MODELS, LANGUAGES, AND HEURISTICS FOR DISTRIBUTED COMPUTING*

by

Robert E. Filman** and Daniel P. Friedman

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT No. 124

MODELS, LANGUAGES, AND HEURISTICS

FOR DISTRIBUTED COMPUTING

ROBERT E. FILMAN AND DANIEL P. FRIEDMAN

MARCH 1982

*To appear in AFIPS '82.
Research reported herein was supported in part by the National Science Foundation under grants MCS77-22325, 79-04183 & 81-02291.

**Author's present address: Hewlett Packard, Palo Alto, CA.

MODELS, LANGUAGES, AND HEURISTICS FOR DISTRIBUTED COMPUTING*

by ROBERT E. FILMAN
Hewlett Packard
Palo Alto, California
and
Indiana University
Bloomington, Indiana

and

DANIEL P. FRIEDMAN
Indiana University
Bloomington, Indiana

ABSTRACT

We are interested in the issues surrounding computer problem solving in systems of loosely-coupled processes. This paper is a compendium of ideas related to the software issues involved in programming distributed systems. We discuss two aspects of this problem: languages and models for distribution, and heuristics for organizing distributed systems. The second section of the paper discusses the nature of distributed languages and models, and presents a comparison of the attributes of several of the major proposals for distributed computing. The third section is a discussion of some heuristics for the organization of multiple-process systems.

INTRODUCTION

The marvels of miniaturized silicon are leading to a world of cheap microprocessors. These microcomputers bring with them the hope of faster and cheaper versions of the conventional, mainframe computer—an army of small automata, eager to increment and loop, ready to go out and solve our computing problems. However, as any manager of a large software project can assure you, a large collection of dumb computing agents does not add up to a working system. Microprocessors need to be told not only what to do, but how to do it. They need to cooperate and communicate in their processing task; but their cooperation must not turn into a bureaucracy, expending more energy on communication than on production.

The next generation of computer architectures will provide users with not just one but many computers to perform their tasks. But improved computational productivity is not achieved by processing power alone. Along with multiple-processor architectures must come the software facility to profitably use that computing power. We call such an integrated, multiple-processor system a *coordinated computing* system. The integration we require in coordinated computing is not merely interprocessor communication, but interprocess cooperation.

This paper discusses the issues surrounding computer problem solving in systems of loosely coupled processes. Toward this end, we have been studying models of communication, programming languages for distributed computing, and heuristics for system organization. In this paper we present a compendium of ideas related to the software issues involved in achieving coordinated computing. We discuss two aspects of the coordinated computing task: languages and models for distribution and heuristics for organizing distributed systems.

The second section of the paper discusses the nature of distributed languages and models and compares the attributes of several of the major proposals for distributed computing. The third section discusses some heuristics for the organization of multiple-process systems. These issues are discussed in greater detail in *Coordinated Computing: Tools and Techniques for Distributed Software.*[1]

MODELS AND LANGUAGES

Requirements for Distribution

Every programming language or model makes assumptions about its computing environment. A programming system for coordinated computing is no exception. Particularly important are the assumptions that such a system makes about the interface it provides the programmer:

1. First, a distributed programming system has multiple, independent, concurrently computing processes or supports some activity that corresponds to doing many independent activities simultaneously.
2. The processing elements must communicate (transfer information) between themselves, though there is a cost (time delay) associated with this transfer. A system that treats interprocess communication as if it were free is a shared-memory, or tightly coupled, system. Such systems avoid many of the difficulties of distribution.
3. No process should be able to perceive the global state or global time of the system. Once again, a system that shares too much information is not truly distributed.
4. Communication should be directed: communications should have both a specific origin and a specific destination. This implies that (pseudo-) broadcast communication mechanisms are somewhat suspect. If a system wishes to use *broadcast* as a syntactic shorthand for a series of directed communications, then the cost of that broadcast should be proportional to the number of destinations.
5. If a system has an explicit notion of process, programs written in that system should be able to create processes dynamically.

