AGENT-BASED DISTRIBUTED PARALLEL PROCESSING

Luciano de Errico

A dissertation for the degree
of Doctor of Philosophy

University of Surrey
Department of Electronic and Electrical Engineering
17th January 1996
Para Livia e Clarissa, pelo amor e suporte.
Para meu pai (in memoriam) e minha mãe.
Acknowledgements

I would like to thank my supervisor, Prof. Chris Jesshope, for introducing me to the subject of my research and for providing the professional guidance and assistance throughout the three years I spent in the University of Surrey. I am specially grateful to the Brazilian agency Fundação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), for providing the tuition fees and financial support during my postgraduate studies in the United Kingdom, and to my employer, Universidade Federal de Minas Gerais - Brasil, for granting a leave of absence and complementing my financial resources for the whole period.

My gratitude also goes to Paul Connolly, my colleague of PhD course, for the many long conversations about agent-based processing. I hope all those discussions have helped him as they helped me. I am also grateful to Slava Muchnick, for clarifying many aspects of language design, to Ivailo Nedelchev and Dima Barsky, for the prompt help with the intricacies of C and Unix, and to Nayef Baker, for the patience in explaining details of the bandwidth reservation problem.

For the support and friendship they provided during all the period of my research, I would like to thank all those in the Computer Systems Research Group, in special Tanya Vladimirova, Roger Peel, Sasha Shafarenko, Emma Blay, Marcel Odhiambo, Graham Wilford, Ian Ashman, Carlos Ramirez, Cruz Izu, and Mohammed Qadar.

Finally, I wish to express my gratitude to Francisco Junqueira Muniz for all the help he provided when I arrived in England with my family, and for the gratifying friendship that developed since then.
Abstract

This work concerns the design and prototype implementation of an agent-based parallel architecture for physically distributed systems. The generic goal is to combine the processing power of widely available, low-cost networks of workstations, providing parallelism inside single applications. The specific goal is to investigate ways of implementing agent-based parallel processing in distributed systems. In this context, an agent is a lightweight mobile process that can freely move in the network and execute when it reaches a processing node.

The Swarm architecture addresses these points by providing an abstract environment that can span many or all machines in the network. The environment is structured as a virtual machine, whose organisation and instruction set are detailed. Swarm is based on the idea of process flow, in which mobile concurrent processes can move and execute asynchronously in a distributed space consisting of data nodes. Each node is capable of permanently storing arbitrary information and references to other nodes, permitting the creation of persistent and distributed data structures in the environment. The main advantage is a flexible programming environment, which combines characteristics of the message-passing and distributed shared-memory approaches.

A subset of the Swarm architecture was implemented as a prototype, coded in C language for operation under the Unix environment, to study and evaluate the model. The prototype executed in a single workstation, simulating the Swarm abstract environment and permitting the validation of the proposed architecture and implemented mechanisms. Both the implementation and the evaluation procedure are explained and discussed.

Results suggest that agent-based processing is feasible in moderately- and tightly-coupled environments, and that the Swarm processing model can be successfully applied to local-area networks and massively parallel computing machines. In particular, applications that manipulate irregular and distributed data structures can benefit from the programming environment provided by the Swarm architecture. These comprise: symbolic processing (artificial intelligence and expert systems), distributed simulation, distributed databases, and intelligent networks.
Contents

1. Introduction ...................................................................................... 1

2. Distributed Parallel Processing ............................................................... 4
   2.1 Characterisation ...................................................................... 4
   2.2 Implementation Aspects ............................................................. 8
      2.2.1 Processing Elements ......................................................... 8
      2.2.2 Communication Sub-System ............................................. 9
      2.2.3 Operating System ............................................................ 11
      2.2.4 Support for Programming ............................................... 13
   2.3 Summary ..................................................................................... 17

3. Approaches to Distributed Parallel Processing .......................................... 19
   3.1 An Overview ................................................................................ 19
   3.2 The Message-Passing Approach .................................................... 21
   3.3 The Shared-Memory Approach ..................................................... 23
   3.4 The Agent-Based Approach ......................................................... 27
   3.5 Summary ..................................................................................... 32

4. The Swarm Architecture ..................................................................... 34
   4.1 The Process Flow Model .............................................................. 34
   4.2 Swarm Abstract Machine ............................................................ 37
      4.2.1 General Organisation and Execution Model ....................... 38
      4.2.2 Data Nodes ...................................................................... 42
      4.2.3 PM Access and Root Nodes .............................................. 44
      4.2.4 Service Nodes ................................................................... 45
      4.2.5 Access Point and Console Node ....................................... 46
      4.2.6 Tasks and Agents ............................................................... 48
   4.3 Swarm Instruction Set ................................................................... 52
      4.3.1 Workspaces and Built-In Structures ................................... 54
      4.3.2 Instruction Formats ............................................................. 58
      4.3.3 Arithmetic/Logical Instructions .......................................... 59
      4.3.4 Data Transfer Instructions ............................................... 59
      4.3.5 Control-Flow Instructions ................................................. 60
      4.3.6 Agent-Control Instructions ................................................ 61
# 5. The Swarm Prototype Implementation

5.1 Prototype Implementation

5.1.1 The Routing Module

5.1.2 The Processing Module

5.1.3 The Access Point

5.1.4 Termination Detection

5.2 Example Application: Bandwidth Reservation

5.2.1 Description of the Problem

5.2.2 Description of the Algorithm

5.2.3 Execution Tests

5.3 Performance Tests

5.4 Summary

# 6. Discussion

6.1 Evaluation of the Architecture

6.1.1 Swarm Agent-Based Processing

6.1.2 Support for Data Structures

6.1.3 Support for End-User Interfaces

6.1.4 Additional Extensions

6.2 Evaluation of the Prototype

6.2.1 Performance

6.2.2 Exception Detection and Handling

6.2.3 User Interface and Debugging

6.2.4 Storage Management

6.2.5 Features Not Implemented

# 7. Conclusions

7.1 Review of the Research

7.2 Applications of the Research

7.3 Suggestions for Future Work

# 8. Bibliography
List of Figures

2.1 Design space of computing systems ................................................................. 5
2.2 Structure of a microkernel operating system ................................................. 11
3.1 Different levels for DPP ............................................................................. 20
3.2 The Linda System ................................................................................. 25
3.3 The Telescript System ............................................................................. 31
4.1 Links in Wave and in Process Flow ......................................................... 36
4.2 Wave and Process Flow Paradigms ......................................................... 36
4.3 Swarm Abstract Machine ........................................................................ 39
4.4 A multicomputer implementation of Swarm ............................................. 40
4.5 A distributed implementation of Swarm .................................................. 41
4.6 A SAM data node .................................................................................. 43
4.7 Swarm service nodes ............................................................................. 46
4.8 A Swarm agent ..................................................................................... 48
4.9 Different workspaces in Swarm .............................................................. 50
4.10 Condition synchronisation for agents in Swarm ....................................... 52
4.11 Swarm instruction set .......................................................................... 53
4.12 Agent built-in structures ....................................................................... 55
4.13 Nodal built-in structures ....................................................................... 57
4.14 PM built-in structures ........................................................................... 58
4.15 Swarm instruction formats ................................................................... 58
4.16 Console node services .......................................................................... 63
4.17 Example program (in pseudo-code) ....................................................... 65
4.18 A connected graph ................................................................................ 65
4.19 Example program in assembly language ............................................... 67
5.1 Structure of the prototype Swarm system ............................................... 71
5.2 Normal agent formats ........................................................................... 73
5.3 RM internal organisation ....................................................................... 74
5.4 PM internal organisation ....................................................................... 75
5.5 PM execution routine ............................................................................ 77
5.6 AP command menu .............................................................................. 78
1. Introduction

The limitations imposed by conventional, single-processor computer architectures prompted research on alternative solutions that could improve processing performance. The main product of these efforts was the appearance of the two different approaches known as distributed and parallel processing. Distributed processing systems offer a loosely-coupled environment that emphasises resource sharing and improved reliability. Parallel processing systems were developed as tightly-coupled environments with many processors dedicated to fast execution of single complex problems. Recent advances in technology opened the opportunity of combining the advantages of both models and perform parallel processing on physically distributed systems. This approach is named here distributed parallel processing (DPP), and its main motivation is to achieve a processing power similar to supercomputers and parallel machines using available, low-cost networks of workstations.

Different solutions for DPP have been proposed by researchers from academic and private institutions. Some adopt a more direct approach, using message-passing and completely exposing the distributed nature of the environment to the programmer [Geist et al. 1994, Gropp et al. 1994, Mainwaring and Culler 1995]. Others offer more abstract environments, with different levels of support for data sharing and parallel process coordination [Carriero and Gelernter 1989, Bal et al. 1992, Jul et al. 1988]. The former are easy to implement but more difficult to program. The latter provide a better programming environment, at the expense of a more complex implementation. Agent-based processing is one technique that combines many of the advantageous features of these approaches. It completely includes message-passing, adding support for remote execution, and offers a shared data space that naturally maps on the underlying distributed system.

An example of an environment for agent-based processing is the Wave system developed by Sapaty [1992]. Wave organises information in a semantic-network style, by constructing a graph where each node holds arbitrary data (representing a value, concept or identifier) and is connected to other nodes by labeled links (representing a relation between the nodes). Mobile processes, called waves, are agents that carry code and data, and execute instructions when they reach a node. The Wave system provides parallel processing by having many waves executing simultaneously at different nodes. Agents execute code written in a corresponding
Wave language, which offers a set of primitives for data manipulation and agent coordination. Wave targets parallel processing in very loosely-coupled environments, such as wide-area networks (WANs) and the Internet, and an experimental system was constructed for Unix and TCP/IP based computer networks [Borst 1992, Sapaty and Borst 1994].

Agent-based processing is a relatively new approach and many of its possible variants have not been adequately studied or developed. For example, starting from the paradigm proposed by Wave it would be possible to derive a simpler model for agent-based DPP in moderately- and tightly-coupled systems, like local-area networks and multicomputer machines. The resulting environment would be an intermediate between message-passing and more abstract models (e.g. shared-structures and object-based), offering a reasonable level of abstraction to facilitate programming without incurring in a complex implementation or an excessive number of primitives. This would represent a compromise between user abstraction and implementation effort not found in most currently available designs. Additionally, the intrinsic mobility of agents would provide extra flexibility and could be explored in different ways, e.g. to reduce communication traffic in the system or to improve load balancing. Finally, by adopting an intermediate format for the executing code, processing could be naturally extended to support heterogeneous and open systems (as already implemented in the Wave [Sapaty and Borst 1994] and the Telescript™ [White 1994] packages).

Driven by these motivations, this thesis aims to study the application of agent-based processing for the development of a programming environment for parallel processing in moderately- and tightly-coupled systems. In order to achieve this and experiment with agent-based processing, it was decided to develop a corresponding architecture and implement a prototype that would permit the evaluation of the design.

After this introductory exposition of motivations and presentation of the research theme, the remaining chapters of this dissertation are organised as follows:

- Chapter 2 characterises distributed parallel processing and discusses the major aspects related to the design of DPP systems.

- Chapter 3 reviews the existing environments for distributed parallel processing, classified according to the main adopted paradigm.
• Chapter 4 presents the Process Flow paradigm for distributed parallel processing and details the agent-based parallel architecture developed in this thesis.

• Chapter 5 details the implementation of a prototype constructed in a UNIX environment and presents an example program for agent-based processing using the prototype.

• Chapter 6 evaluates the architecture and the prototype implementation based on the results obtained, and makes comparisons between the approach used and the ones adopted in similar works.

• Chapter 7 reviews the main results of the research, discusses possible applications, and presents suggestions for future work.
2. Distributed Parallel Processing

This chapter defines the basic concepts and terminology adopted in the thesis. Section 2.1 introduces distributed parallel processing, places it in context, and presents its main advantages and disadvantages. Section 2.2 discusses the major aspects to be considered when designing distributed parallel processing systems.

2.1 Characterisation

In the context of this thesis, a distributed parallel processing (DPP) system is one that permits parallelism within single applications, but executes on a physically distributed environment. In other words, the system appears to the user as a parallel computer but is physically structured as a collection of independent machines connected by a communication service. This characterisation is mainly based on practical considerations, given the lack of consensus in the technical literature about the definition of parallel and distributed systems and the distinction between them.

Authors from the parallel processing area [Almasi and Gottlieb 1994, Hwang 1993] adopt the possibility of parallelism within a single application as the main characteristic of a parallel system. Authors from the distributed processing area [Tanenbaum 1995, Coulouris et al. 1994, Bacon 1993] tend to classify as distributed any system with more than one (central) processing unit, considering parallel systems as a special case (a tightly-coupled variant) of distributed systems. Here, the intention is to emphasise both the logical and the physical aspects. Therefore, a parallel system is understood as a collection of many processing units, with shared or private memory, that cooperate for solution of single problems and are tightly connected by a fast (delays of a few μs) communication sub-system. Conversely, a distributed system is characterised as many independent computers (cpu + memory + I/O), distributed throughout a restricted or large physical area and connected by a moderately fast or slow (delays of 100s of μs to ms) communication sub-system. In this case, the slower communication speed and lack of adequate support from software layers leads to a processing style where applications run locally at a computer and keep their interaction with other distributed system components to a minimum.
The organisations discussed above can be accommodated in the generic design space of Figure 2.1, according to their degrees of physical and logical coupling. Conventional computer systems, that follow the sequential von Neumann model [Burks et al. 1987], only permit coarse-grained pseudo-parallelism (via multiprogramming or multitasking) and occupy a point in the lower left corner. Processor-array systems, corresponding to Flynn’s SIMD category [Flynn 1966], offer centrally controlled, fine-grained parallelism and define the next region. MIMD designs [Flynn 1966] are divided into multiprocessors (with physically shared memory) and multicomputers (with physically private memory), and characterise systems with many independent processors (no central controller). They can provide medium to fine-grained parallelism and represent two degrees of logical coupling on top of similar moderately distributed hardware. Conventional distributed systems have a loosely-coupled logical organisation and a high degree of physical distribution, resulting in coarse-grained parallelism. Finally, distributed parallel processing (DPP) systems are expected to deliver medium-grained (or even fine-grained) parallelism in a physically distributed organisation and corresponds to the lower right region of the design space in Figure 2.1.
The development of DPP systems can be justified by the advantages they can offer:

- **Flexibility**: as most of the structure of a DPP system is going to be in software, the environment can be more easily tailored to particular needs, either statically, reflecting the main type of applications to be executed, or dynamically, to accommodate variations in the workloads. Additional physical computers can also be added to the system, increasing the available processing power with minimum effort. Another possibility is the co-existence of different virtual environments, like different paradigms or even distributed applications running together with parallel ones [Anderson et al. 1995].

- **Better use of resources**: by grouping the resources (processing power, memory and I/O) of all physical computers in a common pool, the system can do a better management by allocating them according to the needs of each particular application, regardless of its location [Litzkow et al. 1988, Carriero et al. 1994]. This extends the concept of processor pool introduced in distributed systems [Tanenbaum 1995, Coulouris et al. 1994].

- **Improved performance**: by combining the processing power of many workstations, a DPP system can possibly rival existing parallel machines and supercomputers. This processing power can then be used for parallel execution within applications, improving their performance [Anderson et al. 1995].

- **Lower cost**: the large market for workstations and local-area networks (LANs) has led to continuous improvements in technology and decreasing in prices. As result, these components offer a much better price/performance ratio than the one offered by parallel machines [Anderson et al. 1995]. Not only system upgrades (by addition of more resources or more computers) are more accessible, but maintenance is widely available and much cheaper than in the case of parallel machines.

- **Support for heterogeneous configurations**: adopting a practice already in use on distributed systems, a DPP environment can support physical computers of different architectures. Proper layers of software can be introduced to hide the differences, providing a uniform layer for application processing.

However, while a DPP system combines many advantages of the parallel and distributed approaches, it also exhibits many of their disadvantages:
• **Experimental environment**: much of the technology is still in the research phase and is not well understood. In contrast, distributed systems have well-defined standards [Coulouris et al. 1994, Bacon 1993] and many parallel machines are commercially available in the market [Almasi and Gottlieb 1994, Hwang 1993] offering tested solutions for parallel processing.

• **Lack of support for programming**: in common with parallel systems, there is no general programming model. Reflecting the experimental stage of the technology discussed above, many of the programming environments for DPP are currently in research. Nevertheless, some packages, like PVM [Geist et al. 1994] and Linda [Carriero and Gelernter 1989, Mattson 1994], have already reached a stable state and been successfully used with applications from different areas.

• **Fast communication**: to be feasible, a DPP system requires high-speed low-overhead communication, not normally present in a distributed environment. Even with the higher bandwidths offered by new communication networks [Newman 1994, Boden et al. 1995], performance is still low due to the overheads imposed by standard protocols such as TCP/IP.

• **Support from the operating system**: most of the existing operating systems for networked workstations follow a monolithic architecture, offer limited flexibility, and limit processing granularity and resource sharing in the network. Recent systems have started to offer improvements [Eykholt et al. 1992], mainly in the form of multithreading, but they still do not provide the support needed by DPP.

• **Security and privacy**: more seriously than in conventional distributed systems, the greater flexibility and resource sharing present in DPP systems compromise privacy and security. Some developers accept this as a characteristic of the environment [Anderson et al. 1995]. Others implement different mechanisms to prevent unauthorised access [White 1994, Gosling and McGilton 1995].

The next section discusses how to achieve the advantages and overcome the disadvantages of distributed parallel processing, in the light of presently available technology.
2.2 Implementation Aspects

Essentially, a DPP system is a parallel machine that requires two main hardware components: processors and a communication network. Complementing the hardware, software layers are necessary in the form of an operating system (or a collection of them) and programming environments and languages for both the end-user and the system-level programmer. These four elements (processing elements, communication sub-system, operating system, and programming support) are the essential aspects to be considered when designing a DPP system. The sections that follow detail each main aspect and examine possible alternatives for implementation.

2.2.1 Processing Elements

A common justification for the development of parallel processing machines is that by using thousands of off-the-shelf microprocessors it is possible to have supercomputer power at a fraction of the cost [Almasi and Gottlieb 1994, Hwang 1993]. The parallel machine would benefit from the lower cost of high-volume production of its main component, the processor element.

A similar argument is now being used in favour of DPP [Anderson et al. 1995, Bell 1992]. The popularity of workstations stimulated mass production and development of technology, leading to a continuous decrease in price and increase in performance of these machines. Parallel computers and supercomputers, in turn, have restricted market and require greater investment in technology, resulting in higher prices. Anderson et al. [1995] claim an improvement in the price/performance ratio of around 80% per year for workstations compared with 20 to 30% per year for supercomputers. In the same article [Anderson et al. 1995], the authors provide a coarse comparison between the acquisition price of similar configurations of a parallel computer (a Thinking Machines CM-5 with 128 40-MHz SuperSparc processors and a fat-tree interconnection network) and a network of workstations (32 Sun Sparcstation-10, with 4 40-MHz SuperSparc processors each, plus ATM switches for interconnection). The result is that the cost of the network of workstations is less than 50% of the cost of the parallel computer, for an identical global configuration of memory, disks, and terminals.
2.2.2 Communication Sub-System

One main practical disadvantage of distributed systems is the low performance of their communication networks (compared with their counterparts in parallel machines). Two major factors affect this performance: the raw capacity of the network, its bandwidth, and the communication delay, the latency. New technology produced a significant increase in the available bandwidth. First generation LANs were limited to a total maximum bandwidth of approximately 10 Mbps (million bits per second) offered by the bus-based Ethernet [Metcalf 1976] and similar networks. This was later extended to 100 Mbps per ring on dual-ring FDDI (Fiber Distributed Data Interface) networks [Ross 1986]. The recent ATM (Asynchronous Transfer Mode) networks [Newman 1994] permit arbitrary network topologies (not necessarily bus- or ring-based) and offer up to 155 Mbps, 622 Mbps or even a future 2.5 Gbps per link, depending on the version (with a total cross-section bandwidth that can be many times the link bandwidth due to the chosen topology). In addition, new alternatives are emerging, like the Myrinet [Boden et al. 1995] that applies in local-area networks an interconnection network technology developed for parallel machines. Myrinet developers claim a data rate of 380 Mbps per link in a raw speed test and high performance per unit cost.

Latency is the time taken for a message to be delivered to its destination. The latency seen by an application aggregates many delays caused by different components of the communication sub-system. Communication in a physical medium (usually copper wire or fiber optics, for LANs) is limited by the speed of light in that medium. The circuitry present in a real network (switches, routers) adds an extra delay component to the physical propagation. Layers of protocol, used to avoid loss of data during communication, insert an extra time to process the message known as software overhead. An extra drawback, if the protocol is executed by the communicating processors, is that it consumes cpu time that could be spent in computation.

The propagation time in the physical medium depends on the distance between communicating points and cannot be changed. One solution is to reduce extra delays in the communication path. This can be achieved by re-engineering the circuitry and internal algorithms or using faster electronic components. Another solution is to adopt a small overhead protocol. For the commonly used TCP/IP protocol, a recent experiment at the University of California at Berkeley [Anderson et al. 1995] shows 456 us of protocol
overhead plus unloaded network latency for a single message in the Ethernet, and 626 us for an ATM network. In contrast, a prototype implementation of Active Messages, a leaner communication layer developed at Cornell University [von Eicken et al. 1995], achieves about 50 us of round-trip (message + response) delay, including a 10 us ATM switch latency in each direction. A third solution is to use latency-hiding techniques. Adequate mechanisms for caching of code and data can be effectively used to hide latency, as demonstrated by the Stanford DASH project [Lenoski et al. 1992]. Local concurrency can also be used to hide latency, by overlapping communication and computation, as in the MIT J-Machine [Dally et al. 1992], or by implementing fast context-switching among many lightweight processes, as in the Tera architecture [Alverson et al. 1990].

The higher communication speed provided by the ATM and the high connectivity permitted by the Internet motivated research into processing using that environment. LANs connect computers that are no more than a few kilometres apart. Wide-area networks (WANs) connect systems in different cities or countries. The common technology used for WANs is leased telephone lines, with maximum bandwidth around 1.5 Mbps in most cases. ATM networks [Newman 1994] can push this limit to 155 Mbps for normal lines and reach speeds of about 1 gigabit per second in major trunks. An internetwork is a connection of many distinct networks (which can be LANs or WANs), using interconnecting elements like gateways, routers and repeaters. The popular Internet [Krol 1994] is an internetwork that evolved from an original project of the USA Department of Defense, in the late 1960s and the 1970s, to link all organisations involved in military research projects. The scheme gradually grew, including other research and academic sites, until it reached the present configuration that connects networks and computers worldwide. Internet-based systems have to face many practical problems:

- security and privacy are almost non-existent;
- the environment is continuously changing, with hosts and services being connected and disconnected dynamically;
- the environment is highly heterogeneous;
- information is very dispersed and disorganised;
- communication is error-prone (requiring protocol layers like TCP to recover from errors) and relatively slow (although this is about to change in the medium-term).
2.2.3 Operating System

The implementation of DPP (distributed parallel processing) can be facilitated if the operating system offers adequate support. The operating systems currently used in local-area networks offer a very restricted model that limits sharing and exposes the multiplicity of machines to the user. Research in the area produced experimental distributed operating systems [Tanenbaum 1995], which provide users with a transparent access to resources, independently of their location (local or remote). The goal is to provide transparency on all levels, automatically performing replication and migration of data, or even processes, if necessary.

Different approaches to the internal organisation of an operating system, like microkernels, have also been proposed. Conventional operating systems are structured as a single monolithic kernel that provides all the necessary services like process management, memory management, file system and communication. Microkernels, instead, have just a minimal functionality: a small number of process support mechanisms, some memory management, interprocess communication mechanisms, and some hardware level I/O. All other services are provided by servers located at the application level (Figure 2.2). This design enhances flexibility, as it is relatively easy to add, remove or modify a server. In addition, different sets of file servers can be present at the same time in the system, providing different programming environments (e.g. Unix-like, MS Windows-like) to the users. Another advantage, in a way transparent to the user, is that servers can be located anywhere in the system.

![Figure 2.2 - Structure of a microkernel operating system](image_url)
Modern operating systems, either of monolithic or microkernel type, adopt more refined process management mechanisms like **multithreading**. In a multiprogramming or multitasking environment, many processes run concurrently in a single processor (or even a pool of processors), keeping separate address spaces and private resources (like open files). When a process is suspended, its associated context must be saved, which includes pointers to memory areas, tables, and the contents of the program counter and other processor registers. This implies high overheads for context switching. Multithreaded environments keep the process (commonly called a **task**) as the unit of resource allocation (address spaces, stacks, files) for program execution, but define **threads** as the units of scheduling. Threads belonging to the same process share all resources defined by the process, reducing the context switching overhead. For this reason, threads are known as **lightweight** processes, in contrast to the traditional **heavyweight** processes. Multithreading gives more flexibility and allows a finer degree of parallelism inside an application.

Examples of distributed operating systems are Locus [Popek and Walker 1985] and Sprite [Ousterhout et al. 1988]. Microkernel designs can be found in the Mach [Black et al. 1992] and Chorus [Rozier et al. 1988] operating systems. Particularly, Amoeba [Mullender et al. 1990] is a distributed operating system that uses a microkernel structure. Recent versions of monolithic operating systems, like Sun Solaris [Eykholt et al. 1992], are offering threads at both user and kernel levels.

Another area for improvement is extended resource sharing in the network. Conventional distributed systems have a limited level of resource sharing, usually restricted to file systems and some peripherals (e.g. printers). However, as networked computers work autonomously, a large amount of processor time is lost in idle workstations. Recent research initiatives like Condor [Litzkow et al. 1988] and Piranha [Carriero et al. 1994] aim at using these idle processor cycles to increase the processing power of the system as a whole, by transparently finding idle workstations and executing processing in them.

An even higher level of resource sharing is proposed by Anderson et al. [1995]. In their scheme, memory space, disk space and processor time in all computers can be transformed into a pool of resources transparently shared. Using these combined resources, virtual memory and file caching can be extended to use network available RAM, a distributed array of networked disks can be implemented, and parallel processing can be achieved by using processor time in idle workstations. Their proposal assumes a cluster of workstations...
connected by high-speed LAN, a low-latency communication protocol and a special layer of software on top of the original operating systems.

Extensive resource sharing (RAM, disks and processors), however, presents many practical problems. In a distributed system, depending on the system configuration, a workstation or personal computer can continue processing stand-alone in case of the failure of some shared resource. With extensive resource sharing, fault-tolerance becomes more complicated and may involve different solutions requiring replication. Extensive resource sharing also leads to weaker security and privacy. In a closed organisation, this can be managed by controlling physical access to computing facilities. In open environments, like universities and the Internet-connected environments, a solution is more difficult as it must provide security without excessive restrictions to access the system.

2.2.4 Support for Programming

The style presently known as imperative programming originated from the technological limitations of early computers, in which users have to precisely specify the steps to solve the problem and translate this sequence into machine language before submitting it for execution, with the extra care of re-using memory positions, as memory space was very limited. Later advances in software and hardware alleviated these problems, but the basic programming style persisted and is still the most popular. A strong reason for this, apart from a natural resistance from users against changes, is that imperative programming permits fine tuning of the program with consequent increase in flexibility and execution performance.

As a consequence, a shared memory environment is preferable when writing parallel programs in the imperative style. A message passing environment, although easier to implement, is more difficult to use as it forces the division of the problem into small communicating parts. Mechanisms for shared memory were first implemented in parallel computers, making use of hardware MMUs (memory management units) and fast communication [Hwang 1993, Tanenbaum 1995]. The uniform memory access model (UMA) applied to multiprocessors and offered, for each processor in the system, an equal access time to all memory positions, regardless of the fact that the memory can be composed of many individual modules. In the nonuniform memory access model (NUMA), a global
address space was physically divided into memory modules distributed to the processors. Access time to a local module was better than the access time to a remote module and this characteristic was visible to the programmer.

In contrast, the approach adopted in multicomputers and distributed systems is the no remote memory access model (NORMA): there is no global memory and all local memories are private (accessible only by the local processor). In this case, a software layer must simulate the effect of a distributed shared memory (DSM). The first effort in this direction was IVY [Li 1988], which provides a virtual memory with pages distributed among the computers. Later research resulted in the shared variable and shared data structure approaches. The former permits sharing of individual variables, instead of whole pages, but obliges the user to annotate those variables and employ critical regions to protect them. The latter is based on the sharing of data structures, which results in a more abstract programming model and permits optimisations to be done without intervention from the user. Distributed shared memory approaches are discussed in more detail in Chapter 3.

The investment made in installed software systems and user familiarity with a specific programming style (like imperative programming) are two major factors against drastic changes. Another problem, which directly affects DPP, is that although parallel programming is becoming more accepted [Almasi and Gottlieb 1994], there is a lack of standards and the cost of moving to a parallel processing approach may not be attractive, even if an existing distributed system can be used as DPP for free.

One solution to attract users is to offer improved performance while keeping a familiar interface for the programmer. This can require automatic parallelisation of conventional sequential code (the dusty decks), at the expense of more complexity in the system and decrease in performance. An alternative solution is to change radically the programming style, offering to the user an interface that is more manageable, better mapped to a specific area of application, or more natural (i.e. closer to the human level). The functional, logic and object-oriented paradigms follow this strategy.

Functional and logic programming use a declarative style [Watt 1990, Ambler et al. 1992], where the programmer is concerned with specifying what he wants to do, not how it can be done. The latter is a matter for the language translator (interpreter or compiler) and runtime system. The advantage for parallel programming (and for DPP) is the possibility of automatically detecting and extracting parallelism. The main disadvantage, at the present, is

The object-oriented approach provides a highly structured form of programming derived from the imperative style. Benefiting from being closer to the imperative paradigm, the object-oriented approach is gaining more acceptance and is considered advantageous from the point of view of software engineering and maintenance. Its basic concept is the encapsulation of data structures together with a set of related operations, forming modules known as objects which can only be accessed by an invocation of an operation defined in its interface. Although there is no consensus about the remaining characteristics of object-oriented approaches, the denomination is reserved for systems that include classes (templates for creation of new objects) and inheritance (mechanisms to create new classes from existing ones) as additional abstractions [Wegner 1987]. However, as shown by Wegner [1987], in practice many of the systems claiming to be object-oriented lack one or more of the above described characteristics or substitute them by alternative ones, like strong typing, data abstraction, concurrency and persistence. Therefore, the loose concept of object-based systems is more adequate as an umbrella category that includes all these variations.

Representatives of the object-based approach for distributed parallel processing are the Emerald system [Jul et al. 1988], the COOL base layer [Lea et al. 1993], and the Actor model [Agha 1990, Agha 1986]. Emerald emphasises object mobility and is close to the agent-based approach which is the topic of this thesis. Emerald is detailed in Chapter 3. COOL (Chorus Object-Oriented Layer) is a support layer for object-based languages that works atop the Chorus microkernel operating system, providing a kind of global address space and generic primitives for object management. Its reported performance is not impressive, though, being one order of magnitude more expensive than traditional remote procedure call [Lea et al. 1993]. The Actors model represents a kind of hybrid between the
object-based and functional paradigms and leads to a programming style that does not use variables. An actor is an active object that encapsulates processes, where each process defines a behaviour. Each actor has a unique address in the system and can interact with other actors by message-passing. When it receives a communication, an actor can perform one or more of the following actions: send messages to other actors whose address it knows; create new actors; or specify a replacement behaviour that will handle the next communication. Changes of state happen by the creation of a new behaviour (i.e. a new process). Actors emphasise concurrency within and among objects, but it is not object-oriented as it does not have classes or inheritance. Given its characteristics, Actors seems to suffer from implementation problems similar to functional systems and research has been directed to develop different languages in multicomputer (MIMD) environments [Agha 1990]. Updated information about the current research in object-based languages and models can be found in [Ciancarini et al. 1995].

Agent-based processing is yet a different approach, which originates from the interest in exploring processing in WANs and the Internet. Search engines [Krol 1994] that navigate the Internet looking for specific information are some of the first outcomes of this approach. Extending the concept of search engines, mobile software agents are being proposed as an efficient way to do processing in the Internet [Kotay and Kotz 1994, White 1994, Wayner 1995, Sapaty and Borst 1994]. In this approach an agent is a mobile process, carrying its own code and data, that visits distributed sites and performs processing at them. This processing may consist of collecting information (like the search engines) or making a transaction like purchasing merchandise. Some agent-based packages, like Telescript™ [White 1994], are already being commercialised while others, like Wave [Sapaty and Borst 1994], are still in the research phase.

Another approach is the Java™ environment developed at Sun Microsystems [Gosling and McGilton 1995]. The proposal is to use the Internet as a public computational resource. Transparently to the user, or under his command, applications would be available for download by demand, executing at the local user machine, or for remote execution with subsequent forwarding of results. The developers argue that this brings flexibility, as the user could be able to allocate as much computing power as necessary for the intended task, and reduces investment in hardware, as the only essential equipment at the user end would be a modem and a terminal with little or no disk storage.
Above all, no extensive experimentation has yet been conducted to evaluate processing in the Internet and the best way to do this is still unknown. Agent-based packages are analysed in more detail in the next chapter.

2.3 Summary

Conventional distributed systems offer good computational power and low cost, but lack integration. Parallel systems and supercomputers present a high degree of coupling and transparency, but are technologically more difficult to construct, use non-standard software and have a higher cost. A recent trend is to combine the advantages of both models, by using a combination of faster communication and improved software, to perform parallel processing on physically distributed systems. This approach is called distributed parallel processing (DPP), and its main motivation is to achieve a processing power similar to supercomputers and parallel machines using available, low-cost network of workstations.

