Intelligent Scheduling in Distributed Computing Systems

Virginia M. Lo
and
David Chen

CIS-TR-86-14
April 15, 1987

Department of Computer and Information Science
University of Oregon

Abstract

Our research into the problem of scheduling in distributed computing systems indicates that several techniques and tools from the area of "expert systems" can be successfully adapted for use in the design of a smart distributed scheduler. In this paper we look at expert system approaches to the representation of imprecise knowledge, techniques for reasoning about imprecise and unreliable knowledge, and means for handling knowledge accumulated over time. We show that that these are precisely the types of knowledge a distributed systems scheduler must deal with in order to make scheduling decisions and we give examples of the use of these techniques in the realm of task assignment and task migration algorithms. We then describe a task migration algorithm we have designed which utilizes rule based programming and expert systems techniques to deal with out-of-date and thus potentially unreliable data in system load tables.

Intelligent Scheduling in Distributed Computing Systems

1. Introduction

Because of the many complexities inherent in the distributed computing environment, it is desirable that distributed system schedulers be designed in a way that gives them a degree of "intelligence" beyond that found in typical schedulers for uniprocessor systems. Distributed system schedulers must have the capacity to make sophisticated decisions based on complex knowledge about the dynamic behavior of the system and its loads. Distributed system schedulers must be extremely versatile and adaptable in order to respond to the diverse and rapidly changing needs of both users and the system itself.

Our research into the problem of scheduling in distributed computing systems indicates that several techniques and tools from the area of "expert systems" can be successfully adapted for use in the design of a smart distributed scheduler. The use of *rule-based programming* provides a very natural and powerful medium for expressing the type of rule-of-thumb heuristics that are characteristic of scheduling algorithms and for expressing scheduling rules in a hierarchical fashion as lower level rules and higher level *meta-rules*. In addition, expert systems techniques for dealing with *uncertain and complex knowledge* can aid schedulers in making numerous decisions and predictions based on information gathered locally and from other processors.

In this paper we take a look at specific techniques from expert systems technology that we have found can be profitably used in distributed scheduling algorithms. In section 2, we give a brief overview of the problem of scheduling in distributed computing systems. Section 3 defines rule-based programming and demonstrates its suitability for use in the design of heuristic scheduling algorithms. Section 4 gives background information on expert systems approaches to the representation of imprecise knowledge, techniques for reasoning about imprecise and unreliable knowledge, and means for handling complex knowledge, such as vast quantities of knowledge and history data. We show that these are precisely the types of knowledge that a distributed system scheduler must deal with in order to make scheduling decisions and we give examples of the use of these techniques for decision-making in the realm of task assignment and

task migration algorithms. Section 5 describes a scheduler we have designed for the preemptive migration of communicating tasks using expert system tools. The migration algorithm makes decisions about when, where, how, and which task to migrate based on imprecise and out of date information. Section 6 gives conclusions and areas for further research.

2. Scheduling in Distributed Computing Systems

The problem of scheduling in distributed computing systems has been an active area of research for over a decade. Surveys of this work can be found in [CHLE80, TaRe85]. To be very brief, in the distributed computing environment, distributed computations consist of loosely coupled collections of communicating tasks which together work towards a common goal. Scheduling of these computations or *task forces* consists of

- *task assignment*: the initial assignment of each task in the task force to a processor,
- *task migration*: the possible preemption of an executing task and transfer of that task to a different processor,
- *local task scheduling*: the multiprogramming of tasks on a particular processor.

In all phases of scheduling the overall goal is to maximize throughput by utilizing the strategies of maximizing parallelism for a given task force, maintaining load balancing among the processors over many executing task forces, and minimizing the overhead of interprocess communication (IPC) between tasks in a task force and of the scheduling algorithms themselves.

Algorithms for task assignment and task migration can be divided into two groups: (1) those based on well-defined mathematical models such as graph theory, integer programming, and deterministic scheduling theory, and (2) those which utilize intuitive, rule-of-thumb heuristics such as bidding, probing, and market-type negotiations. The use of expert systems techniques is especially appropriate for this latter group of scheduling algorithms although many of the techniques can be applied to the mathematical algorithms as well.

3. Rule Based Expert Systems

In this section we give a brief introduction to the concepts of rule based expert systems and illustrate the usefulness of rule based programming for distributed scheduling algorithms. A thorough treatment of expert systems can be found in [BuSh84, HaWL83, Fors84].

