The *MARUTI* System and its Implementation*

Daniel Mossé, Ólafur Gudmundsson, Ashok K. Agrawala
Systems Design and Analysis Group
Department of Computer Science
University of Maryland
College Park, MD 20742
{mosse, ogud, agrawala}@cs.umd.edu

Abstract

Design and development of integrated systems which support real time, fault tolerant, heterogeneous, and secure operation pose new and interesting challenges for system designers and developers. The *MARUTI* system efforts are aimed at addressing the needs of such systems. In this paper, basic principles, structure, and design of the *MARUTI* system are presented. We also present the system prototype, which has been implemented on a UNIX platform. The approach used in mapping the *MARUTI* system objects into UNIX processes is presented along with a brief discussion of our implementation experiences.

*This work is supported in part by contract DASG60-87-C-0066 from the U. S. Army Strategic Defense Command to the Department of Computer Science, University of Maryland. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of Defense position, policy, or decision, unless so designated by other official documentation.
Computer facilities were provided in part by NSF grant CCR-8811954.

1 Introduction

Many large, complex systems consist of numerous components which must interact in a controlled manner. Such systems are controlled by an embedded computer system which plays a crucial role in the successful deployment and operation of the system. In order to meet the control requirements of the system, the embedded computer system must provide support for reliable and secure operation of the software which runs on it. Further, the software must operate within real-time constraints to monitor and generate control signals. While many examples of embedded systems are in use today, most of them address a subset of the requirements. For example, current embedded systems in appliances provide control through static real-time operation.

When an integrated system has to meet real-time requirements and provide support for fault-tolerant, secure, distributed, heterogeneous operations many of the techniques developed for addressing these requirements in isolation cannot be used directly. Integrated systems must be designed by considering the comprehensive requirements during the design phase. In *MARUTI* system we are attempting to address these requirements in a comprehensive manner and demonstrate the feasibility of the approaches developed through the implementation of a prototype of the system.

The design of the *MARUTI* system has followed a few basic principles which have formed the foundation on which the system is designed and built. In this paper we present these principles and give a rationale for them. An overview of the system is presented to show how these principles have been reduced to practice in this system. The prototype of the system has been implemented on UNIX platform to permit a quick implementation and extensive portability. We discuss the approach taken in using UNIX primitives for the *MARUTI* system, and what we have learned in the process. Our experiences in the design and development of the *MARUTI* system show that it is quite feasible to address the requirements for the embedded systems of tomorrow in a comprehensive manner. Clearly, our work to date can only be considered a starting point and a proof of concept.

1.1 Related Work

In order to put our work in perspective, in this subsection we discuss a number of other efforts which are aimed at meeting the requirements of advanced hard real time operating system presented in Section 1. This discussion is not meant to be exhaustive but illustrative of the kinds of systems that have been developed to date. The discussion is organized along the lines of major requirements and how they are addressed by Amoeba[26], ARTS[28, 27], Alpha[8], Clouds[6], Cronus[21, 22], HCS[18], ISIS[3], Mach[1], Spring[24, 25] and V[4] systems. Research in ARTS and Spring is directed towards predictable service in hard real-time environment whereas the research goals of other projects are varied. Alpha is directed towards research in fault tolerant, dependable kernel support with a common language interface while providing support for soft real time applications. It controls priority inversion problems and avoids the occurrence of deadlocks. Amoeba and V are concerned with a high performance distributed operating system using a minimal kernel

approach. Clouds is directed towards the use of object-oriented approach at the operating system level. Mach is concerned with a unified approach towards supporting distributed and concurrent systems while retaining UNIX compatibility.

Hard Real-time Several of the systems discussed here provide support for real-time operation, but rarely hard real-time operation. Commercial *real-time operating systems*, such as Real Time Unix (RTU) and LynxOS, address the real-time requirements by providing an interrupt-driven kernel, fast context switching and a priority- driven scheduler.

The Spring kernel uses explicit time constraints to schedule the critical (hard real-time) tasks, which are guaranteed to execute within their deadlines. ARTS uses a priority inheritance technique to ensure that high priority tasks are executed ahead of low priority tasks. In this system variants of rate-monotonic scheduling are used to provide support for hard real-time operation. The only support for real-time operations in the V System is through the implementation of a small, efficient kernel and fast interprocess communication through VMTP.

