Distributed Real-Time Computing: The Next Generation[*,†]

Professor John A. Stankovic
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

January 3, 1992

Abstract

This paper presents the main issues facing next generation real-time distributed computing from a Computer Science perspective. In particular, the paper discusses scheduling, real-time kernels, communication and clock synchronization, architecture and fault tolerance, and artificial intelligence.

`KEYWORDS: distributed systems, real-time, real-time kernels, real-time scheduling, real-time architectures, real-time communication, real-time databases, real-time artificial intelligence, and fault tolerance.`

[*]This work was funded in part by the Office of Naval Research under contracts N00014-85-K-0389 and N00014-92-J-1048, and NSF under DCR-8500332 and CDA-8922572.

[†]Invited paper for *Journal of the Society of Instrument and Control Engineers of Japan.*

1 Introduction

Hard real–time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. Examples are command and control systems, process control systems, flight control systems, the space shuttle avionics system, flexible manufacturing applications, the space station, space-based defense systems, and large command and control systems. Any of these systems might be integrated with expert systems and other AI applications creating additional requirements and complexities. Most of the hard real–time computer systems are special purpose and complex, require a high degree of fault tolerance, and are typically embedded in a larger system. Also, real–time systems have substantial amounts of knowledge concerning the characteristics of the application and the environment built into the system. A majority of today's systems assume that much of this knowledge is available *a priori*, and hence are based on *static* designs. The static nature of many of these systems contribute to their high cost and inflexibility. Further, while many distributed real-time systems exist, e.g., [9, 22], these systems were designed and built prior to significant new research results in the areas of scheduling, fault tolerance, artificial intelligence, and predictability.

Next generation distributed, hard real–time systems must be designed to be *dynamic* and *flexible*, yet provide basic guarantees for safety critical aspects of the system. Their sophistication will be significantly greater than past systems. In designing and building these new systems we must not make the same mistakes and work with the same misconceptions that plagued past system design [14]. For example, the most common misconception is that real-time computing is fast computing. Approaching real-time computing from this perspective is fraught with difficulties as detailed in [14]. Unfortunately, too much recent work in distributed real-time systems (mistakenly) treats real-time computing as fast computing. In this (short) invited paper, the main issues facing next generation real-time distributed computing are presented, emphasizing the need for predictability [15] rather than speed alone. We organize the discussion into the following real-time areas: scheduling, real-time kernels, communication and clock synchronization, architecture and fault tolerance, databases, and artificial intelligence.

2 Scheduling

Real-time scheduling results in recent years have been extensive. Theoretical results have identified worst case bounds for dynamic on-line algorithms, and complexity results have been produced for various types of assumed task set characteristics. Queueing theoretic analysis has been applied to soft real-time systems covering algorithms based on real-time variations of FCFS, earliest deadline, and least laxity. We have seen the development of scheduling results for imprecise computation (a situation where tasks obtain a greater value the longer they execute up to some maximum value). More applied scheduling results have also been produced with an extensive set of improvements to the rate monotonic algorithm (this includes the deferrable server and sporadic server algorithms [13]), techniques to address the problem of priority inversion [10], and a set of algorithms that perform dynamic on-line planning [7, 23]. We have also seen practical application of *a priori* calculation of static schedules to provide what is called 100% guarantees for critical tasks. While these *a priori* analyses are very valuable, system designers better not be lulled into thinking that 100% guarantees mean that no scheduling error can occur. It is important to know that these 100% guarantees are based on many and often times unrealistic assumptions. If the assumptions are a poor match for what can be expected from the environment (more and more likely in a distributed environment), then even with 100% guarantees the system will indeed miss deadlines. Hence, a key issue is to choose an algorithm whose assumptions provides the greatest coverage over what *really* happens in the environment. For all these scheduling results outlined above, the trend has been to deal with slightly more and more complicated task set and environment characteristics (e.g., multiprocessing and distributed computing and task with precedence constraints). While many interesting scheduling results have been produced, the state of the art still provides piecemeal solutions. Many realistic issues have not yet been addressed in an integrated and comprehensive manner.