The basic structure of a rule based expert system consists of a *knowledge base*, which represents the domain knowledge in the form of domain facts, a *rule system*, that describes rules and heuristics associated with the domain expertise, and an *inference engine* that utilizes the rules and heuristics in the rule system and the domain facts in the knowledge base to approximate an expert's problem solving processes. The rules comprising the rule base describe relationships between the facts in the knowledge base and ways in which new facts can be deduced from existing facts. These rules take the form of IF-THEN statements. Expressions following the IF keyword are called *conditions* and contain a list of facts that must be *true* before this particular rule can be triggered. Statements following the THEN keyword are called *actions* or *conclusions* and indicate the functions to perform or new facts to assert when this particular rule is triggered and selected for execution.

The problem solving procedure involves the acceptance of input conditions which cause the activation or triggering of certain rules. The inference engine uses a *matching strategy* to collect all the triggered rules and a *control strategy* to select which of the triggered rules to execute. The most common matching strategies used by the inference engine are forward-chaining (data driven, bottom-up), backward-chaining (goal driven, top-down), or a combination that finds the solution using both forward and backward chaining.

The use of rules and heuristics to solve problems is one of the most distinctive features in expert systems languages. Unlike conventional *procedural* programming languages which only specify *how* to execute something, the expert systems languages are usually *declarative* and specify *what* to do to solve the problem. Futhermore, there is a clear separation between logic (rule system), data (knowledge base) and control (inference engine) in expert systems. Thus, the system is more modular and is easily extendable. In conventional languages, logic, data, and control are

intermixed so that it is difficult to modify the program.

Another advantage of expert systems for problem solving is the ability of the system to manipulate the problem description itself and to reason at multiple levels of abstraction. In particular, in addition to containing rules about the problem domain itself, the rule base may also contain *meta-rules* which are rules about the rules.

As mentioned earlier, rule-based programming is particularly suited to the design and implementation of the type of heuristic, rule-of-thumb algorithms that have been proposed for distributed systems scheduling, particularly in the domain of task migration [EaLZ85, RaSt85, NiXG85, Smit84]. Many of these algorithms are based on human models of negotiation and consist precisely of IF-THEN rules describing the appropriate action to take under specified conditions. For example, the sender initiated strategies for task migration described in [EaLZ85] can be described by IF-THEN statements which are easily translated into rules in a rule-based language:

Threshold Transfer Policy and Threshold Location Policy:

if the queue length of processor-i $>$ *T and probe_count* $<$ *PROBE_LIMIT,*
then randomly select a processor-j,
    send it a probe message, and
    increment probe_count by 1.

if the probe value returned by processor-j $<=$ *T,*
then migrate task to processor-j.

Similarly, the drafting protocol proposed in [NiXG85] for task migration can be expressed as a series of IF-THEN rules (details have been deliberately omitted):

if processor-i is in H-load state,
then send it a draft-request message.

if a draft-request message is received from processor-j,
then send processor-j a draft-age message.

if draft-age messages have been received from all processors,
then calculate draft-standard,
    compute maximum-draft-age.

if processor-i is in L-load state and draft standard is calculated
and processor-j draft-age equals maximum-draft-age,
then send a draft-select message to processor-j.

Thus, these rule-of-thumb heuristic algorithms are expressed very naturally in the medium of rule based programming.

The ability to specify meta-rules about the lower level domain rules provides schedulers with the ability to be flexible and highly adaptive to dynamic changes in the distributed system. Researchers in the area of task scheduling algorithms have come to the realization that no single scheduling algorithm is suitable for all types of task forces and under all types of system conditions. Meta-level control is needed for flexible and adaptable scheduling, i.e., to decide when to invoke various scheduling algorithms, which particular scheduling algorithm to use, and for tuning or parameterization of algorithms [RaSt86].

For example, regular rules may specify the operation of a specific task migration algorithm such as the threshold and drafting algorithms described above. Meta-rules, on the other hand, may contain knowledge about which regular rules are more useful under specific conditions:

if the global system load is currently high,
then use rules defining receiver initiated migration algorithm.

if the global system load is currently low,
then use rules defining sender initiated migration algorithm.

Similarly, meta-level control can be utilized to decide whether a faster but less optimal task assignment algorithm is called for versus whether the overhead of a complex optimal algorithm is worthwhile, how many iterations of a particular algorithm to execute, how often to collect state information for these algorithms, whether negotiations should be among all processors or a selected subset of processors, what threshold levels of critical parameters should trigger task migration, whether a centralized or decentralized algorithm should be utilized, and so forth.

