High Availability in The Real-Time Publisher/Subscriber Inter-Process Communication Model

Ragunathan Rajkumar* and **Mike Gagliardi**+
*Department of Computer Science
+Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
raj+@cs.cmu.edu, mjg@sei.cmu.edu

Abstract

The real-time publisher/subscriber (RT P/S) communications model has been proposed as a flexible and powerful inter-process communication model for distributed real-time systems [12]. It can also be used as the underlying framework for supporting building blocks such as *extensible cells* and *replaceable software units* for building evolvable distributed real-time systems [7]. However, for the model to be adopted in practice, it must tolerate processor failures and allow repaired processors to rejoin the system on a dynamic basis. Such processor failures and rejoins must also not hurt the efficiency of the steady-state publication/subscription of messages by repetitive real-time processes. In this paper, we present extensions to the RT P/S model to support these capabilities. The solution is structured in two layers. First, an efficient processor membership protocol layer, based on Cristian's *periodic broadcast* membership protocol [4, 6], detects processor failures and rejoins. It provides strong semantics and exhibits a finite delay in detecting processor failures. Secondly, idempotence properties, weak interleaving needs and the benign impact of node failures within the RT P/S information structure enable us to transfer consistent state to newly joining daemons and to manage changes to the information elegantly. The changes are orthogonal to the communication programming interface and also maintain very efficient and analyzable steady-state real-time execution paths. These protocols have been successfully built in the context of both feedback control and multimedia dissemination applications.

1. Introduction

The real-time publisher/subscriber (RT P/S) model [12] is a many-to-many communications model for distributed real-time inter-process communication. The RT P/S model is related to group-based programming techniques [8] and the anonymous communication model [11]. Such a model in a general non-real-time context is also sometimes referred to as "blindcast". In this model, "publishers" act as data sources and "subscribers" act as data sinks. A publisher can also be a subscriber and vice-versa. A *distribution tag* (or *name*) identifies each category of data item published. Many publishers can publish on the same tag, and many subscribers can receive on the same tag. The publisher on a distribution tag need not know how many subscribers are subscribed to that tag. Similarly, a subscriber to a tag need not know how many publishers are publishing on a tag. All that the application(s) have to do for using this communication model is to agree upon tags (names) of the data items that will be published and their formats.

The RT P/S model described in [12] does not tolerate node or process failures, nor does it support dynamic joining of nodes. As a result, all RT P/S daemons must be up and running before the application(s) requiring inter-process communication services can start. If a daemon or a node dies, it essentially cannot come back into the system. The probability of component failures in a distributed system over a relatively reasonable amount of time is fairly high. Hence, the lack of fault-tolerance and lack of rejoining capabilities constitute a serious impediment to the adoption of the *distributed* RT P/S communications model. In this paper, we address this rather serious limitation and allow processors to fail and later rejoin the system dynamically after repair. Since the communications are meant to be real-time in nature requiring quick and predictable communications, we pay special attention to keeping the steady-state path for real-time communications very efficient and analyzable.

The rest of this paper is organized as follows. In Section 2, we present a brief overview of the real-time publisher/subscriber communication model (RT P/S model) and define the requirements for supporting high availability in this model. In Section 3, we discuss a strong membership protocol that can detect and notify processor failures in a finite amount of time as well as allow processors to rejoin the processor group. In Section 4, we discuss the impact of processor failures and joins on the RT P/S communication infrastructure and how they can be accommodated. Finally, in Section 5, we discuss our prototype experiences.

2. High Availability Requirements in The RT/PS Communication Model

2.1. Overview of the Publisher/Subscriber Model

The architecture of the RT P/S communication infrastructure is given in Figure 2-1.

