Experiences with Group Communication Middleware *

Scott Johnson and Farnam Jahanian
Dept. of Electrical Engineering and Computer Science
University of Michigan
1301 Beal Ave.
Ann Arbor, MI 48109-2122
{scottdj,farnam}@eecs.umich.edu

Sunondo Ghosh, Brian Vanvoorst,
and Nicholas Weininger
Honeywell Technology Center
3600 Technology Dr.
Minneapolis, MN 55418

Abstract

Group communication is a widely studied paradigm for building fault-tolerant distributed systems. The Armada project at the University of Michigan is a collaborative effort with the Honeywell Technology Center to study how real-world applications use group communication. In this paper, we describe the results of our experience implementing a fault-tolerant distributed radar tracking system, and discuss how we were able to simplify our design and implementation by utilizing additional services built on top of the group communication model.

Keywords: group communication, distributed systems, middleware, fault-tolerant communication

1. Introduction

For the past two years, the Armada project at the University of Michigan has been collaborating with the Honeywell Technology Center on a study of middleware support for fault-tolerant distributed applications. This experience taught us a great deal about how applications use group communication to implement their fault-tolerance and performance requirements. This paper presents the results of our experience, and describes how we were able to simplify our design and implementation by utilizing a service library containing additional abstractions built on top of the group communication model.

The focus of this paper is not to describe a new group communication service or the potential semantic pitfalls associated with the use of a particular protocol. A large body of published research has already provided the community with many intricate protocols, and has explored the subtle ordering and reliability semantics associated with them. But researchers have also observed that group communication alone is too low-level, and additional services, such as state transfer, may be needed to facilitate the design and implementation of fault-tolerant applications. Our experience confirmed this, as we found that significant functionality had to be added to our application in order to realize its fault-tolerance and functional requirements using basic group communication primitives. We believe that these services are as important as the communication semantics to the distributed application, and must be designed with the same depth and thoroughness. To this end, we collected a variety of services from existing protocols and combined them with additional abstractions of our own design to meet our application's requirements more efficiently.

These abstractions include a process management service, which enables the application to organize processes into specific roles and easily replace failed processes; a set of synchronization and communication primitives, which can simplify complicated message exchanges and synchronize processes at well-known execution points; a failure notification service, which provides alternate mechanisms for notifying the application of process failures; and a group composition service, which enables multiple process groups to be composed together to construct complex, large-scale systems. Although these abstractions were realized in the context of our motivating application, we have designed them based also on experience with other fault-tolerant distributed systems, including interaction with the Naval Surface Warfare Center on the Hiper-D project, and participation in DARPA workgroups on dependability. We therefore believe that they will prove useful in simplifying the design and construction not just of our motivating application, but of fault-tolerant distributed applications in general.

In the following section, we present an overview of the radar tracking application. In section 3 we describe the abstractions in detail, and discuss the motivations behind their use. Section 4 summarizes our results.

*This work is supported in part by a research grant from the Defense Advanced Research Projects Agency, monitored by the U.S. Air Force Rome Laboratory under Grant F30602-95-1-0044.

0-7695-0707-7/00 $10.00 © 2000 IEEE

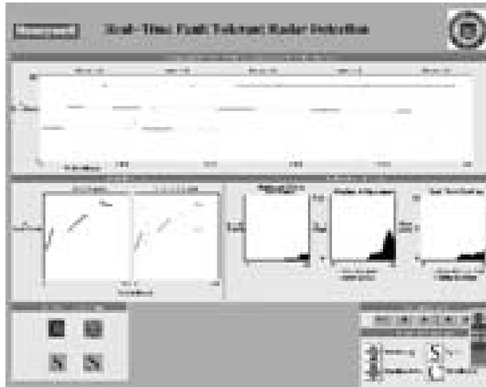


Figure 1. Screenshot of the Hypothesis Testing Application, showing three active tracks and the five most recent frames of radar data.