Meta-level rules can also be used to provide control over the control in the expert system. Meta-rules for *conflict resolution* can specify which rule or rules should be allowed to fire in the event that many rules are triggered by the facts in the knowledge base. The use of meta-rules for

conflict resolution in the domain of distributed scheduling is discussed in more detail in the following sections about decision making under conditions of complex and imprecise knowledge.

4. Uncertain and Complex Knowledge

The scheduler in a distributed system must make decisions about where to place tasks, whether and when to migrate tasks, which tasks to migrate, etc. These decisions are based on information about the tasks to be scheduled (longevity, scheduling frequency, execution costs, communication costs and structure); and on information about the distributed system itself (available resources, system configuration, processor loads, message traffic levels, and system response time). This information is often highly dynamic in nature and is gathered from a variety of sources including compilers, the users, and local system monitors resident on each processor. Thus, the scheduler must make its decisions based on data that is complex, and data that is often imprecise, out-of-date, incorrect, or incomplete.

In general, uncertain information exists because the data we wish to measure is itself imprecise in nature, because the instruments we use to measure or transmit the data are imprecise or error prone, because the rules or heuristics we use to produce the data are uncertain or incorrect, or because external factors such as time delays produce inconsistent and out of date data. When making (scheduling) decisions with uncertain data, our conclusions can still result in reasonable performance levels with the aid of expert systems techniques for representing and reasoning about uncertainty. In the following sections, we discuss expert system approaches to types of uncertain knowledge distributed system schedulers need to handle: imprecise knowledge, unreliable knowledge, and incomplete knowledge. We also discuss a type of complex knowledge that is relevant to distributed scheduling decisions: history data. Because all these classes of knowledge overlap to some extent, many of the techniques we discuss are applicable to more than one class.

4.1. Imprecise Knowledge

Imprecise knowledge involves concepts or quantities whose meaning or value is subject to a range of interpretations depending on context, viewpoint, or external events. In the domain of scheduling in distributed systems, many of the criteria for scheduling decisions depend on the evaluation of imprecise concepts such as load balancing, high degree of parallelism, low interprocess communication costs, better response time, etc. Distributed system schedulers need means for representing these imprecise quantities and for reaching conclusions and making decisions based on imprecise knowledge.

For example, it is desirable to define a rule which describes the system state that triggers task migration:

> *if the current state of processor-i is heavily loaded,*
> *then initiate task migration algorithm.*

Here, we have described the state of a processor as *heavily* loaded, but exactly what do we mean when we say it is *heavily* loaded? Typical interpretations of the term *heavily* loaded include notions that the execution queue of the processor is more than *some length* or the average turnaround time of tasks assigned to that processor is less than *some tolerable time period.* Unfortunately, we have just explained an imprecise word *heavily* loaded with the equally imprecise words *some length* and *some period.* We could try to quantify our notions by using numeric values:

> *if the execution queue length of a processor is more than 10,*
> *or if the average turnaround time of task is more than 0.5 seconds,*
> *then the processor is* heavily *loaded.*

These figures may be derived from empirical studies or from common sense but nevertheless even these numerical values may not be precise enough due to different initial assumptions, poor experimental data, or simply due to faulty instruments used during the experiments. If the turnaround time is 0.45 seconds, then is the processor is no longer *heavily* loaded? We have to be very sure about the 0.5 second limit before we can make such a claim. If the 0.5 second limit is unreliable in the first place, we would have started with an incorrect assumption about average

turnaround time, and hence all calculations and heuristics based on this figure would also be adversely effected. Thus we see that there is impreciseness both in the notion *heavily* loaded and in the rule which defines load in terms of queue length 10 and average turnaround time 0.5 sec.

One of the most important developments in expert systems has been the treatment of imprecise knowledge in the reasoning process. Lotfi Zadeh [Zade78] proposed a system for this process which is called *fuzzy logic*. In this system, the description of an imprecise concept such as *the turnaround time is fast* would generate a set of possibility values such as *the turnaround time is more than 1 second with possibility of 0.1, the turnaround time is between 0.5 and 1 second with possibility of 0.3, and the turnaround time is less than 0.5 second with possibility of 0.6.* The context of the description is also important because a *fast* turnaround time is still slower than a *slow* CPU cycle.

Another approach to reasoning under imprecise knowledge was included in the MYCIN expert system [BuSh85]. MYCIN introduced a numerical system in which *certainty factors* (CFs) are assigned to rules to indicate degrees of belief about the rules and their conclusions. CFs can vary from a value of 1.0, a strong belief, to -1.0, a definite false, with fractional values in between. Using CFs, we can supplement the previous example to make it more precise.