Fault tolerance Support of fault tolerance under real-time constraints has not been explored by many projects. ARTS provides support for exception handling as well as timing errors whereas Alpha provides data replication as well as migration for fault tolerance. However they do not address the impact of these techniques on timing characteristics of applications. In the non real-time systems, such as HCS and ISIS, checkpointing and object replication has been studied in detail. Cronus also allows data replication and migration in case of faults.

Distributed and Heterogeneous Operations Support for distributed operation has been the focus of general purpose operating systems, such as MACH, Cronus and Amoeba. These systems do not provide any support for real-time operations, however.

Several of the real-time systems, such as ARTS, Alpha and Spring, provide support for distributed operations. Spring uses homogeneous multiprocessor hardware with shared memory, while ARTS operates on top of MACH in a distributed homogeneous environment. These systems do not provide support for heterogeneous hardware.

The main focus of HCS and Cronus has been to provide support for heterogeneous environments. HCS uses the techniques of stub generation and late binding. These systems have demonstrated the feasibility of distributed operation in a heterogeneous environment.

Security All of the real-time systems we are discussing provide minimal or no support for secure operations. Systems such as Amoeba, Alpha and Cronus provide minimal, capability-based support for access control. There are a few research efforts under way which are aimed at developing multi-level, secure operation systems. Trusted MACH is a good example of such efforts. The focus of T-MACH is on security aspects only. A secure version of Cronus, Theta, is also being developed.

Application Development Support It is recognized that the development life cycle requires more support for real-time, distributed applications. This support has to be system-specific. Many of the systems

discussed here provide such support to different degrees. The best program development support is available for general purpose systems such as MACH and Cronus. The ARTS system provides ARTC++, a C++[12] based language, a schedulability analyzer called Scheduler 1-2-3 and a real-time monitor/debugger. HARTOS provides a tool to create synthetic workload while MARS provides a maximum execution time analyzer.

Portability Many of the real-time systems we are discussing, such as Spring, have been implemented to run on a specific hardware platform. Implementing these systems on a different hardware requires considerable effort. Many of the techniques used in these systems rely on hardware-specific features, and would have to be redesigned in order to port them to a different platform. The ARTS system by being implemented on top of MACH addresses the portability issues by taking advantage of MACH's portability.

Clearly most of these systems address only a subset of the requirements for advanced hard real-time operating systems presented in Section 1. The goal of the *MARUTI* project is to develop a comprehensive framework for addressing all these requirements. This will advance the state of the art of building hard real-time systems, by developing proven techniques which can be incorporated into other operating systems. The focus of our efforts is in meeting the hard real time and fault tolerance requirements in a heterogeneous, distributed computing environment. Our design will permit us to incorporate techniques for some requirements, such as security, as they are developed by other researchers.

2 *MARUTI* Overview

The main focus of the *MARUTI* project is to study the problems associated with the design, development, and deployment of real-time applications which meet the dependability and security requirements and can be implemented on a heterogeneous, multi-machine platform[2, 23]. The approach taken in this project[20, 11, 10] is to first construct the framework of the operating system, and then conduct studies and develop techniques for issues such as scheduling, resource management, naming, communication, fault detection, fault recovery, security, etc. These techniques are then implemented in a prototype system where they can be tested and evaluated thoroughly.

2.1 Principles

The *MARUTI* system is an object oriented system whose basic building block is an *object* which encapsulates a set of *services*. A service may invoke a service of another object by sending a message to the *service access point (SAP)* for that service. The same structure of objects is used for the operating system as well as the user applications. All hardware resources in the system are encapsulated in objects and access to them is made through SAPs of that object.

In the *MARUTI* system an application or a job is viewed as a rooted, directed, acyclic graph referred to as the *computation graph* of the application. In this graph, a vertex represents a service and an edge depicts the invocation message (possibly containing data) sent from one service to another[9].

An important requirement of a system capable of meeting hard real-time constraints is that all system operations require a bounded amount of time for their execution. In order to support this characteristic in the *MARUTI* system, the concept of objects has been extended to include a control part (or *joint*), in addition to the encapsulated set of services. The joint of an object is an auxiliary data structure which maintains certain static and dynamic information about the object.

