the hybrid deliberative/reactive strategy, could be implemented using L.

The Vision

The long-term vision for this research is that it will play an integral role in developing large teams of heterogeneous cooperating robots where each team member is a functional robot capable of independent actions, though the overall scope and capabilities of an individual team member may be rather limited. The team as a whole may be highly functional and capable, encompassing massive amounts of sensing and processing power that can be brought to bear on a problem. A prime example of this vision would be a planetary explorer team (Figure 1.1) composed of high-altitude imaging satellites, low-altitude aerial recognizance vehicles, planetary rovers, sensor

arrays, and various manipulators all working together to complete a mission. The physical distances between team members, the diversity of the team, and the remoteness of the team's location all combine to create an environment where autonomous cooperating robotics are essential.

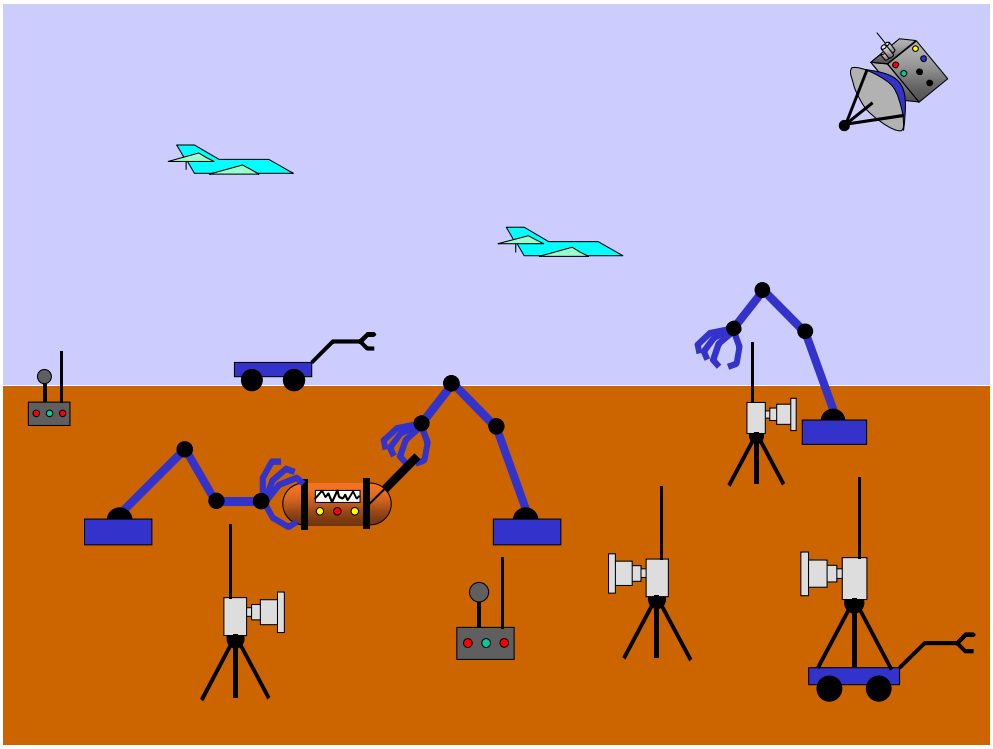


Figure 1.1. Planetary Explorer Team.

Document Overview

Chapter I gives an overview of the problems this research will address and some introductory background material. Chapter II is a literature review on various subject areas in which knowledge was required to complete this research, though the focus of the literature review is on robotics and distributed computing. Chapter III provides an overview of the principles and high-level design choices used to conduct this research. Chapter IV reports the results of this research. Chapter V provides conclusions and indicates areas of future research. The appendixes provide detailed examples or extended coverage on various topics covered in Chapters I through V.

CHAPTER II

LITERATURE REVIEW

Robotics

The field of robotics is often categorized into the sub-fields of industrial robotics and artificially intelligent (AI-based) robotics. Both industrial and AI-based robotics were preceded by mechanically controlled manipulators. Mechanically controlled manipulators were used during World War II. A mechanical manipulator allowed a scientist to work with nuclear material without being physically in contact with the nuclear material. A mechanically controlled manipulator is just a set of mechanical linkages that allow the user to operate one end of the manipulator while moving the other end. Telemanipulators were the next evolution in mechanically controlled manipulators, with much of this work beginning around the mid-1960's. Telemanipulators replaced the mechanical linkages with electronic signals that could be transmitted as radio, optical, or satellite communication. Telemanipulators have the capability to operate over vast distances. Yet it is exactly these vast distances that reduce the effectiveness of telemanipulators. As the distance between the human and the remotely controlled manipulator increases, the complexity of completing a task increases due to the large delay in communication. These delays can result in cognitive fatigue of the operator.

The field of industrial robotics, as its name implies, is primarily concerned with the use of multi-function mechanical manipulators (robot arms) in an automated industrial/factory setting. The automation of a factory floor requires precise movements

to be performed thousands or even millions of time by a mechanical manipulator. The speed and precision of this work requires a highly tuned control system. Control systems such as these will work effectively only in environments that can be precisely controlled. For example, in many factory environments, the temperature, the humidity, and the layout of the factory floor can all be controlled with a high degree of accuracy. Such environments are in direct contrast to the intended environment of AI-based robotics.

AI-based robotics began in the aerospace industry as a way of developing planetary rovers that could explore distant planets without the need for human interaction (Murphy, 1998). AI-based robotics uses technology from the field of AI to construct robots that can operate autonomously in stochastic environments. Creating such systems reduces the burden on the human operators and makes the robot more robust.

Hierarchical Strategy

One of the first and most well known AI-based robots was Shakey, developed at Stanford Research Institute in the late 1960's. Shakey used the STRIPS planner to construct a plan to allow the robot to navigate between rooms. STRIPS is a theorem-prover using first-order logic; it is classified as being in the hierarchical strategy (Murphy, 1998). The hierarchical strategy is characterized by systems that perform the following steps in a loop: sense, plan, act.

In the sense step, attributes about the environment are perceived by the system. A robotics system uses sensors (e.g., temperature, humidity, distance) to perceive information about the environment. Because a physical robot is part of the environment,

information about the robot itself may be perceived. Information gathered during sensing will be used to build or update a world model. The world model can be used to represent objects from the environment, the state of an object, relationships between objects, goals of the system, and constraints or rules that govern how objects can be used. Objects that might be represented in a world model are the robot and obstacles in the environment. The spatial relationship between the robot and the obstacles would be a useful relationship to represent in the world model. A location on a map could be the goal of the system. Constraints could indicate the battery life of the robot or what type of terrain the robot can cross.

In the plan step, a planner uses information in the world model to construct a plan, which is a sequence of actions that when followed will accomplish a system goal. The planner used in such an environment needs to use some form of nonmonotonic logic because the environment changes over time. A fact that is true at one moment of time (e.g., the hallway is empty) may not be true the next moment (e.g., there are people in the hallway).

In the act step, the actions in the plan produced by the planner are executed to accomplish the system goal. The main difficulty in using the hierarchical strategy successfully was the slow reaction time of systems built using the strategy. While modern high-speed computers can address some of the speed difficulties encountered in early systems, the problem, however, was not solely with the computers but the strategy itself. Humans are intelligent creatures capable of perceiving their environment and devising complex plans to accomplish great feats (e.g., building the Hoover Dam,

walking on the moon). Yet, when a person touches something hot, they instinctively react by pulling away. The reaction to pull away is a spinal reflex and occurs before the brain ever perceives pain. If the body treated this as a cranial reflex and waited until the brain perceived pain, the burn injury would be significantly worse because the reaction time would be significantly slower.

Reactive Strategy

The hierarchical strategy was supplanted by the reactive strategy, which grew directly out of work by Rodney Brooks in the late 1980's (Brooks, 1986). The reactive strategy is characterized by systems that create concurrent behaviors by pairing sensing and actions directly together. The reactive strategy does not make use of an explicit world model to plan actions; rather the behaviors are made to interact with the environment and each other in very fast control loops that allow the system to respond quickly to changes in the environment. Thus the major difference between the hierarchical strategy and the reactive strategy is the reactive strategy removes the plan step and is left with only a sense, act loop. This fast response to environmental changes was a driving force in the adoption of the reactive strategy. Systems built using the hierarchical strategy often suffered from extremely long lag times between when the environment would be sensed and when the system would act. These lag times could be so great that the environment would change again before the system could complete its response to a prior change. The subsumption architecture (Brooks, 1986) is the best known architecture in the reactive strategy. The subsumption architecture was originally

implemented in hardware. This close coupling with hardware is evident in the three constructs of subsumption: a control module, an input line suppression operation, and an output line inhibition operation.

A control module is defined as a finite state machine with an initial state of NIL. A control module can be set back to the NIL state at any time by applying a signal to the reset input line of the module. In addition to the reset input line, a control module can have additional input and output lines. The input and output lines are sometimes referred to as input and output wires in the original paper (Brooks, 1986). For each input line to a control module, the module will maintain a buffer, which contains the data of the most recently received signal from the input line. When a new signal is received on an input line, old data in the buffer is overwritten. A control module can transmit data to other control modules via an output line.

A suppression operation can be attached to a data input line. The suppression operation requires an additional control input line, which signals when to perform suppression. When a data input signal is suppressed, the data from the control line is used in place of the original data input line. The data from the control signal will continue to suppress (replace) the data of the data signal for a specified period of time.

An inhibition operation can be attached to a data output line. The inhibition operation requires an additional control input line, which signals when to perform inhibition. When a data output signal is inhibited, no data can be sent on the data output line for the specified period of time. The Subsumption Control Module (Figure 2.1) shows various input and output lines to a control module. The figure shows an input line

being suppressed for a period of 5 cycles (a cycle is dependent upon the clock speed), and an output line being inhibited for 7 cycles.

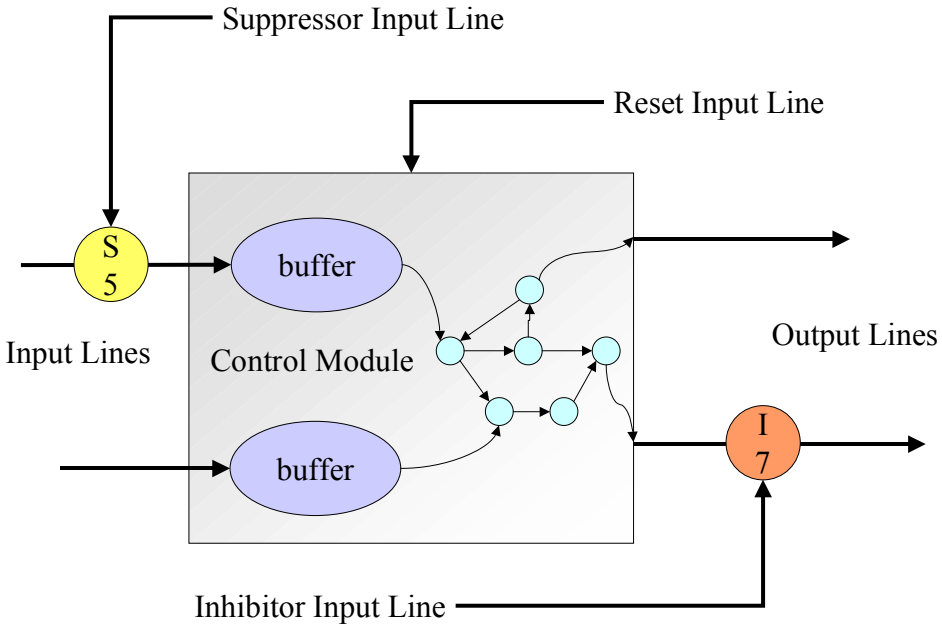


Figure 2.1. Subsumption Control Module.

Multiple control modules are connected together to form a control system. A subsumption control system is constructed in layers. The lowest layer is used to implement basic motor behaviors, which are used to move the robot around in an environment. Ever increasing levels of intelligent behavior are implemented in each subsequent (higher) layer of the architecture. Organizationally, only a higher layer (more intelligent behavior) should ever inhibit or suppress a signal at a lower layer (less intelligent behavior).

Deliberative/Reactive Strategy

The hierarchical strategy can perform high-level reasoning using its planner, yet it is slow to react to changes in the environment. The reactive strategy is quick to react to changes in the environment, yet it is difficult to get it to reason at a level above animal instinct. As a result, the deliberative/reactive strategy was created to fuse the best features from each strategy (planning from the hierarchical strategy and fast reactions of the reactive strategy) into a single strategy. Systems employing the deliberative/reactive strategy often use a three-tier architecture (Gat, 1998) (Figure 2.2):

- Tier-One utilizes multiple threads to implement control loops that can sense the environment and react to changes very quickly. The first tier generally contains no state information.
- Tier-Two, often referred to as a sequencer, contains state information about past events. The second tier uses information in a plan to enable or disable various control loops in Tier-One. By enabling and disabling various control loops, Tier Two directs the system to perform a task. Tier-Two can also gather information about sensed events from Tier-One and provide this information to Tier-Three.
- Tier-Three is the deliberative layer of the architecture and employs a planner to specify the actions the system is to attempt to take in the future. The planner is provided with sensory information from Tier-One, which it uses to build or update a world model.

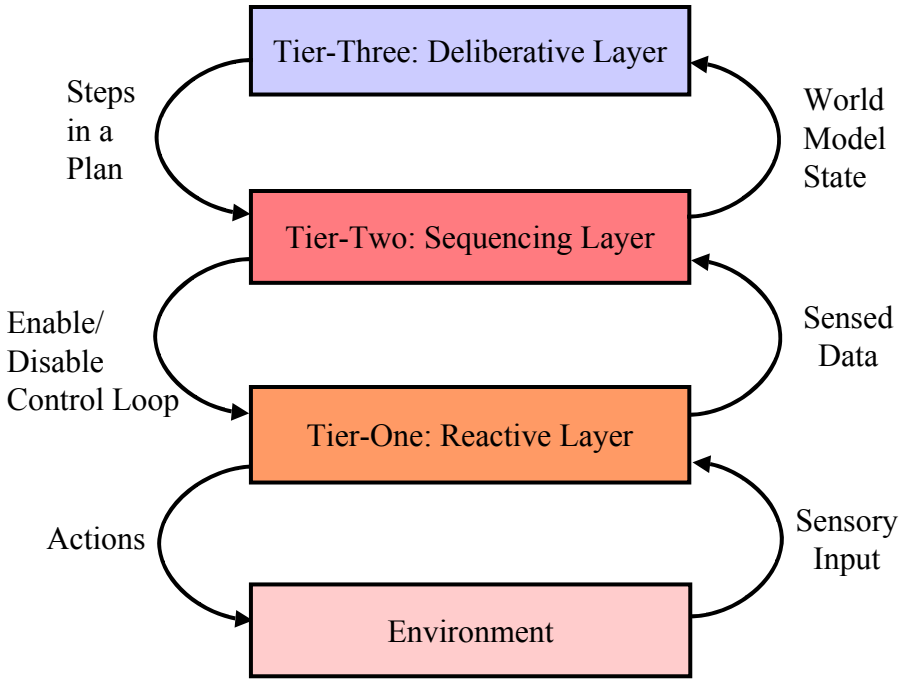


Figure 2.2. Three-Tier Architecture Information Flow.

Model-Based Programming

Model-based programming, which was first introduced in Williams and Nayak (1996), is a particular approach to developing a robotic control system that extends the deliberative/reactive strategy. Model-based programming was used to develop Remote Agent, an autonomous agent architecture deployed on-board the spacecraft Deep Space 1 as part of the New Millennium Program (Muscettola, Nayak, Pell, and Williams, 1998). "The New Millennium Program focuses on testing high-risk, advanced technologies in space with low-cost flights. Any scientific data returned during the missions are bonuses, as the real science is in the future missions that are enabled by New Millennium's technologies" (http://nmp.jpl.nasa.gov/ds1/sci/index.html). Two of the more novel

aspects of model-based programming are the use of qualitative reasoning and synchronous programming language features.

Qualitative Reasoning

Qualitative reasoning is concerned with capturing the intuition or commonsense understanding that humans use to reason about physical systems. These systems use symbolic rules or quantitative equations to model a physical system, though the models tend to be greatly simplified from formal mathematical models that accurately depict all facets of the physical system. The interested reader is directed to Weld and de Kleer (1990) for a more detailed coverage of this topic.

Synchronous Languages

A synchronous language divides time into discrete instances and must support one of the implementation models (Benveniste et al., 2003) shown in Table 2.1.

Table 2.1. Implementation Models.

Event-Driven Implementation Model	*Sample-Driven Implementation Model*
Initialize Memory For each input event do Compute Outputs Update Memory end	Initialize Memory For each clock tick do Read Inputs Computer Outputs Update Memory end

A salient feature of synchronous languages is that they are both concurrent and deterministic. While this may seem a contradiction, synchronous languages use extensive verification techniques at compile time to ensure that concurrent programs are deterministic. The interested reader is directed to Benveniste et al. (2003), Berry (2004), and Halbwachs (1993) for more information on this topic. The basic intuition used to describe how a synchronous language works comes from acyclic combinational circuits (Benveniste et al., 2003; Berry, 1988). Figure 2.3 shows an acyclic combinational circuit composed of three inputs (I, J, and K) and one output (Z). There are two things to note about this simple acyclic combinational circuit. The first is that because the circuit is acyclic, "the [Boolean equations that define the circuit] can be ordered in such a way that any variable only depends on previously defined [input variables or equations]" (Berry, 2004, p. 6). The second is that:

> Boolean values are represented by voltages, say 0V and 5V, and wires and gates have bounded propagation delays. If the input voltages are kept stable to some Boolean voltages, then, after some predictable time, the output voltages stabilize at the right Boolean voltages. This is a physical fact, not a mathematical theorem. Since the number of input configurations is finite, it is possible to determine a maximum output stabilization time δ valid for all inputs. (Berry, 2004, p. 7)

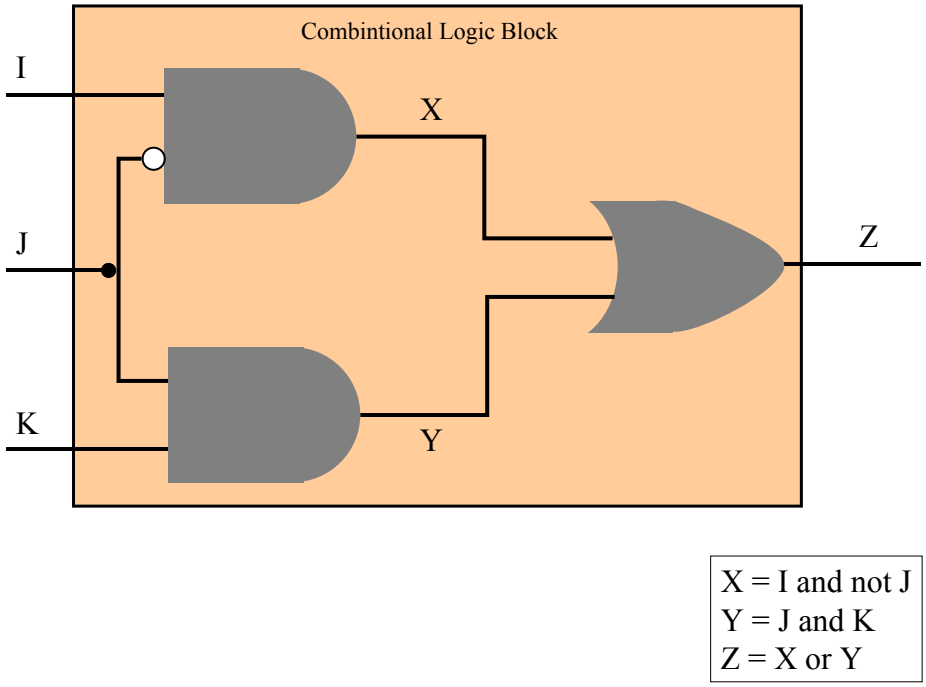


Figure 2.3. Acyclic Combinational Circuit.

Even though the combinational logic block has concurrent elements, it can be made to produce a deterministic result if all inputs are changed synchronously and all outputs are only read after the stabilization time δ. The logic block can be viewed as a function, where the inputting of data to the function and the reading of the result are synchronized with a clock. Once the stabilization time δ is known for various functions, the functions can be combined in a sequence, as in Figure 2.4 (a). The total propagation delay δ' for the sequence is the sum of the delays for each individual function, $\delta' = \sum_{k=0}^{n} \delta^k$. In this way, larger and more complex circuits can be created from simpler circuits, with the overall result of the circuits being deterministic as long as the system

progresses as successive atomic reactions that are synchronized with a clock. Figure 2.4 (b) illustrates how an acyclic circuit can be made to loop by using some state holding element (a register). As before, the input of data to the circuit must be synchronized with a clock, and the outputs of the circuit must not be read until after the stabilization time δ. This type of loop is known as a non-zero-delay loop, because the loop is forced to wait for the synchronization of the clock before it can repeat the loop. This basic strategy of computing a stabilization time for functions and then forcing the functions to synchronize with the clock is used to compile and then execute synchronous languages. Synchronizing blocks of code with the clock ensures that the blocks of code are made non-interfering.

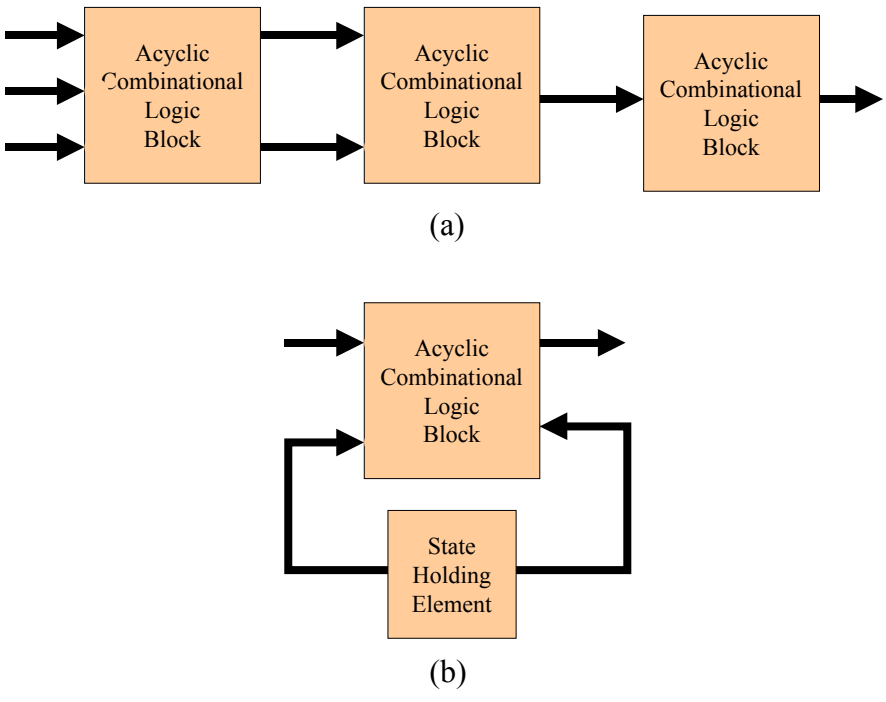


Figure 2.4. Connecting Blocks.

Remote Agent

The Remote Agent (RA) architecture is shown embedded within flight software in Figure 2.5 (Muscettola et al., 1998). Grouping the following components into layers, the resemblance between embedded Remote Agent architecture and the three-tier architecture approach from the deliberative/reactive strategy can be seen:

- Deliberative Layer: Mission Manager, Planner/Scheduler, and Planning Experts System
- Sequencing Layer: Smart Executive and Mode Identification and Reconfiguration
- Reactive Layer: Real-Time Control System and Monitors
- Environment Layer: Ground System and Flight Hardware

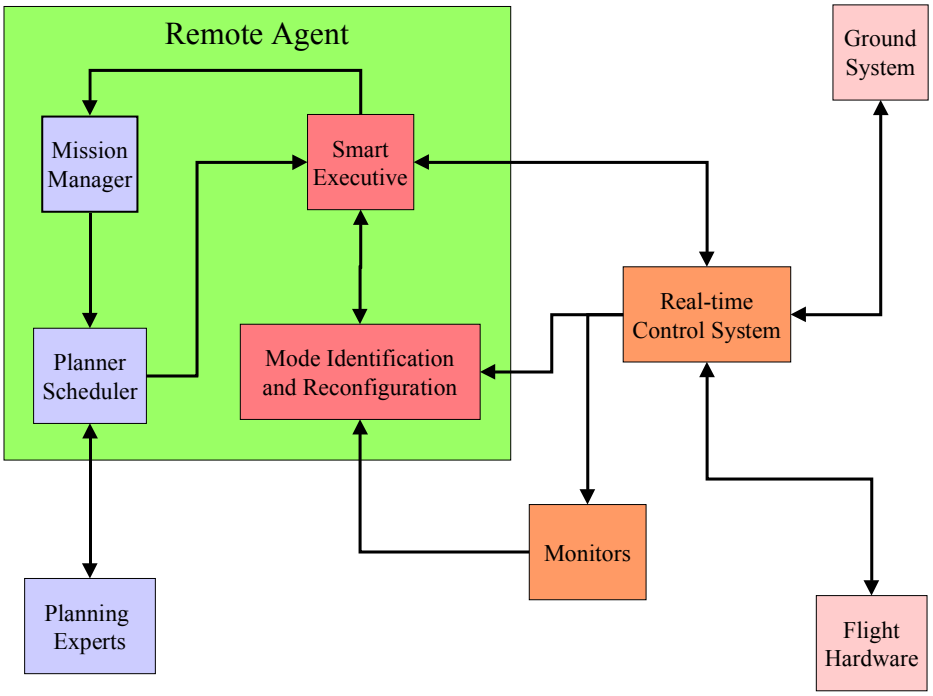


Figure 2.5. Embedded Remote Agent Architecture.

While these groups seem valid, the clean divisions between layer functionality presented in the Deliberative/Reactive Strategy section are not as clean and well defined in Deep Space 1's embedded architecture. This statement is not a criticism, but a statement of fact and a reflection of the difference between theory and practice. Even compilers, which have well defined phases, will have modules or functions that mix code from different phases (e.g., syntax analysis and semantic processing). The components of the RA architecture are the Mission Manger, Planner/Scheduler, Smart Executive, and Mode Identification and Reconfiguration.

The Mission Manager, the Planner/Scheduler, and the Planning Experts work together to perform planning for Deep Space 1. The Planner/Scheduler is used to create

short-term plans to accomplish near-term objectives (a.k.a., periodic planning). The Mission Manager activates the Planner/Scheduler whenever the Smart Executive needs a new plan. The Mission Manager monitors the plans created by the Planner/Scheduler to ensure that short-term plans will not be created that interfere with long-term mission objectives. "Other on-board software systems, called planning experts, participate in the planning process by requesting new goals or answering questions for [the Planner/Scheduler]" (Muscettola et al., 1998, p. 10).

The Smart Executive is a sequencer that executes plans from the Planner/Scheduler "…by decomposing high-level activities in the plan into commands to the real-time [control] system…" (Muscettola et al., 1998, p. 10). The Smart Executive uses feedback from the Real-Time Control System or Mode Identification and Reconfiguration to implement a closed-loop control system that quickly adapts to changes in Deep Space 1's system state. Feedback from the Real-Time Control System consists of observed state variable values, whereas feedback from Mode Identification and Reconfiguration consists of inferred 'hidden' state variable values.

Mode Identification and Reconfiguration consists of a qualitative model (Mode Identification) and a reactive planner (Mode Reconfiguration). Mode Identification uses a qualitative model based on propositional logic to infer the value of a 'hidden' state variable. A 'hidden' state variable is a state variable that cannot be directly observed by any sensor on-board Deep Space 1. Thus, the qualitative model predicts or infers the value of these 'hidden' state variables using observed state variables and the current commanding sequence as inputs. These 'hidden' state variables generally represent high-

level configuration information about the spacecraft, which maps to a user's intuition of how the spacecraft operates. As such, 'hidden' state variables are often used to write the control programs for Deep Space 1. "Thinking in terms of more abstract hidden states makes the task of writing the control program much easier and avoids the error-prone process of reasoning through low-level system interactions" (Williams et al., 2003, pg. 3). Mode Reconfiguration is primarily used to generate very short-term plans, which recover from failures or move the spacecraft into a safe state after a failure has occurred.

RMPL

The Reactive Model-Based Programming Language (RMPL) is a generalization of concepts extracted from Remote Agent. RMPL uses a programming language for writing control programs that is very similar to the synchronous language Esteral. A key difference between an Esteral control program and a RMPL control program is how the control programs interact with hardware. An Esteral control program will directly interact with hardware by reading sensor values and commanding actuators. An RMPL control program interacts with hardware through the Deductive Controller. The Deductive Controller performs a role very similar to Mode Identification and Reconfiguration in Remote Agent. In RMPL, a Partially Observable Markov Decision Process (POMDP) is used in conjunction with qualitative reasoning to estimate the most likely value of a 'hidden' state variable. Temporal constraints can be specified for how long a task is to take. These constraints are compiled with a control program. When the Titan Model-Based Executive executes the program, these temporal constraints are used

to guide the selection of a plan that can meet the desired goals of the system within specified time constraints.

Multi-Agent Systems

Cooperative robotics is a sub-field of Multi-Agent Systems (MAS). MAS research includes work on both communicating and non-communicating agents. While this research is primarily concerned with communicating agents, looking at the larger field of MAS will provide a broader perspective before diving exclusively into the field of cooperative robotics. Some of the major advantages of using MAS are (Balch and Parker, 2002):

- the parallelism obtained by distributing various tasks of a problem to multiple processors,
- the robustness gained by having multiple redundant agents,
- the scalability inherent in developing a system made of multiple simpler agents than one large complex agent,
- the simpler programming needed for the less complex agents, and
- the physical distribution of the agents when real robots are used.

Team Diversity

Team diversity can be an important characteristic when looking at MAS. When all the members of a MAS have the same capabilities, then a single control strategy can be used to coordinate the members of the MAS because all members are equally

interchangeable. The only real difference between agents in a homogeneous team (all members the same) is the physical location of each agent.

As team diversity increases, the complexity of the control strategy needed to coordinate the MAS also increases because team members are not interchangeable. Yet it is exactly these same differences that allow a heterogeneous team (mixture of team members) to attempt to solve problems that a homogeneous team has no hope of solving.

The diversity of teams has been measured using social entropy (Bailey, 1990), which is adapted from information entropy (Shannon, 1949). A simple social entropy equation for computing the diversity of robot teams is as follows:

$H(X) = -\sum_{i=1}^{M} p_i \log_2 (p_i)$ where M is the number of different robotic groups and p_i is the proportion, $p_i = g_i \Big/ \sum_{j=1}^{N} g_j$, of robots in the i^{th} group. Figure 2.6 has five teams labeled A, B, C, D and E. The social entropy number for each team is as follows:

Team A: $0.0 = -(1 * \log_2(1))$

Team B: $0.468 = -((.1 * \log_2(.1)) + (.9 * \log_2(.9)))$

Team C: $1.0 = -((1/2 * \log_2(1/2)) + (1/2 * \log_2(1/2)))$

Team D: $0.708 = -((.2 * \log_2(.2)) + (.8 * \log_2(.8)))$

Team E: $3.322 = -((.1 * \log_2(.1)) * 10)$

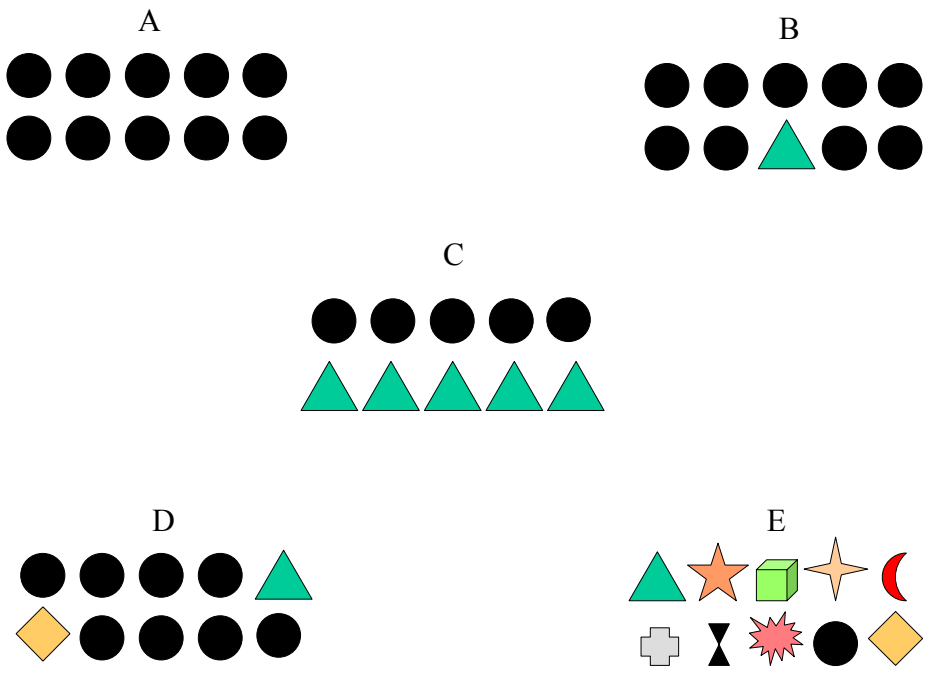


Figure 2.6. Social Entropy Teams.

The social entropy number provides a quantitative measure for comparing the diversity of different teams. A team with no diversity (Team A) has a social entropy of 0.0, whereas a team with a 50-50 split between two groups (Team C) has a social entropy of 1.0. Social entropy numbers cannot be normalized and placed on a scale between 0.0 and 1.0 because there is no upper bound on social entropy. A team with ten members and each member being different (Team E) has a higher social entropy than a team composed of two equal groups (Team C), yet a team with 100 members and each member being different would be even more diverse than the team with 10 members.

