NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

Enterprise Management Network Architecture Distributed Knowledge Base Support

Michel Roboam, Mark S. Fox and Katia Sycara

CMU-RI-TR-90-212

Center for Integrated Manufacturing Decision Systems
The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

November 1990

© 1990 Carnegie Mellon University

Michel Roboam is currently visiting scientist in the Center for Integrated Manufacturing Decision Systems and is sponsored by the AEROSPATIALE Company (France).

This research has been supported, in part, by the Defense Advance Research Projects Agency under contract #F30602-88-C-0001, and in part by grants from McDonnell Aircraft Company and Digital Equipment Corporation.

Table of Contents

1. Introduction 1
2. Distributed Systems Definition 4
  2.1 Distributed Systems Advantages 4
  2.2 Decentralized Systems top-level description 4
  2.3 Distributed System Dimensions 5
    2.3.1 Parallel Distributed Processing Systems 5
    2.3.2 Distributed Problem Solving Systems Definition 6
  2.4 Distributed Systems capabilities 6
  2.5 Distributed Systems Problems 7
3. Enterprise Management Network Node 8
4. Network Layer 12
  4.1 Introduction 12
  4.2 Network Specification 13
  4.3 EMN-node specification 13
    4.3.1 Schemata supporting EMN-nodes initialization 13
    4.3.2 Functions supporting EMN-nodes initialization 16
    4.3.3 Example of EMN-node initialization 16
  4.4 Communication Procedures 19
    4.4.1 Schemata supporting the communication procedures 19
    4.4.2 Functions supporting the communication procedures 20
    4.4.3 Example of communication function implementation 21
      4.4.3.1 Message passing without blocking 22
      4.4.3.2 Message reception 24
      4.4.3.3 Message passing with blocking 25
  4.5 Network Layer example 29
5. Data Layer 30
  5.1 Introduction 30
  5.2 Schemata manipulated 31
    5.2.1 The information schema 32
    5.2.2 The message schema 33
    5.2.3 The answer schema 38
    5.2.4 The communication schema 39
  5.3 Information consistency checking primitives 44
  5.4 Query language 46
    5.4.1 Complete schema request 47
    5.4.2 Simple retrieval 47
    5.4.3 Qualified retrieval 47
    5.4.4 Retrieval with ordering 50
    5.4.5 Retrieval from more than one schema 52
    5.4.6 Retrieval involving queries within queries 53
    5.4.7 Locking and unlocking mechanism 54
  5.5 Data Layer example 55
6. Information Layer 56
  6.1 Introduction 56
  6.2 Access privilege granting 57
  6.3 Automatic information acquisition 59
  6.4 Automatic information management 61
  6.5 Information Layer example 62
  6.6 Information Layer Implementation 63
    6.6.1 Information distribution 64
    6.6.2 The message generation sequence 70
    6.6.3 The updating activity 73
    6.6.4 The message and answer reception sequence 76

UNIVERSITY LIBRARIES
CARNEGIE MELLON UNIVERSITY
PITTSBURGH, PA 15213-3890

6.6.4.1 The message reception sequence 78
6.6.4.2 The answer reception sequence 79
6.7 Information Layer utilization example 81
7. Conclusion 83
Acknowledgement 85
References 86

U
CARN
PITT

List of Figures

Figure 3-1: Example of decentralized system 8
Figure 3-2: The elements of an EMN-node 9
Figure 3-3: Information exchanges overview 10
Figure 3-4: Decentralized system example 11
Figure 4-1: Message passing algorithm 22
Figure 4-2: Message passing steps 23
Figure 4-3: Checking mail box algorithm 24
Figure 4-4: Message reception steps 25
Figure 4-5: Message passing with blocking: step 1 26
Figure 4-6: Message passing with blocking: step 2 27
Figure 4-7: Message passing with blocking: step 3 27
Figure 4-8: Message passing with blocking: step 4 28
Figure 4-9: Network Layer implementation example 29
Figure 5-1: Query elements 30
Figure 5-2: Object flow representation 32
Figure 5-3: Decentralized Knowledge Craft running 40
Figure 5-4: Mutual table consistency checking 45
Figure 5-5: Information flows representation 46
Figure 5-6: Data Layer implementation example 55
Figure 6-1: Information Layer implementation example 63
Figure 6-2: Information distribution sequence 65
Figure 6-3: Hierarchical distribution example 66
Figure 6-4: Distribution sequence for an EMN-node initialization 67
Figure 6-5: Distribution message reception 69
Figure 6-6: Message generation sequence 70
Figure 6-7: Updating message generation sequence 75
Figure 6-8: Message and answer reception sequence 77
Figure 6-9: Answer control sequence 80

List of Schemata

Schema 4-1: Network 13
Schema 4-2: DKC-System 14
Schema 4-3: DKC-Channel 15
Schema 4-4: local-DKC-Channel 15
Schema 4-5: Agent-1-DKC-System 17
Schema 4-6: agent-1-DKC-local-Channel 17
Schema 4-7: agent-2-DKC-Channel 18
Schema 4-8: Agent-1-DKC-System 18
Schema 4-9: DKC-message 19
Schema 4-10: DKC-queued-message 20
Schema 5-1: Information 33
Schema 5-2: Message 34
Schema 5-3: Information-search-message 35
Schema 5-4: Updating-message 36
Schema 5-5: LC-distribution-message 36
Schema 5-6: UT-distribution-message 37
Schema 5-7: distribution-END-message 37
Schema 5-8: Answer 38
Schema 5-9: Answer-example 39
Schema 5-10: Communication 41
Schema 5-11: Communication 42
Schema 6-1: CLASS 64
Schema 6-2: NO-SCHEMA-SPEC 71
Schema 6-3: NO-SLOT-SPEC 72
Schema 6-4: NO-VALUE-SPEC 73
Schema 6-5: Answer-message 78
Schema 6-6: Message-queue 78
Schema 6-7: New-message 78
Schema 6-8: Control-answer 79
Schema 6-9: Answer-queue 79
Schema 6-10: New-answer 81

Abstract

Achieving manufacturing efficiency requires that many groups that comprise a manufacturing enterprise, such as design, planning, production, distribution, field service, accounting, sales and marketing, cooperate in order to achieve their common goal. In this paper we introduce the concept of Enterprise Management Network (EMN) as the element to facilitate the integration of distributed heterogeneous functions of a manufacturing enterprise. The integration is supported by having the network first play a more active role in the accessing and communication of information, and second provide the appropriate protocols for the distribution, coordination, and negotiation of tasks and outcomes. The EMN is divided into six layers: Network Layer, Data Layer, Information Layer, Organization Layer, Coordination Layer, and Market Layer. Each of these layers provides a portion of the elements, functions and protocols to allow the integration of a manufacturing enterprise.

1. Introduction

This report presents the architecture, the elements and the organization of an Enterprise Management Network (E.M.N.) to support the integration of the manufacturing enterprise. The optimization of the manufacturing enterprise can only be achieved by greater integration of activities throughout the production life cycle. Integration must not only address the issues of shared information and communication, but how to coordinate decisions and activities throughout the firm.

Achieving manufacturing efficiency requires that the many groups that comprise a manufacturing enterprise, such as design, planning, production, distribution, field service, accounting, sales and marketing, cooperate in order to achieve their common goal. Cooperation can take many forms:

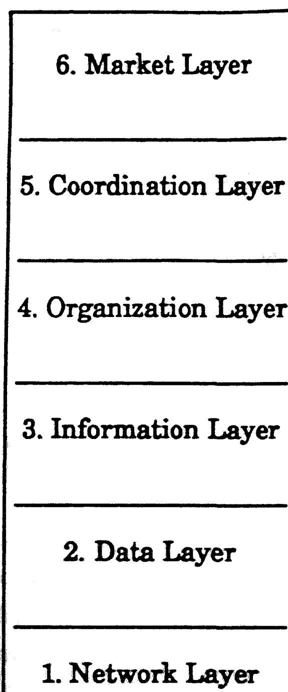
- **Communication of information** relevant to one or more groups' tasks. For example, sales informing marketing of customer requirements, or production informing the controller of production performances.
- **Feedback on the performance** of a group's task. For example, field service informing design and manufacturing of the operating performance of a new product.
- **Monitoring and controlling** activities. For example, controlling the execution of operations on the factory floor.
- **Assignment** of new tasks. For example, a new product manager signing up production facilities to produce a new product.
- **Joint decision making** where groups of "agents" have to negotiate and cooperate in order to achieve their task (which can be antagonistic or not). For example, an inventory manager and a scheduler negotiating to define the manufacturing activity.

An Enterprise Management Network is viewed as the "nervous system" of the enterprise, enabling the functions described above. It is more than a network protocol (e.g., MAP) in that it operates and participates at the application level. Its philosophy is different in terms of participation and structuring. Such a system must be defined in such a generic way that it can be integrated with all kinds of applications an enterprise can use. The following describes the capabilities provided by the Enterprise Management Network Architecture:

- **Information routing**: given a representation for information to be placed on the network and a representation of the goals and information needs of groups on the network, the information routing capability is able to provide the following:
  - Static routing: transfering information to groups where the sender and the receivers are pre-defined.
  - Dynamic routing: transfering information to groups which appear to be interested in the information. This is accomplished by matching a group's goals and information needs to the information packet.
  - Retrospective routing: reviewing old information packets to see if they match new goals and information requirements specified by a group.
- **Closed loop system**: Often, the communication of information results in some activity, which the initiator of the communication may be interested in. The EMN will support the providing of feedback in two modes:
  - Pre-define feedback: operationalizes pre-defined information flows between groups in the organization. For example, production providing feedback to sales on the receipt of orders.
  - Novel feedback: Providing feedback for new and novel messages.

- **Command and control**: Given a model of the firm which includes personnel, departments, resources, goals, constraints, authority and responsibility relations, the EMN will support these lines of authority and responsibility in the assignment, execution and monitoring of goals and activities.
- **Dynamic task distribution**: Supporting the creation of new organizational groups and decomposition, assignment and integration of new goals and tasks, contracting and negotiation are examples of techniques to be supported.

The design of the Enterprise Management Network is divided into six levels:



The *Network Layer* provides for the definition of the network architecture. At this level, the nodes are named and declared to be part of the network. Message sending (or message passing) between nodes is supported along with synchronization primitives (such as "blocking"). Security mechanisms are also provided such as message destination recognition.

The *Data Layer* provides for queries and responses to occur between nodes in a formal query language patterned after SQL [6, 7].

The *Information Layer* provides "invisible" access to information spread throughout the EMN. The goal is to make information located anywhere in the network locally accessible without having the programs executed locally know where in the network the information is located nor explicitly request its retrieval. This Layer also includes information distribution focussed on data classes, keywords and content and security mechanisms such as agent blocking and unblocking and schemata locking and unlocking. All the information queries expressed at this layer use the query language defined at the data layer.

The *Organization Layer* provides the primitives and elements (such as goal, role, responsibility and authority) for distributed problem solving. It allows automatic communication of information

based upon the roles a node plays in the organization. Each EMN-node knows its responsibility, its goals, and its role in the enterprise organization.

The *Coordination Layer* provides the protocol for coordinating the activities of the EMN-nodes through negotiation and cooperation mechanisms.

The *Market Layer* provides the protocol for coordination among organization in a market environment. It supports the distribution of tasks and the negotiation of change and the strategies to deal with the environment.

In this report, we present in details the three first layers of this architecture (Network, Data and Information) which define the distributed knowledge base management [23, 1] supported by the EMN architecture. In the next report [33], we will present the problems of distributed problem solving and how they are covered and supported by the IN architecture.

The purpose of this architecture is to support, through the three first layers, distributed knowledge base and, through the three upper levels, distributed problem solving. Distributed systems have advantages but also inconveniences. Their characteristics are defined in terms of coupling and grain size. Our architecture must be able to support the different types of distributed systems we present in section 2.

In the next section, we focus our attention on the content of an Enterprise Management Network node (EMN-node). We describe its content and characteristics. Then, each of the three first layers of the EMN architecture is described in turn. The actual implementation of this system is presented in [34].

2. Distributed Systems Definition

The Enterprise Management Network Architecture provides the elements and functions to define, implement and support a **distributed system**. A distributed system is a system with many processing and many storage devices, connected together by a network.

2.1 Distributed Systems Advantages

Potentially, this makes a distributed system more powerful than a conventional, centralized one in two ways:

- First, it can be more reliable. Every function can be replicated several times. When a processor fails, another can take over the work. Each file can be stored on several disks, so a disk crash does not destroy any information. We call this property **fault tolerance**.
- Second, a distributed system can do more in the same amount of time, because many computations can be carried out in **parallel**[1].

We will say more about these advantages below.

2.2 Decentralized Systems top-level description

"In a very general terms, a system is said to be distributed when it includes several geographically distinct components cooperating in order to achieve a common distributed task" [2]. But this definition is not true for all the domains. If we consider, for example, games involving two players, the aim of each one is to win the game. So the two agents of this decentralized system do not cooperate, they compete (they cooperate in playing the game, i.e., they follow some rules, but they compete about sub-goals-winning).

The set of nodes in the system is usually organized according to various domain dependent topologies. Decentralized systems in every day life come from a wide variety of areas, e.g., a business firm, a system for traffic control, etc.

The processing nodes in a decentralized system may all be identical in their capabilities or they may each possess specific skills. Whatever the configuration is, in a decentralized system both the control (process) and the knowledge can be distributed throughout the system.

In actuality, there is a range of approaches for decentralized architecture, from an almost centralized system to a distributed system with a centralized planning and control element, to a distributed system with a distributed, hierarchical group of control elements, to a fully distributed, "flat" system in which each element is responsible for its own control.

Moreover, the organization amongst the elements may either be static, remaining the same as time elapses, or dynamic, adapting itself as the requirements of the environment needs it. In any case, the processing nodes, or agents, contain knowledge about themselves and their environment, and a logical capability to work on that knowledge. In other words, the agents have a memory and a processor.

[1]Note we are talking about large grain parallelisms not connection machine style parallelism.

But we have a limitation for the memory aspect: we cannot have in a decentralized agent all the needed information for completely autonomous running (the concept of bounded rationality [35]). This means that we must acquire some information from the other agents of the decentralized system: the agent must **communicate**. Bounded rationality implies that both the information a computing agent can absorb and the detail of control it may handle are limited.

2.3 Distributed System Dimensions

Since almost any real world system is decentralized and, moreover, open in nature [19, 27, 20], the spectrum of categories for decentralized system is infinite. But we can use two attributes to categorize decentralized systems along two continuous dimensions: the degree of **coupling** among the agents (or nodes), and the **grain size** of the processors of the agents.

Coupling is a measure related to links between the agents in the system. Loose coupling means that information exchange amongst the agents is limited. In loosely coupled systems the agents spend most of their time in local processing rather than in communication among themselves. Tight coupling, therefore, indicates that there is no practical physical limit on the bandwidth of the communication channel between the agents. Because of excessive communication, tight coupling also indicates that the concept of bounded rationality of computing does not completely apply [35].

The **grain size** of the processors measures the individual problem-solving power of the agents. In this definition, problem-solving power amounts to the conceptual size of a single action taken by an agent visible to the other agents in the system. If the grain is coarse then the processing nodes are themselves rather sophisticated problem-solving systems with a fair amount of complexity. In coarse-grained applications, the distribution may be characterized to be, therefore, at the task level. Fine grain often indicates that the individual processors are functionally relatively simple, i.e., they do not exhibit any "intelligence" per se, and that their number in the system is substantial. Thus, the distribution in fine-grained applications is at the statement level as opposed to task level distribution.

2.3.1 Parallel Distributed Processing Systems

Decentralized, fine-grained systems with tight coupling are often referred to as parallel distributed processing systems [24, 8, 5, 19]. The processing aspect emphasizes concurrent execution of functionally decomposable tasks.

The objective in parallel distributed processing systems is usually load balancing of shared informational and physical resources. In distributed processing systems, the computational or syntactic motivations for decentralization are highlighted:

- speed,
- performance/cost,
- modularity,
- availability,
- scalability,
- reliability,

- extensibility,
- flexibility.

Although the current trends in the cost and availability of computer hardware would suggest that adding up enough conventional, low cost processors would result in an immense overall computing power with a reasonable investment, this has not proven to be the case. On the contrary, it has been recognized that a severe bureaucracy "bog-down" effect in multiprocessor systems calls for totally new architectural strategies to operate on the higher degree complexities in routine problem solving.

2.3.2 Distributed Problem Solving Systems Definition

As the opposite of PDP, we have distributed problem solving systems. These are defined informally as networks of loosely coupled, relatively coarse-grained, semiautonomous, "artificially intelligengent" asynchronous problem-solving agents, cooperating (or competing according to the domain) to fulfill their global mission. Asynchronous means that the agents are thought to function concurrently [24]. Cooperation means that because no node is capable of solving the entire problem by itself; the nodes have to work as a team and exchange knowledge about the tasks, results, goals, and constraints to solve the global problem or set of problems.

The **degree of cooperation** between the nodes in a decentralized problem-solving system may vary. On one extreme, the nodes may all be pursuing a common goal and be thus fully cooperative. This assumption is often referred to as the benevolent agent assumption. On the other extreme of the cooperation continuum, the nodes are nonbenevolent, i.e., they are self-interested, possessing conflicting goals and preferences. Thus, a process of negotiation to resolve the conflicts becomes crucial.

Decentralized problem-solving architectures with the last set of characteristics mentioned above are often categorized as nearly decomposable systems. In **nearly decomposable systems**, the interactions among the components are weak but not negligible. The emphasis in studying coordination within nearly decomposable systems is on dealing with the problems arising from restricted communication and bounded rationality. In the case of decentralized problem solving, the semantic motivation to pursue decentralization are thus addressed in terms of:

- complexity,
- possibility and
- natural decomposition.

2.4 Distributed Systems capabilities

As mentioned above, a distributed system has to be capable of parallel execution and of continuing in the face of single-point failures, so it must have:

- **Multiple processing elements** that can run independently. Therefore, each processing element, or node, must contain at least a CPU and memory[2].

[2]Note that multiple EMN-nodes may share a processor

- There has to be communication between the processing elements, so a distributed system must have **interconnection hardware** which allows processes running in parallel to communicate and synchronize.
- A distributed system cannot be fault tolerant if all nodes always fail simultaneously. The system must be structured in such a way that **processing elements fail independently**.
- Finally, in order to recover from failures, it is necessary that the nodes keep **shared state** for the distributed system.

2.5 Distributed Systems Problems

All these advantages of distributed systems cannot be satisfied due to the complexity of designing such systems. Some examples of system problems are:

- the amount of **interconnections** and risk of failure,
- the **interferences** between processes,
- the problem of **propagation of effects** between processes,
- the **information inconsistency** due to its duplication,
- the **effects of scale** due to the dimension of distributed systems and
- the **partial failure** of one processor that can perturbate the other ones [29, 18, 22, 27, 15].

The EMN architecture we define in this paper covers most of these aspects. The utilization of Artificial intelligence techniques to support communication and distribution offers help in solving most of these problems, especially propagation of effect, information inconsistency and partial failure.

3. Enterprise Management Network Node

The Enterprise Management Network links together two or more application nodes (EMN-nodes) by providing the "glue" that integrates the manufacturing enterprise through architectures and mechanisms to support decision making at all levels of the organization. For example, the CORTES system [16] is composed of an uncertainty analyser, a planner, a scheduler, a factory model and two dispatchers responsible for several machines (figure 3-1). Each is defined as an EMN-node.

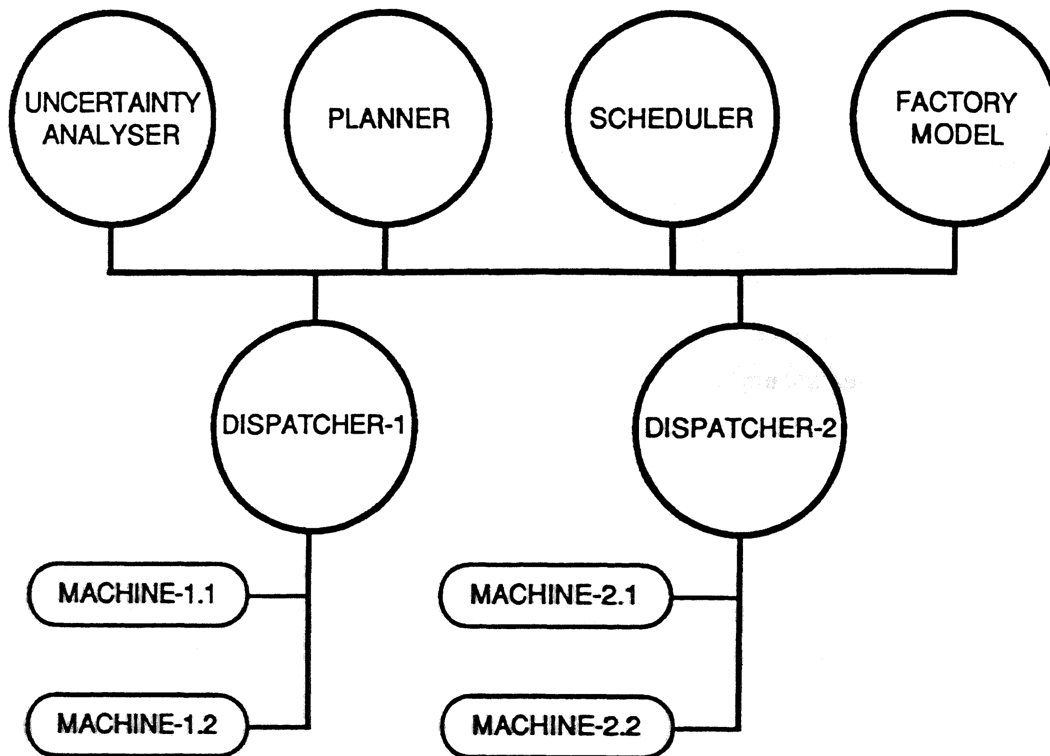


Figure 3-1: Example of decentralized system

Each EMN-node consists of the following subsystems[3] (figure 3-2):

- Problem Solving Subsystem,
- Knowledge Base,
- Knowledge Base Manager, and
- Communication Manager.

The **Problem Solving Subsystem** represents all the rules and functions which allow the EMN-node to solve problems related to its domain. The local execution cycle is triggered either by the internal transactions generated during local problem solving, or by external events forwarded to the EMN-node by the Communication Manager.

Each EMN-node contains a locally maintained **Knowledge Base** to support its problem solving. It is composed of objects which may be either physical objects (products, resources, operations, etc) or

[3] Currently implemented in CommonLisp

conceptual objects (customer orders, process plans, communication paths, temporal relations, etc). The knowledge base is expressed as CRL[4] schemata [26].

The **Knowledge Base Manager** manages information exchanges between the problem solving subsystem and the knowledge base, maintains the consistency of the local knowledge base, and responds to request made by other EMN-nodes. In the Enterprise Management Network, knowledge and data may be distributed throughout the network. It is the philosophy of the system that knowledge does not have to be available locally in order for it to be used by the EMN-node. Therefore, knowledge, in the form of schemata, fall into one of two classes: that *owned* by the knowledge source which must be stored locally, and knowledge *used* by the knowledge source, in which the original is stored at another EMN-node and a copy is stored locally.

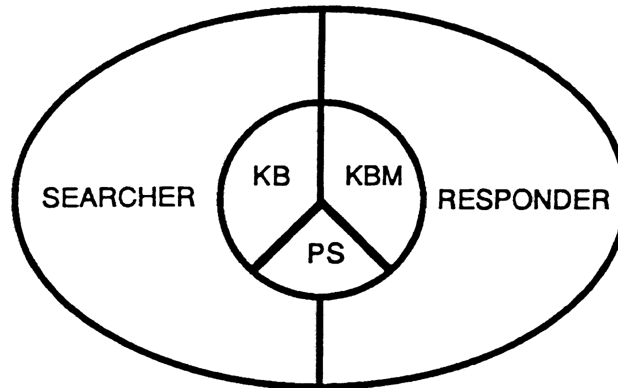


Figure 3-2: The elements of an EMN-node

A problem that arises in supporting the exchanges between the problem solving subsystem and the knowledge base is the unavailability of schemata locally. The problem solver often refers to objects that cannot be found locally, but may be found in another EMN-node's knowledge base. At the time of reference, the problem solver may or may not know where in the Enterprise Management Network the knowledge resides. It is the responsibility of the Knowledge Base Manager to "hunt down" the missing knowledge and to respond to like requests from other EMN-nodes. To accomplish this, the Knowledge Base Manager works with the **Communication Manager**. It both manages the search for information in the EMN and responds to like requests from other EMN-nodes. To perform these activities, the Communication Manager has two modules:

- The **searcher** communicates via message sending with other EMN-nodes. The searcher peforms two tasks: searching for knowledge not available locally, and the updating of knowledge changed and owned by the EMN-node. The policy for updating is defined in section 5.
- The **responder** answers messages originating from other EMN-nodes' searchers, and updates the local knowledge base according to updating messages.

The communication manager manages four types of events:

- ***Triggering***: information that triggers the node's processing.

[4]CRL stands for Carnegie Representation Language.

- ***Dynamic retrieval***: Requests for information not available in its knowledge base but necessary to perform its task. This information needs appear during the internal processing of an EMN-node.
- ***Updating information***: When an EMN-node, as the owner of some schemata, modifies these schemata, the searcher dispatches the modifications to other EMN-nodes that have local copies of these schemata. The responder may or may not update a local copy depending on the usage at the receiving EMN-node. Being the owner of a schema means, the EMN-node is the only one allowed to globally modify the content of a schema. But each EMN-node having a local copy of a schema can locally modify the content of that schema.
- ***Transaction request***: Similar to remote procedure calls.