Recent technical advances have permitted the development of DPP systems. Networks of workstations are widely available at the present and workstations are becoming increasingly more powerful and relatively cheaper. New generations of local-area networks and low-overhead protocols are under development, to permit fast communication with lower latencies. Additionally, wide-area networks and internetworks (e.g. the Internet) have emerged as new environments that can extend the scope of conventional processing. Progress in software technology has been focusing on more efficient and flexible operating systems and extended resource sharing in physically distributed environments. Research in programming paradigms has targeted languages and systems that are more expressive and easier to use. The challenge is how to combine these recent advances and overcome their technical difficulties, producing a final environment that is comfortable to use, reasonably efficient, flexible, and reliable (ensuring security, privacy and fault-tolerance).

This thesis addresses the programming support for DPP. Existing systems that permit good performance offer a lower level of abstraction. Available DPP environments that provide a better abstraction require more complex implementations. This work concerns the development of a programming environment for DPP which combines a good level of abstraction and performance with a simpler implementation. The technology chosen was
agent-based processing. The agent-based approach is detailed in Chapter 3. Chapters 4 and 5 present the architecture and prototype developed in this thesis.
3. Approaches to Distributed Parallel Processing

This chapter reviews existing environments for distributed parallel processing (DPP), classified according to three main paradigms: message-passing; shared-memory; and agent-based. Message-passing and shared-memory presently dominate the field, supported by previous experience from the parallel and distributed processing areas. Agent-based is a new approach that can offer a better compromise between the offered level of abstraction and the complexity of implementation. Section 3.1 gives a general introduction to DPP environments. Sections 3.2 to 3.4 describe each category and discuss typical implementations, considering their advantages and limitations.

3.1 An Overview

Ideally, a DPP (distributed parallel processing) solution should provide the user with an abstract environment that hides physical characteristics (like distribution) and facilitates programming (e.g. by using high-level abstractions or automatic parallelisation). These topics, however, are still a subject of research and no definitive solution has yet been found. The diversity of approaches adopted in currently available DPP environments [Geist et al. 1994, Gropp et al. 1994, Li 1988, Bennet et al. 1990, Carriero and Gelernter 1989, Bal et al. 1992, Jul et al. 1988, Sapaty and Borst 1994, White 1994] reflects this situation.

Depending on the level of abstraction, DPP environments can be offered as language packages, library packages (toolboxes), or low-level base layers. As shown in Figure 3.1, the distinction is not that sharp, as these classes overlap and intermediate layers can be considered part of the applications (languages) or part of the operating system. A language package directly permits end-user programming and offers a consistent set of constructs and rules, which can be similar to some existing languages or can be completely new. A library package supplements an existing language (usually C or Fortran) by providing procedures that handle the new environment. These procedures are linked to the main code during compilation and may access special runtime systems that support the DPP environment. Base layers provide a
set of primitives (low level operations), which can be used as an extension to the underlying operating system or as a new layer that defines a different virtual machine organisation. Whichever approach, the purpose of base layers is to support high level language translators and runtime systems.

Figure 3.1 - Different levels for DPP

Another dimension in the design of DPP environments is the adopted paradigm. The paradigm defines the general scheme for an abstraction and is, therefore, fundamental to any implementation. Reflecting the current state of technology for support of parallel programming, the existing approaches can be classified under two main categories: message-passing and shared-memory. Message-passing has the advantage of mapping easily on the underlying, physical structure of the system: a number of processing units connected by a communication network. However, it imposes a different programming style. There is no common address space and the application must be divided into distinct modules that communicate and synchronise by exchanging messages. In contrast, shared-memory offers the same programming environment of conventional computing systems and is, therefore, considered easier to use. The drawback, though, is a more complex implementation that must manage to create an illusion of a shared memory space on top of a set of disjoint address spaces, each one belonging to a different autonomous computer.

Aiming at a compromise between these two extremes, agent-based approaches are appearing as alternative solutions for processing in distributed environments. By permitting not only data, but computation as well to move between the processing nodes, extra flexibility is obtained, compared with message-passing. As a process is allowed to move to the location that
holds the data, memory can be shared in an easier way. The following sections analyse the three different approaches to DPP - message-passing, shared-memory, and agent-based - discussing representative implementations and considering their positive and negative aspects.

3.2 The Message-Passing Approach

The message-passing approach offers the most direct solution to DPP. The programming style is similar to that of a conventional distributed system: relatively loosely-coupled, large grain modules that communicate by exchanging messages. The major difference is that the network of computers is managed as a single parallel computational resource, with built-in support for process management and facilities for the programmer (like debugging or performance monitoring). The user has access to the system by a library of procedures that he can use from his preferred programming language. Nevertheless, the library provides a low level interface that places on the user the burden of dividing the program into modules, allocating each module to a processor, and directly manipulating the message-passing (e.g. assembling buffers for transmission).

One of the most popular environments for DPP is the Parallel Virtual Machine (PVM) [Geist et al. 1994], a package from the Oak Ridge National Laboratory. PVM has two main components: a user library for process management, communication and synchronisation; and a daemon process that must be installed in each computer, defining the configuration of the virtual machine. A parallel computation is modelled as a collection of processes that runs a parallel virtual machine and communicates by passing messages. Prior to execution, the user must configure his virtual machine by running an initialisation program that installs a copy of the daemon in each specified computer. The user program is then linked with the library and, during execution, the corresponding processes communicate through the daemons (local process - local daemon - remote daemon - remote process). Communication is via UDP/IP protocols, but the user can switch to TCP/IP using library calls. Communication buffers must be explicitly managed by the programmer, who is responsible for their creation, packing and unpacking. Recent versions of PVM have extra features like dynamic configuration of the virtual machine, user-defined process grouping for broadcast and synchronisation barriers, signal handling, and tools for control of the virtual machine, visual programming, debugging and performance monitoring.

21
The main advantage of PVM is its support for heterogeneous environments. PVM works with networks of workstations, MIMD parallel computers, vector supercomputers or SIMD machines. This portability, combined with free distribution policy, has attracted many users, mainly from scientific and engineering areas, resulting in a comprehensive library of parallel numerical code. Incidentally, many scientific and engineering applications map well into the model offered by PVM, and their users are generally familiar with parallel programming due to previous experience with parallel computers. Detailed information about PVM can be found in [Geist et al. 1994]. A comparison of PVM with two other message-passing packages (P4 and TCGMSG), and a shared-data structures package (Linda), is detailed in [Mattson 1995]. In a recent development, a group including major parallel computer vendors and members of the research community defined a standard called MPI (Message Passing Interface) [Gropp et al. 1994], that represents a consensus in the area and should replace PVM in the medium term.

Different extensions to the basic concept of message-passing are also under research. The Active Messages interface [von Eicken et al. 1992, Mainwaring and Culler 1995], developed by a group from the University of California at Berkeley, permits messages to transfer both data and control. Active Messages works in a SPMD (Single Program - Multiple Data) environment: a system service (external to the message layer) creates and distributes a copy of the program to all participating nodes, along with process IDs that will identify each process (one or more per processing node) executing that program. The Active Message layer then provides primitives to store or get data from remote nodes and for remote execution of procedures. For a remote execution, a message is sent to the destination carrying a handler to the procedure plus any required arguments. Although similar in principle to a remote procedure call (RPC) mechanism [Birrel and Nelson 1984] (see Section 3.4), the remote execution provided by Active Messages is much simpler and more efficient. Differently from PVM, Active Messages comprises a set of communication primitives, not a complete package for DPP. However, Culler et al. [1993] developed a medium-level language called Split-C that adapts to the model. References to applications of the Active Messages interface can be found in [Active Message Project 1995] and [von Eicken et al. 1995].
3.3 The Shared-Memory Approach

For DPP, a message-passing alternative is easier to implement because it naturally maps in the structure of a distributed system. In contrast, the shared-memory approach requires more complex techniques if the environment is physically distributed, as there is no global memory and all local memories are private to the local computers [Tanenbaum 1995, Coulouris et al. 1994]. In terms of use, mainly due to the predominance of imperative programming, the reverse is true: shared-memory environments are considered easier to use than message-passing ones and it seems to be a trend towards shared-memory implementations [Lerman and Rudolph 1993]. The difference lies in the style imposed by message-passing, obliging users to explicitly structure programs around data movement between processes. In practice, this makes passing of complex data structures and migration of processes more difficult, as all information must be transferred to the requesting point instead of just passing the reference.

This has motivated the development of distributed shared memory (DSM) environments, as software layers that simulate the effect of global memory in a physically distributed system. The first effort in this direction was IVY [Li 1988]. IVY provides a global virtual memory divided into pages that correspond to physical pages. Each computer has a manager that handles local memory as a cache of the shared virtual address space. Access to a page not present in the local memory causes a trap to the operating system, which sends a request (a message) to the machine that holds the page. This scheme permits a high level of transparency for the user and allows the execution of programs written for shared-memory with minor adjustments. The disadvantage is the coarse level of sharing, a whole page, which can cause excessive communication if data in different pages is accessed frequently during execution.

The penalty in performance of IVY page-based DSM favoured research into alternative solutions, leading to the shared-variable approach. Shared-variable DSM requires the programmer to annotate the shared variables and to use them inside critical regions protected by special locks. During execution, a runtime system manages these locks, protecting accesses and maintaining consistency. By dealing with variables instead of whole pages and by selective use of mechanisms for different situations, shared-variable DSM implementations can achieve better performance than page-based DSM. The main disadvantage is a more complex programming style, with data sharing and synchronisation explicitly defined by the user. Munin [Bennet et al. 1990] is a representative of this category.
Stumm and Zhou [1990] analysed several implementations of DSM and concluded that their performance is sensitive to the shared memory access behaviour of the applications, and that no single analysed DSM algorithm is suitable for most applications. They emphasise that, in cases where performance is critical, the user may have to adopt different algorithms within the same program. Additional disadvantages are the lack of fault-tolerance and support for heterogeneous systems. The conclusion is that DSM is useful for a large class of programs, but in a number of different forms that can be chosen by the programmer. Another survey of DSM systems can be found in [Nitzberg and Lo 1991].

Another approach to shared-memory is the shared-structures model. Instead of sharing whole pages or annotated variables, it focuses on data structures to combine shared access and synchronisation. The advantage is that the implementation details can be hidden, permitting optimisations to be done transparently for the user. The main disadvantage is that the global address space is not linear, and accesses must be done per data structure, usually through a specific set of operations. In other words, the programming style enforces modularisation.

Linda [Carriero and Gelernter 1989] is a well-known package for distributed parallel processing that uses the shared-data structures approach. It uses a single type of basic structure called a tuple, which consists of a sequence of one or more fields, each field containing a value of some particular type (similarly to records in Pascal). Tuples can be inserted or removed from an abstract shared data space known as the tuple space, by using a small set of primitive operations. Instead of defining a new language, this reduced set of operations is available inserted in existing languages like C and Fortran. The advantage for the user is immediate, as no new language needs to be mastered and the extension is simple to understand and manage. Additionally, the Linda system is highly portable, with implementations in many distributed and parallel systems.

Figure 3.2 shows the abstract environment of a Linda system. Processes in any machine can insert a tuple in the tuple space by executing an out operation. The tuple becomes immediately available to all processes in the system, and can be retrieved by an in or read operation. The tuple space uses associative access and, for an in operation, the process must define a template (a sequence of values and variables). The system searches the tuple space until finding a tuple that matches the given template, removes that tuple and returns it to the process (by assigning the values in the tuple to the corresponding variables in the template). If more than one tuple matches the template, only one is retrieved. If no matching tuple exists, the
in operation blocks until the condition is satisfied (by another process inserting a matching tuple in the space). The read performs similarly, but does not remove the tuple from the tuple space. Recent versions of Linda also offer variants of in and read that do not block if the tuple does not exist. The last operation available is eval, which outputs a tuple with an expression to be evaluated. The expressions can be as simple as an arithmetic operation, or as complex as a procedure, and the system evaluates each expression in parallel, substitutes each expression by its result and then places the tuple in the tuple space.

Figure 3.2 - The Linda system

Linda implementations deliver good performance, as demonstrated by Mattson [1994] and Almasi and Gottlieb [1994]. This is achieved through a combination of compilation and runtime techniques to support the tuple space. Tuples, by convention, have a string in the first field that act like a name. Another characteristic is the ordered list of types of the fields in the tuple, its signature. The system uses these characteristics to distribute tuples into groups that have identical name and signature, each group corresponding to a specific location (computer) in the system. This distribution can be defined at compile-time if the name is a constant, or postponed to runtime if the name is a variable. In the first case, the compiled code has a direct address to access the tuple. If the name is a variable, a search will be necessary, but only in a
fraction of the total tuple space. Further optimisations include hashing inside each group, based on a carefully selected tuple field as the key.

The Linda tuple space implements a simple model that uses a shared space essentially for communication. The programming style commonly used with Linda is the manager/worker. Each worker process gets, from the tuple space, job description tuples placed there by a manager process. Workers continuously obtain new jobs and execute them, putting the results in the tuple space, until no more jobs are available. If complex data structures need to be implemented and shared, extra effort is required. In this case, the programmer must model everything using tuples as building blocks (e.g. by connecting tuples using specific name fields). Additionally, as the Linda system provides support only for a few basic primitive operations for individual tuples, control mechanisms for manipulation of data structures (e.g. mutual exclusion, conditional synchronisation, or termination) must be implemented by the user.

Another DPP environment that observes the shared-data structures approach is Orca [Bal et al. 1992]. Unlike Linda, Orca supports arbitrary data structures that can be directly used for computation. Additionally, the Orca system offers a new language and is implemented as a compiler and runtime system. The runtime system is independent of the compiler and provides an abstract layer that manages data structures in a shared data space. Data structures in Orca are based on user-defined abstract data types that encapsulate the internal structures and the operations that manipulate those structures. Once an abstract data type has been defined in the program, data structures of that type can be declared and accessed by invoking their corresponding operations. A fork statement permits the dynamic creation of new processes. The parameters of fork specify the procedure to be executed, the data (including data structures) to be passed, and the processor (computer) where the process will run. When the data structure is passed to one or more forked processes, it becomes shared. The system guarantees that operations on shared data structures are atomic when there is concurrent access by more than one process (mutual exclusion). Condition synchronisation is obtained by specifying guards for the abstract data type internal operations. When the operation is invoked, the guards are evaluated and the process is blocked if all guards result in false. The process continues only if one guard evaluates to true at the invocation or after the blocking.

The Orca runtime system is responsible for the management of data structures, including operation invocation, consistency, migration, and replication. Orca adopts replication of data
structures to improve performance on single and multiple reads, at the expense of keeping the copies consistent in case of modifications. The decision about keeping single or multiple copies of a data structure is taken dynamically by the runtime system, based on information supplied by the compiler about the expected ratio of reads to writes. Copies are kept consistent on writes by locking them in all machines before performing the update. If the underlying system permits reliable broadcasting (as in the Amoeba operating system used for the development of Orca), the runtime system can update all copies with a single broadcast operation. An extra feature in Orca is sequential consistency, which guarantees that operations appear in the same order for all processes (e.g. after two or more processes perform operations modifying a shared data structure, subsequent reads from processes will return the same consistent results).

Evaluations of Orca [Bal and Kaashoek 1993] show that the model works and can achieve efficient execution for applications. The runtime system performs well for a set of many data structures of relatively small size. Manipulation of complex data structures, like arbitrary graphs, that can dynamically grow to bigger sizes presents problems [Bal et al. 1992]. If processes executing in remote processors request frequent access to a graph, there will be an overhead in replicating and making the graph consistent (and the overhead will become worse as the graph becomes bigger). Another problem is that two or more graphs can not be directly joined, as Orca rules out pointers, and the solutions are to copy them into a destination structure or to define them all as belonging to the same graph (which leads to the problem of size). Additionally, extra overheads are present as the runtime system has to enforce that graphs in Orca are type-secure (i.e. any reference to a node that has been removed from the graph generates a runtime error).

3.4 The Agent-Based Approach

A different approach to DPP is to move processes or threads, instead of passing messages. Borrowing a term from the artificial intelligence (AI) area, the mobile process is known as an agent. In AI, an agent is a computational entity that can perform a number of actions in the system, on behalf of its user [Riecken 1994]. In the context discussed here, an agent is a process that can move from a computing node to another, carrying its state (program counter,
status bits, register contents, and, in some cases, code) and executing when reaching each destination.

The immediate advantage of this **agent-based approach** is the elimination of remote data accesses, which becomes relevant if the amount of data to be moved is big or very distributed. A second advantage is increased flexibility, as execution can be moved to a more adequate location, e.g. one that holds the necessary information or resources, or is under light loading at the moment. The main disadvantage of this approach is that it can become inefficient if the state of the moving process is large or if the amount of data to be accessed is relatively small. Therefore, two main design issues are the minimisation of the state to be moved and the distribution of the data.

The idea of moving computation is not new and has already been used in parallel and distributed systems. Process migration [Silberschatz and Galvin 1994] is a technique available to operating systems for load balancing and fault tolerance. However, the large state attached by present operating systems to its processes and the need to recover from possible communication failures makes the implementation of process migration very inefficient [Artsy and Finkel 1989]. Remote procedure call (RPC) [Birrel and Nelson 1984] is a well-known mechanism for distributed programming that permits access to a remote process via a mechanism that resembles a local procedure call. RPCs offer a good level of abstraction for the programmer, but imposes restrictions on its use and can lead to inefficient execution. Trying to overcome these difficulties, work has been developed to increase the flexibility [Stamos and Gifford 1990, Stoyenko 1994] or the performance [Johnson and Zwaenepoel 1993] of the RPC mechanism. However, important limitations of RPC, as the inability of handling pointer-based structures or large amounts of data, still have no solution.

Experiences with lightweight computation migration [Hsieh et al. 1993], in the context of operating systems, demonstrate that this new mechanism can outperform RPC and message-passing for different applications, and could be an efficient alternative to distributed shared memory (DSM). Supporting this argument, the previously described work with Active Messages [von Eicken et al. 1992] proves that remote execution can be implemented in a very efficient way.
This section discusses three independent pieces of work that represent different implementations of agent-based DPP: the Emerald environment, the Wave system, and the Telescript™ package.

The Emerald [Black et al. 1987, Jul et al. 1988, Raj et al. 1991] environment consists of a language and its runtime system. Emerald’s fundamental concepts are objects and mobility. Objects have four components: a name, unique in the environment; a representation, that contains the object local data; a set of operations; and an optional process. An object without a process is called passive, while an object that includes a process is said to be active. Processing happens by having active objects making invocations on other objects, and parallelism occurs by simultaneous execution of active objects. An object can handle multiple invocations running concurrently with the object process, and this process can also invoke operations on its own object. Synchronisation for all these invocations is provided by monitors. Emerald does not use classes or inheritance, but adopts strong typing as objects are derived from abstract types that define the object interface. Aiming at flexibility, the Emerald system permits multiple implementations of the same type to coexist, each adjusted for a different use.

The language has constructs to move, fix, unfix, or identify explicitly the present location of an object and to group or ungroup objects for movement. Mobility can be directly controlled by the program or left to be managed automatically by the system. A particular case happens when an object is passed as a parameter in a remote invocation. The Emerald system can use compile-time information to decide whether to move the objects, or the programmer can explicitly use a special call-by-move mode for passing parameters, which moves the object to the calling location, performs the call, and then moves the object back. All these requirements make an efficient implementation difficult. The strategy in Emerald was to identify the most common operations and optimise the system for them. For example, all objects in the same location share a single address space and can reference each other directly. For objects that cannot move, local invocations are implemented as a local procedure call or, if possible, in-line code. De-reference of mobile objects uses descriptors that point to the present location of the object and are updated when the object moves. A paper by Jul et al. [1988], describing a prototype implementation of Emerald in an Ethernet-based network of workstations, shows an average local invocation time of 20 microseconds, compared with a total remote invocation time of 27.9 milliseconds (3.4 ms of kernel cpu time plus 24.5 ms of communication latency).
Emerald distribution model and mobility primitives were later used in the Amber system [Chase et al. 1989], which was implemented in shared-memory multiprocessors.

The Wave system [Sapaty and Borst 1994] comprises a language and a runtime system that manages a persistent data space consisting of arbitrary graphs. The graphs form a semantic network where nodes represent concepts or entities and links represent relations. Agents are called waves and can navigate in the data space, moving from one node to another (not necessarily following the links). Waves execute when arriving at a node, modifying private (wave owned) data and data local to the node. Execution in a node is mutually exclusive and other waves must enter a nodal input queue, waiting for their turn to run. When a wave finishes execution, it leaves the node giving place to the next wave in the queue. The movement of waves is guided by a pattern-matching mechanism that permits replication and splitting. At the moment of leaving a node, the wave executes a command that specifies a combination of links and node identifiers, which permits wildcards. Each node that satisfies that pattern then receives a copy of the original wave. An additional construct allows different parts of the original wave to be sent to different destinations (splitting). The corresponding language also includes other constructs to control the spatial execution of waves and to construct and modify the persistent data space.

A prototype implementation of Wave for Unix systems, using TCP/IP protocols for communication, has been produced [Borst 1992, Sapaty and Borst 1994] and tested in LANs, WANs and in the Internet, proving that the concept is feasible. Unfortunately, no performance figures or comparison with other DPP environments has been produced. Nevertheless, Wave is unique in the way it combines agents with a structured and shared data space, providing an interesting alternative to both message-passing and shared-memory. The approach introduced by Wave motivated the research work for this thesis, which focus on ways to implement agent-based DPP in LANs and parallel machines. The Wave system is described in more details in Appendix A. Chapter 4 presents a simpler alternative paradigm and a corresponding implementation.

Agent-based processing offers a suitable paradigm for the Internet environment, where all other approaches discussed above are inadequate. The sudden increase in popularity of the Internet, mainly motivated by the appearance of graphical interfaces and the World Wide Web [Krol 1994], has attracted interest from both academic and commercial institutions. Academic experiments with Internet agents [Kotay and Kotz 1994] aim to improve the efficiency of
search engines that navigate the dynamic (and still not properly organised) internetwork environment, looking for specific information. Private company developments intend to use agent-based technology to implement commercial transactions in the Internet. The General Magic Telescript™ package [White 1994, General Magic 1995a] is an example of the latter.

The Telescript system is illustrated in Figure 3.3. Agents are described as objects that carry data and procedures, and that can be sent to specific locations or released to search for a location that satisfies some criteria. Once in its destination, the agent interacts with the host system and starts to execute its code to obtain the necessary information. When the processing is completed, the agent returns to its point of origin and interacts with the home system, delivering the information. Optionally, an agent can visit many places, collecting or delivering data, before returning. Security is a major concern, given the commercial objectives and the weakness of the Internet in this aspect. Another emphasised feature is resilience, important to preserve acquired data in the unstable Internet environment. Agents are programmed in a high-level object-oriented language known as high Telescript, that is translated to a lower level language, the low Telescript. Low Telescript is a stack-based language similar to Forth and Postscript, interpreted by a special engine installed at the destination computer. This process not only provides a machine-independent language and improves efficiency at runtime, but also permits the engine to check the agent identity, protected by encryption, and the agent permissions to execute the coded operations. Extra features allow billing and control of the agent lifetime.

![Figure 3.3 - The Telescript System](image)
Telescript is a commercial enterprise and most of the released documentation is related to publicity and the application programmer interface [General Magic 1995a]. As no independent evaluation or detailed technical information about the internal implementation of Telescript is available, only general comments are possible. The implementation emphasises security and execution in heterogeneous environments by use of interpretation and two language levels. In contrast to Wave, only a simple mechanism for agent replication seems to be available and Telescript appears to focus on single-agent execution. Clearly, agent-based processing is an immature technology and the Telescript project, despite many claims, is still an experiment. The commercial motivations, though, were enough to stimulate development of similar solutions by other private companies, as described in the article by Reinhardt [1994].

3.5 Summary

Existing environments for distributed parallel processing (DPP) can be classified according to three main computational paradigms: message-passing, shared-memory, and agent-based. The message-passing approach naturally maps in the structure of a distributed system and, therefore, is easier to implement. Its main disadvantage is the programming style, based on coarse-grained modules that interact by exchanging messages. Nevertheless, this approach has attracted many users, mainly in the engineering and scientific areas. The shared-memory approach provides an abstraction of global memory on top of a physical structure of independent computers with local private memories. It requires greater implementation effort but offers a more convenient environment for conventional, imperative programming. Limitations in the performance of shared-memory systems and the complexity of its variant, the shared-variable approach, has led to the development of new solutions. Shared-data structures combine the ease of use of the previous approach with a more efficient and abstract programming model. Instead of whole pages or specific variables, the unit of sharing is a data structure. Access to shared data structures must be through a specific set of operations and the environment provides automatic synchronisation. Existing implementations differ in the level of abstraction: some have a single type of data structure, with a fixed set of related operations, that must be used as building-block; others permit the user to define arbitrary data structures and corresponding operations. The former implies extra programming effort when dealing with complex structures while the latter has difficulties in handling arbitrary irregular structures like graphs.
The agent-based approach combines many features of the previous models. It adopts process mobility as a basic mechanism, expanding the functionality of message-passing as not only the data but a complete computation can be transferred between processing nodes. It provides an alternative solution to memory sharing, as processes can be moved to the location of data. If the state of the agent is kept small and a proper distribution of data is adopted for the executing application, agent-based processing seems capable of providing an efficient and flexible mechanism for DPP. This approach is relatively new and not adequately studied, with most recent works focusing on its application for processing in wide area networks and the Internet. The following chapters detail an architecture for agent-based processing in LANs and parallel machines, developed as the research work for this thesis.
4. The Swarm Architecture

This chapter details Swarm, an architecture for distributed parallel processing based on mobile agents. Swarm was conceived as a vehicle for exploratory research into agent-based parallel processing, improving the principles introduced in the Wave system (see Section 3.4 and Appendix A). The main objective was to produce a leaner and clearer architectural definition that facilitates (a software or hardware) implementation. Section 4.1 presents Process Flow, a simpler agent-based paradigm that forms the foundation of the Swarm architecture. A description of the architecture functional organisation is the topic of Section 4.2. Section 4.3 explains the Swarm instruction set and shows an example program in assembly language.

4.1 The Process Flow Model

The initial objective of this research was the implementation of the Wave architecture in hardware. However, a study of the technical material available about Wave revealed several obstacles to an efficient implementation. The main problems found were:

- Wave uses primitive operations which are too complex to be efficiently implemented, considering the current technology;
- the language has an inconsistent semantic definition and is poorly reported, with just a partial syntax presented;
- the language adopts a string-based form, which is unnecessarily lengthy for a low level (machine level) language and results in a terse syntax, inappropriate for a final user;
- the only mechanism used for translation is direct interpretation of code, which increases execution time and prevents the use of compile-time optimisations.
- Wave needs a dynamic environment, where even code can be modified or created during execution, making programming and runtime management extremely difficult.

Considering the characteristics above, the architectural definition of Wave (specially its set of primitives operations) was identified as the main source of complexity. This prompted the
research towards the development of a reduced set of primitives that would facilitate implementation. Organised as intermediate (medium-level) languages, successive versions of this set of primitives were produced, and the work described in [Errico and Jesshope 1994] is a representative of this phase. None of them, however, reached the level necessary for a lean implementation.

Further analysis revealed that the problem was located in the structure of the Wave paradigm. The Wave model uses an abstract approach typical of symbolic processing, mapping data in a semantic network style and using pattern matching (i.e. search) to access this data. This level of abstraction can be reduced by exposing some of the underlying structure. This procedure results in a simpler model which is capable of supporting generic processing, either symbolic or numeric. The resulting paradigm is named Process Flow (PFlow), and is defined by the following set of principles:

- Data is distributed and organised in nodes. A node is a passive, persistent object that stores data (a *persistent object* is understood here as one that has a lifetime independent from the lifetime of the program that created it [Thatte 1986]). Each node has a unique (system-wide) ID and an internal data store. The nodal data store can hold arbitrary data as well as spatial references to other nodes. A volatile nodal store is also available during runtime, acting as a scratch area.

- Processing is accomplished by agents. An agent is a volatile active object, a mobile process that carries its own state and executes instructions when it reaches a node. Once in a node, the agent has read and write access to both its own data and the nodal (persistent and volatile) data. When the agent leaves the node, another agent can take over. Thus, although physically distributed, nodal data is shared among the agents and provides the way for inter-agent communication. Additionally, the atomic execution of the agent while in a node simplifies scheduling and automatically provides mutual exclusion.

- Processing starts when a program creates its own runtime environment in the system and injects one or more starting agents. Each agent can dynamically spawn new agents, replicas of itself with different destinations. Each destination is a direct spatial reference to a node in the system. The agents move and execute concurrently in the
system and the program terminates when all agents terminate or when an abort signal is issued (by the program or by the user).

In Process Flow, the generic, semantic-network-style links of Wave are replaced by unidirectional (direct) references to nodes. This means that to acquire the effect of a Wave link, in Process Flow it is necessary to have both nodes pointing to each other (Figure 4.1). Instead of having the Wave link directions and labels, Logic Flow permits optional labeling of each reference. The result is a more consistent view of the distributed data. In Wave, a wave can move using the established links or no link at all (abstracting to the existence of tunnel links). This only means that the Wave link is not a connection between two nodes (as there is free movement from one node to any other node), but just a qualified relationship between them, which is the view explicitly adopted in Process Flow.

Process Flow has no pattern matching mechanism to navigate among the nodes, as agents can only be directed to a specific node. Node search can be implemented by the application, using the structures and primitives in the system. For instance, a directory node could be established, keeping references to all nodes in some group. These references would be qualified using keys and an agent would perform a search by moving to the directory node and executing a lookup in the local tables. Figure 4.2 compares the Wave and the Process Flow Paradigms.

**WAVE (LOGIC FLOW) PARADIGM:**
- A GRAPH is used to distribute and organise data.
- Each graph node has: a unique address; a label; an internal data store; labeled links to other nodes.
- WAVES are mobile processes that navigate the graph using a pattern-matching mechanism, executing when in a node.
- Waves are dynamically created and communicate by sharing data in the nodes.
- Waves move and execute concurrently in the graph.

**PROCESS FLOW PARADIGM**
- NODES are used to distribute and organise data.
- Each node has: unique ID; internal data store.
- AGENTS are mobile processes that go from one node to another using direct references, and execute when in a node.
- Agents are dynamically created and communicate by sharing data in the nodes.
- Agents move and execute concurrently in this aggregate of nodes.

**Figure 4.2 - Wave and Process Flow Paradigms**
4.2 Swarm Abstract Machine

The Swarm architecture was designed following the Process Flow model. The objective was to define a base layer, for implementation of end-user languages and interfaces, or to provide the basic organisation for a hardware or mixed hardware/software implementation (e.g. for a MIMD parallel computer). Instead of directly offering a user language layer for interface, Swarm defines a parallel virtual machine on top of the existing system. The intention was to offer a simple and clear architectural definition, in the form of an efficient set of primitives (the virtual machine instruction set) and structures (the virtual machine general organisation). The approach resembles the one used in COOL [Lea et al. 1993], a support layer for object-oriented languages developed for the Chorus microkernel [Rozier et al. 1988]. Similarities can also be found in the TAOS operating system [Pountain 1994], that defines a virtual machine and a virtual assembler language, but which is not based on agents.

The nodes in Swarm are distributed throughout an abstract environment known as the nspace. Each node has a unique address in the nspace for direct access and key for associative access (local search using the node key is detailed in Sections 4.3.1 and 4.3.7). The value of the key can be changed by the executing program and does not need to be unique in the system. During program execution, nodes can be created or destroyed, pieces of data can be inserted or removed from nodes and references to other nodes can be stored in any node. In this way, any number of persistent, arbitrary data structures can be created in the nspace. The user has full control over how the nodes are interconnected as well as how deep a node is (i.e. how much data it individually stores).

Swarm agents are lightweight sequential processes that can move in the nspace. Agents communicate by sharing data stored in the nodes and represent different threads of execution in a heavyweight process known as a task. The simultaneous execution of many agents at different nodes provides the parallelism. Execution starts by creating a task, that defines the runtime environment and injects one or more agents in the nspace. Each new agent executes when it reaches its destination node. There, following the instructions in the program, it may access both nodal and agent data and then be killed, suspended or spawned. Spawning produces one copy of the agent for each destination node as specified by the executing program. Agents move asynchronously in the nspace and, if more than one agent arrives at the same node, they will queue for execution. A task terminates when one of the following
conditions is satisfied: no more active agents exist in the nspace; an *abort task* instruction is executed by the program; the task is explicitly aborted by the user; or an exception automatically causes the abortion of the executing task.

The following sections detail the functional organisation of the Swarm virtual machine.

### 4.2.1 General Organisation and Execution Model

Swarm systems adopt a loosely-coupled organisation called **Swarm Abstract Machine (SAM)** (see Figure 4.3). A SAM is a collection of *Processing Modules* (PMs) interconnected by an asynchronous communication network. A PM is a (virtual or real) computer with local memory, managing a number of *data nodes*, each node holding user information. Given the number and size of nodes that a typical application may require, nodes are expected to be mapped into PMs in a ratio of 100:1 to 10,000,000:1. This mapping favours load distribution, leading to a better use of resources. Another advantage is an improved runtime model, as for one-to-one mapping the scheduling overheads tend to be larger compared to instruction execution time.

Additional structures are present in the SAM to accommodate special operations. *Service nodes* provide access to services like terminal windows, files or the underlying operating system. Each instance of the service (e.g. a particular file or window) is accessed via a different service node. A specialised PM, called *Access Point (AP)*, is the main interface to the user and holds the *console node*. Additionally, a *root node* is available on each PM, permitting the management of *lost* agents (agents that can not find their destination nodes) and the creation of data nodes when the PM is empty. Root nodes are automatically created with the respective PMs and cannot be destroyed.