What is still required are analyzable scheduling approaches (it may be a collection of algorithms) that are comprehensive and integrated. For example, the overall approach must be comprehensive enough to handle:

- preemptable and non-preemptable tasks,
- periodic and non-periodic tasks,
- tasks with multiple levels of importance (or a value function),
- groups of tasks with a single deadline,
- end-to-end timing constraints,
- precedence constraints,
- communication requirements,
- resource requirements,
- placement constraints,

- fault tolerance needs,
- tight and loose deadlines, and
- normal and overload conditions.

The solution must be integrated enough to handle the interfaces between:

- CPU scheduling and resource allocation,
- I/O scheduling and CPU scheduling,
- CPU scheduling and real-time communication scheduling,
- local and distributed scheduling [2, 8, 19], and
- static scheduling of critical tasks and dynamic scheduling of essential and non-essential tasks.

3 Real-Time Kernels

One focal point for next generation real-time systems is the operating system. The operating system must provide basic support for predictably satisfying real-time constraints, for fault tolerance and distribution, and for integrating time-constrained resource allocations and scheduling across a spectrum of resource types including sensor processing, communications, CPU, memory, and other forms of I/O. Towards this end, at least three major scientific issues need to be addressed.

- The *time dimension* must be elevated to a central principle of the system and should not be simply an afterthought. An especially perplexing aspect of this problem is that most system specification, design and verification techniques are based on abstraction — which ignores implementation details. This is obviously a good idea; however, in real-time systems, timing constraints are derived from the environment and the implementation. This dilemma is a key scientific issue.
- The basic paradigms found in today's general purpose distributed operating systems must change. Currently, they are based on the notion that application tasks request resources as if they were random processes; operating systems are designed to expect random inputs and to display good average-case behavior. The new paradigm must be based on the delicate balance of *flexibility* and *predictability*: the system must remain flexible enough to allow a highly dynamic and adaptive environment, but at the same time be able to predict and possibly

avoid resource conflicts so that timing constraints can be met [17]. This is especially difficult in distributed environments where layers of operating system code and communication protocols interfere with predictability.

- A highly *integrated and time-constrained resource allocation approach* is necessary to adequately address timing constraints, predictability, adaptability, correctness, safety, and fault tolerance. For a task to meet its deadline, resources must be available *in time*, and events must be ordered to meet precedence constraints. Many coordinated actions are necessary for this type of processing to be accomplished on time. The state of the art lacks solutions to this problem.

Existing practices for designing, implementing, and validating real-time systems of today are still rather *ad hoc*. It is often the case that existing real–time systems are supported by stripped down and optimized versions of timesharing operating systems. To reduce the run-time overheads incurred by the kernel and to make the system *fast*, the kernel underlying a current real–time system

- has a fast context switch,
- has a small size (with its associated minimal functionality),
- is provided with the ability to respond to external interrupts quickly,
- minimizes intervals during which interrupts are disabled,
- provides fixed or variable sized partitions for memory management (i.e., no virtual memory) as well as the ability to lock code and data in memory, and
- provides special sequential files that can accumulate data at a fast rate.

To deal with timing requirements the kernel,

- maintains a real-time clock,
- provides a priority scheduling mechanism,
- provides for special alarms and timeouts, and
- tasks can invoke primitives to delay by a fixed amount of time and to pause/resume execution.

In general, the kernels perform multi-tasking; inter-task communication and synchronization are achieved via standard, well-known primitives such as mailboxes, events, signals, and semaphores.

In real-time computing these features are also designed to be *fast*. However, fast is a relative term and not sufficient when dealing with real–time constraints. Nevertheless, many real-time system designers believe that these features provide a good basis upon which to build real–time systems. Others believe that such features provide almost no direct support for solving the difficult timing problems and would rather see more sophisticated kernels that directly address timing and fault tolerance constraints.