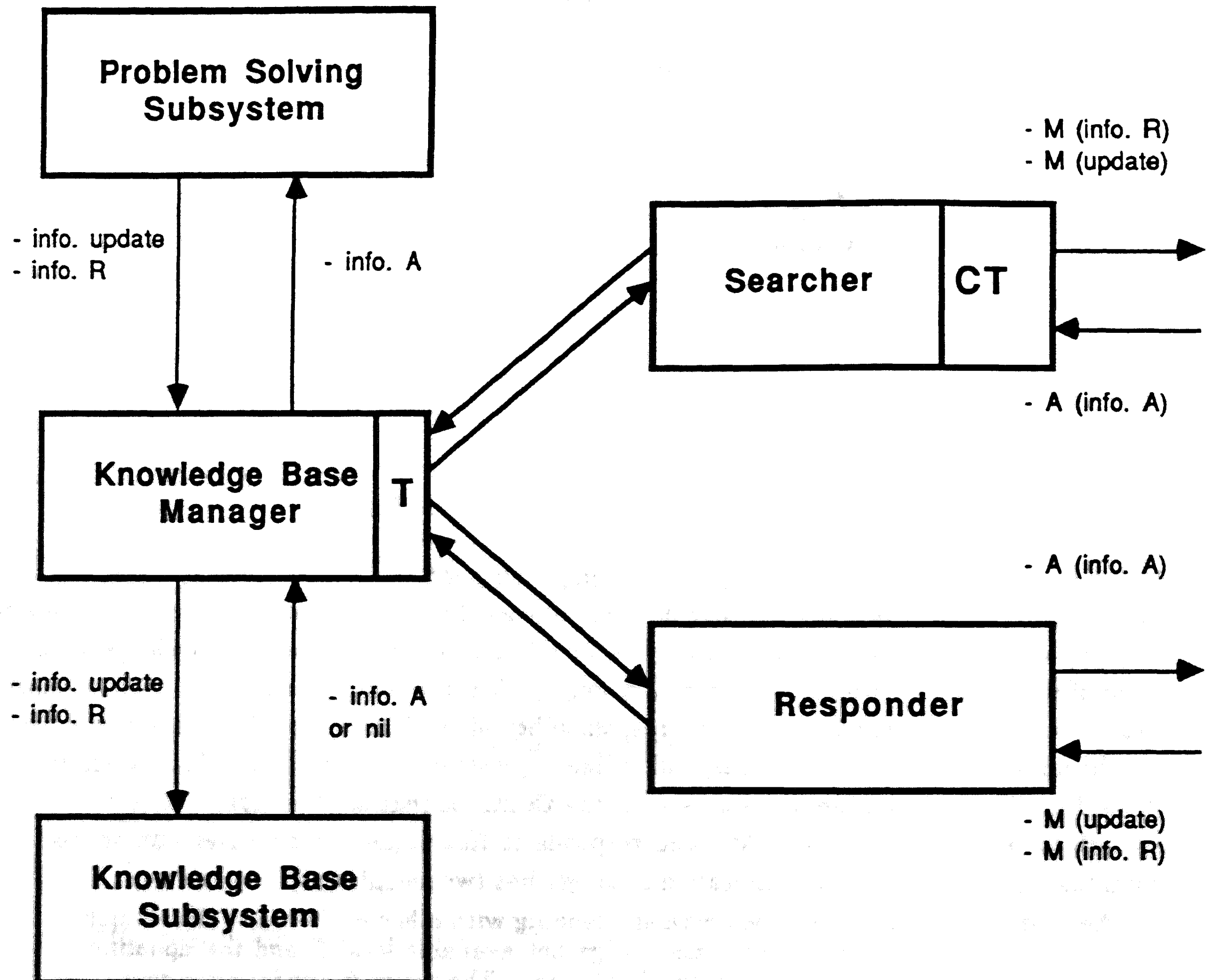


Figure 3-3: Information exchanges overview

We summarize all these exchanges between the modules of an EMN-node in figure 3-3. This figure shows the different types of information sent and received by each module (M stands for Message, A stands for Answer, R stands for Request, T stands for Translator and CT stands for Correspondance Table). Their content will be discussed in the following sections.

To illustrate the functionalities of the three first layers of the EMN architecture, we will consider a decentralized system composed of three agents, connected by a network. Each agent has a specific Problem Solving subsystem (PS) and a specific Knowledge Base subsystem (KB) (figure 3-4). We describe in this first figure an empty decentralized system, e.g., without the Enterprise Management Network. We will extend this example by adding at each Level description the specific elements, functions and protocols defined there.

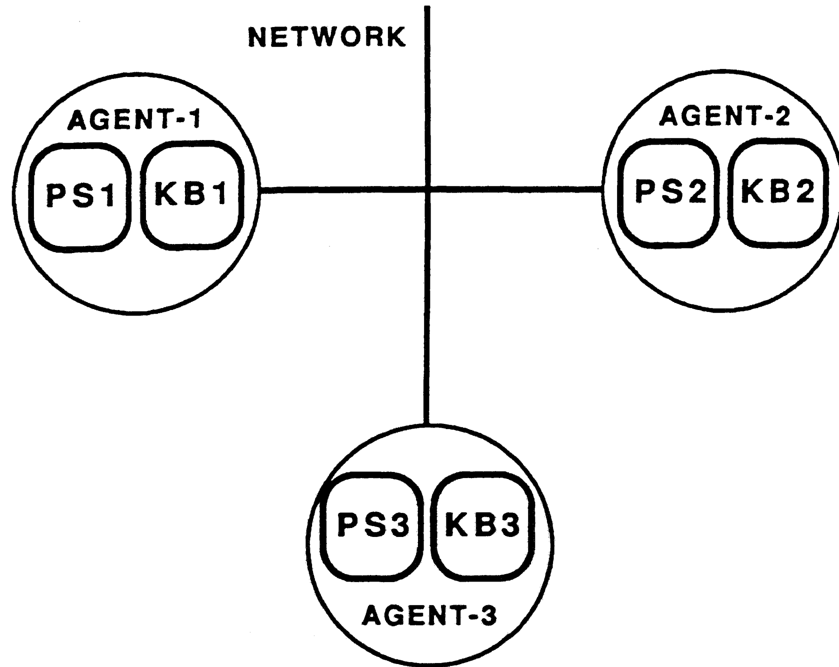


Figure 3-4: Decentralized system example

4. Network Layer

4.1 Introduction

The **Network Layer** defines the EMN-nodes that will participate in the Enterprise Management Network. It assumes the existence of all the hardware and software facilities for this structure such as: a network (in our case DECnet[5]), computers (in our case VAX-station[6] 3200s) and application software (in our case Knowledge Craft[7]). It allows the identification of an EMN-node and specifies its basic elements such as mail box, semaphore box, queues and low level message. In addition to these elements, the Network Layer provides some basic primitives for this architecture. These primitives are message passing functions with blocking and without blocking.

The Network Layer defines the following network components:

- *EMN-nodes* represent problem solving agents. They include the basic communication objects: queues, low level message, mail box and semaphore box[8]. Each EMN-node initialization, is specified by an EMN-node schema (schema 4-2).
- *Channels* define communication links between EMN-nodes. Each channel is defined as an instance of the channel schema (schema 4-3).
- *Messages* can be sent along channels between EMN-nodes. During the information transfer, an EMN-node may be suspended (blocked) while awaiting a reply. Each message is defined as an instance of the network-message schema. These instances are stored in queues (supported by the network-message-queue schema). The message passing and message reception is supported by some basic communication functions dependent on the hardware and operating system.
- *Protection* is provided so that messages can only be processed by legal EMN-nodes.
- *Synchronization primitives* are defined to synchronize the internal problem solving of an EMN-node and communication activities. Primitives such as "block-agent" are implemented to interrupt the problem solving process until the excution of the "unblock-agent" primitive. Blocking is used when information is needed for problem solving but not available locally in the knowledge base system. In that case, the problem solving is interrupted until the reception of the information. The blocking functions use selective interruption: they only suspend the problem solving process and keep running the message sending and message reception processes. This capability is very important in terms of performance as well as coherence of the distributed system. Since the blocking function executes selective blocking of the problem solving, during such an interruption an agent is still able to answer received messages, to update and to distribute information.

---

[5]DECnet is a registered trademark of Digital Equipment Corporation.

[6]VAX is a registered trademark of Digital Equipment Corporation.

[7]Knowledge Craft is a registered trademark of Carnegie Group Inc.

[8]Note that these two last objects are dependent of the used operating system

4.2 Network Specification

Each implementation of our Enterprise Management Network must be specified. For this purpose, we have created a schema called **Network**. It describes the main characteristics of the global decentralized system by defining specific names for the net and its EMN-nodes (schema 4-1).

Schema 4-1: Network

Network		
SLOT	FACET	VALUE
Name	*Value:* *Restriction:*	type string
Type	*Value:* *Restriction:*	network type
EMN-nodes	*Value:* *Restriction:*	type EMN-node-name*

In fact, each instance of the network schema represents a specific implementation of our Enterprise Management Network architecture.

4.3 EMN-node specification

The Network Layer of the manufacturing architecture provides the most primitive functions that enable manufacturing processes to participate in a distributed manufacturing system. All the schemata presented at this layer must be duplicated in every EMN-node.

4.3.1 Schemata supporting EMN-nodes initialization

At the Network Layer, we intend to model the main characteristics of each EMN-node and initialize it as a member of the decentralized system. We have defined an EMN-node as a combination of a Problem Solving subsystem, a Knowledge Base subsystem and a communication subsystem. The communication subsystem is composed of several elements. All the information that identifies an EMN-node must be stored in a schema. The EMN-node identification, has been implemented using a DKC-system[9] schema. This schema indicates all the details related to an EMN-node. The initialization of an EMN-node as a member of the decentralized system is done by creating an instance of the **DKC-system** schema (schema 4-2) in the corresponding EMN-node. This schema initializes an EMN-node and also all the elements necessary for the communication activity: the queues, the timers (one for the updating activity and one for the message reception), the flags and the triggers. This function supports mainly the local initialization of an EMN-node but does not support its instantiation as part of the decentralized system. Only local elements are defined in the DKC-system schema. The creation of this schema is supported by a Lisp function called: DKC-init.

The DKC-system schema contains the name of its local-channel schema. This schema indicates the addresses of its mail-box file and of its semaphore-box file. The third slot of the DKC-system

[9] DKC stands for Decentralized Knowledge Craft

schema is the list of channels with the other EMN-nodes. Each time a channel is created, its name is stored as a value of this slot. The queued-messages slot contains the new received messages. All the other slots define names and addresses of flags and VMS routines to support the message passing functionality.

Schema 4-2: DKC-System

DKC-System		
SLOT	FACET	VALUE
Initialized	*Restriction:*	t/nil
Local-channel	*Value:* *Restriction:*	type dkc-channel
Channels	*Value:* *Restriction:*	type dkc-channel*
Queued-messages	*Value:* *Restriction:*	type dkc-queued-message*
Interrupt-function	*Value:* *Restriction:*	type lispobj/nil
Interrupt-lost	*Restriction:*	nil/t
Update-lost	*Restriction:*	nil/t
Timer-efn	*Value:* *Restriction:*	type integer
Update-efn	*Value:* *Restriction:*	type integer
New-message-efn	*Value:* *Restriction:*	type integer
Trigger-interrupt-id	*Restriction:*	t/nil
Updating-message-trigger-id	*Restriction:*	t/nil

The EMN-nodes of a decentralized system are connected by the network which allows them to transfer information from one agent to another. But this transfer of information cannot be done without knowledge about the existence of other agents on the network. Besides, since an agent has a specific purpose, it includes some specific elements. These elements are the knowledge base sub-system and the problem solving sub-system. As we intend to create an Enterprise Management Network allowing Problem Solving negotiation, we must provide to each EMN-node the capability of recognizing who are the other members of this network. Blind communication is possible and easy to perform, i.e. broadcasting (in some cases we will use this capability), but the direct communication type is more efficient. For this purpose, we define links between the EMN-nodes. We call these links channels.

Channels allow mutual recognition between agents. They are conceptual links. The physical link is the network. Channels must be created in both directions, i.e., EMN-node A1 must have a channel with EMN-node A2, but EMN-node A2 must also have a channel with EMN-node A1. If four nodes in a network are to communicate with each other, then each EMN-node must have three channels.

Schema 4-3: DKC-Channel

DKC-Channel		
SLOT	FACET	VALUE
Node	*Value:* *Restriction:*	type EMN-node-name
MailBox	*Value:* *Restriction:*	type symbol
SemaphoreBox	*Value:* *Restriction:*	type symbol

The creation, in each EMN-node, of an instance of the DKC-system schema initializes the decentralized communication system. A new EMN-node, to become part of the global decentralized system, must create some links with the other already existing EMN-nodes of the system. We must establish strong connections between the EMN-nodes of the DKC structure (Decentralized System). For that purpose, we create, using the **DKC-channel** schema (figure 4-3), channels between the EMN-nodes of the DKC structure. These channels allow us to define the address of each EMN-node[10]. In this way, we are able to transfer information (or schemata) through the network, between the different EMN-nodes.

Schema 4-4: local-DKC-Channel

local-DKC-Channel		
SLOT	FACET	VALUE
Key-words	*Value:* *Restriction:*	type string*
Node	*Value:* *Restriction:*	type local-EMN-node-name
MailBox	*Value:* *Restriction:*	type symbol
SemaphoreBox	*Value:* *Restriction:*	type symbol

An instance of the DKC-channel schema is created between an EMN-node and each of the other EMN-nodes of the global system. But locally, an instance of this schema is also created and called the **local-channel**. This schema contains the information about the local EMN-node name and address and also a list of key-words which are attached to a specific EMN-node. These key-words will be used at the upper level of the EMN Architecture. This slot is only created for the local-channel schema. At the initialization of an EMN-node, this information is not available for the other channels created with the other EMN-nodes. These key-words define the information type an EMN-node is using. This slot is completed automatically by Lisp functions defined at the Data Layer of this architecture.

---

[10]Note that in our specific implementation the address of each EMN-node is defined using mail-box and semaphore-box

4.3.2 Functions supporting EMN-nodes initialization

The initialization of each EMN-node means the instantiation for each EMN-node of the DKC-system schema, of the local-channel schema and of the channel schemata with all the other EMN-nodes of the system. This activity is supported by several Lisp functions, such as dkc-init and open-channel:

- **EMN-node-initialization**: This function allows the creation of an instance of the EMN-node schema. We must use this function at the beginning of the utilization of the Enterprise Management Network architecture. Since, by initializing an EMN-node, we define all the elements needed for communication such as mail box, semaphore, etc., the implementation of a specific EMN-node must start with a call to this function (this function is implemented as the "DKC-init" function).
- **EMN-node-deletion**: This function deletes an EMN-node schema instance. As we can initialize an EMN-node, we can also remove an EMN-node from the decentralized system. Removing an EMN-node is more complicated than it seems. Since all the EMN-nodes know the existence of the other EMN-nodes, to interrupt one EMN-node requires an updating of all this knowledge (this function is implemented as the "DKC-term" function).
- **Link-between-EMN-nodes-creation**: With this function we create an instance of the channel schema. This function provides to each EMN-node the knowledge for direct communication. They achieve a mutual recognition. The role of this function is very important for efficiency (this function is implemented as the "open-channel" function).
- **Link-between-EMN-node-deletion**: With this function we delete an instance of the channel schema, and we suppress a link between two EMN-nodes. As for the removing of an EMN-node from the decentralized system, the deletion of a channel must be done carefully. For example: a channel deletion must be done in both directions (this function is implemented as the "close-channel" function).

4.3.3 Example of EMN-node initialization

At the initialization of each EMN-node, the schemata presented at this layer are created. As we have seen in the previous section, the instantiation of these schemata is supported by Lisp functions. The first step for the Network Layer instantiation in an EMN-node is the utilization of the DKC-init function. This function creates the DKC-system schema and the local-channel schema:

```
(DKC-init 'agent-1
          "[cortes.mailbox]agent-1.sem;1"
          "[cortes.mailbox]agent-1.box;1"
          'interrupt-function-1)
```

This function creates agent-1-DKC-system schema (schema 4-5) and agent-1-DKC-local-channel schema (schema 4-6) (in this example). These schemata define the local elements of the agent-1.

This command has instantiated agent-1 EMN-node and has also created two files: [cortes.mailbox]agent-1.sem;1, which is the semaphore-box file of agent-1 and the [cortes.mailbox]agent-1.box;1 which is the mail-box file of agent-1. In addition, parameters specific to this new EMN-node are defined such as its name, agent-1, which is a unique value. This function also initializes the interrupt-function-1. As we said previously in this chapter, the blocking mechanism used by an EMN-node can either be the generic blocking primitive or a specific interrupt function. In this example, the second possibility is used. The different queues and objects such as the

network-message-schema are created to support the communication primitives triggered by the DKC-init function.

Schema 4-5: Agent-1-DKC-System

Agent-1-DKC-System		
SLOT	FACET	VALUE
Initialized	*Restriction:*	t
Local-channel	*Value:*	agent-1-dkc-local-channel
Channels	*Value:*	nil
Queued-messages	*Value:*	nil
Interrupt-function	*Value:*	interrupt-function-1
Interrupt-lost	*Restriction:*	nil
Update-lost	*Restriction:*	nil
Timer-efn	*Value:*	timer-efn
Update-efn	*Value:*	update-efn
New-message-efn	*Value:*	new-message-efn
Trigger-interrupt-id	*Restriction:*	trigger-interrupt-id
Updating-message-trigger-id	*Restriction:*	updating-message-trigger-id

Schema 4-6: agent-1-DKC-local-Channel

agent-1-DKC-local-Channel		
SLOT	FACET	VALUE
Instance	*Restriction:*	dkc-channel
Key-words	*Value:*	nil
Node	*Value:*	agent-1
MailBox	*Value:*	[cortes.mailbox]agent-1.box;1
SemaphoreBox	*Value:*	[cortes.mailbox]agent-1.sem;1

This function creates also the different queues and schemata to support the different primitives of this layer. In addition, the VMS routines which support these primitives are triggered. The network layer instantiation is completed by the creation of the channels between this EMN-node (agent-1) and the other EMN-nodes of the system. This channel creation is supported by the "open-channel" Lisp function. Each time this function is called, a new channel schema is created between the local EMN-node and the others. If we take the example of the channel creation between agent-1 and agent-2, we have to execute this function:

```
(open-channel 'agent-2
              "[cortes.mailbox]agent-2.sem;1"
              "[cortes.mailbox]agent-2.box;1")
```

This function creates agent-2-DKC-channel schema (schema 4-7) and it represents the link between agent-1 and agent-2 for the message passing purpose. This schema indicates the address of the mail-box and semaphore box files of the agent-2. Besides, it also indicate the agent-2 node name. This name is used to identify a specific EMN-node.

Schema 4-7: agent-2-DKC-Channel

agent-2-DKC-Channel		
SLOT	FACET	VALUE
Instance	*Restriction:*	DKC-channel
Node	*Value:*	agent-2
MailBox	*Value:*	[cortes.mailbox]agent-2.box;1
SemaphoreBox	*Value:*	[cortes.mailbox]agent-2.sem;1

In addition, as a new DKC-channel schema is created, this information is stored as a value of the channels slot of the Agent-1-DKC-system schema (schema 4-8). To create other links with the other EMN-nodes of the system, we follow the same process. Each time the "open-channel" function is called, an instance of the DKC-channel schema is created and agent-1-dkc-system schema is updated.

Schema 4-8: Agent-1-DKC-System

Agent-1-DKC-System		
SLOT	FACET	VALUE
Initialized	*Restriction:*	t
Local-channel	*Value:*	agent-1-dkc-local-channel
Channels	*Value:*	agent-2-dkc-channel
Queued-messages	*Value:*	nil
Interrupt-function	*Value:*	interrupt-function-1
Interrupt-lost	*Restriction:*	nil
Update-lost	*Restriction:*	nil
Timer-efn	*Value:*	timer-efn
Update-efn	*Value:*	update-efn
New-message-efn	*Value:*	new-message-efn
Trigger-interrupt-id	*Restriction:*	trigger-interrupt-id
Updating-message-trigger-id	*Restriction:*	updating-message-trigger-id

4.4 Communication Procedures

4.4.1 Schemata supporting the communication procedures

EMN-nodes communicate via **message** that are stored in a **message-queue.** These two elements are also defined as schemata.

These two schemata (schema 4-9 and schema 4-10) are network level communication elements, i.e. their purpose is message passing. At the other levels we will define other message schemata and queues for more intelligent communication activity. These other schemata will be based on these network layer schemata. The **DKC-message** schema allows the transfer of schemata from one initialized EMN-node to another through the channels. The **DKC-queued-message** schema stores the exchanged DKC-messages before transfering them to or reading them from the mail box of the destination EMN-node. These two schemata use also Lisp functions for their creation.

Schema 4-9: DKC-message

DKC-message		
SLOT	FACET	VALUE
Sender	*Value:* *Restriction:*	type EMN-node-name
Priority	*Restriction:*	0.0 to 1.0
Role	*Value:* *Restriction:*	type lispobj
Dialog	*Value:* *Restriction:*	type lispobj
Superclass	*Value:* *Restriction:*	type lispobj
Class	*Value:* *Restriction:*	type lispobj
Type	*Restriction:*	schema/string/lispobj
Schemata	*Value:* *Restriction:*	type schema*
From-context	*Value:* *Restriction:*	:current-context
To-context	*value:* *Restriction:*	(:from-context)
Parallel	*Restriction:*	t/nil
Metas	*Restriction:*	t/nil
Relation	*Restriction:*	t/nil
Strings	*Value:* *Restriction:*	type string*
Lispobjs	*Value:* *Restriction:*	type lispobj*

The DKC-message schema can support the transfer of schemata of strings and of Lisp objects. To specify the information we are transfering, we indicate its value in the "type" slot of the message. Different combination of context transfer are supported:

- <context> => <context>
- <context-1> => <context-2>
- <context> => :from-context
- :simple => :simple
- :simple => :from-context
- :current-context => <context>
- :current-context => :current-context
- :current-context => :from-context

We attribute to each message a priority. The aim of this parameter is to select from the message queue the high level priority message. We only indicate the message sender (the EMN-node which generates and sends the message) because the message receiver is the message destination.

Schema 4-10: DKC-queued-message

DKC-queued-message		
SLOT	FACET	VALUE
Sender	*Value:* *Restriction:*	type EMN-node-name
Priority	*Restriction:*	0.0 to 1.0
Role	*Value:* *Restriction:*	type lispobj
Dialog	*Value:* *Restriction:*	type lispobj
Superclass	*Value:* *Restriction:*	type lispobj
Class	*Value:* *Restriction:*	type lispobj
Type	*Restriction:*	schema/string/lispobj
Body	*Value:* *Restriction:*	type lispobj*

4.4.2 Functions supporting the communication procedures

In the Network Layer, we define several types of primitives. The first ones allow the initialization of an EMN-node (EMN-node-initialization). This initialization is done by creating instances of the different schemata presented in the first section. In each EMN-node, we must create an instance of the EMN-node schema. This instantiation also implies the creation of instances for the Knowledge-base, knowledge-object, Problem-solving and procedure schemata.

For mutual recognition between EMN-nodes, instances of the channel schema are defined. This definition is also supported by primitives (link-between-agent-creation).

In addition, we define primitives for communication purposes. The two main primitives are message passing and message reception. We can summarize the communication and synchronization functions we define at the Network Layer:

- Message sending
  - **Message-sending-with-blocking**: This function allows the execution of message passing that blocks the running of the problem solving subsystem until the reception of the answer to this message. The implementation of such a function is specific to each EMN-node. For this reason we will provide a generic and also a specific blocking function. The generic one will execute the Problem Solver blocking without conditions. The specific one will use the generic one and will add conditions for blocking.
  - **Message-sending-without-blocking**: This function allows simple message passing. It takes place only for asynchronous message passing. In this case, once message sending is done, the Problem-solving subsystem continues its processing (implemented as the "dkc-write" and the "dkc-send" functions).
- Message reception
  - **Message-reception-with-blocking**: This function allows message reception with blocking of the problem solving of the EMN-node. For this one we also define a generic and a specific function. We can establish priority according to the nature of the information which is received.
  - **Message-reception-without-blocking**: This function allows simple message reception. In this case, no blocking of the problem solving is executed. Mail box checking is performed once the problem solving has finished its activity (implemented as the "dkc-read" function).
- **Blocking**: This is a mechanism for stopping the problem solving subsystem from running. This function is a primitive used by the four other functions defined previously. This primitive causes a selective interruption of the problem solving process out, keeps running two others: message-sending and message-reception (implemented as the "block-EMN-node" function).
- **Unblocking**: This is a mechanism to re-start the running of the problem solving subsystem. It is also a primitive function used by the previous ones. The utilization of this function can only take place after the utilization of the blocking one (implemented as the "unblock-EMN-node" function).

4.4.3 Example of communication function implementation

In this section, we define the algorithms used for a specific implementation of the Network Layer communication functions. This example assumes the utilization of Vax-3200[11], DECnet[12], VMS[13] operating system and Knowledge Craft[14].

---

[11]VAX is a registered trademark of Digital Equipment Corporation.

[12]DECnet is a registered trademark of Digital Equipment Corporation.

[13]VMS is a registered trademark of Digital Equipment Corporation.

[14]Knowledge Craft is a registered trademark of Carnegie Group Inc.

4.4.3.1 Message passing without blocking

Message passing is used to transfer a network-message schema from one EMN-node to another (taking into account protections). This transfer is done using the channel schema and also the mail box and semaphore box of the EMN-node destination of the message. The different steps of this primitive are described in figure 4-1.

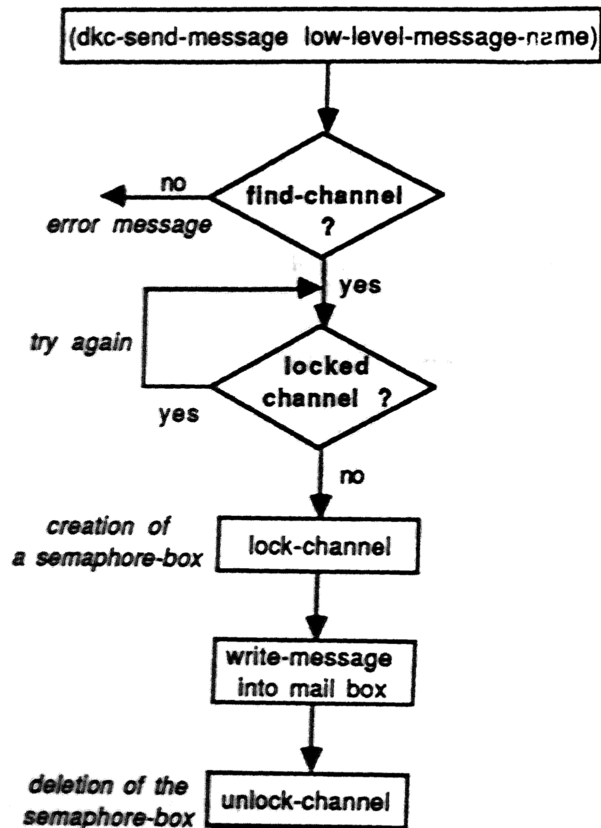


Figure 4-1: Message passing algorithm

The message passing function includes several steps (figure 4-1). The first one concerns th identification of the message destination. Through the creation of channels, each EMN-node ha knowledge about the existing EMN-nodes of the global system. A message can be sent from on EMN-node to another only if the corresponding channel already exists. Once the channel existenc verification is performed, we can test the availability of this channel. The aim of this test is t prevent conflicts, i.e. if two EMN-nodes want to write in the same mail box at the same time.

This second step is the verification of the channel (and also mail-box) status. When the mail bo file already exists, we verify the existence of the semaphore box. When the semaphore box exists, i.e another EMN-node is reading or writing into the mail-box, the EMN-node waits until it is deleted The semaphore box is used to lock both writing and reading of the mail box. When the semaphor box is deleted, we can execute the last step of the message passing primitive, which is a sequence c three phases:

1. lock the channel (by creating a semaphore box)
2. write the message into the destination mail box (if the mail box already exists, we append the new message to this file, if not, we create this file and copy this new message in this file)
3. unlock the channel (by deleting the semaphore box)

The different steps of the message passing function are described on figure 4-2 which gives an example with two EMN-nodes.