This general organisation can be adapted to different implementations. In a multicomputer, each PM can be a physical processor with private memory, connected to a routing device (Figure 4.4). The interconnection network may provide fast, reliable communication, including facilities like broadcast (one to all) or multicast (one to many). Using broadcast, the program code can be sent to all units before execution, and agents need only to carry data and an instruction pointer when they move from node to node. Using multicast, a spawning agent can simultaneously send copies of itself to all its destination nodes. The system may also benefit,
using the fast communication to spread distributed control messages (like the ones used in termination detection).

**KEYS:**

PM = Processing Module  
AP = Access Point

*Figure 4.3 - Swarm Abstract Machine*
In a distributed implementation (Figure 4.5), a PM may be a computing node or a software module running in a workstation, using LAN or WAN as the communication network. A LAN implementation may still rely on an efficient communication and cached code as described above. For a WAN case, broadcast may be too inefficient and agents may have to carry code with them, in a way similar to the Wave architecture.

Regardless of the implementation, a PM works by consuming agents that want to execute in a node residing in it and producing new agents to be sent to other PMs. When an agent arrives at its destination PM, a check is made to verify whether the agent destination node exists or not. If the node exists in the PM, the agent is queued for execution. If not, the default action is to kill the agent. However, alternative actions can be provided by the executing
program to deal with this exception, such as: create a new node and queue to execute on it; search for another node in the PM and move to it; suspend the agent (in a special buffer) for later re-scheduling by another agent; move to another PM; or explicitly cancel the agent (suicide).

KEYS:
RM = Routing Module
PM = Processing Module
AP = Access Point

Figure 4.5 - A distributed implementation of Swarm

When scheduled for execution, an agent can access its own workspace as well as the workspace of the node where it is executing. The agent executes code without interruption (atomically) until it either: runs out of instructions and dies (**exhaustion**); executes a stop instruction and dies (**suicide**); or spawns (**spawning**). If the agent is dead (due to exhaustion or suicide), it is discarded. If the agent is spawning, a copy of it is delivered to each destination,
using the communication network. The node is then released for the next agent to take over. Because the program has total control over the creation and deletion of agents, explosive agent creation is only possible by careless programming. This non-preemptive agent scheduling was chosen to simplify the operation of the PM and provide the necessary mutual exclusion for access to a node. It becomes then the programmer responsibility to assure that atomic program segments (i.e. portions of code between a spawning and another spawning or a stop) are not excessively long (which could degrade the dynamic execution of the task).

The whole task starts by sending one or more agents to nodes in the nspace and terminates when no more active agents exist (i.e. all have died or the only remaining ones are suspended). In this case, the PMs automatically detect distributed termination, killing any suspended agents, clearing all transient data and sending a termination message to the console node at the AP. Optionally, a task can be forced to terminate by any agent executing an abort task instruction in a node, or directly by the user via a command to the Access Point.

4.2.2 Data Nodes

The basic building block in Swarm is the data node (Figure 4.6). Data nodes are structures stored in PMs and have four main components:

- **Persistent Workspace (PW):** holds arbitrary data that continues to exist after processing terminates.

- **Link Table (LT):** holds persistent spatial references to other nodes in the nspace.

- **Transient Workspace (TW):** holds arbitrary data that is used by a particular task and is destroyed when task processing terminates.

- **Task Lock:** if empty, the node is accessible to agents of any executing task. If holding a task ID, only agents of that task can access the node. Agents can lock nodes on behalf of its parent task by setting a special control bit in the agent structure, as explained in Section 4.2.6.

While the Transient Workspace and the Task Lock keep volatile data belonging to an executing task, the Persistent Workspace and Link Table hold permanent nodal data. The Persistent Workspace can hold arbitrary information, but at least two elements will be present:
node address and node key. The node address is unique in the system and identifies the PM where the node resides and the ID number of the node inside the PM. The node address cannot be altered by the user (read-only). The node key is used for associative reference to nodes in a PM and, although persistent, can be changed during runtime. The Link Table is a built-in structure that stores references (addresses) to other nodes. Each entry in this table is called a link and contains two fields: a numeric value called the link name, that qualifies the link, and a corresponding node address. Agents may access the Link Table to insert an entry, remove an entry or to look up. Except for the Task Lock, the other three nodal structures can change size dynamically to accommodate new data. Dynamic management of the nodal structures is automatically performed by the Swarm system.

Figure 4.6 - A SAM data node

Data nodes can be created and destroyed during execution. From any node, an agent can execute a create operation that returns the address of a new node in the current PM. Even if no data node was created in the current PM (characterising an empty PM), nodes can be created by agents executing in the root node. Data nodes are created with one Transient Workspace for each executing task and only the node address, a null node key, and an empty Link Table. Workspace positions can be later filled by executing agents. Data nodes can be destroyed by an agent executing a destroy operation with the specific node address, from any node in the same PM. Note that allocation (and de-allocation, by node destruction) of a node to a PM is then
explicitly defined by the user. As data nodes are persistent, only their Transient Workspaces and Task Locks disappear when the corresponding tasks terminate.

Any data structure can be constructed in the nspace, using the Link Tables in each node to specify the connections between nodes. In addition, the user may control the amount of information each node holds in its workspaces and how the nodes are distributed among the PMs. Accesses to the workspaces are restricted according to their type. For each executing task in the system, there is a distinct Transient Workspace in each node, accessible only to agents that belong to that task. In contrast, there is a single copy of the Persistent Workspace in each node, and all persistent information is shared by all tasks.

4.2.3 PM Access and Root Nodes

When executing, an agent gains access to both its own workspace and the Persistent and Transient Workspaces of the occupied node. Workspaces in other nodes, even if in the same PM, can only be accessed if the agent later moves to one of them. Nevertheless, PM level information can be accessed by any agent executing in one of its nodes. Each PM keeps a Local Directory of its nodes and a Spatial Table with other PM addresses. Using the Local Directory, an agent can search the PM nspace partition for a particular value of node key, receiving as a result the addresses of the nodes (in the current PM) that match the given key. Accessing the PM Spatial Table, an agent can obtain the address of another PM in the system.

Addresses in Swarm always refer to nodes. To accommodate a direct reference to a PM, an abstraction called root node was introduced in the architecture. Each PM has a single root node, created when the PM itself is created in the system. Root nodes can not be destroyed and have an address composed by the PM ID plus a node ID equal to zero. Its node key is fixed to zero and the root node has no Link Table and no additional positions, apart from node address and node key, in the Persistent Workspace. During task execution, for use as a scratch area, the root node has one Transient Workspace per active task.

The primary function of a root node is to be the reference point for a PM. Because root nodes are created with the PM and cannot be destroyed, they will always be present as long as the corresponding PM exists. Programs can use this property to perform searches in the nspace by first moving to the (known) root nodes and then searching the local partition of the nspace.
Another function of a root node is to catch lost agents. An agent arriving at the PM may not access its destination node because it is locked by a different task or is non-existent. This lost agent is automatically diverted to the PM root node, where exception code can be executed. Agents can also be intentionally directed to a root node if they do not have a reference to any other node in that PM. Once in a root node, agents can execute any operation available in a data node, except accessing the non-existent positions in the persistent workspace. Agents that have failed to reach the destination node usually follow one of the following options:

- explicitly commit suicide (which can be automatic if the proper agent mode bit is set);
- search the PM Local Directory for another node address in that PM and move to it;
- search the PM Spatial Table for the address of another PM (i.e. the address of another PM root node) and move to it;
- create a node in the PM and move to it;
- suspend execution, being removed from the current node, to be re-scheduled later by another agent (more details in Section 4.2.6).

4.2.4 Service Nodes

For access to specific resources at the host machine, like terminals, I/O devices, or applications (e.g. a database), Swarm provides special nodes called service nodes (Figure 4.7). Each service node works like a monitor [Hoare 1974] for access to a specific resource (e.g. a file) and accepts special operations related to the nature of the resource (e.g. open, read, write or close a file). Service nodes are created in a PM by an agent executing the proper create operation that specifies the resource. A service node has no Link Table. Apart from the node address, a service node has a key (which may be used to identify sub-categories of services), and a Transient Workspace for use as a scratch area. A specific number of additional positions may be available in the Persistent Workspace for use in I/O operations (accessing the external service connected to the service node). Unlike data nodes and root nodes, service nodes are volatile and private to the task that creates them (i.e. they are automatically locked and cannot be unlocked).
An agent can create a service node from any other node in a PM (data node, root node or other service node). As service nodes are private to a task, only agents from that task can access them. Depending on the nature of the resource some operations may block, suspending the agent on that node. New incoming agents (from the same task) wanting to execute on that node will also be suspended and placed in the node input queue. As soon as the pending operation is completed and the agent finishes its execution on the node, the first agent in the input queue is awakened and starts execution.

Agents directed to a non-existent service node, or to a service node that belongs to another task, are diverted to the PM root node. In this case, agents from a different task that tries to suspend on the service node will be automatically killed. This happens to prevent problems in task termination detection, as agents suspended on a service node are considered active agents (differently from agents suspended on events, as described in the next sections). Task termination is discussed in more detail in Section 4.2.6.

4.2.5 Access Point and Console Node

In its quiescent state, a Swarm system contains only nodes with persistent workspaces, distributed among the PMs. To interact with this environment, a user must open an Access Point (AP) that permits him to configure the Swarm Abstract Machine (SAM) and to run and abort tasks. In the first instance, the user acts like a system manager, able to create, destroy,
update or examine PMs. In the second, the user is a low-level programmer, or a language runtime system, submitting a program in machine language for execution in the system. The Access Point may be closed by the user after the interaction is completed. As the activated SAM exists independently from the Access Point, the latter can be opened or closed many times for user interactions.

Using the AP, a user may create a SAM environment from scratch, by issuing a startup command with some configuration (number and ID of PMs) read from a file. When there is no task executing, the system configuration can be updated by dropping or adding PMs, or the whole environment can be closed by a shutdown command. A newly created nspace is totally clear (no nodes exist, except for the root nodes on the PMs). To populate the nspace a user may start a task that will create nodes by issuing a start task command specifying the program file to be used. The Swarm system takes over, executing the task and notifying the user when it terminates. During runtime, the user can receive messages from the task, input data under task request or, exceptionally, force the task to abort. When the task terminates, all agents are killed and all transient data is cleared (including service nodes created by the task).

Using the nodal workspaces and Link Tables, persistent arbitrary data structures (i.e. structures that continue to exist even after the termination of the program that created them) can be formed in the nspace. These structures can then be accessed by the agents created by an executing program, providing a form of database in a semantic or network style. Depending on how the system is going to be used, these structures can be left on the machine, for different task executions, or they may be removed from it to give space to a new set of structures. Using a gather data command, all nodes can be collected and stored in a file. Later, the user may issue a load data command and install those nodes again in the machine. Optionally, a file in the proper format can be produced directly by the user, specifying the nodes and their (persistent) contents, and then be loaded into the machine, representing an option to the machine initialisation described above. As these files store an exact copy of the PM nodal store, they can only be loaded to a PM that matches the PM ID of the nodes stored in the file (which is the same to all nodes in the file). This happens because the node address specifies a node ID inside a specific PM, and only having meaning inside this PM.

For task execution, the AP creates a special node known as the console. The console is the only node existing in the AP and combines characteristics of root and service node. Like a root node, the console has node ID and node key equal to zero, no link table and cannot be
destroyed by agents. Like a service node, the console is private to a task, with a corresponding transient workspace, and disappears when the task terminates. The console offers basic services that permit data to input and output data to the user (as the standard input and output device for a task).

### 4.2.6 Tasks and Agents

PMs, the AP, and the different kinds of nodes are the elements that form the structure of a SAM. Programs can be executed in this abstract machine, producing results sent to a terminal or file, or stored in the SAM persistent structure. Swarm adopts a **threaded** execution model. When started, a program defines a **task**, a heavyweight process that allocates resources in the SAM, creating an environment for execution. Inside this environment, lightweight processes (threads) known as **agents** are dynamically created and destroyed, and represent the unit of scheduling. Agents share nodal Transient Workspaces and service nodes with other agents belonging to the same task, and nodal Persistent Workspaces with agents of the same or different tasks. Apart from the set of nodal Transient Workspaces and service nodes, there is a distinct console node, a separated buffer for suspended agents in each PM, and an individual distributed termination detection for each task executing in a SAM. When the task terminates (or is aborted), all its resources and locks are removed.

![Figure 4.8 - A Swarm agent](image)

The basic active entity in Swarm, an agent (Figure 4.8) is a mobile process that carries its own state, comprising the following elements:

- **destination address**: address of destination node (PM ID + node ID);
- **task id**: identifies the task to which the agent belongs;
- **status word**: carries the agent **serial number**, received in the last spawning, the **agent workspace pointer**, which points to the last non-occupied position in the workspace, and a set
of agent control bits, that specify actions in case of exceptions (runtime faults) and signalise operation failures:

- **instruction pointer**: points to the next instruction to be executed;

- **agent workspace**: intermediary data private to the agent.

Agents can be created (by spawning), killed (by exhaustion or suicide) or suspended (waiting for a signal to be sent later by another agent, or for the opportunity to execute in a service node). When an agent executes in a node, it may collect a list of destinations for later propagation. When the agent is ready, it executes a spawning that produces a copy of the agent for each destination address. Each resulting agent receives not only its destination address, but also the index of this address in the original destination list. This index is called the **serial number** of the agent and can be used for conditional execution of code when the agent reaches its destination. An agent dies when it runs out of instructions to execute (exhaustion) or by forced termination. The latter occurs when the agent explicitly executes a *stop agent* or an *abort task* instruction, or when the agent is automatically killed by the system because of an exception.

The bits in the control word permit the agent to change some aspects of its behaviour. **Mode bits** can be set to force agent termination in case of runtime error or failure in access the destination node, or to activate the *task lock* or the *task release* modes. In task lock mode, the agent locks each visited node for exclusive access by agents belonging to its parent task only. Regions of nspace can be locked in this way, and the locks are automatically released when the task terminates. Optionally, agents belonging to the locking task may release locked nodes when visiting them with the task release mode set.

If the mode bits forcing agent termination are reset, the agent can identify an exceptional condition by examining the **status bits** and take a program-specified procedure. A typical situation is a failure to access the destination node. The incoming agent is re-directed to execute on the PM root node, where it can detect the cause of the fault (node is locked or non-existent) and take proper action (try another node, suspend execution, create the node, or even suicide). However, if the agent destination address specifies a non-existent PM, the agent is eliminated by the Swarm system.
The instruction pointer (IP), also known as program counter, points to the next instruction to be executed by the agent. As discussed before, depending on the implementation the agent may have to carry the executing code or not. In the latter, the code is broadcasted to all units (PMs and AP) and the agent needs only to carry the IP.

The agent keeps a private workspace to hold the information necessary during processing. Any data stored in the agent workspace is, by the nature of the agent, transient, and permanent data must be transferred to a nodal persistent workspace before the task terminates. Additionally, as the agent workspace is limited in size (due to the number of bits of the Swarm machine instruction used to identify a position in this workspace - see Section 4.3.2), if an agent needs to accumulate more data than it can carry, it would have to leave the excess temporarily in some node, before proceeding execution.

Figure 4.9 - Different workspaces in Swarm
Together, the persistent and transient nodal workspaces and the agent workspaces form the whole data space of a task executing in Swarm, offering different levels of sharing as shown in Figure 4.9. This scheme provides the basic mechanism in Swarm for agent synchronisation. According to Andrews [1991], there are two main types of synchronisation for concurrent processes: mutual exclusion, when processes compete for access to the same resource; and condition synchronisation, when processes are blocked waiting for some condition to be satisfied before resuming execution. Mutual exclusion is automatic in Swarm, as no more than one agent can be executing in a node at the same time (although a PM can manage many agents executing at different nodes). Condition synchronisation is obtained by suspending agents in a PM (Figure 4.10). While processing in a node, an agent can execute a suspend instruction which specifies a value (chosen by the programmer) that identifies an event. The agent is removed from the node and put in a PM managed buffer. More than one agent can be suspended on the same event ID. When another agent, executing in a node in the same PM, issues a wake instruction with the same value for event ID, the agents suspended on this event are re-scheduled for execution in the node where the wake instruction was executed (after the present agent leaves the node). Re-scheduled agents have priority over incoming agents for execution in a node.

Although the transient workspace in a node is distinct for different tasks, persistent data is shared and its access by different tasks can lead to conflicts. The Swarm system provides a basic mechanism for exclusive access in these cases. A task can set a special task lock flag in any data node it wants for private use, while root nodes can not be locked and the console and service nodes are locked by default and can not be released. Agents from other tasks arriving at a locked data node will be diverted to the PM root node, where they can take any of the actions previously described in Section 4.2.3. Alternatively, if the corresponding bit in the agent Status Word is set, the agent will be killed if its destination node is locked. Deadlocks may occur in these cases if locking is done without care. The approach currently adopted in Swarm is just to provide a minimum set of features and leave the management of inter-task conflicts to the user.
Agent executes a SUSP and is suspended on the given event number.

Another agent executes a WAKE in another node of the same PM, giving the event number used before.

All agents suspended on that event number are then re-scheduled for execution on the node where the WAKE was executed (after the previous agent left the node).

Figure 4.10 - Condition synchronisation for agents in Swarm

4.3 Swarm Instruction Set

The set of primitive operations available in the Swarm Abstract Machine defines its machine language. Programs written in a higher level language must be compiled to this machine language before execution, while low level programming may use the corresponding Swarm Assembly Language (SWASM).
Arithmetic/Logical (xxxA = variable, xxxI = immediate)
ADDA/ADDI integer addition
SUBA/SUBI integer subtraction
MUL/A/MULI integer multiplication
DIVA/DIVI integer division quotient
MODA/MODI integer division rest
EQ/LA/EQLI equal comparison
NEQA/NEQI not equal comparison
GT/GEA/GTEI greater or equal comparison
LSA/LSTI less than comparison
GEA/GTEI greater than or equal comparison
EQ/LA/EQLI equal comparison
NEQA/NEQI not equal comparison
ANDA/ANDI bitwise and
IORA/ORI inclusive or
XORA/XORI exclusive or
SHLA/SHLI left shift
SHRA/SHRI right shift
SAR/SARI arithmetic right shift

Data Transfer
LDTW load value from a Transient Workspace (TW) position
STTW store value to a Transient Workspace (TW) position
LDPW load value from a Persistent Workspace (PW) position
STPW store value to a Persistent Workspace (PW) position
LDAW indirect load from a Agent Workspace (AW) position
STAW indirect store to a Agent Workspace (AW) position
SETH set high with immediate
SETL set low with immediate

Control-Flow
BIPT branch if true
BIFF branch if false
JMPR immediate jump (IP relative)
JEA absolute jump
JEAVR immediate jump and save (IP relative)
JEAVA absolute jump and save
JRET return from jump and save

Agent-Control
SPWN spawn agent
STOP stop agent
TERM terminate task
SUSP suspend agent(s) on a given event
WAKE awake agent(s) suspended on a given event
LDSW load agent status word
STSW store agent status word

Special
CRND create node
DTND destroy node
INDL insert node address in Destination List
INLT insert entry in Link Table
DILT delete entry in Link Table
LUET lookup entry in Link Table
RLUET reverse lookup entry in Link Table
LUD lookup Local Directory
LUST lookup Spatial Table

Figure 4.11 - Swarm Instruction set

Swarm loosely imitates the load/store style of reduced-instruction-set computers (RISCs) [Hennessy and Patterson 1990]. The agent workspace takes the place of the register set and the memory is distributed among the nodes, divided in each node into two segments (transient and persistent). Operations only accept agent workspace positions for operands and the first position in the Agent Workspace is fixed to zero (i.e. AW[0] = 0). For manipulation, data in the nodal workspaces must be loaded into the agent and, if necessary, stored back after the operation. Data is organised in 32-bit words and word aligned. Regarding endianess, Swarm
adopts big endian byte ordering inside a word (chosen to match the standard for data transfer between machines in heterogeneous networks [Stevens 1990]).

The present version of the instruction set comprises the 65 instructions shown in Figure 4.11. All instructions are of fixed size (32-bit) and word aligned in memory. There are no instructions for byte and half-word transfers or for floating-point operations in this version, but the instruction set can be further expanded to accommodate these if necessary, including support for 64-bit data. The following sections detail the internal organisation of the workspaces and the Swarm instruction set, and show an example program in assembly language. Detailed descriptions of each individual instruction can be found in Appendix B.

4.3.1 Workspaces and Built-In Structures

The Swarm Abstract Machine has several built-in structures available to the instruction set. These structures comprise all workspaces and other functional elements present in agents, nodes and PMs. Agents have three elements accessible by the user (Figure 4.12): the Agent Workspace (AW), the Agent Status Word (ASW) and the Agent Destination List (DL). The AW has a maximum size of 256 positions, each capable of storing a 32-bit word. In case the program tries to access a position beyond this limit, an exception (runtime error) is generated by the Swarm system, which can automatically kill the offending agent or be detected for appropriate handling (as explained next). From the low-level programmer point of view, the Agent Workspace is equivalent to the register set of a load/store processor architecture: position AW[0] is fixed to zero; position AW[255] can be used to save the current IP value in case of a jump; instructions use AW positions for source and destination operands; data in memory can only be accessed by loading it in an AW position; AW is limited in size (although bigger than the usual register set). The main difference is that, in Swarm, there is one workspace per agent, and an Agent Workspace is private (not visible to other agents). All Agent Workspace positions, except AW[0], are automatically initialised to the EMPTY value in the first agent of a task. Swarm uses as EMPTY value the most negative number in two's-complement (80000000 hex). This provides similar functionality of an empty/full bit per memory position, without requiring special hardware.
The agent status word (ASW) combines four different functions in a 32-bit word that can be accessed by loading it into an AW position. As shown in Figure 4.12, the ASW is divided into three fields. The first field is 16-bit wide and holds the agent serial number (SN), detailed in Section 4.2.6. On spawning, each resulting agent receives a different serial number, which can be read when ASW is loaded in the AW during execution in the destination node. The serial number corresponds to the position of the current node in the original list of destinations of the parent agent, and can be used for conditional execution of code in the destination node.

The next two fields are 8-bit wide and comprise the agent workspace pointer (AWP) and the agent control-bits (ACB). The AWP points to the next available position after the highest AW position referenced by the executing program, up to the present moment. The executing program can examine all fields when the status word is loaded into an AW position.

The SN and AWP fields are read-only. When the content of an AW position is stored in the agent status word, the bits corresponding to these fields are ignored. Only the last field, ACB, can be partially updated. As Figure 4.12 shows, the agent control-bits field has five mode bits and three status bits. The status bits (bits 2-0) are read-only and identify the cause of a
runtime failure: exception (arithmetic overflow, divide-by-zero, illegal opcode, illegal operand, out-of-memory access), destination node locked, or destination node non-existent.

Mode bits can be set by the agent, specifying the task locking mechanism and the default action in case of a runtime failure. Bits 7-5, if set, make the agent stop (die) in case of an exception or a failure to access the destination. If these bits are reset, the agent ignores exceptions and is directed to the PM root node when a node access failure occurs. Bit 4 is the Task Lock bit that, when set, locks the visited nodes preventing access by agents of other tasks. Bit 3, the Task Release bit, causes the opposite action, releasing the lock on visited nodes. If both bits (3 and 4) are simultaneously set, the Swarm system automatically resets them. The agent scheduling mechanism of Swarm implies that changes in mode bits are effective only after a spawning, and do not affect execution in the current node. All spawned agents inherit the mode bits configuration of the parent and the first agent in the task has the default setting: bits 7-3 reset. For the whole task, the default action is to ignore exceptions. A task can be set to abort the task in case of an exception by placing the corresponding task directive at the beginning of the assembly program. Task directives are detailed in Appendix B.

The last visible agent structure is the destination list (DL). An agent begins execution in its destination node with an empty DL, which can be progressively filled with destination addresses, up to a maximum of 65,536 elements. As explained above, when the spawned agents reach their destination nodes, the position of the node address in the original destination list in the parent becomes the agent serial number. A node address can be obtained from the program, by searching local tables, or from a workspace position. Node addresses can be inserted in the DL, but can not be removed. If removal is necessary, the agent must store addresses in workspace positions before committing the final values to the DL. A spawning with an empty DL is equivalent to stopping the agent (i.e. the agent dies). The present version of Swarm uses 32-bit node addresses, logically divided into two halves:

\[
\text{higher half (16-bit)} \quad + \quad \text{lower half (16-bit)}
\]

\[
\text{PM ID} \quad + \quad \text{node ID}
\]

This addressing style allows up to 65,534 PMs in a SAM. The PM ID value FFFF_{HEX} is reserved for broadcast and the value 8000_{HEX} is invalid. Apart from the node ID 0000_{HEX}, reserved for root nodes, all other values are permitted, giving a maximum of 65,535 nodes per PM. Combinations like FFFF_{HEX} + \text{node ID} causes broadcast to all nodes with that ID in each
PM (all root nodes if node ID is 0000\text{HEX}). Agents directed to an invalid or non-existent PM ID are automatically killed by the Swarm system.

A node in Swarm is logically divided into three structures visible to the user (Figure 4.13): the Persistent Workspace (PW), the Link Table (LT), and the Transient Workspace (TW). The first position in the Persistent Workspace (PW[0]) is read-only and stores the node address defined by the system when the node is created. The second position (PW[1]) holds the node key, initialised to EMPTY on node creation and modifiable by the program. Subsequent PW positions are dynamically allocated by demand and can permanently hold any user-defined data. During execution, a read access to a non-used PW position returns the EMPTY value (80000000 \text{hex}). The Link Table is a permanent nodal structure managed by the Swarm system. A LT is organised into entries, each entry in the format \text{linkname (32-bit)} + \text{node address (32-bit)}. Linknames are numeric values used to qualify a reference to a node as a label in a graph arc. The precise meaning of the linknames is defined by the system user. Four special instructions (discussed in Section 4.3.7) are available to insert, remove and look up entries in an LT.

The Transient Workspace is a nodal volatile storage area created by the system when a task is started. On a node there is one Transient Workspace per executing task, and this area is not visible to other tasks present in the SAM. A TW holds temporary data created by the task and its size and initial contents are defined by the program. The default procedure is to initialise all TW positions to the EMPTY value (80000000 \text{HEX}). Optionally, the programmer can use a task directive (task directives are detailed in Appendix B) to initialise positions to specific values. The maximum size for the PW and TW areas is defined by the word size (i.e. $2^{32}$).
Finally, the PM has two visible structures (Figure 4.14): the **Local Directory (LD)** and the **Spatial Table (ST)**. Both are managed by the system and can be accessed for lookup by the program. The Local Directory permits a search for addresses of nodes in the PM with a specific key value. A lookup in the Spatial Table returns one of two different addresses, depending on the linkname given. A zero linkname returns the address of the task console node, while a non-zero linkname returns the address of a root node in another PM in the system. The specific instructions that manipulate the LD and ST are detailed in Section 4.3.7.

![Local Directory](image1)

**Figure 4.14 - PM built-in structures**

4.3.2 **Instruction Formats**

Swarm uses the five instruction formats shown in Figure 4.15. Because instruction operands must be in AW, each operand reference requires an 8-bit field (AW maximum size is 256 positions). The **instruction prefix bit** makes a distinction between format 4, that has a 2-
bit opcode field, and the others, that use a 7-bit opcode. Format 4 uses the extra space to provide two AW references plus a 13-bit constant, and is used by load/store instructions that access the two nodal workspaces (PW and TW). Formats 3 and 2 are used by three-operand and two-operand + immediate value instructions. Set instructions and branches use the format 1, with space for a 16-bit constant. The format 0 is reserved for jump and call instructions that specify 24-bit IP offsets.

4.3.3 Arithmetic/Logical Instructions

Arithmetic, logical and bitwise boolean instructions form the arithmetic/logical group. Operands must be in AW positions and each instruction in this group has both a variable (three-operand) and an immediate (two-operand + constant) version:

variable: \[ Ad := As \text{ operation } As2 \]

immediate: \[ Ad := As \text{ operation const} \]

\((Ad = AW \text{ destination}; Asx = AW \text{ source}; \text{const} = 8\text{-bit constant})\)

Arithmetic instructions comprise the four integer operations (add, sub, mul, div) plus the division rest (mod) operator. They always assume signed (2's-complement format) integer operands and cause exception in case of overflow or division by zero. Logical instructions perform the six logical comparisons (eql, neq, grt, lst, geq, leq). Operands are treated as signed integer numbers and the result is stored in the destination AW position as binary one (true) or binary zero (false). Bitwise instructions permit bitwise boolean operations (and, or, xor) and shifts (shl, shr, sar). The shift left (shl) and shift right (shr) fill new bit positions with zero. The shift-arithmetic right copies the sign bit into the new positions. In all cases, 8-bit immediate constants are converted into the equivalent 32-bit format before the operation is performed.

4.3.4 Data Transfer Instructions

The second group are the data transfer instructions: agent-to-node load/store, intra-agent load/store, and immediate set. The four agent-to-node load/store instructions are used to transfer data between a nodal workspace position (PW or TW) and a position in the Agent Workspace (AW). Their formats are:
load: \[ Ad := NW[Asl + const] \]
store: \[ NW[Ad + const] := Asl \]

(Ad = AW destination; NW = PW or TW; Asx = AW source; const = signed 13-bit constant)

Swarm supports indirect addressing in the agent workspace, i.e. the contents of an agent workspace position pointed by another AW position can be loaded or stored. This allows quick manipulation of linear lists, a data structure that maps well in the agent workspace. The format for the instructions (ldaw, staw) are:

load: \[ Ad := AW[Asl + const] \]
store: \[ AW[Ad + const] := Asl \]

(Ad = AW destination; Asx = AW source; const = signed 8-bit constant)

Immediate set instructions (seth, setl) enable the loading of a 16-bit constant directly into an AW position or, by using the two instructions, a 32-bit constant. The *set high* loads the 16-bit constant into the higher half of the AW position (clearing the lower 16-bit half), while the *set low* loads the constant in the lower half (leaving the higher half undisturbed). The format for both instructions is:

set: \[ Adx := const \]

(Adx = AW destination, high or low half; const = unsigned 16-bit constant)

### 4.3.5 Control-Flow Instructions

The control-flow instruction group encompasses branch and jump instructions. Branches (bift, biff) are conditional changes in the sequence of instruction execution, the condition specified by the content of an AW position being true (i.e. different from zero) or false (i.e. equal to zero). Jumps (jmpr, jmpa) are unconditional changes in the sequence of instruction execution. The *jump and save* variants (jsvr, jsva) save the current value of the IP (which points to the next instruction to be executed) into the AW[255], before updating it. This value can be restored (from AW[255] back to the IP) by execution of the *return* instruction (jret). In case of successive executions of *jump and save* instructions, it is the responsibility of the programmer to save previous values stored in AW[255] in other positions of the agent.
workspace or in nodal workspaces. In other words, creation and management of stacks are left to the user. The formats are:

\[
\text{branch: } \quad \text{IP} := \text{IP} + \text{16-bit const} \quad \text{(if Asl is true or false)}
\]
\[
\text{jumps: } \quad \text{IP} := \text{IP} + \text{24-bit const} \quad \text{(immediate)}
\]
\[
\text{IP} := \text{Asl} \quad \text{(absolute)}
\]

(IP = instruction pointer; Asl = AW source; const = signed constant)

### 4.3.6 Agent-Control Instructions

Agents can be controlled using the \textit{spawn}, \textit{stop}, \textit{terminate}, \textit{suspend} and \textit{wake} instructions. Spawn terminates execution of the agent in the current node, making it available to a following agent, and generates a copy of the agent for each address in the destination list. Stop simply stops execution and kills the agent. Terminate causes termination of the whole task, killing all agents and removing all structures in the SAM that belong to that task. The suspend operation suspends agent execution, specifying an event ID, removing the agent from the node and storing it in a PM managed buffer in a position identified by the event ID given in the instruction. When another agent executes in the same PM a wake instruction with the proper value of event ID, corresponding agents are re-scheduled for execution in the node where the wake was executed. When competing for access to a node, suspended agents have preference over incoming agents. Suspend and wake require one operand, an AW position that contains the event ID specified by the user. The other three instructions have no operands.

Included in this category are two instructions (ldsw, stsw) that load and store the value of the agent status word (ASW). Once loaded in an AW position, the agent can examine the ASW fields and, if necessary, load it back after setting the mode bits (see Section 4.3.1). Read-only fields and bits are not affected by the store operations. The format for these instructions are:

\[
\text{load: } \quad \text{Ad} := \text{ASW}
\]
\[
\text{store: } \quad \text{ASW} := \text{Asl}
\]

(Ad = AW destination; Asl = AW source)
4.3.7 Special Instructions

The last group comprises instructions to manipulate Swarm built-in structures and to create and destroy nodes. The instructions that access the agent Destination List (DL), the nodal Link Table (LT), and the PM Local Directory (LD) and Spatial Table (ST) are:

**insertion:**
- INDL DL @ As1
- INLT LT @ As1,As2 (As1 = node addr, As2 = linkname)

**deletion:**
- DLLT LT !@ As1,As2 (As1 = node addr, As2 = linkname)

**lookup:**
- LULT Ad := LT[As1].node_addr (As1 = linkname)
- RLLT Ad := LT[As1].linkname (As1 = node_addr)
- LULD Ad := LD[As1].node_addr (As1 = node key)
- LUST Ad := ST[As1].node_addr (As1 = linkname)

(Ad = AW destination; Asx = AW source; @ means insert; !@ means remove)

Successive lookups with the same linkname (in LT and ST) or key (in LD) return node addresses with identical qualifier (key/linkname), until the list ends and the lookup returns the EMPTY value (80000000\text{hex}). If there is no node address that satisfies the given qualifier, the EMPTY value is returned. As a special case, a sequence of lookups performed in the Link Table, with an EMPTY value as linkname, will return the node addresses regardless of their associated linknames. This also works for the reverse lookup in the LT (i.e. node address = EMPTY), returning in this case a sequence of linknames, regardless of the associated node addresses. Although LD and ST are PM structures, they can be accessed from any node in the PM. Because DL is limited in size, if an agent tries to insert more than 65,536 addresses in DL, the excess values are discarded.