One key issue that comes up over and over again is the need to provide predictability. However, more than lip service must be supplied. Predictability requires bounded operating system primitives, some knowledge of the application, proper scheduling algorithms (see previous discussion), and a viewpoint based on a *team* attitude between the operating system and the application. For example, simply having a very primitive kernel that is itself predictable is seen as only the first step. What is needed is more direct support for developing predictable and fault tolerant real-time applications. One aspect of this support comes in the form of scheduling algorithms. For example, if the operating system is able to perform integrated CPU scheduling and resource allocation in a planning mode so that collections of cooperating tasks can obtain the resources they need, at the right time, in order to meet timing constraints, this facilitates the design and analysis of real-time applications [17]. Further, if the operating systems retains information about the importance of a task and what actions to take if the task is assessed as not being able to make its deadline, then a more intelligent decision can be made as to alternative actions, and graceful degradation of the performance of the system can be better supported (rather than a possible catastrophic collapse of the system if no such information is available). Kernels which support retaining and using semantic information about the application are sometimes referred to as *reflective* kernels [16].

Real-time kernels are also being extended to operate in highly cooperative multiprocessor and distributed system environments. This means that there is an end-to-end timing requirement (in the sense that a set of communication tasks must complete before a deadline), i.e., a collection of activities must occur (possibly with complicated precedence constraints) before some deadline. Much research is being done on developing time constrained communication protocols to serve as a platform for supporting this user level end-to-end timing requirement. However, while the communication protocols are being developed to support host-to-host bounded delivery time, using the current operating system paradigm of allowing arbitrary waits for resources or events, or treating the operation of a task as a *random process* will cause great uncertainty in accomplishing the application level end-to-end requirements. As an example, the Mars project [3], the Spring project [17], and a project at the University of Michigan [11] are all attempting to solve this problem. The Mars project uses an *a priori* analysis and then statically schedules and reserves resources so that distributed execution can be guaranteed to make its deadline. The Spring approach support dynamic requests for real-time virtual circuits (guaranteed delivery time) and real-time datagrams (best effort delivery) integrated with CPU scheduling so as to guarantee the application level end-to-end timing requirements. The Spring project uses a distributed replicated memory based on a fiber optic ring to achieve the lower level predictable communication properties. The Michigan work also supports dynamic real-time virtual circuits and datagrams, but their work is based on a general multi-hop communication subnet.

Research is also being done on developing real-time object oriented kernels [20, 21] to support the structuring of distributed real-time applications [12]. As far as we know, no commercial products of this type are available.

The diversity of the applications requiring predictable distributed systems technology will be significant. To handle this diversity, we expect the distributed real-time operating systems must use an *open system* approach. It is also important to avoid having to rewrite the operating system for each application area which may have differing timing and fault tolerance requirements. A library of real–time operating system objects might provide the level of functionality, performance, predictability, and portability required. We envision a Smalltalk like system for hard real–time, so that a designer can tailor the OS to his application without having to write everything from scratch. In particular, a library of real-time scheduling algorithms should be available that can be plugged in depending on the run time task model being used and the load, timing, and fault tolerance requirements of the system.

4 Communication and Clock Synchronization

In the design of distributed real–time systems, there is a need for communication protocols that provide for deterministic behavior (guarantees) of the communicating components as well as those that provide a best effort (by a deadline) delivery service. The deterministic service requires protocols that result in bounded message communication delays where the bound is low compared to timing requirements. Protocols providing such service are sometimes called real-time virtual circuits. Such protocols are being developed or exist for arbitrary mesh networks, rings, TDMA-like busses, and globally replicated memory. Most of the work in this area assumes that messages requiring the deterministic service are periodic or occur rarely (e.g., alarms), and are statically specifiable. These requirements are likely to prove inadequate for autonomous real-time systems of the future. For example, these new systems will be highly dynamic and have strong cooperation requirements typical of distributed problem solving software inducing richer communication patterns than simple periodic messages. The dynamics might require that a task dynamically seek a guarantee that a particular message will arrive in time, *before* that task commits to beginning an action.