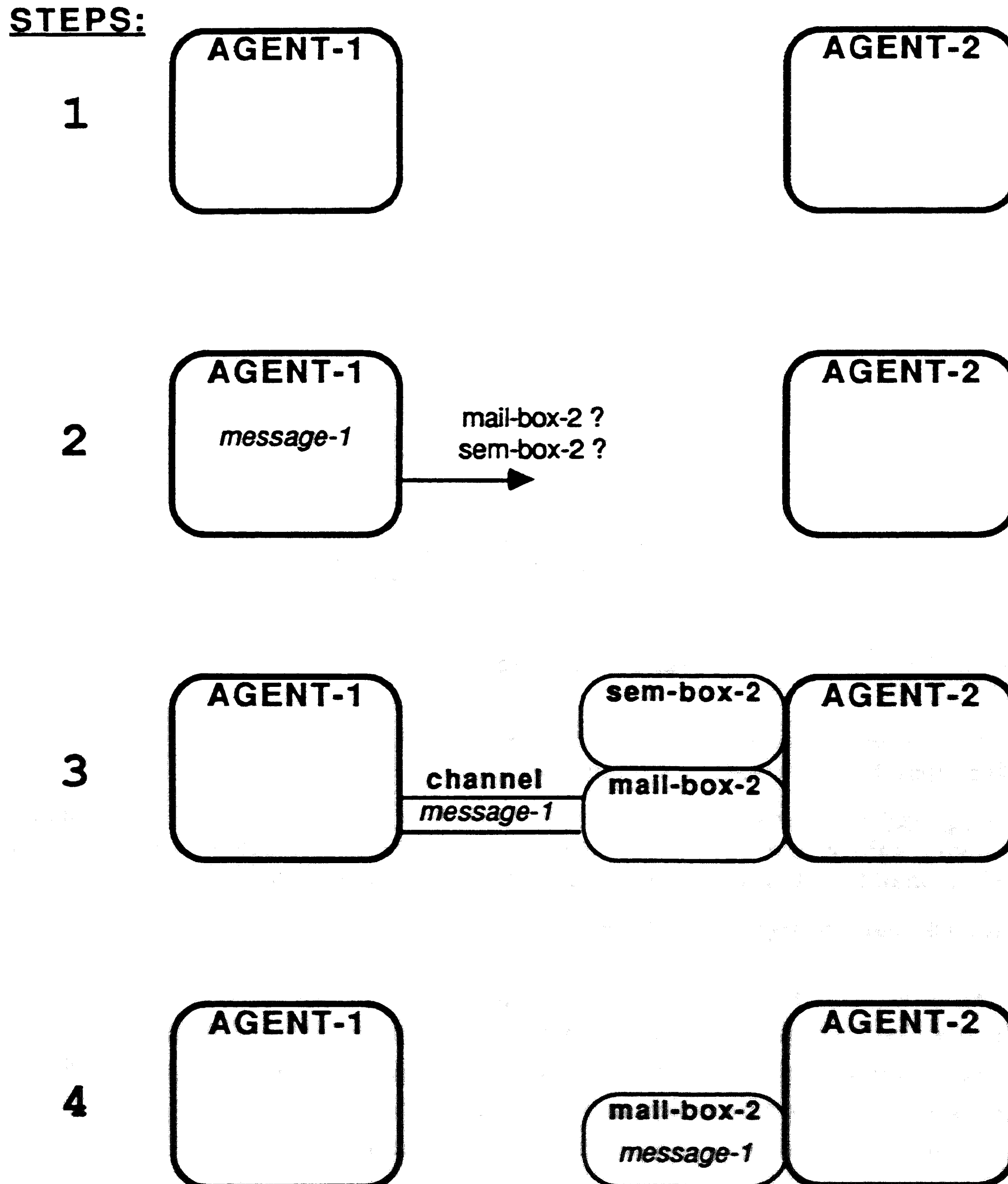


Figure 4-2: Message passing steps

4.4.3.2 Message reception

In the Network Layer, a second function is defined: **message-reception** (figure 4-3). This function allows reading the contents of the mail box. When it is triggered, it checks first the mail box status. When the mail box exists, it means another EMN-node has written a message in it, in which case we verify the existence of the semaphore box. If the semaphore box exists, i.e., the mail box is locked, the EMN-node waits until it is unlocked, i.e. the semaphore box is deleted. When the mail box is unlocked, the second step can be executed.

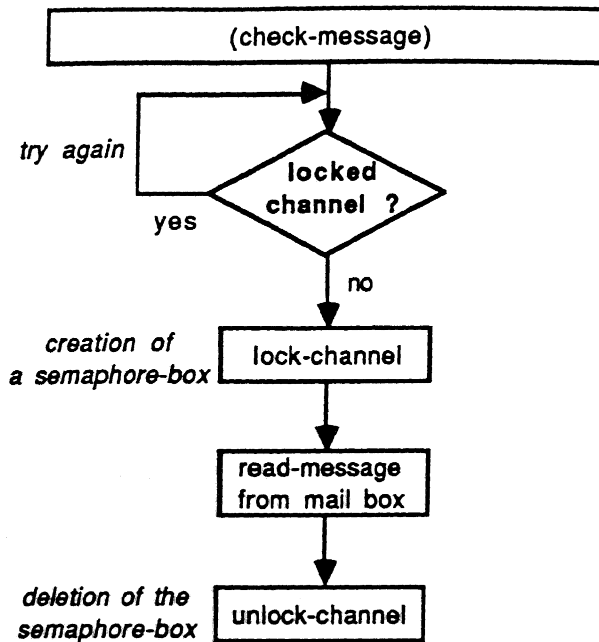


Figure 4-3: Checking mail box algorithm

The second step of this function is performed in three phases:

1. lock mail box (by creating a semaphore box)
2. read mail box (by copying this message from the mail box to the EMN-node problem solving sub-system). Once the message is transfered, it is deleted from the mail box and if the mail box is empty, it is also deleted, i.e. the mail box file is deleted.
3. unlock mail box (by deleting the semaphore box)

This primitive is equivalent to the message passing one except that it reads instead of writing into the mail box and the "mail box existence checking" test is added. The mail box existence checking test allows to suppress or to keep the execution of the second second step of that function, i.e. before reading the mail box we test if this mail box exists or not, if it does not exists, this function return 'nil. If it exists, the second step is executed, i.e. mail box reading.

The different steps of the message reception function are described on figure 4-4 which gives an example with two EMN-nodes.

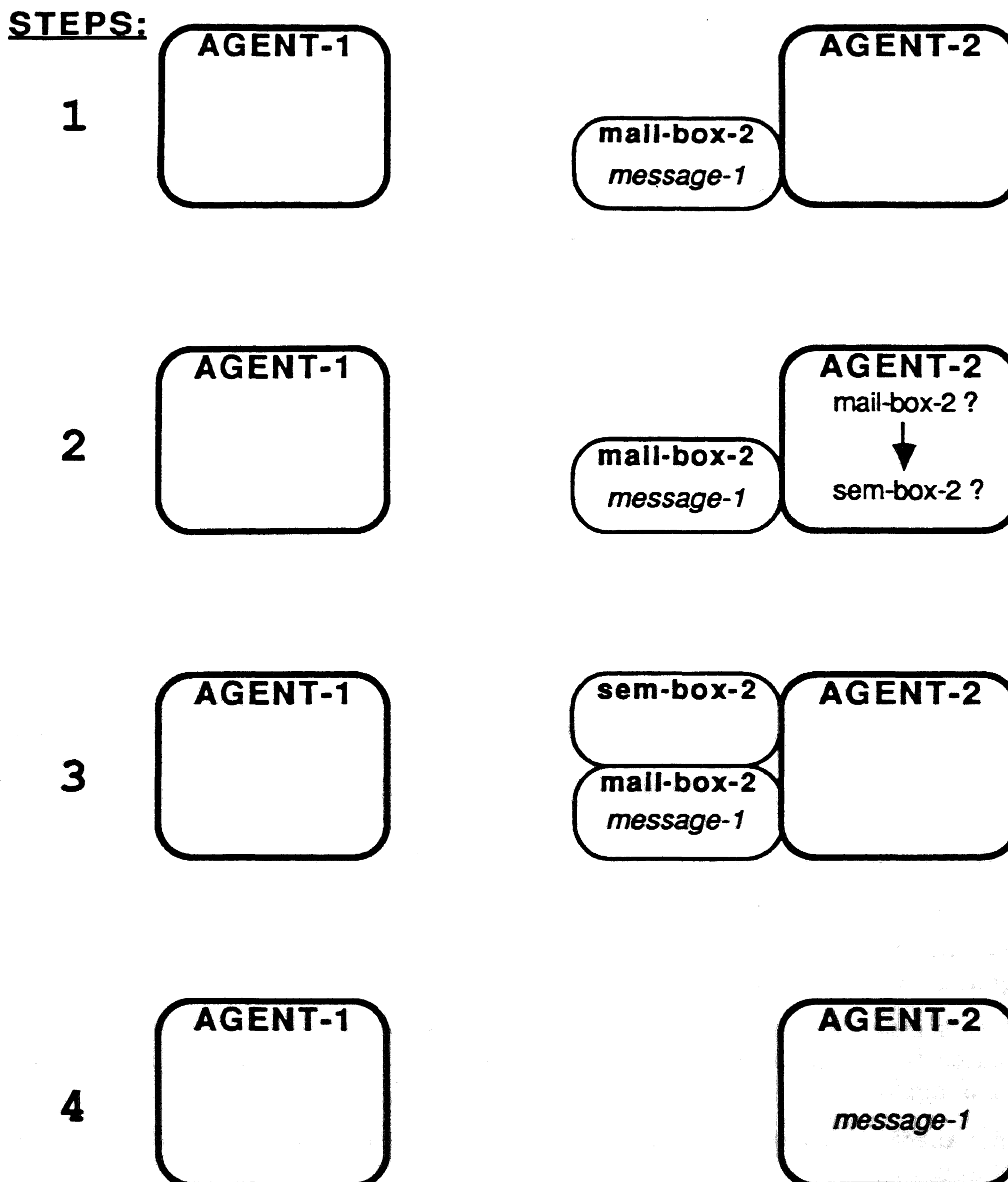


Figure 4-4: Message reception steps

4.4.3.3 Message passing with blocking

The last function provided at the Network Layer is blocking. This function allows interruption of the Problem Solving execution. This interruption occurs when the PS needs information; we suspend its running until the reception of the needed information.

The blocking primitive can be used either with message passing or message reception. We describe an example of message passing with blocking, and the reception of the answer, which unblocks the EMN-node. In the first steps (figure 4-5), the searcher, due to an information lack during the execution of the Problem Solving, generates a message to acquire this information. Problem Solving is blocked to suspend its running until the reception of the needed information.

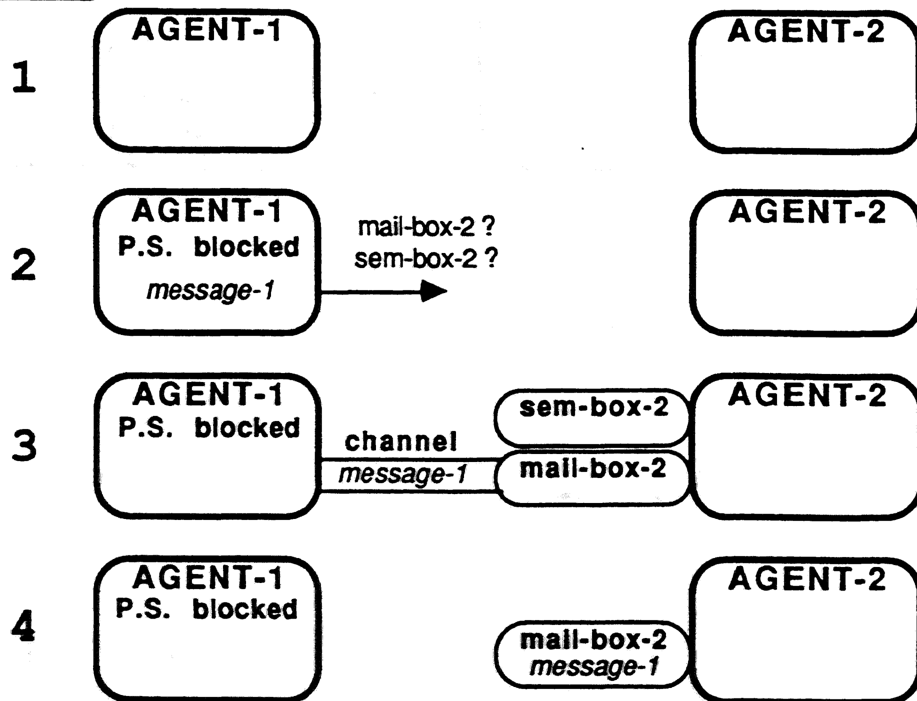


Figure 4-5: Message passing with blocking: step 1

These first steps consist in the passing of message-1 from agent-1 to agent-2:

- The problem solving of agent-1 is blocked when message-1 is generated.
- The searcher verifies the existence of a channel with agent-2 and also the existence of agent-2's mail-box and semaphore-box.
- In this case, both files, i.e. semaphore and mail box files, did not exist, so they are created by the searcher of agent-1. Message-1 is sent through the channel to the mail box of agent-2.
- Once message-1 is copied into the mail box of agent-2, the semaphore box of agent-2 is deleted, i.e., mail-box-2 is unlocked.

In the second step (figure 4-6), agent-2 receives message-1 (generated by agent-1 due to an information lack). In response to message-1, the responder of agent-2 generates message-2.

This next step consists in mail-box-2 checking by the responder of agent-2. This mail box checking is executed according to these steps:

- The responder of agent-2 is triggered by the existence of mail-box-2 file. It checks the existence of semaphore-box-2, i.e., the status of mail-box-2.
- To lock mail-box-2, agent-2 responder creates sem-box-2 file.
- Then it reads the content of mail-box-2.
- Once all the messages contained in mail-box-2 are copied to agent-2, mail-box-2 and sem-box-2 are deleted.

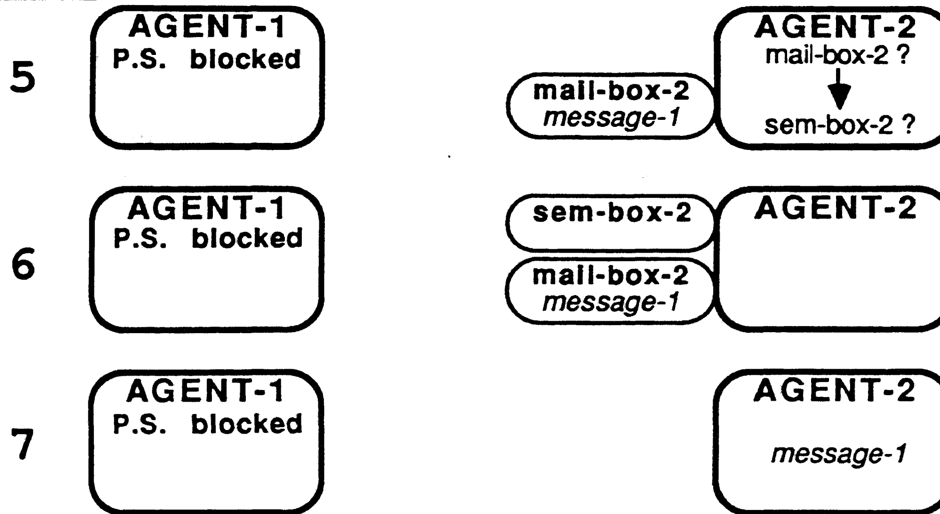


Figure 4-6: Message passing with blocking: step 2

The message generated by agent-2 is sent back to agent-1 using the same sequence as in step one (figure 4-7). Step 3 is the passing of message-2 from agent-2 to agent-1.

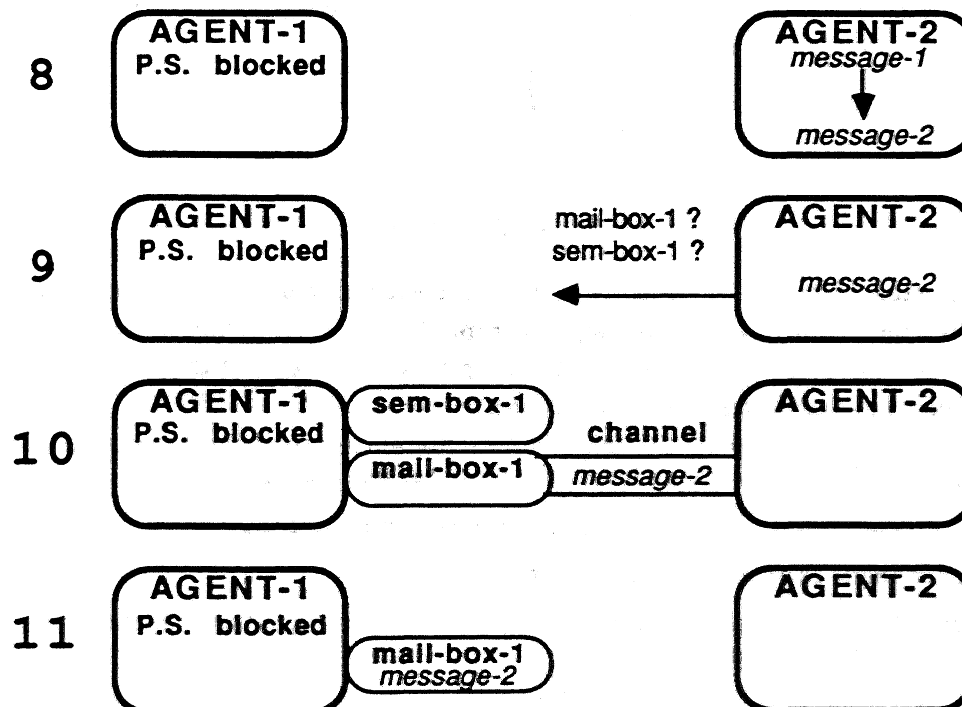


Figure 4-7: Message passing with blocking: step 3

Step 3 contains the following phases:

- The searcher of agent-2 verifies the existence of a channel with agent-1 and also the existence of agent-1's mail-box and semaphore-box.
- In this case, both files, i.e. semaphore and mail box files, did not exist, so they are created by the searcher of agent-2. Message-2 is sent through the channel to the mail box of agent-1.
- Once message-2 is copied into the mail box of agent-1, the semaphore box of agent-1 is deleted, i.e., mail-box-1 is unlocked.

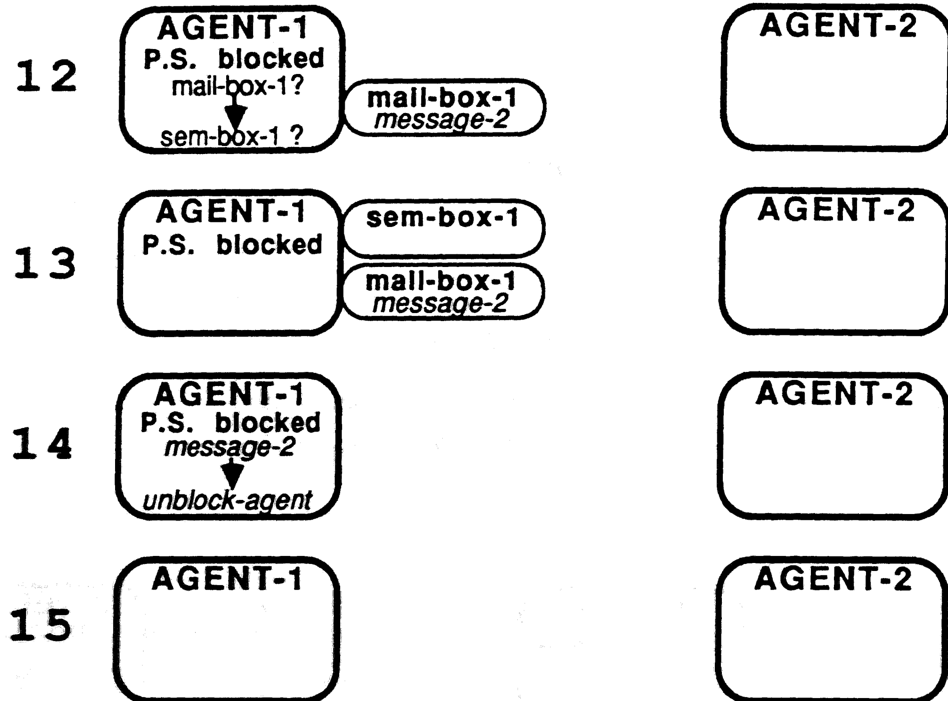


Figure 4-8: Message passing with blocking: step 4

The last step (figure 4-8), illustrates the reception of message-2 by agent-1. When mail-box-1 checking is executed and when message-2 is copied from the mail box to the KBS, problem solving is unblocked. We must specify that the unblocking takes place only if the message contains the needed information. The following steps are executed:

- The responder of agent-1 is triggered by the existence of mail-box-1 file. It checks the existence of semaphore-box-1, i.e., the status of mail-box-1.
- To lock mail-box-1, agent-1 responder creates sem-box-1 file.
- Then it reads the content of mail-box-1.
- Once all the messages contained by mail-box-1 are copied into agent-1, mail-box-1 and sem-box-1 are deleted.
- As message-2 contains the needed information, this information is provided to the Problem Solver and the PS is unblocked.

4.5 Network Layer example

If we continue to describe the example we used in section 2 (figure 3-4), by adding the network layer to this empty structure, we get figure 4-9.

The main modifications which occur in this structure are:

- The initialization of the EMN-node. This includes the definition, for each EMN-node, of the decentralized system, of a name and of an address.
- The creation of links between the EMN-nodes through the utilization of channels. These channels also include the basic primitives for the message passing activity and all the schemata needed by this activity: queues and network-message schemata.

This first layer provides the frame for communication. Each EMN-node is defined and knows about each others in terms of existence. Basic communication functions are provided to support message exchanges between identified EMN-nodes. In addition, security mechanisms such as EMN-node blocking and unblocking are specified.

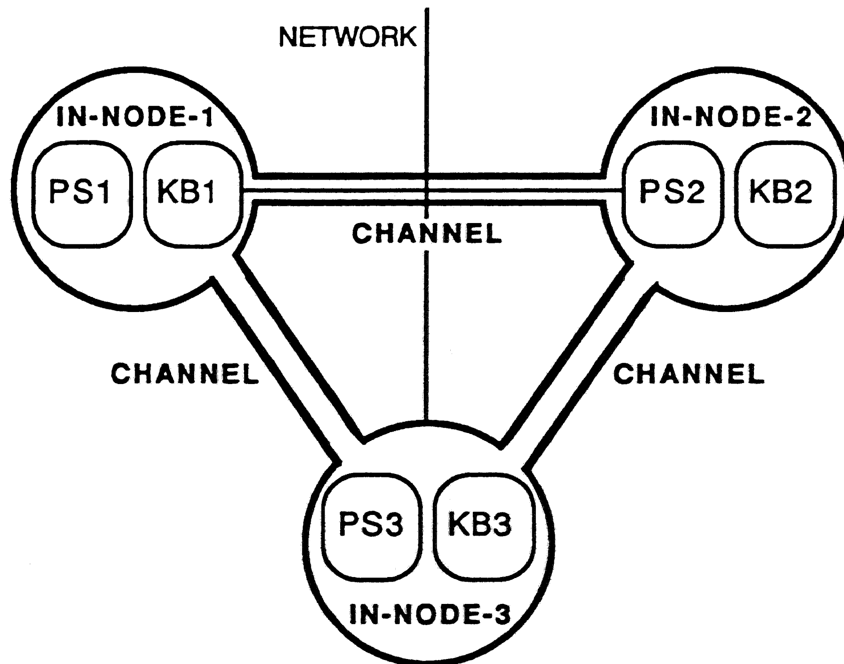


Figure 4-9: Network Layer implementation example

5. Data Layer

5.1 Introduction

Assuming the existence of the Network Layer, we define the Data layer of the manufacturing architecture as the step for the definition of the objects supporting intelligent communication between EMN-nodes.

A decentralized structure, to be coherent, must exchange information between its different EMN-nodes. For that purpose, messages are sent through the network between the EMN-nodes.

The Data Layer provides EMN-nodes with the capability to explicitly request and send information, in the form of schemata, from/to other EMN-nodes. The protocol for requesting and asserting information between EMN-nodes is based on a subset of SQL [6, 28]. In this version, schemata correspond to tables, and slots correspond to fields. Protection is provided at the schema level; access to schemata may be locked and the requesting EMN-node blocked until the schema is unlocked.

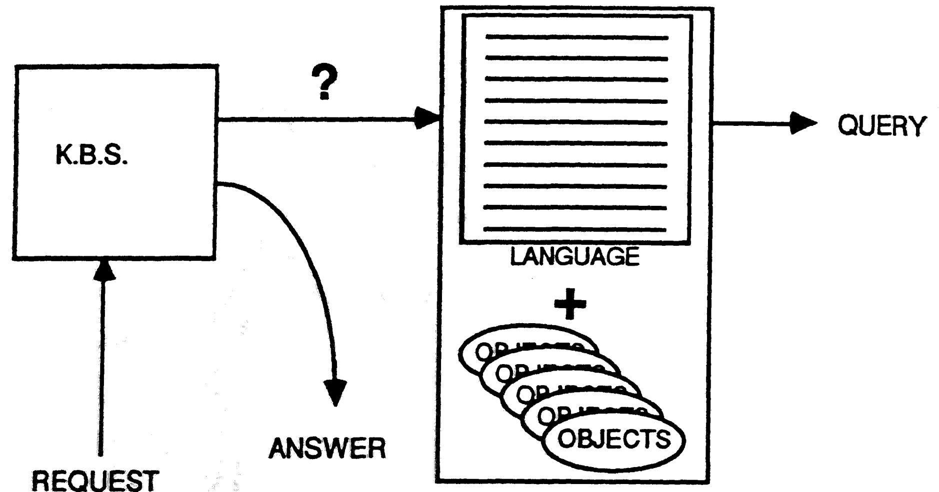


Figure 5-1: Query elements

The Data Layer contains the basic schemata manipulated and the language to express queries for objects belonging to the KBS. These elements are a set of schemata allowing the manipulation of high level information and the definition of a query language used to express an EMN-node request for a specific piece of information in structured way. These requests are defined for KBS objects.