> *if the execution queue length of a processor is more than 10,*
> *or if the average turnaround time of task is more than 0.5 seconds,*
> *then the processor is* heavily *loaded with CF of 0.8.*

Conclusions with different degrees of belief can be combined to yield a new CF for the resulting hypothesis by using predefined formulae based on probability theory.

Cohen [Cohe85] argues that numerical systems are not precise enough. Since the numbers are arbitrarily chosen values, their meanings can be interpreted differently under different circumstances. When a CF is derived as a combination of other CFs, the meaning of the resulting number is particularly difficult to interpret. Finally, numbers are not expressive of the variety of meanings that we wish to ascribe to data, rules, and conclusions. Cohen suggests a scheme of endorsements for data, rule conditions and rule conclusions with quantities such as *good, medium, bad* (for data), *exact, flexible, supportive* (for rule conditions), and *adequate, likely, unlikely* (for

rule conclusion). For example,

> *if the execution queue length of a processor is more than 10,* {*exact*}
> *or if the average turnaround time of tasks is more than 0.5 seconds,* {*flexible*}
> *then the processor is* heavily *loaded with CF of 0.8.*

If the data regarding queue length and average turnaround time have *good* endorsements, the rule conclusion would receive an endorsement of *likely*. Conversely, if the data had *bad* endorsements associated with them, the rule conclusion would be endorsed as *unlikely*.

4.2. Unreliable Knowledge

Unreliable knowledge exists when there is some degree of doubt about the correctness of the knowledge. In the distributed computing environment, unreliability may stem from conclusions based on conflicting data from more than one source, and from data being out-of-date due to message transmission delays.

For example, suppose processor loads are measured in terms of several performance criteria:

> *if turnaround time* $<$ *0.5 sec, then the processor load is low.*
> *if processor queue length* $>$ *10 tasks, then the processor load is high.*
> *if message queue lengths* $>$ *10 messages, then the processor load is high.*

It is possible that the facts in the knowledge base would trigger all three rules, resulting in conflicting conclusions.

In general, meta-rules can be used to deal with conflicts by specifying priorities among the rules:

> *turnaround time is a better indicator of processor load*
> *than processor queue length or message queue length.*

The MYCIN project approaches conflict resolution using CFs and a meta-level heuristic called *unity path* which is hardwired into the control strategy. When faced with many possibilities to choose from, *unity path* selects the conclusion with the most definite belief (i.e., CF is close to -1 or +1).

The next example illustrates unrealiability due to out-of-date information. For both task assignment and task migration algorithms, it is necessary to keep track of the load status of each

processor in order to decide which processors are candidates to receive new or migrated tasks for execution. This type of information can be kept in a local load table with an entry for each processor denoting its status as *heavy, light, or normal* [NiXG85]. Since loads are dynamically changing at all times, the table must be updated by messages received from the processors. At any given moment of time, the status of a processor as reflected in the load table may not accurately represent its true current status. Thus, a simple rule such as

> *if processor-i's load status is low,*
> *then processor-i is a target for migration.*

may cause migration to a processor that cannot accept the task because the load status is no longer low.

The use of CF to deal with uncertain information can be extended to the notion of reliability factors (RF) which reflect the degree of reliability of data, rules, and conclusions. The reduced reliability of out-of-date information can be handled by periodically decrementing the RF of time-critical data in a manner similar to the handling of pages in LRU approximation algorithms for memory managment. Also, if a processor has just received a new message from another processor, all other data received before this new message is older than it.

> *if a new message has arrived from processor-i,*
> *then decrement the RFs of all time-critical messages from processor-i.*

The RF can be manipulated to respond to additional factors such as conflicting conclusions and outright false data. For example,

> *if a task is migrated to a processor which has previously indicated*
> *that it was lightly loaded, but the task is later rejected by the*
> *processor due to its heavy load,*
> *then reduce the RF of the processor status.*

4.3. Incomplete Knowledge

Often a scheduler needs to make a decision when some data are missing or incomplete. One way to resolve this situation is by not making any decision. Hence when a processor needs to choose a task to migrate, do not choose any task because some of the required data are not

available yet. A more plausible alternative is to make a decision but also assert it with a CF which denotes the fact that this decision is not entirely accurate because not all the data are present when the decison was made.