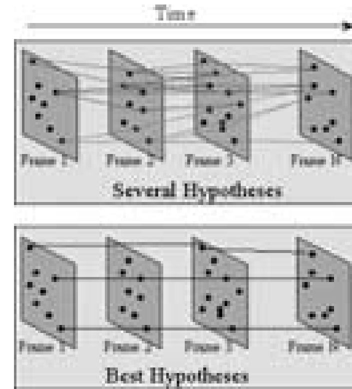


Figure 2. Demonstration of Hypothesis Testing. Originally, there are many possible hypotheses. After scoring and pruning, only the most likely trajectories remain.

2. Application Overview

The application chosen for this study implements a hypothesis testing algorithm, and is part of a larger radar tracking system[9, 8]. Many applications which make statistical predictions based on real-time data use hypothesis testing, including speech recognition algorithms and image understanding algorithms such as medical imaging. The application was implemented using RTCAST, a group communication protocol developed at the University of Michigan. RTCAST provides atomic, totally ordered multicast communication with soft real-time guarantees, and has been ported to Windows NT, Solaris, and Linux. Due to space limitations, we do not describe RTCAST in detail here, but more information is available in [1]. We believe that this method of refining and extending middleware services by implementing a suitable motivating application can be a valuable part of the middleware development process. In addition to the service library we developed, we were able to refine both the interface and the services provided by RTCAST to a degree that would not have been otherwise possible.

Although we selected RTCAST for this implementation, there are a number of group communication protocols which provide similar services and which could also benefit from our experience. Other real-time protocols include TTP [15], and XPA [18]. Non-real-time group communication protocols include ISIS [7], Consul [17], Delta-4 [18], Spinglass [6], and Transis [11].

2.1. Basic Hypothesis Testing Operation

The Hypothesis Testing application takes as input consecutive frames of radar data. Each frame contains both real radar returns and noise. The application then creates hypotheses about which radar returns correspond to real objects, and tracks the trajectories of those objects over time (Figure 1). These hypotheses are then deterministically divided among the available processes for evaluation.

Due to the large number of ways there are to "connect the dots," state explosion can be a significant problem. To mitigate this, each hypothesis is given a score indicating the likelihood that it is correct. After generating and scoring the hypotheses for a radar frame, a group coordinator selects those most likely to be true and the rest are discarded, as shown in Figure 2. At this point, the remaining hypotheses may be redistributed among the available processes to balance the system load before the next frame arrives.

2.2. Fault Tolerance

The Hypothesis Testing application is designed to be run in real-time, mission-critical situations. Because of these requirements, it is imperative that the application be tolerant to process and hardware failures, and be able to recover from those failures while still meeting its deadlines. The primary deadline requirement of each iteration is dictated by the radar frame arrival rate, which is approximately one second. Therefore, if a process fails during the execution of the algorithm, the application must be able to recover the work performed by the failed process up to that point, or else redo it before the deadline. We considered several replication schemes to meet these requirements:

Overlapping Active Replication: Processes are divided into G sub-groups of equal size, and hypotheses are divided equally among all G sub-groups (Figure 3). Every process in a given sub-group maintains the entire hypothesis set for that sub-group. Work is divided round-robin within each sub-group so that no work is duplicated. If a process fails before the current iteration is completed, its work is lost. To recover, the remaining processes in its sub-group first finish their own portion of the hypothesis space. As they finish, they begin working on the failed processor's

0-7695-0707-7/00 $10.00 © 2000 IEEE

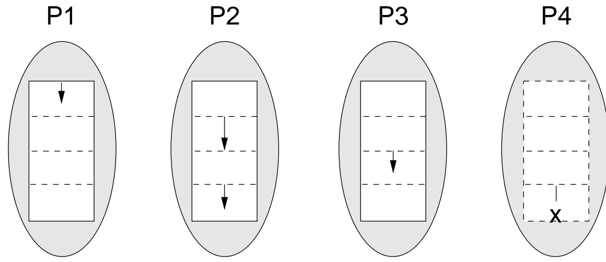


Figure 3. A sub-group performing overlapping active replication. Each process is assigned one quarter of the hypotheses. At this point, process four has failed before finishing. Since process two is done, it has begun working on process four's hypotheses.

hypotheses, starting from separate corners of the hypothesis space, until all hypotheses have been processed. Thus, each sub-group of size S can tolerate up to $S-1$ failures, depending on available processing time, with no loss of data.

