The Unstoppable Orchestra a Responsive Distributed Application

Matthias Werner, Andreas Polze, Miroslaw Malek
Institut für Informatik
Humboldt-Universität zu Berlin
Unter den Linden 6

10099 Berlin, GERMANY

e-mail: {mwerner,apolze,malek}@informatik.hu-berlin.de

ABSTRACT

Responsiveness, the ability to provide real time behavior even in presence of faults, is becoming one of the most sought after properties in distributed computing systems. We present a framework for ''High-Performance Responsive Computing'' in networked systems whose current implementation works on a network of five NeXTSTEP (Mach 2.5)-based HP workstations.

''The Unstoppable Orchestra'' is a demonstration program which runs in our environment. This application simulates a real orchestra consisting of musicians playing different instrumental parts of a classical masterpiece. However, instead of real musicians our orchestra is made up of computers equipped with speakers. In contrast to the real orchestra, where a sudden disappearance of a flutist can hardly be hidden, in our orchestra one ''musician'' can take over his neighbour's part, ensuring uninterrupted performance.

Within this paper we discuss the design issues behind the ''Unstoppable Orchestra'' and show how our programming framework supports synchronization and re-configuration of responsive applications.

Key words: *responsiveness*, *consenus objects*, *real time*, *fault tolerance*, *reconfiguration*.

1. Introduction and Motivation

Responsiveness is a measure of the ability to perform tasks in real time even in presence of faults under a given fault and load hypothesis. We view consensus as a general paradigm of responsive systems. Synchronization, scheduling, initialization, termination, load balancing, data commit or consistency can be considered as consensus problems. In this paper we present a distributed application, where the need for responsiveness is obvious: the ''Unstoppable Orchestra'' which is able to play a music continuously on multiple computers even if some of them fail.

Playing music on a computer system is a non-trivial task. However, a number of programming libraries exist which usually solve the problem for a single computer. The task gets further complicated if a number

of independent computers is involved in playing a single piece of music. Configuration and synchronization of the system are the two main problems the programmer of a distributed sound system has to face. Handling of faults additionally requires reconfiguration of such a system in real time. To motivate our work, let us briefly to describe the main issues the programmer of a responsive application has to deal with.

An orchestra playing some piece of music can be simulated by a number of tasks placed on different hosts of a distributed computer system. Interconnections between those tasks (the musicians) can be established to form a complete graph. These interconnections can be used to send a number of initial synchronization messages and to establish a global time. Finally, the orchestra can start playing at once.

Besides feeding its sound to the speaker, each task now has to pay attention to staying synchronized with the others. Messages have to be exchanged by any pair of ‘‘musicians’’ (‘‘playing computers’’) to keep this synchronization alive. Furthermore all the tasks have to agree on the status of the system, abnormal termination of one task has to be reported to all others.

After detecting a task’s abnormal termination the remaining tasks have to elect one which takes over the job of the failed computer. To continue playing of the additional sound the latter one has to reconfigure its sound device. Then, relying on the global time, it can resume playing of two (or more) sounds. Since the human listener should not even notice, the whole procedure has to be performed as quickly as possible. After a while, the orchestra is done with its performance. In our case, the responsive application has terminated normally.

Current operating systems provide a very limited support to the programmer of fault tolerant real time applications. Only rudimentary support exists for clock synchronization, collection of distributed statuses (checkpointing) and detection of communication failures or a task’s abnormal termination. Our framework implements those features through a responsiveness layer on top of an operating system’s microkernel. It consists of the CORE’s — COnsensus for REsponsiveness — and SONiC’s — Shared Objects Net-interconnected Computer — runtime systems. Both can easily be interfaced by the programmer of responsive applications through an C++ class library.

The remainder of this paper is organized as follows: Section 2 presents the architecture of our framework for ‘‘High-Performance Responsive Computing’’. Section 3 describes in some detail the design of the ‘‘Unstoppable Orchestra’’ application. Section 4 discusses issues of our current, prototypical implementation and presents some experimental results. Section 5 browses through related work and, finally, Section 6 gives conclusions and an outlook on future work.