The design and implementation of the *MARUTI* system has been guided by a few basic principles.

Principle 1 *Time is considered an essential attribute of every entity in the* MARUTI *system.*

The system keeps track of the time of all known events with respect to every *service* in the system. This information is kept in a data structure called *calendar* which is a part of the joint. Therefore, at any time instant, the calendars in the system contain information about all invocations and executions of services known to the system at that time.

The major consequence of this principle is that the time is treated explicitly and uniformly by all system components.

Principle 2 *The basic model of invocation among services is one way invocation.*

The one way invocation model facilitates precise management of resources used by the services and the time. The client server model (e.g. RPC) requires the blocking of the client while the server is active, making the execution time of the client a function of the server execution time. Therefore scheduling a client becomes difficult due to this dependability. One way invocation model avoids this problem and, thus, is well suited for real-time systems.

Principle 3 *Do as much work of preparing an application for execution as possible.*

Real time operating systems should perform, during run time, only those tasks that are strictly necessary. The consequence of following this principle is that overhead during execution is reduced. By incorporating binding and timing information into objects, early during application development, more detailed analysis can be carried out. To facilitate this we have designed a *MARUTI* development model which consists of the following phases.

- **Design**. This phase is the starting point of the development of an application during which the overall design is carried out. The activities during this phase include requirement specification, conceptual design and detailed design. As an output the application design is organized as a DAG with the functionality of each service and information exchange requirements identified. Timing requirements available are included in the output.

- **Compilation**. Individual objects with their services are created. The static resource requirements for each service and object may be assessed at this stage. Where the information about the dynamic resources requirements is not known, placeholders are kept in the joint for this information.

- **Integration**. A computation graph for the object is created by identifying the objects and services needed, and connecting the services with the arcs. The messages to be exchanged by the services are verified syntactically and operationally. The remaining resource requirements for the job are identified and recorded with the application.

- **Resource allocation**. After a processing request is submitted, the resource allocation phase is carried out, in which the resources for the processing of the job are allocated. In this phase reservations are made in the appropriate calendars.

- **Execution**. During this phase the operating system performs dispatching, message passing and reservation enforcement. Previous phases prepare the application for this phase.

A user may only request the real-time processing of a job whose computation graph is prepared before the start time of the job. The user may specify the time constraints for this execution by identifying the *ready time* and *deadline*. It is expected that the ready time, i.e., the earliest start time, is later than the time at which the request is made. If a user wants a job to be executed periodically, additional information about the period and number of times to be executed has to be specified.

The effect of following this principle for the *MARUTI* system is that at every stage in the development and deployment of an application all its components contain the placeholders for all the resource and timing information. The values to these placeholders are assigned as early during the development as they can be assessed, refining them as the application development process proceeds.

Our efforts in identifying the five phases do not imply them to be a strict sequence, one pass operation. Multiple iterations through them may be essential for developing an application successfully.

Principle 4 MARUTI *reserves all resources/services before execution time for all jobs.*

Before accepting a real-time job for execution, the system initiates the resource allocation phase for that job. If all resources necessary for the execution of the job are available within the time constraints requested, the job is accepted for real-time execution. Otherwise the job request is denied. Clearly, the criteria used for accepting a job is different for hard real-time and soft real-time jobs. The user has the option of requesting that a job be executed as a non real-time job in which case the job is accepted for processing on time and resource availability.

Some real-time applications require that each of their services be executed in a non-preemptive manner, while others can be preempted. The design of the *MARUTI* system supports both non-preemptive as well as preemptive execution of real-time applications. The default for hard real-time applications is non-preemptive operation.

Many general purpose applications are designed to execute in a virtual memory environment. The process of translating the virtual address to the real address usually causes unpredictable delays which are not acceptable in a real-time environment. Therefore, in the *MARUTI* system all address translations for

the applications are done prior to the execution time so that during the execution only real addresses are issued.

Clearly adherence to this principle is essential to guarantee the successful execution of hard real-time applications for which a guarantee can only be given if all resources needed can be made available in a timely manner. An interesting consequence of using this principle in the *MARUTI* system is that several types of synchronization requirements of an application can be resolved during the resource allocation phase, requiring no overhead during runtime.

