SPIDER: An Agent-Based Message-Passing Architecture

by

DORIAN BIRSAN
B.Math (Computer Science and Combinatorics & Optimization)
University of Waterloo, 1993

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENT FOR THE DEGREE OF
MASTER OF SCIENCE
in
THE FACULTY OF GRADUATE STUDIES
Department of Computer Science

We accept this thesis as conforming to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
June 1995
© Dorian Birsan, 1995
In presenting this thesis in partial fulfillment of the requirements for an advanced degree at
the University of British Columbia, I agree that the Library shall make it freely available for
reference and study. I further agree that permission for extensive copying of this thesis for
scholarly purposes may be granted by the head of my department or by his or her representa-
tives. It is understood that copying or publication of this thesis for financial gain shall not be
allowed without my written permission.

Department of Computer Science
The University of British Columbia
Vancouver, Canada

Date: July 7, 1995
Abstract

Communication is a driving force in the high-performance parallel computing environment of the 90's. Obtaining good performance eventually requires optimizing communication, hence the importance of controlling network resources. Current network designs offer a closed, rigid interface that emphasizes hardware solutions for routing, leaving little room for communication control. In this thesis we address interface issues between network hardware and communication software. An open, flexible interface is proposed, which does not enforce communication policies, but offers mechanisms for user control over messaging. Control is exerted by programming the network, using a message-centered approach, as opposed to traditional node-centered approaches. This distinction is based on where the control lies: in the message or in the node. Messages are no longer passively communicated from source to destination, but are instead active, intelligent, and self-routing. We call them communication agents.

SPIDER (Simple Programmable Interface Design for Efficient Routing) is a user-programmable routing kernel that supports the message-centered programming paradigm with communication agents. We present a general design and a system prototype for a transputer-based multicomputer. The implementation was used to explore the benefits of programming the network. It was found that the paradigm has expressive power for solving communication tasks, is general, reliable and can improve performance in communication-intensive applications.
# Table of Contents

Abstract ............................................................................................................. ii

Table of Contents ............................................................................................... iii

List of Tables ........................................................................................................ vi

List of Figures ....................................................................................................... vii

Acknowledgment ................................................................................................. viii

CHAPTER 1 Introduction 1
  1.1 Networks: Open vs. Closed ................................................................. 1
  1.2 Motivation .............................................................................................. 3
  1.3 Methodology ......................................................................................... 5
  1.4 Thesis Contributions ............................................................................ 8
  1.5 Thesis Outline ....................................................................................... 9

CHAPTER 2 Programming the Network 10
  2.1 Background ......................................................................................... 10
    2.1.1 Network Routing ........................................................................ 11
    2.1.2 Flow Control ............................................................................... 12
  2.2 Message-Centered Programming ....................................................... 15
  2.3 Example ............................................................................................... 18
  2.4 Summary .............................................................................................. 21

CHAPTER 3 Architecture of SPIDER 22
  3.1 Overall Design .................................................................................... 22
  3.2 Environment Model ........................................................................... 26
  3.3 Environment and Agents .................................................................... 29
  3.4 Agent Architecture ............................................................................ 31
  3.5 Instruction Set .................................................................................... 36
  3.6 Assembly Language ........................................................................... 40
  3.7 Agents and the Real World ............................................................... 41
  3.8 Summary ............................................................................................ 43

CHAPTER 4 Implementation 45
  4.1 Environment ...................................................................................... 46
    4.1.1 Hardware ................................................................................... 46
    4.1.2 Software ................................................................................... 47
  4.2 Kernel Design and Implementation .................................................. 48
A.5 Mesh DOR Agent ................................................................. 120
A.6 Adaptive Mesh Routing Agent ............................................. 123
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.1</td>
<td>SPIDER Instruction Set</td>
<td>39</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>SPIDER assembler directives (pseudo-instructions)</td>
<td>41</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Broadcast using low level ChanIn/Out: chan.tree</td>
<td>75</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Broadcast using virtual channels: vchan.tree</td>
<td>75</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Broadcast using virtual channels: vchan.send</td>
<td>76</td>
</tr>
<tr>
<td>Table 5.4</td>
<td>Broadcast using SPIDER agents: spider.bcast</td>
<td>76</td>
</tr>
<tr>
<td>Table 5.5</td>
<td>Combine: the bitwise OR on a word</td>
<td>81</td>
</tr>
<tr>
<td>Table 5.6</td>
<td>Node-to-Node Routing: no traffic</td>
<td>83</td>
</tr>
<tr>
<td>Table 5.7</td>
<td>Node-to-Node Routing: on hot spot</td>
<td>85</td>
</tr>
<tr>
<td>Table 5.8</td>
<td>Cholesky Factorization of a 65x65 matrix</td>
<td>89</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2.1: Basic message passing architecture ........................................... 11
Figure 2.2: Schematic view of the agent world ........................................... 17
Figure 2.3: Example of node-centered broadcast ........................................ 19
Figure 2.4: Example of message-centred broadcast ..................................... 20
Figure 3.1: System layers ................................................................. 24
Figure 3.2: SPIDER environment ......................................................... 29
Figure 3.3: Agent structure ............................................................... 32
Figure 3.4: SPIDER instruction formats ............................................... 38
Figure 3.5: Programming with SPIDER .................................................. 42
Figure 4.1: SPIDER kernel process organization ...................................... 50
Figure 4.2: Main agent data structure .................................................. 53
Figure 4.3: Routing agent program: agent.prog ................................. 55
Figure 4.4: Assembler output: agent.instr .......................................... 56
Figure 4.5: Routing agent functional interface: agent.c ....................... 57
Figure 5.1: Broadcast agent ............................................................. 63
Figure 5.2: Tree-combine agent ......................................................... 66
Figure 5.3: Adaptive mesh routing agent ............................................. 70
Figure 5.4: Network setup ............................................................... 73
Figure 5.5: Broadcast comparison ....................................................... 79
Figure 5.6: Combine comparison ......................................................... 82
Figure 5.7: Traffic with and without faults ........................................... 87
Acknowledgment

I owe Alan Wagner, my supervisor, a special debt of gratitude. The time he committed to me and his genuine interest in my work help me greatly. I am especially grateful for the opportunity he gave me to investigate a topic which takes a radical position with respect to well-established approaches. I would also like to thank Dr. James Little for taking time to read the thesis and offer suggestions for improvement.

I take this opportunity to thank all the friends who make my stay in Vancouver an enjoyable experience. Skiing, white river rafting, intramurals, or just conversations and companion­ships have created lasting memories (Go “Storm Masters” go!).

I thank the National Sciences and Engineering Research Council of Canada, as well as the British Columbia Advanced Systems Institute for their financial support.

All these could have not been possible without the love and support of my family, whose sacrifices I have many times taken for granted. This thesis is dedicated to them.
CHAPTER 1

Introduction

*The line, it is drawn, the curse, it is cast*
*The slow one will later be fast*
*And the present now will soon be the past*
*The order is rapidly fading*
*The first one now will later be last*
*For the times, they are a changing*

*Bob Dylan - "Times They Are A-Changin'"*

1.1 **Networks: Open vs. Closed**

During the last decade computing has been moving towards openness. Behind this trend is the need to cope with increased complexity and the need for customizability. Deployment of control and processing power is shifting from central, closed monolithic systems to smaller, open, independent systems that can be tailored to the application. This trend spans a broad range of computing, from architectures (personal computers), to operating systems (GLUnix[68], to applications (CORBA, OpenDoc[30]).

To date, multicomputer network designs stand against the trend. Current designs put an emphasis on hardware solutions for routing. Network designers try to provide a general purpose routing system that can optimally route both local and system-wide communication, with
regular or irregular communication patterns, possibly varying dynamically (Ciampolini [11]).

Stringent performance requirements of parallelism has left little room to allow for anything other than a hardware approach. The result is high raw performance with a closed and rigid interface to the user, with communication almost entirely controlled by the network. However, this “black box” approach overlooks a number of important pragmatic issues:

1. **Engineering constraints:**
   
   Building a network is an engineering problem that involves numerous constraints and trade-offs. Practically, there is a fundamental trade-off between reliability, performance, and cost.

2. **Wide range of network domains; heterogeneity:**
   
   The term network denotes anything from parallel machines's interconnects, to LAN’s and WAN’s, with broadcast medium (Ethernet), point-to-point (ATM) or fully connected (optical switches). These networks have different requirements, design, and performance (Lin[41], Sukup[65]). In addition, it is possible for high-performance applications to run on more than one network (Fox[29]).

3. **Measurement bias:**
   
   Focus on performance measurement has been on point-to-point or node-to-node, rather than on performance of the overall application.

4. **Portability:**
   
   It is common practice among programmers to make use of network facilities, even though this would rely on particular features of the network, leading to applications that are tightly dependent on specific networks.
5. **Customizability; efficient use of the network:**

For each application, communication characteristics such as different classes of traffic and communication patterns are difficult to efficiently map to hardware resources.

6. **Coordination:**

Most of the communication in parallel applications requires orchestration, as multiple processes are involved. Little work appears to have been done in this area (Nicolau[46]). Some networks do provide extra functionality for group communication. For example the CM-5 contains a combining network (Leiserson[40]).

### 1.2 Motivation

As applications demand more processing power (The High Performance Computing Act[31]) it is expected that the number of nodes to scale to hundreds and thousands. The cost of increased parallelism is intense communication (Johnsson [34]). Obtaining good performance eventually translates to optimizing communication, hence the importance of controlling network resources. Given the issues mentioned in the previous sections, a new approach with potential for further development is needed, to bring more openness to the network.

The solution proposed in this thesis has been inspired by the analogy of networks with transportation systems. Although the analogy is used throughout this thesis, it is important to notice that we do not try to implement it, but use it in three distinct ways: serving as a starting point, as a metaphor for the user, as well as an explanatory help.1
Networks of roads and highways have evolved over the last 50 years into elaborate and complex systems which have had to face many of the problems of current communication networks. Unlike networks, transportation systems are populated by intelligent agents (vehicles with drivers) that have full control over their movement. In a network, messages are passively carried from one location to another under the strict control of the network nodes. By briefly examining some of the desirable properties of transportation systems, we shall understand the need for intelligent agents in networks and become aware of their absence.

Traffic is regulated by signposts, traffic lights, and rules. These rules are generally the same everywhere, within or across countries. Transportation is a means to achieve other goals, to reliably move people or commodities from one place to another. Although an important part of other activities, transportation is self-contained and traffic controlled independently of the activities that generated it. There is no central authority that controls traffic, as this is too complex. Decisions are taken locally, each car follows its own (pre-established) itinerary or adapts to the local traffic conditions.

The system of roads is under continuing change and changes do not diminish reliability. Changes are only visible locally, other traffic is undisturbed, and changes are quickly accommodated. If a road fails, is congested or is closed, there are, usually, alternative roads to a destination. Cars can take better routes whenever the opportunity arises. How long it takes to get somewhere is usually predictable\(^1\). To beat the rush hour traffic, one can leave earlier and take

---

1. The problem inherent in any metaphor from the physical world, is that it must limit the user to some extent by constraining thought. The less the new systems work like an existing physical form, the more users will be encouraged to think of them as something with new potential. But designers cannot create metaphors without any frame of reference; users will hardly understand a system that is not at all like anything they have known before. Probably the best solution will be one that combines diverse metaphors in a surprising (but not unpredictable) form that will enable users to understand the corollaries and encourage them to think in new ways.

1. Without external interventions, in the long-run road traffic tends to reach equilibrium (Cohen[13])
advantage of the clear roads. All the cars have the same priority on the road. However, special vehicles, such as police cars, ambulances and firetrucks can switch to a high priority mode.

If messages were like cars (active, self-controlled), one can expect the network to become more open, with properties similar to those of a transportation system: adaptivity, portability, distributed control, fault tolerance, prioritized classes of traffic, predictability, separation of concerns (communication separated from computation).

1.3 Methodology

Our approach is based on the idea of an open network which the user is able to program by using messages as intelligent agents. The methodology is a proof-of-concept type addressing the following five issues:

Principle: Opening the Network

An open interface is proposed. The network is no longer viewed as an end-to-end medium; its internal structure (point-to-point links, buffer queues, etc.) is completely visible and the user interface to the network resources can be used to both access and control low-level message passing. Communication policies are part of the user specifications, rather than enforced by the network.

Paradigm: Programming the Network

Solving a computation problem is done by programming the processor. By analogy, we propose to solve communication problems by programming the network. The term “programming the
A network is used to imply something different from network programming where the focus is on high-level message passing and communication is tightly integrated with computation. The idea of programming the network suggests the existence of a communication programmer: there is a programmable interface to the hardware that allows the user to dynamically program and adapt it to each application. The communication programmer controls network resources, in particular the routing, flow control and buffering. Communication is decoupled from computation and managed separately. The idea of separating computation from communication can be also found in Felten’s Ph.D. thesis[22] and Linda (Gelernter[27]). The flexibility of programmability makes it possible to provide mechanisms rather than fixed policies.

**Mechanism: Message-Centered Programming**

To support the paradigm we introduce a system model called **SPIDER** (Simple Programmable Interface Design for Efficient Routing).

For programming the network, SPIDER proposes a dynamic mechanism, which we call *message-centered programming*, as opposed to the static mechanism which is node-centered. Classification into node or message centrism is based on where the communication control is:

- **node-centered**: control is on nodes. Nodes are instructed what to do upon message arrival (Dolter[19]). Network programming is typical example of node-centrism.
• *message-centered:* control is in the message. Messages contain the actions that program the network, and they are executed on each node visited by the message.

SPIDER's message-centered approach describes communication in a more natural way. An interesting analogy would be that of a process in which a bee collects pollen. Whereas in the traditional node-centered approach each flower is instructed to receive the bee, give it the pollen and pass the bee to the next flower, in our approach the bee is instructed to visit each flower and collect the pollen.

**Implementation: Communication Agents**

Inspired from the transportation analogy (Section 1.2) the "intelligence" is no longer concentrated at the nodes, it is distributed to the messages. By putting the control in the message, SPIDER transforms messages into self-controlled objects that deliver data in the network and route themselves. We call these messages *communication agents*. The network is populated with communication agents that travel between nodes according to the instructions they received, in an orchestrated action, with the end goal of reliably and efficiently transporting data needed by the parallel applications running on the machine. To send data to another node, an agent is created, the data is loaded onto the agent, and the agent executes its program to arrive at the destination.

---

1. For simplicity we will use the term "agent" if there is no confusion
Evaluation

Since SPIDER offers a programmable interface, it is expected that a large variety of communication problems can be expressed, including adaptivity and fault tolerance. In addition, an interpreted agent language ensures portability across architectures. Given this increased abstraction, of crucial importance is performance. The focus is not on improving node-to-node communication, but on improving overall application performance by intelligently coordinating communication, especially communication-intense problems.

1.4 Thesis Contributions

The focus of the thesis is on exploring the advantages of an open network with a message-centered programmable interface. The design of the hardware-software interface and its use in controlling communication was investigated. This thesis makes the following research contributions:

1. It introduces the idea of opening multicomputer networks by directly programming them. A dynamic message-centered based on messages as intelligent agents is investigated and contrasted with the static node-centered approach.

2. A design of the agent as an instruction set machine is described. This architecture is the basis of agent-based programming used for controlling low-level network resources in an open, extendable way.

3. A prototype of the system (SPIDER) was implemented. The implementa-
tion provided a number of insights into the paradigm, which helped in its final design.

4. A series of experiments on a 68-node transputer-based multicomputer have been performed and analyzed. The evaluation of the paradigm showed the openness and expressive power of the message-centered programming, the support of communication adaptivity and reliability, and the performance improvement in communication-intensive problems, such as broadcast or message combine, relative to similar node-centered solutions.

1.5 Thesis Outline

The thesis begins with the description of the paradigm in Chapter 2. It provides a brief background on communication features and a description of the message-centered programming. The SPIDER architecture is introduced in Chapter 3. It describes both the environment and its communication agents. Chapter 4 is dedicated to our prototype implementation. Its focus is on the significant issues valid for any implementation. The final major contribution of the thesis is the evaluation of the paradigm. We concentrate on expressivity, performance and adaptivity. The experiments are all contained in Chapter 5. Some of the ideas presented in this thesis intersect and interact with the ideas of others. For the sake of continuity, we present related work at the end in Chapter 6. This chapter is organized as brief description of other research, with references to SPIDER's features. Finally, conclusions and proposal of future explorations are presented in Chapter 7. An appendix provides details of the agent programs used in experiments.
Almasi and Gottlieb\cite{4} define a parallel computer as a “collection of processing elements that can communicate and cooperate to solve large problems fast”. Although there are many parallel computer designs, the scope of this thesis is restricted to multicomputers (Figure 2.1).

The network connects the processing elements so they can cooperate on solving a problem. The network topology is described by a graph whose edges are uni- or bi-directional communication links, and the nodes are crossbar switches or routing chips. The graph describes the logical structure of the interconnection network used for system-wide interprocess communication. The connections are point-to-point, thus, given two nodes that communicate, several other nodes can be involved in routing the message.
There are two essential network features that may or may not explicitly appear in the hardware interface presented to the communication software: network routing and flow control (Snyder[8], Dally[14], Seitz[60]).

2.1.1 Network Routing

Network routing is concerned with determining a path for a message from a source node to a destination node. The route of a message can be described by specifying an ordered list of nodes \( (n_{source}, ..., n_{dest}) \). A route is the set of resources (nodes, links and buffers) used to get from source to destination. The management of these resources greatly affects the efficiency and the correctness of message routing in the network. Routing techniques can be divided into two main classes:

- **oblivious routing**: the path taken by a message is solely determined by its source and destination addresses, regardless of the current state of the network. When all the messages with the same source and destination follow the same predetermined path through the network, the routing is called *deterministic*, otherwise *non-deterministic* (randomized).
• **adaptive routing**: path selection is dynamically determined based on the network state (topology, network traffic, priorities). It can be *minimal* adaptive, when the routing algorithms allow only minimal-length paths, or *non-minimal* (fully) adaptive, when algorithms allow a larger set of paths, possibly of infinite length.