2. A Framework for High-Performance Responsive Computing

Our framework integrates the objectives of responsive systems — fault tolerance and timeliness — with issues associated with parallel computing in distributed environments — idle-time computing, ease-of-use, and communication avoidance. Based on a number of interconnected PC's and workstations we provide a layered software architecture. We use a hierarchical model for each processing element in our system as depicted in Fig. 1 .

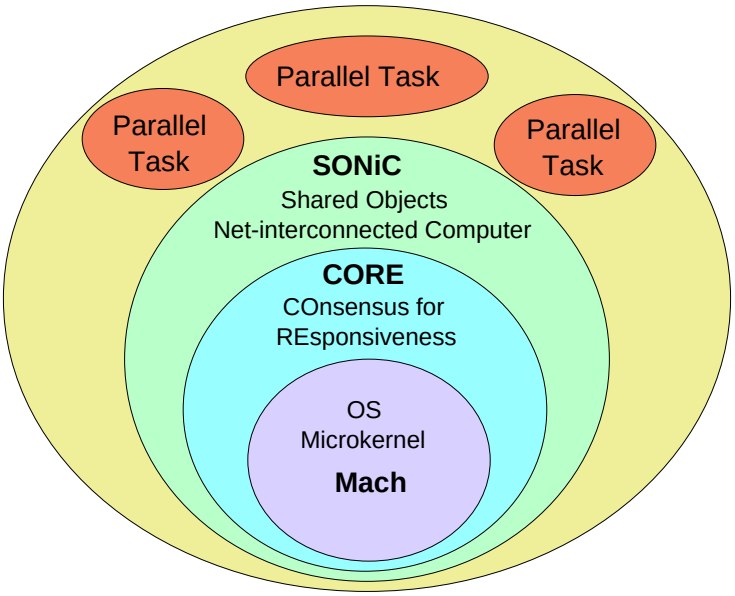

Fig. 1*: Kernel, Responsiveness Layer and Memory Management*

In our model several parallel tasks may run on a number of interconnected computing units. Those tasks are programmed using the parallel programming interface of ‘‘Shared Objects Memory Net-interconnected Computer (SONiC)’’, our object-based distributed shared memory system. This model provides transparent data communication. It is similar to the conventional sequential programming style and is therefore familiar to programmers.

COnsensus for REsponsiveness (CORE) provides protocols, services, and scheduling strategies at the microkernel level for real time parallel computing even in presence of faults. Among other things, it will provide a reliable communication subsystem to the ‘‘Shared Objects Net-interconnected Computer’’. CORE, in turn will use the services of the underlying Mach microkernel operating system. The network drivers of the Mach system interface with Ethernet and will soon interface with ATM as well.