A Heterogeneous Cooperative Team

Chapter 13 from Balch and Parker (2002) by Hershberger, Simmons, Singh, Ramos and Smith presents an interesting problem, which nicely explains the need for heterogeneous cooperative robotics. The problem is to assemble a large-scale structure, such as a building, using robots. The problem requires robots with very different capabilities. The task of lifting the structural beams requires a robot of considerable size and strength. The task of connecting the beams together, on the other hand, requires a robot of medium size with a great deal of physical dexterity. Rather than a single robot, the authors suggest the use of three cooperative robots capable of communicating with each other using Wavelan radio Ethernet:

- A crane, called Robocrane, which is twenty-feet high with six-degrees of freedom built by the National Institute of Standards and Technology (NIST).
- A mobile manipulator, called Bullwinkle, with stereo vision to avoid obstacles and a robotic arm with five-degrees of freedom for connecting the structural components.
- A roving eye, called Xavier, which is four-feet tall and two-feet in diameter with stereo cameras mounted on a pan-tilt unit used to watch over the placement of the structural components.

While the current implementation plans call for fixed teams of three robots working together, the authors have long-term plans to allow for dynamic teams where a single roving eye can assist multiple teams simply by being in a location where it can view both work sites.

Stigmergy

When no direct communication is possible between team members, indirect communication, known as stigmergy (Beckers, Holland, and Deneubourg, 1994) from the field of biology, is still possible because the team members all operate in the same environment. Stigmergy is the concept of an agent altering the environment to influence the behavior of another agent. Stigmergy can be either active or passive. Active stigmergy occurs when an agent changes the environment in such a way that another agent can sense the change in the environment. An ant leaving a pheromone trail that other ants can sense and follow is an example of an active stigmery. Active stigmergy is an important topic in puck clustering work on swarm robotics. The idea behind puck clustering is based on a cemetery creation behavior observed in ants. The behavior is believed to work as follows:

- When an ant dies, it releases a pheromone, which causes other ants to pick up the dead ant and carry it around and then drop it.
- Ants will tend to drop a dead ant in a location where the dead ant pheromone is stronger.
- Over time, a group of dead ants will get clustered into small piles.
- Over a longer period of time, the small piles of dead ants will be clustered into a single large cemetery.

Passive stigmergy occurs when an agent changes the environment is such a way that the change will affect another agent's behavior. An agent locking doors that intersect

with a hall can affect the behavior of other agents that enter the hall. This would be an example of passive stigmergy.

Direct Communication

Direct communication occurs when agents can send messages directly to one another. Two of the more common communication topologies are point-to-point and broadcast communication. Point-to-Point communication occurs when an agent addresses and communicates with one agent at a time. A program exhibits point-to-point communication when it opens a port to a specific machine address using a socket. Broadcast communication occurs when an agent sends a single message that is received by multiple agents. A program exhibits broadcast communication when it uses the broadcast option of IPC sockets. In addition to point-to-point and broadcast communication topologies, tree or graph topologies can also be supported.

Work has been done to reduce the complexity of communication in multi-robot systems (Klavins, 2002) that focuses on dynamically maintaining a graph topology that restricts communication to those agents that are spatially close to each other. By restricting the number of robots that can move at any one time and not communicating with robots that are stationary, the complexity of communication can be reduced from $O(n^2)$ to between $O(n)$ and $O(1)$ depending upon the assumptions that are made.

The control system of a robot used in MAS, whether communicating or non-communicating, could be developed using the hierarchical strategy, the reactive strategy, or the deliberative/reactive strategy (Table 2.2). Non-communicating robots would rely on behaviors that allow cooperation of multiple agents through some form of stigmery.

Communicating robots would directly send messages to other team members. In the case of the reactive strategy, an incoming message would just be treated as sensory data and an outgoing message as an action.

Table 2.2. Robotic Strategies Used in Multi-Agent Systems.

MAS	*Hierarchical Strategy*	*Reactive Strategy*	*Deliberative/Reactive Strategy*
Communicating	✓	✓	✓
Non-Communicating	✓	✓	✓

Mobile Code (a.k.a. Computational Mobility)

The concept of moving code from one computer to another is not a new concept, even if some of the things written since the introduction of Java and Java applets would make one assume this. The paper by Powell and Miller (Powell and Miller, 1983) covers their work in adding process migration to the DEMO/MP operating system. Three of the primary benefits for looking at mobile code are:

- Load Balancing. In load balancing, executing processes on a heavily loaded machine can be migrated to other lightly loaded machines within a network.
- Fault Tolerance. Here a machine with degraded or broken resources will migrate processes to a healthy machine within a network.

- Data Locality. In data locality, a process that needs resources (data) on machine A, but is currently executing on machine B, will be migrated from A to B to provide better data locality. Works such as Barak and Wheeler (1989) and Jul, Levy, Hutchinson, and Black (1988) address the performance benefits of process migration with respect to data locality.

From the perspective of cooperative robotics, load balancing plays little role in our interest in code mobility. However, fault tolerance and data locality are both interesting features from which cooperative robotics can benefit and will be touched on again later. Mobile code or computational mobility comes in two forms (Fuggetta, Picco, and Vigna, 1998):

- Weak mobility is characterized by a system that can migrate executable code to different locations on a network, but cannot migrate the execution state of a thread from a running program. The Java Virtual Machine (JVM) is an example of a system that employs weak mobility. Using the JVM, an applet can be downloaded from a remote server on a network and then executed locally by the JVM.
- Strong mobility is characterized by a system that can migrate both the executable code and execution state of a thread from a running program. The term "process migration" is an older term that can be used interchangeably with "strong mobility." A system needs to perform the following activities to support strong mobility:
    1. Suspend an executing thread or process that is to be migrated.

2. Capture the state of the executing thread or process.
3. Convert the captured execution state into network transmittable data.
4. Transmit the captured execution state to the new computational environment.
5. Receive the transmitted execution state.
6. Reconstruct the execution state of the thread or process from the transmitted data.
7. Restart the suspended thread or process in the new computational environment.

Operating System-Based Process Migration

Much of the early work on process migration (strong mobility) came from research in operating systems. MOSIX (Barak and Wheeler, 1989), Sprite (Douglis and Ousterhout, 1991), Charlotte (Artsy and Finkel, 1989), and the aforementioned DEMO/MP (Powell and Miller, 1983) are all examples of operating systems that support extensions for process migration. Support for process migration within the base operating system suffers from two major drawbacks. The first is the fact that organizations and individual machine owners seldom accept the basic idea that processes can migrate to and from their machines. This issue is not technical but social. Process owners want to know where their processes are executing and what resources they are being given; machine owners want to know who is using their resources and if they are being fairly compensated for the use. The second is that when process migration is

implemented in the base operating system, the code tends to be complex and difficult to maintain (Milojičić et al., 1999).

Non-Operating System-Based Process Migration

To address code complexity issues and the reluctance of individuals and organizations to use operating system-based process migration, the support for process migration can be moved out of the base operating system and into a user controllable package or language. The Condor package (Litzkow and Solomon, 1992) provides a migration mechanism for processes running under the UNIX operating system. An interesting aspect of Condor is that a shadow process is always left behind on the original machine to handle system calls. The Tui system (Smith and Hutchinson, 1998) is another type of package that provides a migration mechanism for programs written in ANSI-C. The novel aspect of Tui was that it allowed process migration between machines with different operating systems and hardware architectures.

Emerald is an example of a programming language and run-time system that implements object-level migration. Emerald is an object-based language. In general, the term "object-based languages" is used to refer to languages that allow objects to be defined but do not allow inheritance. In the case of Emerald, it was not obvious that the authors intended this use of the term, since Smalltalk was also referred to as an object-based language and it clearly supports inheritance. Given that the article was published in 1988, it is possible that the terms "object-oriented" and "object-based" were

interchangeable during this time frame. Emerald provides explicit constructs that allow objects to be migrated to new machines. The five object mobility constructs are:

- locate X. This construct will return the machine on which object X resides.
- move X to Y. This construct will move object X to machine Y.
- fix X at Y. This construct makes object X where it cannot be moved.
- unfix X. This construct reverses the effect of the fix construct and makes object X mobile following a fix.

Emerald was one of the first systems to investigate migrating objects vs. complete processes. While Emerald transparently handles the details of migration, it is the developer's responsibility to write the code that causes migration to occur.

Sumatra is another example of a programming language and run-time system that supports object-level migration. Sumatra is essentially the Java programming language with two additional programming language constructs (Ranganathan, Acharya, Sharma, and Saltz, 1997):

- Object-groups. An object-group is a dynamically created collection of objects that will allow migration at the same time to the same location. A programmer creates an object-group, then adds objects to it via the checkIn() method.
- Execution engines. An execution engine is an abstract notion of a Sumatra Virtual Machine. A programmer can migrate a collection of objects in an object-group by invoking the moveTo() method of an object group.

Sumatra, like Emerald, transparently handles the details of migration, yet, like Emerald, it is the responsibility of the developer to write the code that causes object migration to occur. In the case of the programming language L, the entire migration procedure is made transparent to the developer. There are no constructs that the developer must use to force an object (process) to migrate, nor special tools that must be used to compile a distributed application. Rather the run-time environment for L will determine both when and how to migrate an object.

Distributed Systems

There is no single commonly recognized definition of a distributed system, though the following characterization from Tanenbaum and van Steen (2002, p. 2) is a good starting point: "A distributed system is a collection of independent computers that appears to its users as a single coherent system. This definition has two aspects. The first one deals with hardware: the machines are autonomous. The second one deals with software: the users think they are dealing with a single system." Cooperative robotics as a field of research is inherently tied to research in distributed systems. Coulouris, Dolimore, and Kindberg (2001) have characterized distributed systems by the fact that they have resources that execute concurrently. These resources can fail independently of each other, and there is no global clock that can be used by all the resources to synchronize their actions. Now that distributed systems have been introduced, let us now look at some of the more important features and issues of distributed systems, particularly those that could affect cooperative robotics.

Middleware

Middleware is one or more programs that are intended to provide services that distributed systems require. Middleware derives it name from the fact that many of the services it provides deal with the exchange of information between distributed elements. In addition, middleware can provide common services to applications that are distributed across various hardware platforms. This allows the application to maintain independence from the operating system and network on which the application is deployed. CORBA is an example of a large middleware system that provides many services and is even language independent. Java RMI is a middleware system that is language dependent. One important feature that middleware will provide is marshalling/ unmarshalling.

Marshalling

Marshalling is a mechanism that will take a collection of data items and convert them into a form that is suitable for transmission over a network. Unmarshalling is a mechanism that will reconstruct data items from information transmitted over a network. The term "marshalling" is used to generically refer to the process of marshalling and unmarshalling data.

When a data item to be transmitted is one-dimensional, such as an integer or string, the only concern in marshalling and unmarshalling the data item is to deal correctly with machine byte order vs. network byte order. If the data item to be transmitted has a two-dimensional structure, such as a binary search tree, then additional

problems arise because the data must be converted to a one-dimensional structure before being transmitted over a network. The process of converting multi-dimension data into one-dimensional data is sometimes referred to as serialization.

Figure 2.7 shows a multi-dimensional structure (a binary search tree composed of names) being converted to a one-dimensional structure via a marshalling mechanism. The marshalling mechanism must capture the structure of the binary search tree in addition to the data of the tree. For marshalling to capture the structure of the tree, additional information must be encoded along with the data in the one-dimensional structure. The one-dimensional structure can now be transmitted over a network where unmarshalling will be used to reconstruct the original binary search tree using the data and binary search tree structure information encoded into the one-dimensional structure.

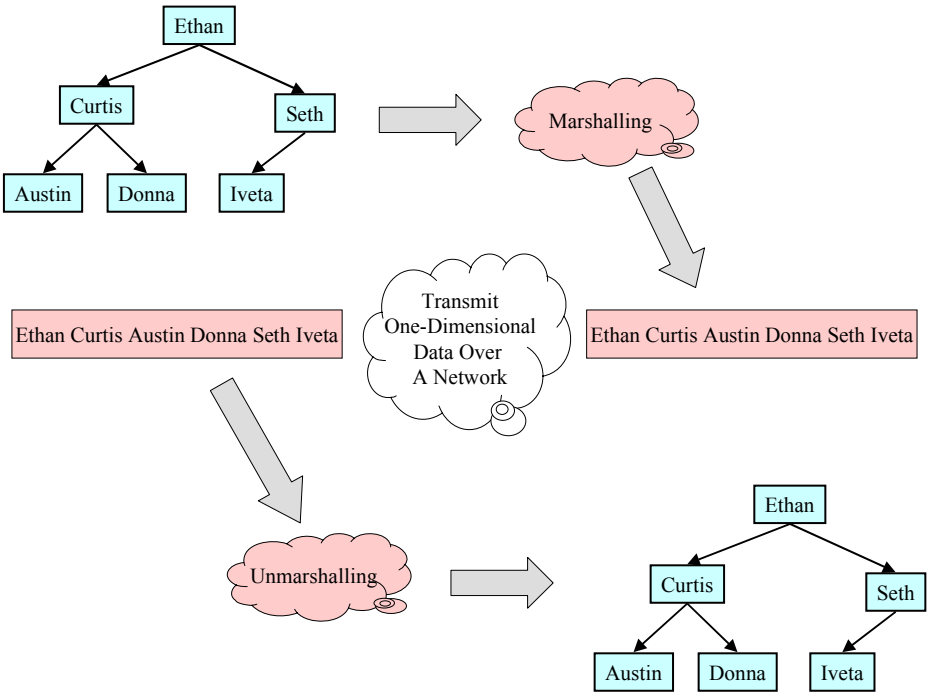


Figure 2.7. Marshalling Multi-Dimensional Data.

Name Service

When a distributed system is built, connection information can be hard-coded such that each element of the system knows how to communicate with every other element in the system. While this can work, it makes the system very hard to port to another location. Rather than hard-coding connection information into the applications, a name service can be utilized. A name service is an associative map, which is accessible by distributed applications. In general, the name service requires at minimum two pieces of information for the associative map:

- A unique identifier for each distributed application.
- Connection information to communicate with the distributed application.

In a TCP/IP network, the connection information required for communication with a distributed application must ultimately resolve down to the IP address of the machine on which the application will execute and the port on which the application is listening for incoming messages. An IP address is a 32-bit numeric identifier, which is used by the Internet Protocol to locate any machine on the worldwide Internet or a machine on a local Intranet. A port is a number between 0 and 65,535 that is associated with an application. When an application is executing, it will listen on a particular port for incoming messages. When a host machine receives a message designated for a particular port, the application associated with the port is notified. The IP address and port work together like a mailing address and a full name. The mailing address gets a letter (message) sent to your home (host machine), and your full name (port) ensures that everyone in your house (host machine) knows that a letter (message) is intended for you

and not someone else in the house. TCP/IP networks are the most common networks used for cooperative robotic applications and the only networks that will be covered in this research.

A name service will provide a registration feature and a lookup feature that can be used by distributed applications. The registration feature is used to populate the associative map with connection information that allows one application to connect to another. The lookup feature searches the associative map for connection information needed to communicate with an application.

Challenges

Seven different challenges that distributed systems can face have been identified by Coulouris et al. (2001). How L addresses these challenges will be covered in Chapter IV of this research. The seven challenges are:

1. Heterogeneity is the concept that distributed systems can be composed of resources that operate on different networks, run on different computer hardware, use different operating systems, are written in different programming languages, and are implemented by different groups of developers. The heterogeneity of a distributed system can affect the representation of bytes in a system (e.g., Big Endian vs. Little Endian). Different programming languages may represent the same data structure differently. Middleware has been created to provide consistent interfaces that hide the underlying differences in networks, computer hardware, operating systems, and programming languages.

2. Openness is the degree to which, or ease of which, a distributed system can be extended or modified. Open systems require published interfaces and uniform communication mechanisms. An important feature in developing an open system is in maintaining a separation of policy and mechanism (Rubini and Corbet, 2001; Tanenbaum and van Steen, 2002). A mechanism is the set of capabilities or features that a system provides. A policy is a plan of how the capabilities or features of one or more mechanisms will be combined to solve a problem. When mechanism and policy are combined in a single program, it is more difficult to alter a single resource without affecting the entire system.
3. Security for some resources is composed of maintaining confidentiality of the resource against unauthorized access, maintaining integrity of the resource against unauthorized alteration or corruption, and maintaining availability of the resource for those authorized to access the resource. A firewall often provides adequate security against those outside a local Intranet, yet it will not provide adequate security to restrict user access within a local Intranet.
4. Scalability is the concept that describes a distributed system that maintains its performance even when there is a significant increase in the number of distributed software or data resources or users of the system. Three techniques have been identified as helpful in designing a scaleable system (Tanenbaum and van Steen, 2002):

- Using asynchronous communication vs. synchronous communication.
- Splitting large software or data resources into smaller resources and distributing the smaller resources.
- Replicating software or data resources.

5. Failure handling is the ability of a distributed system to deal with the loss of one or more software or data resources of the system. Failure handling in distributed systems is particularly difficult because the loss of a resource may not result in the system failing, but may result in an incorrect solution to a problem. A system with failure handling does this either by detecting failures or by tolerating failures. When a system detects failures, it requires a mechanism by which it will be informed of a failure that has occurred. For all the failures that can be detected by a system, a policy must exist that establishes what action is to be taken when the failure is detected. When a system is made to tolerate failures, it need not necessarily detect failures. A common technique for making a system fault tolerant is to replicate software and data resources. When a resource is lost, a duplicate copy of the resource can be used in its place.
6. Concurrency must be dealt with in a distributed system because it is always possible that a single software or data resource of the system will be accessed at the same time by other software resources of the system. Serial equivalence is defined by Coulouris et al. (2001) as:

> If each of several transactions is known to have the correct effect when it is done on its own, then we can infer that if these transactions are done one at a time in some order the combined effect will also be correct. An interleaving of the operations of transactions in which the

> combined effect is the same as if the transactions had been performed one at a time in some order is a serially equivalent interleaving. (p. 475)

A commonly used technique based on serial equivalence for dealing with concurrency related issues is locking. In locking, a mechanism is used to restrict access to a special global resource. Only one concurrent process can gain access to the global resource at any point in time; when access to the global resource is gained, a lock on the resource is obtained. The process must release the lock before another process can lock the global resource. Methods that acquire or release a lock must be synchronized so they will not interfere with each other. The use of locks has three potential pitfalls. First, acquiring and releasing locks on resources is an overhead that can affect the performance of a system. Second, acquiring locks can result in deadlocks. A deadlock occurs when multiple concurrent processes are unable to proceed in their execution because each process requires the use of a resource that another process has locked. Because each deadlocked process is waiting for another deadlocked process to release a lock, all the processes are deadlocked. Third, the more locks and synchronized methods that are used by a system, the less concurrency the system exhibits.

7. "Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components" (Coulouris et al., 2001, p. 23). Eight forms of transparency have been identified by the Advanced Network Systems Architecture Reference

Manual (ANSA, 1998) and the International Standards Organization's Reference Model for Open Distributed Processing (ISO, 1992). Mobility transparency, defined by Coulouris et al. (2001), will be used in place of the original term 'migration transparency,' defined in the ANSA Reference Manual. Two of the most important transparencies are access transparency and location transparency; these two transparencies are sometimes grouped together under the heading of network transparency. The eight transparencies are:

- Access transparency enables local and remote resources to be accessed using identical operations.
- Location transparency enables resources to be accessed without knowledge of their location.
- Concurrency transparency enables several processes to operate concurrently using shared resources without interference between them.
- Replication transparency enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.
- Failure transparency enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.
- Mobility transparency allows the movement of resources without affecting the operation of a program.

- Performance transparency allows the system to be reconfigured to improve performance as loads vary.
- Scaling transparency allows the system and applications to expand in scale without change to the system structure or the application algorithms.

Distributed Architectural Models

Various architectural models are used in distributed systems design. Five of the most commonly used models are the Client-Server Model, Multiple-Tier Server Model, Multiple-Server Model, Peer-to-Peer Model, and Coordination Model. When using the L programming language, there will be no difference in how a developer creates a distributed application when compared to the development of a stand-alone application. To illustrate how the programming language L can be used to implement a distributed application that is independent of how the system will be distributed at run-time, example applications will be implemented that exhibit the salient features from selected distributed architectural models.

Client-Server Model

The Client-Server Model is perhaps the most commonly used distributed architecture. Coulouris et al. (2001, p. 34) refer to the Client-Server Model as "historically the most important" of the Distributed Architectural Models. The Client-Server Model is composed of multiple client objects and at least one server object. The client and server objects can all reside on different servers. Communication in the Client-

Server Model is initialed from the client to the server when a client sends a request to the server. The server, as its name implies, provides one or more services that a client can request. A client request is complete when the server returns a reply to the client request. A client need not maintain communication with a server once the server has replied to the client. At startup, a server in the Client-Server Model may register its existence with some type of name service. The name service will provide a lookup feature, which can be used by clients to find the location of the servers with which they wish to communicate. Other than an initial registration, a server will sit idle until it receives a request from a client. A server need not have any knowledge of clients, except that which it maintains while servicing a request. Once the request is complete, the server can forget all the information about the client and its request. A client may establish communication with multiple servers that all provide different services to the client. Figure 2.8 gives a graphical depiction of the relationship between clients and servers.

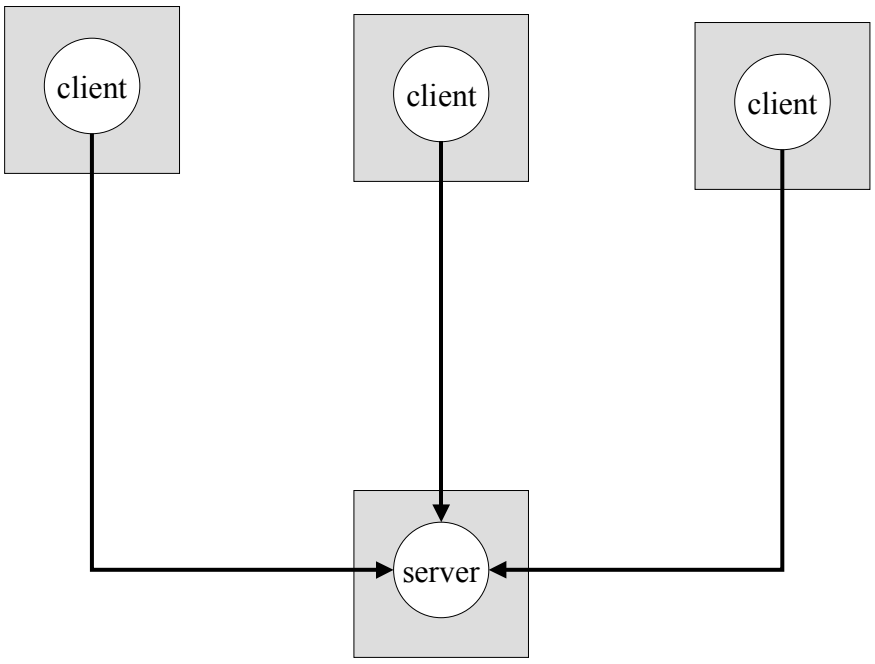


Figure 2.8. Client-Server Model.

Multiple-Tier Server Model

The Multiple-Tier Server Model is a special type of Client-Server Model. The Multiple-Tier Server Model is shown here as a separate model to help identify the unique features of the Multiple-Tier Server Model. The Multiple-Tier Server Model starts with a Client-Server Model, and then extends this model by adding multiple layers of servers. When a new server layer is added, the server layer above takes on a dual role of both server and client. The server acts as a client by sending requests to servers in lower layers, and it acts as a server to any client above it that makes requests for service. Figure 2.9 gives a graphical depiction of the relationship between clients and servers in the multiple-tier server model.

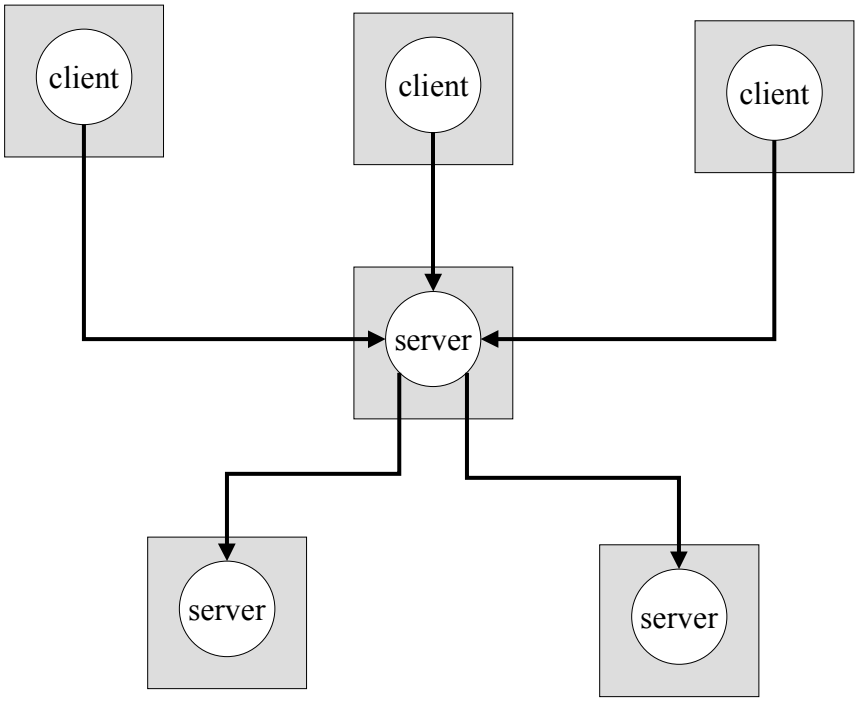


Figure 2.9. Multiple-Tier Server Model.

Multiple-Server Model

The Multiple-Server Model should look like a conventional Client-Server Model from the perspective of the developer, yet its implementation is very different. The Multiple-Server Model relies upon features provided by the network to be implemented correctly. This is in contrast to the other models that reply upon features implemented in the clients and servers to be implemented correctly. The Multiple-Server Model uses multiple servers with replicated data. Each of the servers is mapped to the same network names, such as oak.ttu.edu or www.company.com, even though each server would have a unique address. The use of the common network name allows a client to access the server in a common way, while allowing the local network to determine which physical

server the client will connect to. The local network can check the number of connections per server in order to balance the load between all the replicated servers. The Multiple-Server Model also plays a role in providing a backup in case a server is lost. Figure 2.10 gives a graphical depiction of the relationship between clients and servers in the multiple-server model.

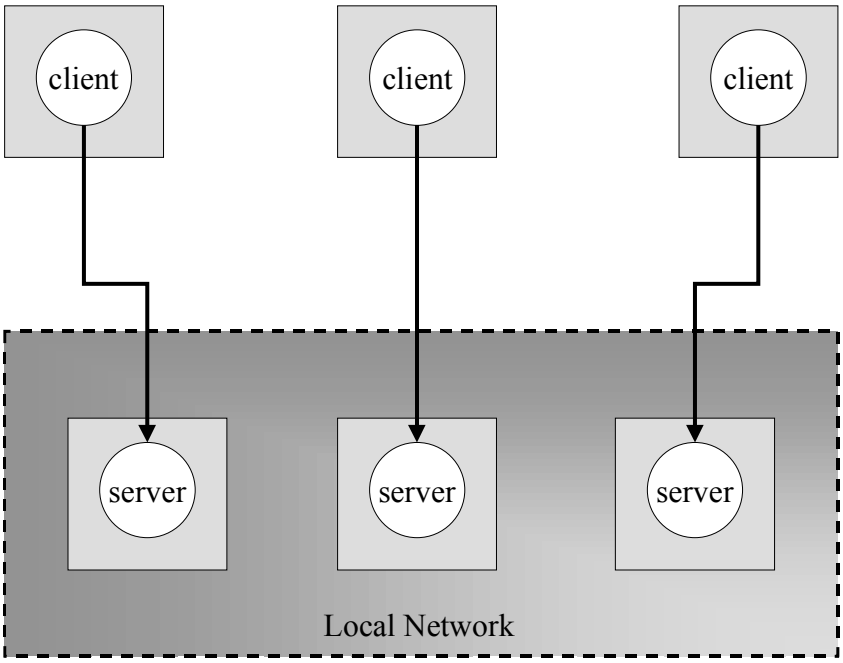


Figure 2.10. Multiple-Server Model.

Peer-to-Peer Model

The Peer-to-Peer Model is our first model to depart significantly from the Client-Server Model. The Peer-to-Peer Model is composed of objects that have roughly the same functionality. These peers cooperate in a distributed environment in order to complete a task. Because all objects are peers, communication can be initiated from any

peer, and the request can be sent to any peer, even oneself. Figure 2.11 gives a graphical depiction of the relationship between the peers.

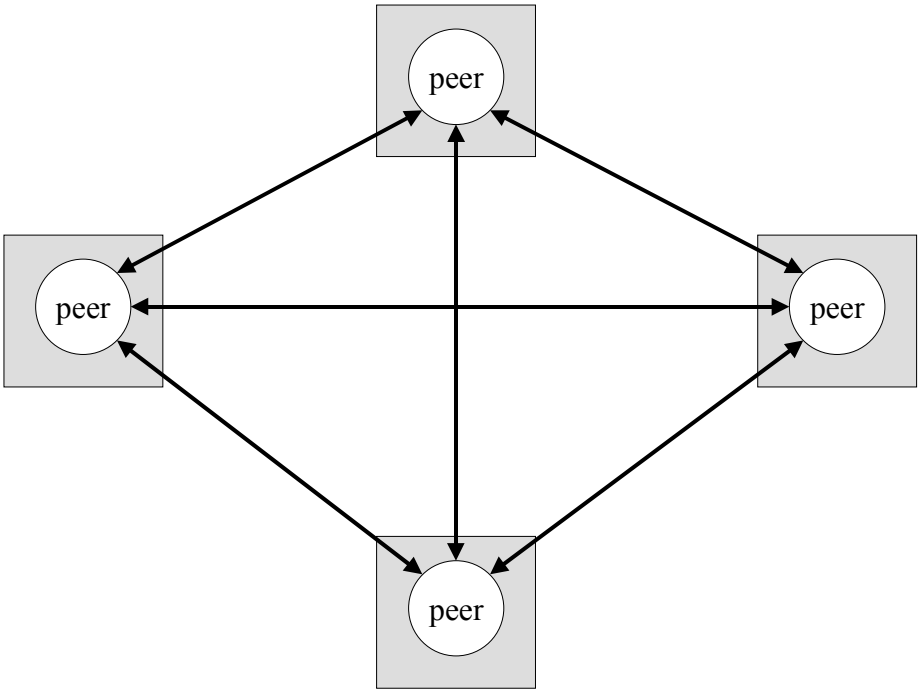


Figure 2.11. Peer-to-Peer Model.

Coordination Model

Information on the Coordination Model has been extracted primarily from two sources: Tanenbaum and van Steen (2002) and Ciancarini and Kielmann (2005). The Coordination Model is the newest architectural model, with most of the research and development in this area occurring after 1990. The first known work in the area was published in 1985 (Gelernter, 1985) as part of the Linda programming system. The Cooperative Model is best understood by introducing the coordination model taxonomy (Table 2.3) by Cabri, Leonardi, and Zambonelli (2000).

Table 2.3. Coordination Model Taxonomy.

		Temporal	
		Coupled	*Uncoupled*
Referential	*Coupled*	Direct	Mailbox
	Uncoupled	Meeting Oriented	Generative Communication

The coordination model taxonomy has two primary dimensions: the temporal and the referential. The temporal dimension expresses whether processes of an architectural model must be executing at the same time to communicate. When processes are temporally coupled, then the processes must be executing at the same time to communicate. When uncoupled, the processes can communicate without both processes executing. The referential dimension expresses whether processes of an architectural model address each other in a direct way or an indirect way. This scheme results in four different coordination models:

1. Direct Coordination Model
2. Mailbox Coordination Model
3. Meeting Oriented Coordination Model
4. Generative Communication Coordination Model

Direct coordination is the most restrictive model because:

- processes in this model must execute concurrently to exchange information,
- each process must uniquely identify itself (e.g., uid@host) to the system,
- for one process to communicate with another, it must know the unique identity of another process.

Mailbox coordination is less restrictive than direct coordination because two processes need not execute concurrently to communicate. This is made possible by services that store messages sent to a non-executing process until the process begins to execute again.

Applications that use meeting oriented coordination are also known as publish/subscribe systems. The advantage of this model over the direct coordination model is that a process can communicate with one or more processes without ever having to know a unique identifier for the other processes. This is made possible by a publish service and a subscribe service. The publish service allows a process to send (publish) a message with associated data. The subscribe service allows a process to subscribe to receive any message that is sent (published) with specific associated data. Unlike mailbox coordination, messages are not stored so processes must execute concurrently to communicate. Figure 2.12 gives a graphical depiction of the relationship between processes that can use publish/subscribe services.

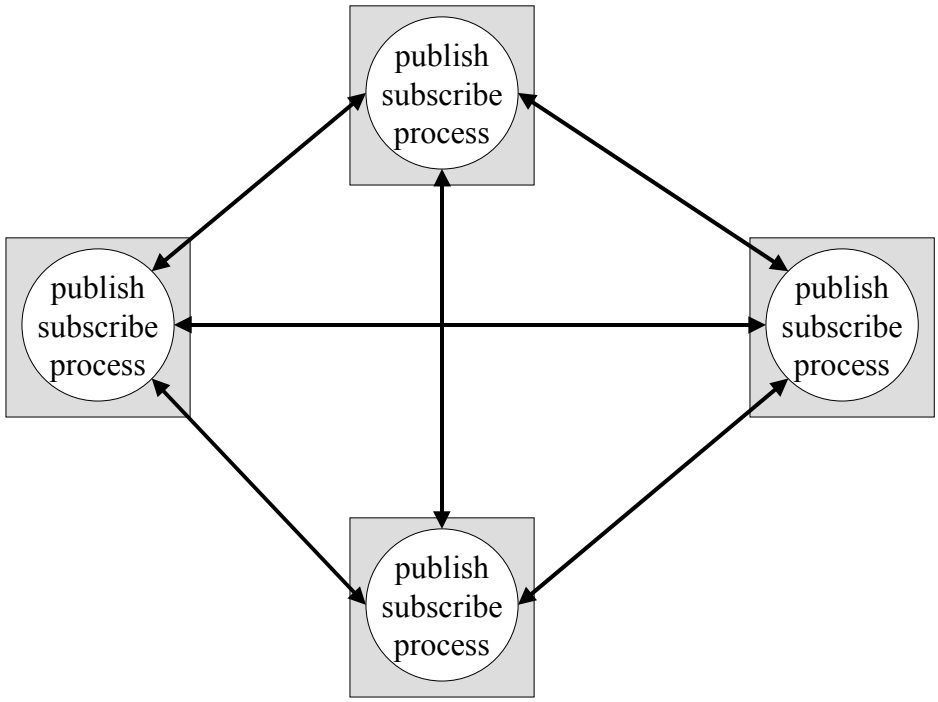


Figure 2.12. Meeting-Oriented Coordination Model.