Cohen has built a similar kind of mechanism in SOLOMON [Cohe85] that uses the rule endorsement model described above to produce rule conclusions when not all the facts are present. The endorsements used in SOLOMON are symbolic ratings rather than numerical values. For example, to choose a particular processor for a task to migrate to, we have a rule with various endorsements in braces:

if the state of processor-1 is lightly loaded {necessary}
and the cost of migration to processor-1 is cheap {necessary}
and this task communicates with other tasks on processor-1 {supportive}
and the state of processor-1 is reliable {supportive}
then migrate task to processor-1.

Suppose the rule condition about the reliability of processor-1 is not available when the decision is required. Rather than waiting for the condition to appear, we can use the endorsement provided in the rule to reach a decision. Since the necessary conditions are both present, and there is support from the third condition, we can go ahead and conclude that the task should be migrated to processor-1. However, because not all the conditions are satisfied, the conclusion is reached with *may-be-too-general* or *not adequately endorsed* endorsemement. It is now up to other rules to decide if this conclusion can be accepted.

4.4. History Data

History data is information about a particular parameter or situation that is sampled and recorded over time. The correct management of history data can aid schedulers in many types of decisions, in the selection of appropriate algorithms, in the analysis of bottlenecks, and in the prediction of future workloads.

For example, in the typical computing environment, certain computations can be classified as repeatedly scheduled task forces. This group includes systems utilities such as editors, compilers, debuggers, and frequently used applications programs such as statistical packages and

database programs. Information about certain characteristics of these task forces is needed by the scheduler in order for it to decide where and when to execute the tasks. For example, most task assignment algorithms use estimates of execution costs, intertask communication costs and patterns, I/O and other resource requests. The quality of this information can be improved if the system monitor accumulates a history profile of these frequently scheduled computations.

With the aid of history data, the scheduler can be smarter by predicting future performance behavior. For example, in an interactive session, if a user repeats the *edit-compile-run* sequence several times in a row, our scheduler should be able to predict with some confidence (ie. CF) that after an *edit* task, the next one will be *compile* and after that would be *run* again. Based on this knowledge, the scheduler can perform task assignment in advance and possibly move load modules or data files to appropriate processors. In addition, the scheduler can use this information in its assessment of processor loads in the near future.

In order to reason about data that is accumulated over time, it is necessary to have a means for representing history data. History data, in general, can be stored either quantitatively or qualitatively. In the quantitative approach past data are stored in a long list. Whenever history data is required for a decision, rules examine the list exhaustively to produce the result. In the qualitative approach a simple symbolic datum such as an average or weighted average is stored to represent a summary of the past history. Of course, the qualitative approach requires much less storage area to keep track of the history data but the stored data may not be as accurate as that of the quantitative approach. For example, when monitoring the load status of a processor, if its history data for the past 10 time units indicates a *high* load, but at time unit 11, load status changes to *low*, and at time unit 12, load status changes back to *high* again, what should we record as the qualitative history data for this processor state at time unit 13? Although knowledge about the average behavior of the system is usually sufficient, occasionally we do need to know about the occurrence of abnormal behavior in the system. There is a clear cost in terms of time and space for the greater discrimination ability of a quantitative representation.

An illustration of an expert system approach to history knowledge is the Ventilator Manager (VM) program [FaKF80], based on MYCIN, that monitors and interprets data in the intensive care unit of a hospital. After a cardiovascular operation, VM monitors the patient's vital signs (pulse rate, respiration rate, blood pressure, etc.) in order to aid in the management of a mechanical ventilator for breathing assistance. VM's rules can reason about data over time. For example, rules exist which check the pulse rate of a patient over 15 minute intervals. If the pulse rate reaches a certain value during the testing period, then certain conclusions are drawn and appropriate suggestionss are produced and printed to assist the doctor's decision. If VM suggests a lower level of breathing assistance, it will generate expectations for the values of the monitored vital signs. Hence, rules interpret current and history data to determine what actions to take and to predict what should happen in the future.

VM uses both quantitative and qualitative approaches in storing history data. Since the most critical period for a patient is about 1.5 hours, VM stores the most recent 1 hour data in a queue (quantitatively), and stores the history data 1 hour ago as a symbolic value (qualitatively). Unfortunately, VM does not fully use the certainty factors associated with each rule conclusion even though VM's design is based on MYCIN. In fact, uncertainty is incorporated implicitly in VM's knowledge base because the data monitored are chosen because they highly correlate with patients' conditions.