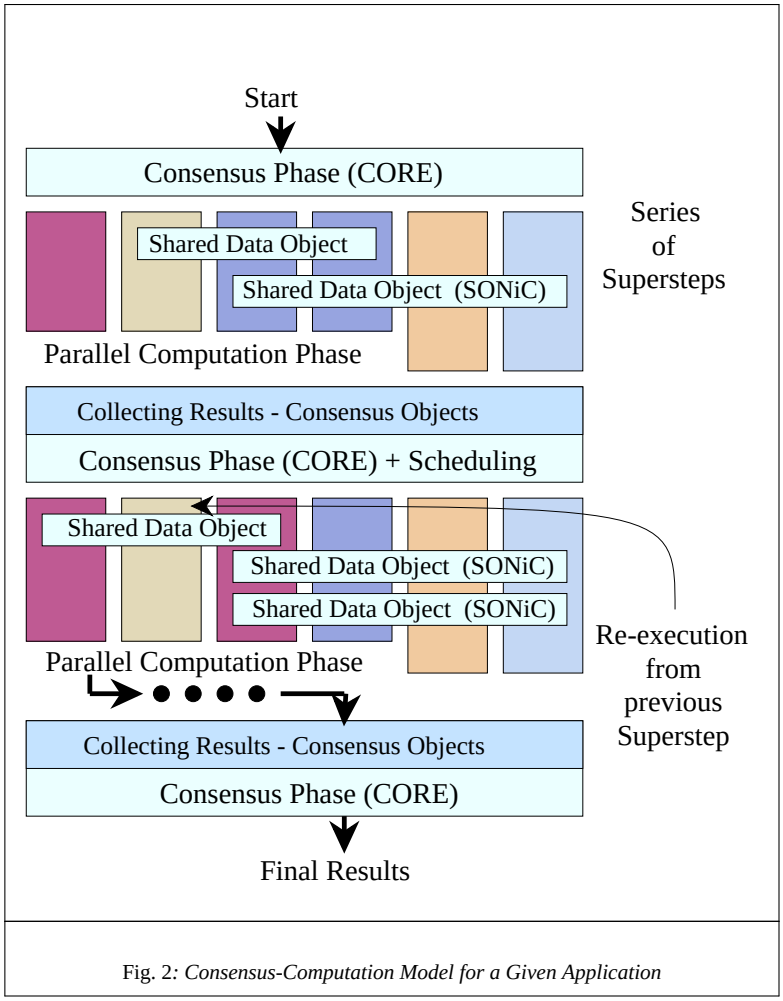

Fig. 2*: Consensus-Computation Model for a Given Application*

Responsive applications in our environment are executed following a consensus-computation model. According to our model, applications perform several independent activities with different deadlines which run in different threads. Those threads are executed on the nodes of our network-based computing system. To ensure fault tolerance, threads are eventually replicated. Re-execution of failed threads on a different node of our system is another way to deal with faults. However, replication works best for threads with short deadlines whereas re-execution can be applied only if a thread's deadline is more than twice the execution time away. Each application is seen as a series of supersteps, consisting of a consensus and scheduling phases and a parallel computation phases. Fig. 2 shows the execution model in our environment.

At the beginning of each consensus and scheduling phase, the CORE subsystem determines the status of the responsive application based on consensus objects. Additionally agreement about time (synchronization) and about faulty and non-faulty processors (system status) is achieved. Consensus objects are implemented as replicated objects with a special, consensus-defined consistency model. The system status information and programmer-specified deadlines are used to evaluate system activity during the previous

computation phase. Afterwards load-distribution and scheduling for the next computation phase is performed. Again, redundancy in space and time (replicated execution and re-execution) is employed to deal with faults.

During a parallel computation phase the program threads communicate via replicated shared objects. Consistency among those objects is maintained following a weakly consistent memory management scheme. The C++ class construct is used to hide details of our implementation of shared objects from the application programmer. A whole hierarchy of classes allows for easy implementation of programmer-defined shared data structure. By using classes from a particular part of the hierarchy the programmer can control which consistency protocols are employed in a particular application. Since the programmer has full control over the memory layout of shared objects, problems like false sharing can easily be avoided. Besides ordinary shared objects, consensus objects appear in a different part of our class hierarchy. By using those objects at appropriate places throughout a program the programmer can influence how the status of a distributed responsive application is made up. Fig. 3 gives a brief overview over our programming framework: a hierarchy of shared classes.