The IPC Daemon: Each IPC daemon consists of three threads of control running at different priorities:

- *Local Manager*: A local manager thread responds to requests for tag creation/deletion and publication/subscription rights from clients on or in close proximity to this node.
- *Update Manager*: The update manager thread manages tag updates. Any local manager thread receiving a request to change the status of a distribution tag sends the status update notification to the update managers on all the other nodes, and then updates its local copy. They may also notify any of their local clients which have a copy of that tag. These asynchronous client notifications are handled by the update thread in the client library.
- *Delivery Manager*: If there are multiple subscribers on a tag at

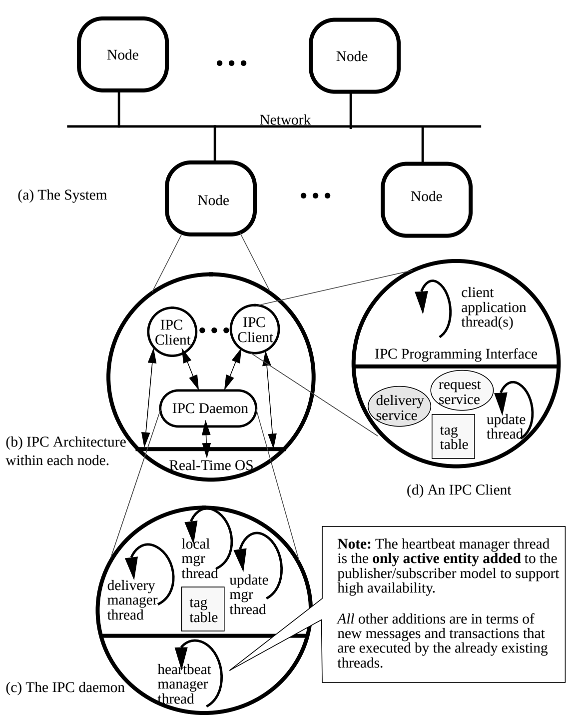

Figure 2-1: The Real-Time Publisher/Subscriber Model with High Availability Support

a remote node, it is often wasteful to send many duplicate network messages to this node. The delivery manager is intended to address this (potential) performance problem. The client just sends one copy of its message to a delivery manager on the node, and the delivery manager delivers the message to all receivers on its local node (using a locally efficient mechanism).

The Client-Level Library: In the user-level library on the client side, a *local request service*, a *delivery service* and an *update service and thread* act as counterparts to the above three threads in each IPC daemon.

The newly added *heartbeat manager thread* within each daemon will be discussed in Section 3.1. The RT/PS implementation [12] is such that a daemon need *not* be running on every node.

2.2. High Availability Requirements

The RT P/S communications infrastructure consists of a common repository of information: the tags in the system along with the list of publishers and subscribers on each tag. Since there are multiple nodes and daemons, this information must be distributed across the daemons. In addition, since the publishers (subscribers) represent real-time activities, they must be able to publish (subscribe to) messages very efficiently during steady-state operations. To satisfy this critical need, the RT P/S library at each client maintains a subset of the distribution tag information which pertains to the publication/subscription needs of that client. When a publisher publishes a message, the list of subscribers is available locally in the client library and the publication happens directly to the corresponding subscribers.

Two kinds of changes must be managed in this infrastructure:

- Any changes to a distribution tag must be globally consistent among the daemons. In other words, the distribution tag information maintained by the IPC daemons represents a distributed (or replicated) database.
- Any changes to information about a tag must be notified to the client(s) using the tag. In other words, the tag information maintained by the library in the clients represents a distributed caching mechanism.

We define the following requirements to support real-time and high availability characteristics:

- **R1.** The ability to tolerate component failure:
  - **R1.1.** The failure of (at least some) nodes must be tolerated.
  - **R1.2.** The failure of (at least some) processes must be tolerated.
- **R2.** The ability to allow rejoining of components:
  - **R2.1.** A failed node must be able to rejoin the system at a dynamic point in time.
  - **R2.2.** A failed process must be able to rejoin the system.
- **R3.** The steady-state path of publication/subscription should be efficient. Specifically, communication between two or more processes must be nearly or more efficient than direct process-to-process communication without the RT P/S model.
- **R4.** The resource demands (e.g. CPU processing, network bandwidth) imposed by the RT P/S infrastructure must be analyzable and have well-defined timing properties.

In this paper, we primarily address how we meet requirements R1.1, R2.1, R3 and R4.