Generative communication is the most flexible (and incurs the most overhead) of all the coordination models. It can store messages for later delivery like the mailbox coordination model, and it uses publish and subscribe services like meeting oriented coordination. Tuples (a record of 0 or more fields) are used to store messages into a persistent tuple space (database). Each tuple is tagged with information, which makes the tuple a self-describing record. Systems using generative communication rely on associative searches, which scan the tuple space looking for tuples that match the search criteria specified by an application. Figure 2.13 gives a graphical depiction of the relationship between processes and the tuple space.

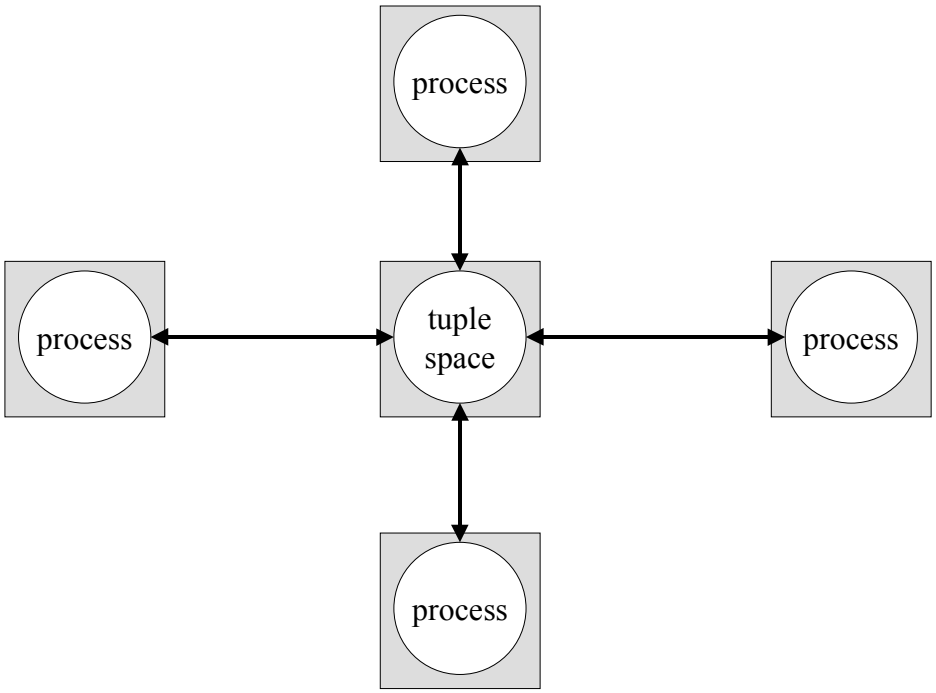


Figure 2.13. Generative-Communication Model.

Interprocess Communication

Interprocess communication is concerned with communication protocols that allow processes that are distributed over a network to communicate with each other. Communication protocols for Transmission Control Protocol/Internet Protocol (TCP/IP) networks are split into connection-oriented protocols and connectionless protocols.

> With connection-oriented protocols, before exchanging data the sender and receiver first explicitly establish a connection, and possibly negotiate the protocol they will use. When they are done, they must release (terminate) the connection. The telephone is a connection-oriented communication system. With connectionless protocols, no setup in advance is needed. The sender just transmits the first message when it is ready. Dropping a letter in a mailbox is an example of connectionless

> communication. With computers, both connection-oriented and connectionless communication are common. (Tanenbaum and van Steen, 2002, p. 59)

Sockets are a basic primitive of interprocess communication. A Universal Datagram Protocol (UDP) socket is used for connectionless communication between two processes, where a Transmission Control Protocol (TCP) socket is used for connection-oriented communication between two processes.

UDP Socket

A UDP or Datagram socket is used to send a message between two processes without any acknowledgment scheme. Without an acknowledgment scheme, UDP sockets cannot recover from lost messages or even ensure that messages will be delivered in the order that they were transmitted. It is the responsibility of the developer to implement a scheme to detect, and recover from, lost messages and to check message ordering if these issues are important. While understanding lost messages should be intuitive, the idea of message ordering is not. Messages can be received in an order different than the order they were sent because in most networks there is more than one possible path a message can follow between any two computers. If the time to transmit data along these paths is different, then messages may be received out of order when messages follow different paths to the same destination. In addition to having no acknowledgment scheme, a UDP socket imposes a fixed message size without any buffering. As a result, the receiving process may truncate a transmitted message if not enough space is allocated to hold the entire message. The primary benefit in using UDP

sockets is that they add very little overhead to an application that must use interprocess communication. UDP sockets are commonly used in cooperative robotics because of their simple use and low overhead. Figure 2.14 shows an example written in Java of how to send a message using UDP sockets.

```
DatagramSocket socket = new DatagramSocket();
int port = 8181;
byte[] buf = new byte[256];
//   Put message in buf.
InetAddress address = InetAddress.getByName("ficus.cs.ttu.edu");
DatagramPacket dp =  new DatagramPacket(buf, buf.length, address, 8181);
socket.send(dp);
```

Figure 2.14. Send a UDP Message Example.

Figure 2.15 shows an example written in Java of how to receive and respond to a message using UDP sockets.

```
sock = new DatagramSocket(8181);
while (true) {
        byte[] buf = new byte[1024];
        // Receive Datagram
        DatagramPacket dp = new DatagramPacket(buf, buf.length);
       socket.receive(dp);
       /*  Code needed to make a response to the message and place it into buf.  */
       // Send Responce
       InetAddress address = dp.getAddress();
       int port = dp.getPort();
       dp = new DatagramPacket(buf, buf.length, address, port);
        sock.send(dp);
}
```

Figure 2.15. Receive and Respond To a UDP Message Example.

IP Multicast

IP Multicast or broadcast communication is used when a sender process needs to communicate the same information to a group of receivers' processes, yet the exact size and members of the group is not known. The basic idea behind IP multicast is that a receiver process joins a multicast group using a specific port number. A sender process can then send a single message to every member of the group by specifying a group address and a port number. IP multicast is implemented using UDP sockets. Broadcast communication using IP multicast is arguably the most common interprocess communication mechanism used in cooperative robotics. Once all the members of a team are in the same multicast group, individual team members can communicate with all other team members by sending a single message. However, because IP Multicast is built on UDP sockets, there is no automatic mechanism to recover from lost messages or

ensure message ordering. As discussed previously in this research, the use of broadcast communication does not ensure cooperation of a team. It simply means that all members are equally informed of a member's intentions. Yet the practical reality of using UDP sockets is that not all members may even receive the correct information if an acknowledgment scheme is not implemented to deal with lost messages and message ordering. Figure 2.16 shows an example written in Java of how to broadcast a message using IP Multicast.

```
byte[] buf = new byte[256];
/* Code needed to put message into buf.  */
InetAddress group = InetAddress.getByName("230.0.0.1");
DatagramPacket dp =  new DatagramPacket(buf, buf.length, group, 8181);
 socket.send(dp);
```

Figure 2.16. Broadcast Sender Example.

Figure 2.17 shows an example written in Java of how to receive an IP Multicast message.

```
MulticastSocket sock = new MulticastSocket(8181);
InetAddress address = InetAddress.getByName("230.0.0.1");
sock.joinGroup(address);
byte[] buf = new byte[256];
DatagramPacket dp = new DatagramPacket(buf, buf.length);
while (Working) {
        socket.receive(dp);
        // Process data in the message
 }
 socket.leaveGroup(address);
 socket.close();
```

Figure 2.17. Broadcast Receiver Example.

TCP Socket

A TCP socket establishes a connection between two processes (Figure 2.18) using two byte streams, each of which allows one-way communication. These two byte streams, which are associated with a single socket, allow two-way communication between the two processes. The process at one end of the connection is considered a client; the process at the other end is considered a server. The server process will create a TCP socket, bind it to a specific port number, and then listen on the port for any client that is attempting to connect with the server. A client process creates a TCP socket and then attempts to connect to a server process by sending data to the server process using the IP address and port number of the server. An acknowledgment scheme is built into

TCP sockets to provide transparent handling of message size, lost messages, flow control, and message ordering.

- A TCP socket allows a process to write data to a byte stream; the byte stream will in turn determine how many bytes it will collect before sending an IP packet on the network. On the receiving end of a byte stream, the byte stream can collect bytes together and provide them as buffered input to the process.
- A TCP socket uses an acknowledgment scheme that allows it to detect lost messages. When a receiver process detects a lost message, the sender process can be directed to retransmit the lost data.
- A TCP socket will attempt to match the speed of reading and writing on a stream by blocking the writer process from sending too much data at any one time.
- A TCP socket can detect when messages are received out of order and reorder them correctly, or when a duplicate message is received, it will discard the duplicate.

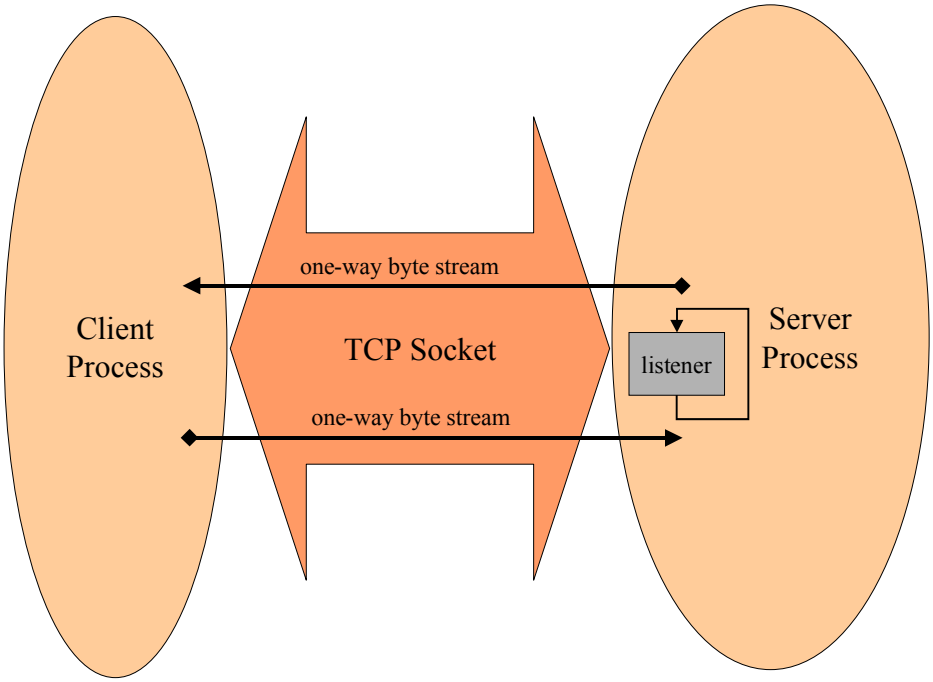


Figure 2.18. TCP Socket.

TCP sockets can be used for communication in cooperative robots, yet their use can be more cumbersome than using UDP sockets. First, each robot will need both server-side and client-side socket code. If a robotic team is composed of fifty robots, then each robot must manage forty-nine sockets to communicate with all the other robots in the team. For one robot to send the same data to the whole team, it will have to loop through all its sockets and write the same data to the output stream. Figure 2.19 shows example client code using TCP sockets written in Java.

```
int port = 8181;
InetAddress ip = InetAddress.getByName("ficus.cs.ttu.edu");
Socket sock = new Socket(ip, port);
ObjectOutputStream writer =  new ObjectOutputStream(sock.getOutputStream());
ObjectInputStreamreader = new ObjectInputStream(sock.getInputStream());
int data = reader.readInt();
writer.writeInt(739);
```

Figure 2.19. Client Code Example.

Figure 2.20 shows example server code using TCP sockets written in Java.

```
int port = 8181;
ServerSocket ss = new ServerSocket(port);
while(true) {
        Socket incoming = ss.accept();
        //   Code needed here to spawn a thread to process the incoming socket.
         ObjectOutputStream writer =
                new ObjectOutputStream(sock.getOutputStream());
         ObjectInputStreamreader =
                new ObjectInputStream(sock.getInputStream());
          writer.writeInt(937);
          int data = reader.readInt();
}
```

Figure 2.20. Server Code Example.

The writeInt and readInt methods allow a developer to deal directly with integer data rather than having to deal with individual bytes. This is a high-level feature of the Java socket implementation. It is important to note that while this server code will work, from a practical standpoint, without additional threads, this server code is useless because the server can only deal with one client at a time. The additional thread handling code is not shown to keep the example simple. In addition, if objects more complex than

integers are to be transmitted over the network, marshalling code may need to be written by the developer.

Remote Method Invocation

In the Java programming language, Remote Method Invocation (RMI) can be used to communicate between two processes. RMI provides a mechanism where a method on a remote object can be invoked using a programming interface that appears the same as invoking a method on a local object (e.g., object.method()). While RMI provides a high-level interface for sharing information between processes, its use is neither transparent nor particularly simple.

> Running even the simplest remote object example requires quite a bit more setup than does running a standalone Java program or applet. You must run Java programs on both the server and client computers. The necessary object information must be separated into client-side interfaces and server-side implementations. There is also a special query mechanism that allows the client to locate objects on the server. (Horstmann and Cornell, 1998, p. 232)

The basic steps required for creating an RMI application are:

1. Write a Java interface for the server, which must extend (inherit from) the interface java.rmi.Remote. Both server classes and client classes will share this interface.
2. Write a server class, which must extend (inherit from) the java.rmi.server.RemoteServer class. Methods of the server class that are to be remotely accessed must throw the exception java.rmi.RemoteException.
3. Write a program to register server-side objects.

4. Write a client class to access methods of the server class. Code that is written to invoke remote methods must catch the exception java.rmi.RemoteException.
5. Compile the code using javac to create Java class file.
6. Generate stubs for client-side processing and skeletons for server-side processing by running rmic on the server class files.
7. Copy server classes and server skeletons to the server computer.
8. Copy client classes and server stubs to the client computer.
9. Execute the bootstrap registry service on the server.
10. Execute the program to register your objects with the registry service.
11. Execute the client program.

The L programming language will use a mechanism similar in appearance to RMI, though from the programmer's perspective the use of this mechanism will be simpler because there will be no difference in how a stand-alone application is written when compared to a distributed application. TCP sockets will be used to support distributed communication, though this detail will be hidden within the execution model of L.

CHAPTER III

METHODOLOGY

This chapter will describe a set of organizing principles and high-level design choices used to establish boundaries in which this research was conducted. These principles and design choices often provide clear directions as to how the language should be designed to meet the objectives of the research:

- reduce the complexity of developing a distributed application,
- maintain tractable communication complexity among cooperating agents.

Organizing Principles

The principles followed to make it easier to develop a distributed application were:

- encapsulation of information,
- separation of cooperative robotic applications into the fundamental concerns of resources and coordinators,
- use of a computational mobility-based approach for communication.

These principles and how they are applied to the design of L are in line with the philosophy that C. A. R. Hoare presented on the design and evaluation of programming languages. Of particular relevance are Hoare's views on:

- adding structure to an existing programming language--"The introduction of program structures into a language not only helps the programmer, but does not injure the efficiency of an implementation" (Hoare, 1973, p. 36).

- imposing conventions through a programming language--"[A programming language] should assist in establishing and enforcing the programming conventions and disciplines which will ensure harmonious cooperation of the parts of a large program when they are developed separately and finally assembled together" (Hoare, 1973, p. 31).

Encapsulation of Information

A choice was made to start with an object-based[4] language as a baseline and then modify the grammar as needed. The syntax of L is similar to Java, though L does not support inheritance and the creating of packages or interfaces. An object-based language was chosen as a baseline because principles of encapsulation are well developed in object-based languages and encapsulation was seen as a critical principle to use in developing the language. Pratt and Zelkowitz define the difference between information hiding and encapsulation as follows:

> Note that information hiding is primarily a question of program design; information hiding is possible in any program that is properly designed regardless of the programming language used. Encapsulation, however, is primarily a question of language design; an abstraction is effectively encapsulated only when the language prohibits access to the information hidden within the abstraction. (Pratt, Zelkowitz, and Marvin, 2001, p.238)

[4] An object-based language supports the creation of classes and objects but does not support inheritance.

An object-based language can be viewed as a procedural language with special constructs to support encapsulation. These encapsulation constructs (e.g., classes, packages, public and private access modifiers) are used to allow a high degree of control to be specified over the scope or visibility of defined attributes. The scope of an attribute (e.g., function, data member) in a procedural language is generally viewed as either being global or local. The scope of an attribute in an object-based language can be viewed as both, depending upon one's perspective. A class defined in most object-based languages has a global scope. Attributes defined within a class can have a scope (e.g., public) that makes them visible outside the class and thus they seem global, or an attribute can have a scope (e.g., private) that makes them visible only within the class and thus they seem local. The ability to nest objects one inside another and to create objects, which are an instance of a class, allows for even more flexibility.

Separation of Fundamental Concerns

The next organizing principle that was followed was to separate the development of cooperative robotic applications into two fundamental concerns.

- The first fundamental concern is the development of a control system for a physical hardware resource (e.g., robot).
- The second fundamental concern is the development of a Multi-Agent System (MAS) to coordinate hardware resources.

Dividing cooperative robotic applications along these lines fits nicely with a strategy used in operating systems of separating a system into mechanism and policy. In

operating systems, a device driver is referred to as a mechanism, while an application that uses device drivers is referred to as a policy.

> The distinction between mechanism and policy is one of the best ideas behind the Unix design. Most programming problems can indeed be split into two parts: "what capabilities are to be provided" (the mechanism) and "how those capabilities can be used" (the policy). If the two issues are addressed by different parts of the program, or even by different programs altogether, the software package is much easier to develop and to adapt to particular needs. (Rubini and Corbet, 2001, p. 2)

Using this principle, along with the principle of encapsulation, a "resource" construct and a "coordinator" construct were added to the programming language L. The resource construct is used to develop a control system (mechanism) for a physical hardware resource. The coordinator construct is used to develop code to coordinate resources. L imposes restrictions on the use of resources and coordinators that ensure they are not used interchangeably (e.g., coding a control system with a coordinator or coordinating multiple resources with a resource).

From the perspective of the virtual machine, resources and coordinators are globally visible. From the perspective of a coordinator, all resource objects are visible, yet all other coordinator objects are invisible. From the perspective of a resource, all coordinator objects and all resource objects are invisible. By restricting the visibility of resources and coordinators in this way, the separation of mechanism and policy is maintained.

The published interface for an object is the set of attributes that an object makes visible to others. To specify the capabilities of an object, a frame of reference beyond that of the published interfaces is often needed. To provide this type of context,

capability primitives have been identified for cooperative robotic applications. The primitives organizes robotic capabilities to make their use more effective in L.

Use of a Computational Mobility-Based Approach

Another organizing principle was the use of computational mobility to manage communication complexity[5]. In a computational mobility-based approach, a computation migrates to the location of data, rather than the data being sent to the location of the computation and then results having to be sent back. In following this approach, a coordinator, which coordinates robotic resources to solve a problem, will migrate to the location of a robotic resource to collect data from the resource and to provide direction to the resource. This is in contrast to a more conventional strategy where all resources broadcast data back and forth to each other. Dividing cooperative robotic applications into resources and coordinators, and then using principles of information encapsulation to maintain separation between them, was critical to developing a system that allows for the transparent migration of a coordinator.

[5] Members of the Automated Reasoning (AR) sub-project (Kenneth Ford, Pat Hayes, James Allen, Robert Morris, Butler Hine and Daniel Cooke) have suggested that computational mobility could be used as an alternative to each robot broadcasting messages to maintain tractable communication between cooperative robots.

High-Level Design

The organizing principles helped address the objective of reducing the complexity of developing a distributed application. Yet to develop a language such that there is no difference in how a developer creates a distributed application when compared to the development of a stand-alone application, two problems remain:

- how to access an object independent of how the objects of the system are distributed,
- how to initialize an application independent of how the objects of the system are distributed.

Remote Object Access

Remote Method Invocation (RMI) from the language Java provides a mechanism where a method on a remote object can be invoked using a programming interface that appears the same as invoking a method on a local object. As this is the same type of mechanism that L required, RMI was selected as a model for how remote objects would be accessed.

Application Initialization

The Problem

In most programming languages, a main function is written for each program. This main function provides a single entry-point into the program, and from this single entry-point, the entire program is initialized (objects are created and given initial data

values). When an application is distributed, multiple programs are written (e.g., clients, servers), each with their own entry-point. To write the main function for a program, the developer must know at development time which objects of the system are to be initialized in a program. L, however, cannot rely on this type of strategy to perform object initialization. To be completely transparent, an L application has to support being written as a single stand-alone program, yet at run-time it must support being executed as multiple distributed programs. This means a developer cannot make an assumption at development-time as to which objects to initialize in a program or even how many programs will be executed at run-time for a given application.

The Solution

To address this issue, the focus needs to be shifted away from how a program is initialized to how objects are initialized. Using object initialization as a focus, there is no main function in an L program to perform initialization. Instead, each resource object and coordinator object is given an independent entry-point (i.e., an init method) that allows the object to be initialized by the virtual machine. To complete the solution, a method, independent of how an application is written, is needed to inform the virtual machine of which objects to initialize. Research this author had done on developing a hybrid deliberative/reactive programming language, known as Io, provided a solution.

The strategy used is to allow a deployment file to be created by application support personnel (or a developer) separate from the code for the application. The deployment file includes a single entry, based on the Uniform Resource Locator (URL)

format from HTML for each resource object and coordinator object that is to be initialized. A special process known as the deployer uses the deployment file and the target code generated by the L compiler as input to create one or more executable programs for an application. Each program created by the deployer will initialize a specific set of objects based on the information provided in the deployment file. A developer never needs to rewrite an application based upon a new deployment of objects. Instead, the deployment file is rewritten and the deployer process is executed again to create a new set of executable programs. Specifying the initialization of resource and coordinator objects in this way is an appropriate strategy for cooperative robotic applications because these objects are associated with physical hardware, and real hardware cannot just appear dynamically; it must be configured prior to its use.

Generating Object Code vs. a Virtual Machine Approach

A design choice between whether the compiler for the language L would generate object code or bytecode, which would be executed by some virtual machine, had to be made. The approach chosen was to develop a compiler that generated bytecode, and a virtual machine that could execute the bytecode. Using a virtual machine to support code mobility aligned this research more closely with other language-based research in code mobility, such as the programming language Sumatra (Ranganathan et al., 1997). Though in the case of Sumatra, the developer must write code to migrate an object to a new host, whereas in L, all migration is transparent to the developer.

Establishing a High-Level Migration Process

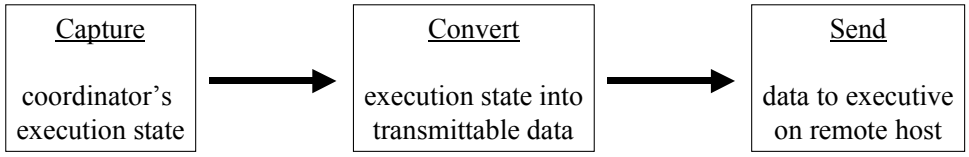
A salient feature of the L virtual machine is its fundamental support for code mobility. A high-level design for migrating a coordinator is depicted in Figure 3.3.

1. In step one, known as detection, the virtual machine checks if a resource being accessed by a coordinator is on the same host as the coordinator or is located on a remote host. If the resource is remote, processing proceeds to step 2; otherwise, the coordinator just accesses the capabilities of the resource.
2. In step two, known as projection, the virtual machine must:
    - Capture the execution state of the running coordinator.
    - Convert the execution state of the coordinator into marshalled data.
    - Send the marshalled data to the virtual machine running the remote resource.
3. In step three, known as injection, the virtual machine on the remote host must:
    - Receive the marshalled data that has been transmitted to it.
    - Reconstruct the execution state of the coordinator from the marshalled data.
    - Restart the coordinator to running so that it can access the local resource.

1•Detection

Is the resource being accessed local or remote?

2•Projection



3•Injection

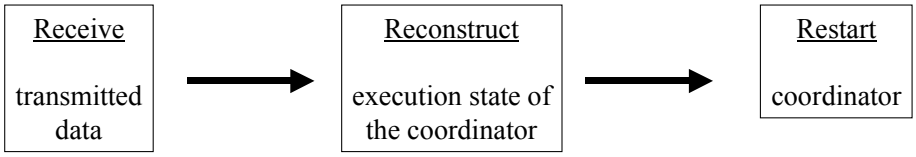


Figure 3.1. Three-Step Migration Process.

CHAPTER IV

RESULTS

This research has achieved the following results:

1. Reduced the burden placed on a developer to implement a cooperative robotics application.
2. Developed a language that can maintain tractable communication between a team of cooperative robots.
3. Identified initial capability primitives and associated guidelines for continued evolution.

This chapter outlines these results and provides additional information on work done to achieve these results. The chapter is broken down into the following sections:

- the Programming Model section covers L from the perspective of what a developer must know to write programs, the syntax and semantics of the language, abstractions the language uses, and the specific obligations or responsibilities a programmer has when programming.
- the Execution Model section covers issues related to compiling, deploying, and executing an application.
- the Example L Applications section presents applications that implement the salient features of a Client-Server application, a Multiple-Tier Server application, and a Peer-to-Peer application, as well as discussing how L

can be used to implement a Tuple Space application and a three-tier architecture.

- the Distributed Systems Challenges section covers how the distributed systems challenges detailed by (Coulouris et al., 2001) and covered in Chapter II are addressed by this research.
- the Capability Primitives section gives example primitives that organizes robotic capabilities to make their use more effective in L.

Programming Model

The complete grammar for L can be found in appendix A. Syntactically the following were added to an object-based language:

resource IDENTIFIER { … }

A resource is a user-defined type that allows the grouping of algorithms and data structures into a single construct. The IDENTIFIER becomes the name of a resource type. Statements of L that perform I/O (e.g., read, write) or which interface with local hardware in any way are only defined within the scope of a resource. The scope of a resource is such that resources cannot access each other. An instance of a resource can only be created by the L virtual machine.

coordinator IDENTIFIER { … }

A coordinator is a user-defined type that allows the grouping of algorithms and data structures into a single construct. The IDENTIFIER becomes the name of a coordinator type. The scope of a coordinator is such that all resources are visible, yet coordinators cannot access each other. A coordinator is restricted from accessing any statement that interfaces with local hardware. An instance of a coordinator can only be created by the L virtual machine.

portal SYNC_X IDENTIFIER (PARAMETER_LIST_X) EMIT_X { … }

A portal is a special type of member function, which can only be defined in a resource. A portal can be passed a list of input arguments and can emit a single argument. All arguments, input or emitted, are passed-by-value, with a deep-copy being performed. When a data structure is passed-by-value, either a shallow-copy or a deep-copy of the data structure can be made. A "shallow-copy" will only copy the pointer that points at the data structure, Figure 4.3 (a). A "deep-copy" will copy a closure of the pointer, Figure 4.3 (b). A portal can be declared synchronized, which will ensure that it is executed by only one thread at a time. Any subsequent thread attempting to concurrently execute the portal will be forced to wait until the first invocation has completed. A portal not declared synchronized can execute on multiple concurrent threads. The scope of a portal is such that only a coordinator can access them.

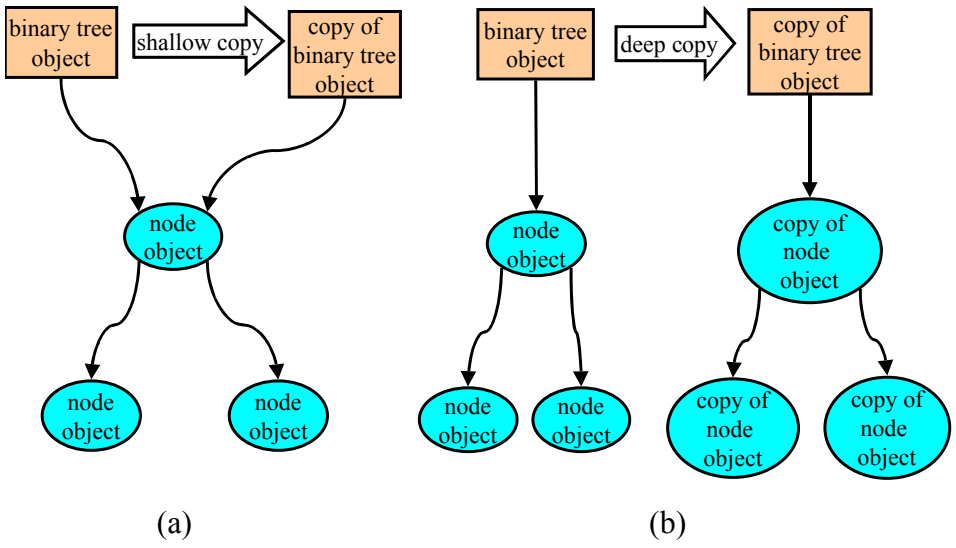


Figure 4.1. Shallow and Deep Copy.

init (string[] IDENTIFIER) { … }

The init method is a required member function of all defined resources and coordinators. The L virtual machine will execute the init method on a separate thread for each instance of a resource or coordinator created. An init method is always passed an array of strings. The strings that are passed to the init method are defined in a deployment file. The first element of the array, arg[0], is always the unique identifier for the instance being created.

class IDENTIFIER USAGE { … }

A class is a user-defined type that allows the grouping of algorithms and data structures into a single construct. The optional usage clause allows a class to be defined as either "resource specific" or "coordinator specific."

Identifying a class as resource specific defines the class within a scope that will allow the class to invoke statements, which perform I/O or interface with local hardware. Without the "resource specific" usage clause, a class is not defined within a scope that will give it access to these operations. The scope in which a resource specific class is defined restricts it to only being accessed by resources or another resource specific class.

Identifying a class as coordinator specific defines the class within a scope such that all defined resources are visible. Without the "coordinator specific" usage clause, a class is not defined within a scope that will give it access to a resource. The scope in which a coordinator specific class is defined restricts it to only being accessed by coordinators or another coordinator specific class.

find methods

The L compiler will generate two kinds of find methods for each user-defined resource. The find methods are used to locate a proxy object for an instance of a resource. A resource instance is not returned by the find methods, as the instance may be located on a remote machine in the network. The first kind of find method will return a proxy for a resource instance using the unique identifier for the resource instance. The unique identifier for a resource instance is generally either hard-coded into a program or passed into a program as a deployment argument. The second kind of find will return an

array, which contains a proxy for each instance of a user-defined resource. The scope in which the find methods are defined restricts them to only being accessed by coordinators and coordinator specific classes.

RC_IDENTIFIER:IDENTIFIER@SERVER

RC_IDENTIFIER:IDENTIFIER@SERVER?DEPLOYMENT_ARGUMENTS

A deployment entry is used to specify on what server an instance of a user-defined coordinator or resource is to be created. The RC_IDENTIFIER is the type of the coordinator or resource to be created. The IDENTIFIER is a unique identifier that will be associated with this instance. The SERVER is the unique Internet name or Internet Protocol (IP) numerical address of a computer on which the instance will be created. An optional list of deployment arguments can be supplied. When the L virtual machine creates an instance of user-defined coordinator or resource, the deployment arguments will be passed into the init method for that instance. All deployment entries for an application are placed into a deployment file, separate from the source code for the application.

Maintaining Tractable Communication

As previously discussed in Chapter I, the communication complexity for N distributed agents to share information with each other is bounded at N^2 (Klavins, 2002). Yet this does not bound the number of messages that are required to get the same N agents to cooperate to complete a task. As a way to estimate the complexity of getting N

agents to cooperate to complete a task, the number of distinct decisions that the team can make to arrive at a consensus will be counted, instead of counting the number of messages exchanged. In this context, a "distinct decision" is when one agent makes a choice, or two or more agents come to a consensus.

Estimating a Broadcast-Based Approach

An estimate of the complexity that is required for a team of N agents to achieve a consensus is to consider the power set of N agents, which contains $2^N - 1$ distinct decisions assuming broadcast communication is used. Each element of the power set represents a partitioning of the team, from which a distinct decision can be made to reach a consensus. Figure 4.2 depicts a Hasse diagram, minus the empty set, for all the distinct decisions of three agents, while Figure 4.3 depicts all the distinct decisions of four agents.

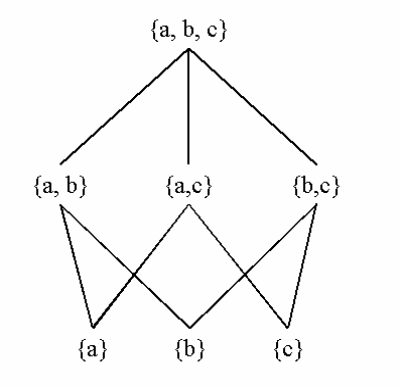


Figure 4.2. Distinct Decisions of Three Agents.