There are two problems that are more difficult to address with adaptive routing, than with oblivious: *deadlock* and *livelock*. Deadlock is a condition that occurs in a communication network where no message can advance towards its destination because of a circular request to routing resources. Livelock also prevents messages from arriving at their destination. This can happen when the message is continually bypassed by other messages in the outgoing queues or if the message cycles in the network.

### 2.1.2 Flow Control

Flow control refers to the way in which messages are transmitted and buffered within the network (Dally[14]). It describes the packetization of messages, their break-up into bits or flits\(^1\) and how these bits are assigned buffers and links in the network. The two basic forms of flow control are:

- **circuit switching**: the entire path from source to destination is acquired first, then communication proceeds. IBM GF11, BBN Butterfly and earlier hypercube machines are circuit-switched.

- **packet switching** or **store-and-forward**: messages are broken up into fixed size packets, each packet is independently routed towards the destination. MPP, Intel

---

\(^1\) a flit is the smallest unit of information that a communication link can accept or refuse
iPSC, NCUBE and T800 transputer-based multicomputers use store-and-forward flow control.

Circuit switching and store-and-forward have evolved into wormhole routing and virtual cut-through, by dividing the packets into flits and sending them in a pipeline fashion. In wormhole routing the flits forming the message are transmitted behind the header of the message, as the path in the network is established. Virtual cut-through allocates buffers to arriving packets as in store-and-forward, but pipelines the transmission of flits as in wormhole. What distinguishes these flow control strategies is the packet collision mechanism (Dally[14], Seitz[60]).

Flow control is primarily a hardware feature. Starting with the Torus Routing Chip (Dally and Seitz[16]) routing has becoming a hardware feature as well (Aoyama[5], Dally[18], Flaig[24], Miller[47]), without, however, entirely replacing communication software (Clarke[12], Felperin[21], Talia[66]). This focus on VLSI integration of both routing and flow control has reduced their distinguishable characteristics, managing them as one feature (Aoki and Dally[15], Shumway[62]). Nevertheless, the established terminology is still being used: programmers deal with messages (application/transportation layer), the underlying communication mechanism breaks messages into packets, with each packet containing routing information (routing layer), and packets are composed of flits, which are the flow control units and have no routing information (flow control layer).

Programming the network involves interacting and managing the buffers, queues and links, and scheduling. In some networks it is also possible to control the topology by programming the crossbar switches (e.g. INMOS C004 and C104 transputer crossbars).
The above terminology is part of the node-centered approach (Section 1.3) to network control. In the message-centered paradigm we propose a 2-level terminology: the application or agent coordination layer and the network or agent layer. The agent coordination layer works with communication problems. A communication problem is a high-level encapsulation of one or more communication actions in the program. Examples of communication problems include: broadcast, scatter-gather, combine. The agent layer consists of communication agents which manage routing and flow control.

For pragmatic reasons (ease of design and hardware availability) agents are based on a point-to-point store-and-forward scheme. Store-and-forward offers the most support for programmability. This does not exclude other schemes, such as pipelined hardware flow control (wormhole routing) which offer less room for direct programmability. It is more convenient to view agents as the unit of control, independent on the underlying network hardware organization. Agents can be tailored to the hardware routing/flow control, and take advantage of it by providing a certain kind of traffic. On networks with hardware routing/flow-control one can implement store-and-forward by simulating point-to-point connections. Likewise, on packet-switched networks, wormhole routing can be implemented at the user level, with communication agents, just like any other communication problem (Logical C with virtual channels simulates T9000 wormhole routing on T800 packet-switched networks[42]).
2.2 Message-Centered Programming

SPIDER (Simple Programmable Interface Design for Efficient Routing) is a user-programmable routing kernel that supports a message-centered paradigm to programming communication by means of active communication agents (Section 1.3). Typically, in the networks encountered, routing and flow control are performed in a node-centered manner. Nodes contain the knowledge and control, whereas messages are passive entities communicated from one node to the next. SPIDER differs from the usual approach of organizing communication around processors/nodes, by organizing it around messages. It extends several ideas proposed in the literature: Reactive Programming (Seizovic[61]), Active Messages (Thorsten[67]) and microprogrammed routing nodes (Dolter[19]) by introducing network control from within the message. By control, we mean all the decisions related to message routing and flow control that are not controlled by the communication hardware.

Having the control in the message gives the user a new, flexible tool to program the communication, in other words, to program the network. In a node-controlled scenario the focus is on computation and high-level communication, ignoring communication details, while in the message-centered approach the focus is shifted to communication details, by ignoring computational aspects of the application.

Both the architecture and the associated programming language enforce a new way of organizing communication. This is done by programming it, by instructing the message what to do in the network in order to deliver the data. This approach gives the message a new status, from a passive shell containing data, to that of an intelligent agent, capable of traveling in the
interconnection network as it chooses. Although there is autonomy on individual basis, there is still some loose orchestrated coordination among agents, as their programs are written with the goal of solving communication tasks in the application. It is not only the message that is given a new status, the network itself is no longer a passive collection of wires, it is now a programmable device, and agents are its programs. Network programmability is divided into two levels, one that interfaces to the hardware, and one that interfaces to the user:

1. Application programmer’s view: the network is a programmable device, and agents are its programs. This involves two issues: high-level specification of agent task (“what”) and the high-level coordination of agents (“where” and “when”).

2. Communication programmer’s view: the agent is a programmable machine interacting with the external network hardware environment. This involves agent behavior implementation, according to the high-level specifications (“how”).

The focus of this thesis is on the communication programmer view. This involves architectural and agent programming issues. In addition, the thesis briefly touches on issues related to the application programmer’s view in Section 3.7, and indirectly in other chapters.

Programming the network is in fact programming communication agents according to a high-level specification of a communication problem. The instructions executed by agents on each node they visit manipulate low-level network resources (outgoing channels, buffer queues) to solve communication tasks. Decisions are decentralized, but the effects are system-
wide. Agents can learn, they can change internal state according to the state of the network and can respond with different behavior. As in real life, it is expected that the more powerful the smaller parts, the more potential the larger system has.

Since communication is separated from computation, agents view both the network and the application as a collection of external devices with which agents can interact through input/output operations. This system view allows for easier modification of the network. Adding new network elements is similar to adding external devices. The agent world is illustrated in Figure 2.2.

A communication agent is a message which visits the nodes to deliver data. Its movement in the network is entirely based on instructions “given” to the message. Conceptually, an agent, consists of two parts: a data part and a program part. The data part is the packet that must be delivered, and the program part is the routing instructions (i.e., a map). This is an overly simplistic view of the system but helps to explain the concept. SPIDER communication agents are similar to actors (Agha[2]) or to transportable agents (Kotz[35]) but they differ in the following two ways: SPIDER agents solve a communication problem, rather than a computational problem; communication agents navigate in the network by routing themselves, instead of being routed by the network.
A number of benefits are anticipated from programming the network with communication agents:

- separation of concerns: less error-prone and easier to program since the user can focus on communication independently from the computation of the application (Felten[22], Gelernter[27]).
- as shown in Chapter 5, the message centered paradigm can lead to a more natural description of communication.
- adaptive and opportunistic behavior can make good use of the current network state to improve performance and fault tolerance (Snyder[8], Ciampolini[11], Ngai[49])
- openness of the message centered approach addresses inter-operability issues across heterogenous networks and network fine-tuning to the application.
- the paradigm unifies communication issues into an architecture and a language, in the same way a Turing machine unifies computation into a machine, and its language. The paradigm is revolutionary, but it naturally synthesizes essential features of existing models (actors[2], network control[19], universal router[69]).

2.3 Example

An example best illustrates the concepts introduced and will make later development easier to understand. Consider the problem of broadcasting in a general network, using a node-centered
approach, and contrast it with the message centered approach. A simple, portable technique is to send the broadcast message from the source to each node by using a node-to-node sending primitive, as shown in Figure 2.3.

```plaintext
broadcast(message)
{
    if (current_node == source) then
        foreach node in the network
            if (node != current_node)
                send(node, message)
    else
        receive(message);
}
```

Figure 2.3: Example of node-centered broadcast

This algorithm makes poor use of network resources since the application has to sequentially execute all the send() calls. It is assumed that routing exists in the underlying message passing system. However, the user has no control over how routing is done, making it difficult to optimize communication.

Now consider a message-based solution to broadcasting using SPIDER. Let us use the idea of flooding the network, starting from a source node, until all the nodes have a copy of the message. In chapter 5 we will describe the SPIDER program that solves the broadcast problem and discuss its merits.

Application programmer’s perspective is: inject an agent into the network that visits all the nodes and delivers the message to each node. From the communication programmer, the task is to deliver the message to all agents as illustrated in Figure 2.4:
CHAPTER 2: Programming the Network

The source node activates an agent to transport the data, and the remaining nodes activate a message handler to extract the data from the agent once the data is downloaded on the node. The agent is injected into the communication medium (the network), the agent spawns copies of itself on every link that connects the current node to its neighbours, and marks the current node as being visited. When an agent (a clone of original agent) reaches a new node, it first checks whether the node had been visited. If so it dies, else the agent downloads its message, marks the node as visited\(^1\) and spawns a copy of itself on all available links. The agent behavior is simple, and ensures that no node downloads the message twice, and that all the nodes in the network (that are reachable from the source node) eventually receive the broadcast data. The broadcast routine is shown\(^2\) in Figure 2.4.

---

1. To unmark the node a "garbage collector" agent can be injected.
2. The actual SPIDER program is given in Appendix A and discussed in detail in Section 5.2.1
Notice that the agent is not part of the application, it is programmed independently, the application only needs a handle of the agent, to inject it in the network. As the agent moves from node to node its program is executed on each node the agent visits.

2.4 Summary

This chapter introduced a message-centered approach to programming the network in which messages act as programmable agents whose role is to reliably perform communication tasks.

A communication systems programmer is responsible for implementing agents, as demanded by the application. Agents can be constructed by the user or re-used from previous applications. Programming an agent defines its behavior; a mapping from percepts to actions. Agents perceive different facts about the environment and trigger corresponding actions (learning, updating network state, etc.). Actions are performed with the goal of solving a communication problem, such as node-to-node routing or group communication. Communication is programmed orthogonally to computation.

An example of using the new paradigm has been presented and compared to the node-centered approach. SPIDER has no built-in solutions for communication problems. It is the programmer who provides them.
CHAPTER 3

Architecture of SPIDER

When you use the word “agent” there is no implication that what the agent does is simple. They all suggest the agent is seen as having some specialized purpose.

Marvin Minsky

3.1 Overall Design

The design of SPIDER has been refined in the course of this research. There were a number of questions which motivated the design:

1. How can the network be programmed? What is message-centered programming? What is a communication agent?

2. What is an appropriate communication agent architecture? What language should agents use?

3. How general should agent behavior be, how much control should be delegated to agents?

4. How to build SPIDER agents and how to use them in applications?
5. What range of problems are to be solved by SPIDER agents and what are the trade-offs of a message-based approach?

6. What are the target machines and what hardware features are needed?

The first major design decision was to make SPIDER a low-level system; built directly on top of the hardware or a low-level communication micro-kernel. This strategy provides faster performance and gives the programmer better access to network resources. Communication systems or application programming interfaces like PVM (Beguelin[6]) or MPI (Gropp[28], MPI[45]) can use SPIDER agents to perform message delivery. Once written, agents can be reused as off-the-shelf communication objects in other applications, on possibly different architectures and networks. This is similar to the idea of libraries in structured programming, or class libraries in object-oriented programming.

Since SPIDER is a low level system supporting communication independently of computation, it does not enforce any particular naming upon the application and can accommodate other naming schemes. However, the drawback is the necessity of a run-time/static intermediate layer to shield SPIDER from the high-level message passing system. This layer has to manage mapping, naming, and linkage between communication agents and the application's calls to the system's communication primitives. However, most of these can statically be done, avoiding run-time overhead. An overview of the organization is shown in Figure 3.1.

The second important design choice was the view of the agent as a machine and the environment as a collection of external devices. The model isolates the agent by separating its design and use from the underlying hardware or software system. This ensures generality, portability,
and more flexibility in further development and modification. The alternative was an approach in which messages contained a sequence of commands but no internal state, thus, no learning.

The third essential decision made was the choice of the agent language and architecture. Three alternatives were identified:

1. interpreted language: each agent program contains instruction that will be interpreted at each node the agent visits. The instructions could be written in some high level language (Lisp, C, Tcl) or in a lower level language (assembly).

2. machine code: the agent program contains host processor object code that will be integrated into the computation on the node.

3. SPIDER machine code: design a special SPIDER chip to support the execution of communication agent.
Our design has taken the first approach: interpreted program, with a newly created SPIDER assembly language. This choice has been motivated by the following reasons:

- an interpreter is relatively easy to implement and modify. Constructing our own interpreter makes it easier to extract execution information, stopping the agent program execution on the current node and restarting it on another node, as required by the message-based programming.

- a new language is necessary in order to capture the communication semantics of agents. The new language is sufficiently small to keep efficiency high, and general enough to support the implementation of communication problems. Eventually, assembly code could be generated from a high-level language compiler.

- the agent construction had to be compact and required the functionality of an instruction set architecture.

The other alternatives did not offer a practical solution: for example, building a SPIDER chip is clearly outside the scope of a thesis. For performance reasons it would be interesting to program the agents in the native instruction set of the host processor. The system would be more difficult to debug and less portable.

---

1. An interpreter allows run-time checking on agent programs, increasing security. However, security was never an issue in this thesis.
3.2 Environment Model

As depicted in Figure 3.1, SPIDER is composed of two parts: a collection of agents and an environment. In this section the design of the environment and its relationship with agents are described.

The environment is the interface between agents and the hardware. It shields the agents from unneeded low-level details by abstracting the essential communication elements (links, queues) required by agents to correctly perform their tasks. As a layer, SPIDER provides portability and the necessary functionality as advertised in its interface to agents.

Rusell and Norvig [57] classify agent environments based on the following distinctions:

- **Accessibility**: can the agent access the complete state of the environment? SPIDER environment is distributed, thus an agent cannot comprehend the global environment at a given time. Agents do have access to the local state of the network.

- **Determinism**: is the next state of the environment completely determined by the current state and the actions selected by agents? The answer is viewpoint dependent: with respect to the environment it is deterministic, but, because of its distributed nature, it appears non-deterministic to the agents.

- **Episodic**: The SPIDER environment is not episodic, agent have continuity in their experience, and a sense of history.
• **Dynamism**: The SPIDER environment changes with the passage of time, and the agent does not look at the environment while it is deciding on an action; this makes the environment dynamic.

• **Continuity**: The SPIDER environment is a discrete one. Agents act in limited series of distinct steps, with clearly defined percepts and actions.

The SPIDER environment draws its architecture from the combination of the two systems it interfaces: the network hardware and the application software. Network hardware provides two structural elements: *nodes* (processor/switches/routers) and communication *links*. An application provides *addresses*, which are the ports where agents are injected in the network or leave the network (download the data). For generality it is important to keep the environment model independent of the application. Addresses capture the essential features of the contact points between network and application.

As a general rule, addresses represent the endpoint of an agent’s life, while nodes are the intermediate places where agents are active, while they are alive. Agents can move from an address to a node, from a node to another node, or from a node to an address. Since this movement can only take place in the network, there is no direct move from an address to another address. This view corresponds to an earlier pictorial description of the agent world (Figure 2.2). Notice that an agent cannot move between two entities without a physical connection (links or memory).

The concept that brings together nodes, links and addresses is the *channel*. A channel is the passage/gateway from a node to another, or from a node to an address. Physically, it is the
endpoint of a link on a node\textsuperscript{1}. Bidirectional links have two channels on each node, one input channel and one output channel. For routing purposes, all channels on the same node must have different names, but names can be reused on other nodes. It follows that channels at the opposite ends of a link could have different names. It is not hard to observe that, at any location, a channel uniquely identifies the entity (node or address) at the other endpoint. Hence, it suffices to only name the nodes and channels, links and addresses being implicitly understood.

As it is, the environment can be used to control some aspects of routing, but programming the network would be quite minimal. To provide more support for programming routing and flow control two other important features are introduced: channel queues and signposts.

Channel queues are bounded buffer queues of agents committed to leaving on the corresponding channel and are waiting for channel availability. This is a common feature of most networks, and where it does not exist, the queue size is assumed to be zero.

Signposts are similar to their analogue in transportation systems. Two classes of signposts are used by SPIDER: routing signs and marking boards. Routing signs act like routing tables, they offer guidance to agents, so that these can make a better routing decision. This is borrowed from the node-centered approach, to put some general knowledge at the nodes, to avoid its replication to the agents. Marking boards are places for agents to leave notes for others, as a means for agents to directly interact with the environment and other agents. A pictorial representation of the environment is shown in Figure 3.2.

\textsuperscript{1} A channel between a node and an address is the endpoint of a software-simulated link.
3.3 Environment and Agents

The high-level agent navigation/routing scenario is the following: the agent is injected in the network at an application address. The agent hops from node to node, along the links, by enqueueing itself on the corresponding channel queue. Eventually, the agent follows a channel to an address, where it will be handled by the application. From an agent's perspective, the channel is a gateway to the other end: after executing the queueing instruction, the agent
"wakes up" on the other node and continues execution from where it left off. Note that agent instructions (Section 3.5) do not distinguish between channels to nodes and channels to addresses, this is the programmer’s responsibility.

It is important to notice the finite nature of environment resources. Agents arriving on the node take up memory (Section 3.4), so only a limited number of agent can be accommodated on the node. Each channel queue can only buffer a fixed number of agents. High-level agent flow control is a bit more complex, and solutions adopted in transportation are helpful here. More specific details about the implementation are given in Chapter 4. For efficiency reasons, only one interpreter is allowed on the node, thus only one agent can be active at any given time. A node is viewed as the intersection of all the links which have channels on the node. Agents arrive asynchronously on these channels and access the interpreter on a first-come first-served basis. Agents can interact with the environment while they are being interpreted (i.e., active). In order for other agents to become active, the currently active agent must terminate or leave the node. We view channels as one-way gateways. Once an agent has enqueued on an outgoing channel it can only be dequeued by another agent or by the system, it cannot dequeue itself. If the corresponding queue is full, the agent blocks until it can proceed. While blocked, other agents are prevented from becoming active. These circumstances can lead to communication deadlock. Prevention and recovery are left to the programmer.