5. Task Migration Rules

We have designed a scheduler for task migration which uses expert system techniques to make its decisions using unreliable data. We treat task migration as the preemptive migration of tasks that are members of a task force and that may have already begun execution on the processors to which they are currently assigned. This approach differs from previous notions of task migration [EaLZ85, RaST85, NiXG85] which consider only independent tasks which do not communicate with other tasks and only at the time of task arrival. Since the cost of preemption and moving a task is very high, preemptive migration is suitable only for long lived and nonterminating computations such as OS servers and demons, process controllers or data base

management applications. However, many of the decisions relevant to preemptive migration are also necessary for non-preemptive migration of independent tasks.

We presume a heterogeneous distributed system with n fully connected processors. Each processor contains a smart scheduling module as well as a local system monitor. Task forces arrive at a particular processor and the tasks are all initially assigned to processors by the local scheduler. Later, if the processor is overloaded or has a task that communicates heavily with tasks in other processors, the scheduler will initiate migration of a task or tasks so that processor loads are balanced, or so that heavily communicating tasks reside in the same processor. Thus, the goal of task migration is to improve overall system throughput by balancing loads and by minimizing interprocess communication.

In order to achieve these goals, the smart scheduler must decide (1) when to migrate tasks, (2) how the migration should be carried out, (3) which task to migrate, and (4) where to migrate. Most of these decisions are based on the current status of the global system state which is determined from imprecise, unreliable, out of date and complex data. Below, we present some examples of our task migration rules and a simple scheme for evaluating the reliability of information about the loads on processors in the distributed system. We use rule-based programming because of the natural way it expresses the type of rule-of-thumb heuristics found in our algorithm and because of the convenience and clarity provided by meta-level rules.

5.1. When to Migrate

Task migration in distributed systems is triggered by changes in system state. In our system, we are concerned with the situation in which a processor's load exceeds a threshold value, or when the interprocess communication between two tasks exceeds a threshold value. These event-triggered actions can be expressed clearly in rule form.

```
MIGRATION-FOR-COMMUNICATION-RULE:
(IF (THE LOCATION OF ?TASK1 IS ?PROCESSOR)
AND (GREATER-THAN (THE COMMUNICATION-COST ?TASK1 TO ?TASK2)
                  (THE COMM-THRESHOLD OF ?PROCESSOR))
THEN (ASSERT (INITIATE-MIGRATION-WITH (?TASK1 ?TASK2))))
```

MIGRATION-FOR-LOAD-BALANCING-RULE:
(IF (GREATER-THAN (THE LOAD IN ?PROCESSOR) (THE LOAD-THRESHOLD OF ?PROCESSOR)
THEN (ASSERT (INITITATE-MIGRATION-WITH (THE ?TASK-LIST IN ?PROCESSOR)))

5.2. How to Migrate

In our system we use a simplification of results from a study by Eager, Lazowska, and Zahorjan [EaLZ85] regarding the relationship between task migration strategies and overall system load. In particular, we assume that under heavy overall system loads, it is better to use receiver-initiated task migration in which lightly loaded processors invite migration; while under light to moderate loads, it is better to use sender-initiated task migration in which heavily loaded processors initiate migration. (According to their studies, some other assumptions are necessary, but we will ignore these for now.)

Meta-rules are used to instruct our scheduler to select the preferred strategy based on system load. In general, the use of meta-rules gives the scheduler greater flexibility by allowing it to dynamically select algorithms; these meta-rules can be themselves modified or deleted at run time, providing a degree of adaptability that is difficult to provide in procedural languages.

USE-SENDER-INITIATED-ALGORITHM-*META*-RULE:
(IF (THE GLOBAL-STATE IS HIGH)
THEN (ASSERT (THE USE-ALGORITHM IS SENDER-INITIATED))
(DISABLE (NOT (SENDER-INITIATED-RULE-CLASS RULES)))
(ENABLE (SENDER-INITIATED-RULE-CLASS RULES)))

USE-RECEIVER-INITIATED-ALGORITHM-*META*-RULE:
(IF (THE GLOBAL-STATE IS (NOT HIGH))
THEN (ASSERT (THE USE-ALGORITHM IS RECEIVER-INITIATED))
(DISABLE (NOT (RECEIVER-INITIATED-RULE-CLASS RULES)))
(ENABLE (RECEIVER-INITIATED-RULE-CLASS RULES)))