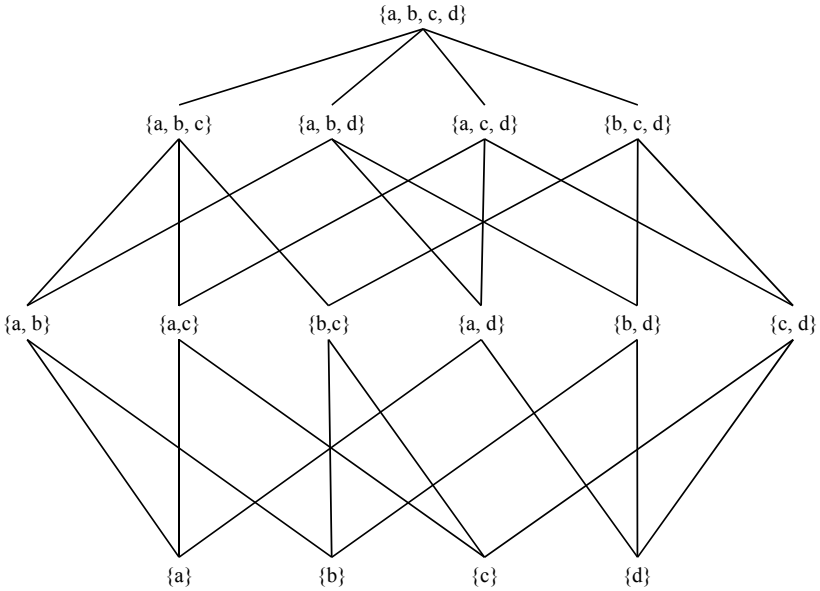


Figure 4.3. Distinct Decisions of Four Agents.

Estimating a Computational Mobility-Based Approach

To reach a consensus, a team of N resources can make 2N distinct decisions when the computational mobility-based approach of L is used. This value is computed by first counting each of the choices that the N resources can make about how to complete a task, for N distinct decisions. Second, one distinct decision is counted for each of the N resources the coordinator migrates to, for a total of 2N distinct decisions. As the number of resources in a team grows, the computational mobility-based communication approach used by L requires significantly fewer distinct decisions than a broadcast approach.

Computing the maximum number of messages that an L application needs to exchange for N resources to cooperate to complete a task is straightforward. We start by counting one message per each migration of the coordinator. A coordinator needs to migrate N times to collect data from N different resources, and another N times to inform the resources of the decision that the coordinator has made after collecting data, for a total of 2N messages (migrations).

Execution Model

A two-pass compiler handles the translation of L source code into intermediate code (L bytecode). A virtual machine is used to execute a program composed of L bytecode. The L compiler is written in SWI-Prolog version 5.0.10 and is 3,774 lines of code. The L virtual machine is written in Java version 1.4.0_01 and is 3,066 lines of code. Appendix J contains a full listing of all bytecode instructions used by L.

The pcall (Portal Call) instruction is of particular importance because it handles all the details of migrating the execution state of a coordinator object. The compiler generates a pcall instruction each time that the portal of a resource is invoked. The execution state of a coordinator object is migrated to the server on which a resource instance is located whenever a pcall is executed and the coordinator object and resource object are not on the same machine.

A deployer process, which is similar to a linker, uses bytecode produced by the compiler and a deployment file to create executable programs for an application. The deployer will generate programs for each server specified in the deployment file. The relationship between the compiler/deployer, source code file, deployment file, and executable program files is depicted graphically in Figure 4.4. Each server-specific program contains all the bytecode defined for an application. At startup, the L virtual machine executing each server-specific program will:

- create an instance of every user-defined resource specified in the deployment file to be on this server.
- create an instance of every user-defined coordinator specified in the deployment file to be on this server.
- invoke on a separate thread the init method for each instance created.
- pass as input arguments to each init method the unique identifier of the instance and any deployment arguments, both of which are specified in the deployment file.

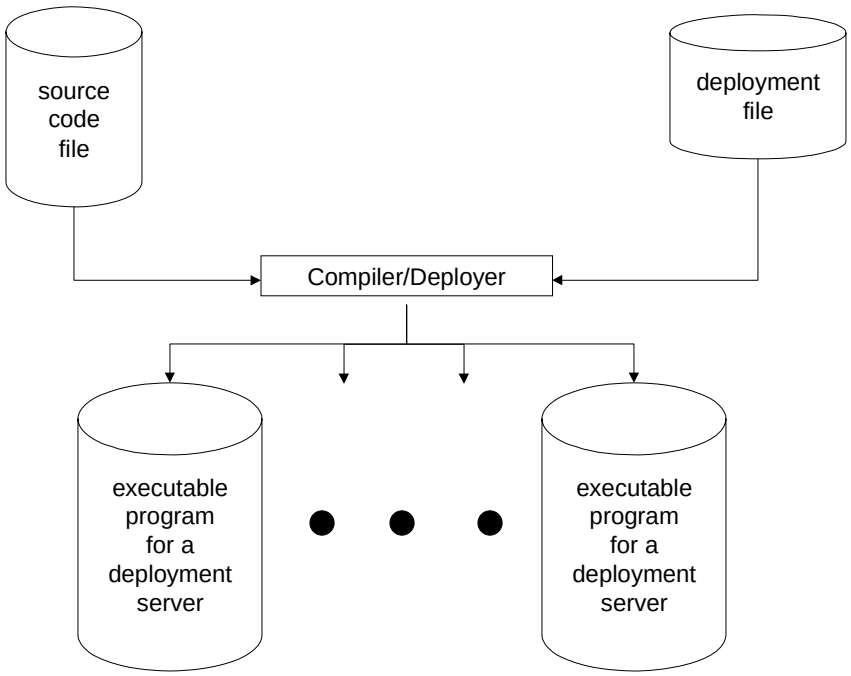


Figure 4.4. Compiler/Deployer.

Example L Applications

To provide concrete examples of how the L programming language works, an example application that demonstrates the salient features of a Client-Server application will be implemented. This evaluation will then look at how few changes are needed to the source code of the Client-Server application to support first a Multiple-Tier Server application and then, finally, a Peer-to-Peer application. The example programs will show that even though L advertises no conventional interprocess communication constructs (e.g., sockets, RMI) for writing a distributed application, distributed applications with functionality that map to existing distributed architecture models can still be created and deployed on conventional hardware. Full source code listings and

program results can be found in appendixes C, D and E. This section will conclude with a discussion on how L can be used to implement:

- a Generative-Communication (Tuple Space) application.
- the three-tier architecture from the deliberative/reactive strategy.

Client-Server Application

Full source code listings and program results for the Client-Server application can be found in appendix C. The Client-Server application was implemented using two resources: Server (Figure 4.5), Client (Figure 4.7) and one coordinator Transport (Figure 4.8).

```
resource Server {
   string id;

   init(string[] arg) {
         while(true) yield();
   }

   …

   portal fib(string who, int k) emit int {
         int oneBack = 1;
         int twoBack = 0;
         int current;
         int i;

         if (k <= 1) {
            badData(who, k);
            return k;
         }
```

```
        else {
            i = 2;
            heading(who, twoBack);
            while(i <= k) {
                current = oneBack + twoBack;
                write(", ");
                write(oneBack);
                twoBack = oneBack;
                oneBack = current;

                i = i + 1;
            }
            write(", ");
            write(current);
            write("\n");
            return current;
        }
    }

    portal fact(string who, int k) emit int {
        …
    }
}
```

Figure 4.5. Server Resource.

The Server resource contains a portal "fib" to compute and emit the k^{th} Fibonacci number, and a portal "fact" to compute and emit the factorial of a number k. Both portals require as input arguments the unique identifier of the Client that has invoked the portal and an integer. The Server is passed a single deployment argument in the init method, the unique identifier of this Server. The init method executes an infinite loop to ensure that the Server does not exit before all test cases are performed.

Reducing the Developer's Burden: Part 1

It is important to note that a developer need not write any interprocess communication code to allow the portal of resource to be invoked by a remote coordinator, as the L virtual machine handles all interprocess communication between coordinators and resources. Figure 4.6 lists the interprocess communication code that is built into the L virtual machine to support interprocess communication with a portal. If this interprocess communication code were not built into the L virtual machine, a developer would have to implement the equivalent of this code for each distributed application.

```
import java.io.*;
import java.net.*;

public class MigServer extends Thread {
        private ServerSocket s = null;
        private int port = 8181;
        private static boolean DEBUG = false;


        public static MigServer create(int port) {
                MigServer server = new MigServer(port);
                if (DEBUG) System.out.println("In create");
                if (server.init())
                        return server;
                else
                        return null;
                }

        private MigServer(int port) {
                this.port = port;
        }

        private boolean init() {
```

```
                if (DEBUG) System.out.println("In init");
                try {
                        if (DEBUG) System.out.println("New ServerSocket");
                        s = new ServerSocket(port);
                }
                catch(IOException e) {
                        System.out.println(e);
                        return false;
                }

                if (DEBUG) System.out.println("Start MigServer");
                start();
                return true;
                }

        public void run() {
                Socket incoming = null;

                while (true) {
                        try {
                                if (DEBUG) System.out.println("Waiting for Connection");

                                incoming = s.accept();
                                if (DEBUG) System.out.println("Accepting Connection");

                        }
                        catch(IOException e) {
                                System.out.println(e);
                                continue;
                        }

                        New MigHandler(incoming).start();
                }
        }
}

import java.io.*;
import java.net.*;
import java.util.Hashtable;

class MigHandler extends Thread {
        private static LBCDictionary lbcd = null;
        private static RCDictionary  rcd  = null;
```

```
private ObjectInputStream reader = null;
private ObjectOutputStream writer = null;
private Socket incoming;

private String command = null; // Current Command
private GenRec g;       // Coordinator Object
private LStack pstack; // Parameter Stack
private LStack astack; // Execution Stack
private int pc;        // Program Counter

MigHandler(Socket incoming) {
        this.incoming = incoming;
        if (lbcd == null)
                lbcd = LBCDictionary.getLBCD();
        if (rcd == null)
                rcd = RCDictionary.getRCD();
}

private synchronized boolean injection() {
        Hashtable migMap = new Hashtable();

        try {
                // Receive Transmitted Execution State
                System.out.println("injection()");
                g      = (GenRec) reader.readObject();
                System.out.println("Coordinator-->" + g.toString());

                pstack = (LStack)  reader.readObject();
                System.out.println("PStack-->" + pstack.toString());

                astack = (LStack)  reader.readObject();
                System.out.println("AStack-->" + astack.toString());

                System.out.println("Next is PC");
                pc     =          reader.readInt();
                System.out.println("PC-->"+pc);

                writer.writeObject(MigClient.INJECTION_END);
                writer.flush();

                // Reconstruct Execution State
                g.reconstruct(migMap);
                System.out.println("Reconstruct coordinator-->" + g.toString());
                pstack.reconstruct(migMap);
```

```
                System.out.println("Reconstruct PStack-->" + pstack.toString());
                astack.reconstruct(migMap);
                System.out.println("Reconstruct AStack-->" + astack.toString());


                /*
                 * Always use the default now but we can use for
                 * code later.
                 */
                LByteCode lbc = lbcd.get("default");
                ExecEngine ee = new ExecEngine(lbc, pc, astack, pstack);

                // Associate the ExecEngine with the Coordinator
                // and restart the Coordinator
                rcd.put(ee, g);
                ee.start();

                return true;
        }
        catch (ClassNotFoundException e) {
                e.printStackTrace();
        }
        catch(IOException e) {
                e.printStackTrace();
        }

        return false;
}

public void run() {
        try {
                System.out.println("MigHandler Running");

                reader =
                        new ObjectInputStream(incoming.getInputStream());
                writer =
                        new ObjectOutputStream(incoming.getOutputStream());

                boolean done = false;

                while ( ! done) {
                        try {
                                command = (String)reader.readObject();
```

```
                                        System.out.print("Command: ");
                                        System.out.println(command);

                                        if
                                    (command.equals(MigClient.PROJECTION_BEGIN))

                                                injection();
                                        else if
                                    (command.equals(MigClient.DISCONNECT)) {

                                                incoming.close();
                                                done = true;
                                        }
                                        else

                                                System.out.println("Unknown Command");
                                    }
                                    catch(ClassNotFoundException e) {
                                        System.out.println("Bad Class :" + e);
                                    }
                            }

                            incoming.close();
                        }
                        catch(IOException e) {
                            e.printStackTrace();
                        }
                        catch(Error e) {
                            System.out.println("Mighandler");
                            e.printStackTrace();
                        }
                }
}
```

Figure 4.6. Virtual Machine Server-Side Code.

```
resource Client {
   string id;
   int counter;
   int seed;

   init(string[] arg) {
         id = arg[0];

         if (arg.length() != 3) {
            write("Client: ");
            write(id);
            write(" did not get enough arguments!\n");
         }
         else {
            counter = s2i(arg[1]);
            seed    = s2i(arg[2]);
            while(true) yield();
         }
   }

   …

   portal request() emit Request {
         …
   }

   portal reply(Reply reply) {
         ending(reply.value);
   }
}
```

Figure 4.7. Client Resource.

The Client resource contains two portals. The "request" portal will emit a Request object that contains information indicating that the Client either wants the factorial of a number computed by a Server or the n^{th} number in the Fibonacci sequence

computed by a Server. The "reply" portal is passed a Reply object that contains the result of a previous request. A Client resource is passed three deployment arguments to the init method: the unique identifier of the Client, a counter, and a seed. The counter and the seed are both used by the Client to generate random requests to the Server. The init method executes an infinite loop to ensure that the Client does not exit before all requests are generated.

```
coordinator Transport {
   Server server;
   Client client;

   init(string[] arg) {
         Request request;
         Reply   reply;
         int     value;
         int     FACT = 0;
         int     HALT = -99;

         server = findServer(arg[1]);
         client = findClient(arg[2]);

         request = client.request();
         while(request.request != HALT) {
                  if (request.request == FACT)
                           value = server.fact(request.requestor,  request.value);
                  else
                           value = server.fib(request.requestor,  request.value);

                  reply = new Reply(value);
                  client.reply(reply);
                  yield();
                  request = client.request();
         }
   }
}
```

Figure 4.8. Client-Server Transport Coordinator.

When the Transport coordinator is initialized via the init method, three deployment arguments are passed: the unique identifier of the Transport coordinator which it does not access, the unique identifier of a Client, and a unique identifier of a Server. The unique identifiers are used to find proxy objects for the Server and Client.

```
server = findServer(arg[1]);
client = findClient(arg[2]);
```

The Transport coordinator performs a loop where it initiates communication with the Client resource to get a Request object from the request portal, gives it to the Server resource using the either the fib or fact portal for servicing, and then passes the Reply object back to the reply portal of the Client resource. This loop is performed until the Transport coordinator is told to halt. Communication between the Client and the Server is asynchronous because a single coordinator is used to move data back and forth between the Client and Server.

Reducing the Developer's Burden: Part 2

Each time the portal of a resource is invoked, a pcall instruction will be executed. If the Transport coordinator and the resource (Client or Server) being accessed are not on the same machine, the pcall instruction will automatically handle migrating the Transport coordinator to the location of the resource. Figure 4.9 lists the code built into the L virtual machine to support interprocess communication between a coordinator and a remote resource. If this code were not built into the L virtual machine, a developer would

be responsible for implementing the equivalent of this code for each distributed application. By incorporating into the L virtual machine all the code necessary for interprocess communication between a coordinator and a resource, the developer is allowed to focus on the interaction between resources and coordinators. In addition, the code that implements the Client-Server application (Figure 4.5, Figure 4.7, Figure 4.8) does not become cluttered with calls to interprocess communication routines or calls that initialize the interprocess communication infrastructure.

```
import java.util.Hashtable;
import java.lang.*;
import java.io.*;
import java.net.*;
import java.net.InetAddress;

class MigClient {
        public static final int PORT = 8181;

        // Protocol
        public static final String PROJECTION_BEGIN = "prj";
        public static final String INJECTION_END    = "einj";
        public static final String DISCONNECT       = "dcon";

        private int migSeq;
        private int port;
        private String host;

        private InetAddress       ip     = null;
        private Socket            sock   = null;
        private ObjectOutputStream writer = null;
        private ObjectInputStream  reader = null;

        public static MigClient create(String host, int port) {
                MigClient client = new MigClient(host, port);
                if (client.init())
```

```
                return client;
        else
                return null;
}

private MigClient(String host, int port) {
        this.host = host;
        this.port = port;
}


private boolean init() {
        boolean rtn = true;
        try {
                ip =InetAddress.getByName(host);
                sock= new Socket(ip, port);

                writer =  new ObjectOutputStream(sock.getOutputStream());
                reader = new ObjectInputStream(sock.getInputStream());

        }
        catch(SocketException e){
                System.out.println("MigClient.init " + e);
                rtn = false;
        }
        catch(UnknownHostException e) {
                System.out.println("MigClient.init " + e);
                rtn = false;
        }
        catch(IOException e) {
                System.out.println("MigClient.init " + e);
                rtn = false;
        }

        return rtn;
}


public synchronized boolean projection(GenRec coord,  LStack pstack,
                                        LStack astack,  int pc) {
        boolean rtn = true;
        MigSequence seq = new MigSequence();
        Hashtable migMap = new Hashtable();
```

```
                try {
                        // Remove circular references and
                        // multiple references to the same GenRec
                        // and replace this with a reference to
                        // a MigRec.
                        System.out.println("MigClient.projection capture coordinator");
                        coord.capture(migMap, seq);
                        pstack.capture(migMap, seq);
                        astack.capture(migMap, seq);
                        System.out.println("MigClient.projection capture astack");


                        // Convert and Send Coordinator Execution State
                        System.out.println(PROJECTION_BEGIN);
                        writer.writeObject(PROJECTION_BEGIN);
                        System.out.println("MigClient.projection writeObject
coordinator\n" + coord.toString());
                        writer.writeObject(coord);
                        System.out.println("MigClient.projection writeObject pstack");
                        writer.writeObject(pstack);
                        System.out.println("MigClient.projection writeObject astack");
                        writer.writeObject(astack);
                        System.out.println("MigClient.projection writeInt pc");
                        writer.writeInt(pc);
                        writer.flush();
                        String einj = (String)reader.readObject();
                        System.out.println(einj);
                        if (!einj.equals(INJECTION_END))
                                rtn = false;
                        }
                        catch (ClassNotFoundException e) {
                                System.out.println("Bad Class :" + e);
                                rtn = false;
                        }
                        catch(SocketException e){
                                System.out.println("SocketException " + e);
                                rtn = false;
                        }
                        catch(IOException e){
                                System.out.println("IOException " + e);
                                rtn = false;
                        }
                return rtn;
        } // projection
```

```
        public synchronized boolean disconnect() {
                try {
                        System.out.println(DISCONNECT);
                        writer.writeObject(DISCONNECT);
                        sock.close();
                }
                catch (IOException e) {
                        System.out.println("disconnection " + e);
                        return false;
                }

                return true;
        }

}   // Class Client


import java.util.Hashtable;
import java.util.Collection;

class MigDictionary extends Hashtable {
        private static boolean DEBUG = false;
        private static MigDictionary migd = null;

        public static MigDictionary getMigD() {
                if (migd == null)
                        migd = new MigDictionary();
                        return migd;
                }

        private MigDictionary() {
                super();
        }

        public MigClient get(String server) {
                if (DEBUG) System.out.println("MigDictionary.get " + server);

                MigClient client = (MigClient)super.get(server);
                if (client == null) {
                        client = MigClient.create(server, MigClient.PORT);
                }
```

```
                return client;
        }
}

import java.util.Hashtable;
import java.util.Collection;

class RCDictionary extends Hashtable {
        private static RCDictionary rcd = null;

        /*
         * The Resource Coordinator Dictionary is always a
         * singleton.
         */
        public static RCDictionary getRCD() {
                if (rcd == null)
                        rcd = new RCDictionary();
                return rcd;
        }

        private RCDictionary() {
                super();
        }

        /*
         * Same as find(String, String) but without any
         * error messages.
         */
        public RCDEntry resource(String dclass, String id) {
                RCDEntry entry = null;
                Hashtable idt = (Hashtable)get(dclass);

                if (idt != null) {
                        entry  = (RCDEntry)idt.get(id);
                }
                return entry;
        }

        /*
         * Method to find a particular instance of a resource.
         * dclass:id
         */
        public RCDEntry find(String dclass, String id) {
                RCDEntry entry = null;
```

```
                Hashtable idt = (Hashtable)get(dclass);

                if (idt == null) {
                        System.out.print("Warning: RCDictionary.find unknown class ");
                        System.out.print(dclass);
                        System.out.print(":");
                        System.out.println(id);
                } else {
                        entry  = (RCDEntry)idt.get(id);
                        if (entry == null) {
                                System.out.print("Warning: RCDictionary.find unable to
find ");
                                System.out.print(dclass);
                                System.out.print(":");
                                System.out.println(id);
                        }
                }
        return entry;
        }

        /*
         * Method to find all instance of a resource class
         */
        public String[] find(String dclass) {
                Hashtable idt = (Hashtable)get(dclass);
                String[] ids = null;

                if (idt == null) {
                        System.out.print("Error: RCDictionary.find unknown dclass ");
                        System.out.println(dclass);
                        System.exit(1);
                } else {
                        Object[] o  = (idt.keySet()).toArray();
                        if (o != null) {
                                ids = new String[o.length];
                                for (int i = 0; i < o.length; i++)
                                        ids[i] = (String)o[i];
                        }
                }
                return ids;
        }

        /*
         * Method to populate the the Dictionary with resource
```

```
         * deployment information.
         * If the resource is located on the same host as the
         * dictionary then the RCDEntry will be updated with
         * the real resource GenRec, if the resource is located
         * on another host. Its location will only be registered
         * and the RCDEntry.resource will remain null.
         */
        public void register(String dclass, String id, String server) {
                Hashtable idt = (Hashtable)get(dclass);

                if (idt == null) {
                        idt = new Hashtable();
                        put(dclass, idt);
                }
                idt.put(id, new RCDEntry(server, null));
        }
}
```

Figure 4.9. L Virtual Machine Client-Side Code.

The flow of information for a Client to request the Fibonacci service of a Server is shown in Figure 4.10. In ① a Request object is sent from the Client resource to the Transport coordinator when the Transport invokes the Client request portal. In ② the Transport extracts information from the Client request and determines that the Fibonacci service of the Server resource should be invoked. In ③ information is passed from the Transport to the Server when the Fibonacci portal is invoked. The resulting Fibonacci number is passed back when the portal completes its processing. In ④ the Transport constructs a Reply object for the original Client request. In ⑤ the Transport coordinator passes the reply back to the Client.

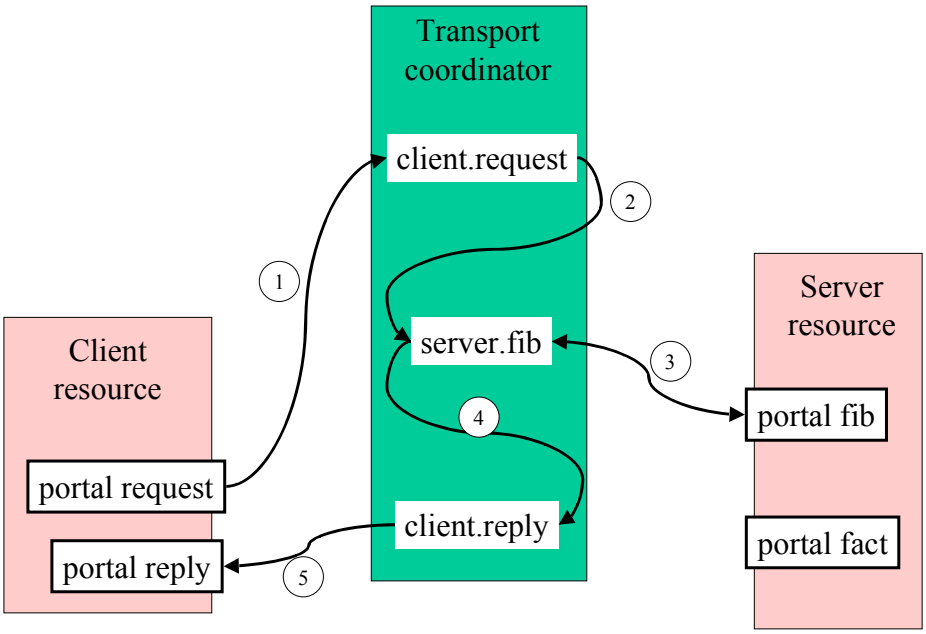


Figure 4.10. Client-Server Information Flow.

Reducing the Developer's Burden: Part 3

To create programs for the Client-Server application now that the source code is written, a deployment file is needed to specific the Client, Server and Transport instances to create and what deployment arguments they will be passed at initialization. An example deployment file that creates two Client resources, two Transport coordinators and a single Server resource is shown in Figure 4.11.

```
//Server:server1@olive.cs.ttu.edu
//Client:client1@ficus.cs.ttu.edu?5&3
//Transport:trans1@ficus.cs.ttu.edu?server1&client1
//Client:client2@hickory.cs.ttu.edu?9&2
//Transport:trans2@hickory.cs.ttu.edu?server1&client2
```

Figure 4.11. Client-Server Deployment File.

A Server resource with unique identifier server1 will be created on host olive.cs.ttu.edu. A Client resource with unique identifier client1 will be created on host ficus.cs.ttu.edu with deployment arguments of 5 and 3. The init method of Figure 4.5 will assign the deployment argument of 5 to the variable counter and 3 to the variable seed. A Client resource with unique identifier client 2 will be created on hickory.cs.ttu.edu with deployment arguments of 9 and 2. One Transport coordinator will be created for each Client resource created. The first with a unique identifier of trans1 is initially deployed to ficus.cs.ttu.edu and has deployment arguments of server1 and client1. The init method of Figure 4.6 will use deployment argument server1 to locate a proxy for the Server resource and client1 to locate a proxy for the Client resource. The second with a unique identifier trans2 is initially deployed to hickory.cs.ttu.edu and has deployment arguments of server1 and client2. The Client-Server application can be deployed to a new configuration by creating a new deployment file and executing the deployer again.

Multiple-Tier Server Application

Full source code listings and program results for the Multiple-Tier Server application can be found in appendix D. The Multiple-Tier Server application was implemented using the same Server resource and Client resources from the Client-Server application. The third resource, named CacheServer (Figure 4.12), was implemented to cache replies generated by the Server. Once the CacheServer has saved a reply, it can be accessed in response to a request without the Server having to recompute the result.

```
resource CacheServer {
   string id;
   CacheTree factTree;
   CacheTree fibTree;


   init(string[] arg) {
         id = arg[0];

         factTree = new CacheTree();
         fibTree  = new CacheTree();
         while(true) yield();
   }

   …

   portal fib(string who, int k) emit int {
         …
   }

   portal fact(string who, int k) emit int {
         …
   }

   portal synchronized fib(int k, int v) {
         fibTree.insert(k, v);
```

```
}

portal synchronized fact(int k, int v) {
      factTree.insert(k, v);
}
      }
```

Figure 4.12. CacheServer Resource.

The CacheServer contains four portals:

1. portal fib(string who, int k) emit int--this portal mimics the interface of the Server's fib portal. The fib portal will emit a solution to the Fibonacci request if it has one stored in the Fibonacci cache tree. If no solution is found in the Fibonacci cache tree, then –999 will be emitted to indicate the solution is unknown.
2. portal fact(string who, int k) emit int--this portal mimics the interface of the Server's fact portal. The fact portal will emit a solution to the factorial request if it has one stored in the factorial cache tree. If no solution is found in the factorial cache tree, then –999 will be emitted to indicate the solution is unknown.
3. portal synchronized fib(int k, int v)--this portal is used to provide a solution to a Fibonacci request. The portal will add an entry into the Fibonacci cache tree with k as a key and v as an associated value. The portal is synchronized so that only one update can be made to the Fibonacci cache tree at a time.

4. portal synchronized fact(int k, int v)--this portal is used to provide a solution to a factorial request. The portal will add an entry into the factorial cache tree with k as a key and v as an associated value. The portal is synchronized so that only one update can be made to the factorial cache tree at a time.

To complete the Multiple-Tier Server application, a coordinator called Transport was implemented. The Transport coordinator for the Multiple-Tier Server application only differs from the Transport coordinator for the Client-Server application in that it will check with the CacheServer for a solution before requesting service from the Server. When a Transport coordinator gets a reply from the Server, it is initially given to the CacheServer for storage before being sent back to the Client that made the original request. To illustrate how easy it is to convert the Client-Server Transport coordinator to the Multiple-Tier Server Transport coordinator, the main body of code for both applications is shown in Table 4.1.

Table 4.1. Client-Server Transport vs. Multiple-Tier Server Transport.

Client-Server Application	*Multiple-Tier Server Application*

server = findServer(arg[1]);
client = findClient(arg[2]);
 	
request = client.request();
while(request.request != HALT) {
  if (request.request == FACT)
	server = findServer(arg[1]);
	client = findClient(arg[2]);
	cache = findCacheServer(arg[3]);
	request = client.request();
	while(request.request != HALT) {
	  if (request.request == FACT) {
	  value = cache.fact(request.requestor,
	  request.value);

Table 4.1. Client-Server Transport vs. Multiple-Tier Server Transport, cont.

Client-Server Application	*Multiple-Tier Server Application*

value = server.fact(request.requestor,
request.value);
 	
else
 	
value = server.fib(request.requestor,
request.value);
 	
reply = new Reply(value);
client.reply(reply);
yield();
request = client.request();
}
	if (value == UNKNOWN) {
	value =
	server.fact(request.requestor,
	request.value);
	cache.fact(request.value, value);
	} }
	else {
	value = cache.fib(request.requestor,
	request.value);
	if (value == UNKNOWN) {
	value = server.fib(request.requestor,
	request.value);
	cache.fib(request.value, value);
	} }
	reply = new Reply(value);
	client.reply(reply);
	yield();
	request = client.request();
	}

The flow of information for a Client to request the Fibonacci service of a Server is shown in Figure 4.13. In ① a Request object is sent from the Client resource to the Transport coordinator when the Transport invokes the Client request portal. In ② the Transport extracts information from the Client request and determines that the Fibonacci service of the Server resource should be invoked. In ③ the CacheServer's Fibonacci portal is invoked to see if the solution has already been computed. In ④ the Transport determines that the Fibonacci service of the Server must be invoked because the CacheServer does not have this solution. In ⑤ information is passed from the Transport to the Server when the Fibonacci portal is invoked. The resulting Fibonacci number is passed back when the portal completes its processing. In ⑥ the Transport prepares to

use the solution returned by the Server to update the CacheServer. In ⑦ the Transport coordinator invokes the CacheServer's Fibonacci portal to update the Fibonacci cache tree. Updating the cache tree will make the solution available to the next Client that makes the same Fibonacci request. In ⑧ the Transport constructs a Reply object to the original Client request. In ⑨ the Transport coordinator passes the Reply object back to the Client.

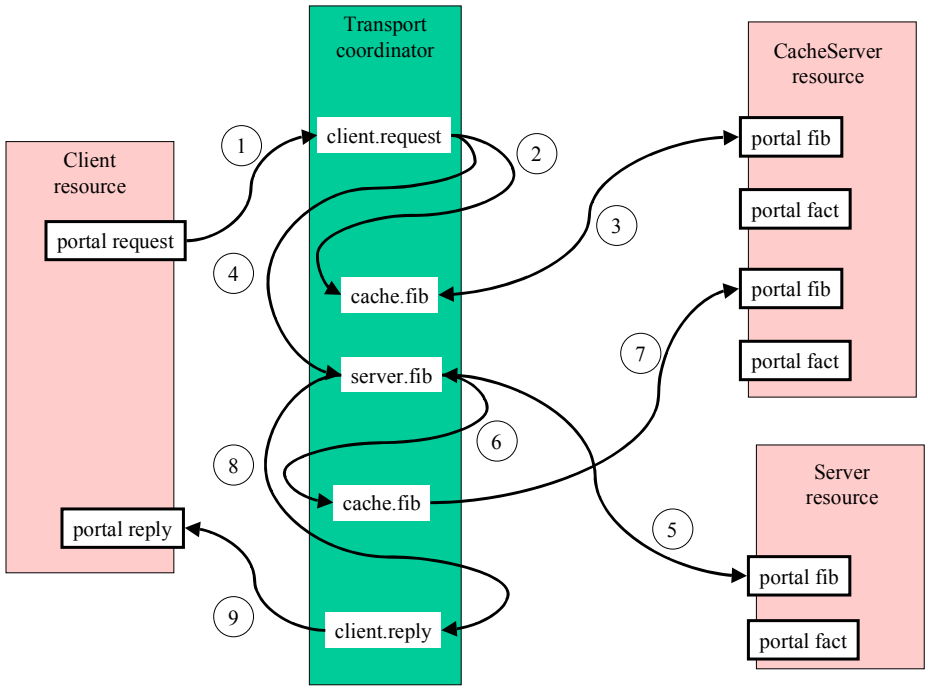


Figure 4.13. Multiple-Tier Server Information Flow.

An example deployment file that creates a Server resource on host olive.cs.ttu.edu, a Client resource on host hickory.cs.ttu.edu, a Transport coordinator on host hickory.cs.ttu.edu, and a CacheServer resource on host ficus.cs.ttu.edu is shown in Figure 4.14.

```
//Server:server1@olive.cs.ttu.edu
//Client:client1@hickory.cs.ttu.edu?17&3
//Transport:trans1@hickory.cs.ttu.edu?server1&client1&cache
//CacheServer:cache@ficus.cs.ttu.edu
```

Figure 4.14. Multiple-Tier Deployment File.

Peer-to-Peer Application

Full source code listings and program results for the Peer-to-Peer application can be found in appendix E. The last distributed architectural model to be implemented is the Peer-to-Peer Model. The Peer-to-Peer application was implemented using a single resource named Peer (Figure 4.15). To construct the Peer resource, the request and reply portals of the Client resource and fib and fact portals of the Server resource from the Client-Server application were merged into a single resource. The most significant change made to the code was in the init method where the unique identifiers for each Peer resource used in an application are passed in as a deployment argument. The unique identifiers for each Peer resource are stored into an array. To allow peer-to-peer communication, each Request object must indicate the service (Fibonacci or factorial) and the Peer resource to perform the service.

```
resource Peer {
   string id;
   int counter;
   int seed;
   int peerCnt;
   string[] peer;

   init(string[] arg) {
      int i;
         id = arg[0];

         if (arg.length() < 4) {
            write("Peer: ");
            write(id);
            write(" did not get enough arguments!\n");
         }
         else {
            peer = new string[arg.length() - 3];
            counter = s2i(arg[1]);
            seed    = s2i(arg[2]);
            i = 3;
            peerCnt = 0;
            while (i < arg.length()) {
                  peer[peerCnt] = arg[i];
                  peerCnt = peerCnt + 1;
                  i = i + 1;
            }
            while(true) yield();
         }
   }
   …
   portal fib(string who, int k) emit int  {   …  }
   portal fact(string who, int k) emit int {  …   }
   portal request() emit Request { … }
   portal reply(Reply reply) { …  }
}
```

Figure 4.15. Peer Resource.