The information flow between EMN-nodes is dependant on each of their needs. These exchanges are carried out to satisfy a request for information not available in the Knowledge Base subsystem of an EMN-node. The request is done at first on a specific type of information (using for example the CRL command GET-VALUE[15]). These exchanges can also be performed for the purpose of

[15]For example, in the current CRL: (GET-VALUE 'Machine 'Capacity) in this case "machine" is the schema name and "capacity" is the slot name. If this value is available in the Knowledge Base subsystem, it is returned; if not, an error message is returned.

consistency. We will see at the next Layer that several kinds of communication processes can be defined. At this Layer we must provide all the elements to support these communication types.

Two sets of elements must be defined (Figure 5-1):

- The schemata manipulated (objects),
- The query language.

The query language allows the expression of a need for information in a generic and understandable way. The objects provide the frame for the information definition and also for the information exchange. Both must take into account several types of communication capabilities and must be compatible (because the query language manipulates the objects and the result of a query is an object).

We have identified four type of objects:

- *Information*: is a reference to the knowledge base. Each information object is represented as a schema or part of a schema (slot).
- *Message*: is defined as a combination of an information need, a producer and a destination. Each message is an instance of the generic message schema.
- *Answer*: is generated to answer a specific message schema. The answer schema includes all the slots of the message schema with, in addition, a status slot (which indicates if the information request is provided or not) and the schema-name-answer slot (whose value is nil or the needed information).
- *Communication schema*: is the schema which provides the capability of the Enterprise Management Network to efficiently acquire and distribute the information. Through the utilization of a dictionary (and of a communication language), each EMN-node has the capability for mutual understanding. The correspondance table allows an efficient information search by defining the owner of the used schemata. The User-table defines the users of the schemata owned by a specific EMN-node.

In the rest of this section we describe these two aspects, objects and language. In addition, we introduce the communication information consistency primitives which maintain coherence between the different tables and schemata used for the communication activity between the EMN-nodes. These tables are defined as slots of the communication schema and are presented in the next section.

5.2 Schemata manipulated

Each EMN-node is an agent of a distributed system. This means each EMN-node has to perform specific activities which represent a part of the global activity of the whole system. Each EMN-node has the capacity to perform its own activity, but there are some limitations in terms of bounded rationality, consistency and coordination. If a system is composed of n EMN-nodes, the realization of n tasks (one by each EMN-node) is not enough to ensure consistency for the global result. Consistency can only be achieved through negotiation and cooperation. Besides, bounded rationality implies distribution of the knowledge in each EMN-node with some restriction on their completeness. All these facts produce a need for a communication activity between the EMN-nodes.

In the Network Layer, we have defined the primitives for a message passing functionality. In this level we are going to use these capabilities by building upon them the frame for more intelligent communication, allowing information search, cooperation, negotiation and coordination.

To communicate means that each EMN-node can exchange information with all the others. Each EMN-node can receive or send messages according to its needs and also according to the requests it receives. We can define what these possible exchanges are. We indicate in figure 5-2 the different transactions which can occur in an EMN-node.

We indicate in this figure two kind of transactions: the request and answer for information and the updating activity.

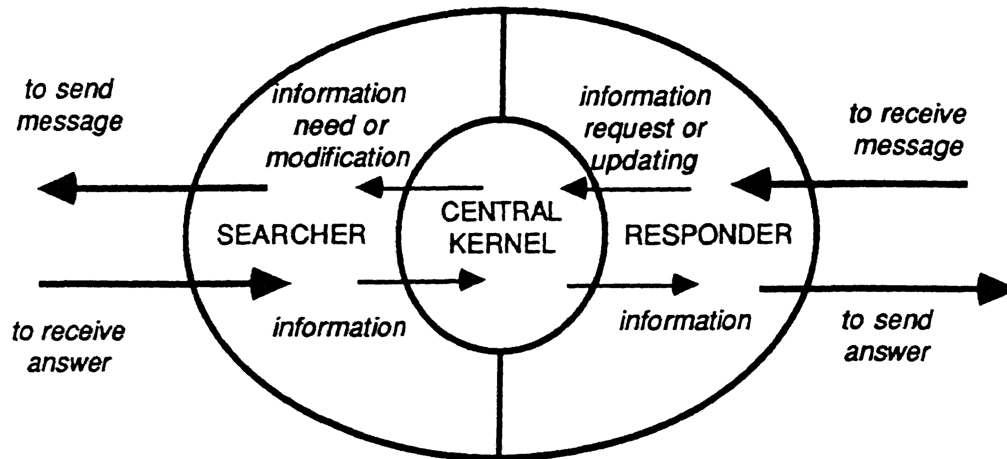


Figure 5-2: Object flow representation

To support these exchanges, an EMN-node manipulates four types of schemata:

- information (or schema),
- message,
- answer and
- the communication schema.

5.2.1 The information schema

An **information** object is a reference to the knowledge base (schema 5-1). It can be an entity or an attribute of an entity. In our case, we consider an information object to be schema, because the elements of the Knowledge Base subsystem are schemata. The problem solving subsystem asks for the value of some slot of some schema. These values are either available or not in its Knowledge Base subsystem. If they are not available, this means that either the schema or the slot (or both) are not present in the Knowledge Base subsystem. In this case, the EMN-node must get the schema from another EMN-node. To get this schema, the searcher generates one (or several) message(s). The responders of the other EMN-nodes generate answers to provide the requested schema to this EMN-node. "Answer" and "message" are schemata. The information object are manipulated by the central kernel. The Knowledge Base subsystem provides them (if they are available) and the problem solving subsystem uses them.

Schema 5-1: Information

Information		
SLOT	FACET	VALUE
Name	*Value:* *Restriction:*	type string
Lexicon	*Restriction:*	type schema-name/slot-name
Lock-status	*Restriction:*	t/nil
Shared-status	*Restriction:*	t/nil

In our definition of what is an information object, we have kept the capability to express it either as a schema or as a slot of a schema. In addition, we introduce the concepts of locked information and shared information. The first aspect concerns the protection of information. The second concerns the behavior of information.

The **locking of an information** can be used when conflicts appear. An example can be: two EMN-nodes want to read and update the same information at the same time. In such a case, a priority is defined between the two EMN-nodes and in between the information is locked. This mechanism provides security in term of information consistency. The concept of **shared information** is defined as follow: an information is said to be shared when it has several owners, i.e., several EMN-nodes allowed to update the information globally along the decentralized system. This concept defines the nature of an information.

5.2.2 The message schema

A **message** object is an information + a destination + a producer (schema 5-2). We have several kind of messages: we have the messages sent due to a request for information coming from the central kernel or we can have updating messages sent because of a Knowledge Base modification. We have created a schema called message to be used by the communication system. This schema is generic. The communication modules, to use it, generate instances of the message schema. For each instance of the message schema, we must fill all the slots. Since a message schema is created due to an information need, an information distribution or an updating activity, we use the name of that information which is either needed or updated or distributed. This information is always a schema.

The first slot, **number**, is a label used to identify the message. In this way we will be able to make the correspondance between an answer and its corresponding message. This label will be used to check if the information request has been resolved or not.

The **type** slot can have four values: info-search, update, distribute-END, distribute-LC or distribute-UT. In this way, we make the distinction between an updating message, a distribution activity and a message created due to an information need. According to the type, the responder which receives a message will generate an answer (if it is the info-search type) or will update its Knowledge Base subsystem (if it is the update type) or will trigger its own distribution process (if it is the distribute-CT or distribute-LC type). The distribute-END message type concerns the deletion of an EMN-node in the global system.

The **priority** slot is filled with a number (0.0 to 1.0). We currently use two values: 0.5 for the search-info message type and 1 for the update message type. But this slot is allowed to receive all kinds of values. This slot will be used when a responder has several messages in its mail box. These messages will be processed according to their priority.

Schema 5-2: Message

Message		
SLOT	FACET	VALUE
Number	*Value:* *Restriction:*	type integer
Type	*Restriction:*	info-search/update/distribute-END distribute-LC/distribute-UT
Priority	*Restriction:*	0.0 to 1.0
Schema-name	*Value:* *Restriction:*	type string
Slot-name	*value:* *Restriction:*	type string
Schema-name-translated	*Value:* *Restriction:*	type string
Slot-name-translated	*Value:* *Restriction:*	type string
Producer	*Value:* *Restriction:*	type local-EMN-node-name
Destination	*Value:* *Restriction:*	type EMN-node-name

The **schema-name** slot is the name of the schema needed by an application and not available in its Knowledge Base sub-system (for example the schema "machine").

The **slot-name** slot is the one needed by an application. We indicate this slot-name just to be sure that the EMN-node which will provide the answer (this means the schema) will include that slot (for example the slot "capacity" of the schema machine).

The **schema-name-translated** slot is the translation of the schema-name into the communication language. As we have seen, each EMN-node has its own internal vocabulary. These vocabularies are different from one EMN-node to another. To solve this problem, and to allow communication, we must use a generic communication language understandable by all the EMN-nodes. When the searcher generates a message to get a schema, the needed schema name is translated into the communication language (to be understood by the responder of the EMN-node which will receive the message). The responder which receives the message translates the needed schema name from the communication language to its internal EMN-node vocabulary.

The **slot-name-translated** slot is the translation of the slot-name into the communication language[16].

The **producer** slot is the local address of the message sender.

The **destination** slot is the destination of the message.

Each time an information object (a schema) is needed by the problem solving subsystem of an EMN-node and not available in its Knowledge Base subsystem, the searcher generates an instance of the message schema. For the updating message it is the same thing. The main problem for this instantiation is to determine the destination and the translation of the slot and schema requested or updated. These slots are completed by the searcher, which determines their values according to heuristic rules and a dictionary. Regarding the distribution activity, it can be triggered for different reasons. The first one is at each EMN-node initialization. The second is at the reception of distribution messages from another EMN-node. The last possibility is for the creation of new schemata.

Schema 5-3: Information-search-message

Information-search-message		
SLOT	FACET	VALUE
Instance	*Restriction:*	message
Number	*Value:*	1
Type	*Restriction:*	info-search
Priority	*Restriction:*	0.5
Schema-name	*Value:*	machine-1
Slot-name	*value:*	capacity
Schema-name-translated	*Value:*	machine-1-translated
Slot-name-translated	*Value:*	capacity-translated
Producer	*Value:*	EMN-node-1
Destination	*Value:*	EMN-node-2

For information search activity, we add the slot: **slot-name**, to be sure that the EMN-node which will provide the answer, will include in it the good schema but also the complete schema (with the needed slot) (we give an example of an information search message schema 5-3). For this type of message, we use the translator function to translate the information need (schema-name slot-name) expressed with the internal EMN-node vocabulary into the generic communication language (schema-name-translated slot-name-translated) understandable by the other EMN-nodes.

---

[16]The communication language we have defined in our specific implementation just supports direct translation of one "word" into another unique "word". This kind of translation is in most cases not enough. The knowledge reconfiguration aspect is not taken into account. In our next implementation we will modify this structure by developing a more sophisticated system allowing us to support a specific communication language dedicated to the nature of the destination EMN-node. We specify these functionalities in the query language we define in the next part.

Schema 5-4: Updating-message

Updating-message		
SLOT	FACET	VALUE
Instance	*Restriction:*	message
Number	*Value:*	2
Type	*Restriction:*	update
Priority	*Restriction:*	1
Schema-name	*Value:*	machine-2
Schema-name-translated	*Value:*	machine-2-translated
Schema-name-updated	*Value:*	(machine-2-translated (ATTRIBUTE (capacity-translated 100-p/h) (type-translated drilling-machine)) (RELATION (is-a machine)))
Producer	*Value:*	EMN-node-1
Destination	*Value:*	EMN-node-2

For updating messages, we add another slot **schema-name-updated**. This slot contains the schema, with its slots and values, which is to be updated in the other EMN-nodes (we give an example of an updating message schem: 5-4). In this kind of message, the translator function is used for the two slots: schema-name-translated and schema-name-updated. We must translate the value of these slots into the generic communication language because this information must be used by other EMN-nodes having not necessarily the same internal vocabulary.

Schema 5-5: LC-distribution-message

LC-distribution-message		
SLOT	FACET	VALUE
Instance	*Restriction:*	message
Number	*Value:*	3
Type	*Restriction:*	distribute-LC
Priority	*Restriction:*	1
Local-channel	*Value:*	(local-channel-1 (RELATION (instance DKC-channel)) (ATTRIBUTE (mbox-name mb1) (semaphore-name sem1) (node EMN-node-1) (key-words kw1 kw2 kw3)))
Producer	*Value:*	EMN-node-1
Destination	*Value:*	EMN-node-2

For distribution activity, according to its nature, we add a specific slot. If we distribute the local-channel schema, the message type will be "distribute-LC" and the message created will contain a slot called **local-channel** having as value the local-channel schema (we give an example of a local-channel distribution message schema 5-5. In this example, EMN-node-1 provides its local channel schema to EMN-node-2.).

Schema 5-6: UT-distribution-message

UT-distribution-message		
SLOT	FACET	VALUE
Instance	*Restriction:*	message
Number	*Value:*	4
Type	*Restriction:*	distribute-UT
Priority	*Restriction:*	1
UT	*Value:*	((article-1-translated (EMN-node-2 EMN-node-3) (article-2-translated (EMN-node-2 EMN-node-4))
Producer	*Value:*	EMN-node-1
Destination	*Value:*	EMN-node-2

For the distribution of local channels, no use is done of the generic communication language. Because, the information distributed concerns mainly file addresses which have unique names. If we distribute the user-table, which is defined as a slot of the communication schema, the message type will be "distribute-UT" and it will contain a slot called UT having value the user-table (we give an example of a user-table distribution message schema 5-6. In this example, EMN-node-1 sends this distribution message to EMN-node-2). The distribution of the UT needs the utilization of the translator and of the generic communication language for the same reason as for the updating activity.

Schema 5-7: distribution-END-message

distribution-END-message		
SLOT	FACET	VALUE
Instance	*Restriction:*	message
Number	*Value:*	4
Type	*Restriction:*	distribute-END
Priority	*Restriction:*	1
Producer	*Value:*	EMN-node-1
Destination	*Value:*	EMN-node-2

The distribute-END messages type are generated when an EMN-node is removed from the global system. In such a case, it informs the other EMN-nodes about its deletion. This message type

contains only the EMN-node name to be deleted. The EMN-nodes can accordingly remove the channel and the value of this EMN-node from all the tables it is member (we give an example of a distribute-END message type in figure 5-7. In this example, EMN-node-1 is removed from the global system and informs EMN-node-2 of this deletion).

5.2.3 The answer schema

The searcher of an EMN-node sends messages to the responder of the other EMN-nodes. The responder, depending on the messages received, must send an answer. This answer is sent to the responder of the EMN-node which has generated the message. To do this, the first responder generates an instance of a generic answer schema. This schema has several slots which must be completed by the responder according to the corresponding message and the available schema within its Knowledge Base subsystem. The generic **answer** schema can be defined by the schema 5-8. The answer schema is the basic schema of the responder of this communication system. For each instance, the responder must fill all the slots of this schema.

Schema 5-8: Answer

Answer		
SLOT	FACET	VALUE
Number	*Value:* *Restriction:*	type integer
Status	*Restriction:*	nil/t/locked
Schema-name-translated	*Value:* *Restriction:*	type string
Slot-name-translated	*Value:* *Restriction:*	type string
Schema-name-answer	*Value:* *Restriction:*	type lispobj
Producer	*Value:* *Restriction:*	type local-EMN-node-name
Destination	*Value:* *Restriction:*	type EMN-node-name

The first slot **number** links a message and its answer. The value of this slot is the same as the corresponding message.

The **status** slot is filled based upon the ability to satisfy the request. If the schema and the slot are available in the Knowledge Base subsystem of the EMN-node, then the value of this slot is t. If not the value is nil. This slot is used by the searcher which generated the message. From the value of the status slot of the answer schema, the searcher can decide whether its information request has been satisfied or not. Based on this status, it either provides the schema to the problem solving subsystem (if the status value is t) or it generates one or more new messages. Another possibility is to have as the status value: locked. This means that the EMN-node possesses the schema, but is not able to provide it due to a lock applied to the schema.

The **schema-name-translated** slot is directly derived from the same slot of the corresponding message schema.

The **slot-name-translated** slot is also directly derived from the same slot of the corresponding message schema.

The main slot of the answer schema is the **schema-name-answer**. This slot contains the answer: the schema requested or nothing, according to its availability in the Knowledge Base.

The last two slots **producer** and **destination** are the reversed values as those of the corresponding message.

Schema 5-9: Answer-example

Answer-example		
SLOT	FACET	VALUE
Instance	*Restriction:*	answer
Number	*Value:*	1
Status	*Restriction:*	nil
Schema-name-translated	*Value:*	machine-1-translated
Slot-name-translated	*Value:*	capacity-translated
Schema-name-answer	*Value:*	nil
Producer	*Value:*	EMN-node-2
Destination	*Value:*	EMN-node-1

We give in schema 5-9 an example of an instance of the answer schema. This answer is the one generated by the EMN-node-2 towards the EMN-node-1 in response to the information-search-message of the figure 5-3. In that case, the EMN-node-2 does not possess the information (status nil).

5.2.4 The communication schema

Each node is independent and autonomous. But to ensure coordination within the global structure, it is necessary to update all the individual subsystems. The EMN-nodes must exchange messages to get information not available in their own Knowledge Base subsystem (figure 5-3). For this purpose the searchers send messages to the responders of the other EMN-nodes.

The main problem in this activity is knowing the schema needed or modified by the central kernel, to determine the destination of the message. The central kernel provides the schema to acquire or to update to the searcher. The searcher must build an instance of the message schema by defining all the elements of the message. The determination of the destination of the message is the main difficulty of this activity.

The other problem concerns knowledge representation and understanding between the different EMN-nodes of our structure. The vocabulary used to identify a piece of information from one EMN-node to another is different. The information represented in an EMN-node by a single schema,

identified by a unique name, can represent, in another EMN-node, a set of schemata. This identification and re-construction of knowledge appears to be one of the main difficulties. To help us in this task and also to provide the capability of a more intelligent communication activity, we define a **communication** schema.

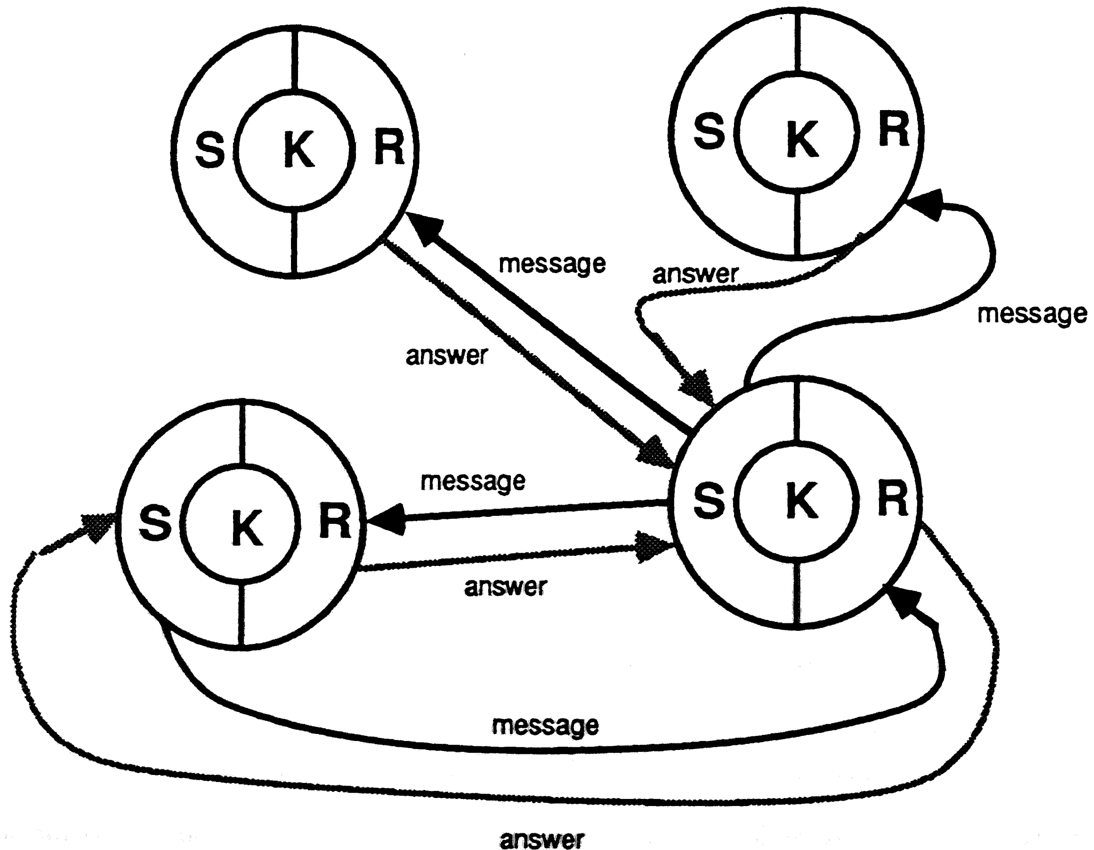


Figure 5-3: Decentralized Knowledge Craft running

A communication system includes some generic and some specific parts. The owner address of a schema and the internal vocabulary of an application (EMN-node) are specific. To be able to exchange information between the EMN-nodes of Decentralized Knowledge Craft (DKC), it is necessary for each EMN-node to know where to get its needed information and to understand the messages. For this purpose we have created a schema called the **communication** schema (schema 5-10).

The first slot, **correspondance-table**, provides the owner address of the schemata which are used by an EMN-node but not available in its Knowledge Base subsystem. We assume that each schema has only one owner except for the shared schemata which have several owners. In the case of shared schemata, we use the "shared-schemata" slot to indicate their list of owners. Each shared-schema has several owners and one main owner. The main owner is the only one allowed to send global updating messages to the users of a shared schema. The other owners must go through the main owner to send global updating messages. An EMN-node is the main owner of a shared schema when that schema appears both in the User-table slot and shared-schemata slot.

The second slot, **user-table**, is used for the updating activity. The local EMN-node has the responsibility for updating the schemata of which it is the owner. It has the responsibility to transfer the updated schemata to the other users. In this slot, we indicate the schemata owned by the local EMN-node and for each of them we indicate the users (the other EMN-nodes which use that specific schema).

Schema 5-10: Communication

Communication		
SLOT	FACET	VALUE
Correspondance-table	*Value:* *Restriction:*	type (information EMN-node-name)*
User-table	*Value:* *Restriction:*	type (information (EMN-node-name [, EMN-node-name, ...]))*
Locked-schemata	*Value:* *Restriction:*	type (information (EMN-node-name [, EMN-node-name, ...]))*
Shared-schemata	*Value:* *Restriction:*	type (information (EMN-node-name [, EMN-node-name, ...]))*
Updated-schemata	*Value:* *Restriction:*	type information*
Dictionary	*Value:* *Restriction:*	type (information string)*
Local-address	*Value:* *Restriction:*	type local-EMN-node-name

The third slot, **locked-schemata**, indicates the list of information[17] object which are locked, i.e. the EMN-nodes cannot access either for read or write. The locking mechanism is applied only to the information object which are shared by two or more EMN-nodes. The regular policy for information updating is the single owner policy, i.e. only one EMN-node is able to globally modify and update a specific piece of information in the entire decentralized system. But, all the EMN-nodes can modify locally the information they use. For shared information, i.e. information object have several owners which are able to globally modify or update them, this policy is modified: each owner can update a shared schema but the global update is done by the main owner, which is unique. This global update can be triggered either by the main owner or by the others, i.e., if the main owner modifies a shared schema, it updates it globally; if another owner modifies a shared schema, it sends an updating message to the main owner which executes the global update according to this received updating message. For these special shared schemata, to keep consistency, we use a locking mechanism. We must lock these shared objects when two owner EMN-nodes want to read and modify the same information globally. This slot contains the list of locked schemata and, for each locked schema, the EMN-node name which causes the lock.

[17]In our current implementation, we consider information object to be schemata.

The fourth slot, **shared-schemata**, indicates the list of information object which have several owners, i.e., several EMN-nodes are able to update globally this information. This slot indicates the list of shared schemata and their owners.

The fifth slot, **updated-schemata**, indicates the list of information object which has been modified between two updating sequences. Each time an information object is modified by an EMN-node, if the information object is owned by this EMN-node, we add the information object as another value of this slot.

The sixth slot, **dictionary**, establishes a correspondance between the internal vocabulary of an EMN-node and the generic communication language.

The last slot, **local-address**, indicates the address of the local EMN-node.

Schema 5-11: Communication

Communication		
SLOT	FACET	VALUE
Correspondance-table	*Value:*	(operation-1 agent-3) (operation-2 agent-3) (operation-4 agent-3) (machine-1 agent-2) (machine-6 agent-2)
User-table	*Value:*	(article-2 (agent-2 agent-3)) (article-3 ()) (article-1 (agent-3)) (article-2 (agent-3))
Locked-schemata	*Value:*	
Shared-schemata	*Value:*	(article-3 (agent-3)) (article-1 (agent-2 agent-3))
Updated-schemata	*Value:*	
Dictionary	*Value:*	
Local-address	*Value:*	'agent-1

We give an example of a communication schema (schema 5-11). This schema is the one created for the agent-1 EMN-node. It contains the schema owned, shared and used by this EMN-node. In this example, the name of the EMN-node is indicated in the local-address slot: agent-1. This communication schema indicated the schemata owned by this EMN-node. They are indicated in the User-Table slot. Besides, for each owned schema, its list of user is defined. In the Correspondance-Table slot, we indicate the schemata used by EMN-node agent-1. For each schema used by this EMN-node, we define its owner. The Shared-Schemata slot indicates schemata which are owned by several EMN-nodes (agent-1 is one of these owners). The agent-1 is the main owner of the two schemata which are shared by several EMN-nodes because they also are indicated in its User-Table. If they were indicated in its Correspondance-Table, the main owner would have been the EMN-node indicated in this table. All the other slots have no values at the beginning because they are filled during the EMN-node running and used by the different communication sequences.

As we have seen previously, one of the main difficulties for communication is the determination of the destination of a message.

A message is sent to a destination to get needed information or to update the Knowledge Base subsystems of the other EMN-nodes. The destination is another EMN-node of the decentralized system (DKC). Each message schema has a slot called "destination". This slot indicates where to send the message. For determining this destination, we have several possibilities: direct communication, friend selection or broadcasting. In the searcher, when we build a message, we try at first to use direct communication. For this purpose, we use the correspondance-table for information search and the user-table for the updating activity. The correspondance-table is a slot of the communication schema which gives the owner address of each schema used by an EMN-node (application) and not available in its domain modeling sub-system. The values of this slot are lists (pairs): the first element of a pair indicates the schema name and the second one the address of the schema owner. We must build one correspondance-table per EMN-node. Similarly for the user-table, for each schema owned by the local EMN-node, we indicate the list of users (EMN-node addresses).