These criteria are software criteria, though they have their origins in the nature of physical computing devices. Our goal is to define a distributed software system to be the programming equivalent of a multiple-processor hardware system, where the processors, though independent, share the work involved in some task. These restrictions are designed to limit discussion to languages and models that can support coordinated problem solving on such systems.

* To appear in AFIPS '82.

There are no mature software systems that exhibit our idea of distribution. Instead, several languages and models have been proposed for dealing with various aspects of distribution. We feel that these languages and models can be characterized by the choices they make in a multidimensional decision space. In this section we discuss the dimensions of that space and plot the location of our sample languages.

Candidate Models and Languages

Models are used to describe mathematical relationships. Programming languages are used to describe processing. Since the mathematical relationships involved in programming and modeling a processing task are similar, some of our examples have aspects of both programming language and model.

Models are constructed to explain and analyze complex system behavior. It is important that a model abstract out the "interesting" part of the system it is attempting to model. Models of concurrent systems usually specify some properties of the interprocess communication mechanism; they are then used to prove properties of the resulting systems. For example, one can set up a model of the communication relationships among processes and use it to analyze the efficiency of an algorithm; or one can set up a model of the information transfer among processes and use it to prove an algorithm's correctness.

Informally, a computer language is a way of providing a sufficiently exact set of directions to a computer. Computer languages are characterized by their syntax and semantics. The syntax of a programming language defines the appearance of the set of legal program strings. The semantics of a programming language specifies the effects of a particular syntactic structure. The details of a programming language syntax (such as the choices of keywords and punctuation) are unimportant (except for issues of human engineering). Instead, it is the semantic actions the programming language can take that interest us.

There have been many proposals for models and languages for distributed processing. This paper contains a brief comparison of 11 of these proposals. We have selected what we feel is a representative sample of ideas from the important systems. Of these systems, we characterize four of these as pure models: Milne and Milner's "concurrent processes,"[2] Fitzwater and Zave's "exchange functions,"[3] and Lynch and Fischer's "shared variable" model[4] and "data flow," by Dennis and by Arvind et al.[5,6]* Typically, models for distribution describe only the communication relationships between processes, without placing limitations on the architecture of the remainder of a system.

*Many workers have worked on data flow systems; there are important differences between the models they have developed. Here we cite only a pair of references. The semantics of data flow models depends on which data flow model is used. Later comparisons will develop this theme.

Three of our examples are model-language hybrids: Hewitt's Actors,[7] Friedman and Wise's frons,[8] and Hoare's Communicating Sequential Processes.[9] Hybrids specify more of the computing process than models but are not as comprehensive as languages.

Our final four examples are programming languages: Brinch Hansen's Distributed Processes,[10] Feldman's PLITS,[11] Andrew's Synchronizing Resources,[12] and the Defense Department's Ada.[13]

Dimensions of Distributed Languages and Models

Every designer of a model or language for distributed computing chooses the facilities that that system will provide. In this section we identify several such dimensions and indicate where each model and language lies in the choice space.** Table I examines the general goals and structure of each system, Table II examines aspects of intrasystem communication, and Table III examines the system perspective of the remaining dimensions.

**Other papers that have engaged in comparative discussion of distributed languages include Mohan,[14] Rao,[15] and a predecessor of this paper.[16]

1. Task domain: the most dramatic differences between these languages and models appears in their choice of problem domain. The models are directed principally at mathematical concerns, such as proofs of algorithm correctness (Correctness) and analysis of algorithmic complexity (Analysis). Some of these systems are concerned with issues of systems implementation (Systems). Some of the language proposals are pragmatic—their authors feel that the choice of constructs eases the task of programming distributed systems (Pragmatics). Two of our pragmatic systems have particular interest in programming problems from artificial intelligence (AI); one is principally concerned with the software engineering problem of requirement specification (RS). Many systems have features directed at several of these task domains.