A Peer resource can request itself to perform either the Fibonacci or factorial service. This does not change the operation of the program. Just as there is no difference in the code whether a Peer is local or remote, there is no difference in the code whether a Peer invokes itself or not.

To complete the Peer-to-Peer application, a single coordinator called Transport was implemented. The Transport coordinator for the Peer-to-Peer application is very similar to the one used for the Client-Server application; it differs only in that it must find the Peer to service the request using the Peer's unique identifier which it gets from the request itself. A Transport coordinator is created for each Peer resource deployed. To illustrate how easy it is to convert the Client-Server Transport coordinator to the Peer-to-Peer Transport coordinator, the main body of code for both applications is shown in Table 4.2.

Table 4.2. Client-Server Transport vs. Peer-to-Peer Transport.

Client-Server Application	*Peer-to-Peer Application*

server = findServer(arg[1]);
client = findClient(arg[2]);
 	
request = client.request();
while(request.request != HALT) {
 	
  if (request.request == FACT)
  value = server.fact(request.requestor,
  request.value);
  else
	client = findPeer(arg[1]);
	
	request = client.request();
	while(request.request != HALT) {
	  server = findPeer(request.server);
	  if (request.request == FACT)
	  value = server.fact(request.requestor,
	  request.value);
	  else

Table 4.2. Client-Server Transport vs. Peer-to-Peer Transport, cont.

Client-Server Application	*Peer-to-Peer Application*

value = server.fib(request.requestor,
request.value);
reply = new Reply(value);
client.reply(reply);
yield();
request = client.request();
}
	value = server.fib(request.requestor,
	request.value);
	reply = new Reply(value);
	client.reply(reply);
	yield();
	request = client.request();
	}

The flow of information for a Peer to request the Fibonacci service of another Peer is shown in Figure 4.16. In ① a Request object is sent from an originating Peer to the Transport coordinator when the Transport invokes the originating Peer's request portal. In ② the Transport coordinator extracts information from the Request object and determines that the Fibonacci service of a servicing Peer is to be invoked. In ③ information is passed from the Transport to the servicing Peer when the Fibonacci portal is invoked. The resulting Fibonacci number is passed back when the portal completes its processing. In ④ the Transport creates a Reply object to the originating Peer's request. In ⑤ the Transport passes the reply back to the originating Peer. The basic flow of information is identical to that of the Client-Server application, except that information flows between Peers rather than between Client and Server.

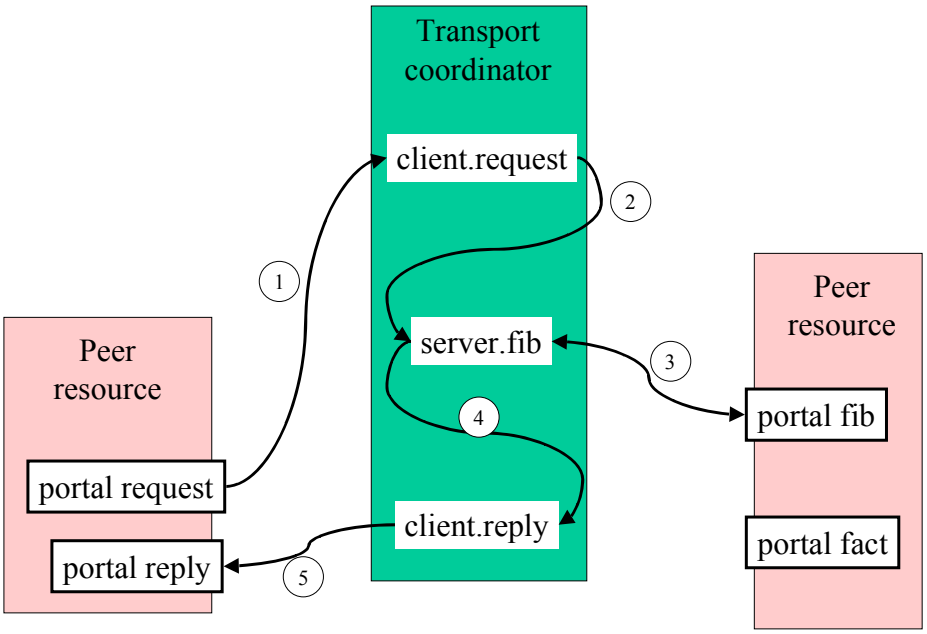


Figure 4.16. Peer-to-Peer Information Flow.

An example deployment file that creates three Peer resources and three Transport coordinators is shown in Figure 4.17.

```
//Peer:peer1@olive.cs.ttu.edu?2&2&peer1&peer2&peer3

//Peer:peer2@ficus.cs.ttu.edu?5&3&peer1&peer2&peer3

//Peer:peer3@hickory.cs.ttu.edu?7&4&peer1&peer2&peer3

//Transport:trans1@olive.cs.ttu.edu?peer1

//Transport:trans2@olive.cs.ttu.edu?peer2

//Transport:trans3@olive.cs.ttu.edu?peer3
```

Figure 4.17. Peer-to-Peer Deployment File.

The three Peer resources are deployed to hosts olive.cs.ttu.edu, ficus.cs.ttu.edu and hickory.cs.ttu.edu. Each Peer resource is given the names of all the Peers to be deployed as deployment arguments. All the Transport coordinators are initially deployed to the host olive.cs.ttu.edu. Each Transport coordinator is passed the unique identifier of one Peer resource. Deploying all the Transport coordinators to a single host helped to solve an initialization and synchronization issue in the Peer-to-Peer application. When Peers are started running, they may begin to communicate with other Peers immediately. This can create an initialization problem in that the other Peers may not have been initialized yet. To overcome this issue, the Peers can be made to synchronize themselves to each other, or a time delay can be created to cause the process to wait until all Peers are up and running. L was easily able to overcome this synchronization issue by deploying all of the Transport coordinators to a single host. As long as an executable containing all the Transport coordinators is started running last, all of the Peer resources will be up and running.

Generative-Communication Application

While a generative-communication application was not implemented, the general features needed to build such an application will be addressed here. Architecturally a Generative-Communication application is very similar to a Client-Server application. The Tuple Client needs three portals: one to emit a read request for the Tuple Server, another to receive a reply back from the Tuple Server, and the last portal to emit a tuple which will be written to the Tuple Server. A Tuple class needs to be created that can be

passed between a Tuple Client resource and the Tuple Server resource. A feature of tuples is that they do not have a fixed format. From an implementation standpoint, this detail can be hidden within the Tuple class itself. The Tuple Server is similar to the Server in the Client-Server application with the added functionality that it can store tuples and then perform an associative search to find relevant tuples in response to a read request. As with the Client-Server application, a single coordinator called Transport needs to be implemented to move tuples between the Tuple Client and the Tuple Server as a result of a read or write request from a Tuple Client.

Implementing the Deliberative/Reactive Strategy

Systems employing the deliberative/reactive strategy often use a three-tier architecture (Figure 2.2). This research will use the following approach to implement a three-tier architecture.

Tier-One: Reactive Layer Resources

A resource is implemented for each local hardware component that must be controlled. The control system implemented for each resource can be based upon subsumption, behavior-based robotics, or any other control strategy that can be implemented in an object-based language.

Tier-Three: Deliberative Layer Resource

A resource is implemented to server as a planner for the system. While a planning system could be implemented directly in L, the more likely scenario would be to interface L with some existing planning system.

Tier-Two: Sequencing Layer Coordinator

The sequencing layer would be implemented as a coordinator. The coordinator would be responsible for moving sensed data from the Reactive Layer Resources to the Deliberative Layer Resource and for taking plans from the Deliberative Layer Resource and using them to enable and disable control loops in the Reactive Layer Resources. Using this strategy, the planning and control system mechanisms are cleanly separated from each other. The policy which is implemented in the Sequencing Layer Coordinator determines how these mechanisms are to be used together to solve a problem. This strategy would allow systems to be constructed that use different pairings of Reactive Layer Resources and Deliberative Layer Resources by simply activating or deactivating different Sequencing Layer Coordinators.

Distributed Systems Challenges

Seven distributed systems challenges (Coulouris et al., 2001) were presented in Chapter II. Which strategy, if any, was used by L to address each of these challenges will now be reviewed.

Heterogeneity

L supports heterogeneity by hiding from an application programmer the details of what computer network, operating system or computer hardware is being used for an application. The L programming language currently has no specific mechanism for interfacing with programs written in other languages. However, the library feature of the L compiler and virtual machine allow the L language to be extended.

Openness

An important feature in maintaining openness in a system is to maintain a separation between mechanism and policy. The programming model for L is based around this separation of mechanism and policy. Within L itself, the primary mechanism for extending the L programming language is through the L library. The policy for using a library operation is defined in the scoping rules for resource specific operations, verses coordinator specific operations and global operations.

Security

Security was not addressed by this research. Research by other authors in the field of distributed system security and mobile code security can be used to address this area.

Scalability

The primary feature for dealing with scalability is the programming model itself. By restricting communication to be coordinator initiated, we have created a model that can maintain a cooperation between N resources with 2N messages. Performance issues still exist with the current state of the art in broadband wireless communication. Yet, it is firmly believed by this author such problems will be rectified in the future.

Failure Handling

While exception handling and the automatic recovery of a coordinator (see appendix I) were not implemented in the prototypes, these two features will provide the basic framework needed to implement a robust failure handling system.

Concurrency

The programming model for L is inherently concurrent with each resource and coordinator always run on a separate thread of control. It is still the responsibility of the developer to establish locks, when necessary, by using the synchronized clause of a method or portal.

Transparency

A fundamental goal of this research was to create a programming model that made developing a distributed application transparent to the developer. To this end, there are no constructs in the language that explicitly deal with interprocess communication,

such as sockets. We will now see how the eight forms of transparency defined in Chapter II are addressed in L.

- Access transparency. The portal of a resource is used identically for local or remote resources.
- Location transparency. The find method for a resource will uniformly retrieve a resource proxy for a resource, whether the resource is local or remote.
- Concurrency transparency. Each resource and coordinator of a system executes concurrently on a separate thread. The synchronized clause can be used to create a lock, which will restrict access to a method or portal to a single thread at any point in time.
- Replication transparency. While multiple instances of resources can be created and deployed throughout a network, there is currently no way to transparently find all resources of a particular type. The find all resources method needs to be implemented to fully support replication transparency.
- Failure transparency. Exception handling and the automatic recovery of a coordinator need to be implemented to support failure transparency.
- Performance transparency. Modifying the deployment file and then repeating the deployment process allows resources to be redistributed throughout the network without altering any code.
- Scaling transparency. A system written in L needs no more than 2N messages to maintain cooperation between N resources.

Capability Primitives

A portal may provide information about the state of a resource (e.g., reading a sensor value) or a portal may cause the resource to change state (e.g., activating an actuator). Both types of portal provide access to the capabilities of a resource, yet these are fundamentally different types of portals. The set of portals that a resource advertises to coordinators is a published interface for that resource. To specify the capabilities of a resource, a frame of reference beyond that of the published interfaces is often needed. To provide this type of context, capability primitives have been identified for cooperative robotic applications. The primitives are used to organize robotic capabilities to make their use more effective in L. This organization will help developers find portals that are needed to accomplish a task by identifying resources that can be used interchangeably. It may also assist them in determining capabilities that are missing from a resource. The primitives presented here are but one possible example of robotic capability primitives and should only be viewed as a starting point that may evolve with each new experience. These primitives were derived by considering what types of information a developer would need to know about the individual capabilities of a robot to be able to use that robot in a cooperative team.

The capability primitives are intended to provide a starting point from which a coordinator can dynamically discover the capabilities of a resource and infer how to use the capabilities. The ultimate goal of this research is to provide the infrastructure that would allow a coordinator to query local resources to find a capability that is needed to accomplish a specific task the coordinator knows how to complete. If a specific

capability is not found that can accomplish part of the task, the coordinator searches for another capability, which is equivalent, or multiple other capabilities that can be combined by the coordinator that would be equivalent. Capabilities, which are determined to be equivalent, need not be equal in their ability to accomplish a task. Thus, a single capability may be more efficient at completing a task than multiple capabilities, which are considered equivalent, that are combined to complete the same task. While work that has been completed in this area will be shown here, the development of the capability primitives is a future research topic.

Example Robotic Capability Primitives

The primitives are divided into the following perspectives:

- Descriptive Capabilities
- Maneuverability Capabilities
- Perception Capabilities
- Skill Capabilities
- Off-line Reasoning Capabilities.

Descriptive

The descriptive capabilities define static attributes of a robotic system. Environment capabilities define the physical environment in which a robotic system can operate to be identified. Some systems can operate in more than a single environment. An arboreal environment would indicate a robotic system that could climb. A fossorial

environment would indicate a robotic system that could burrow or dig underground. Structural capabilities define the physical attributes of a robotic system. The Operating capabilities define general properties that govern the use of a robotic system. This grouping includes information such as emissions from the robot, how much power the robot needs to operate, and the type of power source used by the robot: battery, solar power, or combustion engine.

- Environment
    - Aerial
    - Aquatic
    - Arboreal
    - Fossorial
    - Terrestrial
- Structural
    - Chassis
    - Limbs
    - Tools
- Operating
    - Emissions
    - Power Consumption
    - Power Source

Maneuverability

The maneuverability capabilities define dynamic characteristics about the actuators of a robotic system. When directional and rotational forces are combined, the movement of a three-dimensional body can be described (Figure 4.18). Dynamical systems theory capabilities define the forces needed to move the robot.

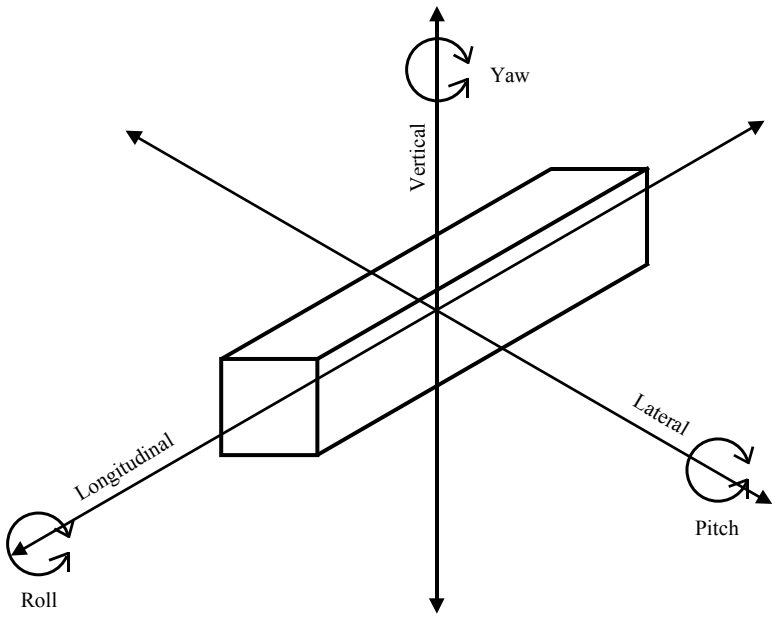


Figure 4.18. Directional and Rotational Forces.

- Directional Force
  - Longitudinal
  - Vertical
  - Lateral

- Rotational Force
    - Yaw
    - Pitch
    - Roll
- Dynamical Systems Theory
    - Kinetics
    - Kinematics
    - Linear Maneuverability Number
    - Propulsor Effectiveness Number

Perception

The perception capabilities define the sensors a robotic system can have. Extended touch, extended hearing, extended taste/smell and extended vision define sensory capabilities that extend the basic capabilities of a human.

- Extended Touch
    - Bump
    - Texture
    - Torque
    - Pressure
    - Temperature
- Extended Hearing
- Extended Taste/Smell

    - pH
    - Oxygen
    - Carbon Monoxide
- Extended Vision
    - Radio
    - Microwaves
    - Infrared
    - Visible Light
    - Ultraviolet
    - X-Rays
    - Gamma Rays
- Electrical
- Magnetic
- Spatial
    - Position
    - Displacement
    - Velocity
    - Acceleration

Skill (nth order)

The skill capabilities define various behaviors that interact with the environment. In general, these behaviors pair sensors with actuators. The term "nth order skill" is used to indicate that skills can be layered upon each other to create high-level skills.

Communication capabilities may at first seem out of place as a skill. However, the general view that skills are just pairing of sensors and actuators works for communication as well. To draw this analogy, we simply view a message sent to a robot as an external stimulus that is perceived by a communication sensor. The robot then reacts to the stimulus by using conventional actuators or a communication actuator that produces a new message (stimulus).

- Reactive Navigation
    - Path Following
    - Exploration
    - Docking
- Fabrication
    - Welding
    - Grinding
    - Painting
    - Drilling
    - Cutting
- Material Handling
    - Hauling
    - Retrieval
    - Stacking
- Assembly
- Security

    - Sentry
    - Reconnaissance
- Communication
    - OSI Model

Off-line Reasoning

Off-line reasoning capabilities define behaviors that are not considered reactive because the behaviors tend to work as batch processes without any real-time feedback from sensors. While these behaviors often use inputs that must be gathered from a sensor, there is little need to connect the behavior directly with an actuator. These behaviors do not directly pair sensors and actuators as do skill capabilities.

- Path Planning
    - Topological
    - Metric
- Mapping
    - Topological
    - Metric
- Map Registration
- Encryption/Decryption

Observations and Guidelines

"The earliest known system of classification is that of Aristotle, who attempted in the 4th cent. B.C. to group animals according to such criteria as mode of reproduction and possession or lack of red blood. Aristotle's pupil Theophrastus classified plants according to their uses and methods of cultivation" (Linnaeus, 2005, p. 1). In the "mid-1700s, Swedish Biologist [Carols Linnaeus] established a simple system for classifying and naming organisms. He developed a Hierarchy (a ranking system) for classifying organisms [based on anatomical resemblance] that is the Basis for Modern Taxonomy" (Johnson, 2005, p. 1). With the advent of microbiology and genetics, a new taxonomy has been developed to classify living organisms. PhyloCode "is designed to name the parts of the tree of life by explicit reference to phylogeny [(the evolutionary relationship between organisms)]" (PhyloCode, 2005, p. 1).

In researching how classifications and taxonomies are derived, the following observations were made:

- few taxonomies are based on natural laws,
- a taxonomy almost always reflects a perspective of its creator,
- a taxonomy is only as good as the data from which it was derived.

This does not mean that a taxonomy (classification) is useless; rather if the classification can help humans study a particular field and the classification is consistent with known data in that field, it provides value. The principle guideline that this research has brought to this author is that a taxonomy or classification should always be viewed as a dynamic

structure that will evolve with the discovery of new data. Thus, the perfect taxonomy of today may become the relic of tomorrow.

CHAPTER V

CONCLUSIONS AND FUTURE RESEARCH

This chapter presents conclusions and observations from this research, as well as a number of possible future research directions.

Conclusions

The goals of this research were to reduce the complexity of developing a distributed application and to maintain tractable communication between cooperating agents. These goals have been met. The results of this research are a programming model, an execution model, a robotic capability primitives, and example applications written in L. Conclusions and observations made regarding this research are:

- The example applications written in L validated that there is no difference in how a developer creates a distributed application when compared to the development of a stand-alone application. The example applications were written and tested on a stand-alone PC. Later these applications were moved to the UNIX environment at Texas Tech University and run as a distributed application. No changes were made to the source code to support this port to a distributed UNIX environment. The only change that was necessary to port the application to a distributed environment was to modify the deployment file and re-execute the deployment process to create new executable programs.

- Process migration has proven to be a viable alternative to broadcast communication for maintaining tractable communication between cooperative robots. The complexity for a team to reach a consensus for broadcast communication has been estimated at $2^N - 1$ distinct decisions; the estimate for a team to reach a consensus using the computational mobility-based approach is 2N. The number of messages that must be exchanged for the same team to reach a consensus has been estimated at only 2N messages.
- The programming model and execution model for L made supporting multiple coordinators uncomplicated. It would have required significantly more effort to have restricted L to support only a single coordinator than it took to support multiple coordinators.
- Creating a virtual machine that could support process migration was more straightforward than originally anticipated. All of the code that is required to support process migration is hidden within a single instruction (i.e., pcall) of the L virtual machine.
- Restricting coordinators from accessing all I/O operations increased the difficulty of debugging an application. While a coordinators cannot be allowed to access local hardware for reasons already discussed, allowing a coordinator to invoke a print operation would have been very helpful during testing. Such a feature could be implemented without jeopardizing the ability of a coordinator to migrate.

Future Research

Optimizing L

Techniques such as peek-hole optimizing were used sparingly on the initial L compiler. A fully optimized compiler and a virtual machine need to be constructed to allow performance evaluations to be performed against other languages. Because of L's unique execution model, optimization techniques not normally associated with compilers or virtual machines may prove to be useful in enhancing the performance of applications written in L.

Memory Management

The marshalling and unmarshalling of individual data structures in order to migrate the execution state of a coordinator is a significant overhead. In particular, the need to scan each data structure checking for cycles is time consuming. An alternative method to marshalling and unmarshalling individual data structures is to allocate a single block of memory that contains the execution state of a coordinator. To migrate a coordinator, the entire block of memory is transmitted as if it were one continuous byte array.

Automatic Recovery of a Coordinator

Multi-Agent Systems (MAS) that utilize a solution where a single agent is selected as a leader are considered more brittle than systems utilizing multiple distributed agents that share the leadership role because the loss of the single leader can paralyze the

entire team. To address the issue of a catastrophic failure of a coordinator, L has been designed to automatically recover a coordinator. This feature was not implemented in the L prototype, but the basic strategy is documented in appendix I. The basic strategy should be refined and implemented so that an evaluation of its use can be performed.

Empirical Studies

While simulations have their place in robotics research, it is often impossible to conclude that a solution is a viable and practical solution until that solution is tried out on a physical robot operating in a real world environment. To fully evaluate the capabilities and limitations of using L to solve cooperative robotics problems, trials need to be performed with physical robots. Performing trials with physical robots is a serious undertaking and will require a significant amount of time and money. The future work to be presented here is a phased undertaking. In the initial phase, a simulation should be constructed for the selected problem. Simulating the problem will allow basic algorithms to be devised and tested in a controlled environment. After a simulation is used to determine that the approach is sound, physical robots should be used to validate the approach. Within the Cooperative Robotics community, a collection of problems have begun to emerge as classical cooperative problems:

1. Formation/Marching Problems require N robots to organize themselves into a formation and then move while remaining in formation. The complexity of the problem is generally increased by:
    - varying the terrain over which the robot formation must march,

- using non- holonomic robots (robots that cannot spin in place),
- increasing the number of robots.

2. Box-Pushing Problems require N robots to organize themselves into a formation relative to a fixed obstacle and then to move the obstacle to a new location. The complexity of the problem is generally increased by:
    - varying the terrain over which the robots must move an object,
    - requiring the box to be carried vs. pushed,
    - increasing the number of robots.
3. Foraging Problems require N robots to organize themselves into a team to collect some object(s) scattered in an environment. When the object(s) to be collected are given the ability to move, foraging problems are generally referred to as Herding or Predator/Prey Problems (Benda, Jagannathan, and Dodhiawalla, 1995). The complexity of the problem is generally increased by:
    - varying the legal moves a predators can make,
    - varying the legal moves a prey can make,
    - varying the intelligence of the prey,
    - varying the size of the environment,
    - varying the number of predators,
    - varying the number of prey.
4. Robotic Soccer problems (Kim and Vadakkepat, 2000; Mackworth, 1993) require two teams of N robots to compete against each other in a game of

soccer. Robotic Soccer is considered the most complex of the cooperative robotics problems. Robot Soccer is considered a "complex real world domain" (Balch and Parker, 2002).

Distributed Learning

Machine Learning using physical robots is a difficult problem due to the large number of trails that a robot must perform before it learns a solution and the complexity of sharing information between distributed robots. Investigation whether the language L would provide any benefits over currently adopted techniques would appear to be a worthwhile research task.

Coordinator vs. Resource Specific Languages

A choice was made to start L with an object-based[6] language as a baseline and then modify the grammar as needed. Both the coordinator and resource construct started from this same baseline. However, coordinators and resources perform very different functions in L. As such, each construct could possibly benefit from a language specifically designed for the types of problems (control system vs. MAS) they address.

[6] An object-based language supports the creation of classes and objects but does not support inheritance.

Capability Primitives

Continue work to flush out the capability primitives. Develop the infrastructure needed to support dynamic capability discovery and to determine one-to-one and one-to-many capability equivalence.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers Principles, Techniques, and Tools*; Addison-Wesley, 1986.

ANSA. *The Advanced Network System Architecture (ANSA) Reference Manual*, Castle Hill, Cambridge, England: Architecture Project Management, 1998.

Arnold, K., Gosling, J., and Holmes, D. *The Java Programming Language*, 3rd; Addison-Wesley, 1999.

Artsy, Y. and Finkel, R. "Designing a Process Migration Facility: The Charlotte Experience," IEEE Computer, 22: 9, pp. 47-56, September 1989.

Bailey, K. *Social Entropy Theory*; Albany, NY: State University of New York Press, 1990.

Balch,T. and Parker, L., Eds. *Robot Teams: From Diversity to Polymorphism*, A. K. Peters, 2002.

Barak, A. and Wheeler, R. "MOSIX: An Integrated Multiprocessor UNIX," *Proceedings of the USENIX Winter 1989 Technical Conference*, pp. 101-112, February 1989.

Beckers, R., Holland, O.E., and Deneubourg, J. "From Local Actions to Global Tasks: Stigmery and Collective Robotics." *In Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, R. Brooks and P. Maes, Eds., pp. 181-189, Cambridge, MA: MIT Press, 1994.

Benda, M., Jagannathan, V., and Dodhiawalla, R. "On Optimal Cooperation of Knowledge Sources−An Empirical Investigation," Technical Report. Bellevue, WA: Boeing AI Center, 1995.

Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Guernic, P. L., and De Simone, R. "The Synchronous Languages 12 years Later," *Proceedings of the IEEE*, Vol. 91, No. 1, January 2003.

Berry, G. "The Foundations of Esterel." Retrieved on 4/15/04 from World Wide Web: ftp://ftp.esterel.org/esterel/pub/papers/foundations.pdf.

Brooks, R. "A Robust Layered Control System for a Mobile Robot," IEEE Journal of Robotics and Automation, RA-2, April 1986, pp. 14-23.

Cabri, G., Leonardi, L., and Zambonelli, F. "Mobile-Agent Coordination Models for Internet Applications," IEEE Computer, vol. 33, no. 2, pp. 82-89, Feb. 2000.

Ciancarini, P. and Kielmann,T. "Coordination Models and Languages for Parallel Programming," 2005. Retrieved on 3/11/05 from World Wide Web: http://mason.gmu.edu/~kmills/swe622/RelatedPapers/paper12.pdf.

Cooke, D.E. *A Concise Introduction to Computer Language: Design, Experimentation and Paradigms*; Pacific Grove, CA: Brooks/Cole Publishers, 2002.

Coulouris, G., Dolimore, J., and Kindberg, T. *Distributed Systems Concepts and Design*; Addison-Wesley, 2001.

Douglis, F. and Ousterhout, J. "Transparent Process Migration: Design Alternatives and the Sprite Implementation," Software-Practice and Experience, 21(8): 757-785, August 1991.

Feddema, J.T., Lewis, C., and Schoenwald, D.A. "Decentralized Control of Cooperative Robotic Vehicles: Theory and Application," IEEE Transactions on Robotics and Automation, Vol. 18, No. 5, Oct. 2002.

Fischer, C. N., and LeBlanc, R. J. Jr., *Crafting a Compiler*; Benjamin/Cummings, 1988.

Forman, G. H. and Zahorjan, J. "The Challenges of Mobile Computing," Computer, 27(4): 38-47, April 1994.

Foster, C. "UCSD Pascal and the PC Revolution." Retrieved on 1/28/05 from World Wide Web: http://alumni.ucsd.edu/magazine/vol1no3/features/**p**ascal.htm.

Fuggetta, A., Picco, G. P., and Vigna, G. "Understanding Code Mobility," IEEE Transactions on Software Engineering, Vol. 24, No. 5, May 1998.

Gat, E. "On-Three Layered Architectures," *Artificial Intelligence and Mobile Robots,* David Kortenkamp, R. Peter Bonnasso, and Robin Murphy, Eds. MIT Press, 1998.

Gaughan, E. D. and Hall, C. E. *College Algebra and Trigonometry*; Brooks/Cole Publishing Company, 1984.

Gelernter, D. "Generative Communication in Linda," ACM Trans. Prog. Lang. Systems, vol. 7, no. 1, pp. 80-112, 1985.

Goodaire, E. G. and Parmenter, M. M. *Discrete Mathematics with Graph Theory*, 2nd; Prentice Hall, Inc., 2002.

Halbwachs, N. *Synchronous Programming of Reactive Systems*; Kluwer Academic, 1993.

Hennessy, J. and Patterson, D. *Computer Architecture: A Quantitative Approach*, 3rd Edition; Morgan Kaufmann, 2003.

Hoare, C. A. R. "Hints on Programming Language Design," *Sigact/Siglan Symposium on Principles of Programming Languages*; Oct. 1973.

Horstmann, C. and Cornell, G. *Core Java 1.1 Volume II – Advanced Features*, Sun Microsystems Press, 1998.

IBM VM. "Short History of IBM's Virtual Machines." Retrieved on 3/3/05 from World Wide Web: http://www.cap-lore.com/CP.html.

ISO. International Standards Organization. *Basic Reference Model of Open Distributed Processing, Part 1: Overview and guide to use*. ISO/IEC JTC1/SC212/WG7 CD 10746-1, International Standards Organization, 1992.

Johnson, J. G. "Biology I." Retrieved on 07/05/05 from World Wide Web: http://www.sirinet.net/~jgjohnso/classification.html.

Johnsonbaugh, R. *Discrete Mathematics*, Macmillan Publishing Company, 1986.

Jones, D-W. "The Taxonomy Revolt." Retrieved on 4/4/05 from World Wide Web: http://www.ridgenet.net/~do_while/sage/v7i6f.htm, March 2003.

Jul, E., Levy, H., Hutchinson, N., and Black, A. "Fine-Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems*, 6(1):109-133, February 1988.

JVM. "Java Virtual Machine." Retrieved on 3/3/05 from World Wide Web: http://java-virtual-machine.net/sun-java-virtual-machine.html.

Kim, J.H. and Vadakkepat, P. "Multi-Agent System: A Survey from the Robot-soccer Perspective," *International Journal Intelligent Automation and Soft Computing*, 6: (1), 3-17, 2000.

Klavins, E. *Communication Complexity of Multi-Robot Systems*; In WAFR '02, Nice, France, December 2002.

Kohlbrenner, E., Morris, D., and Morris, B. "The History of Virtual Machines." Retrieved on 2/2/05 from World Wide Web: http://www.cne.gmu.edu/itcore/virtualmachine/history.htm.

Leestma, S. and Nyhoff, L. *Pascal: Programming and Problem Solving*; Macmillan Publishing Company, 1984.

Lewis, H. R. and Papadimitriou, C. H. *Elements of the Theory of Computation*; Prentice-Hall, Inc., 1981.

Lindholm, T. *The Java Virtual Machine Specification*; Addison-Wesley, 1999.

Linnaeus, C. "Classification History." *The Columbia Electronic Encyclopedia,* 6th ed. Columbia University Press, 2005.

Litzkow, M. and Solomon, M. "Supporting Checkpointing and Process Migration outside the UNIX Kernel," *Proceedings of the USENIX Winter Conference*, pp. 283-290, January 1992.

Lui, J. and Wu, J. *Multi-Agent Robotic Systems*, CRC Press, 2001.

Mackworth, A.K. "On Seeing Robots," *In Computer Vision: Systems, Theory, and Applications*, A. Basu and X. Li, eds.; 1-13, Singapore: World Scientific Press, 1993.

Mano, M. M. *Computer System Architecture*, 3rd Edition; Prentice-Hall, 1993.

Mataric, M. "Behavior-Based Systems: Main Properties and Implications," IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems, 1992.

Mataric, M. "Behavior-Based Control: Examples from Navigation, Learning, and Group Behavior," *Journal of Experimental and Theoretical Artificial Intelligence*, special issue on Software Architectures for Physical Agents; 9(2-3), H. Hexmoor, I. Horswill, and D. Kortenkamp, eds., 1997, 323-336.

Milojičić, D., Douglis, F., and Wheeler, R. (Eds*), Mobility: Processes, Computers and Agents*; Addison-Wesley, 1999.

MIPS R2000 - Instruction Reference. Handout from Dr. Larry Pyeatt's Computer Systems Organization and Architecture class, Texas Tech University, Spring, 2003.

Murphy, R. R. *Introduction to AI Robotics*; MIT Press, 1998.

Muscettola, N., Nayak, P. P., Pell, B., and Williams, B. C. "Remote Agent: To Boldly Go Where No AI System Has Gone Before," Artificial Intelligence, 103(1/2), August 1998.

Nerode, A. and Shore, R. A. *Logic for Applications*, 2nd; Springer-Verlag, 1997.

Pascal. "Pascal History." Retrieved on 2/2/04 from World Wide Web: www.taoyue.com/tutorials/pascal/history.html.

PhyloCode. Retrieved on 07/05/05 from World Wide Web: http://www.ohiou.edu/phylocode.

Powell, M. and Miller, B. P. "Process Migration in DEMOS/MP," *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pp. 110-119, October 1983.

Pratt, T., Zelkowitz, W., and Marvin, V. *Programming Languages: Design and Implementation*, 4th ed.; Prentice-Hall, Inc. 2001.