In order to carry out the resource allocation prior to accepting a job it is necessary that the resource requirements for the job be known. This information is kept in the joints also and is prepared as a part of the compilation and integration phases.

To support the execution of applications in heterogeneous, distributed environment *MARUTI* uses the concept of *semantic links*. These are links that are used by services for sending messages to each other. The semantic links also carry out semantic checks of the information and initiate appropriate actions if an error is detected. These links are also responsible for providing the buffering for the messages and for delivering messages at the right time. Any data conversion and encryption/decryption required to support the heterogeneous/distributed environment is also carried out by the semantic links.

Principle 5 *Only data moves during execution phase, not objects.*

Prior to execution, an object is loaded from secondary storage to main memory, where it resides while executing. In heterogeneous environments, different architectures have different object files, and the appropriate semantic links are created for communication.

A user of *MARUTI* may specify the fault tolerance requirements for the application in terms of the *degree of resiliency* required for the whole application or for parts of it. Thereafter the system provides the fault tolerant operation of the application transparently to the user through the mechanisms implemented uniformly in the system. These mechanisms address a variety of types of faults. The main technique used in the *MARUTI* system is to construct a *resilient computation graph*[15] for the application which is consistent with the fault tolerance requirements specified. The resilient computation graph is generated after the computation graph has been generated during the integration phase but prior to the resource allocation phase .

Clearly, the *MARUTI* system applications have to be developed with extensive analysis of resource requirements, along with the usual functional requirements. In order to support the application development, the *MARUTI* system provides a set of tools which have been designed to provide the necessary analysis support to the application development process. These tools have been designed so that they can be used in a stand-alone manner for the analysis of applications which may not necessarily run on the *MARUTI* system.

2.2 Organization of the *MARUTI* system

All objects in the *MARUTI* system are considered to belong to one of three levels, the *kernel*, the *supervisor*, or the *user* level. The kernel is a collection of core resident server objects which are needed at execution time for actual resource manipulations. The supervisor level objects are used primarily during the resource allocation phase. The objects which are not at kernel or supervisor lever are treated as user level objects.

2.2.1 Kernel Level Objects

The *MARUTI* kernel consists of a set of objects that must be core-resident and that provide the essential run-time support. The primary purpose of these objects is to provide the resource access. These objects include the following:

1. *Dispatcher.* The dispatcher manages the CPU(s). There is one such object at each processing node. The dispatcher is invoked upon the completion or at the start of a service. If there is no real-time job ready to be run, non real-time jobs are executed.

2. *Loader.* The loader manages the main memory at a processing node. The loader's primary task is to load objects into memory, converting the object addresses to absolute memory addresses. The loader is scheduled at an appropriate time 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 server is responsible for sending and receiving the messages. Links are manipulated by *agents*, which are special objects responsible for providing support for remote communication. Links are treated as communication resources and therefore also reserved for the communication of two services during a specified time period.

5. *Resource Manipulators.* A separate resource manipulator object must be present for each unique resource. Such resources include all sensors and actuators in the system along with all other devices.

2.2.2 Supervisor Level Objects

The supervisor level objects carry out the functions of the resource allocation phase. During this phase the processing request is received along with the timing constraints for this execution. This request is analyzed for the availability of the resources and if the resources are available, a reservation is made for them and the request is accepted. Otherwise the user is notified about the rejection of the processing request. Following is a list of *MARUTI* supervisor level objects and their functionality.

1. *Allocator.* This object coordinates the resource allocation phase for a given computation graph. The allocator extracts the resource requirements from the joints of the objects and carries out the allocation of resources. Remote allocators interact to apply a specific allocation policy.

2. *Verifiers.* Verification usage and reservation of each resource is a crucial part of the system. Verifiers assure that all requests to a particular resource are satisfied within the time frame specified by the user.

3. *Binder.* The binder is responsible for "connecting" communicating objects, as well as for verifying that the semantic relation is properly established. Remote binding is done through agents.

4. *Name server.* The name server maps 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.

5. *Login Server.* The login server is the user interface to the *MARUTI* system, and its major task is *command interpretation.* It creates a working environment for the user.

Note that the supervisor level objects are needed to carry out the resource allocation function. In a stable system the resource allocation function can be carried out prior to execution. These objects may not be in the system during its operation, or residing on a remote node.

