MARUTI: A Hard Real-Time Operating System

Olafur Gudmundsson, Daniel Mossé, Ashok K. Agrawala, and Satish K. Tripathi
Systems Design and Analysis Group
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

Abstract

The MARUTI operating system is designed to support hard real-time applications on distributed computer systems while providing a fault tolerant operation. It is an object oriented design and the communication mechanism allows transparent use of the resources of a distributed system. Fault tolerance is provided through a consistent set of mechanisms that support a number of policies. Most importantly, MARUTI supports guaranteed-service scheduling, in which jobs that are accepted by the system are guaranteed to meet the time constraints of the computation requests with a specified degree of fault tolerance. As a consequence, MARUTI applications can be executed in a predictable fashion.

The development of current hard real-time applications requires that the analyst estimate the resource requirements for all parts of the computation, and then make sure that the resources are available to meet the time constraints. It tends to be a cumbersome process. As a part of the MARUTI system, a set of tools has been developed which supports the hard real-time applications during various phases of their life cycle. The present version of MARUTI has been implemented as a prototype running on a UNIX platform. Experiences with the development of this prototype are also presented.

1 MARUTI Overview

The main focus of the MARUTI project at the University of Maryland is to examine the constructs of future distributed, hard real-time, fault tolerant, secure operating systems[4, 8, 9]. This operating system contains a number of new approaches for solving many of the problems in real-time systems of tomorrow. The development is a research effort aimed at exploring the use of time-driven system design for next generation real-time systems [1, 13].

1.1 Structures

MARUTI is an object-oriented system whose basic building block is an object. While the concept of object[2, 3, 6] and its encapsulation has been used in many systems, in order to incorporate these concepts in a hard real-time system several extensions had to be made. Objects in MARUTI consist of two main parts: a control part (or joint), and a set of service access points (SAPs), which are entry points for the services offered by an object. A joint is an auxiliary data structure associated with every object. Each joint maintains information about the object (e.g., computation time, protection and security information) and its requirements (service and resource requirements). Timing information, also maintained in the joint, is dynamic and includes temporal relations among objects. A calendar, a data structure ordered by time, contains the name of the services that will be executed and the timing information for each execution.

An application is depicted by a collection of objects gathered in a computation graph, a rooted directed acyclic graph. The vertices represent the services, and the arcs depict timing or data precedence between two vertices[7]. In MARUTI each application is described in terms of a computation graph. An application has timing and fault tolerance requirements associated with each service. This information is kept in the joint.

Objects communicate with one another via semantic links. These links are called semantic because they perform range and type checking of the information. Objects that reside in different sites need agents as representatives on remote sites. The agents are responsible not only for the remote transmission of messages but, to support heterogeneity, they are also responsible for the data translation of these messages.
There are two types of jobs in MARUTI, namely real-time and non-real-time. A real-time job has a hard deadline and a requested earliest start time. For non-real-time jobs, no time constraints are specified and, therefore, jobs are executed on the basis of time and resource availability. The current version of MARUTI does not have priorities for real-time jobs, and all accepted jobs are treated equally. Priorities can be incorporated, for example, by implementing a scheme for the revocation of some accepted real-time jobs.

MARUTI views the system resources as organized in various partitions. A resource belongs to one, and only one, partition. This concept is used in the implementation of the fault tolerance scheme using replication. In addition, such division of resources is useful for the distribution, load balancing, and fault isolation.

1.2 Approach and Principles

MARUTI is organized in three distinct levels, namely the kernel, the supervisor, and the application level. The kernel is a collection of core-resident server objects. The kernel is the minimum set of servers needed at execution time and is comprised of resource manipulators. The functions of the kernel are dispatching, loading, time manipulation and communication. The main task of the supervisor level objects is to prepare the jobs for execution by making reservations for the services. The services provided at the supervisor level include allocation, schedule verification, binding, login service, and name service.

The resources needed for the execution of the applications are reserved through the resource manipulators at the supervisor level, prior to the start-time of the application. The communication channels, CPU, memory, disks, and all other necessary resources are reserved so that run-time contention is eliminated, and timing guarantees can be satisfied by the system.

The concept of object and the use of the joints allow each access to an object to be direct. Access to an executing object is an invocation of a particular service of that object. Several applications are allowed to invoke a particular service concurrently, with full access and independent timing control.

The use of joints, and specifically of calendars, allows verification of schedulability, reservation of guaranteed services, and synchronization at allocation time. Projection mechanisms support propagation and adjustment of time constraints between different localities and nodes of the CG. These projections maintain the required event ordering and assure the satisfaction of the timing constraints. Protection mechanisms are activated and authorizations are established prior to run-time to allow direct access afterwards.

Jobs in MARUTI are invocations of services in executable objects, where each invocation is a scheduled activity with its own set of resources. The requirements of a reactive system justify supporting real-time and non-real-time execution disciplines. We note that the non-real-time jobs are assumed to be preemptable so that their processing requirements can be satisfied in the time slots available between executions of real-time jobs.