Best K Replication: This scheme is similar to overlapping active replication, but only the top $K\%$ of hypotheses are replicated. Although this does not provide complete recovery, recall that all but the most likely hypotheses are discarded at the end of each round. By carefully choosing K, the application can ensure that hypotheses lost due to failures would likely have been discarded anyway.

Pairwise Active Replication: In this scheme, the processes are organized into non-overlapping pairs. Both processes in each pair analyze the same data on each iteration. If one of the processes in a pair fails no results are lost, since both are doing the same work in parallel. The failed process can then be replaced for the next iteration. One drawback to this approach is that there can be only one failure per pair of processes. However, it is a very simple approach, and is the method used in the final implementation.

3. Higher-level Abstractions and Services

Now that we have described the requirements of hypothesis testing, we discuss the limitations we encountered when trying to implement it using the process group model, and describe the abstractions we propose to address them. These abstractions include some which have been used in other systems, as well as a number of new abstractions we have designed based on our experience.

An important consideration for the abstractions is their integration with the group communication middleware. We have implemented them as a service library which sits on top of RTCAST. It provides a separate application interface, and is implemented as much as possible using services already provided by the underlying middleware. This should enable the abstractions to be ported more easily to other group communication protocols, since only the part of the library that interfaces to the lower-level middleware would need to be modified. We have tried to use only basic group communication services to maximize portability.

3.1. Application-level Process Management

One of the first things we realized when we began implementing hypothesis testing was that we needed more fine-grained control over the organization of processes within the group than was provided by the group communication model. There were a number of application requirements motivating this: we needed to designate process pairs for active replication, we needed to be able to divide incoming radar data among available processes, we needed to elect a group coordinator, and we needed to replace a process in one of these specific roles if a failure occurred.

Our library supports these tasks through an *Application-level Process Management* service, which allows the application to organize the group at a fine-grained level. With this service, applications can dynamically assign processes to specific roles, and replace failed processes without altering the organization of the group. This service is composed of three related abstractions:

Globally Consistent Process IDs: For an application to be able to assign different roles to individual processes, it must be able to identify those processes in a consistent manner at every group member. Many group communication services use some type of logical addressing to distinguish member processes, but in many cases these logical ids are assigned by the middleware and are either not made available to the application at all, or are made available on a read-only basis. This is sufficient for simple tasks such as selecting a group leader, but is too inflexible to use for more complex role assignments, such as the replication pairs in hypothesis testing, since process identities may change after failures and new processes may be given identifiers that do not match the existing group organization.

To address these limitations, we designed a new process id abstraction, which allows the application to assign a logical id to each group member. Group communication is used to propagate updates and ensure that members have a consistent view of the id assignments. If the middleware enforces total ordering, globally consistent process ids can be guaranteed even if multiple processes set ids simultaneously. If not, a globally consistent view can still be guaranteed if only one process (such as a group leader or replication partner) is responsible for setting the id of each process.

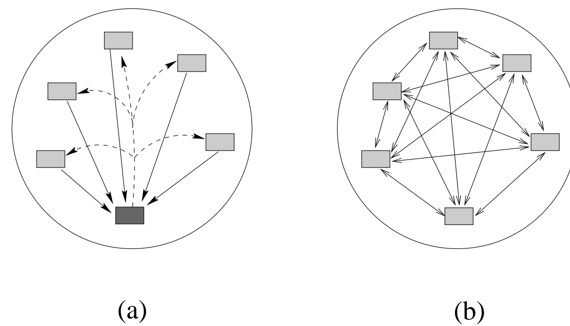
Free Node Management: Resource management has been integrated with group communication in other systems, exemplified by AQuA [10], which manages the availability of communication resources to provide quality of service with Ensemble [13]. Our experience showed us that to reduce the complexity of resource management, the pro-