Ranganathan, M., Acharya, A., Sharma, S., and Saltz, J. "Network-aware Mobile Programs," *Proceedings of the USENIX 1997 Conference*, pp. 91-103, January 1997.

Robertson, P. "The Symbolics Virtual Lisp Machine Or Using The Dec Alpha As A Programmable Micro-engine," Submitted to PLDI 1994. Retrieved on 2/5/04 from World Wide Web: http://pt.withy.org/publications/VLM.html.

Rubini, A. and Corbet, J. *Linux Device Drivers*; O'Reilly Media, 2001.

Shannon, C. E. *The Mathematical Theory of Communication*; Urbana, IL: University of Illinois Press, 1949.

Shoch, J. F. and Hupp, J. A. "The 'Worm' Programs-Early Experience with a Distributed Computation," *Comm. of the ACM*, 25(3):172-180, March 1982.

Smith, P. and Hutchinson, N.C. "Heterogeneous Process Migration: The Tui System," *Software-Practice and Experience*, 28(6):611-639, May 1998.

Tanenbaum, A. S. and van Steen, M. *Distributed Systems Principles and Paradigms*, Prentice Hall, 2002.

Weld, D. W. and de Kleer, J., Editors. *Reading in Qualitative Reasoning About Physical System;* Morgan Kaufmann Publishers, 1990.

Werger, B. "Ayllu: Distributed Port-Arbitrated Behavior-Based Control," *Distributed Autonomous Robotic Systems.* Lynne E. Parker, George Bekey, and Jacob Barhen (eds.); Springer, 2000:25-34.

White, J.E. "Telescript Technology: Mobile Agents," *General Magic White Paper*; Appeared in Bradshaw, J., Software Agents, AAAI/MIT Press, 1996.

Williams, B. C. and Nayak, P. P. "Immobile Robots: AI in the New Millennium," AI Magazine, 17(3), pp. 16-35, Fall 1996.

Williams, B. C., Kim, P., Hofbaur, M., How, J., Kennell, J., Loy, J., Ragno, R., Stedl, J., and Walcott, A. "Model-based Reactive Programming of Cooperative Vehicles for Mars Exploration," International Symposium on Artificial Intelligence, Robotics and Automation in Space; St-Hubert, Canada, June 2001.

Williams, B. C., Ingham, M. D., Chung, S. H., and Elliott, P. H. "Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers," *Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software*, vol. 9, no. 1, pp. 212-237, June 2003.

Wu, J. *Distributed System Design*; CRC Press, 1999.

APPENDIX A

L GRAMMAR

Conventions

NONTERMINALS: ALL UPPERCASE

terminals: all lowercase

→: defined as

∈: empty production rule

Reserved Words

boolean	catch	class	coordinator	else
emit	false	if	init	int
new	null	portal	private	public
real	resource	return	string	
throw	true	try	while	

Compilation Units

COMPILATION_UNIT

→ COORDINATOR_DECLARATION COMPILATION_UNIT_Z

→ RESOURCE_DECLARATION COMPILATION_UNIT_Z

→ CLASS_DECLARATION COMPILATION_UNIT_Z

COMPILATION_UNIT_Z

→ COMPILATION_UNIT

→ ∈

Declarations

COORDINATOR_DECLARATION

→ coordinator IDENTIFIER

{ RC_VARIABLE_DECLARATION_Z INITIALIZATION }

RESOURCE_DECLARATION

→ resource IDENTIFIER

{ RC_VARIABLE_DECLARATION_Z INITIALIZATION

PORTAL_Z METHOD_ DECLARATION_Z }

CLASS_DECLARATION

→ class IDENTIFIER USAGE { FIELD_DECLARATION_Z }

USAGE

→ resource specific

→ coordinator specific

→ ∈

INITIALIZATION

→ init (string[] IDENTIFIER) { STATEMENT_Z }

RC_VARIABLE_DECLARATION_Z

→ RC_VARIABLE_DECLARATION

→ ∈

CLASS_VARIABLE_DECLARATION_Z

→ CLASS_VARIABLE_DECLARATION

→ ∈

FIELD_DECLARATION_Z

→ CONSTRUCTOR_DECLARATION FIELD_DECLARATION_Z

→ METHOD_DECLARATION FIELD_DECLARATION_Z

→ CLASS_VARIABLE_DECLARATION FIELD_DECLARATION_Z

→ ; FIELD_DECLARATION_Z

→ ∈

SYNC_X

→ synchronized

→ ∈

RC_VARIABLE_DECLARATION

→ VARIABLE_DECLARATION ;

METHOD_DECLARATION_Z

→ METHOD_DECLARATION METHOD_DECLARATION_Z

→ ∈

METHOD_DECLARATION

→ MODIFIER SYNC_X MTYPE IDENTIFIER (PARAMETER_LIST_X)

{ LOCAL_VARIABLE_DECLARATION STATEMENT_Z }

CONSTRUCTOR_DECLARATION

→ MODIFIER IDENTIFIER (PARAMETER_LIST_X)

{ LOCAL_VARIABLE_DECLARATION STATEMENT_Z }

CLASS_VARIABLE_DECLARATION

→ MODIFIER VARIABLE_DECLARATION ;

LOCAL_VARIABLE_DECLARATION_Z

→ LOCAL_VARIABLE_DECLARATION LOCAL_VARIABLE_DECLARATION_Z

→ ∈

MODIFIER

→ public

→ private

LOCAL_VARIABLE_DECLARATION

→ VARIABLE_DECLARATION VARIABLE_DECLARATION_X_INIT ;

VARIABLE_DECLARATION

→ TYPE IDENTIFIER

VARIABLE_DECLARATION_X_INIT

→ = NEW_EXPRESSION

→ = EXPRESSION

→ ∈

PARAMETER_LIST_X

→ PARAM PARAMETER_Z

→ ∈

PARAMETER_Z

→ PARAM_PARAMETER_Z

→ ∈

PARAM

→ TYPE IDENTIFIER

EMIT_X

→ emit TYPE

→ ∈

PORTAL_Z

→ portal SYNC_X IDENTIFIER (PARAMETER_LIST_X) EMIT_X

{ LOCAL_VARIABLE_DECLARATION_Z STATEMENT_Z } PORTAL_Z

→ ∈

Expressions

EXPRESSION

→ (EXPRESSION)

→ true EXPRESSION_Z

→ false EXPRESSION_Z

→ null EXPRESSION_Z

→ NUMERIC EXPRESSION _Z

→ MEMBER MEMBER_REF_Z EXPRESSION_Z

EXPRESSION_Z

→ == EXPRESSION

→ && EXPRESSION

→ || EXPRESSION

→ <= EXPRESSION

→ >= EXPRESSION

→ != EXPRESSION

→ = NEW_EXPRESSION

→ = EXPRESSION

→ < EXPRESSION

→ > EXPRESSION

→ + EXPRESSION

→ - EXPRESSION

→ * EXPRESSION

→ / EXPRESSION

→ % EXPRESSION

→ ∈

NEW_EXPRESSION

→ new NEW_EXPRESSION_KIND

NEW_EXPRESSION_KIND

→ CLASS_NAME (ARG_LIST_X)

→ TYPE_SPECIFIER EXPRESSION _ARRAY

MEMBER

→ IDENTIFIER EXPRESSION_FX_ID

Expression Fx Id

→ (ARG_LIST_X)

→ [EXPRESSION]

MEMBER_REF_Z

→ . MEMBER MEMBER_REF_Z

→ ∈

EXPRESSION_ARRAY

→ [EXPRESSION]

ARG_LIST_X

→ EXPRESSION

→ ∈

ARG_LIST_Z

→ EXPRESSION ARG_LIST_Z

→ ∈

Statements

STATEMENT

→ { STATEMENT_Z }

→ if (EXPRESSION) STATEMENT STATEMENT_X_ELSE

→ while (EXPRESSION) STATEMENT

→ try STATEMENT STATEMENT_X_CATCH_LIST

→ throw EXPRESSION

→ return STATEMENT_X_EXPRESSION

→ ;

→ [STATEMENT_Z]

→ EXPRESSION ;

STATEMENT_Z

→ STATEMENT STATEMENT_Z

→ ∈

STATEMENT_X_ELSE

→ else STATEMENT

→ ∈

STATEMENT_X_EXPRESSION

→ EXPRESSION

→ ∈

STATEMENT_X_CATCH_LIST

→ STATEMENT_CATCH STATEMENT_X_CATCH_LIST

→ ∈

STATEMENT_CATCH

→ catch (PARAM) STATEMENT

APPENDIX B

COMPILER GENERATED JAVA SOURCE AND L BYTECODE

The L compiler generates bytecode as well as Java source code as output. To better illustrate this process, the transformation done on L source code will be shown for various constructs of the language.

Assignment Statement

L Source Code

```
i = i + 1;
```

L Bytecode

```
addi, v40, v37, int5
move, v37, v40, 0
```

Java Encoding

```
new Quad( "500", "^4", "^2", "6" ), // addi, v45, v42, int5   pc=6
new Quad( "100", "^2", "^4", "0" ), // move, v42, v45, 0   pc=7
```

If Statement

L Source Code

```
if (i >= 12) {
```

```
        i = i + 10;
    }
```

L Bytecode

```
    sgei, v56, v54, int9   pc=4
    bnz, 8, v56, 0   pc=5
    addi, v57, v54, int10   pc=6
    move, v54, v57, 0   pc=7
```

Java Encoding

```
    new Quad( "702", "^3", "^2", "5" ), // sgei, v56, v54, int9   pc=4
    new Quad( "10", "8", "^3", "0" ), // bnz, 8, v56, 0   pc=5
    new Quad( "500", "^4", "^2", "6" ), // addi, v57, v54, int10   pc=6
    new Quad( "100", "^2", "^4", "0" ), // move, v54, v57, 0   pc=7
```

While Statement

L Source Code

```
    while (i < 70) {
        i = i + 10;
    }
```

L Bytecode

```
    slti, v49, v47, int6
```

```
bnz, 9, v49, 0
addi, v50, v47, int7
move, v47, v50, 0
jmp, 4, 0, 0
```

Java Encoding

```
new Quad( "704", "^3", "^2", "5" ), // slti, v49, v47, int6   pc=4
new Quad( "10", "9", "^3", "0" ), // bnz, 9, v49, 0   pc=5
new Quad( "500", "^4", "^2", "6" ), // addi, v50, v47, int7   pc=6
new Quad( "100", "^2", "^4", "0" ), // move, v47, v50, 0   pc=7
new Quad( "11", "4", "0", "0" ), // jmp, 4, 0, 0   pc=8
```

Function Declaration

L Source Code

```
private int f(int j) {
        return j + 5;
}
```

L Bytecode

```
frame, md13, 0, 0
pop, v82, 0, 0
pop, g, 0, 0
```

addi, v84, v82, int13

return, v84, 0, 0

Java Encoding

```
new Quad( "210", "3", "0", "0" ), // frame, md13, 0, 0   pc=14
new Quad( "401", "^1", "0", "0" ), // pop, v82, 0, 0   pc=15
new Quad( "401", "^0", "0", "0" ), // pop, g, 0, 0   pc=16
new Quad( "500", "^2", "^1", "5" ), // addi, v84, v82, int13   pc=17
new Quad( "220", "^2", "0", "0" ), // return, v84, 0, 0   pc=18
```

Function Call

L Source Code

```
i = f(i);
```

L Bytecode

```
push, g, 0, 0
push, v81, 0, 0
call, 14, 0, 0
pop, v83, 0, 0
move, v81, v83, 0
```

Java Encoding

```
new Quad( "400", "^0", "0", "0" ), // push, g, 0, 0   pc=8
new Quad( "400", "^2", "0", "0" ), // push, v81, 0, 0   pc=9
new Quad( "200", "14", "0", "0" ), // call, 14, 0, 0   pc=10
new Quad( "401", "^3", "0", "0" ), // pop, v83, 0, 0   pc=11
new Quad( "100", "^2", "^3", "0" ), // move, v81, v83, 0   pc=12
```

Finding a Resource

L Source Code

```
Writer w = findWriter("test");
```

L Bytecode

```
push, g, 0, 0
push, string5, 0, 0
find, dic25, 0, 0
pop, v101, 0, 0
move, v97, v101, 0
```

Java Encoding

```
new Quad( "400", "^0", "0", "0" ), // push, g, 0, 0   pc=3
new Quad( "400", "5", "0", "0" ), // push, string5, 0, 0   pc=4
new Quad( "1300", "5", "0", "0" ), // find, dic25, 0, 0   pc=5
```

```
new Quad( "401", "^5", "0", "0" ), // pop, v101, 0, 0   pc=6
new Quad( "100", "^4", "^5", "0" ), // move, v97, v101, 0   pc=7
```

Portal Call

L Source Code

```
j = w.f(k);
```

L Bytecode

```
push, v102, 0, 0
push, v95, 0, 0
pcall, 20, 0, 0
pop, v103, 0, 0
move, v96, v103, 0
```

Java Encoding

```
new Quad( "400", "^6", "0", "0" ), // push, v102, 0, 0   pc=9
new Quad( "400", "^2", "0", "0" ), // push, v95, 0, 0   pc=10
new Quad( "202", "20", "0", "0" ), // pcall, 20, 0, 0   pc=11
new Quad( "401", "^7", "0", "0" ), // pop, v103, 0, 0   pc=12
new Quad( "100", "^3", "^7", "0" ), // move, v96, v103, 0   pc=13
```

APPENDIX C

CLIENT-SERVER MODEL SOURCE CODE AND RESULTS

Source Code

```
coordinator Transport {
   Server server;
   Client client;
   init(string[] arg) {
         Request request;
         Reply   reply;
         int     value;
         int     FACT = 0;
         int     HALT = -99;

         server = findServer(arg[1]);
         client = findClient(arg[2]);

         request = client.request();
         while(request.request != HALT) {

            if (request.request == FACT)
                  value = server.fact(request.requestor,
```

```
                                        request.value);
            else
                value = server.fib(request.requestor,
                                        request.value);

            reply = new Reply(value);
            client.reply(reply);
            yield();

            request = client.request();
        }
    }
}

resource Server {
    string id;

    init(string[] arg) {
        id = arg[0];

        while(true) yield();
    }
```

```
private void badData(string who, int k) {
        write(id);
        write(": Bad Data: ");
        write(k);
        write(" from Client: ");
        write(who);
        write("\n");
}

private void heading(string who, int k) {
        write(id);
        write(" for ");
        write(who);
        write(" :");
        write(k);
}

private void end(int v) {
        write(" = ");
        write(v);
        write("\n");
```

```
}

portal fib(string who, int k) emit int {
        int oneBack = 1;
        int twoBack = 0;
        int current;
        int i;

        if (k <= 1) {
           badData(who, k);
           return k;
        }
        else {
           i = 2;
           heading(who, twoBack);
           while(i <= k) {
                 current = oneBack + twoBack;
                 write(", ");
                 write(oneBack);
                 twoBack = oneBack;
                 oneBack = current;
```

```
            i = i + 1;
        }
        write(", ");
        write(current);
        write("\n");
        return current;
    }
}

portal fact(string who, int k) emit int {
    int f;
    f = 1;

    if (k < 0) {
        badData(who, k);
        f = 0;
    }
    else if (k == 1) {
        heading(who, k);
        write("\n");
    }
    else {
```

```
            heading(who, k);
            f = k;
            while(k > 1) {
                k = k - 1;
                write(" * ");
                write(k);
                f = f * k;
            }
            end(f);
        }

        return f;
    }
}

resource Client {
    string id;
    int counter;
    int seed;

    init(string[] arg) {
        id = arg[0];
```

```
        if (arg.length() != 3) {
            write("Client: ");
            write(id);
            write(" did not get enough arguments!\n");
        }
        else {
            counter = s2i(arg[1]);
            seed    = s2i(arg[2]);
            while(true) yield();
        }
    }

    private void heading(int r, int v) {
        write(id);
        write(" : requesting service ");
        if (r == 0)
            write("fact on ");
        else
            write("fib on ");
        write(v);
        write("\n");
```

```
}

private void ending(int v) {
        write(id);
        write(" : result is ");
        write(v);
        write("\n");
}

portal request() emit Request {
        int     FACT = 0;
        int r;
        int v;
        Request req;

        if (counter > 0) {
           r = counter % 2;
           v = counter * seed;
           if (r == FACT)
                   v = v % 12;
           else
                   v = v % 47;
```

```
            heading(r, v);
            req = new Request(id, r, v);
            counter = counter - 1;
        }
        else req = new Request(id, -99, 0);

        return req;
    }

    portal reply(Reply reply) {
        ending(reply.value);
    }
}

class Request {
    public string requestor;
    public int    request;
    public int    value;

    public Request(string who, int what, int how) {
        requestor = who;
        request   = what;
```

```
        value     = how;
    }

    private Request clone() {
        Request r = new Request(requestor, request, value);
        return r;
    }
}

class Reply {
    public int    value;

    public Reply(int result) {
        value     = result;
    }

    private Reply clone() {
        Reply r = new Reply(value);
        return r;
    }
}
```

Deployment File

//Server:server1@olive.cs.ttu.edu

//Client:client1@ficus.cs.ttu.edu?5&3

//Transport:trans1@ficus.cs.ttu.edu?server1&client1

//Client:client2@hickory.cs.ttu.edu?9&2

//Transport:trans2@hickory.cs.ttu.edu?server1&client2

Compilation and Deployment

mCompile('cs.L', 'cs.dL').

Passed Syntax Analysis

Passed Semantic Analysis

Code written to LByteCode.java

Deploy as follows:

--> pitp://Server:server1@olive.cs.ttu.edu

--> pitp://Client:client1@ficus.cs.ttu.edu?5&3

--> pitp://Transport:trans1@ficus.cs.ttu.edu?server1&client1

--> pitp://Client:client2@hickory.cs.ttu.edu?9&2

--> pitp://Transport:trans2@hickory.cs.ttu.edu?server1&client2

Server olive.cs.ttu.edu is deployed to Class OliveCsTtuEdu.java :

Resource Server:server1

Server ficus.cs.ttu.edu is deployed to Class FicusCsTtuEdu.java :

Resource Client:client1

Coordinator Transport:trans1

Server hickory.cs.ttu.edu is deployed to Class HickoryCsTtuEdu.java :

Resource Client:client2

Coordinator Transport:trans2

Yes

Server Output

Colive$ java OliveCsTtuEdu

server1 for client2 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610

, 987, 1597, 2584

server1 for client2 :4 * 3 * 2 * 1 = 24

server1 for client1 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610

server1 for client2 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377

server1 for client2 :0 = 0

server1 for client2 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

server1 for client1 :0 = 0

server1 for client2 :8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 40320

server1 for client1 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34

server1 for client2 :0, 1, 1, 2, 3, 5, 8

server1 for client1 :6 * 5 * 4 * 3 * 2 * 1 = 720

server1 for client2 :4 * 3 * 2 * 1 = 24

server1 for client1 :0, 1, 1, 2

server1 for client2 :0, 1, 1

Client 1 Output

ficus$ java FicusCsTtuEdu

client1 : requesting service fib on 15

client1 : result is 610

client1 : requesting service fact on 0

client1 : result is 0

client1 : requesting service fib on 9

client1 : result is 34

client1 : requesting service fact on 6

client1 : result is 720

client1 : requesting service fib on 3

client1 : result is 2

Client 2 Output

```
^Chickory$ java HickoryCsTtuEdu
client2 : requesting service fib on 18
client2 : result is 2584
client2 : requesting service fact on 4
client2 : result is 24
client2 : requesting service fib on 14
client2 : result is 377
client2 : requesting service fact on 0
client2 : result is 0
client2 : requesting service fib on 10
client2 : result is 55
client2 : requesting service fact on 8
client2 : result is 40320
client2 : requesting service fib on 6
client2 : result is 8
client2 : requesting service fact on 4
client2 : result is 24
client2 : requesting service fib on 2
client2 : result is 1
```

APPENDIX D

MULTIPLE-TIER MODEL SOURCE CODE AND RESULTS

Source Code

```
coordinator Transport {
    Server server;
    Client client;
    CacheServer  cache;

    init(string[] arg) {
        Request request;
        Reply   reply;
        int     value;
        int     FACT = 0;
        int     HALT = -99;
        int     UNKNOWN = -999;

        server = findServer(arg[1]);
        client = findClient(arg[2]);
        cache  = findCacheServer(arg[3]);

        request = client.request();
```

```
while(request.request != HALT) {

   if (request.request == FACT) {
         value = cache.fact(request.requestor,
                              request.value);
         if (value == UNKNOWN) {
            value = server.fact(request.requestor,
                                    request.value);
            cache.fact(request.value, value);
         }
   }
   else {
         value = cache.fib(request.requestor,
                              request.value);
         if (value == UNKNOWN) {
            value = server.fib(request.requestor,
                                    request.value);
            cache.fib(request.value, value);
         }
   }

   reply = new Reply(value);
```

```
            client.reply(reply);
            yield();

            request = client.request();
        }
    }
}

resource Server {
    string id;

    init(string[] arg) {
        id = arg[0];

        while(true) yield();
    }

    private void badData(string who, int k) {
        write(id);
        write(": Bad Data: ");
        write(k);
        write(" from Client: ");
```

```
        write(who);
        write("\n");
}

private void heading(string who, int k) {
        write(id);
        write(" for ");
        write(who);
        write(" :");
        write(k);
}

private void end(int v) {
        write(" = ");
        write(v);
        write("\n");
}

portal fib(string who, int k) emit int {
        int oneBack = 1;
        int twoBack = 0;
        int current;
```

```
int i;


if (k <= 1) {
   badData(who, k);
   return k;
}
else {
   i = 2;
   heading(who, twoBack);
   while(i <= k) {
         current = oneBack + twoBack;
         write(", ");
         write(oneBack);
         twoBack = oneBack;
         oneBack = current;


         i = i + 1;
   }
   write(", ");
   write(current);
   write("\n");
   return current;
```

```
    }
}

portal fact(string who, int k) emit int {
    int f;
    f = 1;

    if (k < 0) {
       badData(who, k);
       f = 0;
    }
    else if (k == 1) {
       heading(who, k);
       write("\n");
    }
    else {
       heading(who, k);
       f = k;
       while(k > 1) {
            k = k - 1;
            write(" * ");
            write(k);
```

```
                f = f * k;
            }
            end(f);
        }

        return f;
    }
}

class CacheNode resource specific {
    public int key;
    public int value;
    public CacheNode left;
    public CacheNode right;

    public CacheNode(int k, int v) {
        key = k;
        value = v;
        left = null;
        right = null;
    }
}
```

```
class CacheTree resource specific {
   public CacheNode root;

   public CacheTree() {
         root = null;
   }

   public void insert(int key, int value) {
         CacheNode next;
         boolean searching = true;

         if (root == null) {
            root = new CacheNode(key, value);
         }
         else {
            next = root;
            while (searching) {
                  if (key == next.key)
                     searching = false;
                  else if (key < next.key)
                     if (next.left == null) {
```

```
                    searching = false;
                    next.left = new CacheNode(key, value);
                }
                else
                    next = next.left;
            else
                if (next.right == null) {
                    searching = false;
                    next.right = new CacheNode(key, value);
                }
                else
                    next = next.right;
        }
    }
}

public CacheNode find(int key) {
    boolean searching = true;
    CacheNode next = root;

    while (searching) {
        if (next == null)
```

```
                return null;
            else if (key == next.key)
                return next;
            else if (key < next.key)
                if (next.left == null)
                    return null;
                else
                    next = next.left;
            else
                if (next.right == null)
                    return null;
                else
                    next = next.right;
        }
    }

    public void dump(string heading) {
        write(heading);
        write("\n");

        dfs(root);
```

```
            write(heading);
            write("--");
            write("done\n");
        }

        private void dfs(CacheNode p) {
            if (p == null) return;
            dfs(p.left);
            write("  ");
            write(p.key);
            write("-->");
            write(p.value);
            write("\n");
            dfs(p.right);
        }
    }

    resource CacheServer {
        string id;
        CacheTree factTree;
        CacheTree fibTree;
```

```
init(string[] arg) {
        id = arg[0];

        factTree = new CacheTree();
        fibTree  = new CacheTree();

        while(true) yield();
}

private void heading(string type, string who, int k) {
        write("Cache: ");
        write(type);
        write(" request of ");
        write(k);
        write(" from ");
        write(who);
        write("\n");
}

portal fib(string who, int k) emit int {
        CacheNode n;
```

```
        n = fibTree.find(k);
        if (n != null) {
            heading("Known fib", who, k);
            return n.value;
        }
        else {
            heading("Unknown fib", who, k);
            return -999;
        }
}

portal fact(string who, int k) emit int {
        CacheNode n;

        n = factTree.find(k);
        if (n != null) {
            heading("Known fact", who, k);
            return n.value;
        }
        else {
            heading("Unknown fact", who, k);
```

```
            return -999;
        }
    }

    portal synchronized fib(int k, int v) {
        fibTree.insert(k, v);
    }

    portal synchronized fact(int k, int v) {
        factTree.insert(k, v);
    }
}

resource Client {
    string id;
    int counter;
    int seed;

    init(string[] arg) {
        id = arg[0];

        if (arg.length() != 3) {
```

```
            write("Client: ");
            write(id);
            write(" did not get enough arguments!\n");
        }
        else {
            counter = s2i(arg[1]);
            seed    = s2i(arg[2]);
            while(true) yield();
        }
    }

    private void heading(int r, int v) {
        write(id);
        write(" : requesting service ");
        if (r == 0)
            write("fact on ");
        else
            write("fib on ");
        write(v);
        write("\n");
    }
```

```
private void ending(int v) {
        write(id);
        write(" : result is ");
        write(v);
        write("\n");
}

portal request() emit Request {
        int     FACT = 0;
        int r;
        int v;
        Request req;

        if (counter > 0) {
           r = counter % 2;
           v = counter * seed;
           if (r == FACT)
                 v = v % 12;
           else
                 v = v % 47;

           heading(r, v);
```

```
            req = new Request(id, r, v);
            counter = counter - 1;
        }
        else req = new Request(id, -99, 0);

        return req;
    }

    portal reply(Reply reply) {
        ending(reply.value);
    }
}


class Request {
    public string requestor;
    public int    request;
    public int    value;

    public Request(string who, int what, int how) {
        requestor = who;
        request   = what;
```

```
        value     = how;
    }

    private Request clone() {
        Request r = new Request(requestor, request, value);
        return r;
    }
}

class Reply {
    public int    value;

    public Reply(int result) {
        value     = result;
    }

    private Reply clone() {
        Reply r = new Reply(value);
        return r;
    }
}
```

Deployment File

//Server:server1@olive.cs.ttu.edu

//Client:client1@hickory.cs.ttu.edu?17&3

//Transport:trans1@hickory.cs.ttu.edu?server1&client1&cache

//CacheServer:cache@ficus.cs.ttu.edu

Compilation and Deployment

mCompile('mt.L', 'mt.dL').

Passed Syntax Analysis

Passed Semantic Analysis

Code written to LByteCode.java

Deploy as follows:

--> pitp://Server:server1@olive.cs.ttu.edu

--> pitp://Client:client1@hickory.cs.ttu.edu?17&3

--> pitp://Transport:trans1@hickory.cs.ttu.edu?server1&client1&cache

--> pitp://CacheServer:cache@ficus.cs.ttu.edu

Server olive.cs.ttu.edu is deployed to Class OliveCsTtuEdu.java :

Resource Server:server1

Server hickory.cs.ttu.edu is deployed to Class HickoryCsTtuEdu.java :

Resource Client:client1

Coordinator Transport:trans1

Server ficus.cs.ttu.edu is deployed to Class FicusCsTtuEdu.java :

Resource CacheServer:cache

Yes

Server Output

olive$ java OliveCsTtuEdu

server1 for client1 :0, 1, 1, 2, 3

server1 for client1 :0 = 0

server1 for client1 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610

, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418

, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352,

24157817, 39088169, 63245986, 102334155, 165580141, 267914296, 433494437,

701408733, 1134903170

server1 for client1 :6 * 5 * 4 * 3 * 2 * 1 = 720

server1 for client1 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610

, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418

, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352,

24157817, 39088169, 63245986

server1 for client1 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610

, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418

, 317811, 514229, 832040, 1346269, 2178309, 3524578

server1 for client1 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610

, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418

server1 for client1 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610

, 987, 1597, 2584, 4181, 6765, 10946

server1 for client1 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610

server1 for client1 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34

server1 for client1 :0, 1, 1, 2

Cache Server Output

ficus$ java FicusCsTtuEdu

Cache: Unknown fib request of 4 from client1

Cache: Unknown fact request of 0 from client1

Cache: Unknown fib request of 45 from client1

Cache: Unknown fact request of 6 from client1

Cache: Unknown fib request of 39 from client1

Cache: Known fact request of 0 from client1

Cache: Unknown fib request of 33 from client1

Cache: Known fact request of 6 from client1

Cache: Unknown fib request of 27 from client1

Cache: Known fact request of 0 from client1

Cache: Unknown fib request of 21 from client1

Cache: Known fact request of 6 from client1

Cache: Unknown fib request of 15 from client1

Cache: Known fact request of 0 from client1

Cache: Unknown fib request of 9 from client1

Cache: Known fact request of 6 from client1

Cache: Unknown fib request of 3 from client1

Client Output

hickory$ java HickoryCsTtuEdu

client1 : requesting service fib on 4

client1 : result is 3

client1 : requesting service fact on 0

client1 : result is 0

client1 : requesting service fib on 45

client1 : result is 1134903170

client1 : requesting service fact on 6

client1 : result is 720

client1 : requesting service fib on 39

client1 : result is 63245986

client1 : requesting service fact on 0

client1 : result is 0

client1 : requesting service fib on 33

client1 : result is 3524578

client1 : requesting service fact on 6

client1 : result is 720

client1 : requesting service fib on 27

client1 : result is 196418

client1 : requesting service fact on 0

client1 : result is 0

client1 : requesting service fib on 21

client1 : result is 10946

client1 : requesting service fact on 6

client1 : result is 720

client1 : requesting service fib on 15

client1 : result is 610

client1 : requesting service fact on 0

client1 : result is 0

client1 : requesting service fib on 9

client1 : result is 34

client1 : requesting service fact on 6

client1 : result is 720

client1 : requesting service fib on 3

client1 : result is 2

APPENDIX E

PEER-TO-PEER MODEL SOURCE CODE AND RESULTS

Source Code

```
coordinator Transport {
   Peer client;
   Peer server;

   init(string[] arg) {
        Request request;
        Reply   reply;
        int     value;
        int     FACT = 0;
        int     HALT = -99;

        client = findPeer(arg[1]);

        request = client.request();
        while(request.request != HALT) {
           server = findPeer(request.server);

           if (request.request == FACT)
```

```
                value = server.fact(request.requestor,
                                    request.value);
            else
                value = server.fib(request.requestor,
                                   request.value);

            reply = new Reply(value);
            client.reply(reply);
            yield();

            request = client.request();
        }
    }
}

resource Peer {
    string id;
    int counter;
    int seed;
    int peerCnt;
    string[] peer;
```

```
init(string[] arg) {
    int i;
        id = arg[0];


        if (arg.length() < 4) {
            write("Peer: ");
            write(id);
            write(" did not get enough arguments!\n");
        }
        else {
            peer = new string[arg.length() - 3];
            counter = s2i(arg[1]);
            seed    = s2i(arg[2]);
            i = 3;
            peerCnt = 0;
            while (i < arg.length()) {
                    peer[peerCnt] = arg[i];
                    peerCnt = peerCnt + 1;
                    i = i + 1;
            }
            while(true) yield();
        }
```

```
}


private void badData(string who, int k) {
        write(id);
        write(": Bad Data: ");
        write(k);
        write(" from Client: ");
        write(who);
        write("\n");
}

private void replyHeading(string who, int k) {
        write(id);
        write(" for ");
        write(who);
        write(" :");
        write(k);
}


private void end(int v) {
```

```
        write(" = ");
        write(v);
        write("\n");
}


portal fib(string who, int k) emit int {
        int oneBack = 1;
        int twoBack = 0;
        int current;
        int i;

        if (k <= 1) {
           badData(who, k);
           return k;
        }
        else {
           i = 2;
           replyHeading(who, twoBack);
           while(i <= k) {
                current = oneBack + twoBack;
                write(", ");
```

```
            write(oneBack);
            twoBack = oneBack;
            oneBack = current;

            i = i + 1;
        }
        write(", ");
        write(current);
        write("\n");
        return current;
    }
}

portal fact(string who, int k) emit int {
    int f;
    f = 1;

    if (k < 0) {
        badData(who, k);
        f = 0;
    }
    else if (k == 1) {
```

```
            replyHeading(who, k);
            write("\n");
        }
        else {
            replyHeading(who, k);
            f = k;
            while(k > 1) {
                k = k - 1;
                write(" * ");
                write(k);
                f = f * k;
            }
            end(f);
        }

        return f;
}

private void requestHeading(int r, int v, string whom) {
        write(id);
        write(" : requesting service ");
        if (r == 0)
```

```
            write("fact on ");
        else
            write("fib on ");
        write(v);
        write(" from ");
        write(whom);
        write("\n");
}

private void ending(int v) {
        write(id);
        write(" : result is ");
        write(v);
        write("\n");
}

portal request() emit Request {
        int     FACT = 0;
        int r;
        int v;
        Request req;
        string  p = null;
```

```
        if (counter > 0) {
            r = counter % 2;
            v = counter * seed;
        if (r == FACT)
                v = v % 12;
            else
                v = v % 47;

            requestHeading(r, v, peer[counter % peerCnt]);
            req = new Request(id, r,
                              v,
                              peer[counter % peerCnt]);
            counter = counter - 1;
        }
        else req = new Request(id, -99, 0, p);

        return req;
}

portal reply(Reply reply) {
        ending(reply.value);
```

```
    }
}


class Request {
    public string requestor;
    public int    request;
    public int    value;
    public string server;

    public Request(string who, int what, int with, string whom) {
        requestor = who;
        request   = what;
        value     = with;
        server    = whom;
    }

    private Request clone() {
        Request r = new Request(requestor, request, value, server);
        return r;
    }
}
```

```
class Reply {
   public int    value;

   public Reply(int result) {
         value     = result;
   }

   private Reply clone() {
         Reply r = new Reply(value);
         return r;
   }
}
```

Deployment File

```
//Peer:peer1@olive.cs.ttu.edu?2&2&peer1&peer2&peer3
//Peer:peer2@ficus.cs.ttu.edu?5&3&peer1&peer2&peer3
//Peer:peer3@hickory.cs.ttu.edu?7&4&peer1&peer2&peer3
//Transport:trans1@olive.cs.ttu.edu?peer1
//Transport:trans2@olive.cs.ttu.edu?peer2
//Transport:trans3@olive.cs.ttu.edu?peer3
```

Compilation and Deployment

?- mCompile('peer.L', 'peer.dL').