Nodes can be created or destroyed in the current PM during execution, by using the *create node* (cmd) and *destroy node* (dtnd) instructions, with formats:

**create:**
- Ad := node address, As1 = node type

**destroy:**
- As1 = node address

(Ad = AW destination; As1 = AW source)

The create instruction requires a *node type* operand. If the node type is zero, an empty data node (node address, empty node key, empty LT, and one TW per task executing) is created in the PM, and the instruction returns the corresponding address. The destroy instruction removes
from the PM the node whose address is in the given AW position (As1), updating the LD accordingly. Both instructions have no effect on nodes in other PMs. If an agent kills its current node, it is also killed.

4.3.8 Service Calls

Agents can access services when executing in a console node or a service node. As services are memory-mapped, there is no specific instruction for them. Instead, the agent must load and store values in specific PW positions to perform the operation. The PW layout and the list of services of a console node are shown in Figure 4.16. Persistent workspace position PW[3] holds the operand (and must be loaded with data for outputs). The operation is triggered when the service code is loaded in PW[2].

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NADDR</td>
<td>NKEY</td>
<td>I/O command</td>
<td>I/O data</td>
</tr>
</tbody>
</table>

![Console Node "Persistent Workspace"

Figure 4.16 - Console node services

The console node is automatically created when the task starts, but service nodes must be explicitly created during execution. The create node instruction is used to create service nodes, with the format:

create: Ad := node address, As1 = node type, As2 = service ID

(Ad = AW destination; Asx = AW source)

The node type specifies the type of the service (e.g. file, terminal window) and the service ID specifies an instance of that service (e.g. a specific file, a specific terminal window). A file service node can be used as an example. The node type for files is 1 and the service ID must be
the number of an entry registered with the Swarm system (via the AP) before the task was started. The *service ID* is a short form for a possibly long character string that is the name of the file. When the service node is created, the corresponding file is automatically opened and linked to that node. The create operation returns the address of the service node, which can be used as destination for the agent. An agent that moves to a service node can read or write data in the same way used for the console node. When the service node is destroyed, explicitly by a destroy node instruction or automatically, by task termination, the corresponding file is closed. An attempt to create another service node for the same file will fail, return the EMPTY value as the node address.

In addition to the data input and output services, the console node permits a special operation. Another task can be started from the current task, by loading the number (see *service ID* above) of the desired program file in PW[3] and the service code 0 (zero) in PW[2]. A new AP, and console node, is created and the new task is launched, executing independently from the original task. This feature is useful when a processing must be divided into different tasks and each task must wait for the termination of a previous one to start its execution. Using the *launch by task* feature and node locking, this effect can be achieved in Swarm.

### 4.3.9 An Example Program

The agent-based processing style used in Swarm can be better understood by an example. Figure 4.17 shows, in pseudo-code, a Swarm program that finds a path between any two given nodes in a graph. For simplicity, the graph is assumed to be connected and already constructed in the nspace. Links are undirected and each node is labeled with a different, single letter. The topology can be arbitrary, and cycles are allowed. For instance, the graph in Figure 4.18 satisfies these conditions.

The program uses the agent DL and positions in the AW to store the name of the start and end nodes, the addresses of start-node and predecessor node, and the list of visited nodes (the path). One TW position is used to mark nodes as visited and the node address (PW[0]) and node key (PW[1]) are accessed. The node key is used to store the node label, which facilitates the search. The Swarm system automatically initialises all AW and TW positions to the EMPTY value and DL is always clear when the agent starts processing in a node.
Variables:

DL (agent Destination List)

AW (agent workspace):
ASTART label of path start-node
AEND label of path end-node
ASTADDR address of start-node
APRED address of predecessor node
APATH path list

TW (nodal transient workspace):
TMARK visited node mark

PW (nodal persistent workspace):
NADDR node address (PW[0])
NLABEL node label (uses PW[1])

Program (pseudo-code):

begin
input ASTART
input AEND
DL «- 0x11110000
spawn
ASTADDR <- LULD(key = ASTART)
if ASTADDR = empty then stop
APRED <- ASTADDR
DL «- ASTADDR
repeat
spawn
if TMARK != empty then stop
TMARK <- 1
APATH «- NLABEL
if NLABEL = AEND then exit loop
DL «- neigh_addr - APRED
APRED <- NADDR
until “forever”
DL «- console addr
spawn
output APATH
end

# input label of start-node
# input label of end-node
# spawn to root nodes in all PMs
# search for start-node
# if search fails then die
# initialise agent predecessor var
# prepare to move to start-node
# propagation loop
# spawn to neighbours
# if node was already visited then die
# mark node as visited
# insert node label in agent list
# if it is the end-node then exit loop
# insert neighbour addresses in ADL
# (except predecessor)
# update agent predecessor var
# end of propagation loop
# move to console node
# output found path

Figure 4.17 - Example program (in pseudo-code)

Figure 4.18 - A connected graph
There are three distinct phases in the program: initialisation, propagation and output. During the initialisation (lines 1-9), the first agent begins execution in the console node by requesting from the user (the label of) the two graph nodes (lines 2-3). Next, the agent spawns a copy to each PM root node in the system (lines 4-5). On a root node, the agent searches the PM local directory (lines 6-7), looking for the path start-node. On the PMs where no node satisfies the condition, the agent dies. On the PM that has the start-node, the agent initialises its variables and prepares to spawn to that node (lines 8-9). If no PM has a node with the given label, the program terminates on line 7.

In the propagation loop (lines 10-18), the task spawns agents to the neighbour nodes trying possible paths that lead to the end-node. After a spawn, each agent checks if the node was already visited (lines 11-12). If it was, the agent dies. If not, the agent marks the node as visited, inserts the node label in its list of visited nodes (the path), and verifies if it is in the path end-node (lines 13-15). If it is not the end-node, the agent collects the list of neighbour nodes (predecessor node excluded), stores the current address as the predecessor (lines 16-17) and spawns (line 11), re-starting the loop.

The first agent to reach the end-node exits the loop (on line 15) and continues to the output phase (lines 19-22), when the agent propagates to the console node, outputs the contents of its list (APATH), and stops. All other agents will be killed when reaching a visited node (line 7) or spawning with an empty destination list (line 11), because no neighbour except the predecessor node was found. The task terminates when all agents finish processing. As agent propagation is asynchronous, execution is nondeterministic and, in practice, different executions may produce different results.

Expanding the pseudo-code statements into the corresponding sequence of Swarm instructions results in the assembly language (SWASM) version of the program, shown in Figure 4.19. For simplification, the program assumes that in no case does DL receive more than 65,536 addresses or APATH receive more than 246 elements (as the maximum size for AW is 256).
Mapping of variables:

AW[0]  zero
AW[1]  empty
AW[2]  AEND
AW[3]  ASTART
AW[4]  ASTADDR
AW[5]  APRED
AW[6]  AUX
AW[7]  SERVICE_CODE
AW[8]  NEIGHADDR
AW[9]  APATH_PTR
AW[10]  APATH
TW[0]  TMARK
PW[0]  NADDR
PW[1]  NLABEL (uses NKEY position)

Program:

00  ADDI A7 A0 #06  * select "input symbol" service
01  STPW A0 #0002 A7  * execute "input symbol" service
02  LDPM A3 A0 #0003  * input ASTART
03  STPW A0 #0002 A7  * execute "input symbol" service
04  LDPM A2 A0 #0003  * input AEND
05  SETL A6 #1111  * spawn to all PM root nodes
06  SETL A6 #0000
07  SETH A6 #1111  * spavm to all PM root nodes
08  SETL A6 #0000
09  LULD A4 A3  * ASTADDR <- lookup_dir(key = ASTART)
10  EQLA A6 A4 A1  * if ASTADDR = empty then stop
11  BIFF A6 DIE  * APRED <- ASTADDR
12  ADDA A5 A4 A0  * DL = ASTADDR
13  INDL A4  * insert APATH_PTR
14  ADDI A9 A0 #09  * [START MAIN LOOP
15  MAINB: SPWN  * spawn to neighbours
16  LDPM A6 A0 #0000  * AUX <- TMARK
17  NEQA A6 A6 A1  * if AUX = empty then stop
18  BIPT A6 DIE  * if AUX = AEND then exit loop
19  SETL A6 #0001  * AUX <- NLABEL
20  ADDI A9 A9 #01  * APATH_PTR = APATH_PTR + 1
21  STIA A9 #00 A6  * APATH[APATH_PTR] = AUX
22  EQLA A6 A6 A2  * if AUX = AEND then exit loop
23  BIFT A6 MAINE  * [START LOOKUPLT LOOP
24  LULTA: LULT A8 AO  * NEIGHADDR <- LULT(linkname = 0)
25  EQLA A6 A5 A1  * if NEIGHADDR = empty then exit loop
26  BIFF A6 LULTE  * if NEIGHADDR = APRED then restart loop
27  LULTA: LULT A9 A6  * insert NEIGHADDR in DL
28  LDPM A6 A6 #0000  * repeat loop
29  JMPR LULTB  * [END LOOKUPLT LOOP
30  INDL A8  * repeat MAIN LOOP
31  JNPR MAINB  * [END MAIN LOOP
32  LULTB: LDPW A5 A0 #0000  * APRED <- NADDR
33  LULTB: LDPW A5 A0 #0000  * repeat MAIN LOOP
34  LULTB: LDPW A5 A0 #0000  * [END MAIN LOOP
35  MAINE: LUPF A8 A0  * get console node address
36  INDL A8  * spawn to console node
37  SNWN
38  ADDI A7 A0 #05  * select "output symbol" service
39  OUTPB: LDPM A6 A9 #00  * [START OUTPUT LOOP
40  SUBH A9 A9 #01  * decrement APATH_PTR
41  STPW A0 #0003 A6  * output to console
42  STPW A0 #0002 A7  * if APATH_PTR > 9, repeat OUTPUT LOOP
43  EQLA A6 A9 #09
44  BIFF A6 OUTPB
45  DIS: STOP

Figure 4.19 - Example program in assembly language
4.4 Summary

Process Flow is an agent-based paradigm derived from the Wave model. As in Wave, Process Flow defines a parallel processing environment where agents (mobile processes) move and execute on distributed data structures formed by data nodes. Each node is itself a persistent structure that can store arbitrary data and references to other nodes. Process Flow, however, facilitates implementation by simplifying the search mechanism and the node interconnection used in Wave.

Swarm is a parallel architecture for exploratory research into agent-based distributed parallel processing, which has emerged from the Process Flow paradigm. The Swarm architecture defines a virtual parallel processing environment known as the Swarm Abstract Machine - SAM, where data nodes are mapped to virtual processors (PMs) interconnected by an asynchronous communication network. A special processor (AP) gives access to the user and holds a unique console node for basic I/O during program execution. Additional I/O services are provided by PM service nodes, created by the running code. A permanent root node in each PM permits direct access to the PM and serves as collection point for lost agents.

Each active program defines a task, which allocates resources and generates a family of agents that share and execute in the distributed data space defined by the nodes. Agents cooperate to find the final solution and synchronise by mutual exclusion on node access and conditional waits. A task terminates when all its agents finish execution or an abort signal is generated (by the program, the user, or an exception). Agents carry their own state, comprising status bits, instruction pointer, and a private workspace. Once an agent reaches a node, it executes there, having access to data stored in the nodal transient (arbitrary data) and persistent structures (arbitrary data and links to other nodes). Each task executing in the system defines its set of transient workspaces, but all tasks share access to the persistent structures and can lock nodes for exclusive access.

The SAM adopts a RISC-style instruction set, with additional support for access to built-in structures and coordination of multi-agent execution. Parallel applications can be developed directly in this machine language or indirectly, by compilation of a high level language. The agent workspace is used like a bank of registers and data must be moved to it (from nodal workspaces) for manipulation. The architecture supports 32-bit data and provides load/store, integer arithmetic, comparisons, bitwise boolean, and common control (branches and jumps)
operations. Special instructions permit to spawn, stop or suspend agents, to create and destroy data nodes and service nodes, and to manipulate nodal links and PM directories.
5. The Swarm Prototype Implementation

This chapter describes the prototype Swarm system, implemented in a Unix environment. Section 5.1 explains the major design decisions and details the implementation. Section 5.2 presents an example application processed in the Swarm prototype: a bandwidth reservation program for communication networks.

5.1 Prototype Implementation

The primary goal of the prototype implementation was to validate the operating principles and the functional organisation of the Swarm architecture. A secondary goal was to provide a test environment for experimentation with agent-based programming. The design policy was to achieve a minimum working system that could be later expanded to include additional features. Following this policy, the prototype did not implement multitasking or service nodes. Another restriction was that the present version is not yet connected to the LAN, running in a single workstation. However, this did not represent a serious problem, as the prototype has been designed for later connection to the network, and the present single-workstation version is sufficient for simulations.

The Swarm prototype was implemented for Sun SparcStations running Unix (SunOs version 4.1.3). The modules were written in the ANSI-C language and compiled using the GNU gcc compiler. Figure 5.1 shows the general structure of the Swarm Abstract Machine (SAM) implemented in the prototype. A host computer can have one or more Processing Modules (PMs), each capable of holding Swarm data nodes. These PMs are connected to the local Routing Module (RM), which routes agents between modules during program execution and, when no task is running, acts as a local manager that can create additional PMs or delete existing ones. To interact with the system, the user must open an Access Point (AP). Through the AP, the user can examine or modify the current configuration of the abstract machine, or run a program in the SAM. When no further interaction with the Swarm system is necessary, the user can close the Access Point. Alterations in the Swarm system configuration, newly created nodes, and any information stored in the persistent workspace of data nodes are all kept...
in the permanent structure of the Swarm Abstract Machine, even after the user closes his Unix session.

Figure 5.1 - Structure of the prototype Swarm system

KEYS:
RM = Routing Module
PM = Processing Module
AP = Access Point

For implementation in a physically distributed environment (e.g. a network of workstations), each host machine would hold the structure described above and the RMs would be interconnected using the Unix UDP/IP or TCP/IP communication services. The following sections detail the main components of the prototype Swarm system.
5.1.1 The Routing Module

The first step to create a Swarm Abstract Machine is the installation of the Routing Module as a background Unix process in the host computer. The RM acts as the Swarm daemon, responsible for the routing of agents during program execution and creation or removal of PMs during system configuration. In the present, single-workstation version of the system, all agent traffic between PMs is dealt by the local RM. In the planned extended version, RMs in different computers will communicate with each other, routing agent traffic between local and remote PMs.

The prototype Swarm Abstract Machine operates in two distinct modes: command and execution. In the command mode, the user can use system commands to check the system configuration, make alterations to it, or run a task. The RM recognises seven commands, which must be issued from an AP:

1. list system configuration: RM returns a list of existing PM IDs;
2. get nodes from a PM: given a PM ID, RM downloads (to AP) the data nodes contained in that PM;
3. put nodes in a PM: given a PM ID, RM uploads (from AP) data nodes to that PM, overwriting all nodes previously stored there;
4. create a PM: RM creates a new PM and returns the corresponding ID;
5. remove a PM: given a PM ID, RM removes the corresponding PM;
6. run task: RM broadcasts the program to all PMs and starts execution, managing all agent traffic and detecting program termination;
7. disconnect AP: the connection between RM and AP is closed and AP is terminated.

The two first commands (list configuration, get nodes) permit the user to verify the present configuration of the abstract machine. The next three commands (put nodes, create PM, remove PM) are used to alter the SAM configuration, by changing the contents (i.e. data nodes) of a PM, or by adding or dropping PMs from the system. The run task command loads an executable program (in machine code) into the abstract machine and starts its execution, informing the AP when the task terminates. The last command terminates the AP, after closing
its connection with the system. The detailed protocol of each system command can be found in Appendix C.

![Diagram of agent formats](image)

(a) Outgoing agent format

(b) Incoming agent format

Figure 5.2 - Normal agent formats

When the RM starts the execution of a loaded program, the system changes from command mode to execution mode. During execution mode, the RM function is to route agents between modules. Agents created by the executing program, called normal agents, are put in the PM (or AP) output queue in the packed format shown in Figure 5.2.a. The RM continuously scans all output queues, looking for agents, until it gets one. It then reads the packed agent in a temporary buffer and generates one single agent (format in Figure 5.2.b) for each destination. To reduce communication traffic and save buffer space, only the agent workspace block that goes from AW[0] until the last AW occupied position is transmitted. Additionally, the RM also routes control agents. A control agent differs from a normal agent in its format and nature. First, control agents have a minimum, fixed size, equal to the head of a normal agent (16 bytes in the present implementation). Second, control agents are not directly created by the running program, being automatically generated by the Swarm system for the purpose of automatic termination detection. Termination detection and control agents are further discussed in Section 5.1.4.

When the user first installs a Swarm Abstract Machine, or initiates a new one, the Swarm environment is completely empty (with no PMs). The user must then start from scratch, manually creating the necessary PMs and loading data nodes into them. Nodes can be dynamically created by an executing program (using the create node instruction), or can be statically loaded from a previous configuration saved in files (see Section 5.1.3). Each node address is a combination of PM ID and node ID, and point to a specific position inside a
specific PM. Therefore, each configuration file must be loaded in a PM with exactly the same ID of the original PM, and all files of that configuration must be loaded to recreate the original abstract machine configuration, otherwise void references will be present in the system.

The internal organisation of a Routing Module is shown in Figure 5.3. Unix pipes are used for communication between the RM and the PMs (including AP). Transpareently to the user, the RM keeps a built-in structure known as the address book, which has one entry per active module (PMs and AP). Entries are indexed by the ID of the PM in the Swarm system, and contain the Unix process identifier (pid) of the corresponding module and the corresponding pipe file ids for input and output. When the user (via the AP) issues a command to create a new PM, the Routing Module creates a pair of ordinary pipes for communication and forks a new PM process (as a Unix background process). The newly created PM then receives the lowest available PM ID and a corresponding entry is created in the address book. If necessary, the address book is automatically extended to accommodate new entries, up to a maximum of 65,533 active PMs in the system. In the present implementation, the PM ID value 0 (zero) is reserved for the AP and only one active AP is allowed at any one time in the system. Unlike PMs, the AP is not forked by the RM, being started directly by the user as a foreground Unix process running from an arbitrary terminal window in the host machine. When created, the AP communicates with the RM via a pair of Unix named pipes (FIFOs), maintained by the RM.

Figure 5.3 - RM internal organisation

During command mode, the information held in the address book is used by the RM to transfer data between AP and PMs, and to delete (kill) PMs. During execution mode, the same information is used for routing agents between nodes and for the signaling related to abnormal termination. An additional advantage of the address book, relevant for a subsequent networked
version of the prototype, is the provision of location transparency for access to PMs. All specific (Unix related) information of each module is hidden behind the Swarm PM ID. Consequently, a PM could be moved from a host computer to another, keeping its PM ID and having the corresponding entry on each RM address book updated (by replacing the PM input and output fids and Unix pid with the corresponding values for the remote RM that holds the moved PM). This characteristic can be used, for example, to improve load balance among the host computers.

5.1.2 The Processing Module

In the parallel environment of a Swarm Abstract Machine, a Processing Module (PM) is the equivalent of a processor element (PE) in a multicomputer. The major function of a PM is to store and manage a set of Swarm data nodes. The exact distribution of data nodes among the available PMs is defined by the user: directly, by loading a configuration file; or indirectly, by executing a program that creates new nodes (or destroys existing ones). The internal organisation of a PM is shown in Figure 5.4.

Figure 5.4 - PM internal organisation
PMs are created by explicit command from the user, when the system is in command mode. A PM Unix process is forked from the RM, a pair of Unix pipes is connected to it, and the PM persistent structures are initialised as empty. The PM persistent structures are the nodal store, that holds the data nodes, and the local directory, which keeps lists of local node addresses indexed by the values of node keys. If the user loads a configuration file into a PM, these structures are automatically updated by the PM, dynamically changing size to accommodate the current number of nodes.

For execution of a task (execution mode), a PM must create and manage a number of additional, volatile structures: the program store, the nodal transient workspaces, the agent execution buffer, the suspending buffer, and the spatial table. The program store holds the program (machine) code, the specific execution parameters for the task (e.g. activation of automatic termination detection, task abortion on runtime error), and the list of constant data. A transient workspace is dynamically attached to each node in the PM the first time it is accessed by the program, with initial size and contents determined by the task execution parameters. Further accesses to positions beyond the initial size cause automatic extension of that transient workspace. The agent execution buffer holds a complete agent during its execution, comprising its header (task ID, size, type), status word, workspace, and destination list. The suspending buffer is used to store suspended agents, which are on hold waiting for a specific event number. When a specific event is signalised by an agent executing in the PM, the agents suspended on that event are automatically re-scheduled for execution in the node where the signal was produced. Re-scheduled agents have priority of execution over agents from the input queue. The spatial table holds the address of the console node (in the AP) and the address of another PM in the system (or the empty value, if there is only one PM). All these volatile structures are removed from the PM at the termination of the current task. Appendix C details the implementation of the PM persistent and volatile structures.

During execution mode, each PM repeats the same basic sequence of actions shown in Figure 5.5: get (if any) an awakened agent from the suspending buffer or an incoming agent from the input queue; execute it; kill the agent (if it terminates), or suspend it (if it wants to wait), or place agent and destination list in the output queue (if it spawns with a not empty destination list). Interleaved with these actions each PM performs the termination detection routine described in Section 5.1.4.
pm_task_execution
{
    while (task is not terminated) {
        if there is a suspended agent, get it;
        else try to get agent from input queue;
        if got an agent {
            while (not stopped) {
                fetch instruction;
                execute instruction;
            }
        } /* after agent stops */
        if opcode == STOP, do nothing;
        if opcode == TERM, send 'termination agent' to RM;
        if opcode == SUSP, put agent in the Suspending Buffer;
        if opcode == SPWN, put packed agent into output queue;
    }
}

Figure 5.5 - PM execution routine

As previously explained, normal agents are placed in the PM output queue in a packed format (Figure 5.2): head + block of occupied workspace positions + destination list. It is the job of the RM to unfold this packet into a number of individual agents, one per destination address, and route them to the corresponding PM input queue. Therefore, when the PM gets an agent from its input queue, it can be loaded into the execution buffer with minimum effort: a check if the destination node exists; the restore of workspace position 255, the IP save position, which is transported in position 0 (zero); and the initialisation of unused positions to EMPTY and position 0 to zero. This division of work represents a compromise between the two extreme alternatives: having individual agents directly inserted in the PM output queue during spawning; or having a copy of the packed agent delivered to each PM that contains a destination node. The former would waste buffer space and would not save much work from the RM, which would still have to route each individual agent. The latter would save buffer space both for output and input, but would involve a much more complicated algorithm for the RM to scan the destination list, generating a single copy to each PM. The PM routine for inputting an agent would also become more complex, as it would have to decompose the incoming packed agent into individual ones, discarding agents with addresses outside the local PM. Possible problems with buffer space are the subject of a specific discussion in Chapter 6.

5.1.3 The Access Point

The Access Point (AP) is the interface between the user and the Swarm environment. In command mode, the AP sends to the RM the commands issued by the user and displays the corresponding messages. In execution mode, the AP operates as an I/O point for the running
task. During command mode, the AP presents to the user the command menu shown in Figure 5.6. Some commands, as `get nodes`, `put nodes` and `run task`, imply in access to specific files. For the `get nodes` command, the AP generates a binary file containing an image of the nodal store of the specified PM. The contents of the PM nodal store are not affected. A file in the same format can be used to load new contents in a PM nodal store, when the `put nodes` command is issued. A PM can be cleared by loading into it a configuration file containing no nodes. Conversions between the binary format and a readable, ASCII form are performed by an additional program provided with the Swarm system. The `run task` command requires binary code file, previously translated from a user assembly program. Apart from the executable sequence of machine instructions, this file specifies task parameters and the constant data required by the program. Formats of all these files are detailed in Appendix C.

Providing an extra source of information to trace the behaviour of the system, all PMs and the RM maintain log files. During user interaction with the system (command and execution modes), RM and PMs put entries in respective log files, identifying major actions like creation of a PM, start of a task, and termination or abortion of a task. These log files have ASCII text formats and can be later examined by the user.

**SWARM Experimental Environment**

1 - LIST PMs  
2 - CREATE A PM  
3 - DELETE A PM  
4 - GET NODES FROM A PM  
5 - LOAD NODES TO A PM  
6 - RUN A TASK  
7 - CLOSE AP

Choose one of the options:

Figure 5.6 - AP command menu

In execution mode, the AP works like a restricted PM, with no nodes except the special console node. All AP structures are volatile and there is no persistent workspace, local directory or suspending buffer. The AP only holds a transient workspace for the console node, a program store, and an agent execution buffer. As a consequence, some Swarm instructions like suspend or awake an agent, create or destroy nodes, or manipulate Link Table or Local Directory, are completely void when executing in the console node. Instead, a set of primitive I/O operations are possible, as described in Section 4.3.8. Another difference is that
the AP acts as the initiator of the distributed termination detection cycles (see Section 5.1.4 for more details). Figure 5.7 shows the internal organisation of an AP.

Figure 5.7 - AP internal organisation

As explained in Section 5.1.1, the AP is not forked by the RM, but started by the user, from the terminal he is currently using. Communication between the AP and the RM happens through a pair of named pipes, which must be opened in a proper sequence by both sides. The alternative approach of using ordinary pipes and forking the AP from the RM was rejected, due to the idiosyncrasies of the Unix system about control terminals and the need for the AP to access a terminal for interface with the user. Once the AP starts execution and the named pipes are opened, they behave like ordinary pipes.

5.1.4 Termination Detection

As a result of the asynchronous environment of a Swarm Abstract Machine, many agents can still be executing when some output is delivered to the user via the console node. The Swarm prototype provides a minimum termination detection service that informs the user when a task terminates. A task executing in the prototype Swarm system terminates if one of the following situations occurs:

- **abortion by the user**: from the AP, the user issues an abort task command (by pressing CTRL-C);
- **abortion by the program**: an agent executes an abort task instruction;
• **normal termination**: there are no more active (i.e. executing) agents in the environment (excluding agents in suspending buffers).

Task abortion due to an exception (runtime error) is a non-implemented feature in the present version of the prototype. In the case of forced termination (the two first situations above), a special agent is sent to the RM and then broadcasted to all modules, removing all transient structures and returning the whole system to command mode. The same does not happen in normal termination. In this case, the user would have to include in his program extra code that detects the global termination and executes the *abort task* instruction. This can be trivial in some applications, but complex and cumbersome in others. For this reason, the Swarm system can automatically detect normal termination, producing the special agent and clearing the system.

The termination detection algorithm presently used in the Swarm prototype is shown in Figure 5.8. It is a simplified version of the distributed termination detection algorithm developed by Mattern [1990], chosen because of its characteristics: support to asynchronous communication over non-FIFO channels; short control messages; a fixed number of simple variables. In the Swarm environment the termination detection algorithm is executed by the AP and PMs, superimposed on the normal computation. The AP and each PM keep an *agent counter*, a variable that counts the number of normal agents processed minus the number of normal agents produced (by spawning). Only the AP can initiate a termination detection cycle.

Termination detection starts when the AP (the *initiator*) stays idle for the number of seconds specified by the executing program. The default is 1 second, but the user can modify this value using a proper task directive (see Appendix B) at the beginning of the assembly code. A value of zero will disable the automatic termination detection mechanism (leaving this task to the user program). The AP then issues a termination inquiry control agent to the RM, which broadcasts it to all PMs. Each PM replies by sending an echo control agent that carries the value of its agent counter. The AP adds all received values, including its own counter, to obtain the result of the inquiry. If the result is zero, no agent is pending to be received, no agent is executing, and the task is terminated. The AP then issues the terminating agent to RM, finishing the task. If the result is non-zero, there are still agents pending to be executed. In this case, the AP aborts the current termination detection cycle and re-enables the time count for a new cycle.
ap_termination_detection_routine
{
    int ACOUNTER, /* normal agent counter (increm. on sending, decr. on receiving) */
    ECHO_COUNTER, /* echo agent counter (increm. on receiving TECH agent) */
    ACTIVITY, /* 'activity' flag, reset when TD begins and set by any incoming
                normal agent */
    TD_IN_PROG, /* 'termination detection in progress' flag */
    ECHO_ACCUM, /* accumulator for the (normal agent) counter values brought by echo
                agents */
    TDELAY;    /* delay to start TD (avoids excessive restarts) */

    initialise ACOUNTER to zero;
    initialise TD_IN_PROG flag to false;
    initialise TDELAY to default value;

    while(not terminated) {
        start timer with delay == TDELAY;
        block trying to get an agent from the input queue;
        disable timer;
        /* if timer reaches TDELAY seconds, it calls a routine that
           sends a TENG agent to RM */

        if got a normal agent {
            decrement ACOUNTER;
            set ACTIVITY flag to true;
            execute agent;
            increment ACOUNTER for each agent sent to output (ACOUNTER += DL size);
        }
        if got a control agent {
            if a termination agent (TTSK) {
                acknowledge termination by sending a TACK agent to RM;
                free all structures dynamically allocated for task execution;
                return;
            }
            if a termination echo agent (TECH) {
                if it is a new TD cycle (TD_IN_PROG == zero) {
                    initialise variables;
                    set TD_IN_PROG flag to true;
                    set ACTIVITY flag to false;
                    initialise ECHO_ACCUM to the value of the local ACOUNTER;
                    initialise ECHO_COUNTER to zero;
                }
                accumulate into ECHO_ACCUM the value brought by TECH agent;
                increment ECHO_COUNTER;
                if received echoes from all PMs {
                    if ACTIVITY is zero and ECHO_ACCUM is zero,
                    send TTSK agent to RM;
                    else /* cancel TD */
                    set TD_IN_PROG to zero;
                }
            }
        }
    }
}

pm_termination_detection_routine
{
    int ACOUNTER; /* normal agent counter (increm. on sending, decr. on receiving) */

    initialise ACOUNTER to zero;

    while(not terminated) {
        if there is a suspended agent, get it;
        else block trying to get an agent from the input queue;

        if got a normal agent {
            if was not a suspended agent, decrement ACOUNTER;
            execute agent;
            increment ACOUNTER for each agent sent to output (ACOUNTER += DL size);
        }
        if got a control agent {
            if a termination agent (TTSK) {
                acknowledge termination by sending a TACK agent to RM;
            }
        }
    }
}
free all structures dynamically allocated for task execution;
return;
}
if a termination enquiry agent (TENQ)
assembly and send TECH agent to the initiator;
}
}

Figure 5.8 - Termination Detection Algorithm

The Swarm prototype uses control agents to implement the above algorithm. A control agent is created and managed by the system, transparently to the user. Control agents follow the format shown in Figure 5.9. Contrasting with normal agents, control agents have a fixed size, equal to a normal agent head (i.e. four 32-bit words = 16 bytes). The first word identifies it as a control agent and the second word is the task ID (not used in this version). The third word holds the address of the source, which can be a node or a PM (the root node address is used in this case). The last word only carries data when the agent is an echo control agent (carrying the value of a PM agent counter in reply to a termination inquiry).

Figure 5.9 - Control agents

5.2 Example Application: Bandwidth Reservation

All instructions and internal functions of the Swarm prototype were individually tested for correct operation. Nevertheless, absolute proof of total correctness cannot be obtained by this method. The development and execution of application programs allow a more thorough and realistic evaluation of the system, by testing how different functions work together and
exposing the adequacy of the instruction set. The example detailed in this section is a program for bandwidth reservation in communication networks.

Figure 5.10 - A communication network

5.2.1 Description of the Problem

The example application is a program which reserves a path of a specific bandwidth between two given nodes in a communication network of arbitrary topology. If a path with the chosen value of bandwidth cannot be found, one with the closest possible value is returned. If no acceptable path can be found, a failure is returned. The communication network has the general aspect shown in Figure 5.10. Each node is a router that communicates with neighbour nodes using a pair of unidirectional links. Each link has a total bandwidth (given in packets per second) that can be shared among different connections. A Link Table in each node stores the current value of available bandwidth to each neighbour node. Unidirectional connections
between any two distinct nodes, one being the *source* node and the other the *destination* node, can be established by finding a path and reserving some bandwidth along this path. Each node holds a Routing Table with one entry per established connection.

5.2.2 Description of the Algorithm

The basic algorithm used to solve the bandwidth reservation problem using a Swarm Abstract Machine is shown in Figure 5.11. Given a source node, a destination node and a bandwidth value, it starts from the destination node, constructing a spanning tree that contains *best* paths to all nodes. A *best* path is one that follows the criteria: it offers a bandwidth that is either equal to the requested value or the closest value that can be found; and it has the shortest possible length (in number of hops) for that bandwidth. If only a path with zero bandwidth can be found, the algorithm reserves no path. Otherwise, a reservation phase occurs, committing the changes in the Link Tables and creating proper entries in the Routing Table of each node along the path. A final phase cleans up all temporary variables and prepares the system for another run of bandwidth reservation. It must be noted that only one reservation can occur per time, which can be controlled by an additional election algorithm (e.g. a token-passing election algorithm).