For the best effort service (sometimes called real-time datagrams), a number of advances have been made as extensions to collision based protocols. These extensions provide *time aware* scheduling and are more predictable than pure CSMA protocols. However, this service is unreliable in that the timing constraint of a message may be missed. These protocols work by attempting to transmit messages that would minimize the number of messages whose deadlines are not met. Usually, the protocol used mimics earliest deadline first or least laxity first scheduling. These protocols only approximate these algorithms because no one node has a complete picture of all pending messages.

Complicating factors in developing both types of services include the need to support high speed networks, the integration of the low-level protocols with the operating system kernel, I/O modules, and application modules, as well as the inclusion of fault-tolerant features. Generally, the protocols for real-time communication must be supported at the kernel level and deep and expensive protocol stacks are avoided. Along these lines, researchers are proposing various local area network architectures for communication in distributed real-time systems that are efficient, allow specification of service requirements, and provide mechanisms to achieve those service requirements [1].

Distributed real-time systems must have a common view of time. This requires clock synchronization. While there have been many algorithms for clock synchronization, it is likely that the demanding requirements of hard real-time systems will require hardware supported clock synchronization such as provided in [4].

There has also been significant work in developing reliable atomic broadcasts with varying semantics and performance characteristics. For example, some broadcasting protocols support FIFO semantics, others a causal ordering, and yet others are tailored to determine group membership. However, all of these protocols are expensive and do not adequately support predictable timing properties. Since broadcasting is such an important underlying facility for many things including fault tolerance, application cooperation, and state consistency, it is important to develop such solutions for real-time systems.

5 Architecture and Fault Tolerance

Hard real–time systems are usually quite special purpose. Architectures to support such applications tend to be special purpose too. The current trend is one in which more "off-the-shelf" components are being used to produce more generic architectures [11, 18], rather than the previously developed highly special purpose architecture. As an example, consider the SpringNet architecture.

SpringNet [18] is a physically distributed system composed of a network of three multiprocessors each running the Spring Kernel. Each multiprocessor currently contains four application processors, one system processor, and a I/O subsystem. Application processors execute previously guaranteed processes as specified in the execution plan constructed by the scheduler executing on one or more system processors. The system processor [1] offloads the scheduling algorithm and other OS overhead from the application processors both for speed, and **so that external interrupts and OS overhead do not cause uncertainty in executing guaranteed processes.** The I/O subsystem is partitioned away from the Spring Kernel and it handles non-critical I/O, slow I/O devices, and fast sensors.

Each of the nodes is connected via two networks. First, there is an ethernet to support non real-time traffic. Second, a fiber optic register insertion ring connects 2 Mbyte memory boards on each node, supporting 2 Mbytes of replicated memory. This provides a shared memory model for this 2 Mbytes (of physically distributed but logically centralized memory). Each node also has at least 20 Mbtyes of non-replicated memory (4 Mbytes per processor thereby presenting a local memory model for the rest of the memory of the multiprocessor). The replicated memory is implemented via the off-the-shelf product called Scramnet. This replicated memory together with communication software and scheduling constraints are used to provide end-to-end predictable performance. The replicated memory can also be exploited for fault tolerance. In other words, important data structures and other information at a given node are written to the replicated memory board and are then automatically reflected in the replicated memory of all the nodes on the ring. This duplication of information is useful in recovering from several classes of node failure faults including power loss, bus failure, and Scramnet failures that do not cause corruption of the replicated memory. Of course, to enhance fault tolerance it is possible to add other *parallel* register insertion rings, each supporting its own replicated memory.

The SpringNet architecture can scale by connecting rings of replicated memory in an n-dimensional grid. For example, a 2-dimensional grid would have one replicated memory register insertion ring for each row and another replicated memory register insertion ring for each column. Even though the SpringNet architecture resembles a multicomputer, it is important to note that the SpringNet architecture can be physically distributed, limited only by the maximum fiber optic ring size.