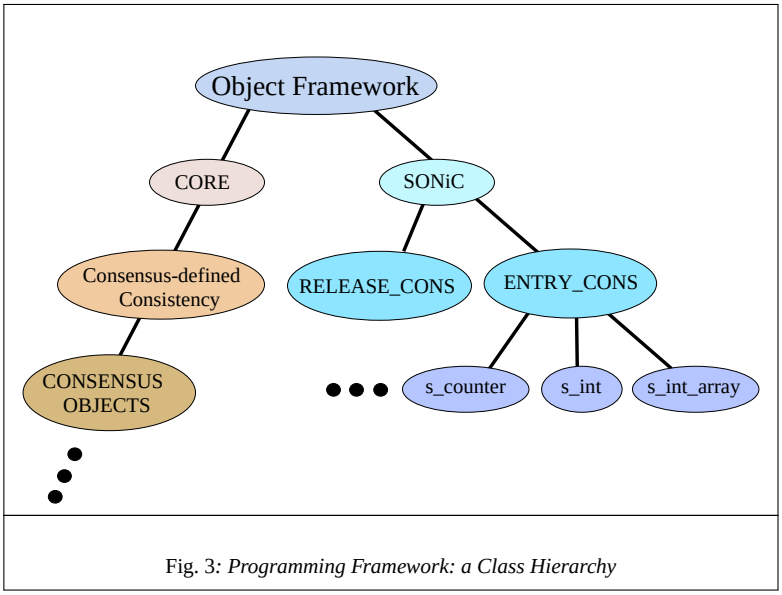

Fig. 3*: Programming Framework: a Class Hierarchy*

3. Responsive Application Design

The service our application provides is the distributed performance of a piece of music consisting of several sounds. Thereby, every instrumental part shall be played by one host. If there are more (non-faulty) hosts than instruments, we use the ‘‘superfluous’’ hosts as stand-by units. When a host fails, another one takes over its part. If there is insufficient number of hosts such each one may play only one instrument, the

application must ensure a fair load distribution. Under such circumstances we may lose some sound quality, since a possible stereophonic effects will diminish with increasing number of failed hosts. After some time, eventually, all instruments are played by a single host. If it fails, we get a system failure. I.e., our system can tolerate n-1 faults with a graceful degradation.

To make a statement about the system's correctness in presence of faults, we need a model of the system's fault behavior. Since in our example the single nodes are independent and connected by a network only, we can apply the fault model by Laranjeira et al. [Laranjeira et al. 93]. This model is an extension of Cristian's fault model of services in a client-server system [Cristian et al. 85] [Cristian 89]. Fig. 4 gives a survey about fault classes in the used model.

Fault Class	Behavior
Fail Stop	A processing element (PE) ceases operation and alerts other processors of this fault.
Crash Fault	A PE loses its internal state or halts. The processor is silent during the fault. This fault class includes the fail stop.
Omission Fault	A PE fails to meet a deadline or begin a task. This fault class includes the crash fault.
Timing or Performance Fault	A PE completes an assignment before or after its specified time frame or never. This fault class includes the omission fault.
Incorrect computation Fault	A PE fails to produce a correct output in response to a correct input. This fault class includes the timing fault.
Authenticated Byzantine Fault	A PE behaves in an arbitrary or malicious manner, but is unable to im- perceptibly change an authenticated message. This class includes the incorrect computation fault.
Byzantine Fault	Every possible fault. This class includes the authenticated Byzantine fault.

Fig. 4: Extension to Cristian's fault model by Laranjeira et al.

For our application we consider all fault classes up to a timing fault. However, also computation faults or Byzantine faults are possible. In these cases, the consensus protocols do not have to mask faulty results, but to detect the faulty hosts and exlcude them, since all participating (not stand-by) hosts are involved in output of the music. The real-time restrictions of our application depend on the human physiology. We

assume, that a desynchronization is hearable, if it is larger than 0.05 seconds. Also, differences in the sound length can be noticed, if they are larger than 0.2 seconds. Hence the fault tolerance times are small and tones cannot be recovered, we use a roll-forward strategy.

As mentioned above, we want to apply the consensus paradigm. Thus, we can identify three consensus problems in our application:

- consensus on the group membership (who are members of the orchestra ?)
- consensus on the time (what part of music is played at what time?), and
- consensus on load distribution (which instrument is played by each member?)

All these problems (membership, synchronization and load distribution) are explored more or less thoroughly. All of them are typical problems in a distributed system. However, we want to solve these problems within a uniform framework. For this purpose, we introduce consensus objects. These objects provide an encapsulation of used consensus protocols. We see problems inherent to our application as typical for a wide selection of responsive applications. The consensus techniques our application relies on ultimately will become part of the CORE layer of our framework for High-Performance Responsive Computing [Malek et al. 95].

4. Experimental Results: A Prototypic Implementation

We have implemented a prototypic version of the ‘‘Unstoppable Orchestra’’ on a network of HP-workstations running the Mach 2.5 (NeXTSTEP) operating system. We use Mach interprocess communication for exchanging messages between members of the orchestra and for achieving consensus over the application’s state. Currently, we use standard Ethernet as interconnection network for our experiments. As prototype for a responsive application the ‘‘Unstoppable Orchestra’’ exploits many concepts which are also present in our framework for High-Performance Responsive Computing. We have tried to structure the application as a set of C++ classes, e.g. for synchronization and fault detection, which can be seen as a part of our CORE framework.