1. Obtain source node address, destination node address, and requested bandwidth.
2. Move to the destination node.
3. Construct a spanning tree that maximises path bandwidth and provides the shortest possible length.
4. If obtained path bandwidth is not zero, follow the path, starting from the source node, committing the reservation (updating Link Tables and Routing Tables).
5. From the destination node, construct a spanning tree, clearing all temporary variables used for the reservation.

Figure 5.11 - Algorithm for bandwidth reservation

Both steps 3 and 5 of the basic algorithm above require the construction of a spanning tree. The method used here is adapted from the *shortest path* algorithm by Chandy and Misra [1982]. For implementation in the Swarm system, the basic algorithm was broken into two programs. This simplifies the programming task, as the Swarm automatic termination detection is used to identify the end of the spanning tree construction.

Figure 5.12 shows the persistent nodal variables used by both programs. Apart from the default node address (%NADDR) and node key (%NKEY) positions in the Persistent Workspace of each node, another five positions are used for the variables that control the construction of the

84
paths (%NEXTND, %NBANDW, %UNLENGTH) and that hold parameters for the Program 2 (%SAVEKEY, %SAVEDST). The Routing Table in each node used subsequent positions in the Persistent Workspace. Position %ROUTTBSZ holds the current size of the Routing Table, in number of entries. Each entry has five fields, that describe the source node, destination node, next node in the path, bandwidth, and remaining path length to destination. Additionally, the Link Table of each Swarm node is used to store the bandwidth-per-link information. The agent workspace positions will be discussed along with the programs.

Program 1 comprises steps 1, 2 and 3 of the algorithm in Figure 5.11. Instead of directly examining the programs in their assembly language version, corresponding pseudo-C language versions are used to simplify the descriptions. The final, assembly language version of both programs can be seen on Appendix D. The pseudo-code for Program 1 is shown in Figure 5.13. Processing always starts, in Swarm, from the console node. The program segment from lines 8-15 inputs the addresses of the source and the destination, and the requested bandwidth for the path. This bandwidth value is not mandatory and the program will try to obtain the closest value if it is not possible to obtain the requested one. Having obtained the destination address, the agent moves to it in lines 14-15. The segment from lines 19-35 is the main loop that builds the spanning tree. After the initialisation of lines 16-18, the small loop of lines 20-23 inserts the addresses of neighbour nodes into the destination list, excluding the predecessor node. Lines 24-25 update the variable that carries the predecessor address and spawns to all nodes in the destination list. Reaching a node, each agent increments the path length (line 26), obtains the
bandwidth to the predecessor node (line 27) and compares this with the requested bandwidth (line 28), lowering the request if necessary. If the incoming agent has a better bandwidth value than the one left by a previous agent, or if the bandwidth value is the same but the path length is smaller, the agent updates the values stored in the node with the ones it brought (lines 29-33). If the values brought by the agent are no better than the stored ones, there is no update and the agent dies (line 34). In the case of success, the agent re-starts the loop, obtaining neighbour addresses, spawning and updating nodes until reaching the source node, where the loop stops (line 36). An agent that reaches the source node and is successful in updating the nodal variables, proceeds with the program segment of lines 36-40, saving the values of destination address and node key already if they were not already saved by a previous agent. The agent then stops and dies at line 41. The task terminates when all agents die, leaving a path from source to destination node.

/* Bandwidth Reservation in a Communication Network - Program 1 */

00 bandwidth_reservation_prog_1()
01 {
02  int 0SRC, /* address of source node */
03  0DST, /* address of destination node */
04  0PRED, /* address of predecessor node */
05  0BANDW, /* required bandwidth */
06  0PATHL, /* length of the path */
07  0AUX; /* auxiliary var for data transfers */
08
09  output_ascii("\nSOURCE NODE: ");
10  0SRC = input_hex();
11  output_ascii("\nDESTINATION NODE: ");
12  0DST = input_hex();
13  output_ascii("\nBANDWIDTH: ");
14  0BANDW = input_hex();
15  insert_dil(0DST);
16  spawn_agent;
17  0PRED = 0;
18  0PATHL = 0;
19  0NBANDW = MAX_POS_NO;
20  do {
21      while( (0AUX = lookup lt(any_key)) != EMPTY) {
22        if(0AUX == 0PRED) continue;
23        insert_dil(0AUX);
24      }
25  0PRED = 0NADDR;
26  spawn_agent;
27  (0PATHL)++;
28  0AUX = reverse_lookup_lt(0PRED);
29  if((BANDW > 0AUX) & @BANDW = 0AUX;
30  if((BANDW > 0NBANDW) || ((BANDW == 0NBANDW) && (0PATHL < 0NLENGTH))) {
31      0NEXTND = 0PRED;
32      0NBANDW = 0BANDW;
33      0NLENGTH = 0PATHL;
34  } else stop_agent;
35  } while(0NADDR != 0SRC);
36  if(0NKEY != MARK_VALUE) {
37    %SAVEKEY = 0NKEY;
38    0NKEY = MARK_VALUE;
39    %SAVEDST = 0DST;
40  }
41 }

Figure 5.13 - Bandwidth Reservation - Program 1
/* Bandwidth Reservation in a Communication Network - Program 2 */

00 bandwidth_reservation_prog_2() {
01 int @SRC, /* address of source node */
02 @DST, /* address of destination node */
03 @PREDE, /* address of predecessor node */
04 @BANDW, /* required bandwidth */
05 @PATHL, /* length of the path */
06 @NEXT, /* address of next node in the path */
07 @NEXTBW, /* bandwidth for next node */
08 @AUX; /* auxiliary var for data transfers */

09 insert_dl(BROADCAST_ADDR);
10 spawn_agent;
11 @SRC = lookup_ldir(MARK_VALUE);
12 if(@SRC == EMPTY) stop_agent;
13 insert_dl(@SRC);
14 spawn_agent;
15 %KEYKEY = %SAVEDST;
16 @DST = %SAVEDST;
17 @PREDE = 0;
18 @BANDW = %NBANDW;
19 @PATHL = %LENGTH;
20 if(@BANDW) {
21 do {
22 /* update Link Table entries */
23 @NEXT = %NEXTND;
24 @NEXTBW = reverse_lookup_ltb(@NEXT);
25 delete_ltb(@NEXT, @NEXTBW);
26 @NEXTBW = @BANDW;
27 insert_ltb(@NEXT, @NEXTBW);
28 /* insert new entry in Routing Table */
29 (%ROUTTBSZ)++;
30 %ROUTTBSZ.SRCND = @SRC;
31 %ROUTTBSZ.DSTND = @DST;
32 %ROUTTBSZ.NXTND = @NEXT;
33 %ROUTTBSZ.BANDW = @BANDW;
34 %ROUTTBSZ.LENGTH = @PATHL;
35 /* move to Next Node */
36 @PATHL--; /* decrement path length */
37 insert_dl(@NEXT);
38 spawn_agent;
39 } while(%NADDR != @DST);
40 } else {
41 do {
42 insert_dl(@DST);
43 spawn_agent;
44 } while(%NBANDW != @DST);
45 %NBANDW = 0; /* reset %NBANDW at Destination Node */
46 insert_dl(lookup_st(0)); /* insert Console Node Address in Dest List */
47 @PREDE = 0;
48 do {
49 while((@AUX = lookup_ltb(any_key)) != EMPTY) {
50 if(@AUX == @PREDE) continue;
51 insert_dl(@AUX);
52 }
53 @PREDE = %NADDR;
54 spawn_agent;
55 if(%NADDR == lookup_st(0)) {
56 if(!(@BANDW)) output_ascii("SORRY, NO PATH AVAILABLE");
57 else {
58 output_ascii("PATH RESERVED WITH BW ");
59 output_decimal(@BANDW);
60 }
61 stop_agent;
62 }
63 if(!(@NBANDW)) stop_agent; /* node already cleaned */
64 else {
65 @BANDW = 0;
66 @NBANDW = 0;
67 @NLENGTH = 0;
68 }
69 } while(1);
70 }

Figure 5.14 - Bandwidth Reservation - Program 2
Figure 5.14 has the pseudo-code for Program 2. Instead of directly requesting the addresses of source and destination node from the user, Program 2 finds them using the information left by Program 1. It starts (lines 10-15) by broadcasting agents to each PM root node. Once in the root node, the agent inspects the Local Directory, looking for a special mark key left by Program 1. If a node with that key is not found, the agent dies. As the key is guaranteed to be unique in the system (what is the user responsibility), only one agent will succeed, obtaining the node address and moving to it.

Reaching the node address, the agent initialises the variables (lines 16-20) and checks if the obtained bandwidth is zero (line 21). A zero value bandwidth skips the reservation loop and moves the agent directly to the destination node (lines 42-43), to start the cleanup phase. A bandwidth value that is greater than zero starts the loop that commits the path reservation (lines 22-40). First, the Link Table entries corresponding to the next node in the path have their values of available bandwidth subtracted from the reserved bandwidth (lines 24-28). Then, a new entry is created in the Routing Table and the proper values are inserted (lines 30-35). The agent moves to the next node in the path and re-starts the loop (lines 37-39). The loop exits when the agent reaches the destination node (line 40).

In the destination node, the agent resets the %NBANDW variable (line 45) and prepares to split into two different directions. The address of the console node is inserted into the destination list (line 46), followed by the addresses of all neighbours (lines 49-52). The agent spawns (lines 53-54). The agent directed to the console node displays there the corresponding message and dies (lines 55-62). The other agents cover the whole network, building a spanning tree (lines 49-54). At each visited node, the agent resets the variables used in Program 1 (lines 65-67). Reaching an already visited node, the agent stops (line 63). When all agents die, the task terminates and the user is notified by the Swarm system.

5.2.3 Execution Tests

Two different mappings of the network of Figure 5.10 were used to test the execution of the bandwidth reservation programs in the Swarm prototype. The first mapping placed all the nodes in a single PM. The second mapping divided the nodes among three PMs. For each mapping, the programs were executed and the contents of the PM nodal stores downloaded to files to observe the results. Figure 5.15 presents the result from an execution of the bandwidth
reservation program using the three-PM mapping. Requested to find a path of bandwidth 100 between nodes $a$ (the source) and $j$ (the destination), the program found the path $a-d-f-i-j$ shown in bold, updating the corresponding persistent variables (link tables and routing tables) on each affected node. The respective configuration files are listed in the Appendix D, showing the contents of each PM before (corresponding to Figure 5.10) and after (corresponding to Figure 5.15) the execution of the bandwidth reservation programs.

The programs were processed in a lightly loaded Sun SparcStation IPX (a Sun 4 architecture with a 40 MHz Sparc processor). First, the programs were executed using a special version of the Swarm system, which produced a file with information about the internal operation of each module. This helped in debugging the programs and verifying if the internal operation of the prototype was corrected. Subsequently, the programs were processed using the normal version of the prototype and the executions were timed, producing the results shown in
Figures 5.16 and 5.17. In all cases, the executions were for the reservation of a bandwidth of 100 between nodes a and j in the network of Figure 5.10.

**Program 1:**
Elapsed Time: 14 seconds (incl. time spent in I/O)
Total CPU time: 6/60 s (= 100 ms)
= 2/60 + 2/60 + 2/60 (AP + PM + RM)
Number of normal agents routed: 22
Number of control agents routed: 5

**Program 2:**
Elapsed Time: 1 second (incl. time spent in I/O)
Total CPU time: 7/60 s (= 117 ms)
= 2/60 + 4/60 + 1/60 (AP + PM + RM)
Number of normal agents routed: 26
Number of control agents routed: 5

**Figure 5.16 - Statistics for the bandwidth reservation program (one-PM)**

**Program 1:**
Elapsed Time: 11 seconds (incl. time spent in I/O)
Total CPU time: 11/60 s (= 183 ms)
= 0/60 + 2/60 + 3/60 + 3/60 + 3/60 (AP + PM1 + PM2 + PM3 + RM)
Number of normal agents routed: 22
Number of control agents routed: 11

**Program 2:**
Elapsed Time: 2 seconds (incl. time spent in I/O)
Total CPU time: 11/60 s (= 183 ms)
= 1/60 + 1/60 + 5/60 + 2/60 + 2/60 (AP + PM1 + PM2 + PM3 + RM)
Number of normal agents routed: 28
Number of control agents routed: 11

**Figure 5.17 - Statistics for the bandwidth reservation program (three-PM)**

As shown by the results, the coarse timing resolution provided by the Unix system calls (minimum of one *tick* = 1/60 second = 16.7 ms) did not permit precise values. Nevertheless, further information was provided by profiling the execution of each module, using the Unix utility gprof together with the corresponding compilation option of the gcc compiler. The second bandwidth program was executed for a one-PM mapping, resulting in the profiles displayed in Figure 5.18. Clearly, processing time in the two modules was dominated by I/O operations. However, due to the short processing time of the program and the coarse timing resolution of the Unix system, no precise timing information could be obtained.

**PM Profiling:**
%time operation
70.0% file manipulation (create, open, close, write)
20.0% pipe I/O
10.0% task initialisation
0.0% others

**RM Profiling:**
%time operation
44.4% pipe I/O
33.3% file manipulation (create, open, close, write)
11.1% creation of FIFOs
11.1% module termination
0.0% others

**Figure 5.18 - Profiling results**
5.3 Performance Tests

The execution of the example program, described above, permitted the observation and verification of the internal operation of the prototype, but did not provide enough information about performance. Although absolute performance was not the main consideration when developing the prototype, more precise figures for the execution costs of different operations were necessary to identify possible points for improvement, providing guidelines for future implementations. With this purpose, a set of test programs was developed to time the execution of main operations in the Swarm architecture: spawnings, node creations, arithmetic/logical operations, load/store operations, branches and jumps, subroutine calls, and table lookups. Each test program was executed three times in a lightly loaded Sun SparcStation IPX (with a 40 MHz Sparc processor) and the results obtained were averaged.

The test program of Figure 5.19 was developed to obtain the communication costs involved in transporting an agent between two nodes. The test comprised moving an agent between the AP console node and a PM root node for 10,000 times. The program initialises AW[1] with the address of the PM 1 root node and sets the size of the agent workspace to be carried by the agent. Because the Swarm prototype loads the agent with the workspace block that goes from AW[0] until the last used position (regardless of the contents of the positions between them), it was only necessary to write a value (the number zero, in this case) to the highest desired position.

Tests were executed for three different sizes of the agent workspace: 2 words (the minimum possible for this test), 128 words, and 255 words (the possible maximum, as position AW[255] is always transported in the place of AW[0], as previously explained in Sections 5.1.1 and 5.1.2). Interference from other internal mechanisms was avoided by disabling the automatic
termination detection, and the termination was obtained by the execution of the TERM instruction at the end of the program. Additionally, with the objective of eliminating the effect of repeated executions of a jump instruction, the propagation sequence was simply repeated as in-line code 5,000 times in the program. The resulting program size was 80 Kbytes.

For each size of agent workspace the test was executed three times and the obtained results averaged, producing the values shown in Figure 5.20. The tests produced total cpu times of 14.7 s (883/60 s), 14.9 s (893/60 s), and 15.3 s (916/60 s), respectively for agent workspace sizes of 2, 128, and 255 words. Comparing the results for different AW sizes, the increase in the total processing time was around 4% (0.6 s) from the minimum case to the maximum case (i.e. from 2 to 255 words), showing that the time required to route an agent was almost independent from the size of the payload (the agent workspace). The average throughput for the three tests (AW size = 2, 128, and 255) was 10,000 agents routed in 15.0 s, what gives a rate of 667 agents/s or a latency of 1.5 ms/agent. Specifically for the tests above, the time spent in initialisation and termination of the task accounted for less than 1% of the total processing time. To provide a base for comparison, a null RPC was executed under similar conditions (client and server processes running in the same light loaded Sun SparcStation IPX used in the previous test, using UDP as communication protocol). The average figure obtained from three executions of the null RPC was 1.83 ms for the round-trip delay (both directions), which gives approximately 0.9 ms in each direction (equivalent to 60% of the time for an agent transfer between two nodes).

<table>
<thead>
<tr>
<th>agents routed</th>
<th>10,000 normal agents + 2 control agents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AW size</strong></td>
<td><strong>2</strong></td>
</tr>
<tr>
<td><strong>module</strong></td>
<td><strong>init</strong></td>
</tr>
<tr>
<td><strong>module</strong></td>
<td><strong>AP</strong></td>
</tr>
<tr>
<td><strong>init</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>exec</strong></td>
<td>301</td>
</tr>
<tr>
<td><strong>term</strong></td>
<td>0</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>303</td>
</tr>
<tr>
<td><strong>elapsed</strong></td>
<td>15 seconds</td>
</tr>
</tbody>
</table>

**AW size** = size of the workspace carried by the agent (in number of 32-bit words)
**init** = task initialisation time (cpu time in ticks, 1 tick = 1/60 second)
**exec** = task execution time (cpu time in ticks, 1 tick = 1/60 second)
**term** = task termination time (cpu time in ticks, 1 tick = 1/60 second)
**TOTAL** = total task time for the module (cpu time in ticks, 1 tick = 1/60 second)
**elapsed** = elapsed time for the test

Figure 5.20 - Results of communication tests

The test program shown in Figure 5.21 was used to time the node creation in Swarm. After moving to the root node of PM 1, the program executes a **create node** operation 1,000 times and terminates with the TERM instruction. As in the communication test program previously
described, the automatic termination detection was disabled to avoid interference and the create instruction was repeated 1,000 times as in-line code, to eliminate the need of jump instructions. The average result from three executions of the test produced a figure of 500 µs for each node creation.

```plaintext
$SET TDDL 0   * disable termination detection
SETH A1 #0001 * AW[1]:= address of PM 1 root node
INDL A1      * insert address in destination list
SPWN         * move to PM 1 root node
CRND A1 A0   * create node
              * (repeated 1,000 times)
TERM         * terminate task
```

Figure 5.21 - Node creation test

The following tests timed the execution of different kinds of instructions found in the Swarm architecture. Each test used the basic loop shown in Figure 5.22, and the result was the average figure for three executions of the test in a lightly loaded SparcStation IPX. For these tests, PM 1 was loaded with the graph shown in Figure 5.10, and the loops executed in node 6, which corresponds to node f of Figure 5.10 (the node with more connections).

```plaintext
$SET TDDL 0   * disable termination detection
SETH A3 #0001 * AW[3]:= address of node 6 in PM 1
SETL A3 #0006 *
INDL A3      *
SPWN         *
SETH A3 #0001 * AW[3]:= 10,000 (number of iterations)
SETE A3 #F000 *

LOOP: instruction(s) to be timed
SUBI A3 A3 #1 * decrement loop counter (AW[3])
EQLA A2 A3 A0 * is loop counter == zero?
BIFI A2 LOOP  * if not, repeat loop
TERM         * terminate task
```

Figure 5.22 - Basic loop for tests

Figure 5.23 displays the obtained results. The first test (Test 0) timed the empty loop. The following tests used the cpu time (process + Unix system calls) for the PM while executing the program, subtracted from the time for an empty loop, and divided by the number of instructions timed (more than one in case of a block). Test 1 timed all 17 immediate arithmetic/logical instructions in a block. Test 2 and test 3 compared the execution times for the two versions of an add instruction (immediate and variable). Test 4 timed the two set instructions, that load a 16-bit immediate value. Test 5 timed all six load/store instructions that move data between the agent workspace and the workspaces available in Swarm. Test 6 produced the execution times for four different control instructions (each control instruction was set to pass control to the next instruction). Test 7 timed empty subroutine calls. Finally, test 8 and test 9 timed lookups...
to the nodal Link Table and to the PM Local Directory. The average of all timed instructions (from test 1 to test 9) gave an execution time of 7.87 μs.

Test 0: Empty loop
21.3 ps/loop

Test 1: Block with all immediate arithmetic/logical instructions (from ADDI to SARI)
8.88 μs/instruction

Test 2: Immediate Add instruction (ADDI)
7.17 μs/instruction

Test 3: Variable Add instruction (ADDA)
7.33 μs/instruction

Test 4: Block with SETH and SETL instructions
6.17 μs/instruction

Test 5: Block with load/store instructions (LDAW, STAN, LDTW, STTW, LDPW, STTW)
8.81 μs/instruction

Test 6: Block with branch and jumps (BIFT, BIFF, JMPR, JMPA)
5.54 μs/instruction

Test 7: Two subroutine calls (JSVR+JRET, JSVA+JRET)
4.54 μs/instruction (or 9.08 μs/empty call)

Test 8: Lookup Nodal Link Table (LTJLT)
10.0 μs/instruction

Test 9: Lookup PM Local Directory (LULD)
9.00 μs/instruction

Figure 5.23 - Results from instruction timing tests

5.4 Summary

A prototype implementation of the Swarm architecture, coded in C language for execution under a Unix environment, was constructed to validate its operating principles and functional organisation. The prototype included all basic features of the architecture, excluding multitasking, service nodes and connection to the network. These elements were removed to facilitate a first implementation and evaluation of the system.

The prototype implements the parallel Swarm Abstract Machine (SAM), consisting of: one or more Processing Modules (PMs), each capable of holding Swarm data nodes; a Routing Module (RM), which routes agents between modules during program execution and manages the system configuration; an Access Point (AP), that holds a console node for interaction with the user for system configuration or I/O during program execution. The structure of the SAM is persistent, and the user can open and close the AP at will, without affecting the SAM. After the RM is installed in a host machine, an AP can be opened and the user can create (or remove)
PMs, and load configuration files that will create a number of data nodes in each PM. Before execution, temporary structures are created in the PMs to store the program and support the executing task. Transparently for the user, the Swarm system provides automatic termination detection of a task, informing the user (via the AP) if the execution normally terminated or was aborted. When the task terminates, all temporary structures are removed and the system is ready for re-configuration or new task execution.

An example application was developed and used to evaluate the operation of the prototype. The application consisted in a program for reservation of a path, of specific bandwidth value, between two given nodes in a communication network of arbitrary topology. The program was executed for a network of 10 nodes using two different mappings of nodes to PMs. The main functions of the Swarm prototype were exercised and traced, validating their correct operation, and timing and profiling information was obtained. Performance tests were executed to determine the execution costs of the main Swarm functions, showing a latency of 1.5 ms/agent (practically independent of the size of the workspace carried by the agent), a node creation time of 500 µs/node, and an average instruction execution time of 7.87 µs.
6. Discussion

This chapter evaluates the Swarm prototype and architecture, based on the results obtained. The main findings are discussed and different aspects of the functional organisation and the implementation are analysed and criticised. Difficulties and limitations of the experiment are highlighted and unresolved questions are discussed. Additionally, comparisons are made between the approach used in Swarm and the ones adopted in similar works.

6.1 Evaluation of the Architecture

The goal of this research was to study the use of agent-based approaching for the development of a distributed parallel processing (DPP) programming environment. A secondary goal was to experiment with agent-based processing. For these purposes, a parallel architecture called Swarm was designed and evaluated using a corresponding prototype implementation. In contrast to most of the existing DPP environments [Carriero and Gelernter 1989, Bal et al. 1992, Jul et al. 1988, Geist et al. 1994, Sapaty 1992, White 1994], Swarm does not target the end-user and, thus, does not offer a high-level interface. Instead, the idea behind Swarm is to provide a base layer that can be used for further development of user-level languages or interfaces for DPP, or even serve as basis for the design of massively parallel machines. The focus is on simplicity, to provide a reduced set of primitive operations and objects that can be efficiently implemented.

The main objective in constructing a prototype system was to simulate and evaluate the operating principles of the Swarm architecture, and provide insights for a subsequent, full implementation. Absolute performance was not a prime concern here, although care was taken to provide a reasonably efficient operation. This general objective was achieved, as the prototype worked and allowed experimentation with agent-based programming and observation of the operational characteristics of the architecture. Although the prototype did not have a network connection, all basic functional elements of the design were present: processing elements (PMs), communication service (RM), and access for the user (AP). This permitted a comprehensive evaluation, concerning both the internal operation and the user interface.
The prototype had all its individual functions exercised and example programs were executed under different conditions of load in the host machine and for different distributions of nodes per PM and number of PMs. Due to limitations in time, a more extensive testing could not be performed. However, the executed tests were sufficient to demonstrate that the prototype operates according to the specifications. Task initialisation, instruction decoding and execution, and the termination detection mechanism worked without problems. The agent routing service provided by the RM (routing module) was traced and revealed no problems. All functions provided by the AP interface were functional, including task abortion by the user and I/O during task execution. Persistent data structures could be created in the abstract machine and modified accordingly by the executed programs.

The work on Swarm is exploratory research, and one main difficulty in its development was the lack of technical literature about agent-based processing. Apart from the documentation available about the Wave system and artificial intelligence works based on the concept of agents, only very recently information about another agent-based DPP package (General Magic’s Telescript [White 1994]) was released. None of them, though, provide enough information about implementation or evaluation, or specifically target LAN-based processing. Most of the guidelines for the design of Swarm had to be derived from the existing literature about DPP in LANs, particularly from published information about implementation of message-passing, shared-memory and object-based environments.

The sections below discuss the major aspects of the Swarm architecture, comparing it with representative examples of the main existing environments for DPP. Aspects regarding the limitations, problems and possible improvements on the prototype implementation are discussed in detail in Section 6.2.

6.1.1 Swarm Agent-Based Processing

The two most distinctive aspects of the Swarm architecture are the use of agents and the way its shared data space is structured. This section focuses on the agent-based approach used in Swarm, leaving the discussion about the Swarm data space to Section 6.1.2. Swarm agent-based processing contrasts with the process schemes used in packages as PVM, Linda, or Orca. Although these permit multiple processes to run concurrently and dynamically create new processes, each process is fixed (as opposed to mobile) and usually heavyweight.
Emphasis is then placed on how data is moved between these fixed processes. Swarm essentially reverses this perspective, fixing the data and moving the processes instead. Processes are lightweight, and mobility and dynamic creation of processes are essential characteristics of the execution model.

The Swarm approach was inherited from the one used in the Wave system and improved for easier implementation. Swarm implements agents in a way that requires minimum storage and results in small overhead for communication and for processing. The agent only carries a small header (four 32-bit words, or 16 bytes), and all remaining information is just data. Although an agent can carry up to 256 (32-bit) words, only the used part of the agent workspace is transmitted, saving communication costs. In addition, agents provide a more powerful mechanism than common message-passing, as not only can they carry data but process state as well. This feature has already been recognised in other works like Active Messages, which is based in similar principles but focuses only at the communication primitives and does not provide any abstraction for data organisation or distributed coordination of processes.

Given the common origin, the most similar approach to Swarm is the Wave system. Both implement a graph-based data space and use agent-based processing. Nevertheless, Wave adopts a more abstract model and provides a wider (and sometimes redundant) range of primitives for coordination and control of agents. Swarm places efficient execution as its design goal, and offers a limited set of basic primitives. Another important difference between Swarm and Wave is that the former is targeted to LANs while the latter, although capable of working in LANs, is structured for processing in WANs and the Internet (possible extensions of the Swarm architecture for WAN-based processing are discussed in the Section 6.1.4). This is also the main difference between Swarm and the Telescript package, developed for processing in the Internet. Additionally, Telescript adopts a higher level of abstraction, as its environment is specialised for support of object-oriented programming, and seems to have no provision for agent spawning. A Telescript agent also appears to be a much heavier process than Wave or Swarm ones, requiring greater overhead to be moved and executed. However, the present communication latency of the Internet environment practically reduces the impact of this overhead on processing. All these characteristics make Telescript more adequate for loosely-coupled, distributed processing than DPP.
A similar remark can also be applied to the Emerald system, where both processes and data are objects. Emerald provides great flexibility, as invocations on objects are location-independent, objects can internally handle multiple simultaneous invocations (using monitors to control access to shared data) and any object can be mobile, independent if it is active (i.e. containing a process) or passive (containing data and related operations). The drawback is a difficult implementation, and Emerald relies heavily on compile-time optimisations to generate the best possible approach for each declared object. In addition, its single form of interprocess communication (synchronous remote procedure calls that are accepted implicitly) and the single mechanism for parameter passing (call-by-reference) makes the Emerald environment more suitable for distributed applications than for parallel processing [Bal 1992].

In contrast, Swarm adopts a simple approach for process creation, scheduling and coordination. Each agent is considered as a lightweight thread that executes non-preemptively when it reaches a destination node. Mutual exclusion is obtained by having only one agent executing per time at a particular node (note that this does not prevent an implementation where many agents execute in parallel at different nodes inside the same PM). Conditional synchronisation is provided by events, with agent suspension and re-scheduling controlled by the executing program (however, due to the slow speed inherent to I/O operations, service nodes automatically manage agent suspension and re-scheduling, which can be seen as a limited form of preemption). Finally, in a single atomic operation, a Swarm agent can leave the current node and spawn a number of copies of itself to different destination nodes in the environment.

These design decisions reflect a belief that optimisations or more elaborated mechanisms should be carried at a higher level, possibly by the high-level language compiler (or directly by a low-level programmer). For example, the persistent structures in the nspace, i.e. the nodes and their persistent workspaces and link tables, provide a structured shared memory where constructs like locks and semaphores can be implemented. Mimicking the RISC approach successfully used in present microprocessors, Swarm's design philosophy is that the role of the base layer should be simply to provide a reduced set of efficient primitives.
6.1.2 Support for Data Structures

The Swarm architecture adopts an abstract data space that is a simplification of the one used in the Wave system (see Section 4.1 and Appendix A). The basic building block is a node, a persistent object that can store data and references (spatial pointers) to other nodes. The abstract space where these nodes are created, manipulated, and destroyed is known as the nspace. This scheme provides a minimum set of primitive elements and is general enough to construct any data structure distributed in the Swarm nspace.

Intentionally, Swarm exposes some of the underlying structure that supports the nspace abstraction. This results from a compromise between the level of efficiency provided by the implementation and the abstraction provided by the architecture. The low-level programmer (or the high-level language compiler) must be aware of the distribution of nodes among the PMs, and must face the restriction that node creation, deletion or search only work inside the PM partition of the nspace. The format adopted for node addresses also reflects this compromise, being in between a physical pointer and a logical pointer. A purely physical pointer would permit efficient access, sacrificing flexibility. A logical pointer would provide location-independent access to nodes, but would require a more complex implementation. In Swarm, only accesses to PMs are location-independent. In this way, the architecture helps to expose the costs of doing expensive operations (e.g. search all nspace, broadcast to all nodes, or move a node to another PM), while still providing a good level of abstraction for the developer of end-user interfaces. Additionally, a form of load balance is possible by moving PMs from a heavy loaded host to another light loaded one. It should be noticed, though, that this operation can be expensive considering the amount of state a PM may be currently holding.

As the abstraction provided by Swarm offers a type of structured data space, with nodes, links and PMs, it is much closer to shared-data structures than to shared-pages or shared-variables approaches. Similarities can be found between Swarm and packages like Orca, Emerald and Linda. However, Swarm implements a lower-level environment of primitive operations and building blocks than the ones offered by Orca, Emerald, and other object-based and object-oriented approaches, which interface directly to the final user via a high-level language and require any data to be encapsulated in an object. Additionally, Swarm nodes are fixed to PMs, and do not require the overhead of copying (as in Orca) or moving (as in Emerald). Differently from Emerald, only the agent (process) can move, which simplifies implementation.
Swarm also contrasts with Linda, which has a small set of primitives and a single structure (a tuple) but implements a higher level of abstraction. While Linda constructs data structures using symbolic links (the tuple names), Swarm links nodes using their addresses (a direct reference). Another difference is that a Linda tuple is a linear sequence of fields and the number of fields is usually small. Instead, Swarm nodes have internal memory and, if necessary, they can easily accommodate internal structures (using internal pointers). The nodal memory is divided into two partitions, volatile and persistent, that can grow according to the needs of the program. The number of primitive operations is bigger than in Linda, but they are simple enough to permit efficient implementation.

6.1.3 Support for End-User Interfaces

As previously explained, Swarm was developed as a base layer for implementation of higher layers like languages or interfaces for the final user. Regarding the instruction set, Swarm provides all basic instructions found in an average RISC processor. Additionally, it also includes instructions that access specific architectural features and permit agent coordination and control, but it does it in an away that does not make the instruction set complex or compromise efficiency. Missing features (detailed in Section 6.2) are the low-level support for management of exceptions and the reduced level of facilities for debugging. Other features not considered in this version of the architecture are floating-point instructions (necessary for most numeric applications) and support for 64-bit data. However, all these can be added to the architecture without difficulty.

While the RISC-style interface offers a familiar environment for language design, the resemblance between semantic networks and the Swarm persistent nspace favour the implementation of database-like paradigms, as logic languages or rule-based system used in the artificial intelligence area. Nevertheless, other generic paradigms can also be implemented. One possibility is a C-like language, similar to the pseudo-C used to describe Swarm programs in the Chapter 5. Other alternatives include functional and object-based (or object-oriented) languages, or even graphical interfaces or the end-user. For example, the Swarm mechanism of suspending agents could be used for selective execution of functions after update of specific variables, facilitating functional language implementations. Yet another approach would be to derive a set of primitives to be added to an existing programming language, as adopted in the
Linda system. Further study is necessary, though, to identify the restrictions that could be necessary to impose on the language or the agent-based model.