Passed Syntax Analysis

Passed Semantic Analysis

Code written to LByteCode.java

Deploy as follows:

--> pitp://Peer:peer1@olive.cs.ttu.edu?2&2&peer1&peer2&peer3

--> pitp://Peer:peer2@ficus.cs.ttu.edu?5&3&peer1&peer2&peer3

--> pitp://Peer:peer3@hickory.cs.ttu.edu?7&4&peer1&peer2&peer3

--> pitp://Transport:trans1@olive.cs.ttu.edu?peer1

--> pitp://Transport:trans2@olive.cs.ttu.edu?peer2

--> pitp://Transport:trans3@olive.cs.ttu.edu?peer3

Server olive.cs.ttu.edu is deployed to Class OliveCsTtuEdu.java :

Resource Peer:peer1

Coordinator Transport:trans1

Coordinator Transport:trans2

Coordinator Transport:trans3

Server ficus.cs.ttu.edu is deployed to Class FicusCsTtuEdu.java :

Resource Peer:peer2

Server hickory.cs.ttu.edu is deployed to Class HickoryCsTtuEdu.java :

Resource Peer:peer3

Yes

Peer 1 Output

olive$ java OliveCsTtuEdu

peer1 : requesting service fact on 4 from peer3

peer1 : result is 24

peer1 : requesting service fib on 2 from peer2

peer1 for peer3 :0 = 0

peer1 for peer2 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34

peer1 : result is 1

peer1 for peer3 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

Peer 2 Output

ficus$ java FicusCsTtuEdu

peer2 for peer3 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 98

7, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 31

7811

peer2 : requesting service fib on 15 from peer3

peer2 : result is 610

peer2 : requesting service fact on 0 from peer2

peer2 for peer2 :0 = 0

peer2 : result is 0

peer2 : requesting service fib on 9 from peer1

peer2 for peer1 :0, 1, 1

peer2 : result is 34

peer2 for peer3 :4 * 3 * 2 * 1 = 24

peer2 : requesting service fact on 6 from peer3

peer2 : result is 720

peer2 : requesting service fib on 3 from peer2

peer2 for peer2 :0, 1, 1, 2

peer2 : result is 2

peer2 for peer3 :0, 1, 1, 2, 3

Peer 3 Output

hickory$ java HickoryCsTtuEdu

peer3 for peer1 :4 * 3 * 2 * 1 = 24

peer3 : requesting service fib on 28 from peer2

peer3 for peer2 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610

peer3 : result is 317811

peer3 : requesting service fact on 0 from peer1

peer3 : result is 0

peer3 : requesting service fib on 20 from peer3

peer3 for peer3 :0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 98

7, 1597, 2584, 4181, 6765

peer3 : result is 6765

peer3 : requesting service fact on 4 from peer2

peer3 : result is 24

peer3 : requesting service fib on 12 from peer1

peer3 for peer2 :6 * 5 * 4 * 3 * 2 * 1 = 720

peer3 : result is 144

peer3 : requesting service fact on 8 from peer3

peer3 for peer3 :8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 40320

peer3 : result is 40320

peer3 : requesting service fib on 4 from peer2

peer3 : result is 3

APPENDIX F

COMPILER TECHNOLOGY

A compiler must analyze a source program to ensure that the program is syntactically correct and then generate some target language that when executed will map to the intended semantics of the original source program (Fischer and LeBlanc, 1988). A compiler is composed of the following phases: lexical analysis, syntax analysis, semantic processing, intermediate code generation, code optimization and code generation. We will look at each of these phases and the symbol table in more detail.

Symbol Table

A symbol table is a data structure used to associate an identifier with various attributes needed during compilation. Some basic information that is associated with an identifier is the name and the type of an identifier. In a language with only global variables and declarations, the name of the identifier is sufficient to uniquely identify a variable or declaration. In block-structured languages, the scope in which the name appears is needed to uniquely identify variables and declarations.

When a block-structured language is to be compiled, the symbol table must be sufficiently complex to be able to deal with nested scope. A simple approach is to maintain a separate symbol table for each scope (Aho, Sethi, and Ullman, 1986). The compiler must determine when the compilation process enters and exits a scope in order to maintain the symbol table correctly.

The symbol table is an essential element of a compiler and the compilation process. Each phase of the compilation process will access the symbol table, either adding information to the symbol table or retrieving information from the symbol table. Because of the compiler's reliance upon the symbol table, it is important that it be well designed and implemented. Fischer and LeBlanc (1988) give the following possible implementations for a simple table:

- Unordered List is the simplest implementation, yet it becomes impractical as the size of the symbol table grows due to the long search times, O(N), to find an entry.
- Ordered List is the next possible implementation, yet in overall search performance, an ordered list is not significantly faster to search, O(N/2), than an unordered list, O(N).
- Binary Search Trees offer a significant improvement in searching performance, O(log(N)), but only if the tree is balanced. If the tree is not balanced, the performance of the binary search tree can degrade to the point when it is the same as an ordered list. This degradation in performance is based upon the order in which data is inserted into the binary search tree. Height balancing trees, such as AVL Trees and Red Black Trees, are more complex to implement and maintain, yet their performance will not degrade as do standard binary search trees.
- Hash Tables are listed as the most common means of implementing symbol tables. The Aho et al. (1986) book offers an entire section on using hash

tables for symbol tables. When hash tables are used, care must be given to the selection of the hash function and the size of the table in order to minimize the number of collisions. While hash tables offer exceptional search performance, O(1), their performance can degrade significantly if excessive collisions occur.

Lexical Analysis

Lexical analysis, also known as a scanner, reads a source program and converts the characters in the source program into tokens. Tokens represent such programming constructs as identifiers, integers, reserved words, delimiters, etc. The tokens produced by the scanner will be passed to syntax analysis for additional processing. To convert the source program into tokens, the program is read one character at a time. The scanner matches the input characters to known patterns to produce tokens.

Regular Expression

Regular expressions are a means of specifying the patterns of various tokens. While a more complex notation is required to express most languages, regular expressions are generally sufficient to express tokens. The following definition for a regular expression is taken from Aho et al. (1986) and Fischer and LeBlanc (1988). An alphabet V is a finite set of symbols. A string over some alphabet V is a finite sequence of symbols taken from V. The length of a string s, written |s|, is a count of the number of symbols in s. A special string of length zero is represented by the symbol $\in$. New

strings are built from old strings by concatenation. Let snow and board be strings formed from symbols in V; snowboard is a string formed by concatenating string snow and board. A language is any set of strings over some fixed alphabet V.

Six meta-characters (Table E.1) are used to define operations that can be applied to languages.

Table E.1. Definition of Meta-Characters.

META-CHARACTER	*DEFINITION OF META-CHARACTER*
(	Open parenthesis used to start a group.
)	Close parenthesis used to end a group.
‘	Single-quote used to quote meta-characters so it can be treated as a symbol from the alphabet. Example: ‘*’ means the symbol *, not the operator *.
*	Kleene closure used to denote zero or more occurrences of a regular expression. Example: if k is a regular expression, then k* means zero or more occurrences of k.
+	Positive closure used to denote one or more occurrences of a regular expression. Example: if k is a regular expression, then k+ means one or more occurrences of k.
|	Or used to denote disjunction of regular expressions. Example: if k and q are regular expressions, then (k | q), means k or q.

To specify a specific number of occurrences of a regular expression, we can allow a positive integer to be used in place of the Kleene, or positive closure meta-character, as in k^3, where k is a regular expression and the three denotes that k will occur three times. Now that we have a notation for specifying the patterns of various tokens, a mechanism is needed to recognize the tokens specified by a regular expression.

Deterministic Finite Automation

By encoding regular expressions into a deterministic finite automation (DFA), we can recognize tokens read from the source file. Lewis et al. (1981) define a DFA as a quintuple $M = (K, \Sigma, \delta, s, F)$ where:

K is a finite set of states,

Σ is an alphabet,

$s \in K$ is the initial state,

$F \subseteq K$ is the set of final states, and

δ is a function from $K \times \Sigma \rightarrow K$, known as a transition function.

A DFA can be represented graphically (Figure F.1) using a transition diagram where:

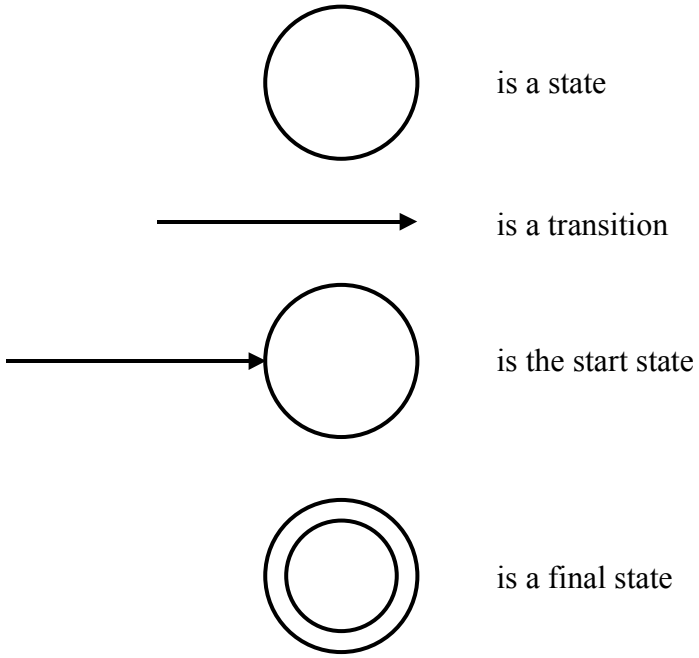


Figure F.1. Transition Diagram Symbols.

A DFA that can recognize the keywords “if,” “while,” and “return” can be encoded as:

$K = \{ s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13} \}$,

$\Sigma = \{e, f, h, i, l, n, r, t, u, w\}$,

$s = s_0$,

$F = \{ s_2, s_7, s_{13} \}$

$\delta =$

α	β	$\delta(\alpha, \beta)$
s_0	i	s_1
s_0	w	s_3
s_0	r	s_8
s_1	f	s_2
s_3	h	s_4
s_4	i	s_5
s_5	l	s_6
s_6	e	s_7
s_8	e	s_9
s_9	t	s_{10}
s_{10}	u	s_{11}
s_{11}	r	s_{12}
s_{12}	n	s_{13}

.

The same DFA can be depicted graphically by encoding the transition function $\delta(\alpha, \beta)$ into the diagram (Figure F.2). To depict state s_0, it will need three transitions coming out of it: one for ‘I,’ one for ‘w,’ and one for ‘r.’

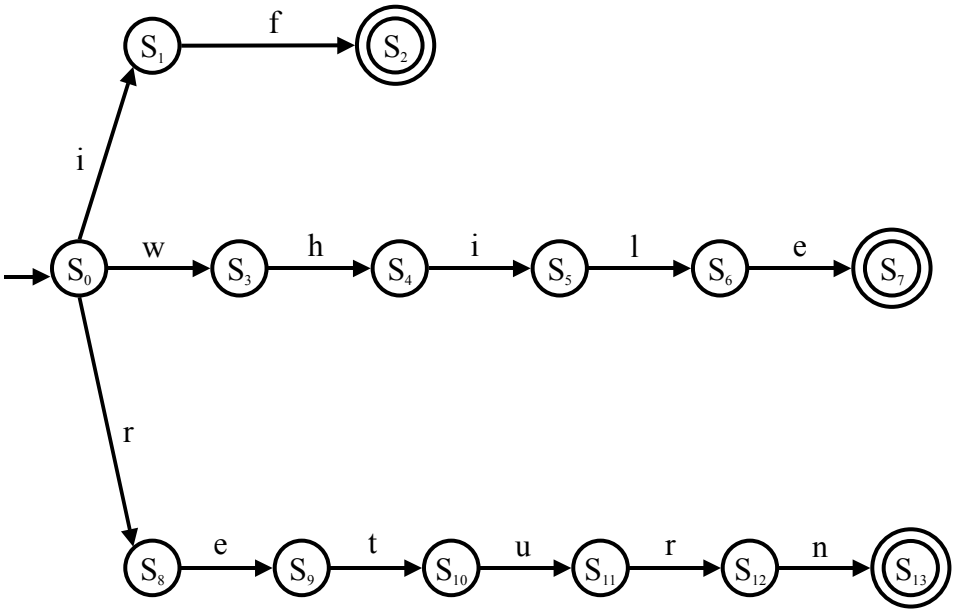


Figure F.2. DFA as a Transition Diagram.

The process of converting a DFA into code is somewhat mechanical and, as such, tools like Lex have been created to generate lexical analyzers. Hand coding a lexical analyzer, while tedious, is not overly difficult. The states of the DFA can be encoded as cases of a switch statement. A variable, currentState, contains the value of the current state in the DFA. Each case statement can read a character from the source file and, depending upon the value, the currentState variable will be set to transition to the next state. Special states, such as the default, can be coded to handle error conditions. When a final state is reached, some token has been recognized from the source file.

Syntax Analysis

Syntax analysis, also known as a parser, takes a sequence of tokens and verifies that the sequence of tokens is a legal part of the programming language. To perform this verification, a formal notation for expressing a programming language is needed. While regular expressions are sufficient for expressing the patterns of a token, a more expressive notation is required for programming languages that can have recursive constructs. Context-free grammars (CFG) have the expressive power that is needed for most programming languages, and they have the advantage that efficient parsers can be constructed for them to determine if a source program is syntactically well formed (Aho et al., 1986). Lewis and Papadimitriou (1981) define a context-free grammar quadruple $G=(V, \Sigma, R, S)$ where:

V is an alphabet,

Σ (the set of terminals) $\subset V$,

R (the set of rules) is a finite subset of $(V - \Sigma) \times V^*$, and

S (the start symbol) $\in V - \Sigma$.

The members of $V - \Sigma$ are called nonterminals. For any nonterminal λ and $\mu \in V^*$, we write $\lambda \rightarrow \mu$ wherever $(\lambda, \mu) \in R$. The following CFG can recognize simple arithmetic expressions:

V = { x, y, +, -, *, /, OP, EXPR },

Σ = { x, y, +, -, *, / },

R = {

EXPR $\rightarrow$ EXPR OP EXPR,

EXPR → (EXPR),

EXPR → - EXPR,

EXPR → x,

EXPR → y,

OP → +,

OP → -,

OP → *,

OP → /

},

S = EXPR.

To determine if a string (sequence of tokens) can be produced by a grammar, we must see if there is a derivation that can produce the string (Figure F.3). Starting with the string "(x + y) / x," we will try to derive this string from the grammar. The symbol ⇒ will be used to indicate derives. We must begin the derivation with the nonterminal start symbol EXPR.

EXPR ⇒ EXPR OP EXPR ⇒ (EXPR) OP EXPR ⇒ (EXPR OP EXPR) OP EXPR

⇒ (x OP EXPR) OP EXPR ⇒ (x + EXPR) OP EXPR ⇒ (x + y) OP EXPR

⇒ (x + y) / EXPR ⇒ (x + y) / x

Figure F.3. Derivation of String (x + y) / x.

LL(1) Grammar

Context-free grammars are a useful class of languages, yet these languages can possess properties that can make building an efficient parser very difficult. To simplify the building of a parser, we would like to restrict the programming language to a grammar that is LL(1). A grammar that is LL(1) has no left-recursion and is deterministic with one token look ahead. The first L in LL(1) stands for scanning a string from left to right, and the second L for producing a leftmost derivation. The 1 in LL(1) is for one token look ahead.

Eliminating Left-Recursion

A grammar G is left-recursive if it contains rules of the form $\lambda' \rightarrow$. To remove left-recursion from G, we must find all rules of the form $\lambda \rightarrow \lambda\mu$ and all rules of the form $\lambda \rightarrow \gamma\mu$, where $\lambda \neq \gamma$. We then replace the rules of the form $\lambda \rightarrow \lambda\mu$ with new rules of the form $\lambda \rightarrow \gamma\lambda'$, $\lambda' \rightarrow \mu\lambda'$ and $\lambda' \rightarrow \in$.

Left-Factoring

A grammar G is non-deterministic if it contains rules of the form $\lambda \rightarrow \mu\alpha$ and $\lambda \rightarrow \mu\beta$. Rule λ can produce two derivations, both starting with symbol μ. Because of this, it is not possible to determine which rule to use with one symbol look ahead. The ambiguity can be removed by replacing these rules with new rules of the form $\lambda \rightarrow \mu\lambda'$, $\lambda' \rightarrow \alpha$, and $\lambda' \rightarrow \beta$. Once this process is complete, the grammar will be deterministic with one symbol look ahead.

Recursive-Descent Parser

Once we have a grammar that is LL(1), a recursive-descent parser can be constructed to accept the language. To construct the parser, a procedure is written for each nonterminal symbol from the grammar. By using a one token look ahead, the parser can unambiguously determine which procedure to execute next. The resulting parser is now called a predictive parser. The simple grammar for arithmetic expressions is:

R = {

EXPR → EXPR OP EXPR,

EXPR → (EXPR),

EXPR → - EXPR,

EXPR → x,

EXPR → y,

OP → +,

OP → -,

OP → *,

OP → /

},

S = EXPR.

To make this grammar LL(1), we must remove the left-recursion from EXPR. The resulting grammar is:

R = {

EXPR → (EXPR) OP EXPR,

EXPR → - EXPR OP EXPR,

EXPR → x OP EXPR,

EXPR → y OP EXPR,

EXPR → (EXPR),

EXPR → - EXPR,

EXPR → x,

EXPR → y,

OP → +,

OP → -,

OP → *,

OP → /

},

S = EXPR.

A predictive recursive-descent parser can now be written which will have two procedures: expr() and op(). By looking a single token ahead, the parser can unambiguously predict what action to take next. The syntactic correctness of a program can be checked by parsing it using grammar rules. Checking the syntax of a program is akin to saying that the program looks correct. When a string is parsed, it can be organized into a parse-tree. The organization that a parse-tree provides allows various programming constructs to be picked out of the program. High-level constructs that need to be identified are things like classes in an object-oriented language, function

declarations, and individual statements. Lower-level constructs that need to be identified are things like identifiers, operators, and constants. If we replace variables x and y in the grammar example with the more general notion that they are identifiers 'id,' we can generate a parse-tree for the string '(x+y)/x' as follows (Figure F.4):

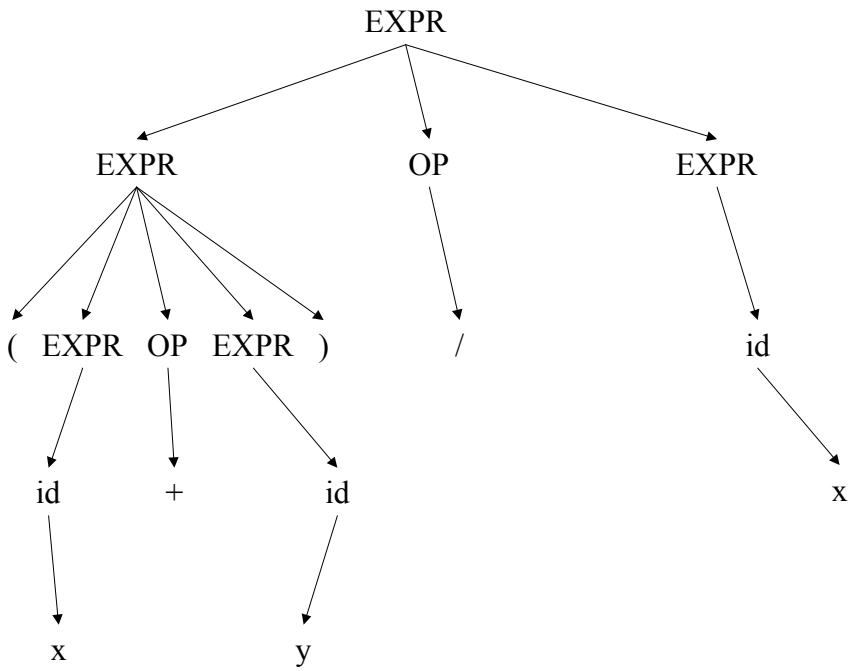


Figure F.4. Parse Tree of String (x + y) / x.

The parse-tree has given an organizational hierarchy to the arithmetic expression. In the context of a program, the simple arithmetic expression would be a subtree of a statement construct parse-tree. The statement parse-tree would in turn be a subtree of some function construct parse-tree. The function parse-tree would in turn be a subtree of a program construct parse-tree. In this way, the entire source of the program would be organized into a hierarchy. This hierarchy can then be searched and researched to extract

important information about the program without having to repeat the scanning and parsing.

Semantic Processing

Semantic processing follows parsing and performs a more in-depth verification of the program. Semantic processing uses attribute information extracted during parsing to perform various type checking. The type checking performed during semantic processing is referred to as static type checking to distinguish it from dynamic type checking which is performed during run-time. A type checker verifies that the type of a construct matches that expected by its context (Aho et al., 1986). Aho et al. (1986) identify four types of checks that can be performed:

- Type Checking. Ensures that an operator will only be applied to operands of a specific type or that function arguments are of a specific type.
- Flow-of-Control Checking. Ensures that as statements are executed in a program, there is always a next statement to be executed. This is particularly true when the flow of control is transferred via break or switch statements.
- Uniqueness Checking. Under specific conditions, it ensures that a symbol is unique. Global declarations must be unique, such as class names and function names. The labels in a switch-statement must be unique.
- Name-Related Checking. In some languages, a label must appear in more than one place, such as at the beginning and end of a block.

Multiple Pass Compilation

In many modern programming languages, it is often possible to use a programming construct before it has been defined. As an example, in Java a function can be called before it is declared. Under such circumstances it is not possible for a compiler to verify the type of a construct that has not yet been declared. As a result, the compiler will have to perform multiple passes through the source code to verify all constructs. In the first pass, identifiers will be entered into the symbol table. As the attributes of the identifier are discovered, they will be entered into the symbol table and associated with the identifier. In the second pass, the type of a variable, return type of a function, or the number and type of arguments passed to a function can be extracted from the symbol table.

Semantic Routines

Semantic routines can be embedded into the parser to perform semantic processing. In addition to performing static checks of the source code, the semantic routines may translate the source code into an intermediate representation using information in the symbol table and the parse-tree. Under specific conditions, a semantic routine may be able to correct a problem when it is encountered. As an example, a floating-point number and an integer cannot be directly added together on most machines. Yet, a semantic routine that is working on the translation of arithmetic expressions can detect such a situation and generate additional code to cast the integer to a floating-point number.

Intermediate Code Generation

Creating an intermediate representation for the code has a number of advantages over generating the target machine code directly. Using an intermediate language helps to separate high-level operations from their possibly low-level, machine-dependent realizations (Fischer and LeBlanc, 1988). While intermediate code can take various forms, the two most common forms are triples and quadruples (a.k.a., quads). Both triples and quads are referred to as three-address codes and generalize assembly code for some virtual three-address machines (Fischer and LeBlanc, 1988).

Triples

Triples are of the form:

TRIPLE → OPERATOR, OPERAND, OPERAND

OPERATOR → mul | div | add | sub | read | write | assign | lt | gt | eq

OPERAND → id | (num)

The OPERATOR field can be any instruction that the virtual machine is to support. Multiply, divide, add, subtract, read, and write are all possible instructions. The OPERAND field is either an identifier or an index to the location of another triple instruction. The index indicates that the instruction should obtain its operand from the output of the referred triple (Table E.2).

Table E.2. Triples for Expression (x + y) / x.

Index	*Triple*
1	add, x, y
2	div, (1), x

Quadruples

Quads are of the form:

QUAD → OPERATOR, OPERAND, OPERAND, OPERAND

OPERATOR → mul | div | add | sub | read | write | assign | lt | gt | eq

OPERAND → id

The OPERATOR field can be any instruction that the virtual machine is to support. Multiply, divide, add, subtract, read, and write are all possible instructions. The OPERAND field is always an identifier, with the third operand field being a result field. Because quads do not directly refer to other quads, temporary variables must be created to hold intermediate results (Table E.3).

Table E.3. Quads for Expression (x + y) / x.

Index	*Quad*
1	add, x, y, r1
2	div, r1, x, r2

Triples are a more concise notational form and do not require the creation of a temporary variable to hold intermediate results as do quads. However, working with triples can be more difficult due to their positional dependency. If a triple is moved or

deleted, all other triples need to be examined to determine if their operands have been changed. During the optimization of a program, instructions are often rearranged or even deleted to make the program more compact and/or make the program execute more efficiently.

Code Optimization

A process whereby the compiler attempts to make the size of the target code or execution time of the program more efficient is known as code optimization. Code optimization can be applied at different phases of the compilation process, but, in general, it is applied after intermediate code is generated and then applied again to the actual target code. The rationale for applying optimizations twice are that while some optimizations can be readily applied to a high-level intermediate code representation, other optimizations can only be applied to low-level machine code when the individual specifications of the hardware are known. While there are various types of optimizations, which we will review shortly, most of these optimizations can be applied either locally or globally to the entire program.

Peephole Optimization

Peephole optimization is a local optimization technique that only looks at a few instructions (either intermediate or target) at a time to find optimization opportunities. Peephole optimization is limited to looking at instructions that occur in a single basic block. A basic block is defined as a sequence of code with no branch-statements.

Applying peephole optimization is relatively simple because of the limited scope and number of instructions that are involved in the optimization process. Because only one basic block and a few instructions can be optimized at a time, peephole optimization has limited ability. While peephole optimization has limited ability, it is, nevertheless, an effective technique for optimizing code. Global optimization techniques are those that can span more than a single basic block. In contrast to local optimization, these techniques are often complex and time consuming to perform. Some of the optimization techniques that can be applied to a program either locally are globally are:

- removing redundant instructions,
- optimizing the calculation of an address,
- making efficient use of registers,
- reusing temporary variables,
- eliminating unreachable code,
- migrating code outside a loop, and
- replacing complex instructions with simpler instructions.

Code Generation

Code generation is the final phase of a compiler and results in the creation of some type of target code. Some of the techniques that will be discussed during code generation can be applied to the generation of intermediate code or the generation of target code, yet all these techniques will be grouped together under the code generation

phase. Code generation can be loosely grouped into three different categories of statements.

Procedure and Function Call Statements

When translating a procedure call, the compiler must generate code to handle the following:

1. Arguments passed to the procedure must be placed in a known location so they can be accessed by the called procedure.
2. The state of the calling procedure must be saved off such that it can be re-established once the called procedure completes.
3. In the case of a function call, the location of any returned data must be placed in a known location so it can be accessed by the calling procedure.
4. There must be a jump statement to the start of the called procedure.
5. At the end of the called procedure, the state of the calling procedure must be re-established, and the flow of control must return to the instruction following the jump to the called procedure.

Control-Flow Statements

Statements such as if-statements and loops can alter the flow of control of an application. When translating such statements into instructions, it is not immediately possible to generate all the instructions of a statement. Consider a grammar rule of the form:

'IF_STMT → if BOOL_EXP then STMT_1 else STMT_2.'

When the if-statement is encountered, the starting address of STMT_2 cannot be known until all the statements in STMT_2 have been parsed. The compiler must generate a branch-statement to the start of the STMT_2 instructions if the Boolean expression of the if-statement is false. In addition, the compiler must insert an unconditional jump instruction at the end of STMT_1 to jump over all of the instructions in STMT_2. In the case of a while loop with a grammar rule of the form:

'WHILE_STMT → while BOOL_EXP do STMT,'

the while-statement must conditionally jump over all the instructions in STMT when the Boolean expression is false. At the end of the instructions in STMT, a jump to the test of the Boolean expression must be executed. When generating code for statements that can change the flow of control, the compiler can first generate labels vs. actual addresses for jump and branch statements. Once all the instructions have been generated, the labels can be converted into actual addresses.

Simple Basic Block Statements

The simplest statements to generate code for are those in which the flow of control is always sequential. For such statements, there is a predefined sequence of instructions that can be used to replace the source code statement. An example of such a translation has already been shown for the statement '(x + y) / x' as part of intermediate code generation.

APPENDIX G

VIRTUAL MACHINES

Virtual Machines

"A virtual machine is the construct of a program (such as CP/370) that behaves so much like a real machine that an OS, or other program written to run alone on a real machine, is fooled into thinking that it is running on a real bare machine by itself!" (Kohlbrenner, http://www.cne.gmu.edu/itcore/virtualmachine/history.htm). Virtual machines have played, and will continue to play, an important role in computational mobility because they provide a hardware-independent environment in which to execute a program. In the previous section on computational mobility, the concept of a computation environment was introduced by Fuggetta et al (1998). A computation environment in this context is just a virtual machine specifically constructed for computational mobility whether that mobility be strong or weak. While virtual machines are important to computational mobility, the history of virtual machines predates any type of code mobility by almost two decades.

Operation Systems

Employees of IBM working at the Yorktown Research Center created one of the first virtual machines around 1965. The virtual machine was created to allow a single physical machine to be divided into multiple smaller virtual machines. These virtual machines needed to be able to manage their own resources so that the effect of adding or

removing features from the system could be evaluated. Over time, this experimental environment was converted into the IBM VM/ESA operating system which is used by the IBM System 370 and the IBM System 390 (Kohlbrenner, http://www.cne.gmu.edu/itcore/virtualmachine/history.htm).

UCSD Pascal

In the late 1960's or early 1970's, the Pascal language, named in honor of the French mathematician Blaise Pascal, was designed as a successor to Algol by Niklaus Wirth working at the Eidgenössische Technishe Hochshule in Zurich, Switzerland (Leestma and Nyhoff, 1984). Pascal was considered a significant improvement over Algol, as it supported the capability to:

- create new data types from simpler existing ones, and
- create dynamic data structures which could grow and shrink at run-time

(Pascal, www.taoyue.com/tutorials/pascal/history.html).

By 1974, Professor Kenneth Bowles at the University of California San Diego wanted to provide an environment with an integrated compiler, editor, and debugger that would allow students to interactively develop programs on the new class of microcomputers, such as the DEC PDP-11 or Apple II. To support this concept, Professor Bowles needed a simpler method of porting a programming language than just rewriting the compiler for each new microcomputer architecture. Allowing a Pascal compiler to generate P-code, an intermediate language, which would be executed by a virtual machine rather than by the native hardware provided just what Professor Bowles

needed. Thus, UCSD Pascal planted the seeds of a programming language that allowed a program to be written once and then run on many different machine architectures. Virtual machines have even played a role in making non-standard hardware environments available to a larger group of users.

Lisp Virtual Machine

Symbolics is a company based around providing a high-end Lisp development environment known as Genera. In an era where windowing systems were seldom seen, Genera provided extensive capabilities for developing Graphical User Interfaces (GUI). To execute the Genera system, Symbolics developed a custom line of workstations specifically designed to efficiently execute Lisp code in hardware. One of the key features of the workstations is their unique object-based memory architecture. "The use of tagged memory to support dynamic typing and generic operations allowed the Lisp machine to give competitive performance in the absence of carefully declared types, a key feature in support of its rapid prototyping capability" (Robertson, 1994). The Lisp machines also had unique features that supported garbage collection. As non-custom RISC-based workstations continued to improve in performance and price, market pressure was placed on Symbolics to provide the Genera system on these standard workstation platforms. Rather than attempt to port the approximately 1.5 million lines of Genera code, a Virtual Lisp Machine was created for the DEC Alpha. To increase performance, some of the Virtual Lisp Machine was written in microcode. Version 1.0 of the Virtual Lisp Machine retained the performance of high-end custom Lisp Machines.

So while the Java Virtual Machine (JVM) and its intermediate language known as Java bytecode are currently the best known examples of a virtual machine, they certainly are not the first or only examples.

APPENDIX H

COMPUTER HARDWARE ARCHITECTURE

As stated previously, a virtual machine is just a program that behaves like a real machine. With that fact in mind, to form a better understanding of how a virtual machine might operate, let us look at how real hardware operates. Computer hardware architectures can be divide into stack-based architectures, accumulator-based architectures, register-memory-based architectures, register-register-based architectures, and memory-memory-based architectures.

Stack-Based Architecture

In a stack-based architecture, each operand of an instruction must be placed on the processor stack to be accessed. The result of an instruction is always placed back on the stack. Instructions are provided to move data from memory to the stack and from the stack to memory. Because only the push and pop instruction can address memory and all other instructions implicitly address the stack, stack-based instructions are referred to as zero-address instructions. Instructions to perform the following operation, A + B = C, would look as follows (Hennessy and Patterson, 2003):

```
PUSH A
PUSH B
ADD
POP C
```

Accumulator-Based Architecture

In an accumulator-based architecture, each instruction implicitly addresses a special register known as the accumulator. Load and store operations are provided to move data from memory to the accumulator and from the accumulator to memory. Accumulator-based architecture instructions are referred to as one-address instructions because, at most, one address is needed for an instruction. Instructions to perform the following operation, A + B = C, would look as follows (Hennessy and Patterson, 2003):

LOAD A

ADD B

STORE C

Register-Memory-Based Architecture

In a register-memory-based architecture, the operands of an instruction can be either a register or memory. Register-memory-based architecture instructions can be either two-address or three-address instructions depending upon the number of operands per instruction to be supported. Instructions to perform the following operation, A + B = C, using two-address instructions, would look as follows (Hennessy and Patterson, 2003):

LOAD R1, A

ADD R3, R1, B

STORE R3, C

Register-Register-Based Architecture

In a register-register-based architecture, the operands of an instruction are always a register. A load and store instruction is provided that can address memory. Register-register-based architecture instructions can be either two-address or three-address instructions depending upon the number of operands per instruction to be supported. Instructions to perform the following operation, A + B = C, using two-address instructions, would look as follows (Hennessy and Patterson, 2003):

```
LOAD R1, A
LOAD R2, B
ADD  R3, R1, R2
STORE R3, C
```

Memory-Memory-Based Architecture

In a memory-memory-based architecture, the operands of an instruction always address memory. Memory-memory-based architecture instructions can be either two-address or three-address instructions depending upon the number of operands per instruction to be supported. Instructions to perform the following operation, A + B = C, using three-address instructions, would look as follows:

```
ADD A, B, C
```

RISC-Based Architecture

A Reduced Instruction Set Computer (RISC) has the following characteristics (Mano, 1993):

1. relatively few instructions,
2. relatively few addressing modes such as immediate and relative,
3. memory access limited to load and store instructions,
4. all operations done within the registers (register-register-based architecture) of the CPU,
5. fixed-length instruction format, which allows for faster instruction decoding,
6. single-cycle instruction execution is possible because of the limited number of instructions,
7. hardwired vs. micro-program controlled.