Another aspect of architecture for real-time computing is the facility with which the worst case execution time can be calculated. Worst-case execution times of programs are dependent on the system hardware, the operating system, the compiler used, and the programming language used. Many hardware features that have been introduced to speed-up the average case behavior of programs pose problems when information about worst case behavior is sought. For instance, the ubiquitous caches, pipelining, dynamic RAMs, and virtual (secondary) memory, lead to highly nondeterministic hard-

[1]Ultimately, system processors could be specifically designed to offer hardware support to our system activities such as guaranteeing processes. We are currently investigating this option.

ware behavior. Similarly, compiler optimizations tailored to make better use of these architectural enhancements as well as techniques such as constant folding contribute to poor predictability of code execution times. System interferences due to interrupt handling, shared memory references, and preemptions are additional complications. In summary, any approach to the determination of execution times of real-time programs has many complexities.

Many real-time system architectures consist of multiprocessors, networks of uniprocessors, or networks of uni- and multi-processors. Such architectures have potential for high fault tolerance, but are also much more difficult to manage in a way such that deadlines are predictably met. Fault tolerance must be designed in at the start, must encompass both hardware and software, and must be integrated with timing constraints. In many situations, the fault tolerant design must be static due to extremely high data rates and severe timing constraints. Ultrareliable systems need to employ proof of correctness techniques to ensure fault tolerance properties. Primary and backup schedules computed off–line are often found in hard real–time systems. We also see new approaches where on–line schedulers predict that timing constraints will be missed, enabling early action on such faults. Dynamic reconfigurability is needed but little progress has been reported in this area. Also, while considerable advance has been made in the area of software fault-tolerance, techniques that explicitly take timing into account are lacking.

Since fault tolerance is difficult, the trend is to let experts build the proper underlying support for it. For example, implementing checkpointing, reliable atomic broadcasts, logging, lightweight network protocols, synchronization support for replicas, and recovery techniques, and having these primitives available to applications, then simplifies creating fault tolerant applications. However, many of these techniques have not carefully addressed timing considerations nor the need to be predictable in the presence of failures. Many real-time systems which require a high degree of fault tolerance have been designed with significant architectural support but the design and scheduling to meet deadlines is done statically, with all replicas in lock step. This may be too restrictive for many future applications. What is required is the integration of fault tolerance and real-time scheduling to produce a much more flexible system. For example, the use of the imprecise computation model, or a planning scheduler (such as the Spring scheduling algorithm), gives rise to a more flexible approach to fault tolerance than static schedules and fixed backup schemes.

6 Databases

A real-time database is a database system where (at least some) transactions have explicit timing constraints such as deadlines. In such a system, transaction processing must satisfy not only the database consistency constraints, but also the timing constraints. Real-time database systems can be found, for instance, in program trading in the stock market, radar tracking systems, battle management systems, and computer integrated manufacturing systems. Some of these systems (such as program trading in the stock market) are soft real-time systems. These systems are designated *soft* because missing a deadline is not catastrophic. Usually, research into algorithms and protocols for such systems explicitly address deadlines and make a best effort at meeting deadlines. In soft real-time systems there are no guarantees that specific tasks will make their deadlines. This is in contrast to *hard* real-time systems (such as controlling a nuclear power plant) where missing some deadlines may result in catastrophic consequences. In hard real-time systems *a priori* guarantees are required for critical tasks (or transactions).

Most current real-time database work deals with soft real-time systems. In this work, the need for an integrated approach that includes time constrained protocols for concurrency control, conflict resolution, CPU and I/O scheduling, transaction restart and wakeup, deadlock resolution, buffer management, and commit processing has been identified. Many protocols based on locking, optimistic, and timestamped concurrency control have been developed and evaluated in testbed or simulation environments. In most cases the optimistic approaches seem to work best.