Since each EMN-node has its own Knowledge Base subsystem, we have a lot of redundancies between them. In addition, since the utilization of these entities is different, their contents can be different. The vocabulary in each EMN-node can be different. Because of this, we must have a standard communication language to exchange messages between the EMN-nodes. To use the content of the messages, we must translate them into the local EMN-node language. For this we use a **Translator** (T) which is a part of the Knowledge Base Manager. So, we have two vocabulary types:

- An internal one: used by the local central kernel (problem solving subsystem, knowledge base subsystem and knowledge base manager). This vocabulary is specific to an application (EMN-node).
- A communication language: used for communication between the EMN-nodes. It is a standard vocabulary used by the responder and searcher of each EMN-node.

The dictionary (or translator) is another slot of the communication schema. It produces a correspondance between the internal vocabulary of an EMN-node and the communication language. The values of this slot are lists (pairs): the first element of a pair indicates the schema name (using the internal vocabulary of an application) and the second one indicates its translation into the communication language. We must build one dictionary per EMN-node. This dictionary structure is the first step in the design of the communication system. This kind of 1-to-1 direct correspondance is not enough in all cases. For example, a schema in an EMN-node can correspond to several schemata in another. We need to define a mapping function able to establish the connection between schemata of different EMN-nodes. Such a function must be able to split up or gather schemata according to the specific context of an EMN-node.

This communication schema represents the specific part of the decentralized communication system, which must be determined for each EMN-node. All the values of the slots are dedicated to an EMN-node. We must determine the schemata needed by the EMN-node and define where is it possible to get them. In addition, the dictionary is dedicated to the EMN-node according to the information it manipulates and exchanges with the other EMN-nodes. The user list which is the schemata owned by an EMN-node, is also specific.

5.3 Information consistency checking primitives

In the Data Layer, schemata are defined to support the different sequences defined in the Information Layer. In the previous section, three different tables have been presented:

- the Correspondance Table (CT): a list of pairs. The first element of the pair is the name of a schema used by the EMN-node but owned by another EMN-node. The second element of this pair is the EMN-node name owner of that schema.
- The User Table (UT): a list of pairs. The first element is a schema name owned by the EMN-node. The second element is a list of EMN-node names which use that schema but are not owner of that schema.
- The Shared Schemata Table (SST): a list of pairs. The first element is a schema name. The second element is a list of EMN-node names which own that schema together with the local EMN-node. For the shared schemata, we earlier defined the concept of main owner. To share a schema means that all the owners are able to modify that schema. But only the main owner can send the updating messages to all the users/owners of that schema. The main owner of a shared schema is the EMN-node which has this schema in its User Table. All the other owners of this schema must have it in their Correspondance Table.

These three tables are used for the communication sequences we will define in the Information Layer. As we can see, there exists some overlap between these tables (for example between the SST and the CT or UT). Besides, we will define in the Information Layer some other schemata called the class schemata which also manipulate information already included in these tables. These schemata are defined at the Information Layer and used by the distribution sequence. The class schemata define set of schemata with their users completed by key-words. These schemata are owned by the EMN-node. To keep consistency between all these tables and schemata, we define some primitives which allow the completion of one table using another. We also define primitives which check the internal consistency of one table (the repetition and the validity of the information contained in the table).

The table consistency checking primitives are the following (these ones allow an consistency checking of a specific table in terms of redundancy and authorized values):

- check-CT-consistency: checks the consistency of the Correspondance Table (this one is used for information search, it includes a list of schemata and their owner).
- check-UT-consistency: checks the consistency of the User Table (this one is used for schemata update, it includes a list of schemata owned by an agent and for each schema its list of users).
- check-SST-consistency: checks the consistency of the Shared Schemata Table (this one is used for the update of the shared schemata, it includes the list of shared schemata and their owners).
- check-class-consistency: checks the consistency of the different instances of the class schema (these schemata are used for the distribution sequence, they include key-words, members and users)[18].
- ckeck-key-words-consistency: checks the validity and non redundancy of the values of the key-words slot of each class schema and of each channel schema.

---

[18]The class schema is defined in the Information Layer.

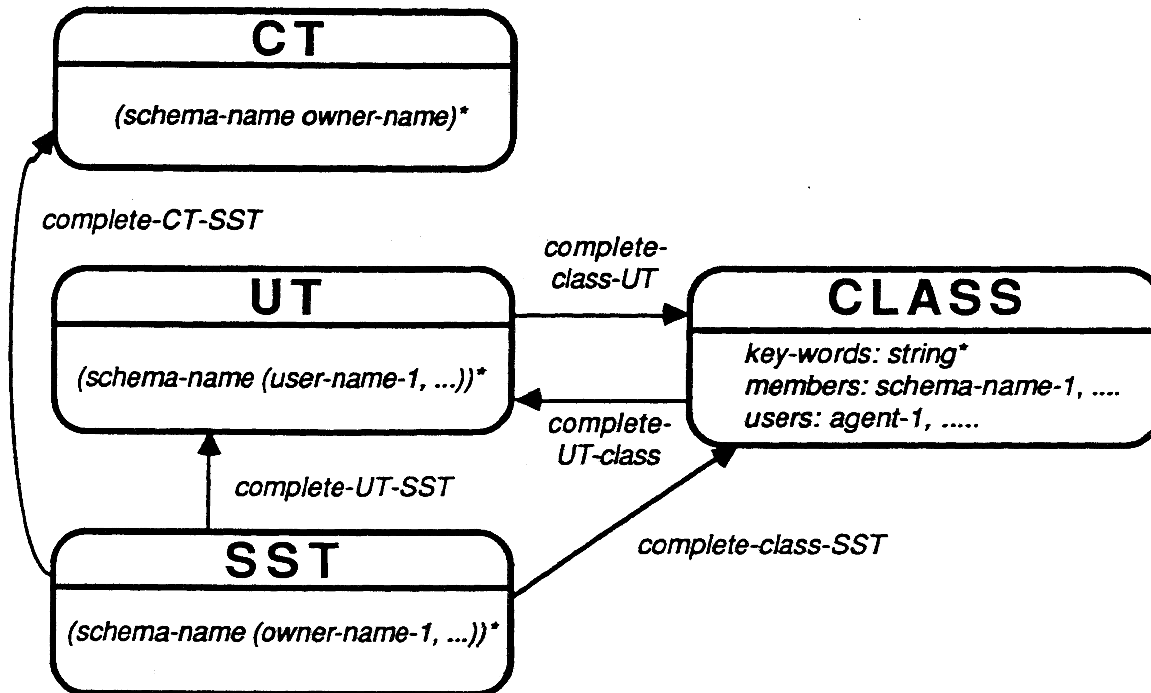


Figure 5-4: Mutual table consistency checking

In addition, mutual consistency and completion checking functions have been defined (figure 5-4). They allow the completion of one table, using the information of another table:

- complete-class-UT: completion of class schemata using the User Table. This function is very important regarding to the information distribution at the EMN-node initialization. The class schemata contains the owned schemata of an EMN-node, their users and in addition some key-words attached to each class schema. This function allows to complete or to create new class schemata according to the content of the User Table.
- complete-UT-class: completion of the User Table using the class schemata. This function checks for the coherence between the User Table and the existing class schemata. For each class member, the User Table must have the corresponding schema. In addition, the users of a specific class member must also be defined as users of that particular schema in the User Table.
- complete-class-SST: completion of the class schemata using the Shared Schemata Table.
- complete-UT-SST: completion of the User Table using the Shared Schemata Table.
- complete-CT-SST: completion of the Correspondance Table using the Shared Schemata Table.
- complete-class-key-words: completion of the class schemata members using the key-words concept attached to each EMN-nodes of the system.
- complete-key-words: this function allows to complete the key-words slot of the local-channel schema. It uses the values indicated in the key-words slot of the different class schemata owned by each EMN-node.

All these primitives are used in the initialization of the EMN-nodes and then, depending on the modifications which occur either to the communication schema or the class schemata, some of them can be triggered.

5.4 Query language

In the previous section, we defined the objects manipulated and needed for an intelligent communication activity. In this part, we define the language which manipulates these objects to express in a structured and unambiguous way requests for information defined in an EMN-node. To define the different query possibilities, we use an SQL-type query language. SQL is a language defined to access a relational data structure. The general level of the language is comparable to that of the relational algebra. SQL (Structured Query Language) is more than a query language. It provides not only retrieval functions but also a full range of update operations, and many other facilities [6, 28].

In our case, we do not use the complete SQL language to support information queries between EMN-nodes. We must adapt its syntax to our purpose. The elements manipulated by SQL are tables, rows and columns (fields). We manipulate schemata and slots.

To use SQL, we have established a correspondance between: table -----> schema and field ----> slot.

In this section, we present the basic constructions for a query. More sophisticated queries can be expressed by combining these basic ones. Our presentation starts with the simple request and goes to sophisticated and precise requests. We conclude this part by describing the locking and unlocking mechanisms.

We will use the same example data base all through this text. Consider 3 schemata: the machine schema, the machine-1 schema and the machine-2 schema. The relations between these schemata are described in the figure 5-5.

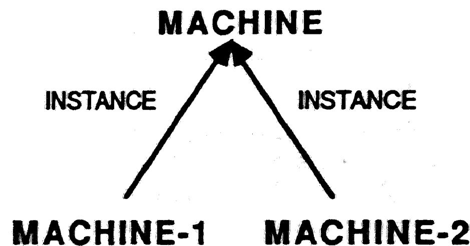


Figure 5-5: Information flows representation

```
{{ MACHINE
      INSTANCE+INV: Machine-1, Machine-2
      IDENTIFICATION:
      CAPACITY:
      COLOR:            }}

{{ MACHINE-1
      INSTANCE: Machine
      TYPE: drilling-machine
      IDENTIFICATION: DM1
      CAPACITY: 100-p/h
      COLOR: red        }}

{{ MACHINE-2
      INSTANCE: Machine
      TYPE: milling-machine
      IDENTIFICATION: MM2
      CAPACITY: 50-p/h
      COLOR: red        }}
```

5.4.1 Complete schema request

To express a complete schema request we have to use this command:

```
SELECT *
FROM schema-name;
```

The result is a request for the entire schema. "*" replaces all the slot-names of this schema-name. We could replace the "*" with the names of all the slots of the needed schema.

Example:

Query: *get the machine-1 schema*

```
SELECT *
FROM machine-1;
```

Result:

```
{{  MACHINE-1
        INSTANCE: Machine
        TYPE: drilling-machine
        IDENTIFICATION: DM1
        CAPACITY: 100-p/h
        COLOR: red       }}
```

5.4.2 Simple retrieval

If we want a more specific request for a slot or a set of slots, we must use the following syntax:

```
SELECT slot-name [, slot-name ...]
FROM schema-name;
```

This function selects one or several slots (the listed ones) in the given schema (schema-name).

Example:

Query: *get the capacity of the machine-1*

```
SELECT capacity
FROM machine-1;
```

Result:

```
{{  MACHINE-1
        CAPACITY: 100-p/h   }}
```

5.4.3 Qualified retrieval

We can improve our selection if, for example, we know one slot value of the needed schema. For this purpose, we can use the WHERE clause.

```
SELECT slot-name [, slot-name] ...
FROM schema-name
WHERE slot-name= 'value;
```

Example:

Query: *get the capacity of the machine whose type is drilling-machine*

```
SELECT capacity
FROM machine
WHERE type = 'drilling-machine;
```

Result:

```
{{  MACHINE-1
        CAPACITY: 100-p/h
        TYPE: drilling-machine }}
```

In this case the result is just one schema. But if, for example, the machine schema had several instances having a type equal to drilling-machine, the result would be a set of schemata.

In the WHERE clause, we can have a condition on either the slot-name or the schema-name. Also, we have no limitation on the number of conditions.

We can add other conditions using the AND term:

```
SELECT slot-name [, slot-name] ...
FROM schema-name
WHERE condition-1
AND condition-2;
```

Example:

Query: *get the capacity of the machine whose type is drilling-machine and where the capacity is more than 70-p/h*

```
SELECT capacity
FROM machine
WHERE type = 'drilling-machine
AND capacity > 70-p/h;
```

Result:

```
{{ MACHINE-1
       CAPACITY: 100-p/h
       TYPE: drilling-machine }}
```

To improve the different conditions applied to both schemata and slots, we can use the comparison and Boolean operators. The WHERE clauses can be very sophisticated and include a long set of restrictions on the information we need to acquire. The reason for such a detail is to allow the user to get exactly the needed information without redundancy.

WHERE CLAUSE CONDITIONS

Condition	Symbol
Equal	=
Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=
Not equal	<>
Between	BETWEEN
Partial equality	LIKE
Equal to one item in a list	IN
Negation	NOT
Logical connector	AND
Logical connector	OR

The conditions on schemata and slots we can create can use a combination of several of these operators. The created WHERE clause conditions can reach all kinds of sophisticated levels.

The operators described so far allow comparison between schemata or slots values. But since knowledge could be described in each EMN-node of a computer system in different ways, we must add operators providing the capability to combine the resulting schemata and slots. For this purpose, we can use first the arithmetical operators.

OPERATORS

Operator	Symbol
Addition	+
Substraction	-
Multiplication	*
Division	/
Count	COUNT
Maximum	MAX
Minimum	MIN
Average	AVG
Sum	SUM

All these operators are described in the SQL language. For their definitions refer to [28]. These operators only combine values. We need something more sophisticated, which allows us to combine schemata and slots.

This collection of operators allows slots to be combined and acted upon in various ways. The result of such operations are, in effect, new values for slots which are not held in storage within the system, but which can be brought forth or created at any time. In this way, we can adapt the values of a slot according to our purposes without changing its real value, just its units.

Example:

Query: *get the capacity of the machine whose type is drilling-machine and where the capacity is more than 70-p/h give the capacity in lot/h (1 lot = 20 parts)*

```
SELECT (capacity/20)
FROM machine
WHERE type = 'drilling-machine
AND capacity > 70-p/h;
```

Result:

```
{{ MACHINE-1
       CAPACITY: 5-lot/h
       TYPE: drilling-machine }}
```

With the different commands we have already defined, we are able to select a part of a schema: for example to get the capacity of a machine.

Query: *get the capacity from the machines whose type is drilling-machine*

```
SELECT capacity
FROM machine
WHERE type = 'drilling-machine;
```

Result:

```
{{ MACHINE-1
      CAPACITY: 100-p/h
      INSTANCE: Machine
      TYPE: drilling-machine }}
```

To complete our example, we define a new instance of the machine schema: machine-3.

```
{{ MACHINE-3
      CAPACITY: 80-p/h
      COLOR: green
      IDENTIFICATION: DM2
      INSTANCE: Machine
      TYPE: drilling-machine }}
```

The result of the previous command is a schema issued from one unique schema. Such a query is not enough if, for example, we intend to get the total available capacity of the drilling-machines. In this case the query will have the following structure:

Query: *get the total capacity from the machines whose type is drilling-machine*

```
SELECT SUM(capacity)
FROM machine
WHERE type = 'drilling-machine;
```

Result:

```
{{ MACHINE
      SUM-CAPACITY: 180-p/h
      TYPE: drilling-machine }}
```

The operators manipulate values. Until now, we have just expressed requests for single schemata. But one of the problem to be able to support with our query language is to have the capability to manipulate, to modify, to group or disjoin knowledge. We must be able to combine several schemata into one representing the needed information. In SQL, there exists three other kinds of functions which allow such a combination:

- the ordering of information,
- joining of information,
- queries within queries.

5.4.4 Retrieval with ordering

The command ORDER BY causes the selection of values in a specific order. This ordering is defined on the values of a slot-name in one or two directions: ascending (ASC) or descending (DESC).

```
SELECT slot-name1, slot-name2
FROM schema-name
WHERE slot-name3 = "value"
ORDER BY slot-name2 DESC;
```

The ordering can also be done on the slots of a schema. The result is a re-organization of the order of the slots in one schema. This ordering can also have two directions.

```
SELECT *
FROM schema-name
WHERE slot-name3 = "value"
ORDER BY schema-name DESC;
```

Example:

Query: *get the machine schema whose type is a drilling-machine, where the capacity is more than 70-p/h in descending order*

```
SELECT *
FROM machine
WHERE type = 'drilling-machine
AND capacity > 70-p/h
ORDER BY machine DESC;
```

Result:

```
{{ MACHINE-1
     CAPACITY: 100-p/h
     COLOR: red
     IDENTIFICATION: DM1
     INSTANCE: Machine
     TYPE: drilling-machine }}
```

Query: *get the names of the instances of the machine schema in descending order*

```
SELECT instance+inv
FROM machine
ORDER BY machine DESC;
```

Result:

```
{{ MACHINE
     INSTANCE+INV: machine-1,
       machine-2, machine-3 }}
```

We can also use the command GROUP BY for ordering. This command causes instances of a schema to be grouped and the groups to be considered as a whole. This command should follow the WHERE clause within the SELECT statement, or immmediately follow the schema-name if no WHERE clause is specified. The grouping can be done according to all kinds of characteristics (type, color, capacity, etc.).

```
SELECT COUNT (slot-name)
FROM schema-name
WHERE ...
GROUP BY ....;
```

Example:

Query: *get machine quantity per type whose color is red*

```
SELECT capacity COUNT(instance)
FROM machine
WHERE color= 'red
GROUP BY type;
```

Result:

```
{{ MACHINE-1
CAPACITY: 100-p/h
COLOR: red
INSTANCE: Machine
TYPE: drilling-machine }}

{{ MACHINE-2
CAPACITY: 50-p/h
COLOR: red
INSTANCE: Machine
TYPE: milling-machine }}
```

Query: *get the total number of machines whose type is drilling-machine*

Result:

```
SELECT COUNT(instance)
FROM machine
WHERE type = 'drilling-machine
```

```
{{ MACHINE
      COUNT-INSTANCE: 2
      TYPE: drilling-machine }}
```

5.4.5 Retrieval from more than one schema

To demonstrate the mechanism which underlies the join, let's begin with a retrieval involving several schemata:

```
SELECT schema-name1.slot-name1, schema-name2.slot-name2
FROM schema-name1, schema-name2
```

The result of such a command is the creation of a new schema having 2 slots: slot-name1 and slot-name2. Each one of these slots comes from a different schema.

Such a query is incomplete because we need to define the name of the new schema including the two selected slots. In fact, we must store these slots into a schema. So we must complete the previous request by defining where to store the information (in our case in schema-name-3).

```
SELECT schema-name1.slot-name1, schema-name2.slot-name2
FROM schema-name1, schema-name2
TO schema-name-3
```

The reason for such a query is to allow the re-construction and adaptation of knowledge to our purpose. Using such a structure, we can redefine knowledge according to the specific needs of a specific EMN-node.

Example:

Query: *join the machine-1 identification and the machine-3 identification in the query schema*

Result:

```
SELECT machine-1.identification,
       machine-2.identification
FROM machine-1, machine-2
TO query;
```

```
{{ QUERY
      MACHINE-1.ID: DM1
      MACHINE-2.ID: MM2 }}
```

We can also use the command UNION which creates a request for information derived from two schemata.

```
SELECT .... UNION SELECT ...
```

The result of such a query is a schema containing all the slots selected from the first schema and all the slots selected from the second schema. In this case we create a schema called UNION.

5.4.6 Retrieval involving queries within queries

All equations have a left side and a right side. A subquery is a full select statement used as the right-side expression within a WHERE clause. It is "sub-" in that it is a query which is subordonate to, or inside of, another query:

```
SELECT slot-name1
FROM schema-name1
WHERE slot-name2 = (SELECT slot-name3
                    FROM schema-name2
                    WHERE slot-name3 = 'value)
```

For the first WHERE condition, we can use all the kinds of operators described previously.

In this case, there is a restriction on the utilization of such a structure. We must have coherence between "slot-name2" and the "SELECT" form to satisfy the equality. This means we must have a one-to-one value correspondance.

Example:

Query: *get the identification of the existing machines*

```
SELECT identification
FROM machine
WHERE instance+inv IN
       (SELECT instance+inv
        FROM machine);
```

Result:

```
{{ MACHINE-1
       IDENTIFICATION: DM1 }}

{{ MACHINE-2
       IDENTIFICATION: MM2 }}

{{ MACHINE-3
       IDENTIFICATION: DM2 }}
```

If one of the comparison operators precedes a subquery, the subquery must return only one value. If a comparison operator is used alone and the subquery returns multiple values, the structure of the query will be inadequate. In above example, we used the operator IN to avoid this problem. But IN can only be substituted if we are testing for equality.

If we want to improve the subquery structure by providing a capability for multiple values between the slot-name2 and the select statement, we can use some operators. We can use comparison operators with multiple-valued sub queries if we follow the operator with one of the words ANY, SOME or ALL.

```
SELECT ... WHERE A = ANY (SELECT ...)

SELECT ... WHERE A < ALL (SELECT ...)

SELECT ... WHERE A > SOME (SELECT ...)
```

Example of retrieval using ANY:

```
SELECT UNIQUE slot-name1
FROM schema-name1, schema-name2
WHERE schema-name1.slot-name2 = schema-name2.slot-name2
AND schema-name2.slot-name3 = 'value
```

This is equivalent to:

```
SELECT slot-name1
FROM schema-name1
WHERE slot-name2 =ANY      (SELECT slot-name2
                                FROM schema-name2
                                WHERE slot-name3 = 'value)
```

The operators <=ANY, >=ANY, >ANY, <ANY, not=ANY are analogously defined.

IN is equivalent to =ANY.

5.4.7 Locking and unlocking mechanism

Locking is a mechanism for protecting transactions from interference by other, concurrently executing transactions, i.e., the presence of one transaction in the system should not cause some other transaction to produce incorrect results.

We have two possibilities: we can lock either a complete schema or specific slots of a schema.

Complete locking schema request:

```
LOCK *
FROM schema-name;
```

The result is a request for locking the entire schema. "*" replaces all the slot-names of this schema-name. We can replace the "*" by the names of all the slots of the needed schema.

Example:

Query: *lock the machine-1 schema* — **Result:**

```
LOCK *                    {{ COMMUNICATION-SCHEMA
FROM machine-1;                LOCKED-SCHEMATA: Machine-1 }}
```

Simple retrieval:

If we want a more specific locking request on a slot or a set of slots, we must use the following syntax:

```
LOCK slot-name [, slot-name ...]
FROM schema-name;
```

Example:

Query: *lock the capacity slot of the machine-1* — **Result:**

```
LOCK capacity       {{ COMMUNICATION-SCHEMA
FROM machine-1;           LOCKED-SCHEMATA: (Machine-1 capacity) }}
```

The unlocking mechanism has the same structure. But in this case we use the UNLOCK function.

Complete unlocking:

```
UNLOCK *
FROM schema-name;
```

Partial unlocking:

```
UNLOCK slot-name [, slot-name ...]
FROM schema-name;
```

5.5 Data Layer example

As we can see, the objects needed for a more sophisticated communication capability have been added to each EMN-node. These capabilities will be used in the upper levels for problem solving negotiation and also for the information search and updating sequences. The elements we add at this layer concern the internal structure of an EMN-node. We define the schemata to support the different sequences of the Information Layer: updating, distribution and information search. If we implement these new elements in our example, figure 5-6 becomes:

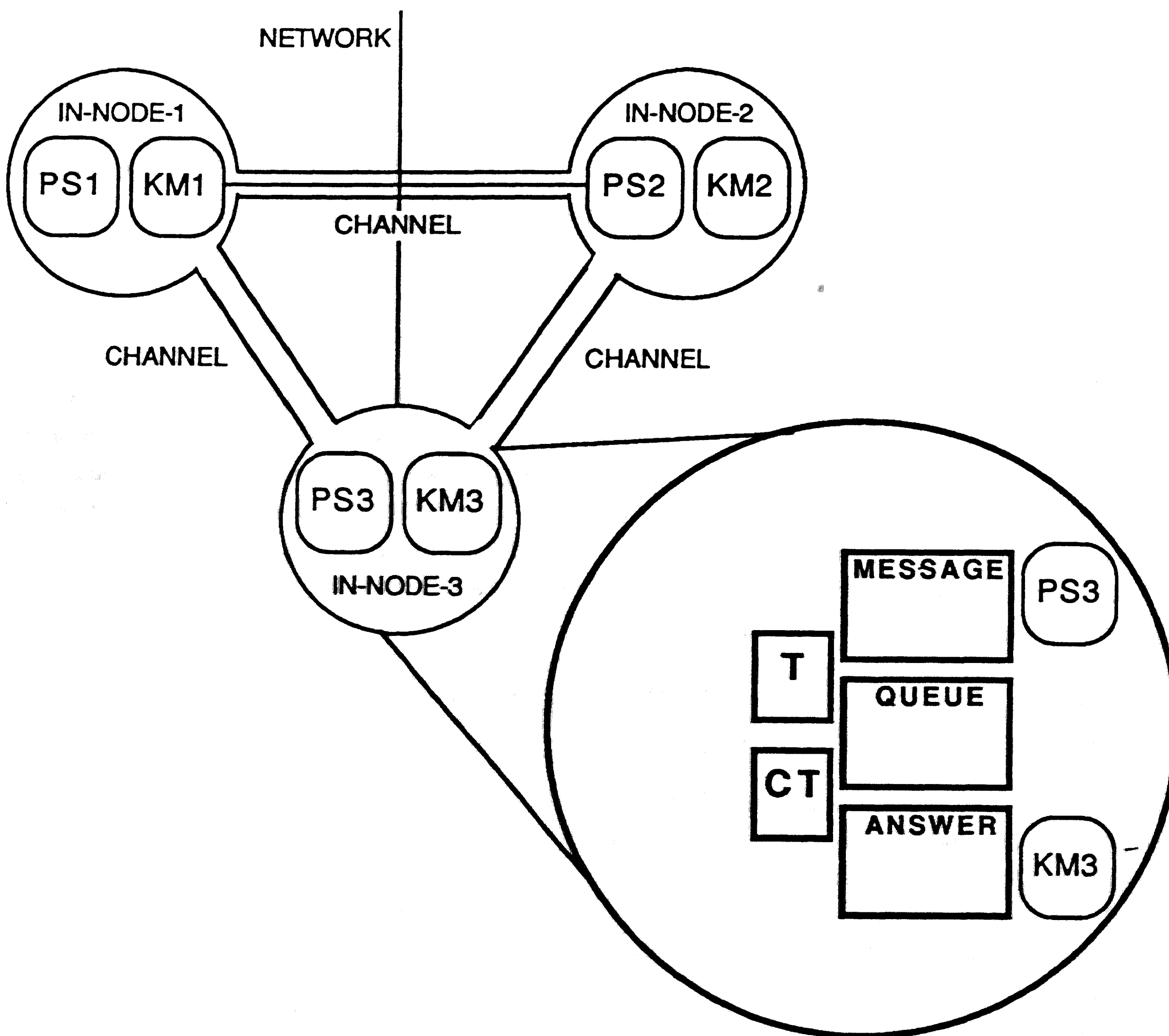


Figure 5-6: Data Layer implementation example

6. Information Layer

6.1 Introduction

The information layer of the manufacturing architecture provides the functions, rules and schemata that support information exchange between the decentralized EMN-nodes of the distributed manufacturing system. In particular, the Information Layer supports:

- automatic information acquisition and
- automatic information management.