In addition to basic communication, SPIDER provides support for fault tolerance. When a fault is detected on an outgoing channel (its associated link, that is) the waiting agents are dequeued and re-interpreted (by re-injecting them back on the node). To the interpreter, this

---

1. Initially, another alternative was considered, which allowed concurrent executing agents, but it was discarded due to process management overhead on architectures that do not provide hardware support, as transputers do.
appears as if the agent came from the incoming channel of the failed link. The agent can detect the lack of progress and take appropriate recovery action. To avoid deadlock, the queue slot occupied by the just dequeued agent is not freed until the agent is successfully sent on the channel or back in the interpreter. Again, fault recovery is part of the agent behavior (programmer’s responsibility).

To summarize, an agent can be in any of the following external states:

- **ready**: waiting in the incoming channel to become active
- **active**: currently executing (being interpreted). It either selectively senses the environment, updates its internal state or its own body, changes the environment, cooperates with other agents, or prepares to move (by enqueueing on a channel queue).
- **waiting**: queued-up to leave on a channel.
- **blocked**: by definition active, however it is blocked, waiting for a free slot in a channel queue.

### 3.4 Agent Architecture

We have experimented with a number of agent architectures and based the design on an instruction set machine that incorporates elements from typical architectures, either to provide similar functionality (such as arithmetic operations) or as a metaphor for special communication control instructions (network instructions are seen as I/O operations). The choice of architecture, especially its instruction set is still an open issue.
A communication agent is an independent machine with its own state, memory and instruction set. First, we define a structure for the message that raises it to the level of a programmable machine. This construct captures both the machine architecture and the message passing semantics of the underlying system. The term "agent shell" is used to imply the structure imposed on the message, and sometimes we refer to an agent as simply a message, to emphasize the underlying entity.

The shell contains three main components, an agent description, the agent state, and the agent body (Figure 3.3). The agent description is the message header used by the SPIDER environment for agent management, the agent state consists of agent attributes and context, and the agent body is the core of the machine/agent.
Agent description

An agent co-exists with other agents in the SPIDER environment. The agent description is a means to delimit agent physical boundary and uniquely identify the agent. It consists of three parts:

- **agent identifier**: At any moment, all the outstanding communication problems employ one or more agents. Agents from different communication problems have different identifiers, while agents associated with the same communication problem share the same identifier and are called peer agents. Unlike plain messages, agents cooperate in solving a problem, so they must be aware of their peers. The common identifier is a simpler alternative to using a hierarchical group naming scheme. Although not necessary, we suggest another use for the identifier: as an escape to the system, for security or to bypass SPIDER.\(^1\)

- **agent size**: gives the agent memory requirements in the environment. The environment has limited resources (memory buffers). When these become exhausted no new agents can be created until other agents are destroyed or leave the node.

- **agent space or location**: the exact location of the agent in the environment. It should not be confused with the [node, channel] location. The [node, channel] pair is the high-level location, but the agent space is the current memory location where the underlying message is stored. Together with the agent size, the agent location is required by the environment in allocating space resources. In message passing terminology, each message has a length and a pointer to the memory the

---

1. In an earlier stage of development we have used this facility to implement a Print() command, and to also interface SPIDER with Logical C with virtual channels.
node has allocated for it. Notice the similarity with the “this” pointer in C++ objects.

Agent state

An agent has an internal state that is preserved as the agent travels in the network, for the entire lifetime of the agent, but it can change in response to the outside environment or to agent internal decisions. We refer to the agent state as *state registers*. The following state registers are used for maintaining the agent context:

- *program counter* (PC): a pointer to the current SPIDER instruction. It makes it possible to preempt agents. This register is automatically incremented after each instruction, or explicitly set as a result of a branch instruction.
- *condition flag* (CD): reflects the results of the previous operation. It is used for logical tests, becoming set/unset according to the truth value of an instruction.
- *base register* (BASE): Since multiple agents share the memory on a node, each agent is invoked as a process with its own address space\(^1\), which are pointed to by the base registers. This register is always initialized by the node when the agent arrives. It is possible for an agent to load a peer’s base register, so it can access the address space of a peer.
- *agent priority level* (PR): Routing processors such as NDF (Song\([63]\)) and CM-5 (Leiserson\([40]\)) provide more logical or physical networks, for data, for control and diagnosis, or for group communication. SPIDER generalizes the idea of disjoint networks and offers support for multiple classes of traffic by the means of

\(^1\) This is not entirely true in our implementation, but it is useful to have this image.
agent priorities, in a similar way transportation systems have emergency vehicles.

- **length (DLEN)** and **address of data (DADDR)**: these two registers keep track of the data carried by the agent. When agents download their data at the destination, a message handler uses the two data registers to extract the appropriate content of the delivered message. It is expected that most of the time these registers will not change, but it is possible for agents to drop parts of the data along the way, to add new data, or to modify it (encode, compress, etc.).

**Agent body**

This is analogous to computer memory, being used for storage and subsequent retrieval of data and instructions. It is the location where the agent behavior (its series of instructions) and the data to be transported are stored. Data to be carried to a destination is indistinguishable from the program unless the instructions explicitly manipulate the data registers, DLEN and DADDR. Addressing is at the byte level and at the word level, where a word is four bytes. There is an anomaly due to the instruction format (Section 3.5), which is half-word long, so instructions are always aligned at even addresses.

Relative to an agent there are two types of the memory: internal and external. The internal memory is the agent body, which the agent can directly access. The external memory is the memory available in the environment. This can be used in a similar way to virtual memory: when a memory fault occurs (agent tries to access an address outside its internal memory) a new chunk of external memory can be added to the internal memory ("page fault handling")\(^1\).
Changing the size of the memory is useful in collective communication, such as data gathering. An agent can start with little memory, just enough to hold its program, and then increase the size as data is gathered (from nodes or from peer agents). One can avoid changing the memory by giving the agent all the space required for the data, but this will consume network bandwidth and may reduce communication performance.

3.5 Instruction Set

Based on the experiments, the current design incorporates the necessary features to support the message-based programming paradigm. A number of constraints and trade-offs needed to be considered: bandwidth and latency vs. agent interpretation time, instruction format vs. instruction complexity, generality vs. efficiency, and ease of prototype implementation. The current instruction set adopts a pragmatic view, in which fast interpretation, uniformity and simplicity of instructions motivated the design. Both sensing and acting are implemented as instructions, so that agents interact with the environment only when required by the communication task, similar to polling external devices.

In the following, the architectural issues related to the instruction set are described, followed by the SPIDER instruction set and its associated assembly language.

Addressing

Addressing is tightly related to the instruction format, but since it was decided to use a small instruction length, it is expected that the operand fields are quite small. However, this should

1. Currently, the implementation does not support this, but the feature can be easily incorporated.
not preclude referencing a large range of memory locations, or using large numbers. Two addressing modes are supported by SPIDER: immediate, the value is the operand itself, and register, the value of operand is the value stored in the specified register.

**Data Types**

Instructions perform their operations on data types. SPIDER has the following data types: numbers: decimal, octal and hexadecimal integers, logical: bytes, words and half-words, and addresses: memory addresses, relative to the base register.

**Registers**

In addition to the state registers, SPIDER has ten general purpose registers. Unlike the state registers, the general purpose registers are not saved between agent's moves. They are mainly used for holding temporary variables or results, or operands too large to fit in the operand field of an instruction.

**Instruction format**

For pragmatic reasons and influenced by current RISC architectures, instructions were designed with a fixed length format, with the provision that this length be as small as possible. It was concluded that 16-bit instructions would suffice. As illustrated in Figure 3.4 SPIDER instructions can have 0, 1 or 2 operands. In addition to its normal usage, register r[0] is also a destination register, used to store the result of an instruction.
CHAPTER 3: Architecture of SPIDER

There are 38 SPIDER instructions, classified into six categories (Table 3.1): active, I/O or perception, arithmetic, logical, data transfer, and conditionals and transfer of control. The active and the I/O instructions are specific to SPIDER, the rest are similar to the instructions found in most RISC instruction sets.

The active instructions operate on the agent as a whole. They have a message passing semantics (OUT, FRK and RPL), or they suspend agent execution for a while (SLP), or destroy/terminate the agent (END). In addition to their communication semantics, FRK and RPL also spawn copies of the agent. Notice that RPL is not necessary, it follows from OUT and FRK (in conjunction with JMP).

The I/O instructions are the operations through which the agents interact with their world. They affect the environment, the agent state and internal variables. Most of these instructions query the environment for later action: ROUT queries the routing signposts, WHR gives the current [node, channel] agent location, MRK and TMRK interact with the marking boards, TLC checks the network topology, LLD tests and returns the size of the channel queues. A spe-
sial instruction, LKP and its variation LKPD, allow agent rendez-vous with peers. This is only possible when the peer is already in a channel queue. In addition to finding out whether the peer exists, if found, LKPD also dequeues it (see the combine agent in Section 5.2.1)

Table 3.1: SPIDER Instruction Set

<table>
<thead>
<tr>
<th>Class</th>
<th>Operation mnemonic</th>
<th>Format</th>
<th>Instruction meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>END</td>
<td>-</td>
<td>Terminate current agent; release resources</td>
</tr>
<tr>
<td></td>
<td>OUT</td>
<td>X</td>
<td>Leave on channel X</td>
</tr>
<tr>
<td></td>
<td>FRK</td>
<td>X</td>
<td>Fork off on channel X</td>
</tr>
<tr>
<td></td>
<td>FRK</td>
<td>X</td>
<td>Replicate on channel X</td>
</tr>
<tr>
<td></td>
<td>SLP</td>
<td>X</td>
<td>Sleep for X microseconds</td>
</tr>
<tr>
<td>I/O</td>
<td>ROUT</td>
<td>X</td>
<td>Read routing table. r[0] ← route to node X</td>
</tr>
<tr>
<td></td>
<td>WHR</td>
<td>R₁, R₂</td>
<td>Get current location, R₁ ← node, R₂← channel</td>
</tr>
<tr>
<td></td>
<td>MRK</td>
<td>X</td>
<td>Write a value X on a marking board on current node</td>
</tr>
<tr>
<td></td>
<td>TMRK</td>
<td>X</td>
<td>Test if mark X exists on current node. Set the condition flag</td>
</tr>
<tr>
<td></td>
<td>TLC</td>
<td>X</td>
<td>Test if channel X is connected/valid. Set the condition flag</td>
</tr>
<tr>
<td></td>
<td>LLD</td>
<td>X</td>
<td>r[0] ← load on channel X (# of agents queued-up)</td>
</tr>
<tr>
<td></td>
<td>LKP</td>
<td>X</td>
<td>Lookup peer agent on channel X. If found, set the condition flag, and r[0] ← peer's base register</td>
</tr>
<tr>
<td></td>
<td>LKPD</td>
<td>X</td>
<td>Same, but dequeue the peer if found</td>
</tr>
<tr>
<td>Data transfer</td>
<td>MOV</td>
<td>R₁, R₂</td>
<td>R₁ ← R₂</td>
</tr>
<tr>
<td></td>
<td>MOVI</td>
<td>I</td>
<td>r[0] ← I</td>
</tr>
<tr>
<td></td>
<td>LDW</td>
<td>X</td>
<td>Load word: r[0] ← Mem[X]²</td>
</tr>
<tr>
<td></td>
<td>LDB</td>
<td>X</td>
<td>Same as above, but load a byte</td>
</tr>
<tr>
<td></td>
<td>STW</td>
<td>X</td>
<td>Store a word: Mem[X] ← r[0]</td>
</tr>
<tr>
<td></td>
<td>STB</td>
<td>X</td>
<td>Same as above, but only store the first byte</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>ADD, SUB, MLT, DIV</td>
<td>R₁, X₂</td>
<td>Addition, subtraction, multiplication, division r[0] ← R₁ op X₂, where op is +, -, *, /</td>
</tr>
<tr>
<td></td>
<td>MOD</td>
<td>X₁, X₂</td>
<td>Modulo division: r[0] = X₁ mod X₂</td>
</tr>
<tr>
<td></td>
<td>RND</td>
<td>X</td>
<td>r[0] = random number mod X</td>
</tr>
</tbody>
</table>
Table 3.1: SPIDER Instruction Set (Continued)

<table>
<thead>
<tr>
<th>Class</th>
<th>Operation mnemonic</th>
<th>Format¹</th>
<th>Instruction meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical (bit operations)</td>
<td>AND, OR, XOR</td>
<td>R₁, X₂</td>
<td>Logical (bitwise): r[0] ← X₁ op X₂ where op is &amp; , l , ^</td>
</tr>
<tr>
<td></td>
<td>NOT</td>
<td>R</td>
<td>R ← ~R. Flip all bits</td>
</tr>
<tr>
<td></td>
<td>SHL, SHR</td>
<td>R₁, X₂</td>
<td>R₁ ← R₁ shifted to the left (right) with X₂ positions</td>
</tr>
<tr>
<td>Conditionals and transfer of control</td>
<td>TEQ, TGT</td>
<td>X₁, X₂</td>
<td>Test X₁ == X₂ and X₁ &gt; X₂. Set the condition flag if true.</td>
</tr>
<tr>
<td></td>
<td>JMP</td>
<td>X</td>
<td>Jump to X. Address is relative to the base register</td>
</tr>
<tr>
<td></td>
<td>BRT, BRF</td>
<td>X</td>
<td>Conditional jump to X. Address is relative to the base register. Branch only if the condition flag is set (unset)</td>
</tr>
<tr>
<td></td>
<td>EXE</td>
<td>X</td>
<td>Jump to subroutine at address X</td>
</tr>
<tr>
<td></td>
<td>RET</td>
<td>-</td>
<td>Return from subroutine to the calling point</td>
</tr>
</tbody>
</table>

1. The format refers to the addressing mode, as well as the operands.
R = register mode. I = immediate mode, X = either immediate or register
When used as values, Val(X) = X when X is immediate, r[X] when register

2. Mem[A] is the memory content at address A. A is relative to the base register

3.6 Assembly Language

The SPIDER assembly language follows the general rules found in most assembly languages. It consists of lines of code, where the symbols on each line follow a precise syntax:

- on each line, all the symbols after a "#" are considered comments and ignored by the assembler
- if a line starts with an assembly language instruction that line is considered an instruction line. The assembly instructions are of two types: pseudo-instructions (Table 3.2), that instruct the assembler to perform certain preprocessing, and
actual instructions (Table 3.1), which are the mnemonics of the SPIDER instruction set and are executed as part of the agent program.

The current assembler is implemented in the AWK scripting language (Aho et al.[3]). In addition to creating the machine language, it also completes most of the agent’s shell fields. This is accomplished by having a small module that is user customizable, so for different kinds of agents, the user can change the default settings of the agent structure. The assembler assumes that all the names of the form \(r_i\) and \(r_{\text{state}}\) are registers, where \(i\) is the register number (e.g. \(r_1\)) and \(state\) is one of the agent state registers (e.g. \(r_{\text{BASE}}\)).

<table>
<thead>
<tr>
<th>Class</th>
<th>Operation mnemonic</th>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>LABEL</td>
<td>name</td>
<td>Specifies symbolic addresses</td>
</tr>
<tr>
<td></td>
<td>DEF</td>
<td>name, value</td>
<td>Macro definition. Assigns value to name</td>
</tr>
<tr>
<td></td>
<td>DATA</td>
<td>byte1, byte2</td>
<td>Initialize the half-word at the current location</td>
</tr>
</tbody>
</table>

### 3.7 Agents and the Real World

This section touches briefly on the interface between SPIDER and programmers. From the communication programmer’s perspective, programming the network means creating SPIDER communication agents, and from the application programmer’s perspective it means using them in a coordinated way. The programming framework is depicted in Figure 3.5:

1. The user provides a parallel program that will be pre-processed by the parallel programming environment. Pre-processing may involve process to processor mapping, naming of entities, etc. Preprocessing will identify
the communication parts of the program (easy if something like MPI is used, very difficult otherwise, without programmer’s intervention), analyze them, and produce a description of them in the form of communication problems. The description of a communication problem contains application and system parameters the agent needs. Using an OOP terminology, this is the constructor interface of an agent.

Figure 3.5: Programming with SPIDER
2. A library of agents (an agency) is then searched for agents matching the required specifications. If no agent is found, the communication programmer must implement the agent with the specified behavior. The agent is then compiled/assembled into a representation that can later be used by the application. Note that unresolved references may exist in this representation; in fact, it is almost guaranteed they will, since the data to be carried is usually known only at run-time.

3. Finally, the agents are linked with the computation module (augmented with appropriate agent-handlers) and with the SPIDER kernel library.

3.8 Summary

In this chapter we have described the architecture of the SPIDER model. This architecture is necessary in order to be able to program the communication agents. The two main parts of SPIDER are the environment and the agents. An agent is a programmable machine, with an instruction set, memory and state. The network hardware is the environment in which these machines are active. The environment is a simple model, similar to a transportation system, with nodes, channels, addresses and routing signs. Programming the network involves the interaction of agents with the environment. Agents access low-level network services (channels, queues, etc.) to control communication for application-tuned routing.

In summary, these relationships are:

\[ \text{Agent} = \text{Program} + \text{Architecture} \]

\[ \text{SPIDER} = \text{Agents} + \text{Environment}. \]
Implementation of a programmable communication kernel for a multicomputer involves a wide range of issues, some quite challenging. This chapter presents some of the issues that are significant to any implementation, and describes a few of the details of our SPIDER prototype.