2.3 Operational Description

In order to consider the actions that take place when a job is submitted to the *MARUTI* system, let us assume that the system is executing a set of accepted jobs. All the kernel objects are core resident and the executing application objects are loaded. The calendars for these objects contain the timing information for their executions.

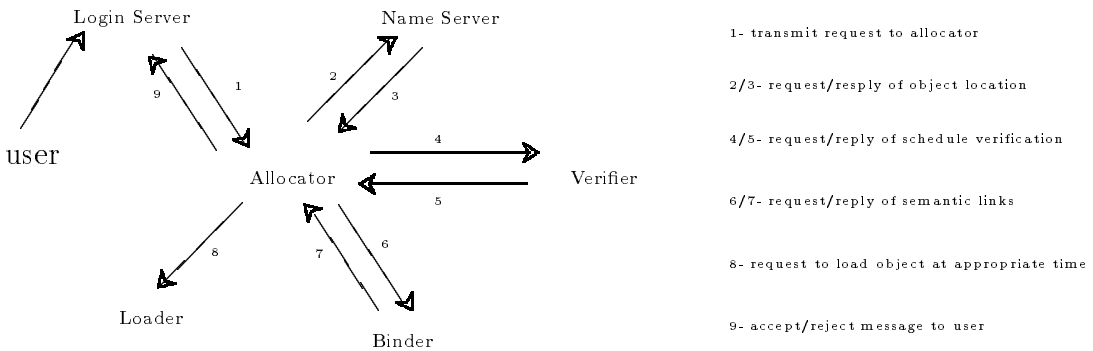

Figure 1: Tagged job through allocation phase

A user interacts with the login server and makes a processing request for an application by specifying its name and the time constraints for this execution. After authenticating the validity of the request, the login server invokes the allocator to initiate the resource allocation phase. The allocator first extracts the resource requirements from the joints of each object, after obtaining their addresses from the name server (See Figure 1).

With this information, the allocator initiates the resource allocation phase for the computation graph. This may require that it distribute the services among the nodes of the distributed system. Thereafter, the allocator requests a schedulability verification and resource reservation from the resource verifiers to assure that all local scheduling constraints are satisfied. This step is followed by the invocation of the binder, which establishes the semantic links required for the computation graph. The binder also makes reservations for the use of the communication resources. Next, a request is sent to the loader to bring the necessary objects into the memory before the execution can begin.

The resource allocation phase ends when all the steps noted above have been completed. A failure of any of these steps may result in the user being notified that the request cannot be accepted. As extensive analysis of resource requirements is carried out during the integration phase, a rejection during the resource allocation phase will usually imply a processing request inconsistent with the system design (e.g., improper time constraints) or failed components. The user may re-submit a rejected request, after modifying its constraints.

During the execution phase, the kernel objects assign resources to executing objects, according to the calendars. Due to prior reservations in time, no resource conflicts occur. When resources are not being used by real-time jobs, non real-time jobs may use them. The system assumes that non real-time jobs are preemptable and can use the resources as and when they become available.

Timely execution of applications can be guaranteed due to the allocation and verification of resources required prior to run time, if resource requirements are properly specified. This is an essential requirement for any hard real-time application. An implication however, is that rather conservative resource estimates may have to be used to get this assurance. When execution of service takes less time than initially estimated and reserved, the available time may be used for processing of other services. However, if the execution does not complete in time, the system recognizes that as a fault condition and may preempt the resources so that the timely execution of other accepted jobs is not affected.

3 Implementing the Prototype of *MARUTI*

In order to verify the concepts of the *MARUTI* system and the validity of the principles used in its design, we chose to implement it as a prototype on a UNIX platform. This decision was taken during the early days of the project (1987-88) due to the rapid growth of the computer technology and our desire of avoiding hardware dependence. A significant advantage of using the UNIX platform has been the availability of mature program development environment and tools. The availability of low level primitives such as device drivers has permitted us to concentrate on the *MARUTI* structure reducing the development time and effort required.

Clearly the choice of using the UNIX platform as a base for the development of the prototype has its limitations. The most significant among these arises from the fact that UNIX policies for managing the system resources are aimed at providing support for general purpose computing applications. Without modifying the kernel extensively these policies can not be changed. We chose to develop techniques to force

UNIX to manage system resources as dictated by *MARUTI* for its applications. Clearly this resulted in slower operation of the system.