The acquisition of information starts from the specific need of a specific EMN-node. Each EMN-node must have the capability to acquire at any moment the schemata used by its Problem solving subsystem but not available in its Knowledge Base subsystem. It must be able to generate requests, to check messages and to generate answers to satisfy these needs. We can identify at this level and for this specific purpose two functionalities:

- the message generation (information search) sequence, and
- the message and answer reception (answer sequence).

The second function, information management, provides the capability for each EMN-node to maintain the consistency of knowledge throughout the distributed system. We can identify three specific functions:

- updating (updating sequence),
- information distribution, and
- communication capability improvement (learning).

The updating activity provides updating of the content of a schema modified by its owner. When the owner of schema modifies it, or when EMN-nodes change a specific value in a schema they are authorized to modify but don't own (they share), the owner EMN-node can generate an updating message to inform all the users of the schema of the modification. Our policy for the updating activity is one owner per schema, i.e., all the schemata can be read by all EMN-nodes but they can only be modified (gloablly throughout the decentralized system) by one EMN-node: their owner. This rule is true for all the schemata except the ones shared by several EMN-nodes. "Shared" means they can be read and modified by several EMN-nodes. But even for these schemata, the updating messages are generated by the unique owner.

Information distribution is concerned with the creation of new schemata in an EMN-node and the initialization of a new EMN-node in the system. In the first case, the EMN-node will have to inform some other EMN-nodes of the existence of new schemata. The main question is who are these EMN-nodes? Also when and what to distribute? To solve this problem, we define a user callable function "distribution-function". This function identifies, according to the nature of the new schema created who the "potential" users of this new schema are. To do this, we define a taxonomy of the information and key words attached to each piece of information. The taxonomy will allow the identification of a group of potential users. The key words will permit improvement of this identification. The second use of the distribution function is either at the initialization of an EMN node or at the reception of distribution messages from a new EMN-node. The distribution function i

triggered at the initialization of each EMN-node, to inform the others of its creation by providing information about its local channel schema, about the schemata it owns (user-table) and about schemata to be updated (class schemata). It is also triggered at the reception of distribution messages from a new EMN-node in the system. The last use of the distribution activity concerns the EMN-node deletion. As EMN-nodes generate distribution messages for their initialization, they also create messages for their deletion. These messages are sent to all the other EMN-nodes to update their tables and channels.

The learning activity uses received messages or information search results, to modify tables (CT/UT/SST) defined at the Data Layer and used for communication. The modification of these tables during the running of an EMN-node improves the capabilities of the EMN-node both to acquire the needed information and also to keep information consistent in the distributed KBS.

The EMN-nodes are able to exchange some information according to their specific needs. In this structure, each EMN-node is responsible for a set of schemata or slots depending on the defined policy. Responsible means that the EMN-node is the only one allowed to update the owned schemata or slots. Each piece of information has a list of users and one owner. The users are the EMN-nodes which receive the updating messages. We have also seen that for some specific schemata, called the shared schemata, we have a multi-owner policy which necessitate the use of the locking mechanism.

For the information search sequence, the policy is the same. Each EMN-node knows, from the content of the communication schema, who the owner of each piece of information available in the DKC system is. The information search sequence uses this knowledge. In case of failure in the information search other policies are applied, such as broadcasting.

Each information is used according to specific views and privileges which are defined by the information owner. The objects (schemata or slots) which are created by an EMN-node belong to it. It is the only one allowed to add, remove or change the content of these objects. But an important part of a distributed system is the ability to access information from all EMN-nodes. This cannot be done without any restriction. Some rights must be defined for the accessing capability. The owner of a piece of information must define for this piece of information who shall be allowed to do what with it.

In this section, we define the access privilege granting policy for the information schema. Then we present in detail the content and the algorithms of the different sequences (updating, information search, distribution and answer) defined at the beginning of this section.

6.2 Access privilege granting

There are four kinds of privileges: select, insert, delete and update. The giving of these privileges is called granting. An EMN-node must grant, for any objects which belong to it (owned by it), any of the four privileges to any users (other EMN-nodes) of these objects. We use the SQL vocabulary from the previous section to define the privileges attached to each schema or slot of a schema.

Each time we want to define the privileges attached to a schema or a slot we must use the word GRANT. This word is followed by one or more of the defined privileges (select, insert, delete and update or all).

```
GRANT SELECT, INSERT, ... (slot-name [, slot-name, ...])
ON schema-name
TO EMN-node-name [, EMN-node-name, ...];
```

After the key words, we indicate the slot name on which these privileges are applied (we can indicate one, several or all the slots of a schema). If we want to apply the privileges to all the slots, we put a "*". After the "ON" key word, we indicate the schema name for which the privileges are defined. Finally, after the "TO", we indicate the list of users having these privileges for this schema.

Example:

Query: *grant SELECT and INSERT privileges for the machine-1 schema to the EMN-nodes 1 and 2*

```
GRANT SELECT, INSERT (*)
ON machine-1
TO EMN-node-1, EMN-node-2;
```

In this case the privileges are applied to all the slots of the machine schema. If we want to have diffe[illegible] privileges for the different slots we have to duplicate this structure.

[illegible]uery: *grant the machine-1 schema SELECT and INSERT privileges for the capacity slot and SELECT for the others to the EMN-nodes 1 and 2*

```
GRANT SELECT, INSERT (capacity)
ON machine-1
TO EMN-node-1, EMN-node-2;

GRANT SELECT (instance, type, identification, color)
ON machine-1
TO EMN-node-1, EMN-node-2;
```

If we want to add all the privileges for the machine-1 schema to EMN-node-2 for example, we must use the ALL key word:

```
GRANT ALL (*)
ON machine-1
TO EMN-node-2;
```

As we can create privileges, we can remove them. For this purpose we use the REVOKE function. This uses the same structure and key words as the GRANT function. But in this case the privileges are revoked.

```
REVOKE SELECT, INSERT, ... (slot-name [, slot-name, ...])
ON schema-name
FROM EMN-node-name [, EMN-node-name, ...];
```

In this declaration, we change one key word: TO becomes FROM.

Example:

Query: *revoke the SELECT and INSERT privileges for the machine-1 schema for the EMN-nodes 1 and 2*

```
REVOKE SELECT, INSERT (*)
ON machine-1
FROM EMN-node-1, EMN-node-2;
```

To complete these elements and the ones described in the previous section, we can use other words from SQL for the updating, creation, deletion and modification of the schemata and slots. In each case a specific function is used:

- slot value updating: **UPDATE** schema-name **SET** slot-name = *new-value*;
- schema creation: **CREATE SCHEMA** schema-name ([slot-name, ...]);
- slot creation: **INSERT INTO** schema-name (slot-name [, slot-name, ...]);
- value creation: **INSERT INTO** schema-name (slot-name) **VALUES** (value [, value ...]);
- schema deletion: **DELETE** schema-name;
- slot deletion: **DELETE FROM** schema-name (slot-name [, slot-name, ...]);

For all of these functions, we can use the WHERE clause condition and all the operators defined in the previous section.

6.3 Automatic information acquisition

Automatic acquisition of information provides an EMN-node with the capability to acquire at any moment and without knowledge of its location in the EMN architecture, the schemata needed by its Problem solving subsystem but not available in its knowledge base subsystem. We can identify at this level two functions:

- the message generation (information search) sequence, and
- the message and answer reception (answer sequence).

Information acquisition occurs automatically, when an EMN-node's problem solver attempts to access information that does not exist in its knowledge base. Four methods are then used by the knowledge base manager to acquire the information:

1. First, the owner of the schema from which the reference to the information was made may be a different EMN-node. Therefore it is reasonable to believe that EMN-node may also have the desired information.
2. Secondly, the schema taxonomy, defined below, contains pointers to those EMN-nodes that maintain schemata of a particular type.
3. Third, the EMN-node maintains a list of EMN-nodes that it corresponds to regularly and may query them.
4. Fourth, as a last resort the request may be broadcast to all EMN-nodes to which it has channels.

We have defined in the network layer the locking and the blocking mechanisms. These two primitives provide security and consistency of the information distributed and shared between the EMN-nodes. Each of the two information acquisition functions can use either the blocking mechanism or the locking mechanism:

- The **blocking mechanism**: During the information search or the information answer, the problem solving activity can be suspended until the end of the search or answer. Blocking is released when the information search or the information answer is finished. The primitive functions for blocking and unblocking have been defined in the network layer.
- The **locking mechanism**: The shared schemata, i.e. updated globally by several EMN-nodes, can have their content locked. This mechanism is triggered when an EMN-node able to update a shared schemata wants to read and then modify its contents. In this case, we want to prevent information inconsistency, by locking the read schema, (if another EMN-node asks for the same schema, it can get the old version of the schema). The schema lock is released once the read and modifications have been done through an updating message by the first EMN-node requesting the schema. If another request is received by the EMN-node for the same schema during this locking period, answers will be generated with a locked status value. An EMN-node which receives an answer with a locked status value can generate another information search message, repeating until it receives the needed schema. A schema lock can only be executed during an information-search message reception for a shared schema, and the request must come from one of the owners of the schema. The primitive functions for locking and unlocking have been defined in the network layer.

Information acquisition uses these mechanisms in specific cases. Blocking is used when an EMN-node needs to synchronize between its internal problem solving activity and communication. In this case, the blocking function is triggered when an information search message is generated (to acquire information not available in the local KBS). The unblocking function is triggered by the reception of an answer containing the needed information. Blocking/unblocking functions perform selective interruption of the problem solving mechanism only, all the other processes (updating, mail box checking and distribution) are continued.

The locking mechanism is triggered by information search messages reception. When an EMN-node receives an information search message which concerns a shared schema and when the agent is the main owner (the schema is member of the UT), the schema is locked. The unlocking of this schema is executed at the reception of an updating message for the schema coming from the EMN-node producer of the information search message. This mechanism uses the locked-schema slot of the communication system.

Message and answer reception is also an automatic process. It can be either synchronous or asynchronous depending on the EMN-node implementation and needs (the synchronization of the communication activity and the problem solving activity is performed using the blocking/unblocking functions). Each message (either an updating, distribution or information search message) or answer (in response to an information search message sent previously) is stored in the mail box of the EMN-node. Periodically, this mail box is checked and its contents evaluated. Processes are triggered according to the nature of the messages (the four types enumerated previously).

6.4 Automatic information management

Automatic information management maintains the consistency of an EMN-node's knowledge throughout the EMN. We can identify three functions:

- information updating,
- information distribution, and
- learning communication capabilities.

Information updating maintains consistency of schemata in the EMN. When the owner of a schema, or an EMN-node authorized to modify a schema it doesn't own but shares, makes a modification, the owning EMN-node generates a message to inform the users of the modification. Shared means they can be read and modified by several EMN-nodes, but global update messages are generated by the single owner (called the main owner). The policy defined for the updating activity is a single owner policy. Periodically the owner of a schema generates updating messages and sends them to the users of the schema if it has been modified during the interval of the updating period. This policy is modified for the shared schemata. The owners of a shared schema can all modify it and have their modifications known by all the users of the schema. But this updating is always done by the main owner, which receives direct updating messages from the other owners and in reponse to these messages generates the global updating messages. The updating activity uses the updated-schema slot of the communication schema. It contains the schemata modified during the previous updating interval.

The information distribution function sends new schemata to EMN-nodes that are potentially interested. The questions are what, when and to whom should schemata be distributed? To solve this problem, we defined a user-callable function: distribution-function. This function identifies who the "potential" recipients are. Types of information of interest to EMN-nodes are maintained in a taxonomic hierarchy. For each class, a set of keywords are used to define the class; that is, schemata that match the keywords are members of the information class. Each class also has a list of EMN-nodes that are interested in the information class. If a recently created schema matches a particular information class, the schema is distributed to the EMN-nodes interested in the class and to any other EMN-nodes interested in classes above it in the hierarchy (subsumed by it).

The other utilization of the information distribution function concerns the EMN-node initialization. When a new EMN-node is created in the system, different schemata are created such as the DKC-system, the local channel schema, the queues schemata, the communication schema and channels with the other EMN-nodes of the global system. To provide each EMN-node enough information about itself, at each initialization, each EMN-node distributes several schemata which characterize their own activity. The first element distributed is the local channel schema. It provides other agents with the addresses of its mail-box, its semaphore-box and also the key-words specific to the new EMN-node. It also distributes its user-table. The user-table defines the schemata owned by the new EMN-node. The other EMN-nodes are able to complete their own local correspondance-table thus improving their information search capabilities. The last information distributed is the members of the class schemata to the different users of these schemata (the class schemata will be defined in the next section).

When an EMN-node is removed from the network, it informs the other EMN-nodes. It sends distribute-END message types which only contain its name. The other EMN-nodes are then able to delete the channel with this old EMN-node and also they remove this EMN-node from all the tables (UT/CT/SST) and class schemata it was member.

When an EMN-node receives distribution messages from a new one, the distribution sequence may be triggered on not. If there is no difference between the DKC-channel schema with this new EMN-node and the local-DKC-channel-schema received in the distribution messages, no distribution is performed. But, if an EMN-node does not have a channel with the new one, the open-channel function is triggered. In addition, the distribution function is triggered to send to the new EMN-node the schemata it can use according to its key-words.

Learning communication capabilities concerns the different tables presented in the Data Layer and used by the different sequences presented previously. It also concerns the class schemata defined for the distribution sequence. These functions allow to complete the different tables used for the communication activity according to the performance of the system and also according to the results of this communication.

Four main functions are defined for improving communication capabilities:

- learn-owner-fn: depending on the result of a broadcast or updating message reception, the system can complete its Correspondance Table, which defines schema owners. The Correspondance Table of the Communication schema is completed when a broadcast succeds, i.e., the agent receives from another one the information it was lookink for, or when an EMN-node receives an update message for a schema it does not own.
- learn-user-fn: depending on the information search messages received by an EMN-node, the system has the capability to complete its User Table, which defines the list of owned schemata and their users. The UT is completed when an EMN-node receives an information search message which concerns a schema owned by that EMN-node.
- learn-shared-schemata: depending on the reception of updating messages concerning owned schemata, the system can complete the Shared Schemata Table. The SST is completed when an EMN-node receives an updating message for a schemata already owned by that EMN-node.
- learn-class-user: depending on the reception of distribution messages from a new EMN-node or from an EMN-node which has created new schemata, the class schemata are completed in terms of members and users using the key-words concept.

6.5 Information Layer example

At this level, the main elements which are added to the previous figure 5-6 are the communication protocols and functions (figure 6-1):

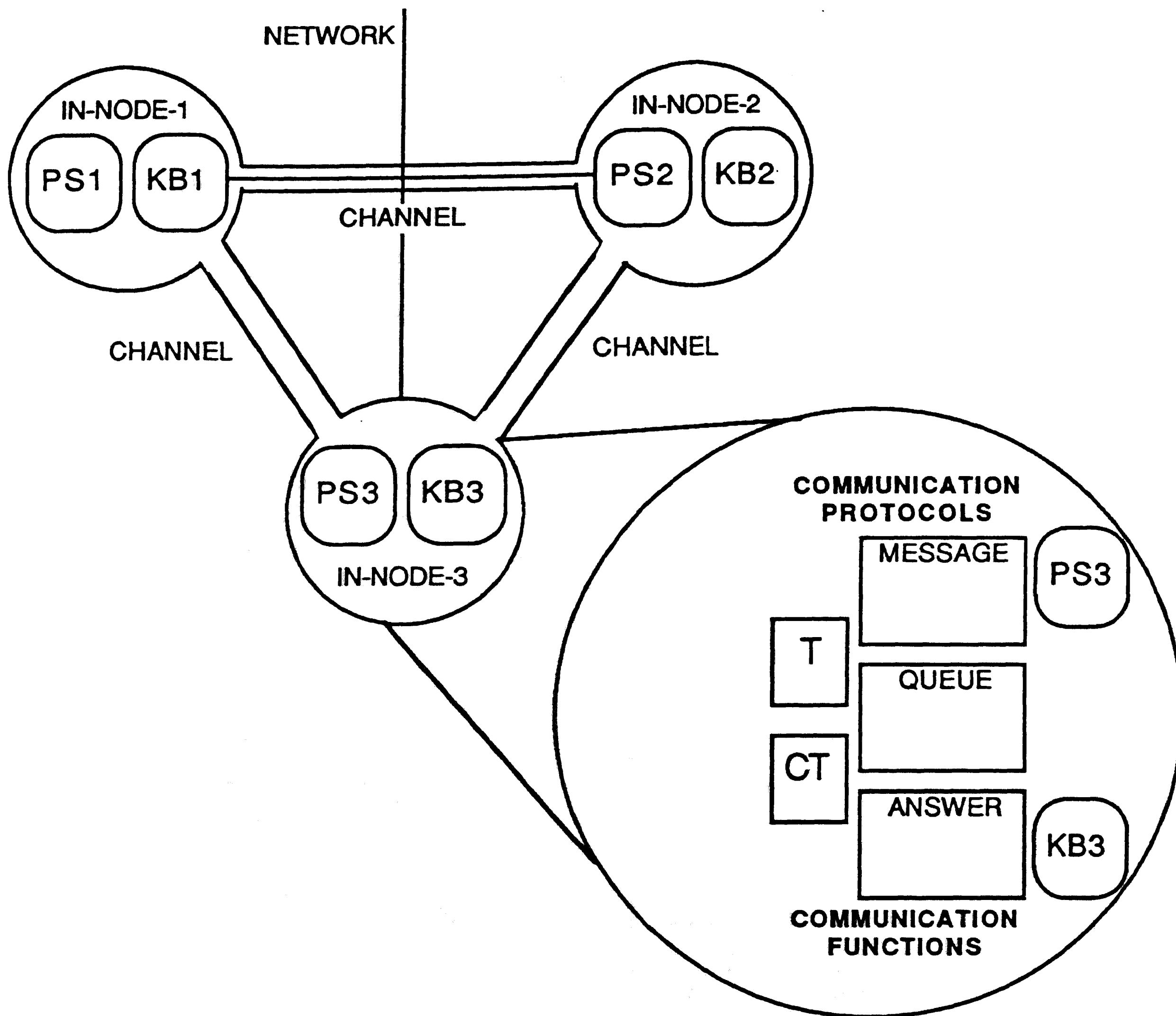


Figure 6-1: Information Layer implementation example

6.6 Information Layer Implementation

The next four sections describe our specific implementation of the information layer into the communication system of the DKC system. We start by a description of the information distribution sequence. The distribution activity triggers the other sequences at initialization. The second sequence presented is the message generation for information search capability. Then, we introduce the updating activity used to keep knowledge consistency in the global system. The last sequence presented is the message and answer reception which either triggers the answer generation functions or the KB update or the distribution sequence or the information search message generation.

6.6.1 Information distribution

This activity is triggered by three different events:

- At the EMN-node initialization: when a new EMN-node is created in the decentralized system, it informs the others of its creation by sending information about itself: its user-table, local channel schema and the class schemata members.
- At the deletion of an EMN-node: when an EMN-node is removed in the decentralized system, it informs the others of its deletion by sending messages about it.
- At the reception of distribution messages: when an EMN-node receives distribution messages containing a local channel schema of a new EMN-node, according to the content of this schema, the EMN-node triggers its own local distribution activity to this new EMN-node.
- When a new schema is created in an EMN-node: this schema must be distributed to possible users. This distribution uses the key-word concept.

The distribution activity uses functions (we define in the next section) to perform selective distribution of information according to different criteria. We present at first these primitives, then in the next two parts of this paragraph, we define the distribution sequence process at an EMN-node initialization and the distribution process at the reception of distribution messages from a new EMN-node.

Schema 6-1: CLASS

CLASS		
SLOT	FACET	VALUE
Key-words	*Value:* *Restriction:*	type string*
Users	*Value:* *Restriction:*	type EMN-node*
Members	*Value:* *Restriction:*	type schema-name*

The functions for the information distribution use mainly the class schema slots. The information distribution is done according to a key-word and taxonomy policy. Each schemata is attached to a specific class. For each class, we define a shema which is an instance of the generic class schema. This class schema contains two main slots: the users and the members. The members slot is a list of schemata that are part of this class. The users slot is a list of EMN-node users of this class.

Distribution can be done in two ways: by the members or by the users. If we want to distribute a specific schema, we have to find its class and then the users of this class. If we want to distribute some schemata to an EMN-node, we have to determine its membership in some information classes and then distribute these classes (figure 6-2 shows the process for distribution from a schema point of view).

Distribution can also be performed from a knowledge classification perspective using class schemata. The distribution messages are generated whenever a member of a class is modified.

In our implementation, we have defined five functions which cover these three aspects:

- distribute-schema: distributes a schema to all its users according to its class membership.
- distribute-schema-hierarchy: distributes a schema and its superclasses, subclasses and instances to all users according to its class membership.
- distribute-agent: sends to a specific agent all the members of the different classes this agent is a user of.
- distribute-agent-hierarchy: sends to a specific agent all the schemata and their superclasses, subclasses and instances of the different classes this agent is user.
- distribute-class: performs a general distribution to all the users (EMN-nodes) of a specific class of all the members (schemata) of that specific class.

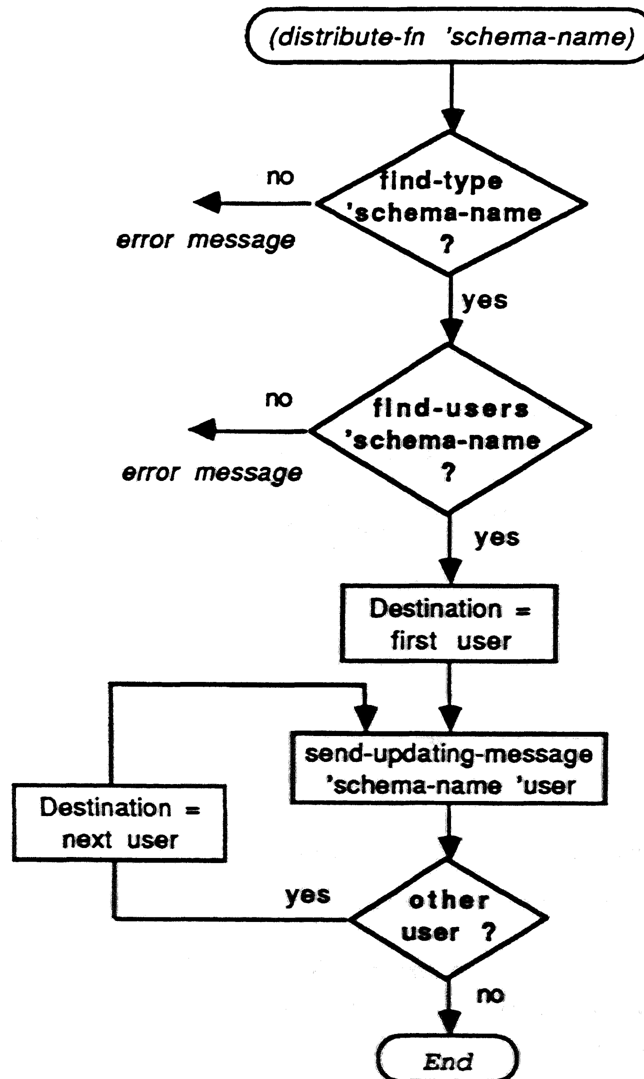


Figure 6-2: Information distribution sequence

On top of these, we have implemented a generic distribution-function which, according to the nature of its parameters, triggers one of these previous functions.

The distribution function takes into account the hierarchy of schemata linked either by an INSTANCE relation, an IS-A relation and one or several user defined relations. The user defined realtion are specific to each EMN-nodes and the ones which have to be taken into account to build the hierarchy of schemata are indicated as a value of the *relation-name* variable. If we distribute a schema, the complete tree under this schema will also be distributed and the direct hierarchy of schemata above will also be distributed (figure 6-3). The inverse relation of both standard IS-A and INSTANCE links and the user defined relations are also taken into account to define the hierarchy of each schema to be distributed.

In the example presented in figure 6-3, we distribute a schema (which is indicated in black) which has IS-A, INSTANCE, IS-A+INV and INSTANCE+INV with other schemata. In this case the distribution of that schema will also imply the distribution of the 2 schemata above it and of the 11 schemata under it.

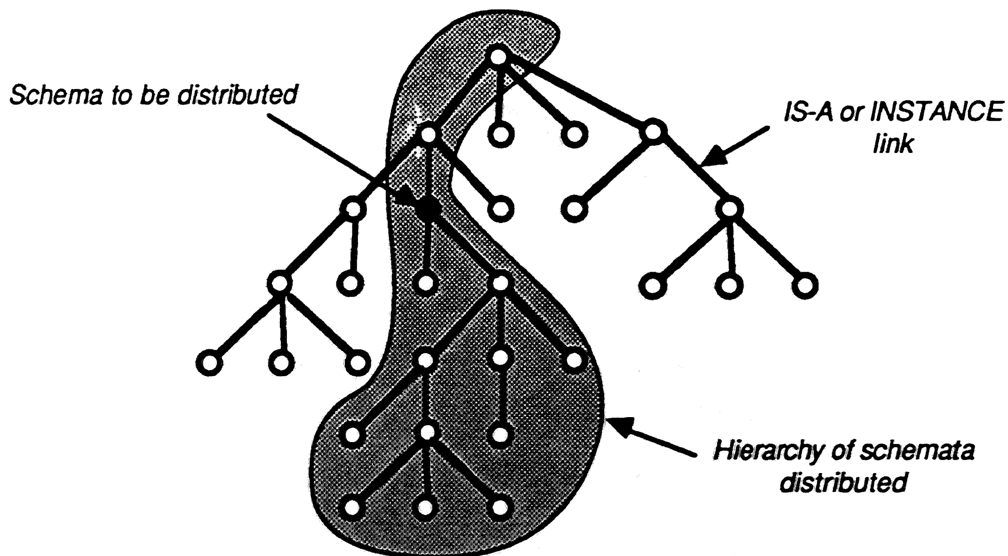


Figure 6-3: Hierarchical distribution example

The purpose of the distribution in the Enterprise Management Network is very important mainly in term of information consistency and acquisition. The role of the distribution function can be divided in two different steps:

- Information distribution at the EMN-node initialization or at the EMN-node deletion
- Information distribution at the reception of other EMN-node initialization messages.

In the first case (figure 6-4), the process for an EMN-node initialization is described at the Network Layer completed by the elements of the Data Layer. At the information Layer, the initialization consists in triggering for the first time the different sequences supported by this layer:

- message generation,
- message and answer reception,
- updating activity and
- distribution.