Addressing Modes

Any instruction, as long as it is not a zero-address instruction, is composed of an operator and one or more operands. The operator is used to specify the action or operation to be performed when this instruction is executed. The operands of an instruction are used to specify the arguments the operator is to work on. The various ways in which the operands of an instruction can be specified are known as addressing modes. Some of the more common addressing modes are described below. Addressing modes are not mutually exclusive; rather they can build upon each other to create more

complex addressing modes. Information for the addressing mode table (Table H.1) came from Hennessy and Patterson (2003) and Mano (1993).

Table H.1. Addressing Modes.

Addressing Mode	*Example Instruction*	*Description*
Register	ADD R1, R2	The operands are always registers. Example: The value in register R2 is added to the value in register R1. The result of the operation is placed in register R1.
Immediate	ADD R1, #3	One of the operands is a constant value. This addressing mode is used to initialize a register to a known value. Example: The constant 3 is added to the value in register R1. The result of the operation is placed in register R1.
Register Indirect	ADD R1, (R2)	One of the register operands contains an address to the location of a data value. Example: Register R2 contains an address; the value at the address will be added to the value of register R1. The result of the operation is placed in register R1.

Table H.1. Addressing Modes, cont.

Addressing Mode	*Example Instruction*	*Description*
Direct	ADD R1, (1024)	One of the operands contains an actual memory address. The value at the memory address will be used by the operation. Example: The data value at memory address 1024 will be added to the value in register R1. The result of the operation is placed in register R1.
Indexed	ADD R1, (R2 + R3)	One operand consists of two parts. The first part will be memory address, and the second part will be an index that will be added to the memory address to compute an actual memory address. This addressing mode is very similar to the Base Register Addressing Mode. Example: Register R2 contains a memory address. Register R3 is an index register whose contents will be added to the address in R2. The data value at the resulting address will be added to the value of register R1. The result of the operation is placed in register R1.
Displacement	ADD R1, 1024(R2)	One operand consists of two parts. The first part is an actual memory address, and the second part is a displacement register that will be added to the memory address to compute a new actual address. Example: The value in register R2 will be added to the constant memory address of 1024, resulting in a new memory address. The value at the new memory address will be added to the contents of register R1. The result of the operation is placed in register R1.

Table H.1. Addressing Modes, cont.

Addressing Mode	*Example Instruction*	*Description*
Memory Indirect	ADD R1, @(R2)	One operand contains an address; the value at the specified address is used as an address to find the actual data value. Example: Register R2 contains an address. The data value at the address of R2 is used as an address to find the actual value that will be added to the contents of register R1. The result of the operation is placed in register R1.
Auto-increment	ADD R1, (R2)+	One of the register operands contains an address to the location of a data value. Each time the instruction is executed, the value of the register is automatically incremented by some amount dX which is proportional to the data type the operator executes on. This mode is very similar to register indirect, only an increment is automatically done. Example: Register R2 contains an address; the value at the address will be added to the value of register R1. The value of register R2 will be automatically incremented by some amount dX. The value of dX is based on the data type being added. The result of the operation is placed in register R1.

Table H.1. Addressing Modes, cont.

Addressing Mode	*Example Instruction*	*Description*
Auto-decrement	ADD R1, (R2)+	Auto-decrement is just like the auto-increment address mode, only a decrement is done in place of an increment. Example: Register R2 contains an address; the value at the address will be added to the value of register R1. The value of register R2 will be automatically decremented by some amount dX. The value of dX is based on the data type being added. The result of the operation is placed in register R1.
Scaled	ADD R1, 1024(R2)[R3]	One operand contains three parts. The first part is a memory address, the second part is a displacement register and the third operand is a scaled register. The scaled register is multiplied by some amount dX. The value of dX is proportional to the data type the operator executes on. The value of the displacement register and the scaled result are added to the memory address to compute a new memory address. Example: The value in register R2 is added to memory address 1024, resulting in a new memory address. Register R3 is multiplied by some value dX. The value of dX is based on the data type being added. The result of the multiplication will then be added to the new memory address, resulting in an additional memory address. The data value at this additional memory address will be added to the contents of register R1. The result of the operation is placed in register R1.

Instruction Pipeline

An individual instruction in a computer is generally processed in five stages. The stages when placed one after the other form an instruction pipeline. A pipeline can be used in parallel to execute multiple instructions that are at different stages of the pipeline process. A pipeline normally consists of the following classic five stages:

- Instruction Fetch Stage. In this stage, an instruction pointed to by the Program Counter (PC) is fetched from memory and placed in the Instruction Register (IR). The PC can be incremented to point at the next instruction in memory.
- Instruction Decode Stage. In this stage, the instruction in the IR is decoded to determine the type of instruction and the registers that will be accessed by the instruction.
- Execution/Effective Address Stage. In this stage, the effective address, a memory address to be accessed, of an operand is computed using the specified addressing mode and register decoded in the previous stage.
- Memory Access Stage. In this stage, the effective address computed in the previous stage is used to fetch data from memory or, in the case of a store instruction, data is placed into memory.
- Write-Back Stage. In this stage, the result of the operation is written to a register. The Figure H.1 shows the five stages of a classic pipeline.
    - Instruction Fetch (IF): Fetch the instruction at the current program counter PC.

- Instruction Decode (ID): Decode the fetched instruction; read register values for the fetched instruction.
- Execution (EX): Execute the instruction on the operands.
- Memory Access (MEM): For a load or store instruction, access memory using the effective address computed in the previous stage of the pipeline.
- Write-Back (WB): Write the result of the instruction into a register.

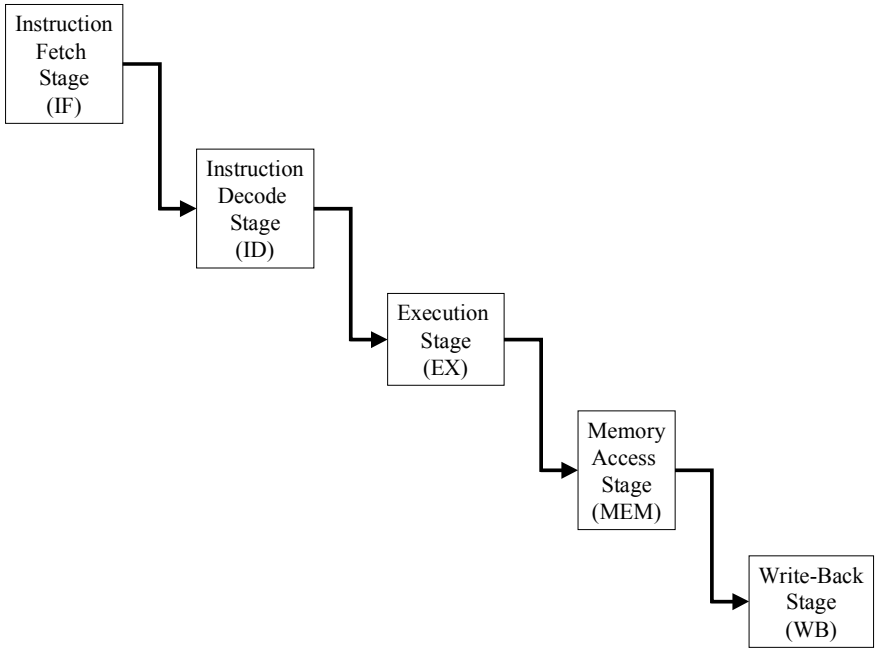


Figure H.1. Classic Five-Stage Instruction Pipeline.

APPENDIX I

AUTOMATIC RECOVERY OF A COORDINATOR

Extending the execution model to support automatic recovery of a failed coordinator is a natural extension to the execution model. The automatic recovery of a coordinator was not implemented in the prototype. The key issue in automatic recovery of a failed coordinator is ensuring that the most current copy of the coordinator is restarted. To facilitate automatic recovery, a coordinator shadow will be used. The shadow is a copy of a coordinator that is left behind on a host machine after the coordinator has projected. The shadow is created from the coordinator as it projects to a new host machine. Creating a shadow at the time of projection ensures that the most current copy of the execution state of the coordinator is captured in the shadow. The Coordinator Shadow figure (Figure I.1) will be used to help illustrate automatic recover. Each box in the Coordinator Shadow figure represents a different host machine. The arrows between the boxes show the order in which the coordinator is projecting to each host machine. The box labeled with an 'C' represents the current location of the coordinator on a host machine. The label 'C' will be used to refer to both the coordinator and the host machine. The text will always identify whether it is referring to a host machine (host C) or a coordinator (coordinator C).

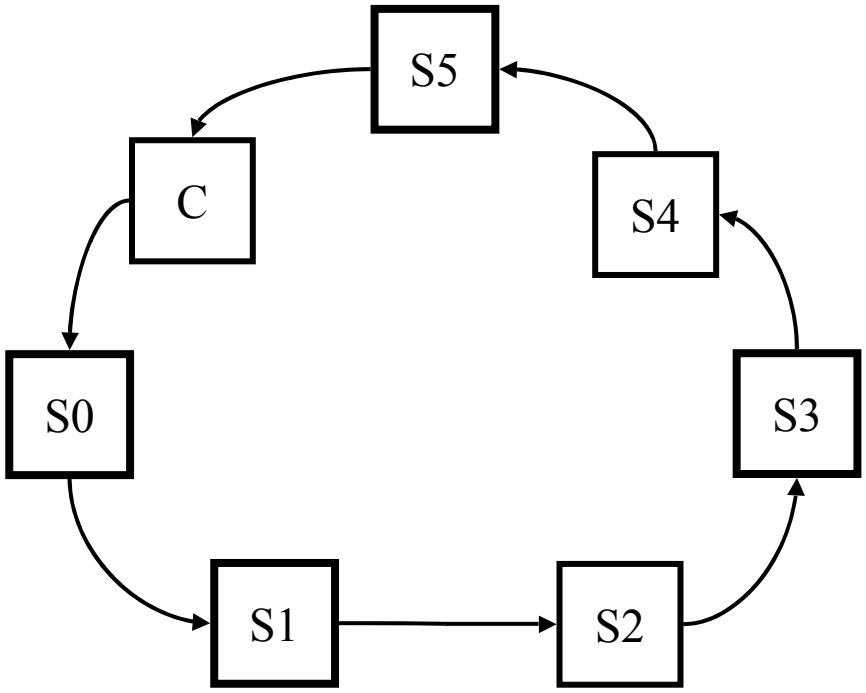


Figure I.1. Coordinator Shadow.

The boxes labeled S0 through S5 represent six different versions of a coordinator shadow on six different hosts for the coordinator C. The labels S0 through S5 will be used to refer to both a specific host machine (host S2) and a specific shadow (shadow S2). The shadow S5 is the most current shadow of the coordinator because it was the last host machine from which the coordinator projected; shadow S4 is the second most current shadow and so on with shadow S0 being the oldest shadow.

Heartbeat

Each shadow receives one heartbeat from the coordinator when the coordinator is projected onto the same host as the shadow. A shadow has a heartbeat interval, which is the maximum of time the shadow will wait to receive a heartbeat before attempting a

recovery. By default, the heartbeat interval will be set to infinity, thus suppressing automatic recovery of coordinators. The heartbeat interval begins to countdown to zero as soon as the coordinator has projected to a new host. To set the heartbeat interval of a coordinator shadow, the following coordinator specific library function can be called: heartbeatInterval(long maxWait).

Before looking at the full issue of automatic recovery of a coordinator, we will first explore how the execution model can handle the catastrophic failure of a host machine. We will use the Unreachable Resource figure to help explain this example. If a coordinator attempts to project to host S0, which has suffered a catastrophic failure, it will be unable to because host C will be unable to establish communication with host S0. As a result, the projection operation will fail. The failure of the projection operation will cause an exception to be thrown within the executing coordinator. The exception will be thrown from the statement where the coordinator was attempting to access the portal of a resource, which is located on host S0. At the instruction level within the L virtual machine, it will be a pcall that throws the exception when communication cannot be established with host S0. Upon catching the exception, the coordinator can determine if the application can proceed without the use of the resource that was located on host S0 or if the coordinator should halt. If the application can proceed, the portal of a different resource can be accessed. In Figure I.2, the coordinator projects to host S1 after failing to project to host S0.

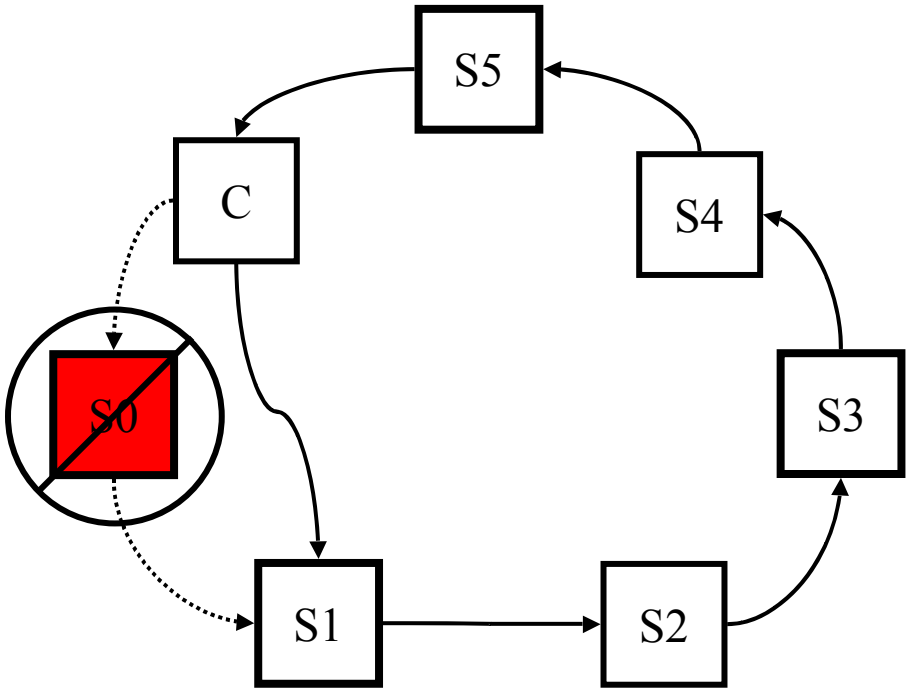


Figure I.2. Unreachable Resource.

The exception that will be raised when a resource cannot be accessed is an UnreachableResourceException. This exception allows a developer to distinguish between some type of memory failure and a possible communication failure. If the application is such that resources can wander in and out of communication range, then the recovery strategy used when a resource is unreachable would be very different from that used if a resource suffered a catastrophic failure. In such an application, the developer would want to distinguish between being unable to reach a resource and memory appearing to be corrupted (null point exception / segmentation error).

Basic Recovery Algorithm

1. When the heartbeat of a shadow Sn is missed, the virtual machine will attempt to restart the coordinator using shadow Sn. As part of the restart, host Sn will attempt to contact host Sn+1. The virtual machine on host Sn can determine which host is Sn+1 for a coordinator by requesting the information from the coordinator's shadow. If communication is established with host Sn+1, the virtual machine will be able to determine if a more current (newer) shadow for the coordinator exists on host Sn+1. If a new shadow exists on host Sn+1, then no restart should be attempted using the older coordinator shadow on host Sn.
2. If communication between host Sn and Sn+1 can be established, then host Sn will attempt to establish communication with every host in the resource host table looking for a newer shadow. If a newer shadow is found, host Sn will not perform a restart using shadow Sn. An algorithm that simply restarts a coordinator from its shadow when a heartbeat is missed has been considered and rejected. While this algorithm seems cleaner, it would only work if all portal operations were idempotent. That is, all portal operations would have to be constructed such that performing the operation multiple times would all have the same effect as if the operation had been performed exactly once. Any portal operation that changes the state of the resource is not an idempotent operation.
3. If the virtual machine on host Sn is unable to find a newer shadow for the coordinator, then a restart will be performed using the shadow on host Sn. When the coordinator is restarted, a RestartException will be raised, thus allowing the

application to take some appropriate action because of the restart. At the instruction level within the L virtual machine, it will be a pcall that throws the RestartExecption.

4. Step 1 will be repeated each time that a heartbeat is missed.

Coordinator Recovery: Simplest Case

Now let us consider a case where the host on which a coordinator is currently executing suffers a catastrophic failure. While resources located on other hosts will continue to operate, the application has lost its coordinator. Utilizing the heartbeat, we can initiate a coordinator recovery. For recovery to occur, the heartbeat interval must be set to a value less than infinity. If the heartbeat of each shadow was set to the same value and the coordinator was projecting in a clockwise rotation around the hosts shown in Figure I.3, then the shadow S0 would be the first to miss a heartbeat. Shadow S0 will be the first to miss a heartbeat because it is the oldest shadow in the Coordinator Recovery Simple Case figure, whereas shadow S5 is the youngest. When shadow S0 misses its heartbeat, the virtual machine that is running on host S0 will begin a recovery starting at step 1 of the basic recovery algorithm.

1. Host S0 attempts to contact host S1. Upon contact with host S1, host S0 will determine that shadow S1 is a newer shadow and recovery on host S0 will stop. When the heartbeat for shadow S1 is missed, a recovery will be started. Host S1 will determine that host S2 has a more current shadow and recovery will stop. The process of missing heartbeats and stopping recovery will continue until host S5 attempts to recover.

2. Host S5 will be unable to communicate with host C, as it has had a catastrophic failure. Upon failing to contact host C, host S5 will attempt to communicate with all hosts known to have a resource, hosts S0 through S4.
3. Through this communication, host S5 will determine that it has the most current coordinator shadow and a restart of the coordinator will be started using the shadow S5. The restart will begin executing the coordinator at the same statement that caused a projection from host S5 to host C; this is always a pcall instruction. When the statement is re-executed, a RestartException will be thrown. If the coordinator catches the exception, it will be able to perform appropriate application recovery actions, such as removing the reference to the resource on host ‘C’ and skipping to the next resource which is located on host S0.

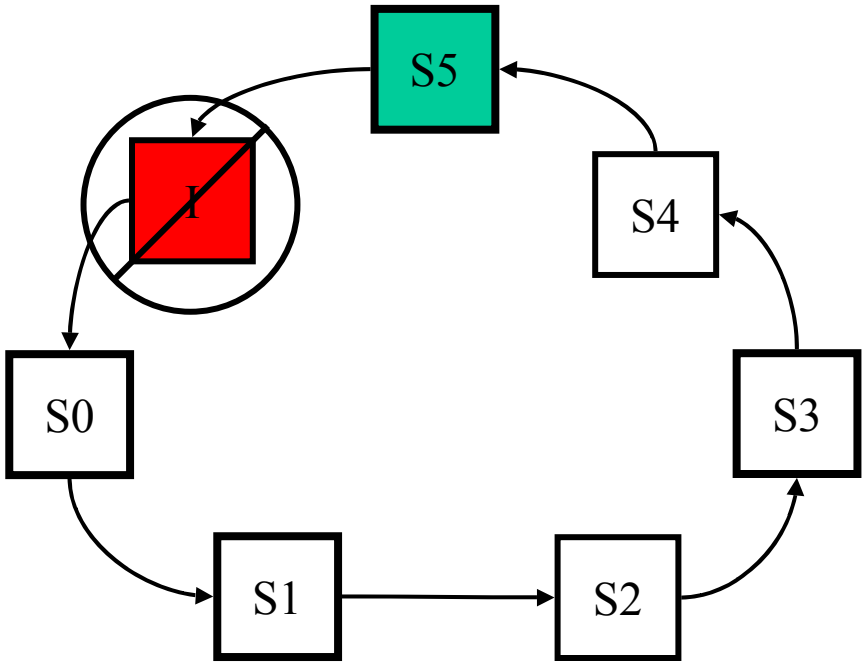


Figure I.3. Coordinator Recovery Simple Case.

Coordinator Recovery: Complex Case

The complex case for coordinator recovery involves multiple points of failure. Figure I.4 shows the case with multiple points of failure. In this case host S1, host S3 and host S5 will all be unable to connect to host Sn+1. This will cause each of these hosts to start to communicate with all possible hosts in the resource host table looking for newer shadows. Hosts S1 and S3 will soon discover that there exists a host with a newer coordinator shadow and stop their recovery, leaving only host S5 to perform the actual recovery as outlined in the simple case.

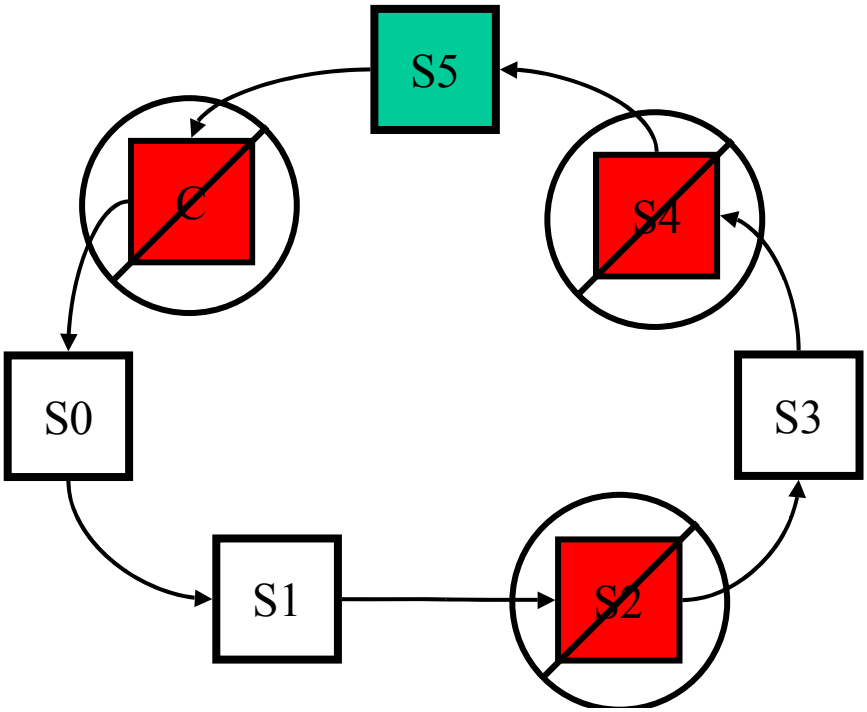


Figure I.4. Coordinator Recovery Complex Case.

Establishing the Heartbeat Interval

A method for establish the heartbeat interval for a coordinator would be:

1. Time how long it takes to access all of the resources of an application.
2. Over multiple iterations through all the resources, compute the maximum time it takes before the coordinator revisits each resource.
3. Set the heartbeat interval to be the maximum time plus some small additional time value (10%) to ensure that recovery does not start each time the maximum value is reached.

Coordinator Exit vs. Coordinator Abort

If the execution model is made to support automatic coordinator recovery, then an application must distinguish between an exit and a failure; otherwise, coordinator recovery could occur each time a coordinator simply exited. The simplest method by which automatic recovery can be suppressed is to set the heartbeat interval of all existing shadows to infinity. When an application is ready to exit, it would need to perform one additional access to every resource after setting the heartbeat interval to infinity: heartbeatInterval(-1).

APPENDIX J

L BYTECODE

The bytecode for L was derived in large part by reviewing the MIPS R2000 instruction set and the instruction set used by the Java virtual machine. The following tables outline the quadruples used by the L virtual machine. Operands for a bytecode are either:

- Var for a variable.
- Label for an integer that is the location of a bytecode instruction.
- Imm for an integer which is immediate data to an instruction.

Table J.1. Jump Instructions.

Op Code	*Description*	*Operands*
bnz	Branch to a label if the variable is not zero.	Label, Var
jmp	Branch to a label.	Label

Table J.2. Move Instruction.

Op Code	*Description*	*Operands*
move	Move variable 2 to variable 1	Var1, Var2

Table J.3. Subroutine Instructions.

Op Code	*Description*	*Operands*
call	Subroutine call, which will branch to label and compute the return counter as program counter + 1.	Label
pcall	Portal call—this single instruction handles coordinator migration and will invoke a portal subroutine. For the pcall instruction to work, a reference to a resource proxy object must be on the pstack before this instruction is executed. The virtual machine will check if the coordinator and the resource are located on the same host machine by checking the resource host table. If both objects are on the same host: the reference to the resource proxy currently on the pstack will be replaced by a reference to the actual resource. A branch will be executed to the label of the portal subroutine. A return program count will be computed and saved to an activation record (frame) for the portal subroutine call. If the resource object and the coordinator object are not on the same host, the coordinator, which is the object currently being executed, and all of its associated execution state will be migrated to the host machine of the resource. Once migration of the coordinator object is complete, the pcall instruction will be executed again.	Label
frame	Creates an activation record on the astack for a called subroutine. The activation record will allocate enough space for local variables using the immediate values. The return counter (return address) for the subroutine will be saved in the activation record.	Imm
return	Pushes a variable onto the pstack and pops an activation record off the astack. The program counter is set to the return counter stored in the activation record.	Var
rtn	Pops an activation record off the astack. The program counter is set to the return counter stored in the activation record.	

Table J.4. Memory Instructions.

Op Code	*Description*	*Operands*
new	Allocate space for a new object using the immediate value as an index into class record table. A reference to the new object is placed in the variable.	Var, Imm
newarray	Allocate an array using the value in variable 2 as the size of the array. A reference to the new array is placed into variable 1. The value of immediate indicates the type of the object to be referenced in the array. This instruction was derived from the newarray instruction of the Java virtual machine.	Var1, Imm, Var2

Table J.5. Parameter Stack Instructions.

Op Code	*Description*	*Operands*
push	Push an object in variable onto the pstack.	Var
pop	Pop an object off the pstack and place it into the variable.	Var

Table J.6. Logical Instructions.

Op Code	*Description*	*Operands*
and	Perform a boolean and operation using variable 2 and variable 3 and place the result in variable 1.	Var1, Var2, Var3
or	Perform a boolean or operation using variable 2 and variable 3 and place the result in variable 1.	Var1, Var2, Var3

Table J.7. Arithmetic Instructions.

Op Code	*Description*	*Operands*
addi	Add integers, add variables 3 to variable 2 and place the result in variable 1.	Var1, Var2, Var3
addf	Add floating-point numbers, add variable 3 to variable 2 and place the result in variable 1.	Var1, Var2, Var3
subi	Subtract integers, subtract variable 3 from variable 2 and place the result in variable 1.	Var1, Var2, Var3
subf	Subtract floating-point numbers, subtract variable 3 from variable 2 and place the result in variable 1.	Var1, Var2, Var3
muli	Multiply integers, multiply variable 3 by variable 2 and place the result in variable 1.	Var1, Var2, Var3
mulf	Multiply floating-point numbers, multiply variable 3 by variable 2 and place the result in variable 1.	Var1, Var2, Var3
divi	Divide integers, divide variable 2 by variable 3 and place the result in variable 1.	Var1, Var2, Var3
divf	Divide one floating-point numbers, divide variable 2 by variable 3 and place the result in variable 1.	Var1, Var2, Var3
mod	Modulus operator, divide variable 2 by variable 3 and place the remainder from the division in variable 1.	Var1, Var2, Var3

There are a large number of compare instructions because the instructions are data type specific. There are six basic compare instructions: eq, sle, sge, sne, slt, sgt which were derived from the MIPS R2000 instruction set and the eqnull instruction which was derived from the Java virtual machine instruction ifnull.

Table J.8. Compare Instructions.

Op Code	*Description*	*Operands*
eqi	Equal integers, if variable 2 is equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
eqf	Equal floating-point numbers, if variable 2 is equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3

Table J.8. Compare Instructions, cont.

Op Code	*Description*	*Operands*
eqs	Equal strings, if variable 2 is equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
eqc	Equal character data, if variable 2 is equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
eqnull	Null Test, check if the object reference variable 2 is null, if null place true in variable 1 else place false.	Var1, Var2
slei	Less than or equal integers, if variable 2 is less than or equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
slef	Less than or equal floating-point numbers, if variable 2 is less than or equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
sles	Less than or equal strings, if variable 2 is less than or equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
slec	Less than or equal character data, if variable 2 is less than or equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
sgei	Greater than or equal integer, if variable 2 is less than or equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
sgef	Greater than or equal floating-point numbers, if variable 2 is less than or equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
sges	Greater than or equal strings, if variable 2 is less than or equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
sgec	Greater than or equal character data, if variable 2 is less than or equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
snei	Not equal integers, if variable 2 is not equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
snef	Not equal floating-point numbers, if variable 2 is not equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3

Table J.8. Compare Instructions, cont.

Op Code	*Description*	*Operands*
Snes	Not equal strings, if variable 2 is not equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
snec	Not equal character data, if variable 2 is not equal to variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
slti	Less than integers, if variable 2 is less than variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
sltf	Less than floating-point numbers, if variable 2 is less than variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
slts	Less than strings, if variable 2 is less than variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
sltc	Less than character data, if variable 2 is less than variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
sgti	Greater than character data, if variable 2 is greater than variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
sgtf	Greater than floating-point number, if variable 2 is greater than variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
sgts	Greater than strings, if variable 2 is greater than variable 3 place true in variable 1 else place false.	Var1, Var2, Var3
sgtc	Greater than character data, if variable 2 is greater than variable 3 place true in variable 1 else place false.	Var1, Var2, Var3

The following library instructions are implemented directly by the virtual machine. To add new library instructs, both the compiler and the virtual machine must be changed.

Table J.9. Library Instructions.

Op Code	*Description*	*Operands*
write_int	Write an integer referenced by the pstack to standard out. The reference is popped from the pstack.	
write_float	Write a floating pointer number referenced by the pstack to standard out. The reference is popped from the pstack.	
write_str	Write a string referenced by the pstack to standard out. The reference is popped from the pstack.	
array_len	Compute the length of an array referenced by the pstack. The reference is popped from the pstack. The length of the array is pushed on the pstack.	
str_concat	Concatenate the second string referenced by the pstack to the first string referenced by the pstack. Both references are popped from the pstack. A reference to the concatenated string is pushed on the pstack.	
i2f	Convert an integer referenced by the pstack to a floating-point number. . The reference is popped from the pstack. A reference to the floating-point number is pushed on the pstack.	
f2i	Convert a floating pointer number referenced by the pstack to an integer. The reference is popped from the pstack. A reference to the integer is pushed on the pstack.	
yield	Cause the current thread to pause and allow other thread to execute.	
s2i	Convert a string referenced by the pstack to an integer. The reference is popped from the pstack. A reference to the integer is pushed on the pstack. An exception is generated if the string is not a valid integer. The exception can not be caught because catch and throw statements were not implemented in the prototype.	

Table J.9. Library Instructions, cont.

Op Code	*Description*	*Operands*
s2f	Convert a string referenced by the pstack to a floating-point number. The reference is popped from the pstack. A reference to the floating-point number is pushed on the pstack. An exception is generated if the string is not a valid floating-point number. The exception can not be caught because catch and throw statements were not implemented in the prototype.	

Table J.10. Find Instructions.

Op Code	*Description*	*Operands*
find	Find a resource proxy for a specific resource by unique name and type. The immediate value is an index into the class record table, this will identify the type of the resource. The unique name of the resource is passed as a reference on the pstack. The reference is popped from pstack. A proxy for the resource is created and a reference to the proxy is pushed onto the pstack.	Imm
findall	The find all instruction is not implemented. When implemented the instruction will use the immediate value to find all the resources for a specific type of resource. A proxy will be created for each resource find. An array will be created to hold all of the found resource proxies. A reference to the array will be pushed onto the pstack.	Imm

L Virtual Machine Addressing Modes

Unlike C, Java does not support a data type that references blocks of memory or supports pointer arithmetic. As a result, the virtual machine was constructed using arrays

of objects to represent a block of memory. To access an object, an index into an array must always be constructed and used in combination with the base array. The base address of the array and the index can never just be added together, because pointer arithmetic is not supported. While this was initially a little cumbersome, the end result was easier to debug because there are no pointers or pointer arithmetic. Some new addressing modes were defined for operating in this environment.

Table J.11. L Virtual Machine Addressing Modes.

Mode	*Syntax*	*Description*
Global	X	This is not a true addressing mode with respect to machine instructions but this is an addressing mode implemented in the virtual machine. The global addressing mode is used to access data in the constant pool.
Local	^X	Local addressing is always used to access the local variables of a subroutine. The ^X syntax informs the virtual machine to use the X^{th} variable from the activation record on top of the astack.
Relative	^X^Y	Relative addressing is always used to access variables relative to an object (resource, coordinator, class) instance. The ^X^Y syntax informs the virtual machine to use the X^{th} variable from the activation record on top of the astack and then use the Y^{th} variable from that object. As a convention the 0^{th} variable of an activation record is always the object on which a subroutine was invoked. Thus relative addressing operands appears as "^0^Y", were why is set to the index of the instance variable to be accessed.
Indexed	X+Y	The indexed address mode is always used to access elements of an array. Y is an index for the array X. The indexed addressing mode can be used in combination with global, local or relative addressing. Only one level of indexing is supported at a time, thus X+Y+Z is not valid.
Immediate	#	The # symbol is replaced by a numerical constant.