An important issue regarding Swarm is whether executable code should be interpreted or dynamically translated to the native code of the host machines. From the environments so far discussed, all LAN-based packages (i.e. Linda, Orca and Emerald) adopt compilation to native code, while all WAN- and Internet-based packages (Wave and Telescript, and the Sun’s Java package discussed in Chapter 2) use interpretation. Although interpretation is a slower approach, it suits heterogeneous environments (different machines with different native codes) and improves security (interpreted code can be checked before execution), two important points for processing in open systems like the Internet. In turn, LAN-based packages tend to emphasise performance and make different compromises in their implementations. Linda consists of a small set of primitive operations added to an existing programming language (C or Fortran). A program first passes through a pre-processor, that translates Linda-specific operations into statements of the used language, and then is compiled down to native code. During execution, the Linda runtime system manages accesses to the tuplespace. Although Linda provides a way of dynamically creating processes (using the eval construct explained in Section 3.3), it is the compiler that specifies where they will run. Thus, both normal and eval-created processes have their host defined at compile-time, with the corresponding native code being generated. Orca also has fixed processes, but they can be dynamically created and allocated (at creation) to any processor in the system. Furthermore, Orca’s objects have internal code (their operations or methods), which must run on different processors as the implementation relies on copying of objects to make them locally available to the requesting process. Orca, then, has been so far only implemented on homogeneous systems [Bal et al. 1992], for which generation of relocatable code (for the same type of processor) satisfies the requirements. Emerald’s intrinsic object mobility brings code generation problems similar to the ones found in Orca. Although not explicitly stated in the published literature [Raj et al. 1991, Jul et al. 1988, Black et al. 1987], current Emerald implementations also seem to require a homogeneous system.

Compilation to native code offers the advantage of efficient execution, as code will be directly executed by hardware and a series of optimisations are possible at compile-time. This strategy works very well in Linda implementations and can be considered responsible for the good performance offered by the package [Mattson 1994], in spite of its associative
addressing style. For simplicity of implementation, the Swarm prototype used interpretation to execute its abstract machine instructions. However, the closeness between the Swarm instruction set and the instruction set of RISC processors present in most workstations, open way to an on-the-fly code translation scheme. This technique has been reported to be successful and efficient in the TAOS [Pountain 1994] operating system, and consists of dynamically translating a virtual machine code into the native processor code during program loading to memory. For Swarm, this would increase the overhead during program loading into the different PMs, but the expectation is for a gain of at least one order of magnitude in execution time.

6.1.4 Additional Extensions

The previous sections already described some extensions to the Swarm architecture. Other possible extensions that deserve discussion are WAN-based and heterogeneous processing, fault tolerance, and privacy/security.

In contrast to Swarm, the other two agent-based packages currently available were developed for use in WAN- and Internet-based environments. The low level of coupling and high communication delays characteristic of these environments reflect in their design decisions. Both Wave and Telescript adopt interpretation as the translation mechanism and have the program source code carried by the agents. A similar approach could be used in the Swarm architecture, but there are intermediate solutions.

One possibility is to use a combination of interpretation and compilation. A internetwork is basically a connection between LAN-based domains. The idea is to explore the higher degree of coupling inside these domains. Processing would start in a specific domain (a LAN) in the way it happens in the present version of Swarm. Only if one or more agents need to access remote domains is that task-related information would be transmitted. This information would comprise the executing program plus some extra data about the organisation of the origin domain (e.g. number of PMs and address of the AP/console node). Once the information reaches the destination domain, the environment for that task is initialised there in the same way it is done presently in Swarm. Obviously, additional issues would have to be taken in account, like how to manage the address space and how to guarantee a reasonable level of security. Further refinements could be added to this scheme, like use of intermediate languages,
which would provide compact code for the most common blocks of operations and reduce communication traffic.

WAN-based processing also requires support for heterogeneous systems. Interpretation and on-the-fly translation, discussed in Section 6.1.3, are two techniques that can be used to permit independence in terms of executing code. The remaining issues are related to the representation of data, such as integer format, floating-point standards, character sets, word size and endianess, that affect any implementation targeting heterogeneous environments. The choices adopt in Swarm try to reflect the most common standards in LAN-based workstations [Stevens 1990]. Therefore, the current version of the architecture uses 32-bit words, arranged in big-endian byte order, representing integers in 2’s-complement format and characters in ASCII code. No built-in support for floating-point is available at the moment. None of these choices, however, interfere with the organisation of the architecture and, if necessary, they can be changed to reflect new technological developments (e.g. 64-bit words and universal character sets).

Concerning fault tolerance, Swarm only supports saving of copies of each PM nodal store into files, before execution of the program. If the task aborts prematurely, e.g. due to a runtime error, the user can simply reload these files into the respective PMs, obtaining the same configuration as before the execution, and try to run the program again. One possible improvement is to have automatic incremental saves of each PM nodal store, so in case of crash the user can roll-back and obtain intermediate states. Another option is to have these saves commanded by special instructions, inserted in the code at specific checkpoints defined by the user. These would be implemented as optional features, as any output to files via Unix system calls would slow down the Swarm task execution. If fault tolerance is essential, more elaborate schemes (e.g. the transaction model used in the Argus system [Liskov and Scheifler 1983]) can be implemented at application level. In this case, all intermediate data could be kept in the transient workspaces of the nodes and only committed to the final persistent positions at end of the processing.

A final point is security. At its present, experimental stage, the Swarm architecture has no provision for security, and privacy is guaranteed only because the system runs in a single user partition under Unix. This is, though, a common characteristic to all LAN-based packages discussed above (Linda, Orca and Emerald). As previously mentioned, security becomes a major issue when extending processing to WANs and the Internet, even if performance needs to
be sacrificed [Harrison et al. 1995]. However, none of the schemes presently used in distributed systems [Woo and Lam 1992] seems to suit agent-based processing environments, and new mechanisms have to be developed. The security features implemented in the Telescript package [General Magic 1995b] offer a good example of what kind of mechanisms are necessary for agent-based processing in the Internet.

6.2 Evaluation of the Prototype

As previously mentioned, limitations in time did not permit more extensive testing of the prototype. Nevertheless, preliminary results could be obtained, permitting an initial evaluation of the implementation. Experimentation with the prototype also exposed some weaknesses and points for improvement. Some of them are the result of limitations and peculiarities of the systems calls provided by the Unix environment. Others are the result of design decisions taken during the prototype development. The sections below discuss these aspects.

6.2.1 Performance

Based on the timing figures presented in Section 5.3, some operational characteristics of the prototype implementation can be produced. The average instruction execution time of 7.87 $\mu$s is two orders of magnitude greater than the best execution time of a machine instruction by the physical processor (25 ns for the 40 MHz Sparc processor). This shows the impact of interpretation of each virtual machine instruction and the gain that could be obtained by converting a source program into native code for execution. The interpretation mechanism, however, seems to have balanced the execution times of different kinds of instructions, as both the fastest (5.54 $\mu$s for branches/jumps) to the slowest (10.0 $\mu$s for LULT) instructions are equally distant from the average 7.87 $\mu$s/instruction. Other instructions revealed execution times very close to each other (8.88 $\mu$s for arithmetic/logical, 8.81 $\mu$s for load/store, 9.08 $\mu$s for an empty subroutine call, and 9.00 $\mu$s for a lookup into the PM Local Directory). Differences between the two versions of arithmetic/logical instructions (i.e. with immediate and variable operands) also seem to be minimal, as shown by a difference of 2% between the execution times for ADDI and ADDA.
The cost of transporting an agent between two nodes revealed to be practically independent from the size of the agent workspace. However, the figure of 1.5 ms/agent is still around 60% more expensive than the one shown by a null RPC. One reason could be the overhead introduced by the interpretation and scheduling mechanisms used in Swarm. Although the prototype makes use of pipes (a simpler and more efficient communication form than the UDP/IP layers used in the RPC test), the internal work that has to be performed to load an agent from the input pipe, to fetch and execute the INDL, and to fetch and execute the SPWN can cancel this advantage, increasing the execution time. Direct translation to native code could improve these operations. A more immediate solution is the simplification of the internal routines that pack and unpack the agents, as the results shown that the agent payload does not influence the communication cost (at least for the communication scheme used in the prototype).

Comparing the obtained results of average execution time for a Swarm instruction and agent transfer time, a communication/computation ratio of 190 is obtained (1.5 ms/7.87 µs). Looking at the assembler code for the bandwidth reservation programs (see Appendix D), the longest program segments between two spawnings have approximately 20 instructions. This implies that program execution time in the prototype would be dominated by communication unless a larger program segment (e.g. a 10 times repetition of a 20-instruction loop) is frequently executed.

The operation of node creation revealed to be two orders of magnitude slower than the average execution time for instructions (500 µs against 7.87 µs). The significant factor in this case is the structure of the PM Nodal Store (shown in Appendix C), which requires three calls to the Unix system call `malloc`, in order to create a new node. Possible optimisations would comprise a simpler structure for the Nodal Store, which could reduce the necessary number of calls to `malloc`, and a customised memory allocation routine that would replace `malloc` and provide a more efficient service.

Although the obtained results provide a general picture of the operation of the Swarm prototype, more testing is still necessary to better characterise the problems and the influence of different factors. For example, to determine how efficient is the lookup Local Directory routine when the number of nodes and stored keys rapidly increases in number, or to obtain the communication costs for multiple agents produced in a single spawning.
6.2.2 Exception Detection and Handling

A limited form of exception handling is present in the prototype. When the agent arrives at a PM, it can be directed to the PM root node if its original destination node does not exist. In this case, the agent can detect the situation, by examining the agent control bits, and take necessary action (which must be coded by the user). For other exceptions, the system just tries to continue execution, adopting a default action according to the error (like discarding values written to illegal positions, returning EMPTY for values read from non-existent places, or ignoring illegal instructions).

The specification of the Swarm architecture (see Chapter 4) provides special bits in the agent status word (ASW) that permit some level of exception control and detection. Besides, the termination detection mechanism could include an error code that identifies the cause of termination, which could then be forwarded to the user via the AP. The IP value corresponding to the offending instruction, if this is the case, could also be forwarded, to help debugging. Additionally, the system could return to the executing program an identification of the exception or permit an automatic jump to a user defined handling routine.

Another important point is related to communication buffer space. The asynchronous nature of the Swarm system leads to a nondeterministic program execution, which makes it difficult to predict the maximum buffer sizes required for each PM input and output queues. The prototype tries to reduce the use of buffer space by having packed agents in the output queue. However, this does not eliminate the problem and the most critical situation is a deadlock caused by a PM that has both the input and output queues completely full. The PM cannot continue, as it cannot output the packed agent stored in its execution buffer and release this buffer to input another agent. The RM is also blocked, as there is no space in the PM input queue to receive another incoming agent destined to that PM. An extra execution buffer in the PM, to hold the incoming agent, or an additional buffer in the RM to keep the agent until it can be accepted by its PM, would represent extra complexity in the respective algorithms and not provide a definitive solution, only postponing the problem.

The solution devised is not to avoid, but to detect the deadlock situation and abort the running task. The approach is to consider lack of buffer space as a situation of resource exhaustion, similar to a stack overflow during program execution in a sequential machine. The idea is to abort the offending task, informing the user of the reason. It is then the user
responsibility to take an appropriate action to avoid the deadlock. One solution is to restructure the program in a way that reduces the number of agents generated or holds some agents while other are executing (throttling). A second solution is to increase the number of PMs and re-distribute the existing nodes among them, avoiding a distribution which locates many frequently accessed nodes (hot nodes) together in the same PM. As each PM has its own set of input and output buffers, this solution is equivalent to increasing the total buffer space in the system. A third solution is to increase the current size of the buffer in each PM. This solution is not possible in the current implementation, as there is no Unix command that permits a change in the buffer size of a pipe. For the default buffer size of a pipe in the order of 4 kbytes and an agent carrying 12 workspace positions (a total size, including the head, of 64 bytes), the maximum number of agents a pipe can hold during program execution is 64, which is too small for any purpose except testing. However, the pipe pair can be substituted by a Unix-family socket pair, that permits buffer re-configuration, and a corresponding command can be added to the AP-RM interface. This approach is to be adopted in a following networked version of the prototype.

6.2.3 User Interface and Debugging

The Swarm prototype only presents a minimum level of user interface, mainly because much of the development effort was concentrated in making the prototype operational. A graphical interface, using X-Windows, and an extended set of commands, providing more options for the user (e.g. filling the contents of all existing PMs in a single file), would facilitate the use of the system. Another essential feature still to be implemented is an assembler, as up to the moment all translation of programs to the corresponding machine code has been done by hand.

Furthermore, extra work is still necessary to provide a good support for debugging. The information provided by the RM and PM log files was revealed to be insufficient for debugging programs. It was necessary to add (to RM and PMs) extra debugging messages, sent to buglog files, as an option activated by conditional compilation of the Swarm modules. These messages produced a trace of the internal operation of each module, facilitating the identification of the problem. The drawback, however, is that task execution takes much longer, as a result of frequent I/O operations to write information into the log files. Therefore, the amount of
debugging information produced by the system should be an option selected by the user, to be used during experimental executions of the program and deactivated for normal running. Additional information can also be provided by a kind of core dump of the Swarm machine. As an option set before execution, the internal machine state of all PMs and AP would be saved in files in case of a crash, and could be examined later.

6.2.4 Storage Management

The present prototype dynamically allocates space for persistent workspaces (including link table), for transient structures (suspending buffer, transient workspaces) and for the RM address book. What is not implemented, however, is a routine to detect when these structures can be reduced in size, to save memory space in the host machine. This is particularly important if the executing program created and then deleted large amounts of data or nodes. Another area for improvement is a better algorithm for dynamic memory allocation. The prototype directly uses the malloc routine provided with the GNU gcc compiler. A tailored, more efficient routine could be constructed, adapted to the specific requirements of the Swarm environment and saving processing time and memory space.

6.2.5 Features Not Implemented

To simplify its implementation, the current version of the prototype did not include three features of the Swarm architecture: service nodes; network connection; and multitasking. In term of service nodes, the prototype provides a minimum: the console node. Using features already implemented, service nodes that access files or other applications in the host could be created. The PM would have to manage automatically the suspension of agents waiting for some I/O and their later awakening. An alternative would be to fork a child process from the PM to deal with the request. This would imply an extra overhead, but this might not be noticeable as I/O access in Unix is already a slow operation. A multithreaded environment, such as the one provided in the Sun Solaris environment, would improve efficiency in this case. Additionally, internal module functions could be programmed to execute concurrently. For example, while an agent is executing in the agent buffer, another agent could be loaded in a spare buffer for input and another could be under output from a third buffer.
Concerning network connection, a networked version of Swarm would require many RMs, one per host machine. Additional protocol would need to be added to keep each RM address book consistent when PMs are created or deleted, or when new RMs are installed in other machines (or even removed from them). Note that this will happen in command mode, not during execution, when the machine configuration should be stable. The structure of the address book itself would need to be expanded, to add an extra field for holding the network address (48 bits, comprising IP address + port number) of the RM that manages remote PMs. All that follows would then work in the same way of a single-hosted version: agents destined to local PMs would be delivered as before; agents destined to remote PMs would be delivered to the remote RM, which in turn would deliver them to the local PMs; agents received would be normally delivered; broadcasts and termination detection would not be affected. From the point of view of a PM, all RMs would be seen as a single, unified routing service. Also, just by updating the RMs address books, whole PMs could be moved from one host to another, providing a form of load balancing. The AP user interface would also have to be extended, giving the option of creating or deleting local or remote PMs, with specification of the host. The AP list configuration command would have to show the respective RM for each PM listed. Other AP commands would remain unchanged. As a result of these additions, necessary to permit network connections, a decrease in the performance of the system is expected. In particular, the communication overhead imposed by protocols like TCP/IP and the reduction in the available bandwidth due to shared use of the physical network (Ethernet or FDDI), may lead to a reduction in the communication throughput obtained in the single-machine version.

Still in the context of a networked version, improvements can also be made in the termination detection mechanism, which implements a simplified version of the algorithm developed by Mattern [1990]. The implementation explores the fact that the RM knows all PMs, and requires that the number of PMs must be sent to all modules before execution begins. The AP can then know, before starting a termination detection cycle, how many echoes it has to expect before deciding about the termination state of the system. Although this is scalable (the AP does not need to know each PM, only how many of them are there), a better solution would be to use the original strategy proposed by Mattern and arrange the AP and PMs in a virtual network for the purpose of termination detection. In this way, each PM would know a restricted number of neighbour PMs to send termination enquiries. On receiving the enquiry, a PM would pass it to its neighbours. Enquiries reaching a PM already visited would be returned to its sender as an echo. A PM that received echoes from all its enquiries would
return an echo to its predecessor. At the end of the process, the AP would have received the accumulated echoes from all PMs and could decide about the termination. In this approach, it is not necessary for the RM to provide the number of modules to each PM and to broadcast termination enquiry messages.

A final issue is multitasking in the Swarm environment. Multitasking support in Swarm follows the general guideline of providing the minimum features and lets the rest to be dealt with by higher layers. However, a comprehensive study of this issue could not be carried out due limitations in time. Some brief remarks however, can be made. A Swarm system could identify different tasks using a task ID attached to the task when it starts execution (the present prototype already reserves a field for this purpose on each agent). Each host (actually RM) would have a range of task IDs reserved for it, which eliminates the need for a central manager (as task ID is 32-bit, this may provide a name space big enough for this partitioning, which can be done similarly with the PM ID + node ID already used in addresses). Also, all transient structures for task execution are created separately for each task, and there is no possible conflict here. The difficult point is how to coordinate tasks that try to access and modify persistent structures (more specifically, nodes and their persistent workspaces and link tables). The Swarm architecture provides low-level locks for this purpose, along with the possibility of detecting a locked node and handling the exception. However, a definite evaluation of this mechanism can only be achieved when these features are implemented and exercised under different conditions.
7. Conclusions

This chapter presents a brief review of the work done, emphasising the main points of the research, discusses possible applications, and describes suggestions for future work.

7.1 Review of the Research

A number of different solutions for parallel processing in computer networks have recently emerged. Some solutions adopt a message-passing style that maps well in the distributed environment, but makes programming more difficult. Others combine the distinct memory spaces of the machines in the network into a single shared data space, as an abstraction that facilitates programming but often requires a complex implementation. Differences also exist regarding the nature of the physical environment. While some solutions are developed for use in LANs and explore the high communication rates available in these environments, others target larger loosely-coupled domains, such as WANs and the Internet. The Swarm architecture evolved from previous work on agent-based processing and is aimed at efficient parallel execution of decentralised algorithms in a physically distributed environment. It combines advantages of message-passing and shared-memory schemes, providing a flexible approach that can be applied to LANs, WANs and the Internet.

The Swarm architecture delivers a parallel abstract machine that includes processing elements and a communication service. Swarm supports distributed data structures built from nodes that persistently store data and spatial pointers to other nodes. Processing is carried out by agents, which are mobile processes executing like threads within a higher level task that allocates resources. The architecture offers flexibility for the implementation of high level layers, like languages and end-user interfaces, combined with a reasonable level of abstraction as provided by shared-data structures packages.

A Swarm prototype was coded in the C language, running under Unix on a single-workstation, for evaluation of its functional organisation, instruction set and operational characteristics. Programs were executed for different configurations and the internal operation traced. Even though the prototype was restricted, the results showed that the architecture is
feasible. In addition, a number of aspects that permit further optimisations could be identified for improvements in future versions. The prototype also helped to validate the underlying agent-based paradigm used and proved its effectiveness.

Although absolute performance was not a main concern during the development of the prototype, preliminary results showed an average latency of 1.5 ms for propagation of an agent between two nodes in different processing modules (but in the same host), an average instruction execution time of 7.87 µs, and a node creation time of 500 µs. Tests also have shown that, for the prototype implementation, the propagation time for an agent is practically independent from the agent payload (i.e. the amount of data carried in the agent workspace). Considering that the prototype uses interpretation and is a non-optimised implementation, these results compares favourably with conventional mechanisms like remote procedure calls (RPCs). However, extensive testing could not be performed, due to limitations in time, and more testing is still necessary to better characterise the problems found in the implementation and provide the necessary level of feedback into the design of the architecture and the model.

7.2 Applications of the Research

The present version of the Swarm architecture adopts a SPMD (Single Program Multiple Data) approach. Before execution starts, the program is broadcasted to all processing elements (PMs) and agents have only to carry a minimum state comprising identification, destination address, status word, instruction pointer and data. Instructions follow a RISC-style format that can permit efficient processing (using on-the-fly translation to native code), maps well to existing processors, and provides a flexible base-layer for language or interface implementations.

Swarm can be used to implement a parallel processing package for LAN-based workstations or a multicomputer machine, using a medium-level, C-like language with specific constructs for dealing with parallelism. Optionally, more abstract interfaces can be constructed, using Swarm as a support layer. Parallel logic languages or rule-based systems could benefit from the primitives offered by the architecture.

The processing model currently implemented in the Swarm architecture can naturally support irregular and distributed data structures, where a great number of small sequences of
actions should be executed simultaneously at different nodes. These characteristics match applications like relational and network databases, knowledge-based processing and artificial intelligence systems. Scientific and engineering applications that model irregular structures (graph-like) can also benefit from the processing environment offered by Swarm. The intrinsic mobility of agents can be used in distributed simulations or as a support layer in intelligent communication networks and mobile communication networks.

Finally, agent-based processing provides a promising form of processing for WANs and the Internet, already in use by some pioneering works. The Swarm architecture can be extended to cover these environments, providing a bridge for parallel processing in parallel machines, LAN networks, and larger domains of internetworked machines.

7.3 Suggestions for Future Work

The research described in this thesis only represents the initial stage of an exploratory investigation in agent-based processing. Much work is still to be done. The most immediate work regarding Swarm consists of implementing a networked (LAN) version of the prototype, including support for service nodes, and improving the exception handling mechanism. An important feature, missing in the present prototype, is communication buffer overflow detection and the provision of means to extend communication buffer space when this problem occurs.

Another point that needs further investigation is the strategy for translation of code. The preliminary results obtained from the prototype showed an average instruction execution time that is two orders of magnitude bigger than the execution time for a native (physical) machine instruction. On-the-fly translation from Swarm code to native machine code is one technique that could improve execution performance. Other necessary improvements comprise: better support for programming and debugging; better interface for the user; improved routines for the management of the internal storage; and better support for fault tolerance.

A final feature is multitasking, which would permit concurrent execution of many distinct tasks in the same Swarm environment. However, further study in the subject is necessary to properly define and evaluate the best strategies for multitasking implementation.
A more thorough and detailed evaluation of the architecture is also necessary, which should imply the implementation of different programs under different distributions of data in terms of nodes and PMs. An analysis of patterns of communication (i.e. agent traffic) produced by these programs would improve the understanding of the operations of the architecture and allow further improvements. Implementation of applications from different areas, complemented by a study of their requirements, would permit identification of other deficiencies or lacking features. A generic study of agent-based processing is also necessary to establish parameters for evaluation and highlight the fundamental issues related to this approach.

Finally, Swarm does not represent (and neither intended to be) the final solution for agent-based DPP. Different combinations using concepts from other paradigms can still be tested, adding alternatives like easier mobility of data nodes (including experimentation with logical addresses), co-existence of fixed and mobile processes, multiple process execution inside a node (using monitors like Emerald and Orca), and different scheduling schemes (e.g. preemptive ones).
8. Bibliography


Appendices
A. The Wave System

Wave is an architecture and a language developed for agent-based parallel processing in physically distributed systems. Much of the general organisation of the Swarm architecture, presented in this thesis, is based on concepts first introduced in Wave. Therefore, a description of the Wave system complements the background for the presented work and helps to understand some of the motivations and design decisions. This appendix is a short version of a report by Errico [1993], which presents an overview of Wave, comprising paradigm, architecture and language. For more details and examples, the reader should refer to [Sapaty and Borst 1994], which also includes a general description of the Wave system implementation.

A.1 Paradigm

The idea of using graphs to coordinate processing in loosely-coupled systems was introduced almost simultaneously by Francez [1980] and Dijkstra and Scholten [1980], as a technique for distributed termination detection. In a later paper, Chang [1982] showed how the same approach could be extended to a broader range of applications and gave it a name: Echo Algorithms.

Chang's Echo Algorithms is defined for a general graph where each node can be considered as an autonomous processor, with its own local storage, and each edge is a bidirectional communication link offering ordered (no message overtake) and reliable communication. Each node only knows its neighbourhood and communicates with its neighbours by message passing. Two types of message are used: explorers and echoes. Processing is performed by a parallel graph traversal divided into two phases. In the forward phase, the starting node (the initiator) sends explorers in parallel to all its neighbours. On receiving an explorer for the first time, a node will mark the corresponding edge and then send explorers in parallel through all the other edges (except the marked edge). The echo phase starts when a subsequent explorer arrives at a node already visited by its first explorer (i.e. a node that already has a marked edge). The explorer will then be turned into an echo and returned along the same edge it came. Each node will wait until it collects an echo for each explorer it sent and only then will the node propagate an echo through its marked edge. As a special case, a node with only one edge will always
transform incoming explorers into outgoing echoes. The initiator, which has no marked edge, will obtain the result of the processing when it receives all its echoes.

Echo Algorithms basically work by making a parallel traversal of the graph. No central controller, global clock or shared memory is used. The actual speed of communication for each edge is not fixed, being different for each physical environment used or even changing with time in the same environment (due to variations in load, for example). Thus, different executions of an echo algorithm may produce different traversals of the same graph. Each graph traversal builds a spanning tree in an execution time proportional to the diameter of the graph [Chang 1982]. Applications of Echo Algorithms in distributed processing can be found in [Andrews 1991] and [Tel 1994].

The Wave paradigm developed by Sapaty [1992] combines ideas from Echo Algorithms, semantic networks [Winston 1993], and pattern matching approach to logic problems [Bic 1984], among others. There are two key concepts in the Wave paradigm: the graph structure and the waves (agents). A graph is the main structure for both representation of information and processing control. Information is represented by the labels of nodes and links of the graph and by the actual interconnection of different nodes by links. Waves are moving processes (process = instructions + program counter + intermediary data), free to navigate through the graph, guided by a pattern matching mechanism that specifies the links to be passed and the nodes to be reached. Once a wave reaches its destination node, it may execute there, performing local operations (modifying data belonging to itself or to the node) and spawning new waves to other nodes. In this spawning a wave may replicate or split itself, providing each child with a copy of the parent data and a proper pattern to be matched in order to find a destination. Processing begins when a wave is injected at some point in the graph, starting a family of concurrent processes distributed throughout the graph. Waves will cooperate to find a solution and interact by sharing information in the graph itself and in the data stores located at each node.

As with Echo Algorithms, in Wave a graph is the main control structure with processing nodes capable of storage and edges that are bidirectional communication links. However, what is passed between nodes are not messages but full processes. Also, it is not mandatory for a node to send a copy of the incoming process to all its neighbours. The process itself carries information about how it is going to move in the graph, including possible replications or splittings. Therefore, in Wave the user has total freedom in defining the graph traversal.
A.2 Architecture and Language

The Wave architecture defines a distributed data network like the one shown in Figure A.1. This network space is arranged as a graph with persistent nodes interconnected by (oriented or not oriented) links. Both nodes and links are labeled. The node label, or name, is a numeric value or a character string, representing an arbitrary piece of information. The link label expresses a relation between two nodes (in numeric or symbolic form). A label must not be null, as the node or link that receives a null label is automatically deleted from the network. Nodes and links are abstractions that do not correspond directly to the underlying physical system. Depending on the implementation, many nodes can be grouped in the same physical computer and many links can share the same physical communication channel or be implemented simply as structures in main memory.
Nodes communicate using **surface** and **tunnel** links. Surface links correspond to the links between nodes discussed above. Tunnel links are temporary connections that permit communication between two non-adjacent nodes. A third type of link, **loop** links, is an abstraction used by program segments that loop back to the same node at the end of each step. They are represented by unlabelled links starting and finishing at the same node.

At runtime, nodes can also hold temporary (volatile) data generated by the program execution. Each normal node contains a Wave Language Interpreter (WI), that interprets the instructions carried by the **waves** (agents) that arrive and execute at that node. The execution of Wave programs requires two special nodes in the network space: an **entry node** and a **terminal node**. The entry node has the reserved address zero, and is capable of processing instruction and storing volatile data (but no persistent data). The first instruction of a Wave program is always executed on the entry node, that can access, or be accessed by, any node via tunnel links. The entry node can not be deleted and is always pre-existent to the rest of the network, being the start point to creation of new network spaces. The terminal node has no content, address or wave processing capabilities. It is just an input/output point that is accessed via a special variable in each normal node, to deliver information from or to the user. More complex forms of I/O can be performed by directly accessing the underlying operating system via special **external calls**.

As previously explained, the Wave agent, called a wave, is a complete process that carries its code, program counter, and temporary data. Waves carry the program source code, written in Wave language, which is interpreted at each node visited by them. The Wave language adopts a string-based format, where constants, variables, operators and delimiters are all represented by short sequences of characters. Figure A.2 summarises the main characteristics of the Wave language, as described in [Sapaty 1992] and [Sapaty and Borst 1994]. There are three classes of variables. **Nodal variables** are volatile local variables fixed to a node, shared by all waves accessing that node. **Frontal variables** are the ones private to a specific wave and are carried by it. **Environment variables** are special information accessible by the wave that arrives at a node, comprising: address of the current node; content (persistent value) of the current node; address of the predecessor node (i.e. the node where the wave came from); direction (sign) and label of the passed link; and terminal variable (which is written to output data to user, and read to input data from user).
wave → {move,}
move → unit [operator unit] [rule](wave)
unit → {string;} | variable
string → 'char' '[*]' le{ld}l | [+|-] dig{ld} | + | - | @
variable → N{ld} | F{ld} | C | A | P | S | L | T
operators → ~ | [-|-] <= | <= | + | - | * | / | & | A | |% | :: | # | !
rule → SQ | AS | AP | OS | OP | RP | WT | ID | CR

Conventions:
- definition of syntactic categories
| = alternative delimiter
( = optional constructs
{} = constructs that may be repeated zero or more times
(= separators used, if defined, when there are more than one construct)
char = any character (including non-alphanumeric ones)
lcl = lower case letter
ld = letter or digit
dig = digit

Rule Constructs:

<table>
<thead>
<tr>
<th>SQ</th>
<th>AS</th>
<th>AP</th>
<th>OS</th>
<th>OP</th>
<th>RP</th>
<th>WT</th>
<th>ID</th>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>And Sequential</td>
<td>And Parallel</td>
<td>Or Sequential</td>
<td>Or Parallel</td>
<td>Repeat</td>
<td>Wait</td>
<td>Indivisible</td>
<td>Create</td>
</tr>
</tbody>
</table>

Operators:

| ~ | / | <= | < | <= | = | < | + | * | / | & | A | % | :: | # | ! |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| belongs | not belongs | less than | less or equal | equal to | assign | add | subtract | multiply | divide | append | convert to set | merge vector into string | split string into vector | find/replace by index | find/replace by content | hop | halt |

Spatial Variables:

<table>
<thead>
<tr>
<th>N</th>
<th>F</th>
<th>C</th>
<th>A</th>
<th>P</th>
<th>S</th>
<th>L</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodal Variables</td>
<td>Frontal Variables</td>
<td>Content of Node</td>
<td>Address of Node</td>
<td>Predecessor Address</td>
<td>Sign of Link</td>
<td>Content of Link</td>
<td>Terminal</td>
</tr>
</tbody>
</table>

Special Symbol:

<table>
<thead>
<tr>
<th>@</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Broadcasting</td>
</tr>
</tbody>
</table>

Halt Codes:

<table>
<thead>
<tr>
<th>$0</th>
<th>$1</th>
<th>$2</th>
<th>$3</th>
<th>$4</th>
</tr>
</thead>
<tbody>
<tr>
<td>general failure</td>
<td>success</td>
<td>halt w/ track saved</td>
<td>halt w/ track deleted</td>
<td>local failure</td>
</tr>
</tbody>
</table>

Figure A.2 - Summary of the Wave Language

The Wave language accepts three scalar types: decimal integer, string and address (node address). Additionally, a form of heterogeneous list is supported, with corresponding operators to insert, delete and search elements. While environment variables can only hold scalars, nodal and frontal variables are considered lists, with the scalar case being considered as a one-element list. The language is weakly typed.

Much of the Wave syntax is dedicated to control the way waves replicate and interact during program execution. The basic operator for propagation is the hop operator, which commands the wave to move to other nodes, specifying the links to be passed and the nodes to
be reached. Many variations are possible, from propagation via a specific surface link to a particular node, to replication to a group of nodes that have some specified name, via tunnel links. Additional effects, like splitting the original wave into different child waves, are accomplished by the combination of delimiters and control operators. As a program in conventional languages is composed by command lines, a Wave program is composed by a sequence of constructs called moves, with dots as separators:

\[ \text{move1.move2.move3} \ldots \text{.moveN} \]

Each move consists of an action, which may be a local operation or a jump (hop) through the network space. The first move in the sequence is known as the head and the following moves are called the tail of wave. The head executes in the current node, being deleted from the wave after that, and defines the propagation of the tail. If the head contains a propagation command to other nodes in the network space, then the next move in the sequence becomes the new head and the wave continues execution on these nodes. Otherwise, if the head contains no propagation command, the next move also becomes the new head, but execution occurs on the same node (a loop). Subsequent moves will continue to execute on the same node, sequentially, until a propagation command is found. If it is necessary to have many moves executing concurrently in the same node, these moves must be separated by commas, instead of dots.

A move may be empty, simply causing the wave to loop on the same node, without performing any action. Another possibility is to have a move composed by an embedded wave, between brackets. In this case, different dependencies can be created when the embedded wave is preceded by special control operators, called rules. Wave provides rules for: sequential, conditional and repeated (loop) execution of code; AND- and OR-parallel execution of waves; and barrier of synchronisation (wait). A special rule permits the wave, during a propagation, to create a node or link, if they do not already exist in the network. Conditional execution is loosely modeled after the SNOBOL language [Griswold et al. 1971], providing three options: the next move only executes if the current move succeeds; the next move only executes if the current move fails; the next move executes regardless of whether the current move succeeds or fails. Operations that can return success or failure are comparisons and propagations (hops).