Most hard real-time database systems are main memory databases of small size, with predefined transactions, and hand crafted for efficient performance. The metrics for hard real-time database systems are different than for soft real-time databases. For example, in a typical database system a transaction is a sequence of operations performed on a database. Normally, consistency (serializability), atomicity and permanence are properties supported by the transaction mechanism. Transaction throughput and response time are the usual metrics. In a soft real-time database, transactions have similar properties, but, in addition, have soft real-time constraints. Metrics include response time and throughput, but also include percentage of transactions which meet their deadlines, or a weighted value function which reflects the value imparted by a transaction completing on time. On the other hand, in a hard real–time database, not all transactions have serializability, atomicity, and permanence properties. These requirements need to be supported only in certain situations. For example, hard real-time systems are characterized by their close interactions with the environment that they control. This is especially true for subsystems that receive sensory information or that control actuators. Processing involved in these subsystems are such that it is typically not possible to *rollback* a previous interaction with the environment. While the notion of consistency is relevant here (for example, the interactions of a real-time task with the environment should be consistent with each other), traditional approaches to achieving consistency, involving waits, rollbacks, and abortions are not directly applicable. Instead, compensating transactions may have to be invoked to nullify the effects of previously committed transactions. Also, another transaction property, namely *permanence*, is of limited applicability in this context. This is because real-time data, such as those arriving from sensors, have limited *lifetimes* – they become obsolete after a certain point in time. Data received from the environment by the lower levels of a real-time system undergoes a series of processing steps (e.g., filtering, integration, and correlation). We expect the traditional transaction properties to be less relevant at the lowest levels and become more relevant at higher levels in the system.

While the hard real-time system should guarantee all critical transaction deadlines and strive to meet all other transaction deadlines, this is not always possible. In this case it is necessary to meet the deadlines of the more important transactions. Hence, metrics such as maximizing the value imparted by completed transactions and maximizing the percentage of transactions that complete by their deadline are primary metrics. Throughput and response time are secondary metrics, if they are used at all.

Outstanding questions in real-time database work include:

- should all the data be integrated in one uniform database,
- should there exist several subsystems, each better tailored to the specific needs of transactions that comprise that subsystem, and if so, what are good interfaces between those subsystems,
- what is the relevance of properties traditionally associated with transactions to hard real-time databases (it is sometimes true that in some real-time systems, *an incomplete result "on time"* is considered better than a late but complete result. In the context of transactions, this implies that committing a (partially completed) transaction that is expected to be *lost* may be acceptable, depending on the nature of the transaction and on the consistency of the data modified by it.),
- what distributed operating system and communication primitives are required to support predictability in such systems, and
- what is the correct form that active databases should take when used in a real-time setting, i.e., when triggers of an active database can be initiated by timing requirements and where other triggers must complete by a deadline?

Current real-time database research has focussed on database systems with flat, soft real-time transactions operating on centralized databases with read/write objects. Extensions to real-time database research is needed along four dimensions:

- enhancing the *data model* from a simple read/write data model to an abstract data type model,

- enhancing the *transaction model* to nested transactions,
- enhancing the *system model* to include distributed database systems, and
- enhancing the *nature of timing constraints* to include hard real-time constraints.

7 Artificial Intelligence

Many complex real-time applications now require or will require knowledge-based on-line assistance operating in real-time [5, 6]. This necessitates a major change to some of the paradigms and implementations previously used by AI researchers. For example, AI systems must be made to run much faster (a necessary but not sufficient condition), allow preemption to reduce latency for responding to new stimuli, attain predictable memory management via incremental garbage collection or by explicit management of memory, include deadlines and other timing constraints in search techniques, develop anytime algorithms (algorithms where a non-optimal solution is available at any point in time), develop time driven inferencing, and develop time driven planning and scheduling. Rules and constraints may also have to be imposed on the design, models, and languages used in order to facilitate predictability, e.g., limit recursion and back-tracking to some fixed bound. Coming to grips with what predictability means in such applications is very important.