3. Detection of Node Failures and Joins

The ability to tolerate node failures (requirement R1.1) and to allow processors to rejoin the system (requirement R2.1) can both be facilitated by the knowledge within the RT P/S infrastructure about processor failures and rejoins. This is typically done in distributed systems using a variation of a processor membership protocol. The "group membership problem" has been widely studied for both synchronous and asynchronous systems (e.g. [3, 5, 9, 10]) and experimental systems using the concept have been built (e.g [3, 2, 16]). However, these protocols have traditionally been used in non-real-time systems. Their impact on predictable timing behavior and hard deadlines has been rather unclear and is the subject of this paper. Abdelzaher et al. report on a recent piece of work with similar objectives in [1]. The Team project in the University of California at San Diego is studying techniques to tolerate timing, crash and communication failures while still meeting hard real-time deadlines.

Our membership protocol, described in detail in [14], assumes a synchronous communication network and is a variation of Cristian's "periodic broadcast" membership protocol [4, 6]. A group leader periodically broadcasts "`Are You Alive?`" messages to which other group members respond with "`I Am Alive`" messages. Group leader deaths are detected by the absence of "`Are You Alive`" messages and the next ordered group member becomes the group leader and the sequence repeats. Our group membership protocol does *not* support network partitioning, *nor* does it support node and communication performance failures, where a node (message) can be delayed and act well again (be delivered) after some arbitrary amount of time.

3.1. The Heartbeat Manager Thread

A processor membership protocol is executed on each node in the system and enables the various RT P/S daemons to agree upon which node is currently alive and which node has failed. As illustrated in Figure 2-1, this protocol is executed by a "heartbeat manager thread" and is the only new thread of control added to the infrastructure. Additional message types and transactions had to be added to support the new protocol(s) added to support high availability. The application client interfaces remained exactly the same.

3.2. Properties of the Processor Membership Protocol

Our membership protocol has the following properties.

> **Theorem 1:** The failure of a processor in the processor group leads to an identical sequence of membership changes in the remaining processors.
>
> **Theorem 2:** The time to detect *and* notify a processor failure is bounded by $3*T_{\text{heartbeat}}$ (where $T_{\text{heartbeat}}$ is the interval between successive heartbeats), under the assumption that at most one processor failure can happen within the interval.

3.3. Strong and Weak Membership Protocols

The property of Theorem 1 makes our membership protocol a *strong* membership protocol [9]. In a strong membership protocol, all members are guaranteed to see the same sequence of membership changes. In a *weak* membership protocol, all members will *eventually* see the same processor membership when node failures and joins stabilize. Our strong membership protocol can be easily converted into a weak membership protocol.

4. Management of RT P/S Information Under Failures/Joins

A distribution tag is said to be *strongly consistent* if all the publishers on that tag see the same exact sequence of (alive) subscribers on that tag. A distribution tag is said to be *weakly consistent* if all the publishers on that tag will *eventually* see the same (alive) subscribers on that tag. It is very desirable that the convergence of subscribers on a tag in a weakly consistent RT P/S system has a small and bounded delay.

> **Lemma 3:** A publisher (subscriber) which is no longer alive to publish (receive) any messages does *not* affect the consistency property of a distribution tag.

Remark: Despite Lemma 3, it is highly desirable that information relating to dead processors is cleaned up for two reasons:

1. If cleanup does not occur, publishers and subscribers can no longer utilize any information about the number of publishers and subscribers on a tag to make decisions.
2. Cleanup must occur before a dead processor can rejoin the system to keep RT P/S information consistent.

4.1. Choosing Between Strong and Weak Consistency

Should we choose to enforce strong or weak consistency on distribution tags? We are guided on this rather complex question by end-to-end system arguments posed by Saltzer, Reed and Clark in the operating system community. The tradeoff can be framed as the following questions: "Do we want to delay every participant in the distributed system identically and at continuing high overheads for each message communication?" "Or can we expose different parties initially to different delays/views while providing much higher steady state performance?". We are in favor of the latter, and choose to implement a more efficient and more widely useful RT P/S infrastructure at the small expense of relatively minor startup delays. For those minority applications which may require strong consistency, such consistency can be built on top of the RT P/S model (at a possibly much higher expense).