5.3. Who to Migrate

The decision of which task or set of tasks to migrate depends on the current inter-processor communication costs between the two tasks that reside in different processors, frequency of communication with other tasks in the same processor, and anticipated communication and execution pattern of the migrating tasks. The selection of tasks also has to consider effects on the

balancing of processor loads in the system. The following rules select one or more candidates for migration.

```
SELECT-MIGRATION-TASK-HIGH-COMMUNICATION-RULE:
(IF (INITIATE-MIGRATION-WITH (?TASK1 ?TASK2))
AND (THE LOCATION OF ?TASK1 IS ?PROCESSOR1)
AND (THE LOCATION OF ?TASK2 IS ?PROCESSOR2)
AND (NOT.EQUAL ?PROCESSOR1 ?PROCESSOR2)
AND (THE COMMUNICATION OF ?TASK1 IN ?PROCESSOR1 IS LOW)
AND (THE COMM-FREQUENCY OF ?TASK1 IN ?PROCESSOR1 IS LOW)
THEN (ASSERT (THE MIGRATION-CANDIDATE IS ?TASK1 FOR COMMUNICATION)))
```

```
SELECT-MIGRATION-TASK-LOAD-BALANCING-RULE:
(IF (INITIATE-MIGRATION-WITH ?TASK-LIST)
AND (INCLUDES ?TASK IN ?TASK-LIST)
AND (THE LOCATION OF ?TASK IS ?PROCESSOR)
AND (THE COMMUNICATION OF ?TASK IN ?PROCESSOR IS LOW)
AND (THE COMM-FREQUENCY OF ?TASK IN ?PROCESSOR IS LOW)
THEN (ASSERT (THE MIGRATION-CANDIDATE IS ?TASK FOR LOAD-BALANCING)))
```

When many tasks are eligible for migration, it may not be desirable or possible to migrate all of them. Additional rules can be used to prioritize the final selection from many candidates. For example, a task is a better choice for migration if it fulfills both the criteria of load balancing and minimizing IPC. If no such task exists, the rules can specify that one of the criteria is preferred over the other, or that a task be selected randomly.

```
SELECT-FINAL-TASK-BOTH-ENDORSEMENT-RULE:
(IF (THE MIGRATION-CANDIDATE IS ?TASK FOR ?REASON1)
AND (THE MIGRATION-CANDIDATE IS ?TASK FOR ?REASON2)
AND (NOT.EQUAL ?REASON1 ?REASON2)
THEN (ASSERT (READY-TO-MIGRATE-TASK ?TASK)))
```

```
SELECT-FINAL-TASK-PREFER-COMMUNICATION-RULE:
(IF (THE MIGRATION-CANDIDATE IS ?TASK1 FOR COMMUNICATION)
AND (THE MIGRATION-CANDIDATE IS ?TASK2 FOR LOAD-BALANCING))
THEN (ASSERT (READY-TO-MIGRATE-TASK ?TASK1)))
```

5.4. Where to Migrate

Whether the reason for migration is load balancing or the minimization of IPC, the decision of *where* to migrate is determined by consulting the processor load table. This table stores the load status of all the processors in the system as *high, normal, or low* based on load status messages received from the processors. The contents of this table may not be accurate because of

physical and temporal limitations. We use a simple system of recency and reliability ratings to improve the selection decision.

```
SELECT-MIGRATION-SENDER-INITIATED-RULE:
(IF (THE STATE OF ?PROCESSOR IS (NOT HIGH))
    AND (THE RECENCY OF ?PROCESSOR IS RECENT)
    AND (THE RELIABILITY OF ?PROCESSOR IS RELIABLE)
    AND (THE USE-ALGORITHM IS SENDER-INITIATED)
THEN (ASSERT (THE MIGRATE-TO-CANDIDATE IS ?PROCESSOR FOR SENDER-INITIATED)))
```

The terms "recent" and "reliable" used in the previous rules represent a qualitative measurement of quantitative values called recency and reliability factors. The qualitative values are translated from quantitative values that range from 0 to 10, where 0 means very old or very unreliable and 10 means very recent and very reliable.

The recency factors are updated in an event-driven manner. Whenever a new load status message is received from a particular processor, the corresponding recency factor is set to the highest recency value (10). The recency factor for all other processors is decremented by the recency adjustment factor for each processor. This adjustment factor can be a preset static value or can be dynamically adjusted. This scheme eliminates the need to periodically adjust the recency factors because only relative recency information is needed to figure out if the load message is qualitatively recent. Note that the recency factors only give information about the ordering of message *arrivals* at a given processor. Unfortunately due to the distributed nature of this system, it is difficult to determine if a message from processor-i is actually *generated* before a message from processor-j . (We are currently refining this scheme to account for message transmission delays.)