0-7695-0707-7/00 $10.00 © 2000 IEEE

cess management service must also support free node management. This service is implemented as an interface to an external resource manager, which is responsible for keeping track of which processors are available in the system, and for implementing a policy to select which processor to use when a new process is requested. This policy can use any metric that meets application requirements, such as CPU utilization or network latency. When the application requests a new process, the service library queries the resource manager for a processor which meets the desired policy. It then creates a new process on that processor using an execution string provided by the application. We have implemented a basic resource manager for the hypothesis testing application, which selects the available processor with the lowest CPU utilization. It would be interesting to explore how this service could be integrated with AQuA's resource management framework.

State Transfer: The final component of application-level process management is a state transfer service. This type of service has been included in other group communication protocols, for example ISIS [7]. It allows the application to specify a buffer which contains state information needed by new members. When a process joins the group, the current contents of this buffer are copied to the new member as part of the join operation. The current mapping of application-assigned process ids can be transferred at the same time, ensuring that new processes will have a current view of the group's application-level organization when they join. This information is guaranteed to be delivered before the process receives any other messages, ensuring that it will be in a consistent and well-known state before processing data or generating new results. This removes most of the burden of state transfer from the application programmer, and provides a much cleaner and simpler semantic for fault-tolerant applications to use when implementing failure recovery. This service can be implemented using a lower-level state transfer service if available, or by intercepting and halting message transmissions to the new member until it has received the state update message.

To illustrate the utility of application-level process management, we now show how it simplified the design and implementation of our application. In hypothesis testing, one application process is initially responsible for creating the group and starting other processes, which is done using the free node management service. This process then assigns each of the N processes an id from 0 to $N-1$. Process 0 becomes the group coordinator. Process pairs for the pairwise replication scheme are chosen based on the process id modulo 2 (processes 0 and 1 are the first pair, 2 and 3 are the second pair, etc.). Work is divided round robin, such that the first pair gets the first $\frac{2}{N}$ of the data, the next higher pair gets the next $\frac{2}{N}$, etc. Since there is a global view of the process ids, all of these decisions can be made by each process independently, without explicit communication. If a process fails, its partner uses the free node management service to start a new process, assign it the failed process' identifier, and transfer hypothesis state to it using the state transfer service. Once again, this can be done without explicit coordination among the active processes.



(a) (b)

Figure 4. High-level Communication Primitives: (a) Query-Reply (b) Scatter-Gather.

3.2. Synchronization and Communication Primitives

Group communication middleware typically provides a group multicast service, which enables a process to send messages to some or all of the group members with guarantees such as atomicity, reliability, and delivery order with respect to other messages. We have observed that many distributed applications frequently have to perform more complicated exchanges, for example to share results, report information to a group coordinator, or synchronize processes at various execution stages. For example, in hypothesis testing, we needed to enable the group coordinator to query the group for the best hypotheses, allow group members to exchange workload data for load balancing, and ensure that all processes were ready before continuing with the next iteration. Although these forms of communication can be implemented using the group multicast primitive, they often require extra work on the part of the application designer.

In our hypothesis testing implementation, we found that a set of synchronization and communication primitives would simplify the application design. These primitives build on the basic multicast service to provide more robust functionality as a single operation, and have been included in other communication systems such as MPI [16], but are not found in many group communication services that we are aware of. These primitives can be implemented using only a reliable multicast. They include:

Barrier Synchronization: When initializing the application, we needed to ensure that all processes were ready before it could begin accepting radar data. To achieve this, we implemented a barrier synchronization primitive. When executed, it blocks the process until all other processes have

0-7695-0707-7/00 $10.00 © 2000 IEEE

reached the same execution point.