The prototype described here is produced as proof-of-concept demonstration of SPIDER’s feasibility, and to explore the paradigm of “programming the network”. The programming goal of the implementation was to conduct experiments and identify the benefits and limitations of the proposed paradigm. Experiments are described in the following chapter, here the focus is on design and implementation of system features. The implementation we discuss in this chapter is specific to our transputer-based multicomputer, but its high-level design is general enough to ensure SPIDER’s portability to any similar multicomputer system.
4.1 Environment

The experimental testbed in this thesis was the transputer-based multicomputer in the Department of Computer Science at the University of British Columbia.

4.1.1 Hardware

Our multicomputer consists of 68 T800 transputer nodes and 10 crossbar switches. The system is hosted by a Sun-4 workstation via an Sbus card, with 4 ports that connect the system to the host. Most of the nodes have 1 MB of external memory, and a few have 2 MB.

The INMOS T800 transputer is a 32 bit microprocessor with a 64 bit on-chip floating point unit. A T800 running at 20MHz has a sustained processing rate of 10 MIPS and 1.5 Mflops. It has 4 KB of on-chip RAM and four bit-serial communication links. These communication links allow networks of transputer nodes to be constructed by direct point-to-point connections with no external logic. Each link runs at an operating speed of 20 Mbits/sec and can transfer data bidirectionally at up to 2.35 MB/sec. The design of the SPIDER kernel assumes a reliable point-to-point communication mechanism.

The system consists of several concurrent processes. Support for concurrent processes could be available either in hardware (as in Transputers) or in a low-level communication system (like Logical C). The T800 processor has a microcoded scheduler that enables any number of concurrent processes to be executed together. Processes can execute at one of two priority levels: high and low. A high priority process, once selected, runs until it has to wait for a communication, a timer input or until completion. If no process with a high priority is ready to pro-
ceed, then one of the ready low priority processes is selected. Low priority processes are time sliced every millisecond. A low priority process is only permitted to run for a maximum of two time slices before the processor deschedules it at the next descheduling point.

Process context switch times are less than 1 µs, as little state needs to be saved. The transputer has a 32 bit timer clock which is accessible only to high priority processes and is incremented every microsecond; this clock was used for obtaining accurate measurements for our experiments, together with an external synchronization device.

Each C004 crossbar switch has 32 data links and a control link. These are programmable through the control link and each can have 16 bidirectional connections. Thus, the system is reconfigurable and an appropriate interconnection network for an application can be chosen. As the system is not fully connected, there are some restrictions on the possible configurations. For our experiments we statically configure the system into an appropriate interconnection network in software, although SPIDER could, in theory, work with dynamically configured networks, this was not investigated.

4.1.2 Software

Although the implementation discussed in this chapter is specific to our transputer-based multicomputer, its high-level design could be ported to similar multicomputer systems. There were two implementation alternatives: using a concurrency library only, or building the system directly on top of the network hardware. For pragmatic reasons, availability and low overhead, the prototype was implemented using Logical C (Logical Systems[42]).
Logical C is a runtime system, which provides a C interface to transputers. The Logical C concurrency library provides wrappers for the low-level in/out instructions. We have modified Logical C transputer system files, as well as the program loader on the host. These changes allow us to control the initialization of SPIDER kernel processes and to set up SPIDER environment data structures. Moreover, all the internode communication is handled and managed by SPIDER, with the possibility to bypass it, should the necessity arise. In the early stages of system development the virtual channel version of Logical C was used, and the two worked together. Basically, SPIDER trapped all the messages, and then it passed control to the virtual channel system, regaining control after the virtual channel protocol ended.

The advantage of using Logical C is more than just ease of implementation, but it is also its increased portability. Moving to another architecture would only involve the implementation of some system library calls (such as the message passing primitives) and possibly some code twiddling.

4.2 Kernel Design and Implementation

SPIDER is designed as a small kernel running on each transputer node, managing communication agents injected into the network by user processes. The kernel is loaded during node bootstrapping, and it takes control of the communication before the application\(^1\) starts.

---

1. In this chapter an application is anything running on top of SPIDER, not necessarily the user application. A run-time system implementing MPI using SPIDER qualifies for this name.
Our system view of SPIDER is that of a communication kernel, whose objective is to provide an execution environment for SPIDER agents. A copy of the kernel must be present on each node in the system. The kernel supplies the following three services:

- interpretation of agent programs: provides a thread of execution for each agent program. For efficiency reasons there is no multi-agent execution, that is, an agent is not preempted by other agents; once an agent starts execution it continues execution until completion.

- resource management on the node, and communication control. The kernel allocates buffers to agents (an agent requires memory to store its body), protects shared resources (like the execution environment), and manages incoming and outgoing agents (using buffer queues).

- support for the SPIDER instruction set: use appropriate facilities of the underlying system or from its own resource management modules (load on links, etc.) to implement execution of instructions.

These three services can be provided by different designs; here we propose a simplified version of a minimal design. The kernel consists of four basic modules on each node, instantiated as processes (Figure 4.1):

**Channel input module:** link_in

For each link, this process is associated with the corresponding incoming channel. Link_in is responsible for receiving agents from the neighbouring node and allocating them the necessary memory. Each agent is then passed to the interpreter module for program execution. If the
application injects an agent, no link_in process exists for that channel, the agent is directly passed to the interpreter on that node.

Channel output module: link_out

On each link this process is associated with the corresponding outgoing channel. Link_out detects when the link is available, dequeues an agent and sends it to the neighbour node. In addition, link_out performs some error handling activities (Section 3.3). All agent resources (memory) are returned to the buffer manager after the agent is sent. The queue of
agents is only decremented after the dequeued agent is sent on the link or successfully re-injected in the interpreter. This strategy prevents extra buffering for agents waiting for access to the interpreter without being in a queue (only queues are controlled by the buffer manager).

The input/output processes guarding the hard links increase the efficiency by exploiting the inherent parallelism within a node (DMA independent channels).

**Interpreter module:** interpeter

The interpreter process runs on each node. It receives agents from the link_in processes and interprets their instructions: it reads instructions from the agent body as directed by the program counter and the base register, decodes them, and then it carries out the operations. Operations may involve simple processing, updating environment or agent state, or interacting with other modules to obtain information needed by the agent's "sensors" (Section 3.5). When agents leave the node, the interpreter enqueues them on appropriate channel queues. If the instruction was an OUT, the current execution ends, but if it was FRK or RPL only a copy of the agent is enqueued, the current active agent continues on.

**Buffer manager module:** scheduler

The buffer manager's function is to control the allocation of node resources, memory in particular, to ensure a fair sharing of the environment by the agents. Initially the scheduler was implemented as a process, but later its functionality has been delegated to the interpreter and to the link_out processes, for more efficiency and for easier updating. The scheduler module now consists of the enqueuing and dequeueing routines, and a few other small management utilities. Since the channel queues are finite, they are managed as a reader/writer
with a finite buffer concurrency model, so the current executing agent blocks when the channel
to leave on is full, and it proceeds only after another agent from that channel has left the node.
The queueing policy is that of a priority queue, based on agent priority level (Section 3.4).

In addition to the system processes we just described, there is a special process used to
download the data to the application. Unlike the other processes, the downloader is known
outside SPIDER.

Downloader

The downloader process replace the link_in and the link_out processes on channels
to addresses. The downloader is started for each communication problem on each node requir­
ing data. A message handler is passed as an argument; its role is to manage the data brought by
agents, as directed by the application. The message handler is invoked on each agent arriving
on the address channel, thus, it needs to return TRUE or FALSE, according to whether all
expected agents arrived or not, so that the downloader terminates or continues waiting.
Before terminating, downloader signals the application the receive is done. The application
can receive the data or continue processing, thus multiple downloaders can be active at the
same time. Note the similarity with a dispatcher using Active Messages. This mechanism was
chosen for its generality and simplicity, although more experimentation is required to prove its
efficiency or lack, thereof.

The interpreter and the hardware link processes are started at initialization time,
whereas the downloader is created and destroyed dynamically for each new address.
These processes communicate with each other over soft transputer links. All link_in processes share a common soft link to the interpreter, which is protected by a semaphore. This ensures a First-In-First-Served processing of arriving agents. To avoid copying of packets between kernel processes only the header (9 words) is passed in all on-chip communication. The agent body is accessible from the information in the header (Figure 4.2). Memory occupied by agents is released when the agent leaves the node or under the agent’s control.

```
typedef struct {
    unsigned int id; /* Agent identifier */
    unsigned int len; /* Size of agent (prog+data) */
    unsigned char* mem; /* Agent body address */
} _S_Header;

typedef unsigned int _S_Register;
typedef struct {
    _S_Register pc; /* Program counter */
    _S_Register cd; /* Condition flag */
    _S_Register pr; /* Agent priority */
    _S_Register base; /* Base register */
    _S_Register dlen; /* Length of data to download */
    _S_Register daddr; /* Data offset */
} _S_State;

typedef struct {
    _S_Header hdr; /* Agent description */
    _S_State state; /* Agent state */
} _S_Agent;
/* There is no agent body, it is implicitly malloc()-ed */
```

Figure 4.2: Main agent data structure
4.3 System Tools

The last part of our prototype addresses programming issues, the tools needed to construct and use communication agents. To keep the implementation manageable, without sacrificing its generality, we make use of existing development tools such as C cross-compilers, link editors, and the AWK programming language.

The view of an agent at the lowest programming level is a structure \(_S_{Agent}\) (Figure 4.2) with the agent body replaced by a pointer to some memory location on the node the agent resides. In order to use an agent in an application, the \(_S_{Agent}\) structure must be filled in, and the appropriate memory buffer initialized with the agent program and the data to be transported. One could write an agent for each communication problem in every application, but this is impractical. A better approach is to create a representation of agents, a template, from which agents can be created at run-time. The assembler converts SPIDER agent programs into an intermediate representation in Logical C. For each agent we can create an agent C file, which can be compiled and put in a library. The applications can link these files, by using the functional interface to an agent. Let us consider an example of this process, to better understand the implementation ideas.

Example: send/receive SPIDER agent

Consider the node-to-node routing problem, where a node sends a message to another node. Let us start from the bottom with the implementation of the agent.
First, an agent assembly program is written, as shown in Figure 4.3. Note the two names that are not known inside the agent, id and dest. They parameterize the agent; agents with similar behavior can be instantiated from this template by supplying values for id and dest at runtime.

```
# dest = destination node
# id   = communication problem identifier, also used as
#        download channel name

LABEL start
  MOVI  dest  # Destination node
  MOV   r_4   r_0   # If dest>15 a register is needed
  WHR   r_1   r_2   # Get location
  TEQ   r_1   r_4   # Are we done?
  BRT   end    # If yes, then stop
             # else
  ROUT  r_4   # Get route to dest
  OUT   r_0   # Move further
  JMP   start # On next node start again

LABEL end
  OUT   id    # Download message
```

**Figure 4.3:** Routing agent program: agent.prog

The agent program is then processed by the assembler, and a piece of C code, agent.instr (Figure 4.4) is produced. The SPIDER assembler has a user customizable module to automatically fill in the agent fields with the necessary information. For each instruction, the helping routine do_instr() writes the corresponding instruction fields (opcode and operands) to the appropriate agent memory location. As shown next, all variables are bound at linking time, and the agent is constructed at run time.

The integration of agent with applications is done through a wrapping functional interface. For our example (and for most cases), three such functions must be written (Figure 4.5):
CHAPTER 4: Implementation

/* _S_Agent a; */

a.hdr.id = id;
a.hdr.len = send_size + 20;
a.hdr.mem = (UCHAR*)smalloc(a.hdr.len);

a.state.dlen = send_size;
a.state.daddr = 20; /* Offset */
a.state.pc = 0;
a.state.cd = 0;
a.state.pr = priority;
a.state.base = (UINT)a.hdr.mem;
bcopy(send_addr, a.hdr.mem + a.state.daddr, send_size);
do_instr1(_S_ARITH, MOVI, IMM_BIT, dest, a.state.base+0);
do_instr2(_S_ARITH, MOV, REG_BIT, 4, REG_BIT, 0, a.state.base+2);
do_instr2(_S_IO, WHR, REG_BIT, 1, REG_BIT, 2, a.state.base+4);
do_instr2(_S_TEST, TEQ, REG_BIT, 1, REG_BIT, 4, a.state.base+6);
do_instr1(_S_BRANCH, BRT, IMM_BIT, 8, a.state.base+8);
do_instr1(_S_IO, ROUT, REG_BIT, 4, a.state.base+10);
do_instr1(_S_ACTIVE, OUT, REG_BIT, 0, a.state.base+12);
do_instr1(_S_BRANCH, JMP, IMM_BIT, 0, a.state.base+14);
do_instr1(_S_ACTIVE, OUT, IMM_BIT, id, a.state.base+16);

Figure 4.4: Assembler output: agent.insr

1. the message handler function to be passed to the downloader, 
   msg_handler()

2. the function that injects the agent into the network, send()

3. and the function that receives the data, recv() (this is a blocking receive).

Notice the extra parameters to the send/receive functions: id is the communication problem identifier and priority is the agent priority level. Generally, these two parameters are obtained from a run-time system (see Sections 3.1 and 3.7). The output from the assembler is included in the file agent.c.
CHAPTER 4: Implementation

```c
msg_handler( void *arg, _S_Agent *a )
{
    struct {int len; char *buf;} *p = arg;
    bcopy( a->hdr.mem + a->state.daddr, p->buf, p->len);
    return TRUE;
}

send(int dest, int send_size, void *send_addr, int id, int priority)
{
    _S_Agent a;
    #include "agent.instr"    /* Assembler generates this file */
    inject_agent(&a);
}

recv(int recv_len, char *recv_addr, int id)
{
    int signal;
    Channel *chan = ChanAlloc();
    struct {int len; char *buf;} recv_arg = {recv_len, recv_buf};

    start_downloader(id, msg_handler, &recv_arg, chan);
    ChanIn(chan, &signal, sizeof(int));
}
```

Figure 4.5: Routing agent functional interface: agent.c

4.4 Summary

In this chapter a prototype implementation of SPIDER was presented. It consists of cooperative kernel processes on each node of a transputer-based multicomputer, and software tools for agent implementation. The processes, based on Logical C concurrency library, manage the communication links and the interpretation of agents instructions. A SPIDER assembler, implemented in AWK, was used to translate agent programs into agent C templates to be linked with the applications. The application instantiates an agent by providing it with the required parameters.
CHAPTER 5

Evaluation of SPIDER

SPIDER introduced a new way of performing communication, by means of communication agents to program the network. Feasibility of such a system is clear from the previous chapter; here we provide experimental evidence for the paradigm’s benefits and limitations.

This chapter is organized as a combination of experiments and analysis. The experiments include a series of collective and individual communication problems that illustrate the concepts and provide a testbed for analyzing SPIDER’s features.

5.1 Assessment Criteria

Initially, the goal of this project was to devise a better routing tool. In the course of the research, SPIDER developed into a programming paradigm rather than simply a routing tool. Ciampo-
lini[11] and Talia[66] identify the following requirements for a routing tool: non-intrusion, topology independence, fairness, traffic independence, and adaptivity. The intended use of a communication system determines the main criteria for evaluating the routing techniques and its user interface. Generally, the following are sought:

- efficiency in supporting programming models (flexibility)
- efficiency in mapping to parallel machines (portability)
- generality of the routing system (universal router)
- practicality of the system (easy to use and good performance).

Our assessment approach combines these requirements into a set of four evaluation criteria which capture the essential features of SPIDER: expressivity, performance, adaptivity and fault tolerance, openness and generality.

1. *Expressivity* refers to being able to specify communication tasks by means of agent programs, as well as ease of use.

2. *Performance* is a measure of how fast communication tasks are accomplished. The focus is on the overall performance of the application and on the trade-off between raw speed and flexibility.

3. *Adaptivity* and *fault tolerance* express the capability of adapting to traffic conditions and the use of all links to distribute messages over available paths. It also includes topology independence and protocols to prevent deadlock and livelock.

4. *Openness* and *generality* address the issues of user control, portability
and universality of the paradigm. Agents should be able to move across different architectures and still be able to route themselves and solve their communication tasks. The system should not enforce the communication strategies, but support their co-existence, by letting the agents use appropriate routing schemes. The message-based programming mechanism should be universal and not dependent on a particular hardware.

The chapter concludes with an application, Cholesky Factorization, that brings together these criteria, by comparing its SPIDER implementation with one using Logical C [42].

5.2 Expressivity and Openness

To show the expressive power of the new paradigm, several representative collective and individual communication problems were implemented. Having created communication agents for these problems, one can employ them as underlying support for message passing programming interfaces such as MPI (Gropp[28], Message Passing Interface Forum[45]). In principle, all that is needed to interface SPIDER with a system like MPI is a naming tool, which maps MPI entities (groups, communicators) into SPIDER environment entities (nodes, channels, agent identifiers).

5.2.1 Collective Communication

There is a large theoretical literature on optimal algorithms for collective communication under various constraints (Leighton [38], [39], Johnsson[34]). Traditionally, algorithms for the routing of messages have focused on delivering the message, regardless of the activity of other
messages. SPIDER offers the means to coordinate communication, to devise better routing schemes that fit the communication patterns of the application.

Typical communication problems involving cooperation are: broadcast, multicast, scatter, gather, combine, scan, to name just the most common ones. To investigate performance and usability, two problems were explored: broadcast and combine.

5.2.1.1 Broadcast

Broadcast is one of the most fundamental communication problems in parallel systems and has been intensively studied in the literature. Some systems implement it directly in the hardware, other systems like Tiny[12] and Trollius[52] provide software support for it. However, many systems do not provide any built-in support, in which case it is accomplished by node-to-node send/receive. Finally, there are also less portable but more efficient algorithms which create a broadcast tree tailored to the application and architecture.

The message-centered solution is based on a simple flood-filling algorithm that is both efficient and portable (Figure 5.1). Basically, a broadcast communication agent is injected in the network and on each node the agent downloads its data (line 18), marks the node as visited (lines 10-13), and then replicates itself on links connected to unvisited nodes (lines 20-28 and 34-44). To download data, the agent forks off a copy of itself on the address channel (line 18), created by the application for this purpose. It is the responsibility of the application to create the download handler to extract the data from the agent.