It is well known that standard implementations of UNIX do a rather poor job of managing time[19]. Clearly this aspect of UNIX makes it unsuitable as a base for a hard real time system. However, by increasing the time granularity of the application we have still established the proof of concepts used in the *MARUTI* system. While the prototype implementation can not be used to determine the finest time granularity the *MARUTI* system can provide, it clearly established the usefulness of the approach used in this system.

UNIX is available on a large variety of hardware platforms. We have implemented the *MARUTI* system writing our code in a way that the system could be ported easily from one UNIX platform to another. To date we have ported it to five different platforms without making any changes to the UNIX systems of any of the platforms and with minimum adaptations required for *MARUTI* code.

The development tools and environment of UNIX are useful not only to system implementors but also to application writers, who had at their disposal all the UNIX facilities. Examples include window systems, compilers, debuggers, makefiles, etc. We had to enhance some facilities and build new ones to be able to support real-time environments. An example is the constructs added to the programming language to handle the time constraints and task synchronizations.

3.1 Mapping Services into UNIX and Objects

One characteristic of *MARUTI* objects is that they provide multiple services through several Service Access Points(SAP). Services within one object may share data. The object oriented design lends itself well to mapping of services into UNIX processes and the most straightforward implementation has each service as a separate UNIX process. However, this idea is unattractive because some versions of UNIX do not provide shared memory. Secondly, every UNIX process loads into its runtime image system libraries, increasing the size of every runtime image. Many systems have limited real memory and any increase in runtime image size increases the number of page faults. Thirdly, most UNIX systems have a hard coded limit on the number of processes that can be running at any given moment, and the number of file descriptors that can be open from each process.

Therefore we chose to implement each object as a process. This approach solves the problem of sharing memory among services, reduces the memory used and decreases the number of processes.

In *MARUTI* , services communicate with each other as specified in the computation graph. In implementing the communication we had one option of using a system object acting as store-and-forward entity for all communication among objects. The other option was to use direct communication among processes using mechanisms such as sockets for each semantic link. We have chosen the first option of having a communication server which does become a critical element of the system. It also requires two extra context switches for every message. As we use sockets, each message gets copied three additional times compared to direct connection. This approach is being improved using shared memory.

UNIX operating system manages the execution of processes assuming a general purpose, time sharing environment. In order to control the execution of *MARUTI* , we have to be able to influence the UNIX scheduling decisions. This could be achieved by using STOP and CONT signals. In order to avoid conflicting with debuggers, we chose not to use these signals. Instead we block the execution of objects (processes) on reading from a socket. The execution can be started by the communication server by placing data on the socket.

At startup, *MARUTI* objects execute their initialization code, in which a socket connection is established the communication server. At the end of the initialization the object tries to read from the socket and blocks itself waiting for a message. At the start time of a service, the dispatcher instructs the communication server to deliver a particular message to the object containing the service. The message is received by the object and forwarded to the appropriate service based on the information in the message header. When the service execution completes, the object attempts to read from the socket again and thus blocks itself. While a service executes, the dispatcher waits for either a timer signal to indicate that the execution has exceeded its estimated time, or a message from the communication server that the object has completed its execution. An example of this process is shown in Figure 2, where A is the finishing object, and B the next one to start, according to the calendar.

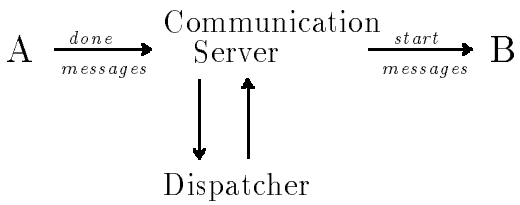

Figure 2: Interaction of objects, dispatcher and CS

3.2 Language

Since *time* is an essential property of real time applications, programmers need special constructs to express time in programs. In addition to the timing functionality, the encapsulation (objects) requirements also must be present. Rather than designing a new language and writing a new compiler for it to achieve the necessary functionality, we chose to extend an existing language and translate our extensions using a precompiler[17, 16].