1.3 Fault Tolerance and Security

Fault tolerance is an integral part of the design of MARUTI. The joint of each object may implement the fault detection, monitoring, local recovery and reporting. For redundancy management, each joint contains a consistency control mechanism to manage replicas. The resource allocation algorithm supports a user-defined level of fault tolerance, where redundancy can be temporal (execute again) or physical (parallel execution).

Physical redundancy supports node and link failures uniformly, using roll-forward recovery techniques. Primitives are provided to the user to specify the desired level of fault tolerance. The fault tolerance may be achieved by replication or retries, alternate services or different modules, voting mechanisms or majority queries, backward and forward recoveries.

A capability based approach is used for protection and security. Protection and security requirements of a system are completely specified prior to execution of the jobs. The access capabilities have to be provided by the user, and are checked during allocation. Our model of semantic links lends itself well handle encryption and decryption of messages. We are investigating the level of security that can be

1Reactive systems are those that accept new jobs while executing already-accepted guaranteed jobs.
achieved using this approach.

1.4 Applications

To support the application development, MARUTI provides several tools that can be used to assist in functions such as design, verification, testing, and maintenance of applications. These tools have been designed to meet unique information needs of hard real-time applications.

MARUTI applications are written as object oriented programs, primarily using existing programming languages. These languages have been extended to include the primitives and constructs necessary for MARUTI applications. A precompiler has been provided to convert the MARUTI program into standard programming language that automatically generates joints.

While MARUTI supports many different fault tolerance mechanisms, applications can be written without the knowledge of the actual mechanism to be used. The user has the option of augmenting the fault tolerance with custom-designed algorithms, or leaving the fault management to the system's default mechanisms.

2 MARUTI Components

System objects in MARUTI have the same structure as user objects. The application level contains the user applications and the tools to support user program development. The system objects primarily function at the kernel and supervisor levels. Let us consider in some detail these objects.

2.1 Kernel Components

The MARUTI kernel consists of a set of objects that must be core-resident and that provide the essential run-time support. These object include the following:

1. Dispatcher. The dispatcher is invoked upon the completion of a service or at the start of another service. The next executable service is selected based on information from the calendar. If there is no real-time job ready to be run, non-real-time jobs are executed.

2. Loader. The loader's task is to load objects into memory. It also converts the object addresses to absolute memory addresses. The loader is activated at the latest possible time but before the execution initiation.

3. Time Server. The time server provides the knowledge of time (past, present, and future) to executing objects. It also aids the dispatcher in choosing jobs, by providing alarms and watchdog services.

4. Communication Server. The communication media are considered resources of the system, and as such, a reservation calendar is established for each physical link. A reservation is made at both ends of the communication link, and a service is reserved according to the requirements of the agents handling that link. The communication server is responsible for sending and receiving the messages.

5. Resource Manipulators. The loader, the dispatcher and the communication server are the memory manipulator, the CPU manipulator, and the media manipulator, respectively. Extending this concept, other resource manipulators may also be necessary for other resources.

2.2 Supervisor Level Objects

The supervisor objects prepare the computation to take place, making timely execution possible through the pre-allocation of resources, so that, at run time, dispatching of already reserved system services is the only action performed. Following is a list of MARUTI supervisor level objects and their functionality.

1. Allocator. The allocator extracts the resource requirements for a processing request from the joints of the objects of the computation graphs for the request. The allocator then proceeds to allocate and verify the allocation of the resources. The first step in this process is to recognize the fault tolerance requirements, and the number of partitions to be used in allocation.

A local allocator at each computer node is responsible for managing all the local resources. The allocator accepting the processing request may have to interact with allocators at several other nodes to carry out the allocation of resources consistent with the processing and fault handling requirements of the request.
2. Verifiers. Verification of each resource usage and reservation is a crucial part of the system. All requests to a particular resource have to be satisfied (within the time frame specified by the user) for computation to succeed. The verifier assures that the resource allocation is consistent with all the constraints.

3. Binder. The binder is responsible for "connecting" communicating objects, as well as for verifying that the semantic relation is properly established. The resulting addresses are still not absolute, and only the loader changes them. Remote binding is done through agents, and links are established from an object to a local agent. Agents are connected through the underlying network.

4. Login Server. The login server is the user interface to MARUTI, and its major task is command interpretation. It creates a working environment for the user. Furthermore, the security checks are initiated at the user log-in time.

5. Name server. The name server bridges the different name spaces, from the human oriented names (strings) to machine oriented names (addresses). In addition, the name server keeps track of status of machines and keeps information about locations of objects.

3 Experimenting with a Prototype

The focus of our research effort has been to establish the feasibility of the design concepts in MARUTI. In order to develop a working implementation for this purpose, the most expeditious route was to implement it as a prototype on a suitable platform.

In building a prototype, a major decision is the choice of the platform that will serve as the host. For portability and due to the availability of excellent program development environments, UNIX was the platform used in the first prototype version. However, in the prototyping process we recognized that the UNIX system is not a very hospitable host for a distributed, reactive, hard real-time operating system. We faced many problems and produced solutions as needed. In this section we describe the implementation experience so far, and what we have learned from it.