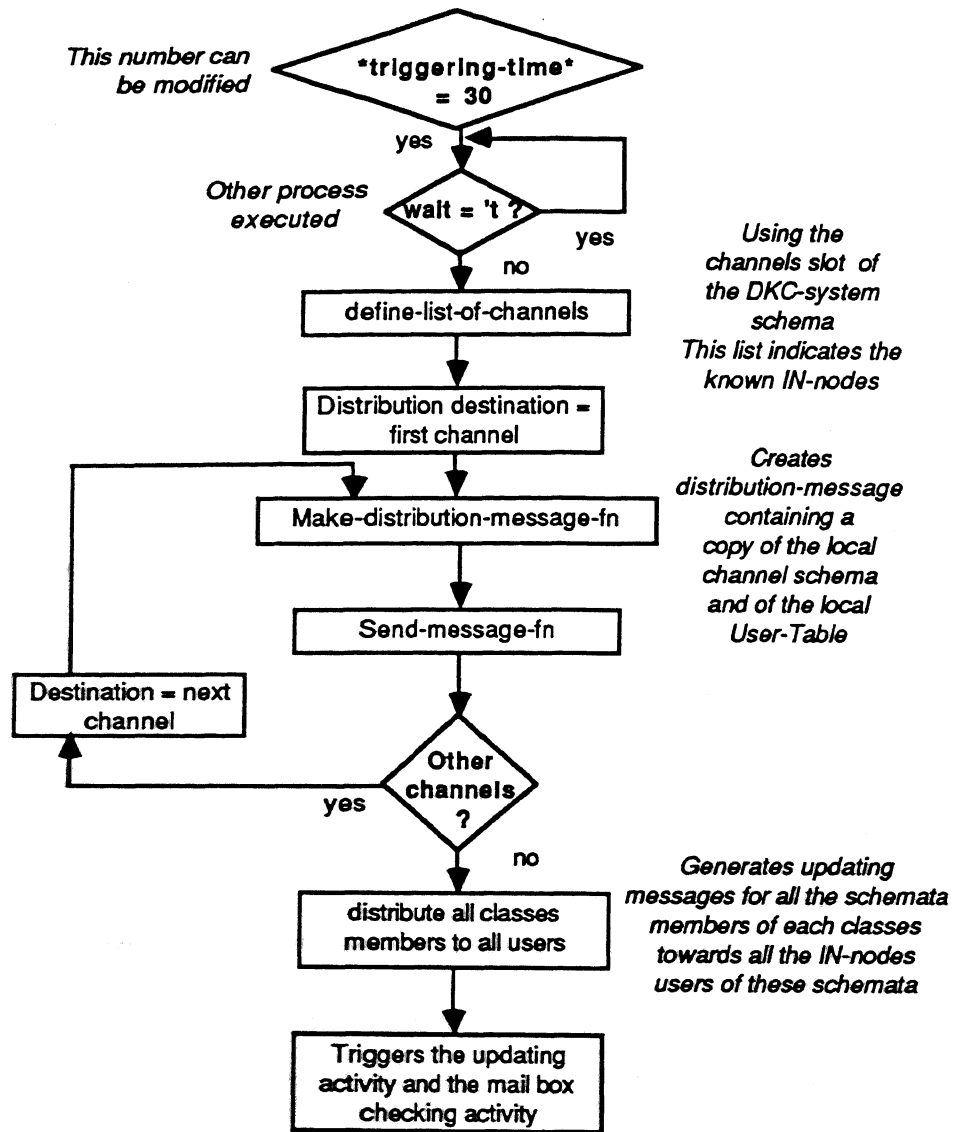


Figure 6-4: Distribution sequence for an EMN-node initialization

The first sequence triggered is the distribution activity. This activity consists in distributing the schemata owned by each EMN-node of the decentralized system to all the others EMN-nodes users of these schemata. This distribution is executed according to the different classes of schemata defined in each EMN-node, according to the list of schemata owned by each EMN-node (this list is specific as a value of the User-Table slot of the communication schema) (each EMN-node is responsible for distributing the schemata they own) and in correspondance with the key-words of the other EMN-nodes (the ones recognize by each EMN-nodes, i.e., supported by a channel schema). The distribution in an EMN-node is excuted for the different classes by matching the key-words of each class with the key-words of each other EMN-nodes. In addition to the class schemata members distribution, each EMN-node sends to the other recognized EMN-nodes, i.e., supported by a channel schema, a copy of its local channel schema and of its local User-Table. By this way, each EMN-nodes get the key-words atached to all the others (they can complete their class schemata members by

matching key-words) and besides, they can complete their local Correspondance-Table knowing the owner of some schemata. The aim of this distribution at the initialization of each EMN-node is mainly to improve the communication capabilities between the EMN-nodes of the system but also to ensure coherence and to provide to each EMN-node knowledges about the others and knowledges about the schemata they need for their internal problem solving activity. Another goal of this distribution for EMN-node initialization is to inform all the other already existing EMN-nodes of the creation of a new decentralized system member. Once an EMN-node is initialized, the distribution function triggers the others processes such as updating, message and answer reception and message generation. These other sequences support the regular running of the internal problem solving activity.

The distribution of the schemata owned by this new EMN-node is done by using updating message schema. For the distribution of the local channel schema, we use the distribute-LC-message schema created by the **make-LC-distribution-message**. For the distribution of the user-table, the new EMN-node creates a distribute-UT-message created by the **make-UT-distribution-message**. In addition, the distribution function triggers according to the modification performed on the different tables and schemata (CT/UT/SST/class) the consistency functions defined at the Data Layer. The triggering of all this activity is supported by a VMS routine system. This routine tiggers the **distribute-agent-fct** which executes the distribution for each known EMN-node (having a channel schema with this new EMN-node) of the different information (local-chanel, UT and classes).

To make a summary of this distribution activity for an EMN-node initialization, we can identify these different steps:

1. The EMN-node to be initialized checks the different channels already created with the other EMN-nodes of the system.
2. The EMN-node sends to each of these known EMN-nodes a copy of its local channel schema.
3. The EMN-node sends to each of these known EMN-nodes a copy of its local User-Table.
4. For each class schema it owns, it generates an updating message for all the members (schemata) of these different classes towards all their users (EMN-nodes).

For the distribution regarding to EMN-node deletion, the sequences are equivalent. The only difference is in the nature of the information distributed. In this case, we use the distribute-END message type.

The second utilization of the distribution sequence is at the reception of distribution messages coming from another EMN-node (figure 6-5). As we have described in the previous paragraph, each EMN-node sends distribution messages (which are in fact updating messages) towards the other already existing EMN-nodes of the DKC-system. At their reception, each EMN-node triggers its own distribution function according to the information received about the new EMN-node. These different steps are executed:

1. It checks the existence of a channel schema with this new EMN-node. Accordingly, it creates the schema or not and completes the information (using the local channel schema received from that new EMN-node) about this new EMN-node (for example the key-word list attached to this EMN-node).
2. It completes, using the received User-Table of that new EMN-node, its own Correspondance-Table.

3. It creates the sehemata sent by this new EMN-node according to its own need (defined by its local key-words).

4. It completes, using the key words of this new EMN-node, the different class schemata users slot.

5. It triggers the distribution sequence and sends to this new EMN-node its own local channel schema (for information about its key-words) and updating schemata according to the key-words of that new EMN-node matching with the key-words of the class schemata it owns. In addition, it sends its local User-Table.

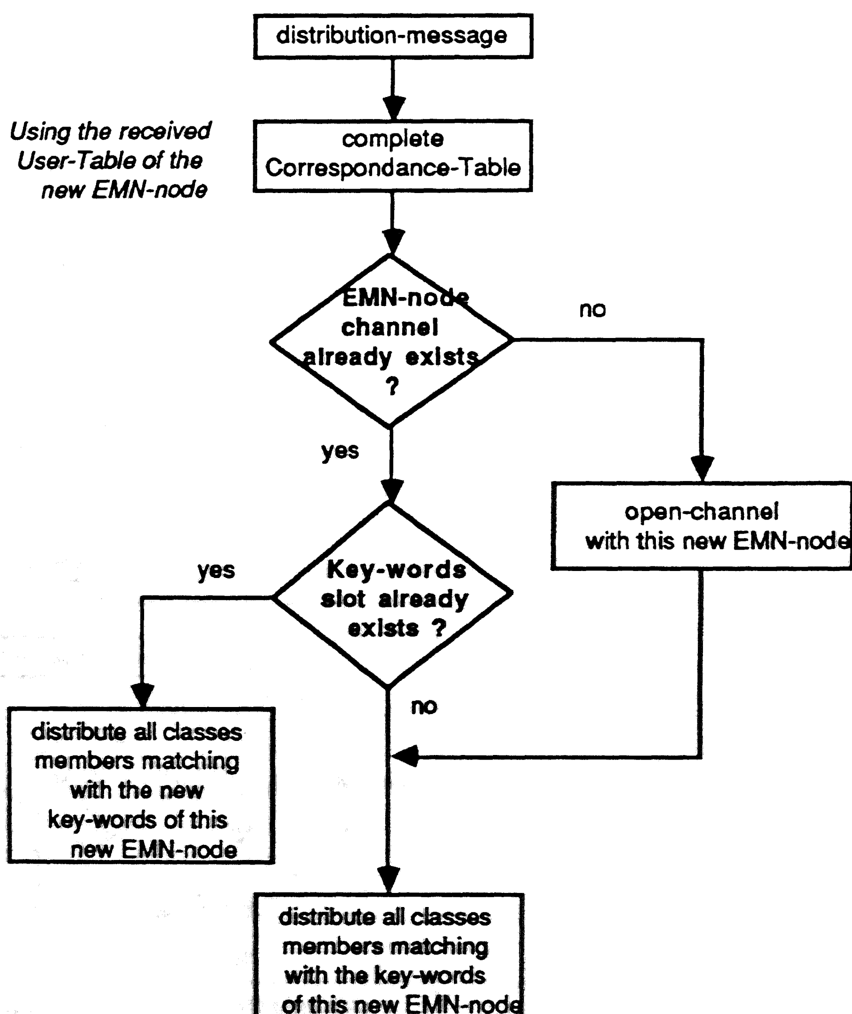


Figure 6-5: Distribution message reception

This sequence is executed between two EMN-nodes until the acquisition by both of them of all the information regarding they key-words. The main test to stop replying to a distribution messages sequence is the comparison between the received local channel schema of an EMN-node and the existing copy of that schema in the EMN-node which receives these distribution messages.

The distribution sequence is only used when an EMN-node is initialized in the global system. For all the other modifications about schemata or for all information acquisition between EMN-nodes, we use the other processes defined in the next sections.

6.6.2 The message generation sequence

The message generation sequence is presented in figure 6-6. We introduce the different functions used for this activity, and we develop the contents of these functions in this section.

The function **make-message-fn** is the basic function of the message generation sequence. It is used to generate instances of the message schema. This function creates schemata called message-1, message-2, ... ,message-n. This function is the basic function of the searcher. All the other functions will be used either to trigger or to complete some slots of the message schemata generated by the make-message-fn function.

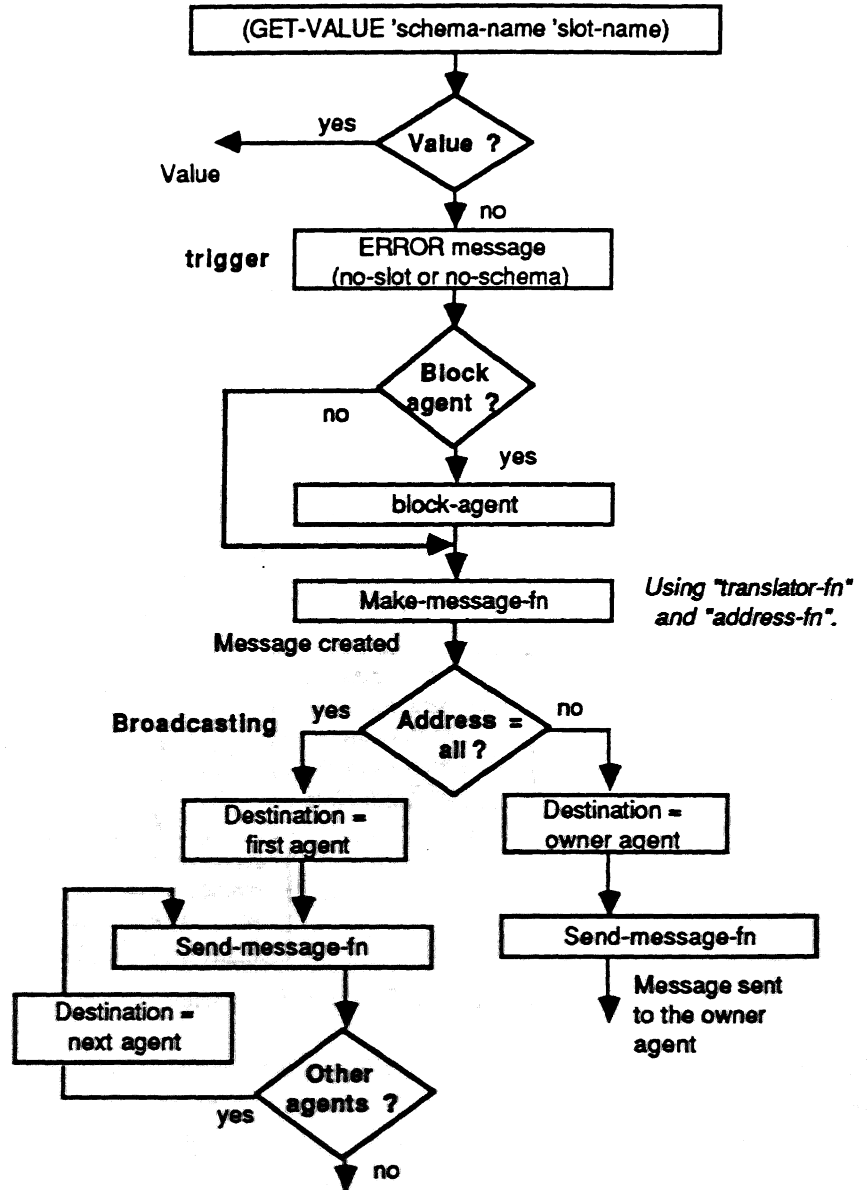


Figure 6-6: Message generation sequence

The make-message-fn function creates an instance of the message schema. It fills the slots of the created message by calling other functions. To fill the message slot, this function starts with the given data. As a message is created according to an information need (which is a schema), the first elements provided are the needed schema name and slot name (because the command GET-VALUE provides them). With these two data, we can fill the two slots "schema-name" and "slot-name" of the generated message schema. For the "instance", "priority" and "type" slots, their values are always the same for each generated message. The value of the slot "number" is filled with the value of n integers. This integer is automatically incremented by one each time a new message is created.

For the four other slots, we use the "translator-fn" and "address-fn" functions. The first function translates the schema name and slot name needed from the internal vocabulary to the communication language. The second one defines where to send a message depending on the needed schema. Both of them use the communication schema.

The triggers of the make-message-function are the three following schemata: no-schema-spec, no-slot-spec, no-value-spec. They are triggered when the CRL system creates an error message using the system-error schema.

Before describing these triggers, we must define the utilization of the EMN-node-blocking mechanism. We have defined a function called **Block-EMN-node** whose purpose is to suspend the internal problem solving activity of an EMN-node. This function only suspends this process. The others, such as mail-box-checking, updating, or distribution, are maintained. The block-EMN-node function allows the interruption of the internal problem solving activity when some information is not available locally and the communication system has to acquire it through message sending. In this case, the internal problem solving activity is interrupted until reception of the needed information, and is then re-started using the unblock-EMN-node function.

The **unblock-EMN-node** function is triggered by the receive-answer sequence. Depending on the content of answer received in response to an information search (with block-EMN-node), we unblock the internal problem solving of the EMN-node. The unblock-EMN-node function is triggered when an answer contains the needed information. In this case and only in this case, the internal problem solving of the EMN-node can continue its processing.

Schema 6-2: NO-SCHEMA-SPEC

NO-SCHEMA-SPEC		
SLOT	FACET	VALUE
Instance	*Restriction:*	error-spec
Type	*Restriction:*	value
Signal	*Restriction:*	nil
Value	*Restriction:*	*(lambda (x)* *(when *trace** *(format t"~%There is no schema by that name and"))* *(create-schema (get-value x 'schema)))*

The schema **no-schema-spec** is the first trigger of the make-message-fn function. This schema is an instance of the error-spec schema. It is connected to the "no-schema" slot of the "system-error" schema. Each time the CRL system finds a "no-schema" error type, it dispatches the "value" slot of the "no-schema-spec" schema. The "value" slot executes the make-message-fn.

The "no-schema-spec" does not trigger the make-message-fn because if we have a no-schema error, we also have a no-slot error. Therefore we have given the trigger responsibility to the next schema: no-slot-spec.

Schema 6-3: NO-SLOT-SPEC

NO-SLOT-SPEC		
SLOT	FACET	VALUE
Instance	*Restriction:*	error-spec
Type	*Restriction:*	value
Signal	*Restriction:*	nil
Value	*Restriction:*	*(lambda (x)* *(when *trace** *(format t"~%There is no slot by that name."))* *(make-message-fn (get-value x 'schema)* *(get-value x 'slot))* *(send-message-fn (get-value 'message 'instance+inv))* *(create-slot (get-value x 'schema) (get-value x 'slot)))*

The schema **no-slot-spec** is the second trigger of the make-message-fn function. This schema is also an instance of the error-spec schema. This one is connected to the "no-slot" slot of the "system-error" schema. Each time the CRL system finds a "no-slot" error type, it dispatches the "value" slot of the "no-slot-spec" schema. The "value" slot triggers the make-message-fn function.

The no-slot error type can be generated in two cases: if there is no schema (so there is also no slot) or if there is no slot (but the schema exists). So we have given this error type the responsibility to trigger the make-message-fn function. Once a message is created, we must send it to its destination (which is indicated by the destination slot of the message). For this purpose we use the send-message-fn function.

The last trigger of the make-message-fn function is the **no-value-spec** schema. This schema is also an instance of the error-spec schema. This one is connected to the "no-value" slot of the "system-error" schema. Each time the CRL system finds a "no-value" error type, it dispatches the "value" slot of the "no-value-spec" schema. The "value" slot executes the make-message-fn.

Once an instance of the message schema is generated, we must send it to another EMN-node to get the information (schema). The purpose of the **send-message-fn** is to send the messages. This function sends an instance of a message from one EMN-node to another. We have several possibilities for the destination of a message: if the destination is known (from the destination slot of the message) we use direct communication; if not, we broadcast (in this case the destination slot of the message has the value: all).

Schema 6-4: NO-VALUE-SPEC

NO-VALUE-SPEC		
SLOT	FACET	VALUE
Instance	*Restriction:*	error-spec
Type	*Restriction:*	value
Signal	*Restriction:*	nil
Value	*Restriction:*	*(lambda (x) (when *trace* (format t"~%There is no value for this slot.")) (make-message-fn (get-value x 'schema) (get-value x 'slot)) (send-message-fn (get-value 'message 'instance+inv)) (new-value (get-value x 'schema) (get-value x 'slot) 'nil))*

Direct communication means we have a unique destination for the message. This destination is the owner EMN-node of the needed schema.

Broadcasting means we send the message to all the EMN-nodes of the DKC system. For this, we use the Correspondance Table (CT). The send-message-fn sends the message to all the EMN-nodes indicated in the CT. We have a loop which analyses the EMN-node addresses in the CT and sends the message to each one. The message is sent just once to each EMN-node.

When we send a message from the local EMN-node to another, we copy the message into the mail box of the other EMN-node, using the "dkc-send" function. To be able to use it, we must transform our message into a **dkc-message**. So, we create an instance of the simple dkc-message schema with the value message-name, for the schemata slot of the dkc-message created.

The dkc-send function copies the dkc-message into the **dkc-queued-message** of an EMN-node. This queue is the file interface from one EMN-node to the others. We will see, in the message reception section, that this is not the only queue. The difference between this one and the others is its protection. To allow information exchange, we must have some free files open for writing and reading. This is the purpose of this queue.

6.6.3 The updating activity

The previous section concerned information exchanges between EMN-nodes of the decentralized knowledge craft. To ensure coordination, we must add a second functionality to our structure: the updating activity.

Since the information is shared by all the EMN-nodes of the DKC, to maintain consistency we must update the Knowledge Base subsystem of each EMN-node according to modifications made by others. We make the assumption that each schema has a unique owner.

The owner is the only one allowed to update a schema globally, i.e., for all the EMN-nodes. But each user of a schema can locally modify the content of a schema it uses. Each time an EMN-node

modifies schemata it owns in its Knowledge Base subsystem, it generates updating messages and sends them to the EMN-nodes who are users of these schemata. An updating message is an instance of the message schema with the specific value update for the "type" slot.

The function **make-updating-message-fn** creates these instances. We summarize in figure 6-7 the sequence for the realization of the updating activity in an EMN-node. We define the trigger of this activity and the different functions executed during this sequence. The EMN-nodes which receive the updating messages are called the users of the updated schemata. The list of users of each schema owned by an EMN-node is indicated in the user-table slot of the communication schema. The updating messages use the generic communication language through the "mini-translator-fn" function.

The purpose of an updating message is to update a schema in the other EMN-nodes who use it. We know the schema name to be updated. This name is used to fill the "schema-name" slot of the generated updating message. We also know the destination of the message (address of the user). This address is provided by the trigger of the make-updating-message-fn function. The "instance", "type" and "priority" slots are automatically filled (by the data related to an updating message). The "number" slot is filled with the integer "n". This number is incremented by one each time an instance of the general message schema is created. This instance can be either an information research message type or an updating message. To start the "make-updating-message-fn", we must have a trigger. In our case, we have an automatic trigger using the SYS$SETIMR VMS routine.

The syntax of this routine is: **SYS$SETIMR** *[efn], daytim,[astadr], [reqidt].*

- efn: is the event flag to be set when the timer expires.
- daytim: is the time at which the timer expires.
- astadr: is the AST service routine that is to execute when the timer expires.
- reqidt: is the identification of the timer request.

This routine starts the updating activity after a set period. This period is determined by the value of the *updating-interval* variable. When we initialize the communication system of an EMN-node, by creating an instance of the DKC-system schema, we arm the updating trigger (**arm-update** function) for the first time. This function calls the SYS$SETIMR VMS function, using the Lisp call-out function. The values provided to the SYS$SETIMR function are the delay: the value of the variable *updating-interval* (after a conversion of this value into a day time using a dkc function) and the function to fire: the updating-message-trigger (through its address, updating-message-trigger-id). The updating sequence uses other tests such as the "wait = 't" test. This checks if other communication processes are executing. For security purposes, we allow the execution of only one communication process at a time. The second test concerns the verification of owned modified schemata. If no schemata have been modified between two updating sequences, no updating messages are sent and the updating process is re-armed. The last point we want to define concerns shared schemata. In this system, we define two types of information, single-owner and multi-owner. We have described the policy for the single owner type of schemata. As previously described, for the multi-owner, several EMN-nodes are able to modify the contents of a schema. These modifications could have global repercussions on the system. In this case such a schema is called *shared*. The shared schemata of an EMN-node are indicated in the shared-shema slot of the communication

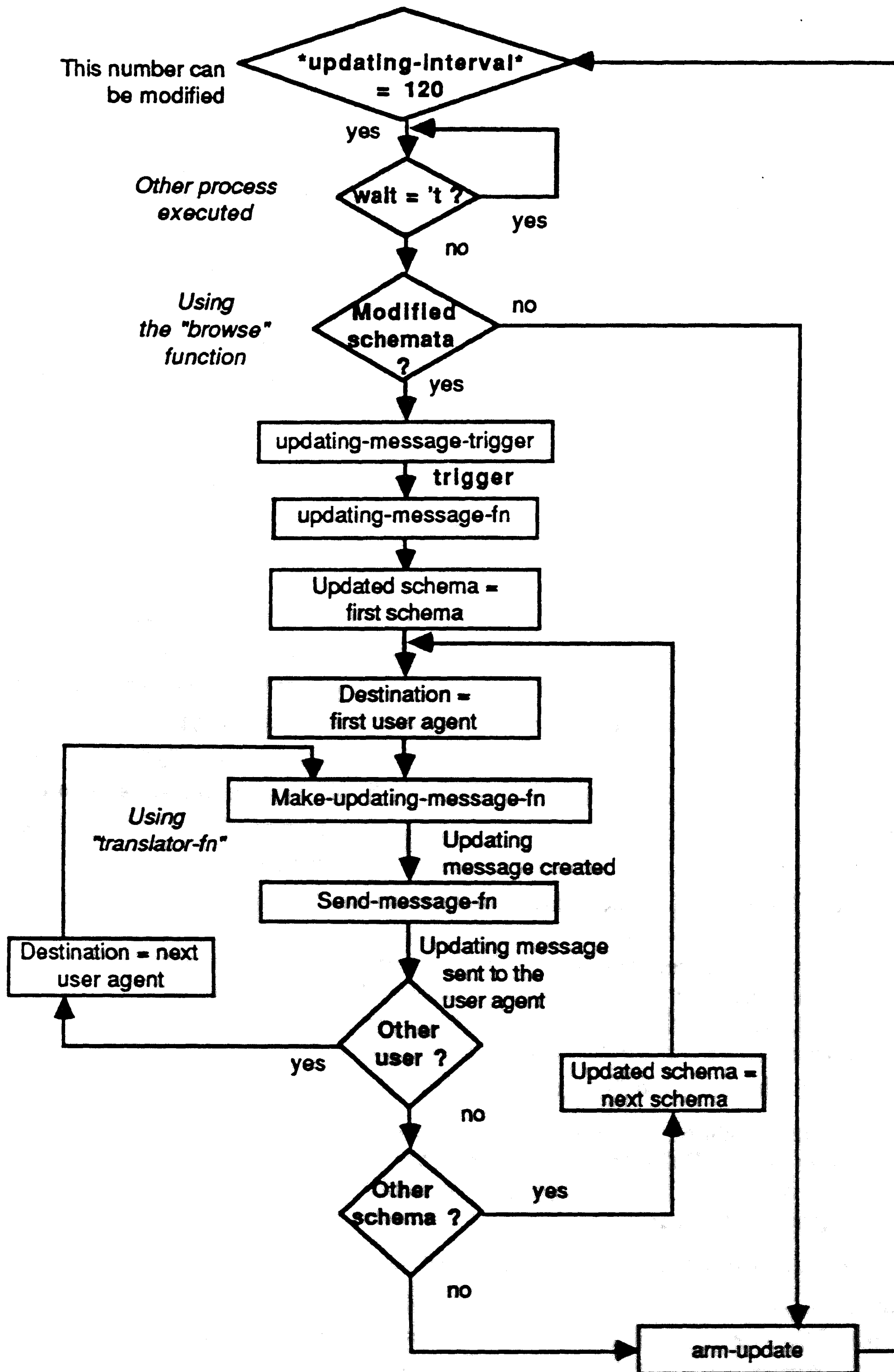


Figure 6-7: Updating message generation sequence

schema. In this slot, we indicate the list of shared schemata and for each of them the list of owners. For the shared schemata, we also apply a single-owner policy for the updating activity: only one of the owners is allowed to send global updating messages. Each time a shared schema is modified by

one of its owners if this owner is not the main one, it sends an updating message to the main owner. In the next updating sequence, the modified schema will be updated globally.