Control of waves scattered throughout the network (e.g. for the wait, AND-parallel, and OR-parallel rules) requires built-in distributed termination mechanisms. For this purpose, Wave adopts the Echo Algorithms approach. In parallel with the main computation, branches
of a termination tree (called *tracks*) are automatically created each time a wave spawns. When a wave stops processing (dies), an echo (control message) is sent back to its spawning point. Each spawning point waits until it receives echoes from all the waves it spawned, and then sends its own echo to the predecessor spawning point. At the end of all processing (all waves dead), the final echo reaches the entry node, signalising global termination. Some rule constructs, however, can insert control points at intermediary positions of the termination tree, allowing partial termination to be detected. This provides extra flexibility to the programmer, at the expense of a more complex system implementation.

### A.3 An Example Program

Figure A.3 shows a simple Wave program to find the shortest paths of all nodes in an arbitrary connected graph (like the one shown in Figure A.1, with link labels replaced by numeric values), taking as a starting point the node whose name (content) is \texttt{a}. The algorithm is a variation of shortest path algorithm developed by Chandy and Misra [1982]. Five variables are used: the nodal variable \texttt{N1}, that stores the most recently obtained distance from node \texttt{a} to the current node; the nodal variable \texttt{N2}, that stores the address of the predecessor node; the frontal variable \texttt{F1}, that carries the distance obtained by each wave; the environment variable \texttt{L}, that holds the label (in this case, a numeric value) of the passed link; and the environment variable \texttt{P}, that holds the address of the wave predecessor node.

The program begins propagating to node \texttt{a}, where the variables \texttt{F1} and \texttt{N1} are initialised to zero. The next move to be executed is an RP-embraced block, with a *hop to all neighbours* command in the first move. The RP construct then establishes a *control locus* in node \texttt{a} and propagates the tail following the hop operator (#) to all direct neighbours. This tail reaches the neighbours nodes, where it starts execution updating the distance from node \texttt{a}, checking if the node was not previously visited (\texttt{N1} equal to \texttt{null}) or if the newly obtained distance is less than the one already stored in the node, and updating information on the node (\texttt{N1} and \texttt{N2}). For each tail that succeeds, the RP construct transfers itself to the neighbour node and starts new execution there. If all tails originated from the RP construct fail (i.e. both the checks, \texttt{N1}
undefined and new distance < old distance, give FALSE as result), the RP-embraced block also fails and does not transfer itself to other nodes.

```plaintext
@#a.F1=0.N1=0.RP(#.F1+L.N1==,F1<N1.N1=F1.N2=P)
```

hop to node a;
F1 ← 0;
N1 ← 0;
do {
    hop to neighbours;
    F1 ← F1 + L;
    if N1==' ' or F1 < N1 {
        N1 ← F1;
        N2 ← P;
    } else failure;
while not a failure;

Figure A.3 - An Example Wave Program

When all waves finish processing, each node holds the address of its predecessor node in N2 and the distance to it in N1, which permits the reconstruction of the corresponding shortest path. Unfortunately, nodal variables like N1 and N2 are volatile, being removed when the Wave system detects program termination. One way to save this information for later use is to create extra nodes, and save the data as contents of these nodes. Another way is to extend the above program to output its results before termination. This can be achieved by linking two program segments using a SQ rule construct (format: SQ( first_segment, second_segment)). The SQ (sequence) construct suspends execution of the second program segment until all distributed activities associated with the first program segment completely terminate.

The final version of the shortest path program becomes:

```plaintext
SEQ((@#a.F1=0.N1=0.RP(#.F1+L.N1==,F1<N1.N1=F1.N2=P)),
    (@#.FPATH=C.RP(#N2.FPATH&C).T=FPATH))
```

The first segment executes as previously described. The second segment starts when the final echo reaches the entry node, confirming termination of the first part. Then, from the entry node, the tail (FPATH=C.RP(#N2.FPATH&C).T=FPATH) is broadcasted to all nodes. On each node, FPATH is initialised with the corresponding node content (variable C) and the RP construct starts working, propagating to the predecessor node in the shortest path and appending the node content to FPATH. When the wave reaches node a and executes a hop to predecessor (#N2.), the RP construct fails because N2 is undefined in node a. The final move
(T=FPATH) is then executed, sending to the terminal node the sequence of node names that
forms the shortest path, for example:

\[
\begin{align*}
a \\
b; a \\
d; b; a \\
c; d; b; a
\end{align*}
\]
B. Swarm Instruction Set

Conventions:

\( Ad = \text{destination position in AW (AW}[n], 0 <= n <= 255) \)
\( As1 \text{ or } As2 = \text{source positions in AW (AW}[n], 0 <= n <= 255) \)
\( Imm_n = \text{signed immediate value, n bits wide (n = 8, 13, 16, or 24)} \)
\( Imu_n = \text{unsigned immediate value, n bits wide (n = 8 or 16)} \)

Formats:

\[
\begin{array}{c|c|c}
\text{1} & \text{7} & \text{24} \\
\hline
\text{Opcode} & \text{Imm (IP offset)} \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{1a} & \text{7} & \text{10} \\
\hline
\text{Opcode} & \text{As1} & \text{Imm (IP offset)} \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{1b} & \text{7} & \text{10} \\
\hline
\text{Opcode} & \text{Ad} & \text{Imm} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\text{2} & \text{7} & \text{16} \\
\hline
\text{Opcode} & \text{Ad} & \text{As1} & \text{Imm} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
\text{3} & \text{7} & \text{16} \\
\hline
\text{Opcode} & \text{Ad} & \text{As1} & \text{As2} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\text{4} & \text{5} & \text{8} & \text{8} \\
\hline
\text{Opcode} & \text{Imm (high)} & \text{Ad} & \text{As1} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\text{4} & \text{5} & \text{8} & \text{8} \\
\hline
\text{Opcode} & \text{Imm (low)} & \text{Ad} & \text{As1} \\
\end{array}
\]

KEYS:

- PFX = Instruction Prefix
- Opcode = Instruction Code
- Ad = Destination Position in AW
- As1, As2 = Source Position in AW
- Imm = Immediate Value (constant)
- AW = Agent Workspace
- Numbers on top specify the size of each field (in bits)

B.1 Arithmetic Operations

The variable variants (xxxA) interpret As1 and As2 contents as 2's-complement 32-bit integers. The immediate variants (xxxI) interpret As1 contents as a 2's-complement 32-bit integer and Imm as a 2's-complement 8-bit integer converted to 32-bit.

**ADDAA/ADDI**

Mnemonics

<table>
<thead>
<tr>
<th>ADDA</th>
<th>Ad</th>
<th>As1</th>
<th>As2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>Ad</td>
<td>As1</td>
<td>Imm</td>
</tr>
</tbody>
</table>

Format 3 (ADDDA)
2 (ADDI)

Function
Ad ← As1 + As2 (ADDA)
Ad ← As1 + Imm8 (ADDI)

Integer Add: adds the contents of As1 and As2 (or As1 and Imm8) and stores the result in Ad.

SUBA/SUBI

Mnemonics
SUBA Ad As1 As2
SUBI Ad As1 Imm8

Format
3 (SUBA)
2 (SUBI)

Function
Ad ← As1 - As2 (SUBA)
Ad ← As1 - Imm8 (SUBI)

Integer Subtract: subtracts the contents of As2 from As1 (or Imm8 from As1) and stores the result in Ad.

MULA/MULI

Mnemonics
MULA Ad As1 As2
MULI Ad As1 Imm8

Format
3 (MULA)
2 (MULI)

Function
Ad ← As1 * As2 (MULA)
Ad ← As1 * Imm8 (MULI)

Integer Multiply: multiplies the contents of As1 by As2 (or As1 by Imm8) and stores the result in Ad.

DIVA/DIVI

Mnemonics
DIVA Ad As1 As2
DIVI Ad As1 Imm8

Format
3 (DIVA)
2 (DIVI)

Function
Ad ← As1 / As2 (DIVA)
Ad ← As1 / Imm8 (DIVI)
Integer Divide: divides the contents of As1 by As2 (or As1 by Imm\_8) and stores the quotient in Ad.

**MODA/MODI**

**Mnemonics**  
MODA Ad As1 As2  
MODI Ad As1 Imm\_8  

**Format**  
3 (MODA)  
2 (MODI)  

**Function**  
Ad ← As1 mod As2 \hspace{1em} (MODA)  
Ad ← As1 mod Imm\_8 \hspace{1em} (MODI)  

Modulo: divides the contents of As1 by As2 (or As1 by Imm\_8) and stores the rest in Ad.

**B.2 Logical Operations**

The variable variants (xxxA) interpret As1 and As2 contents as 2’s-complement 32-bit integers. The immediate variants (xxxl) interpret As1 contents as a 2’s-complement 32-bit integer and Imm\_8 as a 2’s-complement 8-bit integer converted to 32-bit.

**EQLA/EQLI**

**Mnemonics**  
EQLA Ad As1 As2  
EQLI Ad As1 Imm\_8  

**Format**  
3 (EQLA)  
2 (EQLI)  

**Function**  
Ad ← 1, if As1 = As2 \hspace{1em} (EQLA)  
Ad ← 0, if As1 ≠ As2 \hspace{1em} (EQLA)  
Ad ← 1, if As1 = Imm\_8 \hspace{1em} (EQLI)  
Ad ← 0, if As1 ≠ Imm\_8 \hspace{1em} (EQLI)  

Equal: stores 1 (true) in Ad if the content of As1 is equal to the content of As2 (or to Imm\_8), or 0 (false) otherwise.

**NEQA/NEQI**

**Mnemonics**  
NEQA Ad As1 As2  
NEQI Ad As1 Imm\_8  

**Format**  
3 (NEQA)  
2 (NEQI)  

Equal: stores 1 (true) in Ad if the content of As1 is equal to the content of As2 (or to Imm\_8), or 0 (false) otherwise.
Function

Ad $\leftarrow 1$, if $A_{sl} \neq A_{s2}$ (NEQA)  
Ad $\leftarrow 0$, if $A_{sl} = A_{s2}$ (NEQA)  
Ad $\leftarrow 1$, if $A_{sl} \neq I_{mm8}$ (NEQI)  
Ad $\leftarrow 0$, if $A_{sl} = I_{mm8}$ (NEQI)

Not Equal: stores 1 (true) in Ad if the content of $A_{sl}$ is different from the content of $A_{s2}$ (or from $I_{mm8}$), or 0 (false) otherwise.

**GRTA/GRTI**

Mnemonics

GRTA Ad Asl As2  
GRTI Ad Asl Imm8

Format 3 (GRTA)  
2 (GRTI)

Function

Ad $\leftarrow 1$, if $A_{sl} > A_{s2}$ (GRTA)  
Ad $\leftarrow 0$, if $A_{sl} \leq A_{s2}$ (GRTA)  
Ad $\leftarrow 1$, if $A_{sl} > I_{mm8}$ (GRTI)  
Ad $\leftarrow 0$, if $A_{sl} \leq I_{mm8}$ (GRTI)

Greater Than: stores 1 (true) in Ad if the content of $A_{sl}$ is greater than the content of $A_{s2}$ (or than $I_{mm8}$), or 0 (false) otherwise.

**LSTA/LSTI**

Mnemonics

LSTA Ad Asl As2  
LSTI Ad Asl Imm8

Format 3 (LSTA)  
2 (LSTI)

Function

Ad $\leftarrow 1$, if $A_{sl} < A_{s2}$ (LSTA)  
Ad $\leftarrow 0$, if $A_{sl} \geq A_{s2}$ (LSTA)  
Ad $\leftarrow 1$, if $A_{sl} < I_{mm8}$ (LSTI)  
Ad $\leftarrow 0$, if $A_{sl} \geq I_{mm8}$ (LSTI)

Less Than: stores 1 (true) in Ad if the content of $A_{sl}$ is less than the content of $A_{s2}$ (or than $I_{mm8}$), or 0 (false) otherwise.

**GEQA/GEQI**

Mnemonics

GEQA Ad Asl As2  
GEQI Ad Asl Imm8

Format 3 (GEQA)  
2 (GEQI)

139
Function

\[ \text{Ad} \leftarrow 1, \text{if } A_{sl} \geq A_{s2} \quad (\text{GEQA}) \]
\[ \text{Ad} \leftarrow 0, \text{if } A_{sl} < A_{s2} \quad (\text{GEQA}) \]
\[ \text{Ad} \leftarrow 1, \text{if } A_{sl} \geq \text{Imm}_8 \quad (\text{GEQI}) \]
\[ \text{Ad} \leftarrow 0, \text{if } A_{sl} < \text{Imm}_8 \quad (\text{GEQI}) \]

Greater or Equal: stores 1 (true) in Ad if the content of As1 is greater than or equal to the content of As2 (or to Imm\(_8\)), or 0 (false) otherwise.

LEQA/LEQI

Mnemonics

<table>
<thead>
<tr>
<th>LEQA</th>
<th>Ad</th>
<th>Asl</th>
<th>As2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEQI</td>
<td>Ad</td>
<td>Asl</td>
<td>Imm(_8)</td>
</tr>
</tbody>
</table>

Format

3 (LEQA)
2 (LEQI)

Function

\[ \text{Ad} \leftarrow 1, \text{if } A_{sl} \leq A_{s2} \quad (\text{LEQA}) \]
\[ \text{Ad} \leftarrow 0, \text{if } A_{sl} > A_{s2} \quad (\text{LEQA}) \]
\[ \text{Ad} \leftarrow 1, \text{if } A_{sl} \leq \text{Imm}_8 \quad (\text{LEQI}) \]
\[ \text{Ad} \leftarrow 0, \text{if } A_{sl} > \text{Imm}_8 \quad (\text{LEQI}) \]

Less or Equal: stores 1 (true) in Ad if the content of As1 is less than or equal to the content of As2 (or to Imm\(_8\)), or 0 (false) otherwise.

B.3 Boolean (Bitwise) Operations

The variable variants (xxxA) interpret As1 and As2 contents as unsigned 32-bit patterns. The immediate variants (xxxI) interpret As1 contents as an unsigned 32-bit pattern and Imm\(_8\) as an unsigned 8-bit pattern converted to 32-bit.

ANDA/ANDI

Mnemonics

<table>
<thead>
<tr>
<th>ANDA</th>
<th>Ad</th>
<th>As1</th>
<th>As2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDI</td>
<td>Ad</td>
<td>As1</td>
<td>Imm(_8)</td>
</tr>
</tbody>
</table>

Format

3 (ANDA)
2 (ANDI)

Function

\[ \text{Ad} \leftarrow A_{sl} \text{ AND } A_{s2} \quad (\text{ANDA}) \]
\[ \text{Ad} \leftarrow A_{sl} \text{ AND } \text{Imm}_8 \quad (\text{ANDI}) \]

Bitwise AND: stores in Ad the result of a bitwise AND between the contents of As1 and As2 (or Imu\(_8\)).
IORA/IORI

Mnemonics
IORA Ad As1 As2
IORI Ad As1 Imu8

Format
3 (IORA)
2 (IORI)

Function
Ad ← As1 OR As2 (IORA)
Ad ← As1 OR Imu8 (IORI)

Bitwise OR: stores in Ad the result of a bitwise inclusive OR between the contents of As1 and As2 (or Imu8).

XORA/XORI

Mnemonics
XORA Ad As1 As2
XORI Ad As1 Imu8

Format
3 (XORA)
2 (XORI)

Function
Ad ← As1 XOR As2 (XORA)
Ad ← As1 XOR Imu8 (XORI)

Bitwise XOR: stores in Ad the result of a bitwise exclusive OR between the contents of As1 and As2 (or Imu8).

SHLA/SHLI

Mnemonics
SHLA Ad As1 As2
SHLI Ad As1 Imu8

Format
3 (SHLA)
2 (SHLI)

Function
Ad ← As1 SHL As2 (SHLA)
Ad ← As1 SHL Imu8 (SHLI)

Shift Left: shifts left the content of As1, by the number of bit positions specified by As2 (or Imu8), storing the result in Ad. Incoming bit positions at right are filled with zeroes.

SHRA/SHRI

Mnemonics
SHRA Ad As1 As2
SHRI Ad As1 Imu8

Format
3 (SHRA)
Function $Ad \leftarrow As1 \text{ SHR } As2$ (SHRA)  
Function $Ad \leftarrow As1 \text{ SHR } Imu8$ (SHRI)

Shift Right: shifts right the content of $As1$, by the number of bit positions specified by $As2$ (or $Imu8$), storing the result in $Ad$. Incoming bit positions at left are filled with zeroes.

SARA/SARI

<table>
<thead>
<tr>
<th>Mnemonics</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>SARA</td>
<td>3 (SARA)</td>
</tr>
<tr>
<td>SARI</td>
<td>2 (SARI)</td>
</tr>
</tbody>
</table>

Function $Ad \leftarrow As1 \text{ SAR } As2$ (SARA)  
Function $Ad \leftarrow As1 \text{ SAR } Imu8$ (SARI)

Arithmetic Shift Right: shifts right the content of $As1$, by the number of bit positions specified by $As2$ (or $Imu8$), storing the result in $Ad$. Incoming bit positions at left are filled with the value of the original most significant bit (equivalent to the sign bit) in $As1$.

**B.4 Data Transfer Operations**

**LDTW**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDTW</td>
<td>4</td>
</tr>
</tbody>
</table>

Function $Ad \leftarrow \text{TW}[As1 + \text{Imm}_{13}]$

Load from Transient Workspace: loads $Ad$ with the content of the nodal transient workspace position $\text{TW}[As1 + \text{Imm}_{13}]$.

**STTW**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>STTW</td>
<td>4</td>
</tr>
</tbody>
</table>

Function $\text{TW}[Ad + \text{Imm}_{13}] \leftarrow \text{As1}$
Store to Transient Workspace: stores the content of As1 in the nodal transient workspace position TW[Ad + Imm₁₃].

LDPW

Mnemonic LDPW Ad PW[As1 + Imm₁₃]
Format 4
Function Ad ← PW[As1 + Imm₁₃]

Load from Persistent Workspace: loads Ad with the content of the nodal persistent workspace position PW[As1 + Imm₁₃].

STPW

Mnemonic STPW Ad TW[As1 + Imm₁₃]
Format 4
Function PW[Ad + Imm₁₃] ← As1

Store to Persistent Workspace: stores the content of As1 in the nodal persistent workspace position TW[Ad + Imm₁₃].

LDAW

Mnemonic LDAW Ad AW[As1 + Imm₈]
Format 2
Function Ad ← AW[As1 + Imm₈]

Load from Agent Workspace: loads Ad with the content of the agent workspace position PW[As1 + Imm₈].

STAW

Mnemonic STAW Ad AW[As1 + Imm₁₃]
Format 2
Function AW[Ad + Imm₈] ← As1

Stores to Agent Workspace: stores the content of As1 in the agent workspace position AW[Ad + Imm₈].
SETH

Mnemonic  SETH  Ad  Imu_{16}
Format    1b
Function  \[ \begin{align*} 
& Ad_{\text{high}} \leftarrow \text{Imui}_{16} \\
& Ad_{\text{low}} \leftarrow 0_{16} 
\end{align*} \]

Set High: loads the high 16-bit half of Ad with the given immediate value [Imu_{16}] and the low 16-bit half of Ad with zeroes.

SETL

Mnemonic  SETL  Ad  Imu_{16}
Format    1b
Function  \( Ad_{\text{low}} \leftarrow \text{Imui}_{16} \)

Set Low: loads the low 16-bit half of Ad with the given immediate value [Imu_{16}] and leaves the high 16-bit half of Ad unchanged.

B.5 Control-Flow Operations

BIFT

Mnemonic  BIFT  As1  Imm_{16}
Format    1a
Function  \[ \text{if As1} \neq 0, \; \text{IP} \leftarrow \text{IP} + \text{Imm}_{16} \]

Branch If True: if the content of As1 is true (\( \neq 0 \)), add Imm_{16} to the content of IP (instruction pointer).

BIFF

Mnemonic  BIFF  As1  Imm_{16}
Format    1a
Function  \[ \text{if As1} = 0, \; \text{IP} \leftarrow \text{IP} + \text{Imm}_{16} \]
Branch If False: if the content of Asl is false (= 0), add Imm_{16} to the content of IP (instruction pointer).

**JMPR**

Mnemonic: JMPR Imm_{24}

Format: 0

Function: IP ← IP + Imm_{24}

Relative Jump: add Imm_{24} to the content of IP (instruction pointer).

**JMPA**

Mnemonic: JMPA Asl

Format: 3

Function: IP ← IP + Asl

Absolute Jump: add the content of Asl to the content of IP (instruction pointer).

**JSVR**

Mnemonic: JSVR Imm_{24}

Format: 0

Function: AW[255] ← IP, IP ← IP + Imm_{24}

Relative Save and Jump: save the current content of IP (instruction pointer) in the agent workspace position AW[255], and then add Imm_{24} to the content of IP.

**JSVA**

Mnemonic: JSVA Asl

Format: 3

Function: AW[255] ← IP, IP ← IP + Asl

Absolute Save and Jump: save the current content of IP (instruction pointer) in the agent workspace position AW[255], and then add the content of Asl to the content of IP.
JRET

Mnemonic: JRET
Format: 3
Function: $IP \leftarrow AW[255]$

Return from Jump: restore the content saved in the agent workspace position $AW[255]$ back to IP (instruction pointer).

B.6 Agent-Control Operations

SPWN

Mnemonic: SPWN
Format: 3
Function

Spawn: sends one copy of the current agent to each node address in the agent Destination List.

STOP

Mnemonic: STOP
Format: 3
Function

Stop Agent: stops execution of the current agent, cancelling it.

TERM

Mnemonic: TERM
Format: 3
Function

Terminate Task: terminates execution of the current task.
SUSP

Mnemonic: SUSP As1
Format: 3
Function

Suspend Agent: suspends the current agent and places it in the Suspending Buffer, under the event value given by the content of As1.

WAKE

Mnemonic: WAKE As1
Format: 3
Function

Wake Up Agent: awakes all agents in the Suspending Buffer, which are suspended under the event value given by the content of As1. These agents will queue for execution in the current node, after the current agent leaves it, having priority over new incoming agents.

LDSW

Mnemonic: LDSW Ad
Format: 3
Function: Ad ← ASW

Load from ASW: loads Ad with the content of the agent status word (ASW).

STSW

Mnemonic: STSW As1
Format: 3
Function: ASW ← As1

Store to ASW: stores the content of As1 in the agent status word (ASW).
B.7 Special Operations

CRND

Mnemonic   CRND Ad As1
Format     3
Function

Create Node: creates a new node in the current PM, of type given by the content of As1 (As1 = 0 for a data node), returning the node address in Ad.

DTND

Mnemonic   DTND As1
Format     3
Function

Destroy Node: destroys the node in the current PM that has the address given by the content of As1.

INDL

Mnemonic   INDL As1
Format     3
Function   DL <= As1

Insert into Destination List: the node address stored in As1 is inserted in the agent Destination List. If the Destination List is already full, the value is lost.

INLT

Mnemonic   INLT As1 As2
Format     3
Function   LT <= pair(naddr = As1, lname = As2)

Insert into Link Table: the pair node address (content of As1) + link name (content of As2) is inserted in the Link Table of the current node.
**DLLT**

**Mnemonic**  DLLT As1 As2  
**Format**  3  
**Function**  $\downarrow \text{LT}(\text{naddr} = \text{As1}, \text{lname} = \text{As2})$

Delete from Link Table: the pair node address (content of As1) + link name (content of As2) is deleted from the Link Table of the current node.

**LULT**

**Mnemonic**  LULT Ad As1  
**Format**  3  
**Function**  $\text{Ad} \leftarrow \text{LT}(\text{name} = \text{As1}).\text{naddr}$

Lookup Link Table: searches in the Link Table of the current node, returning in Ad the node address that matches the given link name (content of As1). If there is no entry with the given link name, returns EMPTY (80000000h hex). Successive queries with the same link name will return node addresses that match that link name, until Empty is returned signalling end-of-list. Successive queries with the a link name equal to EMPTY will return all stored node addresses, regardless of their real link names, until Empty is returned signalling end-of-list.

**RLLT**

**Mnemonic**  RLLT Ad As1  
**Format**  3  
**Function**  $\text{Ad} \leftarrow \text{LT}(\text{naddr} = \text{As1}).\text{name}$

Reverse Lookup Link Table: searches in the Link Table of the current node, returning in Ad the link name that matches the given node address (content of As1). If there is no entry with the given node address, returns EMPTY (80000000h hex). Successive queries with the same node address will return link names that match that node address, until Empty is returned signalling end-of-list. Successive queries with the a node address equal to EMPTY will return all stored link names, regardless of their real node addresses, until Empty is returned signalling end-of-list.
LULD

Mnemonic  LULD Ad Asl
Format  3
Function  Ad ← LD(key = Asl).naddr

Lookup Local Directory: searches in the Local Directory of the current PM, returning in Ad the node address that matches the given value of node key (content of Asl). If there is no entry with the given key, returns EMPTY (80000000|!C x). Successive queries with the same key will return node addresses that match that key, until Empty is returned signalling end-of-list.

LUST

Mnemonic  LUST Ad Asl
Format  3
Function  Ad ← ST(key = Asl).naddr

Lookup Spatial Table: if the given key (content of Asl) is zero, returns in Ad the address of the console node for the current task. If the given key is different from zero, returns in Ad the address of the root node of another PM in the system.

B.8 Task Directives

$SET TW

Mnemonic  $SET TW[n] value
Format  (not applicable)
Function

Set TW Position: initialises the given nodal transient workspace position (TW[n]) with the given 32-bit value. The default is initialising all TW positions with EMPTY.

$SET TDDL

Mnemonic  $SET TDDL value
Format  (not applicable)
Function

Set Termination Detection Delay: sets the value, in seconds, of the delay to start automatic termination detection (TD) for the task. If the value is zero, TD is disabled. The default is TD enabled, with delay of 1 second.

$SET TONEXC

Mnemonic  $SET TONEXC

Format      (not applicable)

Function

Set for Termination on Exception: sets the task to terminate in case of exceptions. The default is to ignore exceptions.
C. Swarm Implementation

C.1 Instruction Opcodes

**SWARM Instruction Set Opcodes (in hexadecimal)**

7-bit opcodes (prefix = 0):

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SPWN</td>
<td>10</td>
</tr>
<tr>
<td>01</td>
<td>STOP</td>
<td>11</td>
</tr>
<tr>
<td>02</td>
<td>TERM</td>
<td>12</td>
</tr>
<tr>
<td>03</td>
<td>SUSP</td>
<td>13</td>
</tr>
<tr>
<td>04</td>
<td>WAKE</td>
<td>14</td>
</tr>
<tr>
<td>05</td>
<td>LDSW</td>
<td>15</td>
</tr>
<tr>
<td>06</td>
<td>STSW</td>
<td>16</td>
</tr>
<tr>
<td>07</td>
<td>RIFT</td>
<td>17</td>
</tr>
<tr>
<td>08</td>
<td>RIFF</td>
<td>18</td>
</tr>
<tr>
<td>09</td>
<td>JMPR</td>
<td>19</td>
</tr>
<tr>
<td>0A</td>
<td>JMPA</td>
<td>1A</td>
</tr>
<tr>
<td>0B</td>
<td>JSVR</td>
<td>1B</td>
</tr>
<tr>
<td>0C</td>
<td>JSVA</td>
<td>1C</td>
</tr>
<tr>
<td>0D</td>
<td>JRET</td>
<td>1D</td>
</tr>
<tr>
<td>0E</td>
<td>CRND</td>
<td>1E</td>
</tr>
<tr>
<td>0F</td>
<td>DTND</td>
<td>1F</td>
</tr>
</tbody>
</table>

2-bit opcodes (prefix = 1):

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LDTW</td>
</tr>
<tr>
<td>1</td>
<td>STTW</td>
</tr>
<tr>
<td>2</td>
<td>LDPW</td>
</tr>
<tr>
<td>3</td>
<td>STPW</td>
</tr>
</tbody>
</table>

C.2 Internal Protocols

**SWARM Experimental Environment - INTERNAL PROTOCOLS**

*** NOTES ***

* RM TO AP INTERFACE *

While there is no AP connected, RM stays idle.
As soon as AP opens connection (two named pipes), RM enters in
COMMAND MODE, when it receives and executes commands from AP.
The commands are:

- CLOSE CONNECTION WITH AP
- CREATE A PM
- REMOVE A PM
- GET NODES FROM A PM
- PUT NODES IN A PM
- LIST PMs
- RUN TASK

Each AP command is 32-bit long and may (or not) be followed by extra
information:

- the UNIX PID of a module (a 32-bit int numbers);
- the ID of the PM to be accessed (a 32-bit number);
- the number of 32-bit elements to be transferred (a 32-bit number);
- data to be transferred (each data element being a 32-bit value).

* RH TO PM INTERFACE *

PM starts in COMMAND MODE, when it receives and executes commands from RM. The commands are:

- GET NODES FROM A PM
- PUT NODES IN A PM
- RUN TASK

Each RM command is 32-bit long and may (or not) be followed by extra information:

- a number of 32-bit parameters;
- the number of 32-bit elements to be transferred (a 32-bit number);
- data to be transferred (each data element being a 32-bit value).

* BOTH INTERFACES *

In case of command failure, the 32-bit value FAILURE (a negative value) is sent.

Command RUN TASK loads a program and makes PMs, RM and AP enter the EXECUTION MODE. In this mode, the RM only routes agents between modules, while PMs and AP receive, execute and produce agents. After termination of the executing task, RM, AP and PMs return to COMMAND MODE.