The broadcast agent has the following properties:

- specifying agent actions is based on a simple flood-filling mechanism;
# Agent visits all the nodes, by replicating on all available links.
# Mark each visited node to avoid duplication of bcast message.
# Notice that the bcast message is not sent to the source node.

```
1 DEF bcastnode source  # Broadcast node
2 DEF bcastlink id     # channel to the application
3
4 MOVI bcastnode  # When node > 15 use register to encode value
5 MOV r_3 r_0
6 TMRK id  # Has the node been visited?
7 BRT end  # If yes, die
8 # else
9 MKR id  # Mark the node as visited
10 WHR r_1 r_2  # Current node and link
11 TEQ r_1 r_3  # Is this the source node?
12 BRT start  # If yes, replicate on links
13 # else
14 FRK bcastlink  # Download data on soft link
15 # and continue on
16 LABEL start  # Replicate on all available links
17 MOV r_0  # Pass link number to bcast routine
18 EXE bcast  # Broadcast on link 0
19 MOV 1
20 EXE bcast  # Broadcast on link 1
21 MOV 2
22 EXE bcast  # Broadcast on link 2
23 MOV 3
24 EXE bcast  # Broadcast on link 3
25
26 LABEL end  # Terminate
27 END 1  # Terminate and release resources
28
29 ## This procedure replicates the agents on channel r_0
30 LABEL bcast
31 TLC r_0  # Is the link connected?
32 BRF return  # If no, return
33 # else
34 TEQ r_0 r_2  # Have I arrived on this link?
35 BRT return  # If yes, return
36 # else
37 RPL r_0  # Replicate on this link
38
39 LABEL return  # Return from procedure
40 RET

Figure 5.1: Broadcast agent
```
• the control is in the agent, nodes (the user program on the nodes) do not have to know about how broadcasting is performed. With respect to the application it is like a hardware broadcast;
• the agent is independent from the network topology;
• the agent is independent from the application but requires a set of parameters (source and id). By parameterizing the agent it is more general; for example, the same agent was used in different applications (performance testing and Cholesky Factorization)
• programming is done at a low-level, at the network interface level, the agent makes explicit use of the SPIDER system model.

In Section 5.3.1, we return to this example for performance analysis.

5.2.1.2 Combine

Combine, also known as prefix summation, is a communication problem in which a node gathers data from others and merges it. More formally, consider a binary associative operator $\oplus$ and a set of values $y_i$ at each node $n_i$, $i = 1, 2, ..., N$. The goal is to compute the value

$$x = y_1 \oplus y_2 \oplus ... \oplus y_N.$$  

The node receiving the result $x$ is called the root node. A simple node-centered algorithm is for each node to directly send its data $y_i$ to the root node for merging. However, due to a limited fan-in at the root (small network degree compared to the network size), the network becomes congested towards the root, and the computation load (for merging) occurs at the root, without parallelism. A second algorithm is tree-based, by combining messages $y_i$ along a tree, as they
move towards the root. Each node synchronizes with its children, combines their data with its own, and finally forwards the result to its parent. A combine tree is not portable (it requires a combine tree for each network/application) and it is highly synchronous. An asynchronous version is possible, but it requires more programming effort.

The solution in SPIDER is simple and elegant. As in broadcast, it is application independent and the algorithm is easy to express as an agent behavior. For comparison purposes, two agents were created: one for the combine along the tree and one for combining at the root only. The combine operation $\odot$ is bitwise OR, and $y_i = i$.

To do a combine, each node injects a combine agent in the network and the root starts a message handler to extract the combined data. The interesting behavior is that of the tree-combine agents which is based on a preemptive opportunistic strategy (Figure 5.2).

In the case of a tree-combine there is the possibility to take advantage of rendez-vous that are likely to occur. Each agent moves towards the root (line 44); when an agent arrives on a node, it first checks the queue of agents waiting to leave the node on the same channel (lines 13-15). If one of those agents is a peer agent, it will dequeue that agent (line 15), combine the peer's data with its own data (lines 28-36), and then leave the node (line 44). The agent preempted is destroyed (lines 39-41). In this way, whenever the opportunity arises, messages/agents are combined, taking advantage of the waiting time in the queues, reducing the number of messages arriving at the root, without incurring the overhead of synchronization. There is one more detail to be taken care of: since fewer agent arrive at the root, how does the root know when to extract the combined data? The solution we adopted is that each agent keeps track of
CHAPTER 5: Evaluation of SPIDER

1 # This agent moves up the combine tree and, when it finds a peer
2 # it removes it from the queue, combines its data with peer's
3 # and enqueues itself in the queue (Full listing in Appendix A)
4
5 LABEL counter  # Counts how many agent are merged
6 DATA 0x1 0x0  # Initially, count = 1
7 ...
8 LABEL start  # Beginning of program
9 WHR r_2 r_3  # Get agent location: node/link
10 # If reached the root, prepare to download, else look for peer
11 ...
12 LABEL lookup  # Look for peers on this node
13 ROUT rootnode  # Get link to follow to root
14 LKPD r_l  # Lookup peer agent.
15 BRF out  # No peers, move w/o combining
16 # else, merge data with peer's
17 ...
18 # Update agent count. Notice the use of two address spaces
19 LDW counter  # Get the combine counter of agent
20 MOV r_BASE r_3  # Move to peer's address space
21 LDW counter  # Get combine counter of peer
22 ADD r_4 r_0  # Add combine counters.
23 MOV r_BASE r_2  # Restore the base register
24 STW counter  # Update combine counter
25
26 # Do the actual combine. Merge data byte by byte
27 LABEL loop
28 TEQ ...  # While length > 0 loop
29 LDB ...  # Load current byte from agent
30 LDB ...  # Load current byte from peer
31 OR ...  # Combine the two bytes: OR
32 STB ...  # Store result into agent data
33 ADD ...  # Increment current address
34 SUB ...  # Decrement length of data
35 JMP loop  # Go to the beginning of loop
36
37 LABEL loopend
38 MOV r_BASE r_3  # Move to peer's address space
39 END 1  # Destroy peer
40 MOV r_BASE r_2  # Restore the base register
41
42 LABEL out
43 OUT r_1  # Leave on combine link
44 JMP start  # On next node start again

Figure 5.2: Tree-combine agent
the number of agents it combines with (lines 6-7), updating the counter appropriately (lines 19-25). The message handler at the root will monitor the arriving agents by checking their agent count, and combining their data upon arrival. When the total count coincides with the number of nodes in the combine the data is passed to the application.

We can make the same observations we did for the broadcast agent. In addition, this agent also shows an example of using the agent’s local memory to store the state of the computation (lines 6-7, 20-25). Note the data to be carried is indistinguishable from the program data. Simple load/store instructions, coupled with data movement instructions show how to switch between peer address spaces. Currently, this is accomplished in a preemptive way, by the active agent looking up the peers already on the node, buffered in channel queues, and then reading and writing their memory (body). The complete agent program is given in Appendix A.

5.2.2 Individual Communication

An example of expressive power as well as openness of SPIDER is its ability to route messages by using different routing schemes, including the ability for several schemes to co-exist in the same application. This could be useful when routing can be optimized for certain communication patterns and spatial characteristics of the topology, or when multiple classes of traffic\(^1\) are required. In Section 5.4 the adaptivity of SPIDER is discussed in the context of using a customized routing scheme for mesh routing. The routing schemes were implemented as agents that deliver messages according to the specified routing scheme. These agents are equivalent to

\(^1\) Different qualities of service exist in the application (“best effort” delivery, “guaranteed latency” delivery, etc.)
node-to-node send/receive primitives. We experimented with five routing agents. The program listing for each agent is given in Appendix A.

1. Basic routing agent.

A simple node-to-node routing agent which consults the signposts (i.e., routing tables) in the environment (Section 3.2) at each node. Signposts are passive entities that supply information and do not directly control the routing. The routing tables are constructed using a shortest-path algorithm. Agents, when programmed to, can intelligently choose between routes. The basic routing agent follows exactly the routing information received at the node.

2. Valiant 2-step randomized algorithm.

The agent first moves to a random destination, and then to the specified destination. This is a non-deterministic adaptive agent. The two routing steps are performed in a similar fashion to the basic node-to-node SPIDER agent, by consulting the signposts.

3. Priority routing agent

This agent is identical to the basic routing agent, but has higher priority, and is expected to work better when certain messages require some guaranteed latency. In the current implementation priorities are used only for the outgoing channels from a node, not on the incoming channels.

4. Mesh DOR (dimension order routing).

In this case we show how routing information can be supplied in the agent rather than the environment. The experiments were performed on 2D meshes, using routing tables computed using a shortest path algorithm. Instead of relying on these routing tables (signposts), agents were
given information about the actual topology and instructed to navigate using a standard mesh routing strategy, dimension order routing (DOR): the agent first moves along the X-axis and then moves along the Y-axis. In general, at node \((x, y)\) an agent can move in four possible directions, to nodes: \((x-1, y), (x+1, y), (x, y-1)\) or \((x, y+1)\). Each agent uses its local memory to store the current position in the mesh. In addition, topology/routing information is carried by the agent in its memory. This information is a mapping between the four mesh directions and the channels corresponding to the hardware links on each node\(^1\). Agents update their current position as they move, therefore no routing information from the environment is needed.

5. Adaptive mesh routing agent.

The adaptive agent (Figure 5.3) is an extension of the DOR mesh routing agent. Adaptivity was added by having the agent check the load on the links/channels corresponding to the X and Y directions on shortest paths to the destination (lines 31-36). The agent then moves in the less loaded direction (lines 37-44). On equal load, it moves in the X direction.

The two mesh routing agents do not require the physical topology to be a mesh, all it requires is an embedding of the mesh. It is the programmer who provides the mesh routing information. Alternatively, routing can be done by using an agent that traverses the network and labels it (i.e. marks signs to be used for routing), for other agents (its peers) to follow. This approach seems appropriate when different phases of routing are required and the use of an initialization agent takes little time compared to subsequent communication.

\(^1\) We have optimized the size of the information required: the table was reduced by half, by letting an agent to only move in two directions, within the sub-mesh defined by the source and destination. Depending on the adaptive algorithm used, more information may be required.
CHAPTER 5: Evaluation of SPIDER

Figure 5.3: Adaptive mesh routing agent
The ease at which different communication problems were supported by SPIDER’s message centered programming paradigm demonstrates the expressivity and general openness of the system. Openness is a byproduct of following the transportation guidelines in the design of SPIDER. Because agents are interpreted, they are architecture independent, making their use possible in heterogenous networks, with agents freely travelling between different networks. Once the SPIDER environment is ported on different network types, both the agents and the application can run without change. As noted in Section 5.2.2, SPIDER offers a general and open interface to the network without enforcing any routing scheme. The user can implement its own routing algorithm, and many navigation strategies can co-exist in the system. Low-level access to network resources is achieved through special I/O instructions, directly from the agent. The examples showed that communication can be solved without enforcing policies, but rather by providing a programmable interface to the user.

5.3 Performance evaluation

The added flexibility and power does increase message latency. In order to assess the effects of programming overhead we measured a series of communication problems and an application in SPIDER and compared them with similar programs written in Logical C, with and without virtual channels (Logical Systems[42]).

Low-level ChanIn/ChanOut implementations tend to offer performance comparable to that of the hardware, as very little software overhead is involved. Virtual channel implementa-
tions were used as a software reference base, using both customized and general solutions. SPIDER overhead can be divided into three categories:

- **Program and state storage overhead**: each agent has to store its program and state (plus some header information). This will consume more network bandwidth.

- **Program interpretation overhead**: agent instructions are decoded and executed at each node.

- **System overhead**: is due to the SPIDER architecture (processes, memory allocation/deallocation, interprocess communication, semaphores, accounting). Most of these overheads are present in other systems as well.

All of the experiments were performed on the hardware and software environments described in Chapter 4. A physical 8x8 mesh topology (Figure 5.4) was used, on which several other logical topologies (trees) were embedded. The dotted lines in Figure 5.4 represent the 2x2, 3x3, ..., 7x7 and 7x8 submeshes used in our experiments. There was one node outside the mesh, which was mainly used as a root in the experiments for broadcast and combine. We refer to it as “root”.

For timing accuracy, some nodes (those shown with a circle inside the node) were connected to a signal generator. On each timed node, the clock events were intercepted and used to synchronize all the processes (Jiang [33]). The high-priority transputer clocks were used, giving measurement accuracy in the order of a microsecond. Since not all the nodes had access to the global clock, some minor trade-offs were made, and we describe them for each experi-
5.3.1 Collective Communication

It is expected that collective, coordinated communication to be sensitive to the routing method, and that it could benefit from the new features introduced by SPIDER. We find that for comparable levels of portability and coding complexity, the broadcast and combine agents we implemented outperform similar Logical C programs, especially when the size of messages or the traffic in the network increases.
5.3.1.1 Broadcast

The experiment consisted of evaluating four systems/algorithms to obtain: a low-level hardware broadcast time, a software lower bound, a software standard broadcast time, and SPIDER:

1. **Low-level (hardware) broadcast:** `chan.tree`
   
   We used the Logical C low-level communication with ChanIn/ChanOut. A breadth-first-search spanning tree of the network was constructed. Each node receives a broadcast message from its parent and forwards it to its children. This customized program is highly optimized, incurring almost no overhead. The extra time required by the other systems/algorithms tested is the software overhead.

2. **Lower bound for software broadcast:** `vchan.tree`
   
   The program is identical to the previous one, except that it uses virtual channels (VChanIn/VChanOut primitives). This is a lower bound on the software since it is tailored to the topology and incurs little overhead. This low overhead is also a consequence of the optimization in the system, which efficiently handles neighbour-to-neighbour communication. However, the algorithm is hard coded into the application, making it less portable.

3. **Standard software broadcast:** `vchan.send`
   
   This is the standard broadcast algorithm where the source sends the message to each node, one at a time. We chose this algorithm because it is simple, portable, and commonly used on many systems. Its implementation was done in Logical C with virtual channels.
4. **SPIDER broadcast agent:** spider.bcast

The algorithm and the agent used are those described in Section 5.2.1, using a flood filling strategy.

For evaluation, meshes were used, varying in size from $2 \times 2$ to $7 \times 8$. In addition, the size of the broadcast message was varied from 1 byte to 10KB. For accurate timing measurement, the clocks of the broadcast node (the one outside the mesh) and the farthest node from it (the one at the opposite mesh corner) have been synchronized using an external global clock. The algorithms were timed from the injection of the broadcast message at the root until the farthest node was reached. The results of the experiments are shown in Tables 5.1, 5.2, 5.3 and 5.4.

**Table 5.1:** Broadcast using low level ChanIn/Out: chan.tree

<table>
<thead>
<tr>
<th>Mesh size</th>
<th>Broadcast size (bytes)</th>
<th>1</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 \times 2$</td>
<td>1</td>
<td>.05</td>
<td>.25</td>
<td>2.05</td>
<td>20.07</td>
</tr>
<tr>
<td>$4 \times 4$</td>
<td>1</td>
<td>.07</td>
<td>.47</td>
<td>4.09</td>
<td>40.13</td>
</tr>
<tr>
<td>$5 \times 5$</td>
<td>1</td>
<td>.08</td>
<td>.62</td>
<td>5.50</td>
<td>54.27</td>
</tr>
<tr>
<td>$6 \times 6$</td>
<td>1</td>
<td>.09</td>
<td>.77</td>
<td>6.90</td>
<td>68.10</td>
</tr>
<tr>
<td>$7 \times 7$</td>
<td>1</td>
<td>.11</td>
<td>.93</td>
<td>8.31</td>
<td>82.37</td>
</tr>
<tr>
<td>$7 \times 8$</td>
<td>1</td>
<td>.12</td>
<td>1.00</td>
<td>8.94</td>
<td>88.11</td>
</tr>
</tbody>
</table>

**Table 5.2:** Broadcast using virtual channels: vchan.tree

<table>
<thead>
<tr>
<th>Mesh size</th>
<th>Broadcast size (bytes)</th>
<th>1</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 \times 2$</td>
<td>1</td>
<td>.51</td>
<td>.81</td>
<td>2.52</td>
<td>21.61</td>
</tr>
<tr>
<td>$4 \times 4$</td>
<td>1</td>
<td>.71</td>
<td>1.11</td>
<td>4.70</td>
<td>43.01</td>
</tr>
<tr>
<td>$5 \times 5$</td>
<td>1</td>
<td>.85</td>
<td>1.39</td>
<td>6.24</td>
<td>57.96</td>
</tr>
<tr>
<td>$6 \times 6$</td>
<td>1</td>
<td>1.00</td>
<td>1.67</td>
<td>7.86</td>
<td>72.96</td>
</tr>
<tr>
<td>$7 \times 7$</td>
<td>1</td>
<td>1.15</td>
<td>1.97</td>
<td>9.35</td>
<td>87.75</td>
</tr>
<tr>
<td>$7 \times 8$</td>
<td>1</td>
<td>1.23</td>
<td>2.11</td>
<td>10.01</td>
<td>93.75</td>
</tr>
</tbody>
</table>
The results of the experiment using the low level broadcast are shown in Table 5.1. Note that the time increases with message size and network size. As one would expect, the communication time is the time to send the bits on the links, which is directly proportional to the number of bits sent and with the length of the communication path(s). Table 5.2 shows the results of the software lower bound for broadcast. As expected, the figures are close to those of Table 5.1. When the broadcast message is small (less than 100 bytes) the small overhead due to Logical C virtual channels is visible. In contrast, as it is evident from Table 5.3, the traditional software broadcast algorithm has a significant performance loss as soon as the mesh network is larger than 2 × 2, independent of message size. As the message size exceeds 1KB, the performance penalty
becomes even more obvious, the broadcast time being 10 times greater than the customized broadcast. We attribute this behavior to the increased virtual channel multiplexing on physical communication links, especially around the root.