2. Processes: Most models and languages have an explicit process entity (Process). Others view tasks as the creatures of program execution, to be solved by the system as a whole, without keeping the notion of explicit, communicating processes (Tasks).

3. Dynamics: In any system the set of processes can either be statically determined at system generation (Static), or dynamically created during system execution (Dynamic). All of the systems studied that have dynamic process creation can allocate new processes; one can also generate them by the recursive, lexical expansion of program text (Lexical). Task-based systems can, of course, dynamically create new tasks.

Most languages that allow dynamic process creation restrict the new processes to a type determinable at system initiation (compilation). Some systems allow the creation of new varieties of processes and tasks during execution (new). A system that creates new processes invariably provides names for (or, equivalently, pointers to) these new processes. These names can be passed around the process network, creating new communication channels. Systems that rely on a static network of processes usually determine interprocess communication paths lexically.

4. Synchronization: Systems with explicit processes choose between synchronous communication, where all communicants must attend to every communication (Synch) and asynchronous communication, where processes can begin a communication and continue with other activities (Asynch). This is the difference between "call" and "send-and-forget." Following Ada,[13] we call the period of synchronization in communication *rendezvous*.

5. Buffering: A system that supports asynchronous communication can place a bound (Bounded) on the size of the communication buffer or can allow an unbounded (Unbounded) number of messages to be initiated. Systems that require synchronous communication have no use for unbounded message buffers; every message is processed when it is sent.

6. Information Flow: In communication, information flow can be unidirectional (Uni) (from one process to another), bidirectional simultaneous (between processes only at the synchronous time of communication) (Bi-Sim), or bidirectional delayed (where one process can compute a reply during rendezvous) (Bi-Del). None of our models or language provides for bidirectional delayed communication where both processes compute during rendezvous, though there is no theoretical reason to disallow it.

7. Control: The communication process can be initiated by an active caller to a passive receiver (Act-Pas), by an active caller to an active receiver (Act-Act), or by two equal communicators (Equal). The receiver of a request sometimes has control over the order in which the requests are processed. In the languages and models studied, this control includes input and/or output guards (I/O-G), time out on lack of response (Tm-O), choice of certain classes of requests (Choice), pattern matching on messages (Pat), and selective search and examination of all pending messages (Ex-Sr).

8. Connection: Communicants can refer to a named port or channel that is external to all processes (Port), the name of process itself (Name), or a port within a called process (Entry). The chart details the naming required of both the message sender and the message receiver for systems that do not treat communicators equivalently.

TABLE I—Goals and structures

Model	(A) Task domain	(B) Explicit Processes	(C) Dynamic Process Creation
Concurrent processes, Milne & Milner	Correctness	Processes	Dynamic, new
Exchange functions, Fitzwater & Zave	Pragmatics (RS)	Processes	Static
Shared variable, Lynch & Fischer	Correctness, Analysis	Processes	Static
data flow, Dennis; Arvind et al.	Pragmatics	Tasks	
Actors, Hewitt	Pragmatics (AI), Correctness	Processes	Dynamic, new
frons, Friedman & Wise	Pragmatics, Correctness	Tasks	
CSP, Hoare	Systems, Correctness	Processes	Static
Distributed processes, Brinch Hansen	Systems	Processes	Static
PLITS, Feldman	Pragmatics (AI), Systems	Processes	Dynamic, new
Synchronizing resources, Andrews	Systems	Processes	Static
Ada, DoD	Systems, Pragmatics	Processes	Dynamic, new, lexical

9. Time and consciousness: In synchronous communication, the process that initiates a communication may be forced to complete that communication, or it may have some facility for aborting the communication (such as a time out). We say that a process activity that causes uninterruptable waiting is a loss of consciousness for a process. Languages and models sometimes provide mechanisms by which a calling process can regain control. These mechanisms include instantaneous time outs, time outs, input and output guards (I/O-Guards), input guards (I-Guards), and rescindable offers. A process that never loses consciousness is always conscious; a process that only awakens when invoked is reactive.