The precompiler automatically generates the control part and joints of the objects with all the service and resource requirements. The precompiler also converts each service into a procedure-like call which is treated as an asynchronous invocation of the service. The approach of using the precompiler also has the advantage that the base language can be changed rather easily by generating the appropriate libraries in the new language, and modifying the translator.

3.3 Distributed Operation

The main problem of running a hard real-time system in distributed environment is clock synchronization [5]. Since time-based schedules are highly dependent on the notion of global time, scheduling decisions must take into consideration the clock granularity and drift. At present, we are using NTP[13, 14] for clock synchronization.

In distributed systems, processing nodes may have different data representations. Our message based communication model lends it self well to heterogeneous operation and with all remote communication going through agents, which handle the data translations. The current scheme uses XDR[7] as the intermediary format.

4 Concluding Remarks

The basic design of the *MARUTI* system is based on a few principles identified in this paper. Starting from these principles the system has been designed and a prototype has been implemented to run on UNIX platform. Our success in this effort to date has established the feasibility of the approach we have taken in the design of an integrated system.

A key feature of the *MARUTI* system has been its uniform treatment of time for all its entities. While we have demonstrated this feature through our implementations, several extensions will be made in the future. For example, we recognize the need for having soft and hard real-time components in an application. The mechanisms for permitting such applications to execute in *MARUTI* environment can be implemented as straight forward extensions.

The prototyping effort has not only provided a proof of concept of *MARUTI* , it has been useful as a learning tool. For example, our initial design contained a single scheduler which was used for resource allocation as well as for execution of services, during which it carried out dispatching functions. Clearly, allocation and verification are not hard real-time activities, but dispatching is. Recognizing this we divided the functions into two objects, the verifier and the dispatcher. This change made the design much simpler and could be achieved easily. Further, this split in functionality enhances the distributed operation in that the verifier and the dispatcher may execute on different processing nodes.

One of the major limitations of implementing *MARUTI* on a UNIX platform has been the lack of direct control over the resources of the system and the resource allocation policies. In order to get direct access to the resources without losing the portability, we plan to port the *MARUTI* system to run on MACH. We believe that many abstractions in MACH, e.g. tasks, threads, and user level kernel servers, are well suited for the implementation of *MARUTI* objects and services. The MACH abstractions will also be helpful in implementing multi-resource scheduling policies.

In order to establish the usefulness of this system, we have implemented several demonstration programs. These programs include a simulated air traffic control for 2-D space, a block sorter, and an AI planning based

pursuer/Invader. During the next few months we plan to acquire additional problems from the guidance, navigation, and control domain, as well as actual example applications.

The basic approach taken regarding security requirements in the *MARUTI* system is to carry out security checks prior to the execution. We are examining how realistic security requirements can be incorporated in a real-time environment. This will require not only the examination of the applicable techniques for security enforcement but also policy issues such as what to do when a processing request has to be rejected on security grounds.

During the next phase of our effort we are incorporating the fault tolerance capabilities and establishing the feasibility of the approach of constructing a resilient computation graph for applications. Resource allocation policies have to be extended to accommodate the constraints imposed by the fault tolerance requirements.

The *MARUTI* system is being developed within the framework of *MARUTI* project. In this project we are not only designing and building the *MARUTI* system but also conducting research on topics such as resource scheduling and allocation, fault tolerance, detection and recovery, and security. We are also addressing the problems of the real-time application development and necessary life cycle support through the development of *MARUTI* tools.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tervanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Summer USENIX Techical Conference and Exhibition*. USENIX, 1986.

[2] Ashok L. Agrawala. Systems engineering approach to time-driven systems. In *CompEuro 90 Tel Aviv Israel*. IEEE, 1990.

[3] Kenneth P. Birman, Thomas A. Joseph, and Frank Schmuck. *ISIS - A Distributed Programming Environment: User's Guide and Reference Manual*, March 1988.

[4] David R. Cheriton. The V kernel: A Software base for Distributed Systems. *IEEE Software*, 1(4):19–42, April 1984.

[5] Falviu Cristian. Reaching agreement on processor group membership in synchronous distributed systems. Technical report, IBM, 1990.

[6] P. Dasgupta and R. J. LeBlanc Jr. The structure of the clouds distributed operating system. In *Proc. of Workshop on Operating Systems for Mission Critical Computing*, Sept. 1989.