Our argument that a weakly consistent RT P/S system is sufficient for many distributed real-time and multimedia applications is based on two observations about real-time systems:

1. **Domain semantics:** Real-time systems tend to be systems which control the physical environment and hence exhibit physical inertia. Techniques such as "coasting", "model-based control" or repetition of previous sensor or actuator data are very commonly used to substitute for data that is temporarily unavailable due to various reasons. For example, in self-navigating vehicles, if an object cannot be clearly detected due to a temporary masking effect, the vehicle can still use its current position, velocity and acceleration for its next output. In multimedia systems, such as video on demand or multi-point video or audio conferencing systems, there might be an initial delay when different people hear a smaller or bigger set of people, but this is almost always acceptable in practice.
2. **Application Usage**: In the RT P/S model, data is published on a tag repeatedly. When one or more subscribers are added to a tag, any real-time data that is published to them is not going to be of instantaneous use. The fact that subscribers can be dynamically added at any time implies that the application must expect that subscriber information can change. For example, in the "replaceable software unit" built using the RT/PS model [7], one or more new subscribers on a tag must stay online for a few *seconds* before one or more old subscribers can be turned off. This delay is used to ensure that a new subscriber can synchronize with the rest of the system before taking over the functionality of a current software module. The publication loop, meanwhile, is running at the order of tens of *milliseconds*. The fact that a publisher saw some subscribers in a different sequence for a relatively short interval hardly matters.

   When multiple publishers publish on the same tag, the data that they publish must be (a) identical (b) similar (e.g. obtained from similar sources such as sensors) or (c) related (such as sending various parts of the same video frame). In the first two cases, a subscriber can take any publisher data. In the last case, one or a small number of frames will be dropped. In contrast, if the tag were strongly consistent, it can incur an additional delay resulting in frames being unacceptably delayed (or dropped in a real-time interactive application).

At this point, the keen reader would question the need for strong membership at the processor group layer (see Theorem 1 and Section 3.3) while using weak consistency at the tag layer. The answer to this question is rather subtle and is discussed below.

It must be recalled that the RT P/S communication model presented in [12] does *not* guarantee any ordering or agreement among messages when messages are published using distribution tags. This choice is deliberate and represents the trade-off among efficiency, delayed communications and application-level needs. Completely strong consistency in the model can indeed be obtained by using a strong membership protocol, strongly consistent distribution tags and ordered communications at the publication layer. This model would be roughly analogous to the ISIS paradigm of distributed computing [8]. Based on the application characterizations above, we argue that this paradigm is unnecessarily constraining and inefficient in the domains of real-time and multimedia systems. Unnecessary and perhaps unacceptable overhead and delays may be introduced by such comprehensively strong consistency in all the layers. In our strong membership protocol, applications cannot access the membership information if it is marked as unstable. Such transient inaccessibility to information would percolate to higher and higher layers with added performance and delay concerns.

We argue that the strong ordering requirement is also not necessary at the application publication layer. If a similar strong ordering requirement is placed on the publication layer, the publication/subscription of messages must be delayed if publisher/subscriber information is changing and/or other publications/subscriptions are happening at the same time. The two-phase commit described in Section 4.3 that is necessary to allow a processor to rejoin the system dynamically is a good indication of how things can be slowed down if ordering and strong consistency requirements are placed at all layers. In order to facilitate application convenience, however, the RT P/S communication model presents the time-stamp at time of publication, the sequence number of publication on a tag by a particular publisher, and {`node id, process id`} information for publishers for applications to use when necessary. Similar arguments at the distribution tag layer led us to avoid strong consistency at the RT P/S layer as well. To date, we have had extensive experience with fairly complex application prototypes in feedback control [15] and network-based video dissemination to multiple receivers [13]. All these experiences indicate little or no loss of functionality and very high performance with the weakly ordered publication model and weakly consistent RT P/S information.

The question, however, still remains whether one needs to use a weak or a strong membership protocol in real-time or multimedia systems. The difference in run-time costs between these two protocols is *not* significant, at least for a relatively small number of machines. When the cost differential is not high, it seems better to strive for the stronger semantics. However, we are driven to choose the strong membership protocol for a much better reason. Seeing the same sequence of membership changes enables various nodes to reach the same consistent decision locally without any additional 2-phase communications. This is particularly useful, for example, when a primary-backup approach needs to be used, a very likely requirement when an RT P/S daemon is not running on every node. When the primary for a particular service fails, all nodes can automatically fall back to the same node as the backup. Similarly, when a backup fails, all nodes can automatically choose an identical node as a new backup using a consistent algorithm.