10. Fairness: Systems can strive for fairness. Sometimes this notion of fairness is a weak fairness, the idea that every attempted action eventually gets its turn (Weak). Alternatively, a system can specify a stronger notion of fairness, asserting that each process will get its "rightful" turn (Strong). Stronger fairness can usually be implemented with queues. On the other hand, a formalism may make no claims about fairness at all (Anti).

11. Failure: Most models and languages treat processes as perfect computers and communication as invariably secure (Fail). Some of the systems have some mechanisms for dealing with process and communication failure.

12. Shared Memory: Some of the systems have explicit shared-memory mechanisms, in addition to distributed ones.

TABLE II—Communication

Model	(D) Synchronization	(E) Buffering	(F) Information flow	(G) Control	(H) Syntactic-Conn. Sender	Receiver
Concurrent Processes	Synch	Bounded	Bi-Sim	Equal	Port	Port
Exchange Functions	Synch	Bounded	Bi-Sim	Equal	Port	Port
Shared variable	Asynch	Bounded	Uni	Equal	Port	Port
data flow[a]	Asynch	Bounded/ Unbounded	Uni	Act-Pas	Entry	none
Actors	Asynch	Unbounded	Uni	Act-Pas	Name	none
frons	Asynch	n.a.	Uni	n.a.	Port	Port
CSP	Synch	Bounded	Uni	Act-Act I/O-G	Name	Name pattern—match
Distributed processes	Synch	Bounded	Bi-Del	Act-Act I-G, Pat	Entry	none
PLITS	Asynch	Unbounded	Uni	Act-Act[b] Filter	Name	sender filter
Synchronizing Resources[c]	Asynch/ Synch	Unbounded/ Bounded	Bi-Del Uni	Act-Act[d] I-G, Ex-Sr	Entry	Entry
Ada	Synch	Bounded	Bi-Del	Act-Act I-G, Tm-O	Entry	Entry

[a]Different data flow models make different choices regarding infinite buffering.
[b]PLITS processes can filter messages by sender or "transaction key."
[c]Synchronizing resources supports both synchronous (call) and asynchronous (send) mechanisms.
[d]Synchronizing resources can examine and sort calls before selecting which to serve.

TABLE III—Other Issues

Model	(I) Time & Consciousness	(J) Fairness	(K) Failure	(L) Supports Shared Memory
Concur. Proc.	Rescindable Offer	Anti		
Exch. Funct.	Instantaneous Time Outs	Strong		
Shared Var.	Always Conscious	Weak		Supports
data flow	Reactive	-n.a.-[a] /Anti		
Actors	Reactive	Weak		
frons	Reactive	Weak	Convenient redundancy	
CSP	I/O Guards[b]	Anti		
Dist. Proc.	I-Guards	Strong		
PLITS	Always Conscious	Strong		
Synch. Res.	I-Guards	Anti		Supports
Ada	I-Guard, Time Outs	Strong	Extensive mechanisms	Supports

[a]Some data flow models are deterministic. In a deterministic system, fairness is irrelevant.

[b]Hoare's earliest proposals excluded output guards. Later works on CSP have included them.

HEURISTICS FOR COORDINATION

Incremental Computation

Despite the paucity of failure mechanisms in our models and languages, any real system needs mechanisms to cope with failures. On the statement level, these mechanisms need to handle the disruptions of lost messages and failing processes. On a more global level, a profitable organization of a distributed system may be performed, not as a sequential program, but as a set of computing "agents" who make progress toward solving subtasks.