Distributed Lock Management: We have also designed a distributed lock management service which implements traditional OS mutual exclusion primitives, such as semaphores and mutexes, between group members, even if they are running on different hosts. One process is responsible for holding each semaphore or mutex, and operations on that object are conducted through reliable multicasts. If the group communication service provides total delivery, then a central coordinator for each object is not necessary, as each member can maintain a local copy and all members will see the same sequence of updates to the object. Although it is certainly possible for an application to implement this functionality, providing it as an abstraction further reduces implementation effort. In addition to serializability, this abstraction also provides a simpler method for atomically conducting operations requiring multiple messages.

Query-Reply: The query-reply primitive allows a process to send a query to the group and collect a response from each group member as a single operation (Figure 4a). In the hypothesis testing application, the group coordinator needs to repeatedly query the members of the group about their hypothesis scores. After each query, the coordinator collects a reply from each member and decides which hypotheses to keep and which to discard. Our experience suggests that this type of operation is fairly common in other fault-tolerant distributed applications as well.

Scatter-Gather: This primitive enables each process to report some information to the group, and collect the information sent by other members as a single operation (Figure 4b. This is similar to the query-reply, but in this case every process receives a copy of the collected responses. We used this to implement replication in hypothesis testing, where Processes need to send updates to their replication partners about their current position in the workload, and to collect the same information from their partners in return. We have observed this form of communication in other applications as well. For example, it can be used for load balancing, where each process could send the size of its current workload to the rest of the group, and collect the same information to use in deciding where to send excess work.

3.3. Failure Notification

For many fault-tolerant applications, failure detection and notification is a key service provided by the middleware. In our experience, some applications use group communication just for failure detection, and do not even need the multicast service. Traditionally, most group communication protocols have focused on the semantics of failure notification. Although this is important, we found that it is also important to consider the mechanism by which failure notifications are delivered to the application. For example, RTCAST originally only delivered failure notifications in order with respect to data messages. There may be times when applications can not afford to wait until messages preceding a failure have been delivered. This is true in hypothesis testing with overlapping active replication, since there is no communication during data processing and replicas need to be notified immediately so they can begin recovery.

We have designed a failure notification service which provides several different notification methods that can be used interchangeably by the application, depending on its dynamic failure reporting requirements. Although these mechanisms have been used in other group communication protocols, our experience suggests that it is useful to provide an interface which allows the application to dynamically select its notification methods. However, based on our experience implementing these changes in RTCAST, we believe that the modifications required are minor and relatively easy to make. These methods include:

In-band: Failure notifications are inserted into the message stream, and are delivered to the application like a regular data message. The advantage of this method is that failure notifications can be delivered with the same ordering semantics as data messages. We used this method when selecting the best hypotheses during hypothesis testing.

Callbacks: Some applications may want receive notifications immediately instead of waiting until preceding messages have been delivered. With callbacks, the application provides a function which is called by the service library when a failure occurs. We used callbacks in hypothesis testing to enable replication partners to begin recovery quickly.

Polling: With polling, the application explicitly checks for failures by calling a special function which returns information about the next unreported failure. This is more efficient when the application does not care about immediate failure notification. This method is used during the initialization of hypothesis testing, since any failed processes can be replaced after initialization with no loss of data.

3.4. Group Composition

As mentioned in section 2, hypothesis testing is one component of a larger radar tracking system. This system has a number of components, each of which has its own fault-tolerance and performance requirements. Most of these components could benefit from group communication, especially if it was extended with the services we have described. Unfortunately, there are a number of problems that arise if the system is implemented using a single process group. First, it would be very difficult to manage such a large number of processes working simultaneously on many unrelated tasks. Any failures would be reported to processes in all components, even ones which wouldn't be affected by the failure. In addition, messages for each component would be delivered to all processes in the system, resulting in increased overhead and bandwidth contention. Note that

0-7695-0707-7/00 $10.00 © 2000 IEEE

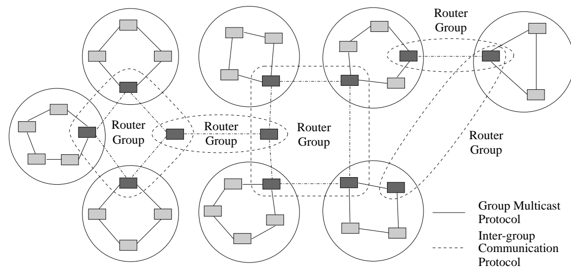


Figure 5. Diagram of the group composition framework, showing several process groups and the inter-group routers used to connect them.

these problems are not specific to the radar tracking system, but have been observed in other large-scale distributed applications and group communication services [1, 4, 3].