Finally, as can be seen in Figure 2-1, the RT P/S daemon interface to the membership information is transparent to the actual membership protocol in use by the heartbeat manager. This also allows us to experiment with various protocols in the future.

4.2. Pro-Active and Reactive Threads of Control

The notion of a *pro-active thread* and a *reactive thread* is central to our management of *weak* consistency of the RT P/S data. A pro-active thread is the source of changes while a re-active thread only responds to requests for changes from other sources (including pro-active threads). At a lower level of the hierarchy is the processor membership service offered by the heartbeat manager thread. This is a pro-active thread since it can trigger some changes to occur within the RT P/S system. However, it does *not* affect any RT P/S information directly. It can affect the information only indirectly through sending any processor membership changes to the local manager threads. All actual actions on the tag information are triggered by the local manager thread on all daemons since they are the recipients of creation/deletion/update requests from the client applications. By keeping this pro-active thread as the starting point of all data changes is key to our data management techniques within RT P/S. This results in complete "localization" of change requests at each node.

The two other threads in the daemon are *reactive* in nature. First, the update manager thread only reacts to updates from a remote local manager thread. If the local manager threads do not send it any requests, they cannot affect the RT P/S data. Secondly, the delivery manager thread reacts to publications from a remote delivery service and does not actually form part of the RT P/S information maintenance infrastructure. It is always a passive reader and never a writer.

4.3. State Transfer to Joining Node(s)

When a newly joined node comes up, it must obtain the latest RT P/S information from other nodes so that it can then become a legitimate daemon providing correct services to its clients. Since creation, deletion and update requests on tags can be issued at any given time, inconsistencies can occur. The consistent and complete transfer of state to a new daemon can be accomplished using a *two-phase commit protocol* that is designed to work in conjunction with (a) the processor membership protocol of Section 3 which allows multiple nodes to die or rejoin the system simultaneously, and (b) the local manager and update manager threads of the RT P/S daemons. In the worst case, it is possible that a rejoining daemon has to compete with other joining daemons and wait too long for its state transfer request to be honored. In this case, the daemon just aborts and retries its request after a while.

The two-phase commit protocol is executed by the local manager thread at each RT P/S daemon. A newly joining daemon first obtains membership information from its heartbeat manager and contacts the leader of the group with a "state transfer request". On receiving the request, the leader runs the two-phase commit protocol. If two or more join daemons issue join requests simultaneously, all but the first daemon may act as "dumb but willing" participants in the two-phase protocol.

4.4. Consistency of the State Transfer Operation

Theorem 4: If a daemon, which services creation, deletion and update requests only from its own node, dies during the processing of an update request, the RT P/S data maintained by the other nodes is weakly consistent.

Theorem 5: If a daemon dies during the 2-phase commit protocol, the RT P/S data maintained by the other daemons is consistent.

4.5. Principles Facilitating High Availability in the RT P/S Model

The RT P/S anonymous communications model exhibits some properties which are extremely conducive to implementing high availability and consistency of distributed information without compromising efficiency.

1. *Idempotence of RT P/S operations*: An operation is said to be "idempotent" if repeated execution of the operation achieves the same effect as a single execution of the operation. An example is the constant assignment operation "`x := 5`". In the RT P/S model, all update requests to add/delete publication/subscription rights are made idempotent. As a very beneficial result, constraints to enforce global ordering are minimal. In particular, if a particular request fails after updating a subset of the nodes' RT P/S, one just needs to redo the complete operation.
2. *Weak interleaving due to localization*: An update request allows for a process to request only rights for itself. This request must be channeled through the local manager thread of the daemon hosting that process's node. The interleaving of update requests from two different nodes does not lead to any consistency problems since they carry out mutually exclusive and idempotent operations and can be processed concurrently. The global ordering and resultant delays associated with strong consistency requirements are avoided.
3. *The benign impact of node failures*: Suppose that the update request of a process fails midway due to the failure of its node (or RT P/S daemon). In essence, some number of remote update manager threads would have received this update but not the others. As a result, some nodes will have outdated information with respect to others. This can generally be a serious problem in distributed systems. But in the RT P/S context, due to Lemma 3, the nodes with the updates will detect the failure of the node and clean up those superfluous entries anyway.