The idea of useful progress may seem a foreign notion. Most conventional programming languages are a-step-at-a-time, imperative formulations. The validity of the successive steps is entirely dependent on the successful completion of the previous steps. However, there are other possible formulations for expressing computable functions. Production systems (such as Newell's[17]) are one example. A production system consists of two parts: a working memory and a set of productions. Each production has two pieces, a *pattern* and an *action*. When some part of the working memory matches the pattern of a particular production, that production fires, executing its action. Actions are programs; they typically add elements to the working memory. Elements are never removed from the working memory. Thus, the firing of a production never makes another production's firing cease to be valid. One possible organization of a distributed system is a set of productions that communicate through a working memory. The "blackboard" of the Hearsay-II model[18] is an example of such a central communication depository.

Other examples of computing systems that make progress include theorem proving[19] and suspending evaluation in Lisp-like systems.[8] In a theorem-proving system, the proof of a theorem does not invalidate the truth of any other theorem. A task expressed as a theorem to be proved can be worked on by many inference rules at the same time. In a suspending CONS system, tasks are created as the natural action of computation. When a processing element finishes a task, it "stings" that task with its value.[8] Sting is an interlock-free test-and-set primitive. If the particular object to be stung has already been stung (by someone else), the operation becomes a "no-op." Thus, if a swarm of processes are working on a problem, the first sting of the answer is permitted to succeed. Later stings have no effect; thus, the system exhibits functional behavior throughout.

Programming languages predicated on this idea of incremental discovery can be more easily distributed than systems that require the standard sequentiality.

Economic Models

Distributed computing networks are not the only organizations that require internal cooperation and communication. Human economic activity shows both some of the same requirements and some of the same goals as a distributed computing network. There are some interesting parallels between human economic systems and potential organizational models for distributed systems.

How are economies organized? One important dimension is centralization. In a centralized system, there is a master directorate (node) that sets the goals of the system and divides the task into subpieces, with each subtask specified for a particular worker. When the task is modular and well defined, it is possible to organize a distributed system in this fashion. Efficiency can be achieved in such a structure if the task is well understood, and the initial allocation of subgoals and resources can be made to reflect this understanding. However, central planning does not lend itself well to ill-defined problems. Additionally, there may be a communication overload from the planning node to the workers while most of the communication ability of the system—between the working nodes—goes unused.

It is only a small step from a fully centralized economy to a partially centralized (hierarchical) model. The central authority defines the major tasks. These are parceled out to regional subauthorities, each of whom is allotted a resource of workers. This structure can be iterated. At the limit, it resembles a corporate hierarchy tree. Hierarchical organization can respond well to local aberrations. However, its response to dramatic global changes is somewhat slower, because command must filter through several command layers. Hierarchical systems are better matched to the physical distribution of the world than systems based on pure centralized control.

An alternate approach to processor organization is a laissez-faire economy. Each task has certain goals and an allocation of currency. Currency can be used to purchase processor power and to generate new tasks. When a task has exhausted its currency, it can appeal to its own source (banker) for more. Its banker can then decide, on the basis of the results that the task presents, whether to grant that task more resources. The scheme can be applied recursively, to the banker's banker, and so forth, back to the resources of the human being who originated the request. Such a scheme lends itself to ill-defined tasks—ones where a promising line can be recognized but not necessarily generated—and to useful-progress programming models. Though such systems are not fragile, there are difficulties both in focusing the organization in the presence of a rapidly changing environment and in terminating the activity of tasks that have ceased to be useful. A variation of this mechanism was used by the agenda priority schemes of Lenat.[20] Our development parallels a remark of Hewitt.

One could also imagine distributed systems organized as mixed economies (partially centralized and partially free market) or as indicative planned systems (with centralized goals and directives shaping a free-market economy.)

Another proposal for organizing distributed computing is Smith's contract nets.[21] Processes that have subproblems broadcast their request to the other processes. A free process, or one that has particular knowledge about that task, "bids" to obtain the contract. Contract nets are a protocol; Smith does not elaborate on how particular tasks would be organized in the contract net approach.

CONCLUSIONS

Distribution promises inexpensive and efficient computation. To realize that promise, much work needs to be done both to define the right models for distribution and to select the appropriate algorithms for apportioning computations among processes.