In addition to these changes within AI, real-time AI (RTAI) techniques must be interfaced with lower level real-time systems technology to produce a functioning, reliable, and carefully analyzable system. Should the higher level RTAI techniques ignore the system level, or treat it as a black box with *general* characteristics, or be developed in an integrated fashion with it so as to best build these complex systems? What is the correct interface between these two (to this point in time) separate systems. Integrating RTAI and low level real-time systems software is quite a challenge because these RTAI Applications are operating in non-deterministic environments, there is missing or noisy information, some of the control laws are heuristic at best, objectives may change dynamically, partial solutions are sometimes acceptable so that a tradeoff between the quality of the solution and the time needed to derive it can be made, the amount of processing is significant and highly data dependent, and the execution time of tasks may be difficult to determine. These sets of demanding requirements will drive real-time research for many years to come.

Not only is it important to develop real-time AI techniques, but it is also necessary to determine what must change at the low levels to provide adequate support for the higher level, more application oriented tasks? Answers to these questions have been somewhat arbitrarily chosen on an application by application basis. It is too early to synthesize the general principles and ideas from these experiments as sufficient evaluative data is not available.

Competing software architectures for real-time AI include production rule architectures, blackboard architectures, and a process trellis architecture. Some real-time AI systems have been built by carefully and severely restricting how production rules and blackboard systems are built and used. Research is on-going to relax the restrictions so that the power of these architectures can be utilized, but at the same time providing a high degree of predictability. The process trellis architecture, used in the medical domain, is a highly static approach while the other two are much more dynamic. The trellis architecture (because it is static) has potential to provide static real-time guarantees for those applications characterized by enough time to completely compute results from a set of inputs before the next set of inputs arrive. This approach is suitable for certain types of real-time AI monitoring systems, but its generality for complex real-time AI systems has not been demonstrated.

In a distributed setting, high level decision support requires organizing computations with networks of cooperative, semi-autonomous agents, each capable of sophisticated problem solving. Theories of communication and organizational structure for groups of cooperative problem solving agents must be developed. These theories must include problem solving under uncertainty and under timing constraints.

8 Summary

Most distributed, critical, real–time computing systems require that many competing requirements be met including hard and soft real–time constraints, fault tolerance, protection, security, and significant computational requirements. In this list of requirements, the real–time requirements have received the least formal attention. We believe that it is necessary to raise the real–time requirements to a central, focusing issue. This includes the need to formally state the metrics and timing requirements (which are usually dynamic and depend on many factors including the state of the system), and to subsequently be able to show that the system indeed meets the timing requirements. Achieving this goal is non-trivial and will require research breakthroughs in many aspects of system design and implementation. For example, good design rules and constraints must be used to guide real–time system developers so that subsequent implementation and *analysis* can be facilitated. This includes proper application decomposition into subsystems and allocation of those subsystems onto distributed architectures. The programming language must provide features tailored to these rules and constraints, must limit its features to enhance predictability, and must provide the ability to specify timing, fault tolerance and other information for subsequent use at run time. Execution time of each primitive of the Kernel must be bounded and predictable, and the operating system should provide explicit support for all the requirements including the real–time requirements [17]. The architecture and hardware must also adhere to the rules and constraints and be simple enough so that predictable timing information can be obtained, e.g., caching, memory refresh and wait states, pipelining, and some complex instructions all contribute to timing analysis difficulties. The resulting system must be scalable to account for the significant computing needs initially and as the system evolves. An insidious aspect of critical real–time systems, especially with respect to the real–time requirements, is that the weakest link in the entire system can undermine careful design and analysis at other levels. Research is required to address all of these issues in an integrated fashion.

References

[1] K. Arvind, K. Ramamritham, and J. Stankovic, "A Local Area Network Architecture for Communication in Distributed Real-Time Systems," *Real-Time Systems*, Vol. 3, No. 2, May 1991.