4.6. Tag Creation and Deletion

In our current RT P/S model, a tag can be deleted only by its creator. Also, two processes on different nodes can create a tag with the same name. Hence, strictly speaking, the creation operation on a tag is neither idempotent nor localized. A two-phase protocol among the daemons must be used to avoid possible race-conditions between two or more creation requests for the same tag so that only one succeeds and the others fail. Again, only startup delays occur and are generally acceptable.

4.7. Dealing with Processor Failures

Each local manager thread registers with its heartbeat manager thread to receive any membership changes. When a processor dies, each local manager thread receives the notification of processor death and subsequently notifies its update manager thread. The latter, in turn, cleans up all local publishers/subscribers from that node and sends a message to the notification thread of any clients impacted by the processor failure[1]. Due to the use of a strong membership protocol, all cleanups will be updated in similar sequence[2]. The cleanup operation of the tag table by the update manager thread on a processor failure is not in the critical real-time path. However, any updates to the tag table in a client need to be atomic and a processor death may trigger many tags to be updated. Any locks on the tag table held for a long time can indeed affect a real-time publisher or subscriber thread in the client by possibly introducing long durations of blocking/preemption. Currently, on processor death, we lock the local tag table in a client and update all entries in one single swoop. In reality, since we are constrained by the number of sockets and the number of message queues, this is not a major problem.

4.8. Dealing with Processor Rejoins

When a processor rejoins the system and has RT P/S tag state information transferred successfully, the only change in the other daemons is that they have added this processor to their list of update manager threads that must notified on requests that change tag state.

4.9. Getting Membership Information

Application threads can register with a heartbeat manager thread (on the same or different processor) to be notified of any changes in membership information (processor deaths or joins). In addition, an on-demand blocking call is available to obtain the latest stable membership information.

4.10. General Properties

It is important to note that despite the membership protocol, the state transfers and the various asynchronous notifications going on in the system, the real-time threads acting as publishers and subscribers have near instant access to the publisher/subscriber information that they need at all times (i.e., real-time processing and communications are not affected by the changes for high availability support). Deadlocks are not possible under any conditions and (weak) consistency is maintained.

4.11. Schedulability Analysis

Three factors are important in analyzing the impact of this infrastructure on schedulability. First, each node in the system can become the leader in our processor membership protocol. The group leader is the point of contact for a new node trying to join the system and obtain a consistent copy of the RT P/S tag table information. As a result, when the schedulability analysis of any node is carried out, its task set must include the leadership responsibilities both at the membership layer and the RT P/S local manager layer. Secondly, if multiple nodes join the system at the same time, it is possible that undesirable timing effects can happen

---

[1] An alternative would be to have each notification thread within a client to register itself as well

[2] Alternatively, the use of a weak membership protocol would eventually converge to the same information as well. Since processor deaths are not really a problem from Lemma 3, this situation does not in itself argue for a strong membership protocol.

at the leader node. One way of dealing with this problem is to process at most one join request within a specific time interval. Finally, the priority assignment of the various threads in the RT P/S daemon and the client library is also critical. In [12], the delivery manager thread had the highest priority within the daemon followed by the update manager thread and finally the local manager thread. The update thread in the client library was run at a higher priority than all client threads in the process using the RT P/S services. The heartbeat manager must be assigned a priority such that it gets to execute at least every $T_{\text{heartbeat}}$ but its deadline must be much shorter so that the heartbeat messages can be sent out, and responses received from other nodes.

5. Concluding Remarks

An interprocess communications model for highly available distributed real-time systems must be able to function with processor failures and allow repaired processors to be brought back into the system. In this paper, we have studied this problem using the real-time publisher/subscriber communications model. We present a 2-layered approach. A processor membership protocol with strong membership and timing properties is used by a higher daemon layer to maintain the communications infrastructure in a weakly consistent fashion. This structure allows processor failures to be tolerated and for new processors to rejoin the RT P/S system dynamically. We have also been primarily motivated by the need to maintain high performance in the steady-state sending/receiving path of real-time messages. This can be done by trading off against stronger consistency requirements which are mostly unnecessary in real-time and multimedia systems.