As shown in Table 5.4, one of the characteristics of broadcast performed by SPIDER agents is the uniform increase in communication time, proportional with the network size and with the number of bytes broadcast. This indicates that the SPIDER overhead is relatively constant. Consider each of the following overheads that make up the total overhead.

- Program storage overhead is fixed: it takes 3 words for the agent description, 6 words for the state registers and 24 words for the agent instructions, for a total of 33 words, or 132 bytes. As the data size increases, the SPIDER header overhead becomes a small portion of the total message size. For example, for 10KB broadcast the overhead is less than 2%.

- Program interpretation time per node also remains constant, at about 2ms. All instructions take constant time, except those that copy like replicate (RPL). The copy overhead due to spawning of broadcast agents is considered to be part of the system overhead, thus the interpretation time is constant per node, independent of message size. Program interpretation overhead decreases as a ratio of the total communication time as message size increases.

- System overhead can be approximated by $0.5ms + k*data\_length\_bytes$, with $k=0.1$ accounting for data copying. The data copying that occurs in the execution of the fork (FRK) and replicate (RPL) instructions can be eliminated by using a reference counter to allocate and free the buffers. There is no additional buffer
copying between kernel processes, other than a constant 9-word agent part (description and state registers).

By comparing the numbers in Table 5.4 with those in Tables 5.1 and 5.2 one can notice that, as message size increases, spider.bcast overhead is similar to that of chan.tree and vchan.tree. As expected, the agent broadcast time is always greater than the software lower bound. However, the relative gap between spider.bcast and vchan.tree (Table 5.4 and Table 5.2) decreases with message size, spider.bcast is 10-20 times slower for 1-byte messages, but is only 1.5 time slower when data is of the order of kilobytes. A less clear boundary exists between spider.bcast and vchan.send. From Table 5.4 and Table 5.3 we notice that for 100-byte messages or less, spider.bcast is not as fast as vchan.send. On the other hand, for messages over 100 bytes spider.bcast becomes faster, and when network size increases as well, spider.bcast is about 5-6 times faster. Figure 5.5 shows the spider.bcast communication time relative to that of vchan.tree and vchan.send for 10K messages.

5.3.1.2 Combine

The experiment consisted of evaluating five systems/algorithms performing a combine-OR, to obtain a low-level hardware combine time, a software lower bound, a software standard combine time, and two versions of SPIDER combine agents. A secondary goal was to compare the two versions of SPIDER agents and verify that combining along the tree is better than combining at the root. The five algorithms/systems were:

1. Low-level (hardware) combine: chan.tree
Logical C low-level ChanIn/ChanOut primitives were used. A breadth-first-search spanning tree of the network was constructed. Each node receives combine messages from its children, combines the data and forwards the result to its parent. This customized program is highly optimized, incurring almost no overhead. The extra time required by the other solutions tested is the software overhead.

2. Lower bound for software combine: `vchan.tree`

The program is identical to the previous one, except that it uses virtual channels (VChanIn/Out primitives). This is a lower bound on the software since it is tailored to the topology and incurs little overhead. This low overhead is also a consequence of the optimization in the system, which efficiently handles neighbour-
CHAPTER 5: Evaluation of SPIDER

3. **Standard software combine:** vchan.root

   This is the standard combine algorithm where each node sends its value to the root and the root does the combine. This algorithm is simple, portable, and used on many systems. Its implementation was done in Logical C with virtual channels.

4. **Root-combine agent:** spider.root

   Once injected into the network this agent travels to the root using routing signs. The root starts a message handler that extracts and combines the data as agents arrive.

5. **Tree-combine agent:** spider.tree

   This is the agent we described in Section 5.2.1, with preemptive opportunistic behavior. It moves up in the tree, combining with the peer agents it finds along its path.

For evaluation, meshes varying in size from $2 \times 2$ to $7 \times 8$ were used. One initial difficulty we encountered in this experiment was node synchronization. In a real application nodes are only loosely synchronized, not all of them begin at the same time. Moreover, hardware limitations constrained us to using the global clock on a few nodes only (Figure 5.4). It was thus necessary to devise a simple synchronization. First, the timed nodes were synchronized by the global clock. The network was then partitioned into regions in which the timed nodes broadcast a synchronization message. Upon receiving the broadcast message, each node calculated a waiting
time based on an estimate of the time required for all the nodes to receive the message. The results of the experiment are summarized in Table 5.5. A comparison of the software algorithms/systems is shown in Figure 5.6.

**Table 5.5:** Combine: the bitwise OR on a word

<table>
<thead>
<tr>
<th>Mesh size</th>
<th>spider.tree</th>
<th>spider.root</th>
<th>chan.tree</th>
<th>vchan.tree</th>
<th>vchan.root</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 x 2</td>
<td>3.69</td>
<td>2.90</td>
<td>.09</td>
<td>1.11</td>
<td>3.37</td>
</tr>
<tr>
<td>4 x 4</td>
<td>10.73</td>
<td>10.23</td>
<td>.26</td>
<td>2.58</td>
<td>64.50</td>
</tr>
<tr>
<td>5 x 5</td>
<td>16.17</td>
<td>17.15</td>
<td>.36</td>
<td>3.31</td>
<td>110.13</td>
</tr>
<tr>
<td>6 x 6</td>
<td>20.87</td>
<td>24.51</td>
<td>.45</td>
<td>4.07</td>
<td>173.80</td>
</tr>
<tr>
<td>7 x 7</td>
<td>24.52</td>
<td>34.70</td>
<td>.55</td>
<td>4.07</td>
<td>244.73</td>
</tr>
<tr>
<td>7 x 8</td>
<td>27.50</td>
<td>42.64</td>
<td>.59</td>
<td>5.38</td>
<td>280.02</td>
</tr>
</tbody>
</table>

First, consider the differences between the two SPIDER combine agents spider.tree and spider.root. Observe that the methods perform similarly when the network is small (little contention at the root), but as the network size increases, the root becomes the bottleneck and spider.tree is more efficient, by almost a factor of 2. This clearly shows the advantage of opportunistic combining at the intermediate nodes.

In comparison with the other methods (as plotted in Figure 5.6), the performance of spider.tree lies in between vchan.tree and vchan.root. The lower software bound, vchan.tree is about 10 times slower than the hardware time. In addition, spider.tree is about 4-5 times slower than vchan.tree. We attribute the large gap to the fact that the actual data size is small relative to the total size of the agent, and to the lack of other network traffic. As a result of constant SPIDER overhead, the gap factor remains fairly constant across all network sizes. The other virtual channel combine, vchan.root is sensitive to network size, as network size increases, the combine time increases faster than observed in the other
strategies. When network size exceeds $6 \times 6$, `spider.tree` is 10 times faster than `vchan.root`. Although both methods are portable and easy to implement, it is clear that SPIDER has a better performance.

![Combine comparison](image)

**Figure 5.6:** Combine comparison

### 5.3.2 Individual Communication

Clearly, since SPIDER is based on point-to-point communication, it cannot outperform the underlying system. In this section we are interested in the cost incurred by SPIDER on end-to-end routing. We have performed several experiments on a variety of systems, with different routing strategies for SPIDER. These routing strategies can co-exist since the knowledge is in the agent, not the nodes.
The physical network topology for these experiments was an $8 \times 8$ mesh where sender and receiver nodes were at opposite mesh corners. The longer path length helped in emphasizing timing, message distribution and traffic differences of the various routing schemes. Time measurement was simple to perform: the sender and receiver were always synchronized by the global hardware clock.

The experiments consisted in single sends/receives in an unloaded network. In the next section similar experiments, but with a hot spot introduced in the network, are described. The routing systems/techniques used were:

1. Logical C virtual channels: simple VchanOut/VChanIn between the sender and receiver. The underlying routing system is non-adaptive and simulates the T9000 wormhole routing.

2. Basic routing agent, as described in Section 5.2.2

3. Mesh DOR agent, as described in Section 5.2.2

4. Adaptive mesh routing agent, as described in Section 5.2.2.

The experimental results are given in Table 5.6.

**Table 5.6:** Node-to-Node Routing: no traffic

<table>
<thead>
<tr>
<th>Routing Strategy</th>
<th>Message size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>virtual channels</td>
<td>1.85</td>
</tr>
<tr>
<td>SPIDER: Routing table</td>
<td>7.42</td>
</tr>
</tbody>
</table>
Table 5.6: Node-to-Node Routing: no traffic

<table>
<thead>
<tr>
<th>Routing Strategy</th>
<th>Message size (bytes)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>100</td>
<td>1000</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>SPIDER: Mesh Routing (DOR)</td>
<td>23.63</td>
<td>24.47</td>
<td>32.13</td>
<td>108.06</td>
<td>865.53</td>
</tr>
<tr>
<td>SPIDER: Adaptive Mesh Routing</td>
<td>32.94</td>
<td>33.75</td>
<td>41.55</td>
<td>117.04</td>
<td>876.96</td>
</tr>
</tbody>
</table>

In the unloaded network the basic SPIDER routing agent is within a few milliseconds from virtual channel routing time when the message size is less than 1KB, but its performance deteriorates as the message size exceeds 10KB. We attribute this behavior to the wormhole routing used by virtual channels, which is known to be fast in light-loaded networks (as in our experiment). The other SPIDER agents are much slower for small message size, about 20 milliseconds, but their behavior resembles that of the simple routing SPIDER agent when message size exceeds 1KB. This is expected since the overhead of carrying mesh routing information and the extra program interpretation at a node (which is fixed for a given network) offsets the time needed for sending small data. With the increase of size, data becomes the dominant factor, making the SPIDER overhead small relative to the sending of data. The adaptive mesh agent is about 10ms slower than the oblivious one, at about 0.8 ms per hop. This is due to the extra fixed overhead per node.

5.4 Adaptivity and Fault Tolerance

There has been much debate on the usefulness of adaptive routing schemes, and a large body of research on related issues exists in the literature (Borodin[9], Snyder[8], Dally[18], Per-
tel[53], Pfister[54]). Our intent is not to prove or disprove these techniques, but rather to show that SPIDER supports them. Adaptive, fault-tolerant SPIDER agents are presented, and their behavior and performance is analyzed.

5.4.1 Adaptive Routing

The experiment we conducted was similar to the one used for node-to-node routing except we introduced a hot-spot. The agents are the same as the one described in Section 5.3.2. The experiment was designed so that one link was heavily used by another communication problem. This link has been chosen so that it was on the path from the sender to the destination (as determined from the routing tables). Both adaptive and non-adaptive techniques were timed and compared. Table 5.7 shows the results of the experiment.

Table 5.7: Node-to-Node Routing: on hot spot

<table>
<thead>
<tr>
<th>Routing Strategy</th>
<th>Message size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Virtual channels</td>
<td>2.05</td>
</tr>
<tr>
<td>Basic routing agent</td>
<td>35.18</td>
</tr>
<tr>
<td>Mesh DOR agent</td>
<td>49.85</td>
</tr>
<tr>
<td>Adaptive mesh routing</td>
<td>35.16</td>
</tr>
</tbody>
</table>

By comparing Tables 5.6 and 5.7 one can notice the effect of the hot spot on routing performance. With the hot spot, virtual channel routing gets a slowly increasing penalty across message sizes (due to wormhole routing), whereas the SPIDER basic routing agent is affected more for smaller message sizes. The non-adaptive mesh routing agent's performance also deteriorates. Without a hot spot the oblivious DOR was 10ms faster than the adaptive one. With the
hot spot it is slowed down by as much as 30 ms. For the adaptive agents the effect of the hot spot is minimal since it could avoid the hot spot. The only minor slow down was due to contention for the interpreter on one of the hot nodes.

5.4.2 Fault Tolerant Routing

The need for fault-tolerant network increases as multicomputers grow larger. Most networks have built-in redundancy in the form of multiple paths from source to destination. However, because of performance constraints, few networks provide support for fault-tolerance (Bolding[8], Dally[18]). The approach taken in SPIDER is to delegate fault handling to the programmer, by providing the following mechanism: when a link fails, the corresponding link_out processes dequeues the agents on the channel queue and re-interprets them. Agents can detect the lack of progress and initiate appropriate recovery action. Agents can also check the size of the channel queue and initiate the recovery when the queue is full.

The experiment consists in of several simulated link faults. On selected links, represented by crossed lines in the lower part of Figure 5.7, permanent faults are created. The links were selected on different possible paths from a source to a destination. Fault tolerant agents were implemented on top of the basic routing agent. Agents detect faults by either observing they arrive from the same node they just left, or that the desired outgoing channel queue is full. The recovery action is simple: randomly select, if one exists, an available channel different from the one used to arrive at the node.

As shown by the arrows in Figure 5.7, 1000 agents were sent between two nodes, first in a network without faults (upper figure), and then in a network with faults (lower figure). When
Figure 5.7: Traffic with and without faults
CHAPTER 5: Evaluation of SPIDER

no faults occurred, all the agents followed the same path, as shown by the thick arrows (recall that basic routing behavior follow the suggestions from routing signs). When the network contained faults traffic diffused along the non-faulty links. The size of the arrows in the lower part of Figure 5.7 signifies the number of messages on the corresponding link. All 1000 agents did arrive at the destination.

The performance of the fault-tolerant agents was reasonably good, relative to the basic routing agents (with no fault-tolerant behavior): in a fault-free network, fault-tolerant agents were 34% slower than basic routing agents; in a faulty network, the fault-tolerant agents were only 5% slower than they were in the free-fault network.

5.5 Application: Cholesky Factorization

In order to analyze application time rather than point-to-point time, we concluded the series of experiments with an application, the Cholesky Factorization of matrices. Implementations in SPIDER and Logical C with virtual channels were compared. The original algorithm, written in Trollius (OSCRC[52]), is synchronous: a node does not process a column unless previous columns have been processed, either by itself or by other nodes. Since SPIDER is more flexible, we also experimented with a modification of the first algorithm\(^1\), that allows for more asynchronous communication to better take advantage of the message-centered programming approach. Given the agents (the broadcast agent and the basic routing agents described in Sections 5.2.1 and 5.2.2), there was little code difference between the original application and the new ones.

---

1. Only a couple of lines of code were moved from inside a loop to another loop.
Table 5.8: Cholesky Factorization of a 65x65 matrix

<table>
<thead>
<tr>
<th>Routing Strategy</th>
<th>Mesh size</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 x 2</td>
<td>3 x 3</td>
<td>6 x 6</td>
</tr>
<tr>
<td>virtual channels</td>
<td>3323</td>
<td>3513</td>
<td>4434</td>
</tr>
<tr>
<td>SPIDER: original algorithm</td>
<td>3856</td>
<td>3954</td>
<td>4162</td>
</tr>
<tr>
<td>SPIDER: modified algorithm</td>
<td>2359</td>
<td>2316</td>
<td>2237</td>
</tr>
</tbody>
</table>

Table 5.8 shows the time required to factor a 65 x 65 matrix on 2 x 2, 3 x 3 and 6 x 6 meshes. It is evident that the problem does not scale well for both virtual channels and the spider version of the original algorithm, but it does for the asynchronous version with SPIDER agents. With the same algorithm, virtual channels perform better than SPIDER when the network size is small (2 x 2 and 3 x 3) but SPIDER outperforms virtual channels when network size is larger (6 x 6), demonstrating the advantages of SPIDER for communication-intense problems. Moreover, due to the better use of network resources, the modified version of the algorithm written in SPIDER outperforms the other two by almost a factor of two.

5.6 Summary

In this chapter the message-centered paradigm was evaluated according to the following criteria: expressivity, performance, adaptivity and fault-tolerance, openness and generality. It was found that programming the network is a powerful tool for expressing communication in a natural way, that can lead to performance improvement by controlling communication. In general, the paradigm provides better performance than equivalent (in terms of portability, ease of coding) node-centered approaches. However, in problems with less communication the node-centered solutions performed faster, because the network overhead was less.
CHAPTER 6

Related Work

This chapter brings together different research topics related to our message-centered paradigm of programming the network. This body of work spans software and hardware communication systems, Active Messages and Reactive Kernel paradigms, and Actor and software intelligent agent models. SPIDER is compared with these systems by identifying the differences and similarities in design and problem domain.

6.1 Communication Systems

The problem of routing messages has been intensively studied in a variety of contexts (Borodin[9], Felperin[21], Leighton[39]). Traditionally, routing of messages is an implementation of algorithms that are defined for general network topologies (shortest-path algorithms) or regular...
topologies (dimension order routing), and cannot be changed. Sometimes, these algorithms are extended with some sort of adaptive behavior or non-determinism. SPIDER is at the opposite extreme, there is not a built-in routing scheme, rather the user chooses his own.

There are two main approaches to the routing problem: software routing systems and hardware routing processors.

### 6.1.1 Software Systems

The complexity of software communication systems ranges from simple routing harnesses like Tiny (Clarke and Wilson[12]) to complex message passing interfaces such as MPI (Message Passing Interface Forum[45]). In between, there are run-time systems or full operating systems, that provides not only end-to-end routing but also group communication and other system services, like buffering or multitasking: Logical C (Logical Systems[42]), PVM (Beguelin[6]) Trollius (Fielding[23]), Helios (Powell[56]), Taos (Pountain[55]).

Higher level libraries that can sit on top of SPIDER (such as MPI) are not directly relevant, because they do not deal with the actual routing of messages, but rather with the interface they offer to the application programmer. SPIDER can provide agencies, libraries of agents, which can be used to implement application programming interfaces.

We briefly describe two such systems, Tiny and Ordered Dimension, as they are representative for the class of message routing software systems.
Tiny

Tiny is a message routing harness for T800 transputers developed at EPCC (Clarke and Wilson[12]). The routing of messages is based on routing tables, containing shortest paths between any two nodes in the network, employing a store-and-forward technique. Routing is fully determined by the information contained in these routing tables. The only flexibility Tiny offers is the choice, at initialization time, between oblivious routing and adaptive routing. The adaptive scheme is not deadlock free. Unlike Tiny, SPIDER agents may or may not use routing tables. In addition, an arbitrary number of routing schemes can co-exist simultaneously in the system, and deadlock avoidance can be controlled by the user (although it may not be trivial to do it).