Reliability factors are updated using information received after a task has been transfered to the desired destination. If the migrating task is rejected by a processor even though the load table indicated that that processor was a good candidate, the reliability factor will be decremented according to a reliability adjustment factor similar to the recency adjustment factors. Thus, reliability factors reflect the load stability of processors: if a processor's load fluctuates wildly, the load status messages will not be a reliable measurement of the actual load status.

6. Conclusion

It is clear that decision making in distributed computing systems can benefit from the use of techniques from AI to deal with uncertain and complex knowledge. The fact that there are common concerns in the areas of distributed computing and artifical intelligence has already been recognized through the emergence of the area called distributed artificial intelligence (DAI). Work in DAI has been concerned primarily with higher level issues related to problem solving involving multiple agents, such as planning, coordination, and problem representation. However, more recently there has been some attention to the use of AI techniques for lower level decisions involved in basic operating system functions such as scheduling. In particular, Pasquale [Pasq86] has taken an approach very similar to ours and has looked at the use of expert systems for the purposes of distributed system management.

There are many questions to be resolved if an expert system decision maker is to be used as an operating system component. We are currently concentrating on the issue of *time-related knowledge*, specifically out-of-date information and history information. We feel that by combining notions of time from distributed systems such as those proposed by Lamport [Lamp78] and Jefferson [Jeff86] with representations of time from the AI community, we should be able to increase the quality of decision-making for scheduling and other arenas of distributed systems management.

Bibliography

[BuSh85] Buchannan, B. G., and Shortliffe, E. H., eds., *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Reading, MA: Addison-Wesley Publishing Co., Inc., 1985.

[CHLE80] W. W. Chu, L. J. Holloway, M. T. Lan, and Kemal Efe, "Task Allocation in Distributed Data Processing," *IEEE Computer*, Nov. 1980, pp. 57-69.

[Cohe85] Cohen, P., *Heuristic Reasoning about Uncertainty: An Artificial Intelligence Approach.* Boston, MA: Pitman Publishing Inc., 1985.

[EaLZ85] Eager, D.L., Lazowska, E.D., and Zahorjan, J., "A Comparison of Receiver-Initiated and Sender-Initiated Dynamic Load Sharing", University of Washington Technical Report 84-04-01, Apr. 1984.

[EaLZ85] Eager, D.L., Lazowska, E.D., and Zahorjan, J., "Dynamic Load Sharing in Homogeneous Distributed Systems", University of Washington Technical Report 84-10-01, Oct. 1984.

[FaKF79] Fagan, L. M., Kunz, J. C., Feigenbaum, E. A., and Osborn, J. J., Representation of dynamic clinical knowledge: Measurement interpretation in the intensive care unit. In *Proceedings of 6th International Joint Conference on Artificial Intelligence* (Tokyo), 1979, pp. 260-262.

[Fors84] Forsyth, R., ed., *Expert Systems: Principles and Case Studies.* New York, N.Y.: Chapman and Hall, 1984.

[HaWL83] Hayes-Roth, F., Waterman, D. A., and Lenat, D. B., eds., *Building Expert Systems.* Reading, MA: Addison-Wesley Publishing Co., Inc., 1983.

[Jeff86] Jefferson, D.R., "Virtual Time", *ACM TOPLAS*, Vol. 7, No. 3, July 1985, pp. 404-425.

[Lamp78] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System", *CACM 21*, July 1978, pp. 558-565.

[NiXG85] Ni, L.M., Xu, C-W, and Gendreau, T.B., "A Distributed Drafting Algorithm for Load Balancing", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 10, October 1985, pp. 1153-1161.

[Pasq86] Pasquale, J., "Knowledge-Based Distributed Systems Management", University of California at Berkeley EECS Technical Report No. UCB/CSD 86/295, June 1986.

[RaSt85] Ramamritham, K. and Stankovic, J.A., "Dynamic Task Scheduling in Hard Real-Time Distributed Systems", *IEEE Software*, July 1984, pp. 65-75.

[RaSt86] Ramamritham, K. and Stankovic, J.A., "Meta-level Control in Distributed Real-Time Systems", private communication, 1986.

[Smit80] Smith, R. G., "The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver", *IEEE Transactions on Computers*, Vol. C-29. No. 12, Dec. 1980, pp. 1104-1113.

[TaRe85] Tanenbaum, Andrew and vanRenesse, R., "Distributed Operating Systems", *ACM Computing Surveys*, Vol. 17, No. 4, Dec. 1985, pp. 419-470

[Zade78] Zadeh, L. A. 1965. Fuzzy sets. *Information and Control* 8: 338-353.

——. 1978. Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets and Systems* 1: 3-28.