6.6.4 The message and answer reception sequence

As in the updating message generation, the answer generation must be triggered. For this, we use the same VMS routine with a different interval: *dkc-interrupt-interval*. The function which is triggered by the SYS$SETIMR VMS routine checks in the mail box for messages which have been received. If there are no messages, the VMS routine is re-armed. If messages are found, an answer is created and the VMS routine is re-armed. The message (or answer) reception is automatic. The utilization of the SYS$SETIMR VMS routine provides this automation. The VMS routine starts after a period determined by the "daytim" parameter set to *dkc-interrupt-interval*, for the message (or answer) reception activity. The message and answer reception sequence is summarized in figure 6-8. It indicates the different Lisp functions used. But these functions are not enough. When the responder of an EMN-node receives a message or when the searcher receives an answer, they must react according to individual behaviour and also according to their content. For this, we use rules and schemata.

Depending on the nature of the messages an EMN-node receives in its mail box, different processes are executed. The first distinction we make is based on the nature of the received message: message or answer. Then, the distinction is done on the type and/or status of the message or answer. The different status are: t, nil and locked for the answer schema and the different types are: update, information-search and distribution for the message schema. The distribution type has three subtypes: distribute-UT, distribute-LC and distribute-END. These characteristics allow to distinguish the messages. Processes are then triggered accordingly (figure 6-8). The different sequences performed depending on the message or answer status and/or type are:

- If it is a **message**, it can be either an information request or an updating message or a distribution message.
    - If it is an **information-search message**, the answer process is executed. This process checks in the KBS if the needed schema is available and its nature (owned, used or shared). If it is not available, an answer with a nil status is generated. If it is available, according to its nature different functions are executed:
        - If it is a **used** schema: an answer with a t status is generated.
        - If it is a **shared** schema: an answer with a t status is generated and it is locked locally if the EMN-node requesting this information is one of the owners of the schema. The schema is unlocked when the EMN-node receives an updating message for it.
        - If the schema is **locked**: an answer with a locked status is generated.
    - When a received message is an **updating message**, the KBS is updated according to the contents of the message. At this point, the communication system checks if the updated schema is locked or not. If it is locked, the communication system unlocks it.
    - When we received a **distribution message** type, it can be either a distribute-LC message type, or a distribute-UT message type, or a distribute-END.
        - If it is a **distribute-LC** message type, it contains a copy of the local channel schema of a new initialized agent, the responder compares this received schema and its local copy (if it exists) and triggers its own local distribution sequence accordingly. If the channel does not exist with this new EMN-node,

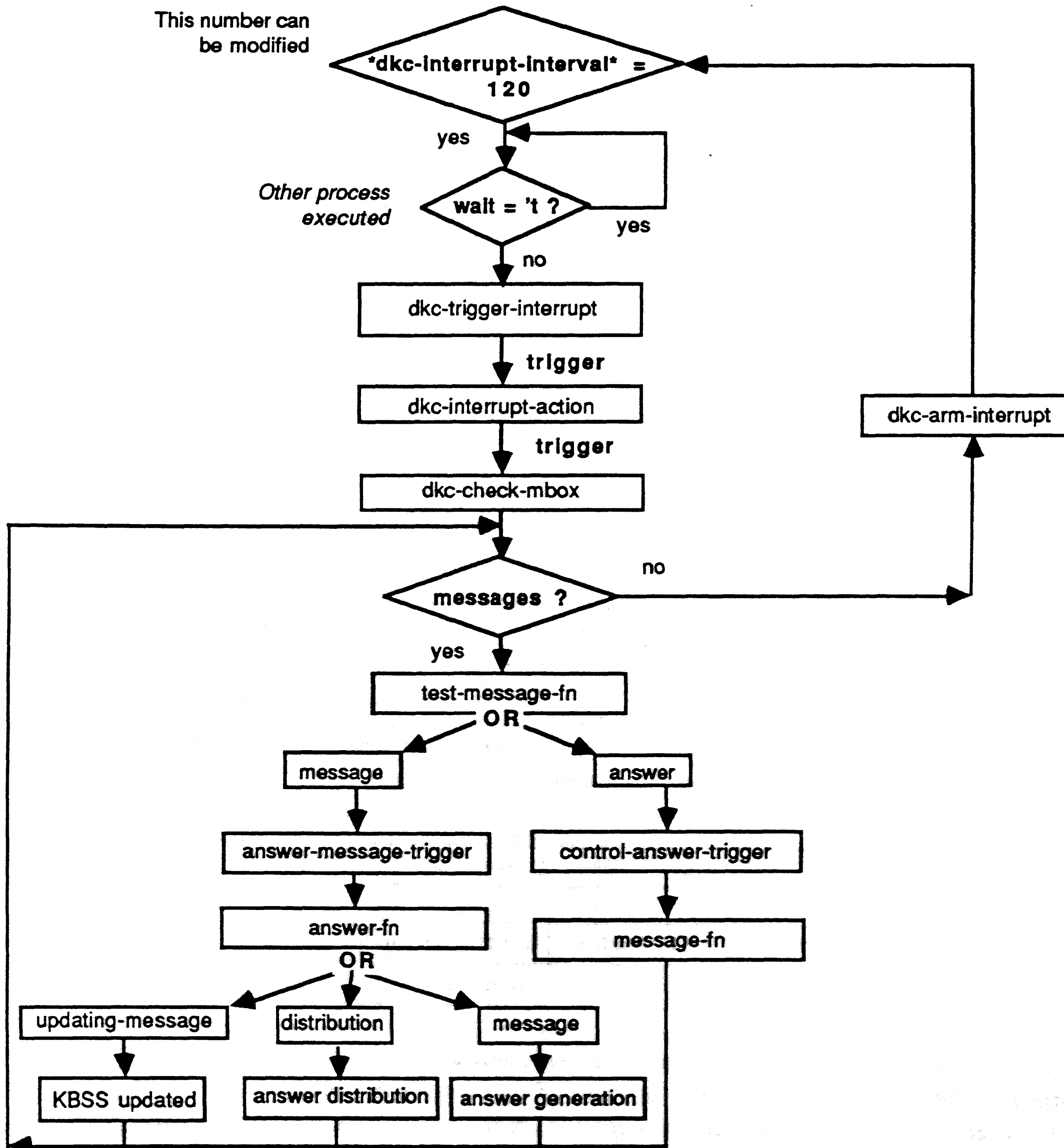


Figure 6-8: Message and answer reception sequence

it is created. The key words attached to this new EMN-node are examined and according to their matching with class schemata key-words, updating messages are generated.

- If it is a **distribute-UT** message type, it contains the User-Table of a new initialized agent. Accordingly, the responder triggers some learning functions to complete the Correspondance-Table using the UT of this new agent. In addition, the class schemata users and the Shared-Schemata-Table are completed.

- If it is a **distribute-END** message type, it informs of an EMN-node deletion. In such a case, the responder deletes the channel schema with this agent and also remove its name from all the tables and class schemata.

- For received **answers**, we can distinguish several cases. These cases are defined according to the status of the answer:
  - If the status is **locked**: another message is generated to the EMN-node which sent this answer.
  - If the status is **t**: the schema is provided to the KBSS and PSS.
  - If the status is **nil**: we apply several policies such as friend selection and broadcasting (if not already applied). In effect other messages are generated to acquire the needed schema.

6.6.4.1 The message reception sequence

To be able to generate an answer to each message, we must store it in a queue. A queue must be attached to a process. So we create a process called **answer-message**. The queue of this process is the **message-queue**.

Schema 6-5: Answer-message

Answer-message		
SLOT	FACET	VALUE
Instance	*Restriction:*	process
Active-queues	*Restriction:*	message-queue

Schema 6-6: Message-queue

Message-queue		
SLOT	FACET	VALUE
Instance	*Restriction:*	imperative-queue
Queue-pointer	*Restriction:*	$message

Answer-message and message-queue are represented as schemata which are, respectively, an instance of the process schema and an instance of the imperative-queue schema. The message queue is the place where the system stores the received messages. Once a message has been placed in the queue, the system starts the answer-message process. The aim of this process is to generate an answer to each received message.

Schema 6-7: New-message

New-message		
SLOT	FACET	VALUE
Is-a	*Restriction:*	event
Message-queue	*Restriction:*	message-queue
Action	*Restriction:*	answer-fn

In a queue we store events. The events of the responder are the messages that have been received. So we create a schema called **new-message**, which is an event. All the messages which can be received by the responder will be instances of this event schema. For this, we add a new value in a message schema to its slot "instance": gets the value new-message and we create a new slot: "event-time" with the value: gets (relative-time nil 5 0 0 0). The value of the event-time corresponds to 5 seconds. Since a received message is also an instance of the message schema, the instance slot has two values: message and new-message. The event-time slot corresponds to the time at which we want to answer the received message. We store these events in the message-queue. The action performed on these events is the "answer-fn" function. When the clock reaches the value of the event-time, the system executes the contents of the action slot. This function creates an answer to the message.

6.6.4.2 The answer reception sequence

This next sub-section describes how the searcher manages the answers received from the responders of the other EMN-nodes, corresponding to the messages it has sent (figure 6-9).

Schema 6-8: Control-answer

Control-answer		
SLOT	FACET	VALUE
Instance	*Restriction:*	process
Active-queues	*Restriction:*	answer-queue

Schema 6-9: Answer-queue

Answer-queue		
SLOT	FACET	VALUE
Instance	*Restriction:*	imperative-queue
Queue-pointer	*Restriction:*	$answer

To be able to check the answer(s) received for each message, we must store them into a queue. Since a queue must be attached to a process, so we create a process called **control-answer.** The queue of this process is the **answer-queue.** The control-answer and answer-queue are represented as schemata which are instances of, respectively, the process schema and the imperative-queue schema.

The events of the searcher are the answers which are received. We create a schema called **new-answer**, which is an event. We store these events in the answer-queue; the action performed on these events is the "message-fn" function. We add in each answer received a new value to the slot "instance": gets the value new-answer and a new slot "event-time" with the value (relative-time nil 5 0 0 0). The instance slot of an answer already exists in the schema. So the value: new-answer is added to this slot. The event time is the time when we will apply to this event its action function: message-fn. This function will check if the received answer provides the needed schema. If yes, the schema will be provided to the problem solving subsystem of the EMN-node. If not, the function will

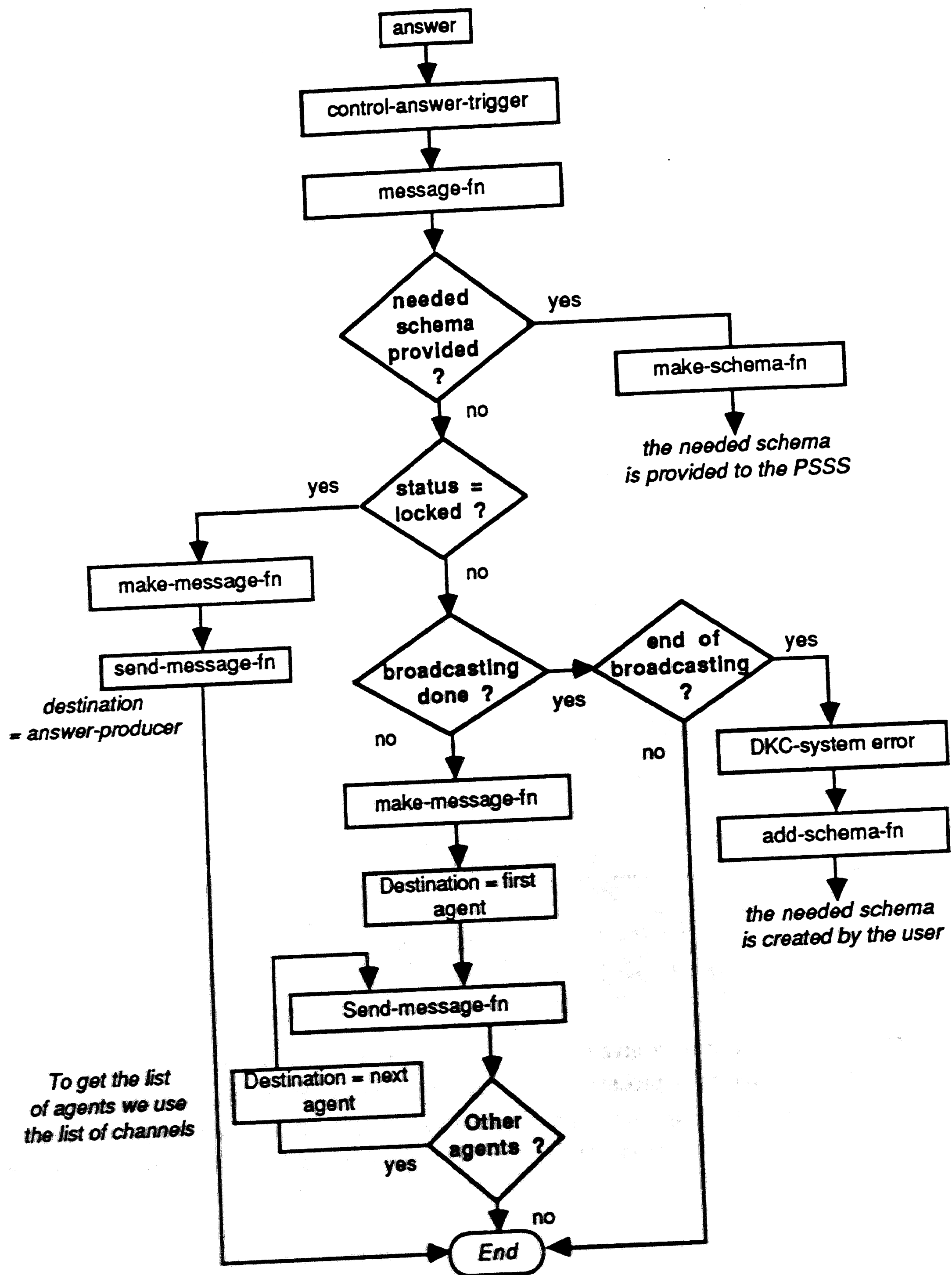


Figure 6-9: Answer control sequence

create a new message (using broadcasting) or will return an error message if it is impossible to g[...] the schema (in this case, we have already used broadcasting).

Schema 6-10: New-answer

New-answer		
SLOT	FACET	VALUE
Is-a	*Restriction:*	event
Message-queue	*Restriction:*	answer-queue
Action	*Restriction:*	message-fn

6.7 Information Layer utilization example

In this section, we present an example of the Information Layer functionalities. This example is the demonstration scenario presented at the 2nd Center for Integrated Manufacturing Decision Systems (CIMDS) conference.

This scenario shows (a) accessing information from many parts of the enterprise and integration, (b) direct communication of heterogeneous agents, (c) broadcasting capability, (d) how information sent by one agent changes the decision of another.

This scenario uses six different agents in the system:

- one manager: responsible for preparing the process plan,
- one inventory manager: responsible for the stock management,
- and four scheduler: each one responsible for scheduling the activities of some machines (each scheduler is responsible for different machines).

Assuming the existence of the lower levels of the architecture (initialization, channels, mail-boxes, distribution, etc.) the manager gets an important order that he wants to expedite through the factory floor.

1. The manager looks up the process plan for the order. The process plan is displayed in the network form.
2. The manager controls the machine availability using a GANTT chart representation. The manager only possesses the machine-1 availability information. But when the GANTT chart display is triggered, messages are sent to acquire the availability of the other machines (machine-2 through machine-5). These machine schemata are owned by the schedulers and are shared schemata, i.e., they can be updated either by the manager or by the schedulers. But the searcher of the manager does not know the specific owner of each machine. Each one of the machine-2/machine-5 is owned by a different scheduler. They are the main owners of these schemata, the manager is a simple owner able to only trigger global update of these schemata through the main owner.
3. So a broadcasting is performed by the searcher of the manager and four information-search messages (one for each machine) are sent to each of the other agents of the system (the four scheduler and the inventory managers).
4. The schedulers send back the calendars (the inventory sends an answer with a nil status) which are displayed in the GANTT chart. These calendars are received through answers with a 't status. What is shown so far is the capability to access information from many parts of the enterprise and integrate it.

et

5. The GANTT chart shows the schedule of the activities in the order. The order requires also some material. The manager inquiries of the inventory agent delivery dates of the material. In fact the manager just inquiries for the delivery dates of the articles used for the realization of the different products scheduled and the searcher generates direct communication messages towards the inventory agent. In this case, the searcher knows the owner of these schemata.
6. The inventory agent sends messages of quantity and delivery dates. But delivery dates necessitate delay in order processing (say, all activities must be shifted by one day)
7. GANTT chart displaying the shifting (schedule change) that resulted from the information received from the inventory is sent as updating messages to the main owner of each machine schema (at this point we know the main owner of each specific machine-2/machine-5 due to the result of the broadcasting: learning functions have been triggered).
8. Each scheduler then executes the global updating of the new schedules towards the users of these schedules. This illustrates the updating activity for information consistency.

7. Conclusion

The Enterprise Management Network is designed to facilitate the integration of heterogeneous functions distributed geographically. Integration is supported first by having the network play a more active role in the accessing and communication of information, and second by providing the appropriate protocols for the distribution, coordination and negotiation of tasks and outcomes.

As described in this paper, the Data Layer provides the ability to perform "standard" SQL-like queries across the network. The Information Layer provides a node with the ability to "invisibly" access information anywhere in the network, without explicitly referring to its location or its retrieval.

Our design of the Enterprise Management Network has the following characteristics:

- **Modular layered architecture**: as we have defined six levels of descriptions for a decentralized system, we can implement in a specific case either part of or the complete architecture.
- **High level decentralized communication system** which flexibly supports cooperative decision making: our structure includes a decentralized communication system which, using the frame based structure, allows the exchange information (schemata) between decentralized EMN-nodes.
- **User transparent**: the decentralized communication system is implemented in each EMN-node of the decentralized system. It has the capability to provide the needed information to the EMN-node. As this communication system is not specific to a particular EMN-node, it has been defined as a shell. The EMN-node does not have to know where to get the needed information. The communication system has the rules and capability to play this role. In our specific implementation of this communication system, the trigger of the information search is the CRL[19] command: (GET-VALUE schema-name slot-name)[20].
- **Declarative layer** specification provided by the frame based representation. Each EMN-node has its own local knowledge and data base.
- **Accessibility** of information to different parts of the organization. Each EMN-node has translation mechanisms to enable communication with others.
- **Understandability** of information through a common communication language.
- **Awareness** of problems and communication to appropriate EMN-nodes using a communication schema.
- **Focussed** information dissemination.
- **Responsiveness** of EMN-nodes through rules and translation mechanisms.
- **Flexibility** of communication due to support for many types of interaction and of representation through a frame based representation.

The current implementation of these three first levels of the Enterprise Management Network Architecture are described in [34]. Our future work concerns the specification and implementation

[19]Carnegie Representation Language (CRL) is a registered trademark of Carnegie Group Inc.

[20]For example (GET-VALUE 'machine 'capacity). In this case, "machine" is the schema name and "capacity" is the slot name. If the value is available in the Knowledge Base of the EMN-node, it is returned. If not, one or more messages are generated by the decentralized communication system.

of the three next layers: coordination, organization and market. These layers will support distributed problem solving activities [12, 30, 25, 9, 17, 11, 3, 4, 36, 37, 13, 21]. They will be based on models [33, 32, 31, 14, 10, 38, 39] which will describe manufacturing architecture in terms of decision taken, links and inter-actions between the different members of this architecture. The design of these layers will be based on the work performed in [32].

Acknowledgement

Robert Frederking, Charles Marshall from Digital Equipment Corporation, and the rest of the CORTES and CARMEMCO projects have contributed through their comments to the development of this Enterprise Management Network Architecture.

We also would like to thank particularly Joe Mattis for all the help and advice he provided for the implementation of this system. We use as basis of our message passing function the KM interprocess message passing utility for VAXLisp he designed.

References

[1] Adler, M.R., and Simoudis, E.
Integrated Distributed Expertise.
In *Proceedings of 10th International Workshop on Distributed Artificial Intelligence.* Bandera, Texas, 1990.

[2] Alford, M.W., and all.
Distributed Systems - Methods and tools for specification.
Springer-Verlag, 1985.
Lecture notes in Computer Science 190.

[3] Barr, A., Cohen, P.R., and Feigenbaum, E.A.
The Handbook of Artificial Intelligence, Volume 4.
Addison-Wesley Publishing Company, Massachusetts, 1989.

[4] Bond, A.H., and Gasser, L.
Readings in Distributed Artificial Intelligence.
Morgan Kaufmann, 1988.

[5] Corkill, D.D., and Lesser, V.R.
The Use of Meta-Level Control for Coordination in a Distributed Problem Solving Network.
In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 748-755. Morgan Kaufmann Publishers, Inc., 95 First Street, Los Altos, CA 94022, 1983.

[6] Date, C.J.
An introduction to data base systems.
Addison-Wesley Publishing Company, Massachusetts, 1981.

[7] Date, C.J. and White, C.J.
A guide to SQL DS.
Addison-Wesley Publishing Company, Massachusetts, 1989.

[8] Davis, R., and Smith, R.G.
Negotiation as a Metaphor for Distributed Problem Solving.
Artificial Intelligence 20:63-109, 1983.

[9] Decker, K., and Lesser, V.
A Scenario for Cooperative Distributed Problem Solving.
In *Proceedings of 10th International Workshop on Distributed Artificial Intelligence.* Bandera, Texas, 1990.

[10] Doumeingts, G.
GRAI method: A design methodology for computer integrated manufacturing systems (in French: Methode GRAI: methode de conception des systemes en productique.
PhD thesis, Laboratoire GRAI, Universite de Bordeaux, Bordeaux, France, 1984.

[11] Durfee, E.H., and Lesser, V.R.
Using Partial Global Plans to Coordinate Distributed Problem Solvers.
In *Proceedings of 10th International Joint Conference on Artificial Intelligence.* Milan, Italy, 1987.

[12] Durfee, E.H., and Montgomery, T.A.
A Hierarchical Protocol for Coordinating Multiagent: An Update.
In *Proceedings of 10th International Workshop on Distributed Artificial Intelligence.* Bandera, Texas, 1990.

[13] Engelmore, R., and Morgan, T.
Blackboard Systems.
Addison-Wesley, 1988.

[14] Erkes, K., and Clark, M.
Public domain report number 1.
Technical Report, ESPRIT Project 418, Open CAM System, April 1987.

[15] Fox, M.S.
An Organizational View of Distributed Systems.
IEEE Transactions on Systems, Man, and Cybernetics SMC-11(1):70-80, 1981.

[16] Fox, M.S., and Sycara, K.
Overview of the CORTES project: a Constraint Based Approach to Production Planning, Scheduling and Control.
Proceedings of the Fourth International Conference on Expert Systems in Production and Operations Management. , May, 1990.
Submitted for publication.

[17] Gasser, L., and Huhns, M.N.
Distributed Artificial Intelligence, Volume II.
Pitman Publishing & Morgan Kaufmann Publishers, 1989.

[18] Gasser, L., Braganza, C., and Herman, N.
MACE: A Flexible Testbed for Distributed AI Research.
In Michael N. Huhns (editor), *Distributed Artificial Intelligence*, chapter 5, pages 285-310. Pitman Publishing & Morgan Kaufmann Publishers, 1987.

[19] Hayes-roth, F.
Towards a framework for distributed AI.
1980
In: Randy Davis Ed, Report on the workshop on distributed AI, SIGART Newsletter.

[20] F. Hayes-Roth and V.R. Lesser.
Focus of Attention in a Distributed Logic Speech Understanding System.
In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 27-35. 1977.

[21] Hayes-Roth, B.
A Blackboard Architecture for Control.
Artificial Intelligence 26 , 1985.

[22] Huhns, M.N.
Distributed Artificial Intelligence.
Pitman Publishing & Morgan Kaufmann Publishers, 1987.

[23] Huhns, M.N., Bridgeland, M.L., and Arni, N.V.
A DAI Communication Aide.
In *Proceedings of 10th International Workshop on Distributed Artificial Intelligence.* Bandera, Texas, 1990.

[24] Hynynen, N.J.
A framework for coordination in distributed production management.
PhD thesis, Acta Polytechnica Sandinavica, Helsinky, Finland, 1988.

[25] Klein, M.
Supporting Conflict Resolution in Cooperative Design Systems.
In *Proceedings of 10th International Workshop on Distributed Artificial Intelligence.* Bandera, Texas, 1990.

[26] *Knowledge Craft*
Carnegie Group Inc, Five PPG place, Pittsburgh PA 15222, 1985.

[27] Lesser, V.R.
Cooperative Distributed Problem Solving and Organization Self-Design.
SIGART Newsletter :46, October, 1980.

[28] Lusardi, F.
The database Experts' guide to SQL.
Mc Graw Hill, New York, NY, 1988.

[29] Mullender, S.
Distributed Systems.
ACM Press, New York, N.Y., 1989.

[30] Parunak, H.V.D.
Toward a Formal Model of Inter-Agent Control.
In *Proceedings of 10th International Workshop on Distributed Artificial Intelligence.* Bandera, Texas, 1990.

[31] Roboam, M., Doumeingts, G., Dittman, K., and Clark, M.
Public domain report number 2.
Technical Report, ESPRIT Project 418, Open CAM System, October 1987.

[32] Roboam, M.
Reference models and analysis methodologies integration for the design of manufacturing systems (in French: Modeles de reference et integration des methodes d'analyse pour la conception des systemes de production.
PhD thesis, Laboratoire GRAI, Universite de Bordeaux, Bordeaux, France, 1988.

[33] Roboam, M., Fox, M.S., and Sycara, K.
Enterprise Management Network Architecture - The Organization Layer.
Technical Report CMU-RI-TR-90-22, CIMDS, Carnegie Mellon University, Pittsburgh, PA, 1990.

[34] Roboam, M., and Fox, M.S.
Distributed communication system: user manual.
Technical Report, CIMDS, Carnegie Mellon University, Pittsburgh, PA, 1990.

[35] Simon, H.A.
Model of man.
John Wiley, 1957.

[36] Sycara, K.
Resolving Goal Conflicts via Negotiation.
In *Proceedings of the Seventh National Conference on Artificial Intelligence [AAAI-88].* 1988.

[37] Sycara, K. and Roboam, M.
Intelligent Information Infrastructure for Group Decision and Negotiation Support of Concurrent Engineering.
In *Proceedings of the 24th Hawaii International Conference on System Sciences.* 1991.

[38] Tardieu, H., Rochfeld, A., and Colletti, R.
La methode Merise, principes et outils.
Paris, Les editions d'Organisation, 1983.

[39] Tardieu, H., Rochfeld, A., Colletti, R., Panet, G., and Vahee G.
La methode Merise, demarche et pratiques.
Paris, Les editions d'Organisation, 1985.