Our application consists of a number of player-tasks running on different nodes of the network. Each task maintains a set of Mach ports which represent connections to every other task. Thus, we establish a complete interconnection graph among the members of our orchestra. A master-task — the conductor, if you will [1] — then allows to initiate playing of a particular piece of music with n instrumental parts.

[1] In contrast to our system, the human conductor would ‘‘synchronize’’ the human musicians playing their instrumental parts in an orchestra,

Once we have established connections between members of our orchestra, we can start playing music. However, to ensure simultaneous playing by all ‘‘musicians’’, the player-tasks must be synchronized. We assume that the physical (quartz) clock’s drift between the different musician’s hosts can be ignored for the duration of the performance. Thus, we basically have to synchronize all player-tasks before starting to play. Otherwise, in addition we have to run a synchronzation protocol.

We further assume that exactly one player-task is present on every host in our environment. Therefore we can use the ‘‘Network Time Protocol (NTP)’’ [Mills 92] to adjust the participating hosts’ clocks. This protocol is based on the idea of sending a number of messages between two peers, putting local and remote timestamps on the messages and estimating the mean transit delay by comparing those timestamps. Once the transit delay for a particular connection is known, another pair of timestamped messages is exchanged. However, this time the timestamp values are used for adjusting the participating computers clocks. The whole NTP is much more complex than the sketch presented here, e.g., it deals with hierarchies of clocks and more than just two peers.

Now, if we have synchronized clocks on the player-task’s hosts, we can start our performance using a little trick: If the master (‘‘conductor’’) indicates to one player-task (‘‘the first violin’’) to start, this task sends a message to every other player. However, before actually start playing its instrument every player sleeps for two seconds (i.e., every player-task executes a `alarm(2)`-system call). The advantage of `alarm()` is the wake up: a signal is sent to the sleeping tasks — on a full second. Since message transit delay is much smaller than one second and since clocks on our computers are synchronized, all players wake up at the same moment (with < 1 ms precision) and start playing. Playing music happens by writing a soundfile’s contents to the special sound device. NeXTSTEP provides a sophisticated programming library for dealing with soundfiles which we do not discuss in detail here.

Once the performance is started, all the player-tasks are monitoring each other. Two means are used to detect up to `(n - 1)` crash- and timing faults. Mach provides (network-transparent) kernel notifications for a variety of events. One particular interesting event is the (abnormal) termination of a task owning receive rights to a communication port. In this case the Mach kernel sends notification messages to all tasks holding send rights to that particular port. Since we have established a complete graph of interconnections between player-tasks, each surviving player receives a kernel notification if one ‘‘musician’’ disappears from the orchestra. A round of consensus is necessary to agree on the candidate which has to take over the disappeared player-task’s voice. During this process eventually even a complete reassignment of instrumental parts to player-tasks may happen. Whereas the crash of a particular task can be easily detected using kernel notifications, another technique has to be used in cases where a whole computer crashes.

Mach provides consistent notification about system state, however, no kernel notification is sent at the particular moment of a computer's crash but a re-started computer notifies all its peers ''that there was a crash''.

Sending ''I am alive'' messages to peer tasks is an easy way to detect timing faults. In our application each player-task sends permanently messages to all other player-tasks. If no such task arrives for a given period of time from a particular task, that ''musician'' is assumed to be faulty. Then a round of consensus among the remaining orchestra members takes place and a decision is made about the candidate who continues the disappeared musician's part. We employ a number of different threads within each player-task to receive kernel notifications and ''I am alive''-messages, to send periodically messages to the peer player tasks and, finally, to perform the actual playing. Fig. 5 shows the internal structure of each player in the ''Unstoppable Orchestra.''