Tiny supports only one-to-all broadcast. The broadcast strategy is based on a broadcast tree built during system initialization. In a lightly-loaded network it is likely that this method is efficient (recall SPIDER was slower than a tree broadcast method), but when the communication increases, the links dedicated to broadcast can become congested and it is not possible to make use of other uncongested links.

At the architectural level, Tiny is a communication kernel consisting of a collection of agents running on each transputer. The notion of agent should not be confused with that of SPIDER. In Tiny an agent is a kernel process that manages some resource. Each agent handles an external link or a channel between Tiny and a user process. This communication architecture design is quite common in other message routing systems as well, and we adopt a similar basic design for the SPIDER environment. There are five different agents used with Tiny to implement the router, somewhat similar to SPIDER kernel processes.
The only interface Tiny offers to the user is a set of read/write communication primitives and a broadcast function. Having such a minimal interface makes Tiny efficient, but inflexible. The routing information is initialized at configuration time and cannot be changed fine tuned to the user application. Tiny can run on any topology, but its adaptive routing technique is of little practical use as it can lead to deadlock or livelock. The non-adaptive scheme is unable to avoid congestion when the communication load is unbalanced.

**Ordered Dimension**

This message routing system was developed by the Inmos Central Application Group, based on the ordered dimensions approach proposed by Dally and Seitz[17]. The network channels are partitioned into classes, and classes are ordered in a way that reflects the topology of the network. A message having traversed the channels of a class does not revisit them after travelling through the channels of the next class. As a result deadlock cannot occur.

Ordered Dimension routing cannot be applied to any topology, but can be applied to regular topologies such as the hypercube, torus or ring. Message routing is oblivious and the path taken by a message is statically defined and is derived from the channel dependency graph, where, due to channel ordering, there is only one path between two nodes. Although this can lead to network congestion, messages do arrive in the order they were sent, a useful property. The scheme has very little fault tolerance and does not provide support for group communication. SPIDER mesh routing agents are particular cases of this routing scheme, but SPIDER also provide adaptivity.
6.1.2 Routing Processors

With the rapid development in the field of VLSI design, research in network routers has been very active, although practical experience with routers is still limited. The routers used in most parallel machines are oblivious, and adaptive routers have only appeared in a few machines, such as HEP, CM-2 and CM-5. Oblivious routers have a simpler hardware complexity and work well under light randomized network traffic, while adaptive routers are more flexible, fault-tolerant and handle hot-spots better (Dally[15], Snyder [8], Felperin[21]). Since a router is a fundamental component of a parallel computer, in order for the machine to be fast, routers demand aggressive designs, that are very difficult to get right. Also, the efficiency of a router can only be assessed in the context of its use, rather then performance figures on bandwidth and latency. In this section we will review some of the most important routers and compare them with SPIDER.

Torus and Mesh Routing Chips

The first generation of routing processors, Caltech's Torus and the Mesh Routing Chips (Dally and Seitz[16], Flaig[24]) were designed for regular networks such as tori and meshes, and routing was based on dimension order routing. Wormhole routing was the choice of flow control. In such networks, messages are completely routed by the hardware, with no interference from software. Both chips were oblivious and limited to special topologies.

The Network Design Frame

The Network Design Frame (NDF) is a self-timed routing chip developed at MIT (Song[63]). NDF was designed for n-dimensional mesh networks with wormhole routing. It provides two
logical networks (by virtual channels), one for user messages and one for system messages. SPIDER link_in processes offer the same functionality by filtering the agents, based on agent identifier.

Although still an oblivious router, NDF separates the messages into classes, in a similar way the modern architectures separate the user from the supervisor mode in a program. Similar to TRC and MRC, this router is not general, it cannot route in arbitrary topologies.

Message-Driven Processor (MDP)
The Message-Driven Processor is the chip used in the J-Machine, a parallel computer developed at MIT (Nuth and Dally[51]). It contains both a processing unit and a routing unit. Its design was based on two basic ideas: (a) it should implement a general-purpose multicomputer processing node, to provide communication, synchronization and naming mechanisms required to support several different parallel programming models, and (b) it should be an inexpensive VLSI component.

The J-Machine network is a 3-dimension mesh network, with two virtual channels to support two logically independent message priorities. MDP is not a general purpose router, it cannot handle arbitrary topologies; moreover, it is not a pure router, it is also a processing chip. Its instruction set promotes a tight integration of communication with computation. There is no adaptivity and no way to avoid the formation of hot spots in the network. SPIDER is similar to MDP with respect to supporting prioritized messages and different programming models (although another layer on top of SPIDER must do the necessary mappings).
IMS C104 / T9000

The Inmos C104 is the routing chip for the T9000 series, implementing interval labelling routing (van Leuuwen[37], May.[44]). The C104 is a full 32x32 non-blocking crossbar, based on wormhole routing. With a T9000 and the IMS C104 it is possible to define communication channels between two processes independently of where they are located in the network.

The interval labelling scheme associates non-overlapping intervals (consecutive Transputer labels) to each link. As a message arrives at a node, the interval to which its destination label belongs is selected, and the message is sent out the corresponding link. Interval labelling uses wormhole routing. However, routing is oblivious; there is no possibility to adapt the message routing according to the communication load in the network. To address this issue (hot spots, congestion, etc.) the C104 chip uses Valiant's two-phase routing, in which a message is first sent to a random address; the disadvantage of this scheme is that it doubles message latency.

As shown in Chapter 5, this 2-step randomized routing scheme has been implemented as a SPIDER agent, and it co-existed with other schemes. Note that the SPIDER agent does not implement the wormhole flow control.

CM-5 Router

The CM-5 (Leiserson et al.[40]) is a distributed memory multiprocessor whose nodes are interconnected by two disjoint data networks, a diagnostic network and a control network (for synchronization and for group communication). SPIDER offers two software alternative two
disjoint networks: one by using higher agent priorities, and the other by programming collective communication according to application optimization criteria.

CM-5 interconnection network provides multiple paths from a source processor to a destination processor. To route a message from one processor to another, the message first moves up the tree to the least common ancestor of the two node processors. The selection of what upward link to follow is done at random, with the restriction that the link is not congested by other messages at each level. Next, the message moves from the common ancestor to the destination on a unique downward path. A random routing algorithm is used to balance the network load and avoid hot spots in pathological cases.

Although the CM-5 router is adaptive, its adaptive scheme cannot be controlled by the programmer and debugging of parallel programs can be difficult due to communication randomness.

**Chaotic Router (CR) and Reliable Router (RR)**

CR is a randomized adaptive packet router based on a pipeline design (Bolding et. al.[8]). It tries to minimize the impact of the buffer queue management and uses a deflection adaptive algorithm for routing. RR was designed for 2-dimensional mesh topologies (Dally et al[18]). It uses adaptive routing coupled with a link-level protocol for increased reliability.

As yet, these routers have only been used in academic research, although they seem quite promising. Unlike most other routing processors, the Chaotic Router addresses issues such as livelock, adaptivity (based on link congestion), fault tolerance, it even handles out-of-order packets, albeit in a rigid way. SPIDER approach to fault-tolerance is a bit different, it detects
the faults, and let the agents recover from it. The agents should be intelligent enough to adapt to the new circumstances.

UMich-SPIDER

The original name was SPIDER (Scalable Point-to-point Interface DrivER), but to avoid confusion with our system, we call it UMich-SPIDER, as it has been developed at University of Michigan (Dolter[19]). Controlling network resources is accomplished statically, node-centered: at initialization, a microprogram is downloaded on each node, and the message headers are read by the microprogram to select the appropriate microprogrammed routing strategy. Messages contain a header that can be read by the microcode which selects among the routing schemes supported. In contrast, SPIDER allows different routing schemes to be used for the same message, dynamically, as the message/agent travels in the network. The microcode also supports fault tolerance, however this was not addressed by the authors (Dolter[19]).

UMich-SPIDER is a network adapter that provides scalable communication support for point-to-point distributed systems. Unlike other routing processors, UMich-SPIDER does not implement specific protocols in VLSI, it provides hardware support for higher-level host policies. It pushes software control close to the links, by dedicating a small processor to each channel. This is similar to our system where we do this in software. Another similarity with our SPIDER system is the spawning facility: a packet can spawn multiple copies of itself at each hop on the route. Spawning makes it possible to efficiently support broadcast or other group communication.
Having most of the functionality at the hardware, rather than software level, can give UMich-SPIDER a performance advantage over SPIDER. However, our system is based on a different paradigm, message-centered, rather than node-centered, that is more dynamic, open and portable.

6.2 Messages as Action Triggers

6.2.1 Reactive Kernel

The Reactive Kernel (RK) is a node operating system for the second generation of multiprocessors (Seizovic[61], Su[64]). Its novelty was the idea of reactive scheduling. Messages are treated as tokens to run, and the running entity is a handler lying on the node where the message arrives. Unlike RK, SPIDER contains the instructions in the message itself, rather than in the handler on the node.

Basically, RK can be viewed as a message processor, and handlers as actors (see the Actor model below). A handler is invoked when the message at the head of the receive queue is tagged for it; as a “reaction” to the message, it may send messages, create new handlers, and changes its behavior (become a new handle). A handler is like a kernel process, but it is stateless (no information preserved between two invocations).

Unlike SPIDER, message routing is not part of RK, it is separated, mostly because the hardware provides routing chips. The routing system interacts with the rest of the node via dedicated hardware (a message interface, MI), through the send and receive queues. The programmer has no control on how the routing is done, but one may try to use the handlers to support
this kind of functionality. However, doing so will force each node to contain a handler for each message, which may consume memory. The use of handlers also requires same memory image of the application on each node.

6.2.2 Active Messages

Active Messages is a communication mechanism that uses the control information at the head of a message as the address of a user-level instruction sequence that will extract the message from the network and integrate it into the on-going computation (Thorsten et al.[67]). This is similar to the Routing Kernel idea, although the handlers are different: an Active Message handler is in the user-address space and must execute quickly and to completion. Handlers do not perform any computation on the messages. Their actions are similar to those of an exception handler. The sender launches the message into the network and continue with the computation; the receiver is notified or interrupted on message arrival and runs the handler. This is a pipelined view of the network, thus it will improve over the standard send/receive message passing mechanisms.

Like SPIDER, Active Messages do not address naming issues, its programming model is one driven by messages. Active Messages separate communication from the computation, however the user does not have control over communication. We have used this mechanism as the entry/exit points in SPIDER, to implement injection of agents into the network and their extraction (the downloader process). One could implement some SPIDER features using Active Messages, but the basic difference still remains, the control is in the node, rather than distributed to where it is needed, in the messages.
6.3 Actor Model

The Actor Model (Agha[2]) does not attempt to solve routing or other communication problems, but it is included here because it is the closest analogue in parallel computation to our model of communication agents.

Actors are computational objects that encapsulate a state and expected behavior, with an interface (methods) that manipulates the local state when a method is invoked. The Actor Model unifies objects with concurrency: actors are autonomous and concurrently executing agents (objects) which run asynchronously and can send each other messages. An actor is a two-tuple consisting of a mail address (for other agents to send messages) and a current behavior. In response to receiving a message, an actor may either asynchronously send a message to a specified actor, create an actor with a specified behavior, or become a new actor, by replacing current behavior.

This model has real world analogy of open systems, with continuous flux of new information, massive concurrency and multiple actors functioning concurrently and cooperatively. Decisions are locally made and inconsistencies are solved through a mechanism for conflict resolution and reasoning in the presence of contradictions. There is no global control for routing, to avoid a bottleneck, communication in actors is point to point. The Actor Model is very related to our own agent model in terms of framework. It is the context in which the models are used that are different. We captured the essential features of actors and created a new communication programming paradigm with communication agents playing the analogue role of computation actors.
6.4 Software Agents

SPIDER is a type of software agent, the latest research tool in Artificial Intelligence (Russel and Norvig[57]). However, the software agent models have an implicit assumption that they solve some processing/computation problem. In SPIDER, just like computation, communication itself is treated as an agent model. It is just like the analogy between the macro and microscopic world, SPIDER being a low-level agent model.

Our design was not inspired by the existing agent architectures used in either AI or on the Internet (softbots, Telescript). Rather it evolved from the low level system view of routing in multicomputers, but the end result has quite a few similarities with agents in those domains. The assumption was that processes execute on nodes and that data moves between processes. Communication agents are used to deliver this data. In comparison, the software intelligent agents on the Internet are programs that visit nodes and execute the same algorithm on each node. Their transportation is done using the network protocols (TCP/IP), without user access to low-level network resources.

A more recent agent model is that of transportable agents (Kotay and Kotz[35]). Basically, they move computation where the data is, but do not deal with communication issues like SPIDER communication agents do. These transportable agents are capable of suspending their execution, transporting themselves to another host (high-level interaction with the network), and resuming execution from the point at which they were suspended. There are similarities to SPIDER, however, a major difference is the execution and what is being acted upon. SPIDER agents execute instructions in order to reach a destination, whereas transportable agents exe-
cute instructions in order to obtain a result, where moving is done transparently (possibly using SPIDER agents!).
CHAPTER 7

Conclusions and Future Work

"Veni, Vidi, Vici"

Julius Caesar

7.1 Review of Contributions

This thesis has explored a message-centered approach to communication in multicomputer networks. It has demonstrated that programming the network is suitable for controlling communication/routing. Programmability, using a message-centered approach is a natural, flexible way to provide the necessary communication mechanisms without enforcing policies. In the new paradigm messages become communication agents, with intelligence and mobility. This approach works best in communication-intense problems, without being restricted to them. Each of the specific contributions is reviewed here.
CHAPTER 7: Conclusions and Future Work

A New Paradigm

In this thesis a practical paradigm, message-centered programming of the network, has been proposed. It has potential for development, conforms with the current trends of openness, and can improve performance in computation-bound parallel applications. The primary characteristic is the access of low-level network services by organizing communication around messages, rather than nodes. Messages have been transformed into communication agents, which deliver data and self-route in the network. Programming the network removed the inflexible network interface and replaced it by an open, more flexible one. Through agents, programmers can coordinate communication and tailor it to the current application, by directly managing network resources.

System Design

The design of the SPIDER system was described in Chapter 3. It has similarities with agent models used in AI, consisting of an environment (the interface offered by networks) and agents (the means to control network resources and carry data). The agent architecture offers a machine view of the agent. The instruction set and the assembly language are the interface to the communication programmer. Agent behavior is specified by initializing its state and body (writing the program and loading the data). The system architecture promotes a new way of organizing communication, by programming it, independently of the computation.
Prototype Implementation

The implementation showed that the paradigm is feasible and provided a number of insights into the paradigm, which in turn contributed to its final design. The implementation was used in numerous experiments with the paradigm. Even though the implementation is for transputer-based multicomputers, we expect it is easily ported to other architectures with similar features (such as concurrent processes support and reliable point-to-point connections).

Experimentation

The experiments demonstrated the successful application of our paradigm to some of the practical problems in message passing: node-to-node routing and group communication. They showed the expressivity, openness and generality of the paradigm. They also demonstrated the adaptive, fault tolerant nature of the paradigm, as well as its good performance in communication-intense problems, such as collective communication. In applications with little communication such as node-to-node routing in traffic-free networks SPIDER performed less efficient, due to large overhead ratio with respect to the total communication. In all cases of collective communication, SPIDER performed faster than similar node-centered solutions, but less efficient than the highly customized node-centered solutions.

7.2 Future Work

This work was mostly intended to introduce the paradigm and show its suitability in solving communication. Several outstanding issues remain to be investigated or improved.
7.2.1 Instruction Set

An important issue is the effect of the instruction set on performance. The agent language has evolved continuously during this research, but more experimental support is required for understanding the trade-offs to be made. Another issue is that of the generality of the active and I/O instructions. Currently the environment is viewed as a collection of external devices that are accessed by specific instructions. Instructions are customized, making the model efficient. However, in heterogenous networks which may contain different abstractions, it may be more appropriate to use generic I/O instructions for all external devices\(^1\), in the same way Unix uses file descriptors and I/O system calls like open, read, write. The real problem here is finding the commonality of services needed on all the networks.

7.2.2 Different Architectures

To port SPIDER to architectures other than transputers one must address two issues: mapping of the environment (nodes, channels, address, signs) to actual network resources, and programmer's access to the low-level network services. In order to port the system to a workstation environment, one must work with either a broadcast network (Ethernet) or a point-to-point network (ATM). SPIDER implicitly assumes a point-to-point underlying network. Programmer's access to the data link layer (Ethernet) is necessary. However, security and protections are important issues in this environment.

---

\(^1\) Memory mapped device drivers are probably required.
7.2.3 Security

Adding security to the system can be important. In our setting, it was not required, as we were the owners of the machine, but when users need access to other machines, accessing low-level services could lead to problems. Agent identifiers could be used as capabilities, so that the interpreter would only execute certain instructions if the necessary capability is present. For example, special system agents can exist in the system, and they can perform management operations, such as un-blocking the queues, garbage collection, erasing signs, and so on.

7.2.4 Injecting/Extracting Agents

To inject an agent in the network the application needs an agent handle. We have chosen to hide the agent by a function call with certain application parameters (usually buffer address and length, agent priority and identifier, etc.), but more research is required. To extract an agent's data from the network we used a mechanism somewhat similar to Active Messages. For each communication problem a dispatcher (downloader) invokes a user message handler every time an agent arrives with data for that communication problem. There is a potential performance penalty since the downloader is started every time a node must receive data. A more efficient approach, using eventually just one dispatcher, should be investigated.