[2] C. Hou and K. Shin, "Load Sharing with Considerations of Future Task Arrivals in Heterogeneous Distributed Real-Time Systems," *Proceedings IEEE Real-Time Systems Symposium*, December 1991.

[3] H. Kopetz, A. Damm, C. Koza, and M. Mulozzani, "Distributed Fault Tolerant Real-Time Systems: The Mars Approach," *IEEE Micro*, pp. 25-40, 1989.

[4] H. Kopetz and W. Ochsenreiter, "Clock Synchronization in Distributed Real-Time Systems," *IEEE Transactions on Computers*, Vol. 36, No. 8, pp. 933-940, August 1987.

[5] V. Lesser, J. Pavlin, and E. Durfee, "Approximate Processing in Real-Time Problem Solving," *AI Magazine*, Vol. 9, pp. 46-61, 1988.

[6] C. Paul, A. Acharya, B. Black, and J. Strosnider, "Reducing Problem-Solving Variance to Improve Predictability," *CACM*, Vol. 34, No. 8, August 1991.

[7] K. Ramamritham, J. Stankovic, and P. Shiah. "Efficient Scheduling Algorithms for Real–Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Computing*, Vol. 1, No. 2, pp. 184-194, April 1990.

[8] K. Ramamritham, J. Stankovic, and W. Zhao, "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *IEEE Transactions on Computers*, Vol. 38, No. 8, pp. 1110-1123, August 1989.

[9] J. Schoeffler, "Distributed Computer Systems for Industrial Process Control," *IEEE Computer*, Vol. 17, No. 2, pp. 11-18, February 1984.

[10] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, Vol. 39, No. 9, pp. 1175-1185, 1990.

[11] K. Shin, "HARTS: A Distributed Real-Time Architecture," *IEEE Computer*, Vol. 24, No. 5, May 1991.

[12] S. Shrivastava and A. Waterworth, "Using Objects and Actions to Provide Fault Tolerance in Distributed Real-Time Applications," *Proceedings IEEE Real-Time Systems Symposium*, pp. 276-287, December 1991.

[13] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," *Real-Time Systems*, Vol. 1, pp. 27-60, 1989.

[14] J. Stankovic, "Misconceptions About Real-Time Computing: A Serious Problem for Next Generation Systems," *IEEE Computer*, Vol. 21, No. 10, October 1988.

[15] J. Stankovic and K. Ramamritham, "What is Predictability for Real-Time Systems," *Real-Time Systems Journal*, Vol. 2, pp. 247-254, December 1990.

[16] J. Stankovic, "On the Reflective Nature of the Spring Kernel," invited paper, *Proceedings Process Control Systems '91*, February 1991.

[17] J. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real–Time Systems," *IEEE Software*, Vol. 8, No. 3, pp. 62-72, May 1991.

[18] J. Stankovic, D. Niehaus, and K. Ramamritham, "SpringNet: A Scalable Architecture for High Performance, Predictable, and Distributed Real-Time Computing," submitted for publication, also Univ. of Massachusetts, TR 91-74, October 1991.

[19] M. Takegaki, H. Kanamaru, and M. Fujita, "The Diffusion Model Based Remapping for Distributed Real-Time System," *Proceedings IEEE Real-Time Systems Symposium*, December 1991.

[20] H. Tokuda and C. Mercer, "ARTS: A Distributed Real-Time Kernel," *ACM OS Review*, Vol. 23, No. 3, July 1989.

[21] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time MACH: Towards a Predictable Real-Time System," *Proc USENIX MACH Workshop*, October 1990.

[22] M. Wright, M. Green, G. Fiegl, and P. Cross, "An Expert System for Real-Time Control," *IEEE Software*, Vol. 3, No. 2, pp. 16-24, March 1986.

[23] W. Zhao, K. Ramamritham, and J. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real–Time Systems," *IEEE Transactions on Software Engineering*, Vol. 12, No. 5, pp. 567-577, May 1987.