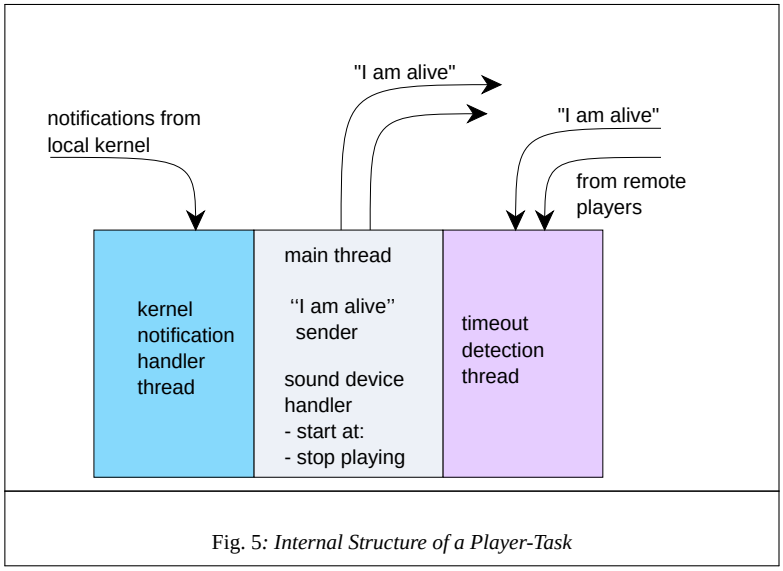

Fig. 5*: Internal Structure of a Player-Task*

Currently all the code related to kernel notifications is encapsulated in a separate C++ class — which maintains a separate thread for receiving those notifications. Another class implements maintenance of complete interconnection graphs between a number of peer-tasks. Periodically exchanging ''I am alive'' messages along the edges of such a graph can be initiated by a call to a particular member function (which spawns another thread) of the graph's class. Both classes are a fundamental part of our framework for ''High-Performance Responsive Computing.''

The implementation of synchronization among the ''Unstoppable Orchestra's'' musicians is rather preliminary. However, the underlying NTP-protocol proves to be highly usable. In line with our framework, we will ultimately implement a class for synchronized virtual clocks, This class will use a similar techniques like NTP and the `alarm()`-system call to provide synchronous notifications to a number of tasks.

5. Related Work

Ensuring certain quality-of-service parameters such as timeliness, delay, jitter, and guaranteed bandwidth for interactions in distributed environments has become a central focus for many research projects in the field of multi-media applications. However, only few applications have been reported, that use responsive computing techniques. In contrast, a number of general approaches to fault tolerant real time computing exist and have been implemented in programming frameworks.

The Software-Implemented Fault Tolerance (SIFT) project for aircraft control provides a means of replicating tasks among a number of tightly coupled processing elements configured in a special architecture [Wensley et al. 78]. Tasks are scheduled off-line to fill some master scheduling period and are executed cyclicly regardless of whether or not they need to be executed. A configuration manager keeps track of which processing elements actually execute a task so that tasks dependent on the result can request these outputs and perform a majority vote on them prior to execution.

The Multicomputer Architecture for Fault Tolerance (MAFT) also provides fault tolerance and timeliness for a specific architecture [Walter et al. 85]. In order to overcome the overhead hurdles seen in SIFT project, each processing element consists of two processors: one which handles all task scheduling, synchronization, and voting of the processing element, and another is responsible for executing application task and controlling devices. Fault tolerance is archieved via replication and by using Byzantine agreement algorithms on all globally needed data such as application result and scheduling information.

The Maintainable Real-Time System (MARS) relies on fail-silent assumptions of its processing elements [Kopetz et al. 89]. Here, the nodes of a distributed system consist of a number of tightly synchronized processing elements each of which executes the tasks assigned to its particular node. Since the system uses static scheduling all tasks must be scheduled a priori.

A typical application-oriented approach to responsiveness is given in [Harrick et al. 95]. The authors describe a failure recovery for data loss in video file servers. That method uses the inherent redundancy in video streams. It assumes video data that are dispersed among several disks. A clever dispersal algorithm provides a graceful degradation in case of a disk crash.