7.2.5 Interface to Applications

Currently, there is a simple agent interface to the application. Basically, for each agent class we introduced a specific interface. For example, to perform broadcast, two primitives are provided: bcast_send() to inject the agent, and bcast_recv() the message handler\(^1\). However, a better

\(^1\) One can hide the downloader by making bcast_send() the receive routine, but then receive has a blocking semantics.
method of using agents in applications should be designed and the interface with other systems such as MPI explored. In Gropp[28] a model of an MPI implementation is described. It suggests the separation of the underlying communication system (ADI - abstract device interface) from managing communicators, derived data types and topologies. If SPIDER is used as an ADI, then a mapping of MPI entities (communicators, groups) to SPIDER entities (nodes, channels, addresses) is first required. Since SPIDER uses pairs (node, address) as communication endpoints, a run-time mapper can be ported from other systems with similar low-level naming schemes (such as PVM). Four functions of the ADI must be supported by SPIDER:

1. sending and receiving: Section 5.2.2 provides a large choice of communication agents for end-to-end communication. Gropp[28] suggests the use of end-to-end primitives for collective communication, but here SPIDER can help by providing agents for this kind of problems (Section 5.2.1).

2. data transfer: The main issue is non-contiguous data. It is the SPIDER instruction set that requires investigation for solutions to this problem.

3. queueing: Message queueing refers to the receiver's queues of pending messages and of unexpected messages (for which no matching receive has been issued). SPIDER's asynchronous downloaders with appropriate message handler have the necessary semantics for this requirement.

4. miscellaneous device-dependent functions: MPI contains some device independent routines, such as probing and waiting. Agent solutions are possible; deadlock prevention is an issue here.
Thus, with some more work, it is possible to use SPIDER as the underlying communication system for MPI.

### 7.2.6 Solving Communication Problems

The most important of all is to solve as many communication problems as possible and use the system in real applications. We have given examples of how to solve some collective and individual problems, as a starting point for later development. Implementing classes of customizable agents would be a useful development. Specific examples of agents would be agents for pipelined flow control (wormhole routing, virtual cut-through), or to dynamically reconfigure the network.

Pipelined flow-control agents can be useful for sending large data. Data is broken into smaller packets, and one agent leads the way, marking the nodes and reserving channel queue slots for the agents to follow.

Dynamic network reconfigurability is possible in some networks such as Inmos C100-based networks. Currently, there are no instructions to reconfigure a network, but this is straightforward to implement. Moreover, the ability for agents to learn makes it possible to dynamically change the network topology.

### 7.2.7 Formal Framework

There has been no mention of formally analyzing the use of agents and their algorithms. It is interesting to analyze and predict the performance of different agents (such as broadcast and combining agents) both as individual communication problems or as part of larger applications.
Suggested measures are the number of instructions executed, total task time or network traffic generated.
Bibliography


[27] Gelernter D.: "Generative Communication in Linda", *ACM Transactions on Programming Languages and Systems*, vol7(1), 1985


[52] Ohio Supercomputer Center Research Computing: Trollius Command Reference and Trollius Library Reference, The Ohio State University, 1992


[59] Sapaty S., Corbin M., Borst P.: "Towards the development of large-scale distributed simulations", *12th Workshop on Standards for Interoperability of Distributed Simulations*, Orlando, FL, March 1995


APPENDIX A Agent Program Listing

A.1 Broadcast Agent

# This agent visits all the nodes, by replicating on all available links.
# It marks the node it visits to avoid message duplication
# Notice that the bcast message is not sent to the source node.
DEF bcastnode source # Broadcast node
DEF bcastlink id # Soft link to the application

MOVI bcastnode
MOV r_3 r_0 # Just in case node > 15
TMRK id # Has the node been visited?
BRT end # If yes, die
# else
MRK id # Mark the node as visited
WHR r_1 r_2 # Current node and link
TEQ r_1 r_3 # Is this the source node?
BRT start # If yes, replicate on links
# else
FRK bcastlink # Download data on soft link
LABEL start # Replicate on all available links
MOVI 0 # Pass link number to bcast routine
EXE bcast # Broadcast on link 0
MOVI 1 # Pass link number to bcast routine
EXE bcast # Broadcast on link 1
MOVI 2 # Pass link bcast routine
EXE bcast # Broadcast on link 2
MOVI 3 # Pass link to bcast routine
EXE bcast # Broadcast on link 3

LABEL end # Terminate
END 1 # Terminate and release resources

## This procedure replicate the program on link LINK
LABEL bcast
TLC r_0 # Is the link connected?
BRF return # If no, return
# else
TEQ r_0 r_2 # Have I arrived on this link?
BRT return # If yes, return
# else
RPL r_0 # Replicate on this link

LABEL return # Return from procedure
RET
A.2  Tree-combine Agent

# This agent moves up the combine tree and, whenever it finds a peer
# it removes it from the queue, combines its data with that of the peer's
# and enqueues itself in the queue

LABEL counter
DATA 0x1 0x0  # Counts how many agents are merged
DATA 0x0 0x0  # Initially, count = 1

LABEL start
DEF combinelink id
DEF rootnode 1  # Link on which agent leave
WHR r_2 r_3  # Set root to 1
TEQ r_2 rootnode  # Get agent location: node/link
BRF lookup  # Reached the root?
MOV r_1 r_0  # If not, find peer agents to combine
JMP out  # else

LABEL lookup
ROUT rootnode  # Prepare to download
MOV r_1 r_0  # (complete last operation)
LKPD r_1  # Enqueue for downloading
BRF out  # Look for peers on this node

MOV r_2 r_BASE  # Get link to follow to root
MOV r_3 r_0  # Link to follow to the root
LKPD r_1  # Lookup peer agent. Result in r_0
BRF out  # No agent found, move w/o combining
MOV r_4 r_0  # else, peer found, combine with it
ADD r_4 r_BASE  # Its base register is stored in r_0
LDW counter  # by the LKPD operation
MOV r_4 r_3  # r_4 will hold the offset where the
LDW counter  # memory operations occur
ADD r_4 r_0  # Save current base register
MOV r_BASE r_3  # Save peer agent base register
MOV counter r_0  # Get the combine counter of agent
MOV r_BASE r_2  # Save current counter
MOV r_2 r_BASE  # Move to peer's address space
LDW counter  # Get combine counter of peer
ADD r_4 r_0  # Add combine counters.
MOV r_BASE r_2  # Restore the base register
STW counter  # Update combine counter

# Do the actual combine. Loop through each byte and merge data
# The loop is ended when data_length bytes have been combined.

MOV r_4 r_DADDR  # Get data offset
MOV r_5 r_DLEN  # Get data length
APPENDIX A: Agent Program Listing

LABEL loop
  TEQ r_5 0  # While length > 0 loop
  BRT loopend
  LDB r_4  # Load the current byte from agent
  MOV r_6 r_0  # Save current byte
  MOV r_BASE r_3  # Move to peer's address space
  LDB r_4  # Load the current byte from peer
  OR r_6 r_0  # Combine the two bytes: OR
  MOV r_BASE r_2  # Restore the base register
  STB r_4  # Store result into agent data
  ADD r_4 1  # Increment current address
  MOV r_4 r_0  # (increment finished)
  SUB r_5 1  # Decrement length of data
  MOV r_5 r_0  # (decrement finished)
  JMP loop  # Go to the beginning of loop

LABEL loopend
  MOV r_BASE r_3  # Move to peer's address space
  END 1  # Release memory used by peer
  MOV r_BASE r_2  # Restore the base register

LABEL out
  OUT r_1  # Leave on combine link
  JMP start  # Start from the beginning (replicate)

A.3 Basic Routing Agent

# This is the equivalent of a send primitive
# dest = destination node
# if = communication problem identifier

DEF dwnld id  # Application channel id

LABEL start
  MOVI dest  # Destination node
  MOV r_4 r_0  # If dest>15 a register is needed
  WHR r_1 r_2  # Get location
  TEQ r_1 r_4  # Are we done?
  BRT end  # If yes, then stop
  ROUT r_4  # else
  OUT r_0  # Get route to dest
  JMP start  # Move further
  # On next node start again

LABEL end
  OUT dwnld  # Download
A.4 Valiant 2-step Randomized Routing Agent

# Point-to-point agent based on simple deterministic routing

# src = the source node
# dest = destination node
# if = communication problem identifier

LABEL intDest
    DATA 0x0 0x0  # Space for intermediate dest
    DATA 0x0 0x0
    DEF dwnld id  # Link to application
    DEF numnodes 65
    RND numnodes  # Get a random dest
    STW intDest  # Store intermediate dest

LABEL start1  # Route to intermediate dest
    LDW intDest  # Get intermediate dest
    MOV r_4 r_0
    WHR r_1 r_2  # Get location
    TEQ r_1 r_4  # Are we done?
    BRT start2  # If yes, then route to final dest
    ROUT r_4
    OUT r_0
    JMP start1  # On next node start again

LABEL start2  # Route to final destination
    MOVI dest
    MOV r_5 r_0
    WHR r_1 r_2  # Get location
    TEQ r_5 r_1  # Are we done?
    BRT end  # If yes, then stop
    ROUT r_5
    OUT r_0
    JMP start2  # On next node start again

LABEL end  # Download message
    OUT dwnld

A.5 Mesh DOR Agent
APPENDIX A: Agent Program Listing

# SPIDER program to send messages in a mesh. DOR (dimension order routing)

# Assume that (Xcurr, Ycurr) and (Xdest, Ydest) are the mesh coordinates
# of the sender and the receiver nodes.
# Since the current coordinate change at every node, they are stored in the
# first two bytes after the routing table.
# It is assumed that they are no greater than 15 (i.e. 15x15 mesh).
# Xcurr ≤ Xdest, Ycurr ≤ Ydest

LABEL routeTable  # Start of routing table
    DATA 0x88 0x88  # 2-9
    DATA 0x88 0x88
    DATA 0xcc 0xcc  # 10-17
    DATA 0xcc 0xcc
    DATA 0x88 0x88  # 18-25
    DATA 0x88 0x88
    DATA 0x88 0x88  # 26-33
    DATA 0x88 0x88
    DATA 0x88 0x88  # 34-41
    DATA 0x88 0x88
    DATA 0x88 0x88  # 42-49
    DATA 0x88 0x88
    DATA 0xcc 0xcc  # 50-57
    DATA 0xcc 0xcc
    DATA 0x00 0x00  # 58-65
    DATA 0x00 0x00

LABEL XcurrAddr  # Location of current mesh x coord
    DATA Xcurr 0  # Store the current x and y

LABEL YcurrAddr  # Location of current mesh y coord
    DATA Ycurr 0  # Store the current x and y

DEF dwnld id  # Download link

LABEL start
    MOVI XcurrAddr  # Get memory address holding Xcurr
    MLT r_0 2  #
    LDB r_0  # Get Xcurr
    MOV r_1 r_0
    MOVI YcurrAddr  # Move to the next byte, for Ycurr
    MLT r_0 2
    LDB r_0  # Get Ycurr
    MOV r_2 r_0
    TEQ r_1 Xdest  # Xcurr = Xdest ?
    BRF notX
APPENDIX A: Agent Program Listing

TEQ  r_2   Ydest  # Ycurr = Ydest ?
BRF  notY
OUT  dwnld
END  1

LABEL notY
EXE  incrementY
MOVI 1
EXE  getlink
OUT  r_0
JMP  start

LABEL notX
TEQ  r_2   Ydest  # Ycurr = Ydest ?
BRF  notXY
EXE  incrementX
MOVI 0
EXE  getlink
OUT  r_0
JMP  start

LABEL link1
EXE  incrementX
OUT  r_5
JMP  start

LABEL getlink
# This procedure returns link for one of the four directions
# It expects the encoding of the direction in r_0 as
# 0 = (x+1,y), 1=(x,y+1), 2=(x-1,y), 3=(x,y-1)
# NOTE: FOR NOW, 0 = x direction, 1 = y direction
# It returns the link in register r_0

# The routing table is store as a 4-bit entity for each node,
# 2 bits in each possible direction (there are only 2 such directions)
# The start of the routing table is at the beginning of the program.

MOV  r_3  r_0  # Save the (x,y) direction
WHR  r_9  r_4  # Get current (real) location
MOV  r_4  r_9  # Save the node number
MOVI 31  # Due to limitation on size.
SHL  r_9  r_0  # Get node parity
MOVI 29  # Due to limitation on size.
SHR  r_9  r_0  # "Parity" = which half of the byte
SHR  r_4  1  # Divide node by 2
APPENDIX A: Agent Program Listing

SUB r_4 1  # Get byte offset in routing table
LDB r_0  # Load the byte from routing table

# Extract the link by doing appropriate shifts
SHR r_0 r_9  # Get the half at the low end
SHR r_0 r_3  # Do this twice (2*0 or 2*1 shift)
SHL r_0 15  # Do twice, 4-bit operand limit :
SHL r_0 15  # Do twice, 4-bit operand limit :
SHR r_0 15  # r0 now holds the link
SHR r_0 15

LABEL incrementX
# This procedure increments the x coordinate by one
MOVI XcurrAddr
MLT r_0 2  # Make correct address
MOV r_3 r_0  # Temporary save
ADD r_1 1  # Increment x
STB r_3  # Store content of r0=x at Xcurr
RET

LABEL incrementY
# This procedure increments the y coordinate by one
MOVI YcurrAddr
MLT r_0 2  # Make correct address
MOV r_3 r_0  # Temporary save
ADD r_2 1  # Increment y
STB r_3  # Store content of r0=y at Ycurr
RET

A.6 Adaptive Mesh Routing Agent

# SPIDER program to send messages in a mesh. Adaptive DOR

# Assume that (Xcurr, Ycurr) and (Xdest, Ydest) are the mesh coordinates
# of the sender and the receiver nodes.
# Since the current coordinate change at every node, they are stored in the
# first two bytes after the routing table.
# It is assumed that they are no greater than 15 (i.e. 15x15 mesh).
# Xcurr ≤ Xdest, Ycurr ≤ Ydest

LABEL routeTable  # Start of routing table
DATA 0x88 0x88  # 2-9
APPENDIX A: Agent Program Listing

DATA 0x88 0x88
DATA 0xcc 0xcc
DATA 0x88 0x88
DATA 0x88 0x88
DATA 0x88 0x88
DATA 0x88 0x88
DATA 0x88 0x88
DATA 0x88 0x88
DATA 0x88 0x88
DATA 0x00 0x00
DATA 0x88 0x88
DATA 0xcc 0xcc
DATA 0xcc 0xcc
DATA 0x00 0x00
DATA 0x00 0x00

LABEL XcurrAddr
DATA Xcurr 0

LABEL YcurrAddr
DATA Ycurr 0

DEF dwnld id

LABEL start
MOVI XcurrAddr
MLT r_0 2
LDB r_0
MOV r_1 r_0
MOVI YcurrAddr
MLT r_0 2
LDB r_0
MOV r_2 r_0
TEQ r_1 Xdest
BRF notX
TEQ r_2 Ydest
BRF notY
OUT dwnld
END 1

LABEL notX
EXE incrementY
MOVI 1
EXE getlink
OUT r_0

LABEL notY
EXE incrementY
MOVI 1
EXE getlink
OUT r_0
APPENDIX A: Agent Program Listing

JMP | start

LABEL | notX
| TEQ | r_2 | Ydest # Xcurr < Xdest
| BRF | notXY # Ycurr = Ydest ?
| EXE | incrementX # else Xcurr < Xdest, Ycurr = Ydest
| MOVI | 0 # Encode (x+1,y)
| EXE | getlink # r0 will hold the link
| OUT | r_0 # Leave the node on (x+1,y)
| JMP | start

LABEL | notXY
| MOVI | 0 # Encode (x+1,y)
| EXE | getlink # r0 will hold the link
| MOV | r_5 | r_0 # Store the link
| LLD | r_0 # Get link load
| MOV | r_7 | r_0 # Store value in r7
| MOVI | 1 # Encode (x,y+1)
| EXE | getlink # r0 will hold the link
| MOV | r_6 | r_0 # Store the link
| LLD | r_0 # Get link load
| MOV | r_8 | r_0 # Store value in r8
| TGT | r_7 | r_8 # Which link is busier ?
| BRF | link1 # else
| EXE | incrementY
| OUT | r_6 # Go on (x,y+1)
| JMP | start

LABEL | link1
| EXE | incrementX # Go on (x+1, y)
| OUT | r_5
| JMP | start

LABEL | getlink
# This procedure returns link corresponding to one of the four possible
# directions to go in a 2D mesh
# It expects the encoding of the direction in r_0 as
# 0 = (x+1,y), 1=(x,y+1), 2=(x-1,y), 3=(x,y-1)
# NOTE: FOR NOW, 0 = x direction, 1 = y direction
# It returns the link in register r_0

# The routing table is store as a 4-bit entity for each node,
# 2 bits in each possible direction (there are only 2 such directions)
# The start of the routing table is at the beginning of the program.
MOV r_3 r_0            # Save the (x,y) direction
WHR r_9 r_4            # Get current (real) location
MOV r_4 r_9            # Save the node number
MOVI 31 r_9 r_0        # Due to limitation on size.
SHL r_9 r_0            # Get node parity
MOVI 29 r_9 r_0        # Due to limitation on size.
SHR r_9 r_0            # "Parity" = which half of the byte
SHR r_4 1              # Divide node by 2
SUB r_4 1              # Get byte offset in routing table
LDB r_0                # Load the byte from routing table

# Extract the link by doing appropriate shifts
SHR r_0 r_9            # Get the half at the low end
SHR r_0 r_3            # Do this twice (2*0 or 2*1 shift)
SHR r_0 r_3
SHL r_0 15             # Do twice, 4-bit operand limit :(
SHL r_0 15
SHR r_0 15             # Do twice, 4-bit operand limit :(
SHR r_0 15
RET                   # r0 now holds the link

LABEL incrementX
# This procedure increments the x coordinate by one

MOVI XcurrAddr          
MLT r_0 2               # Make correct address
MOV r_3 r_0             # Temporary save
ADD r_1 1               # Increment x
STB r_3                 # Store content of r0=x at Xcurr
RET

LABEL incrementY
# This procedure increments the y coordinate by one

MOVI YcurrAddr          
MLT r_0 2               # Make correct address
MOV r_3 r_0             # Temporary save
ADD r_2 1               # Increment y
STB r_3                 # Store content of r0=y at Ycurr
RET