An advantage of prototyping MARUTI has been to have a running system in a very short time, compared with the time it would have taken to implement the system on a bare machine. As a consequence, we could enhance our understanding of the design through feedback. Since the underlying operating system provided the necessary low level services, our prototyping effort could concentrate on the unique features of MARUTI. The isolation and encapsulation provided by the objects architecture of MARUTI furnishes a good design for prototyping.

3.1 UNIX: Problems and Solutions

Since UNIX is not designed to be a real-time operating system, several problems were encountered in using it as the underlying system. Therefore, we had to develop solutions to accommodate the usage of UNIX. In this section we present a description of the most common issues encountered when implementing the prototype, along with our solutions.

The UNIX scheduler tries to reevaluate its run-queue every clock-cycle. The real-time scheduler is quite different from the UNIX scheduler. Our solution is simple and incurs little overhead. In the UNIX run-queue we keep only the presently running task (decided according to the real-time schedule). By maintaining a single task in the run-time queue, the rescheduling of undesired tasks is avoided. Clearly, for this solution to work the machine must not operate in the usual multi-user mode and should not have any other users in the system.

In addition to scheduling and reshuffling of tasks, in UNIX there exists a problem with the communication among objects. Upon start-up, an object connects to the communication server via a predefined socket. Then it informs this server about its name and process identification number. Each open socket requires a file descriptor. Since UNIX limits the number of file descriptors available, this is an inherent limitation that restricts the number of objects in the system at any given time. Furthermore, system objects (9 altogether) also require file descriptors, further restricting the number of active user objects which may be a part of an application.

Another feature of UNIX (for non-real-time systems) is automatic swapping that occurs under certain conditions. Swapping incurs a high and unpredictable overhead, which,
in turn, may cause tasks to miss the deadlines. As an example, SUN-3 Memory Management swaps out any process that has been blocked for 30 seconds or longer. This automatic swapping must be avoided since we assume memory resident processes during execution. Due to these factors, MARUTI loads new objects with enough time to perform start-up actions, minimizing the time it would be memory resident while in a blocked state. The actual loading time is decided in accordance with the user's specification of job start time and size of objects to be loaded.

In UNIX, several demons run in the background (e.g., rwhod, sendmail, and nsfd) and are invoked on demand or periodically. Most of these daemons perform small tasks, but some are big in size and spawn processes (for example, the sendmail demon). The presence of these demons causes irregularities in timing of the execution of MARUTI objects. Therefore, to be on the conservative side, we allow extra time for execution of each object.

Another major problem in using the UNIX system as the base for prototyping MARUTI was the way clock is handled in these systems. The clock accuracy is rather poor with the clock tick time of 10ms or 20ms. Most actions in UNIX system are tied to clock ticks, yielding a relatively poor time management capability for MARUTI, which sits on top of UNIX. Furthermore, distributed clock synchronization algorithms are needed in order to coordinate actions among the distributed resources of the systems.

Clearly this prototype is inadequate for us to determine the speed with which MARUTI is capable of responding. On the other hand, the functionality of the system has been well tested, since we can scale the time constraints and executions to the desired granularity. In this regard we achieved the major objective of our effort.

4 Concluding Remarks

The MARUTI project is examining new ways of building real-time systems which provide a comprehensive set of services in support of the requirements of mission critical systems including hard real-time, distributed operation, and fault tolerance. We started out with a new design, and one of the goals of the project was to investigate new ways of implementing this system. The design incorporates several primitive mechanisms at each stage and permits easy adoption of policies. One of the results of this work has been a novel formulation of a theory of time-driven designs [1], in contrast to interrupt driven design of traditional real-time systems. We believe that the time driven design is simpler, more easily verifiable, and more suitable for hard real-time operation. Our experience so far has shown that the complexity of the applications is substantially reduced by this approach.

MARUTI has been designed as a platform for the study of distributed, fault-tolerant, hard real-time systems. Its characteristics facilitate the development and testing of many different classes of applications. We organized this project in a way to allow us the greatest flexibility in using MARUTI as a research tool. Our first implementation is a prototype on top of UNIX, running in a distributed environment. This has allowed us to port MARUTI to a number of platforms and upgrade to faster and better machines as they become available. The current prototype was designed mainly for functional verification and ease of modification[12].

A major advantage of building the prototype of MARUTI on top of UNIX has been the availability of all the program development tools. However, UNIX is not a good platform for hard real time systems in which we want to achieve a high degree of time determinism. We can not enforce timing constraints any better than the time resolution provided by the operating system of the platform. Our efforts provide a feasibility validation for the approach taken in MARUTI. The prototype is written in a portable version of UNIX that allows us to run it on different hardware platforms. The current version has been tested in a heterogeneous hardware environment.

It is essential that a real time system exercise a timely control over all the resources. The UNIX system, however, has its own resource management policies. We had to devise techniques for controlling the UNIX resource managers in accordance with MARUTI behavior. Clearly this is not an efficient way of managing resources. We are in the process of developing a kernel version of MARUTI, in which the kernel objects will manipulate the system resources directly.

---

2A mission critical system is any system where fault in certain parts can have serious implications
References