To support a more efficient design, we have created a framework[14] which supports the modular composition of process groups (Figure 5). This framework allows composed groups to exchange messages while enforcing end-to-end delivery semantics. Using this method, we were able simplify the design of hypothesis testing's overlapping active replication scheme, by placing each replication subgroup in its own process group.

Other protocols have been proposed which support multiple process groups. These include the Totem multi-ring protocol [2], the Causal daisy architecture [5], and fault-tolerant total order multicast to asynchronous groups [12]. A detailed comparison of these approaches with our group composition framework is available in [14].

4. Conclusion

This paper described our experience with the implementation of a fault-tolerant radar tracking application. It described how we were able to simplify our design and implementation by utilizing a service library containing additional abstractions built on top of the group communication model. We believe that this comprehensive approach to providing higher-level services will make it possible for applications to take full advantage of the group communication paradigm.

References

[1] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. RT-CAST: Lightweight multicast for real-time process groups. In *Proceedings IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 250–259, June 1996. Additional information is available in a techical report.

[2] D. Agarwal, L. Moser, P. Melliar-Smith, and R. Budhia. A reliable ordered delivery protocol for interconnected local-area networks. In *International Conference on Networking Protocols*, 1995.

[3] Ö. Babaoğlu, A. Bartoli, and G. Dini. Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. *IEEE Transactions on Computers*, 46(6):642–658, June 1997.

[4] Ö. Babaoğlu and A. Schiper. On group communication in large-scale distributed systems. *ACM SIGOPS Operating Systems Review*, 29(1):612–621, Jan. 1995. Also appears as Proceedings ACM SIGOPS European Workshop, September, 1994.

[5] R. Baldoni, R. Friedman, and R. Renesse. Hierarchical daisy architecture for causal delivery. Technical report, Cornell University, 1996.

[6] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2), May 1999.

[7] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, Aug. 1991.

[8] H. T. Center. Application level benchmark results. Technical Report C0011, Rome Laboratory, New York, Aug. 1998. Contract number F30602-94-C-0084.

[9] H. T. Center. Application level benchmark specifications. Technical Report C0010, Rome Laboratory, New York, Aug. 1998. Contract number F30602-94-C-0084.

[10] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. Sander, D. Bakken, M. Berman, D. Karr, and R. Schantz. Aqua: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS 98)*, pages 245–253, Oct. 1996.

[11] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, Apr. 1996.

[12] U. Fritzke, Jr., P. Ingels, A. Mostefaoui, and M. Raynal. Fault-tolerant total order multicast to asynchronous groups. Technical report, Centre National de la Recherche Scientifique, Jan. 1998. To appear in SRDS '98.

[13] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.

[14] S. Johnson, F. Jahanian, and J. Shah. The inter-group router approach to scalable group composition. In *Proceedings 19th IEEE International Conference on Distributed Computing Systems*, pages 2–13, June 1999.

[15] H. Kopetz and G. Grünsteidl. TTP – a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, Jan. 1994.

[16] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1.1 edition, June 1995. Available at http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html/.

[17] S. Mishra, L. Peterson, and R. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1:87–103, 1993.

[18] P. Verissimo, P. Bond, A. Hilborne, L. Rodrigues, and D. Seaton. The extra performance architecture (xpa). In D. Powell, editor, *Delta-4 - A Generic Architecture for Dependable Dist. Computing*. Springer-Verlag, 1991.

0-7695-0707-7/00 $10.00 © 2000 IEEE