The protocol for each command is as follows (SUCC == on success; FAIL == on failure):

*** OPEN CONNECTION WITH AP (not a direct command) ***

AP

RM

open APtrRM

pipe for wr  open APtrRM

pipe for rd

open RPMoAP

pipe for rd  open RPMoAP

pipe for wr

--------- AP UNIX PID --------->

<-------- RM UNIX PID ----------

*** CLOSE CONNECTION WITH AP ***

AP

RM

--------- CLOSE_AP --------->

<-------- AP UNIX PID ----------

*** CREATE A PM (not a direct command for a PM) ***

AP

RM

--------- CREATE_PM --------->

fork PM - - - - - - - - - - >

--------- PM ID --------->

<--------- PM ID -------- (SUCC)
<-------- FAILURE ------ (FAIL)

<-------- new PM ID ------ (SUCC)

<-------- FAILURE ------ (FAIL)

*** REMOVE A PM (not a direct command for a PM) ***

**AP**

**RM**

----------------------- REMOVE_PM ---------------------->

----------------------- PM ID ------------------------->

SIGTERM
to PM - - - - - - - - - - - ->

cleanup
and die

------------------ removed PM ID ------- (SUCC)

<-------- FAILURE ------ (FAIL)

*** GET NODES FROM A PM ***

**AP**

**RM**

----------------------- GET_NODES --------------------->

----------------------- PM ID ------------------------->

----------------------- GET_NODES --------------------->

<-------- count ------ (SUCC)
<-------- data ------ (SUCC)
<-------- FAILURE ------ (FAIL)

<-------- count ------ (SUCC)
<-------- data ------ (SUCC)
<-------- FAILURE ------ (FAIL)

*** PUT NODES IN A PM ***

**AP**

**RM**

----------------------- PUT_NODES --------------------->

----------------------- PM ID ------------------------->

----------------------- PUT_NODES --------------------->

----------------------- count ------------------>

----------------------- data ---------------------->

----------------------- count ------------------>

----------------------- data ---------------------->

<-------- count ------ (SUCC)
<-------- FAILURE ------ (FAIL)

<-------- count ------ (SUCC)
<-------- FAILURE ------ (FAIL)
*** LIST PMS ***

AP                      RM
--------------------- LIST_PMS ------------------>
<---------- count ---------- (SUCC)
<---------- data ---------- (SUCC)
<---------- FAILURE ------- (FAIL)

*** RUN TASK ***

AP                      RM                      PM
--------------------- RUN_TASK ------------------>
--------------------- count --------------------->
--------------------- program --------------------->
--------------------- RUN_TASK --------------------->
--------------------- count --------------------->
--------------------- program --------------------->
<---------- count ---------- (SUCC)
<---------- FAILURE ------- (FAIL)
<---------- RUN_TASK --------- (SUCC)
<---------- no of PMs ------ (SUCC)
<---------- next PM ID ----- (SUCC)
<---------- FAILURE -------- (FAIL)
<---------- FAILURE ------------ (FAIL)

EXEC MODE               EXEC MODE               EXEC MODE
(SUCC)                  (SUCC)                  (SUCC)
<---------- TERM agent ------- TERM agent ------->
<---------- TACK agent ------- TACK agent ------->

*** invalid command ***

AP                      RM
--------------------- invalid command ---->
<---------- FAILURE ------------------>

*** SYSTEM SHUTDOWN (not a command) ***

AP                      RM                      PM
SIGTERM -----> SIGTERM  SIGTERM
<---------- SIGTERM ---< to RM
<---------- SIGTERM ---< to AP
cleanup
and die
SIGTERM

155
C.3 File Formats

**SWARM Experimental Environment - FILE FORMATS**

**PM Configuration File:**

- `COUNT` (number of 32-bit words that follow)
- `PM_ID` (PM ID)
- `NNODES` (number of nodes in this PM)
- `LASTNID` (ID of the last node in this list)

<table>
<thead>
<tr>
<th>NODE_ID (node ID)</th>
<th>repeated</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PWS_SIZE</code> (size of persistent workspace, in 32-bit words)</td>
<td>for each node</td>
</tr>
<tr>
<td><code>LT_SIZE</code> (size of link table, in 32-bit words)</td>
<td>node</td>
</tr>
<tr>
<td><code>PWS_CONT</code> (contents of persistent workspace)</td>
<td></td>
</tr>
<tr>
<td><code>LT_CONT</code> (contents of link table)</td>
<td></td>
</tr>
</tbody>
</table>

**Program File (Machine Code):**

- `COUNT` (number of 32-bit words that follow)

<table>
<thead>
<tr>
<th>Task Parameters Section</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>APID</code> (AP ID)</td>
<td></td>
</tr>
<tr>
<td><code>NEXTPMID</code> (ID of a &quot;next&quot; PM)</td>
<td></td>
</tr>
<tr>
<td><code>TPARSZ</code> (size of task parameters section, in 32-bit words)</td>
<td></td>
</tr>
<tr>
<td><code>CODESZ</code> (size of code section, in 32-bit words)</td>
<td></td>
</tr>
<tr>
<td><code>CDATASZ</code> (size of constant data section, in 32-bit words)</td>
<td></td>
</tr>
<tr>
<td><code>TWSSZ</code> (initial size for nodal transient workspaces, in 32-bit words)</td>
<td></td>
</tr>
<tr>
<td><code>TDDELAY</code> (delay, in seconds, to initiate termination detection cycle)</td>
<td></td>
</tr>
<tr>
<td><code>RFU</code> (reserved for future use)</td>
<td></td>
</tr>
<tr>
<td><code>RFU</code> (reserved for future use)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code Section</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constant Data Section</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>constant data</td>
<td></td>
</tr>
</tbody>
</table>

C.4 Internal Structures

**SWARM Experimental Environment - INTERNAL STRUCTURES**

/* PERSISTENT STRUCTURES */

/* RM Address Book element structure */

typedef int adbentry[4];

/* The Address Book is a dynamically allocated array of elements *adbentry* */

/* In the Address Book structure, each entry has the format:

- `adbook[PM ID][0]` = adbook[PM ID][READ] = fd of downstream end of PM to RM pipe
- `adbook[PM ID][1]` = adbook[PM ID][WRITE] = fd of upstream end of RM to PM pipe
- `adbook[PM ID][2]` = adbook[PM ID][UPID] = Unix PID (process ID) of the PM process
- `adbook[PM ID][3]` = adbook[PM ID][NXT] = pointer to the next (active) entry

NOTE: AP is considered a PM with PM ID == 0, using entry adbook[0][x] */
/* PM Nodal Header element structure */
struct nodehd {
    int *pws;    /* Persistent Nodal Workspace */
    int pwssize; /* Size of Persistent Nodal Workspace */
    int *lt;     /* Nodal Link Table */
    int ltsize;  /* Size of Nodal Link Table */
};

/* The PM Nodal Store is a dynamically allocated array of pointers to elements */
/* The first entry in the Nodal Store holds the Nodal Store header: */
node[0].pws[0] == node[0].pws[PMID] == pmID
node[0].pws[1] == node[0].pws[NNODES] == number of nodes currently stored
node[0].pws[2] == node[0].pws[STORESIZE] == current size of Nodal store

/* Persistent Nodal Workspace */

/* Nodal Link Table */

dynamically allocated array of pointers to elements

/* PM Local Directory Key List structure */
struct ldkeyl {
    int keyval;   /* Key value */
    struct ldnodel *nodelst; /* node list for that key */
    struct ldkeyl *next; /* pointer to next key in the list */
};

/* PM Local Directory Node List structure */
struct ldnodel {
    int nodeid;   /* Node ID */
    struct ldnodel *next; /* pointer to next node in the list */
};

/* The PM Local Directory is a dynamically allocated array of pointers to elements */

/* ********************************************** */

/* TRANSIENT STRUCTURES */

/* PM Program Store structure */
/* The PM Program Store is a dynamically allocated array of int with the size given by */
/* the program parameters: TPARSZ + CODESZ + CDATASZ + 1. The last position is the "null" */
/* position used to received writes to read-only elements in the systems (e.g. AW[0]). */
/* */

/* Agent Execution Buffer structure */
/* The Agent Execution Buffer is a dynamically allocated array of int with the size */
/* given by: (agent header size) + (maximum size for agent workspace) + (maximum size for agent */
/* destination list). For the present prototype implementation, these values are, */
/* respectively: 4 + 256 + 256. */
/* */

/* Task Transient Workspace element structure */
struct ntws {
    int size;    /* current size of Nodal Transient Workspace */
    int *data;   /* data positions */
};

/* The Task Transient Workspace is a dynamically allocated array of pointers to */
/* elements */
/* Each entry in the Task Transient Workspace (i.e. each ntws element) corresponds to */
/* a node */
/* present at the PM Nodal Store. Allocation of a Transient Workspace for a node is */
/* postponed until */
/* the first reference to that node in the task (i.e. the first time it is visited by an */
/* agent on */
/* that task ). */
/* */

/* PM Suspending Buffer Event List structure */
struct sbevl {
    int evnb;    /* event number */
    struct sbagl *agentlist; /* suspended agent list for that event */
    struct sbevl *next;    /* pointer to next event in the list */
};

/* PM Suspending Buffer Agent List structure */
struct sbagl {
    int *agent; /* pointer to the suspended agent */
    struct sbagl *next; /* pointer to next agent in the list */
};

157
/* The PM Suspending Buffer is a dynamically allocated array of pointers to elements */

/* PM Release List Event List structure */
struct rlevl {
    int evval; /* event number */
    int nodeid; /* ID of node where the event WAKE operation was executed */
    rlevl *next; /* pointer to next event in the list */
};

/* The PM Release List is a companion structure to the Suspending Buffer, that holds a list of events "released" by the currently executing agent */

/* PM Task Environment structure (holds pointers to all structures used by the task) */
struct tenv {
    struct nodehd **node; /* pointer to PM Nodal Store */
    struct ldkeyl **ldir; /* pointer to PM Local Directory */
    int *tpar; /* pointer to Task Parameters Store */
    int *tcode; /* pointer to Task Code Store */
    int *tcdat; /* pointer to Task Constant Data Store */
    int *tndat; /* pointer to Task "Null Element" Position */
    struct ntws **twbs; /* pointer to Task Transient Workspaces */
    int *aghd; /* pointer to Agent Header */
    int *agws; /* pointer to Agent Workspace */
    int *agdl; /* pointer to Agent Destination List */
    struct wbevl **wbuf; /* pointer to Task Suspending Buffer */
    struct rlevl *rlist; /* pointer to Task Release List */
};

/** **************************************************************************
C.5 High-Level Algorithms

C.5.1 Routing Module (RM)

**************************************************************************
main {
    initialise signal handling;
    create and initialise log file;
    create and initialise Address Book;
    create AP to RM FIFO;
    create RM to AP FIFO;
    wait for AP to connect;

    /* Command Loop */
    while(1) {
        block until AP sends command;
        switch(AP command) {

**************************************************************************
case CLOSE_AP:
    disconnect AP;
    wait for AP to connect;
    break;

case CREATE_PM:
    search Address Book for first vacant position;
    if Address Book is full, try to extend it;
    try to create PM;
    return PM ID or FAILURE token;
    break;

case REMOVE_PM:
    get (from AP) the PM ID;
    try to remove PM;
    try to resize Address Book to a smaller size;
    return PM ID or FAILURE token;
    break;

case GET_NODES:
    get (from AP) the PM ID;
    transfer data from PM to AP;
    break;

case PUT_NODES:
    get (from AP) the PM ID;
    transfer data from AP to PM;
    break;

case LIST_PMS:
    return FAILURE and break if there are no PMs;
    output number of elements to be transferred;
    output list of PM IDs;
    break;

case RUN_TASK:
    call task_exec;
    break;

default:
    return FAILURE to AP;

) /* end of switch */
) /* end of while */

} /* END OF MAIN */

/* ************************************************************ */

/ * Transfers program to all PMs, aborting in case of error. If successful, enters EXECUTION MODE, routing incoming agents to their destinations. */

{ get the number of words to be transferred from AP;
  get and discard dummy values of AP ID and 'next' PM ID;
  initialise command, AP ID and number of PMs;
  for all active PMs: {
    send command;
    send no. of bytes to transfer;
    send AP ID for the task;
    send ID of 'next' PM;
  }
  transfer program from AP to all PMs;
  get acknowledgement from all PMs;
  if one or more PMs returned FAILURE {
    abort operation by sending FAILURE to AP and all PMs;
    return;
  }
  else { /* no PM returned FAILURE */
    re-send the RUN_TASK command to AP and all PMs, switching them to EXECUTION MODE;
    send to AP the no of PMs and the ID of 'next' PM;
  }

} /* Task Execution Loop */
while(not terminated) {
    wait for any pipe ready to be read;
for each ready pipe: {
  read agent header into agent buffer;
  if that's a normal agent {
    load agent body;
    replicate agent to its destinations;
  }
  else { /* that's a control agent */
    switch(control type) {
      case TTSK: /* if a termination agent */
        exit all loops (switch, for and while);
        break;
      case TENQ: /* if a termination enquiry agent */
        send enquiry to AP and all PMs;
        break;
      case TECH: /* if a termination echo agent */
        send agent to its destination;
        break;
    } /* end of switch */
  } /* end of else */
} /* end of for loop */

notify AP and all PMs of termination by broadcasting TTSK agent;
clear all incoming pipes;

/** .............................................................. */
/**
 ** Routing Module (RM)
 */
/** .............................................................. */

C.5.2 Processing Module (PM)

/** .............................................................. */
/**
 ** SWARM Experimental Environment - version 0.3
 ** Processing Module (PM)
 */
/**
 ** DECEMBER 1995
 */
/**
 ** Copyright 1995 Luciano de Errico
 */
/** .............................................................. */

main {
  get PM ID from RM;
  initialise signal handling;
  create and initialise log file;
  create and initialise Nodal Store;
  create and initialise Local Directory;
  if all initialisations were OK, return PM ID to RM;
  else return FAILURE to RM;

  /* Execution Loop */
  while(1) {
    block until RM sends a command;
    switch(RM command) {
      case GET_NODES:
        output contents of Nodal Store to RM;
        break;
      case PUT_NODES:
        input contents of Nodal Store from RM;
        break;
      case RUN_TASK:
        call task_exec;
    } /* end of switch */
  } /* end of while */
} /* END OF MAIN */

160
/* Task Execution */
{
create transient structures;
if there was a problem in the initialisation, send FAILURE to RM;
else {
    return count to RM;
    get return code from RM;
    if RM returned FAILURE {
        free all dynamically allocated structures;
        return;
    }
initialise internal variables;
/* Execution Loop */
while(!terminated) {
    if there are agents released from the Suspending Buffer,
        get one of them;
    else {
        block until getting an agent from the input queue;
        if normal agent, decrement agent counter (for termination detection);
    }
    if that's a normal agent {
        call exec_agent;
        increment agent counter for each agent sent to output;
    } else { /* that's a control agent */
        switch(control type) {
            case TTSK: /* if a termination agent */
                exit the while loop;
                break;
            case TENQ: /* if a termination enquiry agent */
                assembly and send TECH agent to TD initiator (AP);
                break;
        } /* end of switch */
    } /* end of else */
} /* end of while loop */
acknowledge termination by sending a TACK agent to RM;
free all dynamically allocated structures;
}
/* exec_agent */
/* executes normal agent */
{
    initialise local variables;
    if there is no nodal transient wa, create and initialise it;
    /* Agent Processing Loop */
    while(!not end of processing) {
        if agent was set to stop on non-existent node and the 'non-existent node' flag is set
            or if IP points beyond code, set instruction to STOP;
        else
            fetch instruction pointed by IP and increment IP;
            get opcode;
            switch(opcode) {
                case SPWN, STOP, TERM or SUSP:
                    exit the while loop;
                    break;
                case WAKE, LDSW, STSW, CRND, DNSD, INDL, INLT, DLMT, LMT, LMTL, LMTD, or LUST:
                    execute proper actions;
                    break;
                case any arithmetic-logical instruction:
                    execute proper actions;
                    break;
                } /* end of switch */
            } /* end of while */
    } /* if current node was destroyed or node key changed, update Local Directory;
            switch(opcode) {
                case TERM:

161
assembly and send TTSK agent to RM;
break;
  case SUSP:
    get event number;
    place agent in the Suspending Buffer;
    break;
  case SPWN:
    place agent in the output queue;
    break;
  }
}

C.5.3 Access Point (AP)

main
{
initalise signal handling;
open pipes to/from RM;
send AP UNIX PID to RM;
get RM UNIX PID;
/* Command Loop */
while(1) {
  display user menu;
  get user command;
  switch(user command) {
    case LIST_PMS:
      send command to RM;
      get reply from RM;
      display information;
      break;
    case CREATE_PM:
      send command to RM;
      get reply from RM;
      display information;
      break;
    case REMOVE_PM:
      get PM ID from user;
      send command and PM ID to RM;
      get reply from RM;
      display information;
      break;
    case GET_NODES:
      get PM ID and name of the output file from user;
      try to open file;
      if failure, display error message and break;
      send command and PM ID to RM;
      get count (size) and data from RM, and write them to file;
close file;
display information;
break;

case PUT_NODES:
    get PM ID and name of the input file from user;
    try to open file;
    if failure, display error message and break;
    send command and PM ID to RM;
    send count (size) and data to RM, from file;
    close file;
    get reply from RM;
    display information;
    break;

case RUN_TASK:
    call task_exec;
    break;

case CLOSE_AP:
    send command to RM;
    get reply from RM;
    display information;
    exit;
    break;

default:
    display error message;

} /* end of switch */
} /* end of while */

} /* END OF MAIN */

/*  Task Execution */

/* Task Execution */
task_exec
{
    get program file name from user;
    try to open file;
    if failure, return;
    create transient structures;
    if failure, return;
    send RUN_TASK command to RM;
    count (file size) and program to RM;
    get return from RM;
    if RM returned FAILURE, free allocated structures and return;
    get number of PMs (for term. detect.);
    get ID of 'next PM';
    initialise termination detection mechanism;
    initialise "starting" agent;
    initialise 'task abort by user' detection mechanism;
    /* Execution Loop */
    while (terminated) {
        /* process agent */
        if that's a normal agent: {
            call agent;
            increment agent counter for each agent sent to output (counter += DL size);
        }
        else { /* that's a control agent */
            switch (control type) {
                case TTSK: /* if a termination agent */
                    exit while loop;
                    break;
                case TENQ: /* if a termination enquiry agent */
                    just ignore TENQ agent;
                    break;
                case TECH: /* if a termination echo agent */
                    initialise TD cycle, if necessary;
                    accumulate value brought by TECH agent;
                    if received all echoes
                        if accumulate value == zero
                            assembly and send TTSK agent to RM;
                    else /* still pending agents */

163
cancel termination detection cycle;
break;
}  /* end of switch */
}  /* end of else */
}  /* end of if(task.aghd[LENGTH]) */

start timer (SIGALRM) for start of termination detection;
block until getting an agent from input queue;
cancel timer (SIGALRM);
decrement agent counter if received a normal agent;

}  /* end of while */

de-activate 'task abort by user' detection mechanism;
de-activate 'termination detection' mechanism;
acknowledge termination (send TACK agent) to RM;
free all dynamically allocated structures;
display message to user;

}  /* end of if(task.aghd[LENGTH]) */

start timer (SIGALRM) for start of termination detection;
block until getting an agent from input queue;
cancel timer (SIGALRM);
decrement agent counter if received a normal agent;

}  /* end of switch */

user_abort
/* generates a TTSK agent when CTRL-C is pressed by the user (SIGINT) */
{
    assembly and send TTSK agent to RM;
}

start_td
/* generates a TENQ agent when SIGALRM is received (after program-specified delay) */
{
    block SIGINT, to avoid collision on output pipe;
    assembly and send TENQ agent to RM;
    release SIGINT;
}

exec_agent
/* executes normal agent */
{
    initialise internal variables;

    /* Agent Processing Loop */
    while(not end of processing) {
        if agent was set to stop on non-existent node
        and the 'non-existent node' flag is set
        or if IP points beyond code, set instruction to STOP;
        else
            fetch instruction pointed by IP and increment IP;
            get opcode;
            switch(opcode) {
                case SPWN, STOP, or TERM:
                    exit the while loop;
                    break;
                case SUSP, WAKE, CMD, DTND, INLT, DLLT, DLLT, or LULP:
                    ignore;
                    /* this instruction is invalid in the console node */
                    break;
                case LDSW, STSW, BIFT, BIFF, JMPR, JMPA, JSVR, JSVA, JRET, INDL, or LUST:
                case any arithmetic-logical instruction:
                case SETH, SETL, LDAN, STAN, LDTW, or STTW:
                    execute proper actions;
                    break;
                case LDWPW or STWP:
                    /* Only the first four PW (Persistent Workspace)
                     positions are available:
                     PW[0] == Console Node addr (read-only)
                     PW[1] == Console Node nodekey (read-only)
                     PW[2] == I/O opcode (read/write)
                     PW[3] == I/O data (read/write)


164
Positions 2 and 3 are used for console I/O
execute proper actions;
break;
}
/* end of switch */
} /* end of while */

/* do proper termination procedure */
block SIGINT to avoid collisions on output pipe;
switch(opcode) {
    case TERM:
        assembly and send TTSK agent to RM;
        break;
    case SPWN:
        place agent in the output queue;
        break;
}
release SIGINT;
}

END OF
Processing Module (AP)
END OF
D. Bandwidth Reservation Programs

D.1 Variables

Nodal Persistent Workspaces:

| PW[0]    | NADDR (node address)     |
| PW[1]    | NKEY (node key)         |
| PW[2]    | NEXTNND (stored "next node address") |
| PW[3]    | NBNANDW (stored "bandwidth to reserve") |
| PW[4]    | NLENGTH (stored "length of the path") |
| PW[5]    | SAVEKEY (save location for source node key) |
| PW[6]    | SAVEDST (save location for address of destination node) |
| PW[7]    | ROUTTBHZ (size of routing table, in no of entries) |
| PW[8-..] | ROUTTB (routing table, each entry of size 5 words: ) |
|          | (ROUTTB[0] = Source Node Address ) |
|          | (ROUTTB[1] = Destination Node Address ) |
|          | (ROUTTB[2] = Next Node Address ) |
|          | (ROUTTB[3] = Bandwidth ) |
|          | (ROUTTB[4] = Path Length to Destination ) |

Nodal Link Tables:

store pairs bandwidth-naddr (linkname := bandwidth).

Agent Workspaces:

| AM[0]   | @ZERO (constant: zero)       |
| AM[1]   | @EMPTY (constant: EMPTY)    |
| AM[2]   | @AUX1 (auxiliary var for data transfers) |
| AM[3]   | @AUX2 (auxiliary var for data transfers) |
| AM[4]   | @AUX3 (auxiliary var for data transfers) |
| AM[5]   | @SRC (address of source node) |
| AM[6]   | @DST (address of destination node) |
| AM[7]   | @PREDE (address of predecessor node) |
| AM[8]   | @BANDW (bandwidth to reserve)  |
| AM[9]   | @PATHL (length of the path)  |

Nodal Transient Workspaces:

store I/O messages.

D.2 PM Configuration Files

Contents of the PM Nodal Stores before the execution of Bandwidth Reservation:

<table>
<thead>
<tr>
<th>PM1 File:</th>
<th>PM2 File:</th>
<th>PM3 File:</th>
<th>Field:</th>
</tr>
</thead>
<tbody>
<tr>
<td>4B</td>
<td>4D</td>
<td>63</td>
<td>COUNT</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>PM ID</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>NMNODES</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>LASTNID</td>
</tr>
</tbody>
</table>

166
<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NODE ID</strong></td>
<td><strong>PWS SIZE</strong></td>
<td><strong>LT SIZE</strong></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>00010001</td>
<td>00020001</td>
<td>00030001</td>
</tr>
<tr>
<td>00000061</td>
<td>00000064</td>
<td>00000067</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>80000000</td>
<td>80000000</td>
<td>80000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NODE ID</strong></td>
<td><strong>PWS SIZE</strong></td>
<td><strong>LT SIZE</strong></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>00010002</td>
<td>00020002</td>
<td>00030002</td>
</tr>
<tr>
<td>00000062</td>
<td>00000065</td>
<td>00000068</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>80000000</td>
<td>80000000</td>
<td>80000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NODE ID</strong></td>
<td><strong>PWS SIZE</strong></td>
<td><strong>LT SIZE</strong></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>00010003</td>
<td>00020003</td>
<td>00030003</td>
</tr>
<tr>
<td>00000063</td>
<td>00000066</td>
<td>00000069</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>80000000</td>
<td>80000000</td>
<td>80000000</td>
</tr>
</tbody>
</table>

167
### Contents of the PM Nodal Stores after the execution of Bandwidth Reservation:

<table>
<thead>
<tr>
<th>Field</th>
<th>PM1 File</th>
<th>PM2 File</th>
<th>PM3 File</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE ID</td>
<td>4D</td>
<td>63</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PWS SIZE</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>LT SIZE</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>PWS[0]</td>
<td>00000001</td>
<td>00020001</td>
<td>00030001</td>
<td></td>
</tr>
<tr>
<td>PWS[1]</td>
<td>00000061</td>
<td>00000064</td>
<td>00000067</td>
<td></td>
</tr>
<tr>
<td>PWS[2]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>PWS[3]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>PWS[4]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>PWS[5]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>PWS[6]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>PWS[7]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>PWS[8]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>PWS[9]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>PWS[A]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>PWS[B]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>PWS[C]</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>LT[0].LNAME</td>
<td>00010001</td>
<td>00100001</td>
<td>00100003</td>
<td></td>
</tr>
<tr>
<td>LT[0].NADDR</td>
<td>00000001</td>
<td>00000000</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>LT[1].LNAME</td>
<td>00030002</td>
<td>00030002</td>
<td>00030002</td>
<td></td>
</tr>
<tr>
<td>LT[1].NADDR</td>
<td>00000002</td>
<td>00000002</td>
<td>00000002</td>
<td></td>
</tr>
<tr>
<td>LT[2].LNAME</td>
<td>0000012C</td>
<td>0000012C</td>
<td>0000012C</td>
<td></td>
</tr>
<tr>
<td>LT[2].NADDR</td>
<td>00030003</td>
<td>00030003</td>
<td>00030003</td>
<td></td>
</tr>
<tr>
<td>LT[3].LNAME</td>
<td>80000000</td>
<td>80000000</td>
<td>80000000</td>
<td></td>
</tr>
<tr>
<td>LT[3].NADDR</td>
<td>80000000</td>
<td>80000000</td>
<td>80000000</td>
<td></td>
</tr>
</tbody>
</table>

168
**D.3 Program 1**

* Bandwidth Reservation in a Communication Network - PROGRAM 1

00 ADDI A2 A0 #0  
01 ADDI A3 A0 #3  
02 JSVR CONSOLEIO  
03 LDPW A5 PW[A0+#3]  
04 ADDI A2 A0 #4  
05 ADDI A3 A0 #8  
06 JSVR CONSOLEIO  
07 LDPW A6 PW[A0+#3]  
08 ADDI A2 A0 #9  
09 ADDI A3 A0 #B  
0A JSVR CONSOLEIO  
0B LDPW A8 PW[A0+#3]  
0C INDL A6  
0D SPWN  
0E ADDA A7 A0 A0  
0F ADDA A9 A0 A0  
10 SETH A2 #7FFP  
11 SKTLL A2 #FFFP  
12 STFW A2 PW[A0+#3]  
13 I/LOOP:  
14 LULT A2 A1  
15 EQLA A3 A2 A1  
16 BIFT A3 ENDL LOOP  
17 EQLA A3 A2 A7  
18 INDL A3  
19 JKMP L/LOOP  
1A ENDTL LOOP:  
1B SPWN  
1C ADDI A9 A9 #1  
1D RLTL A2 A7  
1E GRTA A3 A8 A2  
1F BIFT A3 NOCHANGE  
20 ADDA A8 A2 A0  
21 NOCHANGE:  
22 LDPW A2 PW[A0+#3]  
23 GRTA A3 A8 A2  
24 BIFT A3 UPDATE  
25 EQLA A3 A8 A2  
26 LDPW A4 PW[A0+#4]  
27 LSTA A3 A9 A4  
28 BIFT A3 DIE  
29 UPDATE:  
2A STFW A7 PW[A0+#2]  
2B STFW A8 PW[A0+#3]  
2C STFW A9 PW[A0+#4]  
2D LDPW A2 PW[A0+#0]  
2E BIFT A3 L/LOOP  
2F LDPW A2 PW[A0+#1]  
30 SKTL A4 #0F0F  
31 EQLA A3 A2 A4  
32 BIFT A3 DIE  
33 STFW A2 PW[A0+#5]  
34 STFW A4 PW[A0+#1]  
35 STFW A6 PW[A0+#6]  
36 DIE:  

* A2 points to beg_msg  
A3 points to end_msg  
CONSOLEIO:  
37 LDTPW A4 TW[A2+#0]  
38 STFW A4 PW[A0+#3]  
39 ADDI A0 #5  
3A STFW A6 PW[A0+#2]  

* issue command
ADDI A2 A0 #0
GRTA A4 A3 A8
BIFF A4 CONSOLEIO
ADDI A4 A0 #2
STPW A4 PW[A0+#2]
JRET

* increment A2
* Is A2 > end_msg?
* IF NOT, re-start loop
* get command for input HEK
* issue command
* return (data is in PW[3])

$SET TW[0] #0A534F55 "IFSOU"
$SET TW[1] #82434520 "RCEb"
$SET TW[2] #4E4F4453 "MODE"
$SET TW[3] #33200000 "ib"
$SET TW[4] #0A444553 "IÚES"
$SET TW[5] #54494E41 "TINA"
$SET TW[6] #54494F41 "TIAb"
$SET TW[7] #54554E4D "TINA"
$SET TW[8] #453A2000 "E:ib"
$SET TW[9] #0A42414E "IFBAN"
$SET TW[A] #44574944 "DNID"
$SET TW[B] #54483520 "TM:ib"

Bandwidth Reservation in a Communication Network - PROGRAM 1
Object Code in Hexadecimal:

00000056 count $56

00000000 APID #0
00000000 NEXTFHID #0 (dummy)
00000009 TPARSZ #9
00000041 CODESZ #41
000000C CDATASZ #C
00000000 TWSSZ #C
00000001 TDDELAY #1
00000000 RFU #0
00000000 RFU #0

27020000 00 ADDI A2 A0 #0
27030003 01 ADDI A3 A0 #3
0B000034 02 JSVR CONSOLEIO
C0500003 03 LDWF A5 PW[A0+#3]
27020004 04 ADDI A2 A0 #4
27030008 05 ADDI A3 A0 #8
0B000030 06 JSVR CONSOLEIO
C0060003 07 LDWF A6 PW[A0+#3]
27020009 08 ADDI A2 A0 #9
2703000B 09 ADDI A3 A0 #B
0B00002C 0A JSVR CONSOLEIO
C0080003 0B LDWF A8 PW[A0+#3]
10000060 0C INDL A6
00000000 0D SPWN
16070000 0E ADDA A7 A0 A0
16090000 0F ADDA A9 A0 A0
38027FFF 10 SETH A2 #7FFF
39027FFF 11 SETH A2 #7FFF
E000203 12 STPW A2 PW[A0+#3]
13020100 13 LLOOP: LOIT A2 A1
18030201 14 EQLA A3 A2 A1
07030004 15 BIFT A3 ENDLLOOP
18030207 16 EQLA A3 A2 A7
0703FFF8 17 BIFT A3 LLOOP
10000200 18 INDL A2
09FFWFP9 19 JMPL LLOOP
C0070000 1A ENDLLOOP: LDWF A7 PW[A0+#0]
00000000 1B SPWN
27090091 1C ADDA A9 A9 #1
40027000 1D MLLT A2 A7
1D030800 1E GRDA A3 A8 A2
08030001 1F BIFF A3 NOCHANGE
16080200 20 ADDA A8 A2 A0
C0020003 21 NOCHANGE: LDWF A2 PW[A0+#3]
1D030802 22 GRDA A3 A8 A2
07030005 23 BIFT A3 UPDATE
18030802 24 EQLA A3 A8 A2
08030010 25 BIFF A3 DIR
C0040004 26 LDWF A4 PW[A0+#4]
18030904 27 LSTA A3 A9 A4

171
D.4 Program 2

* Bandwidth Reservation in a Communication Network - PROGRAM 2

00 SETH A2 #FFFF
01 INDL A2
02 SPWN
03 SETH A2 #0000
04 LULD A5 A2
05 EQLA A3 A5 A1
06 BIFF A3 DIE
07 INDL A5
08 SPWN
09 LDPW A2 PW[A0+#5]
0A SETH A2 PW[A0+#1]
0B LDPW A6 PW[A0+#6]
0C ADDA A7 A0 A0
0D LDPW A9 PW[A0+#3]
0E LDPW A9 PW[A0+#4]
0F EQLA A3 A8 A0
10 BIFF A3 BUILDPATH
11 INDL A6
12 SPWN
13 JMPR ONDEST

* update Link Table
14 BUILDPATH: LDPW A2 PW[A0+#2]
15 BILT A3 A2
16 DLLT A2 A3
17 SUBA A3 A3 A8

* nextnode <- NEXTND
* bw <- LT(naddr == nextnode addr)
* compute new bw (= old bw - @BANDW)
* create new LT entry bw-naddr

* update Routing Table

* get value of ROUTTBSZ

* increment value of ROUTTBSZ

* stored new value of ROUTTBSZ

* compute start point of new entry: (old ROUTTBSZ) + (entry size) + (ROUTE start)

* Source Node Addr <- SRC

* Destin. Node Addr <- DST

* Next Node Addr <- nextnode

* Bandwidth <- @BANDW

* Path Length <- @PATHL

* decrement @PATHL

* spawn to nextnode

* Is node addr == DST?

If NOT, re-start loop

* NBANDW <- zero

* console node addr <- ST(key == zero)

* DL <- console node addr

* @PRED <- zero

If YES, exit loop

* NeigAddr <- LT(key < any)

* Is NeigAddr == EMPTY?

If YES, re-start loop

* DL <- NeigAddr

* re-start loop

* ©PRED <- current node addr

* @PRED <- current node addr

* get this node address (NADDR)

* Is this node == console node?

If YES, output message

* NLENGTH <- zero?

If YES, die (node already cleaned)

* NEXTND <- zero

* ©BANDW <- zero

* NLENGTH <- zero

* re-start loop

* Is ©BANDW == zero?

If NOT, output success msg

* begin fail_msg

* end of fail_msg

* output failure msg

* stop agent

* begin of succ_msg

* end of succ_msg

* output success msg

* output buffer <- ©BANDW

* get command for output DECIMAL

* issue command

* begin of end_succ_msg

* end of end_succ_msg

* output end of success msg

* A2 points to beg_msg

* A3 points to end_msg

* get msg segment in TW[A2]

* output buffer <- msg segment

* command for output ASCII

* issue command

* increment A2

* Is A2 > end_msg?

If NOT, re-start loop

return

SET TW[0] #0X504154 * "1FPAT"
* Bandwidth Reservation in a Communication Network - PROGRAM 2

* Object Code in Hexadecimal:

```
00000070  count  #70
00000000  APID  #0
00000000  NEKTPID  #0 (dummy)
00000009  TPARSZ  #9
00000038  CODESZ  #58
0000000F  CDATASZ  #F
00000000F  TWSSZ  #F
00000001  TDELAY  #1
00000000  RFU  #0
00000000  RFU  #0

3802FFFF  00  SETH A2 #FFFF
10000200  01  INDL A2
00000000  02  SPWN
380200F0  03  SETH A2 #0F0F
14050200  04  LULD A5 A2
1B030501  05  EQUL A3 A5 A1
07030848  06  BIFT A3 DIZ
10000500  07  INDL A5
00000000  08  SPWN
C0020005  09  LDWP A2 PW(A0+W5)
E0000201  0A  STFW A2 PW(A0+1)
C0060006  0B  LDWP A6 PW(A0+W6)
16070000  0C  ADDA A7 A0 A0
C0080003  0D  LDWP A8 PW(A0+W3)
C0090004  0E  LDWP A9 PW(A0+W4)
1B030000  0F  EQUL A3 A8 A0
08030003  10  BIFE A3 BUILDPATH
10000600  11  INDL A6
00000000  12  SPWN
09000014  13  JNEF ONDEST
C0020002  14  BUILDPATH: LDWP A2 PW(A0+W2)
40030200  15  RLLT A3 A2
12020203  16  DLLT A2 A3
17030308  17  SUBA A3 A3 A8
11000203  18  INLT A3 A3
C0030007  19  LDWP A3 PW(A0+W7)
27040301  1A  ADDA A4 A3 #1
E0000407  1B  STFW A4 PW(A0+W7)
29030305  1C  MULL A3 A3 #5
27030309  1D  ADDA A3 A3 #0
E0030500  1E  STFW A5 PW(A3+W0)
E0030601  1F  STFW A6 PW(A3+W1)
E0030202  20  STFW A2 PW(A3+W2)
E0030803  21  STFW A8 PW(A3+W3)
E0030904  22  STFW A9 PW(A3+W4)
28090901  23  SUBA A9 A9 #1
10000200  24  INDL A2
00000000  25  SPWN
1B030206  26  EQUL A3 A2 A6
0803F8EC  27  BIFE A3 BUILDPATH
E0000003  28  ONDEST: STFW A0 PW(A0+W3)
15020000  29  LUST A2 A0
10000200  2A  INDL A2
```