6. Conclusions and Future Work

With the ‘‘Unstoppable Orchestra’’ we have designed and implemented a typical responsive application. Our application consists of a number of player-tasks which run on different nodes of a workstation-

network. Like musicians in a real orchestra the player-tasks are synchronized and are able to perform different instrumental parts of a piece of music. However, in contrast to the real orchestra our application can hide the disappearance of all but one player-tasks. A consensus protocol is used to decide which task should take over a terminated task's part. Of course, at some point the stereophonic effect of the performance disappears. Thus, by employing dynamic reconfiguration our application implements graceful degradation.

We have applied a multi-threaded design for the distributed player-tasks. Mach kernel notification messages and ''I am alive'' messages are accumulated by different threads. Based on those messages all the player-tasks receive a consistent view of the status of the whole distributed application. The functionality necessary for maintaining an application's global status has been encapsulated within two easy-to-use C++ classes. Both classes appear in the foundation of the CORE (COnsensus for REsponsiveness) layer of our framework for High-Performance Responsive Computing.

Synchronization within our prototype implementation of the ''Unstoppable Orchestra'' is achieved using NTP (the Network Time Protocol) and a somewhat tricky implementation detail in context of the `alarm()`- system call. Ultimately, the CORE layer will contain a separate C++ class which will implement synchronized virtual clocks based on similar techniques. All communication currently is implemented as message-passing (Mach interprocess communication) based on Ethernet. In an unloaded state this communication base proves to be able to provide sufficiently accurate synchronization and fault detection to satisfy the human listener.

In the future we will study the impact of quality-of-service parameters of different communication media on the synchronization and fault detection mechanisms currently used in our application. We are porting this application to our testbed installation where four Pentium-based computers running the Mach operating system are interconnected via ATM. We plan to experiment with a wide range of applications demanding high level of responsiveness.

References

[Barborak et al. 93] Barborak, M., Malek, M, Dahbura, A,
The Consensus Problem in Fault-Tolerant Computing;
ACM Computing Surveys, 25(1993)2, 171-220.

[Cristian et al. 85] Cristian, F., Aghili, H., Strong, R., and Dolev, D.;

Atomic Broadcast: From Simple Message Diffusuion to Byzantine Agreement;

IBM, Technical Report RJ 5244 (54244), 1985.

[Cristian 89] Cristian, F.;

Synchronous Atomic Broadcast for Redundant Broadcast Channels;

IBM, Research Report, RJ 7203 (67682), 1989.

[Harrick et al. 95] Harrick, M.V., Prashant, J.S., and Sriram, R.;

Efficient Failure Recovery in Multi-Disk Multimedia Servers;

25th Symposium on Fault-Tolerant Computing, 1995.

[Kopetz et al. 89] H.Kopetz, A.Damm, C.Koza M.Mulazzini, W.Schwabb, C.Senft, R.Zainlinger;

Distributed Fault-Tolerant Real-Time Systems: The MARS Approach;

IEEE Micro, vol. 9, 1, 1989.

[Laranjeira et al. 93] Laranjeira, L., Malek, M., Jenevein, R.;

Nest: A Nested-Predicate Scheme for Fault Tolerance;

IEEE Transaction on Computers, vol 42, 11, 1993

[Malek et al. 95] Malek, M., Polze, A., Werner, M.;

A Framework for Responsive Parallel Computing in Network-based Systems;

Proceedings of International Workshop on Advanced Parallel Processing Technologies, Bejing, China, September 1995.

[Mills 92] Mills, D. L.;

Network Time Protocol (Version 3) Specification, Implementation and Analysis;

RFC-1305, via ftp from NIC.DDN.MIL, March 1992.

[Walter et al. 85] Walter,C., Kieckhafer, R., Finn, A.;

MAFT: A Multicomputer Architecture for Fault-Tolerance in Real-Time Control Systems;

IEEE 1985 Real-Time Systems Symposium, 1985.

[Wensley et al. 78] Wensley,J., L.Lamport, J.Goldberg, M.Green, K.Levitt, P.Melliar-Smith, R.Shostak, C.Weinstock;

SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control;

Proceedings of the IEEE, vol 66, 10, 1978.