ACKNOWLEDGMENTS

We thank John Barnden, Jim Burns, Mitch Wand, and Dave Wise for comments on earlier drafts of this paper. Research reported herein was supported (in part) by the National Science Foundation under grants numbered MCS77–22325, MCS79–04183, and MCS81–02291.

REFERENCES

1. Filman, R. E., and D. P. Friedman. Coordinated Computing: Tools and Techniques for Distributed Software. (To appear, 1982).
2. Milne, G., and R. Milner. "Concurrent Processes and Their Syntax." *Journal of the ACM*, 26 (1979), pp. 302–321.
3. Fitzwater, D. R., and P. Zave. "The Use of Formal Asynchronous Process Specifications in a System Development Process." *Proceedings of the Sixth Texas Conference on Computing Systems*, 1977, pp. 2B–21:2B–30.
4. Lynch, N. and M. Fischer. "On Describing the Behavior and Implementation of Distributed Systems." In G. Kahn (ed.), *Proceedings of International Symposium on Semantics of Concurrent Computation*. Berlin: Springer, 1979, pp. 147–171.
5. Dennis, J. B. "First Version of a Data Flow Procedure Language." In B. Robinet (ed.), *Programming Symposium*. Berlin: Springer, 1974, pp. 362–376.
6. Arvind, K. Gostelow, and K. Plouffe, "The (Preliminary) ID Report: an Asynchronous Programming Language and Computing Machine." Technical Report 114, Department of Information and Computer Science, University of California, Irvine, May 1978.
7. Hewitt, C., G. Attardi, and H. Lieberman. "Security and Modularity in Message Passing." *Proceedings of the First International Conference on Distributed Systems*, 1979, pp. 347–358.
8. Friedman, D. P., and D. S. Wise. "An Approach to Fair Applicative Multiprogramming." In G. Kahn (ed.), *Proceedings of International Symposium on Semantics of Concurrent Computation*. Berlin: Springer, 1979, pp. 203–225.
9. Hoare, C. A. R. "Communicating Sequential Processes." *Communications of the ACM*, 21 (1978), pp. 666–677.
10. Brinch Hansen, P. "Distributed Processes: a Concurrent Programming Concept." *Communications of the ACM*, 21 (1978), pp. 934–941.
11. Feldman, J. A. "High Level Programming for Distributed Computation." *Communications of the ACM*, 22 (1979), pp. 353–367.
12. Andrews, G. "Synchronizing Resources." *ACM Transactions on Programming Languages and Systems*, 3 (1981), pp. 405–430.
13. Department of Defense. "Military Standard Ada Programming Language." Dept. of Defense Standard MIL-STD-1815, December 1980.
14. Mohan, C. "A Perspective of Distributed Computing: Models, Languages, Issues & Applications." Working Paper DSG-8001, Department of Computer Science, University of Texas, March 1980.
15. Rao, R. "Design and Evaluation of Distributed Communication Primitives." Technical Report 80-04-01, Department of Computer Science, University of Washington, April 1980.
16. Filman, R. E., and D. P. Friedman. "Inspiring Distribution in Distributed Computing." *Proceedings of ACM SIGOPS/SIGPLAN Workshop on Fundamental Issues in Distributed Computing*, 1980.
17. Newell, A. "Production Systems: Models of Control Structures." In W. Chase (ed.), *Visual Information Processing*. New York: Academic Press, 1972, pp. 463–526.
18. Erman, L. D., F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty." *ACM Computing Surveys*, 12 (1980), pp. 213–254.
19. Loveland, D. W. *Automated Theorem Proving*. Amsterdam: North Holland, 1978.
20. Lenat, D. B. "Automated Theory Formation in Mathematics." *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, 1977, pp. 832–842.
21. Smith, R. "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver." *Proceedings of the First International Conference on Distributed Systems*, 1979, pp. 185–192.