We have implemented the membership protocol, the two-phase state transfer protocol and the registration sequence of a joining processor successfully in the context of a real-time POSIX testbed. Students in a *Distributed Systems* course at CMU have also successfully implemented many different versions of these protocols to disseminate different video streams to multiple receivers using the RT P/S model. These implementations have been carried out on various platforms (including DEC Unix, HP-UX, Solaris and Windows NT) using various languages (including Java, C++ and C) and a variety of GUIs (including Motif, Netscape and Windows 95). Processors are able to fail and rejoin the system without impacting the efficiency of the critical path of video publication and reception. Preliminary benchmarks on fast workstations show that during publication, the network saturates before the processor, strongly indicating that the publication paths are very streamlined.

References

1. T. Abdelzaher, A. Shaikh, F. Jahanian and K. G. Shin. "RTCAST: Lightweight Multicast for Real-Time Process Groups". *Best Student Paper Award Winner, The Second IEEE Real-time Technology and Applications Symposium* (June 1996).

2. A. Bhide and E. N. Elnozah and S. P. Morgan. "A Highly Available Network Server". *USENIX* (1991), 199-205.

3. K. P. Birman and T. A. Joseph. "Reliable Communication in the Presence of Failures". *ACM Transactions on Computer Systems 5*, 1 (February 1987), 47-76.

4. F. Cristian. Agreeing on who is present and who is absent in a synchronous distributed system. Proceedings $18^{th
$ Fault-Tolerant Computing Symposium, IEEE, 1988, pp. 206-211.}

5. F. Cristian and R. Dancey. Fault-Tolerance in the Advanced Automation System. Tech. Rept. RJ7424, IBM Research Laboratory, San Jose, April, 1990.

6. F. Cristian. "Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems". *Distributed Computing 1*, 4 (1991), 175-187.

7. Gagliardi, M., Rajkumar, R., and Sha, L. "Designing for Evolvability: Building Blocks for Evolvable Real-Time Systems". *The Second IEEE Real-time Technology and Applications Symposium* (June 1996).

8. Ken Birman and Keith Marzullo. "The ISIS Distributed Programming Toolkit and The Meta Distributed Operating System.". *SunTechnology 2*, 1 (1989), .

9. Jahanian, F., Rajkumar, R. and Fakhouri, S. "Processor Group Membership Protocols: Specification, Design and Implementation". *Proceedings of the IEEE Symposium on Reliable and Distributed Systems* (October 1993).

10. S. Mishra and L. L. Peterson and R. D. Schlichting. A Membership Protocol based on Partial Order. Proceedings of International Working Conference on Dependable Computing for Critical Applications, February, 1991.

11. Oki, B., Pfluegl, M., Siegel, A. and Skeen, D. "The Information Bus - An Architecture for Extensible Distributed Systems". *ACM Symposium on Operating System Principles* (1993).

12. Rajkumar, R., Gagliardi, M. and Sha, L. "The Real-Time Publisher/Subscriber Communication for Inter-Process Communication in Distributed Real-Time Systems". *The First IEEE Real-time Technology and Applications Symposium* (May 1995).

13. Students of 15-612 Distributed Systems, R. Rajkumar, Instructor. "The Multimedia Information eXchange (MIX) System Project Reports". *Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA* (Spring 1996).

14. R. Rajkumar and M. J. Gagliardi. High Availability Considerations in The Real-Time Publisher/Subscriber Inter-Process Communication Model. Software Engineering Institute, Carnegie Mellon University, July, 1996.

15. Sha, L., Rajkumar, R. and Gagliardi, M. "The Simplex Architecture:An Approach to Build Evolving Industrial Computing Systems". *The Proceedings of The ISSAT Conference on Reliability* (1994).

16. N. P. Kronenberg and H. M. Levy and W. D. Strecker and R. J. Merewood. "The VAXCluster Concept: An Overview of a Distributed System". *Digital Technical Journal* (September 1987).

Table of Contents

List of Figures