[7] Sun Microsystems Inc. Rfc1014, xdr: External data representation standard. Network Working Group Request for Comments: 1014, July 1987.

[8] E. D. Jensen, R. K. Clark J. D. Northcutt, S. E. Shipman, F. D. Reynolds, D. P. Maynard, and K. P. Loepere. Alpha: An operating system for the mission-critical integration and operation of large, complex, distributed real-time systems - an overview. In *Proc. of Workshop on Operating Systems for Mission Critical Computing*, Sept. 1989.

[9] S. T. Levi and A. K. Agrawala. Temporal Realations and Staructures in Real-Time Operating Systems. Technical Report CS-TR-1954, University of Maryland, College Park MD 20742, December 1987.

[10] Shem-Tov Levi and Ashok L. Agrawala. *Real Time System Design*. McGraw-Hill, New York N.Y, 1990.

[11] Shem-Tov Levi, Satish K. Tripathi, Scott D. Carson, and Ashok L. Agrawala. The *MARUTI* hard real-time operating system. *ACM Operating System Review*, 23 no 3, June 1989.

[12] Stanley B. Lippman. *C++ Primer*. Addison-Wesley, Reading, Mass., 1989.

[13] Dave L. Mills. Internet time synchronization: The network time protocol. Network Working Group Request for Comments: 1129, October 1989.

[14] Dave L. Mills. Network time protocol (version 2) specification and implementation. Network Working Group Request for Comments: 1119, September 1989.

[15] D. Mossé and A.K. Agrawala. Resilient computation graphs for distributed real time environments. Technical report, Univ of Maryland, 1991.

[16] Vivek Nirkhe and William Pugh. Application of Partial Evaluation to Hard Real-Time Programming. In *Eighth IEEE Workshop on Real-Time Operating Systems and Software and 17th IFAC/IFIP Workshop on Real-Time Programming*. Pergamon Press, May 1991.

[17] Vivek Nirkhe, Satish Tripathi, and Ashok Agrawala. Language Support for the Maruti Real-Time System. In *Real-Time Systems Symposium*, December 1990.

[18] D. Notkin and et. al. Interconnecting Heterogeneous Computer Systems. *Comm. of the ACM.*, March 1988.

[19] Ólafur Gudmundsson, Dheeraj Sanghi, and Ashok L. Agrawala. Invisible resource usage in unix. Technical Report 2509, University of Maryland, College Park MD 20742, July 1990.

[20] Ólafur Gudmundsson, Daniel Mossé, Ashok L. Agrawala, and Satish K. Tripathi. *MARUTI* a hard real-time operating system. In *Second IEEE Workshop on Experimental Distributed Systems*, pages 29–34. ieee, 1990.

[21] R. E. Schantz and et. al. CRONUS, A Distributed Operating System: Phase 1 Final Report. Technical Report Report No. 5885, BBN Labratories Inc., 1985.

[22] R. E. Schantz, R. H. Thomas, and G. Bono. The Architecture of the CRONUS Distributed Operating System. In *Proceedings of the 6'th International Conference on Distributed Computing Systems*, pages 250–259, May 1986.

[23] J. A. Stankovic and K.Ramamrithan. The Design of the Spring Kernel. In *Proceedings of Real-Time Systems Symposium*, San Jose, California, December 1987. IEEE.

[24] J. A. Stankovic and K. Ramamritham. The design of spring kernel. In *Proc. IEEE Real-Time Syst. Symp.*, Dec. 1987.

[25] J. A. Stankovic and K. Ramamritham. The spring kernel: Operating system support for critical, hard real-time systems. In *Proc. of Workshop on Operating Systems for Mission Critical Computing*, Sept. 1989.

[26] A. Tanenbaum and R. Von Renesse. Distributed Operating Systems. *ACM Computing Surveys*, pages 419–470, December 1985.

[27] H Tokuda and C. W. Mercer. The arts kernel: Towards predicatable distributed real-time systems. In *Proc. of Workshop on Operating Systems for Mission Critical Computing*, Sept. 1989.

[28] Hideyuki Tokuda and Clifford W. Mercer. ARTS: A Distributed Real-Time Kernel. *ACM SIGOPS Operating System Review*, 23(4):29–53, July 1989.