To appear in the Proceedings of the *First IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, Kyoto, Japan, April 20-22, 1998.

Building Middleware for Real-Time Dependable Distributed Services

Franco Travostino, Laura Feeney, Philippe Bernadat
The Open Group Research Institute
11 Cambridge Center
Cambridge, MA 02142, USA
{f.travostino, l.feeney, p.bernadat}@opengroup.org

Franklin Reynolds
Nokia Research Center
3 Burlington Woods Drive - Suite 260
Burlington, MA 01830
franklin.reynolds@research.nokia.com

Abstract[1]

We consider a real-time, distributed service to be dependable if it continues to have timely, predictable behavior even in the presence of partial failures. Services with this property are desirable in a host of real-time scenarios, including factory floor automation, medical monitoring equipment, and combat systems.

Most distributed services built with contemporary fault-tolerance toolkits are not dependable; they exhibit unpredictable, albeit logically correct, behavioral patterns under failure conditions.

We have designed and implemented middleware explicitly for real-time dependable services. We aimed at maintaining sub-second worst-case guarantees for failure detection and recovery, even when failures conspire with network load and CPU load to undermine determinism. The paper reports our experience in marrying software fault tolerance and real-time disciplines, from the definition of the requirements to the characterization of the resulting system.

1. Introduction

From futuristic avionics to our own homes, there is a growing demand for distributed services with real-time properties, such as predictability and responsiveness. As failures are intrinsic in distributed services, these real-time properties must be preserved despite partial failures such as host crashes and link failures (e.g., a distributed service that must react to an attack that has already inflicted damage). In this case, we describe the services as real-time dependable distributed services (RDDSs).

1. This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Rome Laboratory of the Air Force Materiel Command (AFMC).

Building an RDDS is a challenging proposition. Previous work [3][26] [21][16][17] in the area of fault management and fault tolerance has shown the practicality of investing in “middleware” layers; within these layers, complex failure scenarios are transformed into simpler failure semantics, which benefit applications by allowing them to concentrate on their context-sensitive reactions to failures. This same work has demonstrated that middleware layers where real-time is an afterthought fall short in providing real-time guarantees for RDDSs. The real-time problems detected in these layers [7] typically manifest themselves with lock-ups and ill-defined semantics to the application. It is often the case that throughput optimizations within these middleware layers adversely impact predictability under failure. Other problems can be traced to middleware layers that defeat the application context and the end-user argument (e.g., the middleware identifies and reacts upon false dependencies among messages).

The paper describes our experience in building middleware for RDDSs, from the definition of requirements to the characterization of the resulting software. Our approach differs from previous work on software fault management in several ways:

- We combine fault management and real-time disciplines within an object-based framework. The result is a family of solutions and trade-offs, rather than a point solution.
- Our techniques include programming with system resources [29] on a per communication channel basis, to avoid QoS-crosstalk and cascades of failures caused by resource shortages.
- We characterize our work under adverse circumstances, i.e., the ones where failures conspire with network load and CPU load to defeat determinism.

Our work explicitly aims at hundreds of milliseconds worst-case timings for failure detection and recovery. Our fundamental focus on real-time has driven the choice of al-

gorithms, APIs, implementation techniques, interactions with the host system, and tools for analyzing the behavior and real-time characteristics of the resulting combination of application, middleware, and host system. Our middleware exports a process group abstraction to RDDSs; we have dubbed it GIPC, for Group IPC.

It is appropriate to think of GIPC as delivering ISIS-like functionality, but with real-time predictability built in from the ground up.

We found additional challenges in tailoring the middleware to a wide variety of real-time environments, making it highly adaptive, and re-using it within environments where scalability requirements prevail over determinism and responsiveness.

We use GIPC to build RDDSs upon commodity hard-

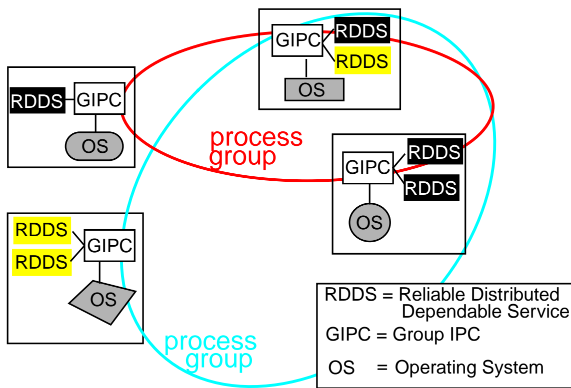


Figure 1: High level organization.

ware (e.g., PCs linked by Ethernet, FDDI, Myrinet) and to provide sub-second guarantees within local area networks. Typical deployment scenarios for our target RDDSs are: factory floors, medical monitoring equipment, and military command and control distributed applications.

This paper describes the features of GIPC. First we analyze our requirements; then we introduce the model adopted, the features of the inner protocols and layers, the functionality exposed to the applications, and the implementation choices. The last sections analyze empirical results from test applications; among these, we describe a dependable, distributed control application for a computer-controlled mechanical apparatus.

2. Middleware for real-time dependable distributed services: requirements

2.1. Functional requirements

Our target RDDSs must participate in fault management. While application-transparent fault tolerance facilities [24][25] are attractive because they reduce the burden on the application developer, our real-time emphasis requires that RDDSs be in control of the underlying "translucent" [30] middleware layers to the greatest extent possible. We believe that process groups are a powerful abstraction for RDDSs; the application interacts with the group through *join*, *leave*, *post*, and *receive* explicit operations. We complement these typical process group verbs with new verbs for negotiating QoS contracts with middleware, on behalf of an individual group participant or of the whole group. QoS contracts include: type of membership, delivery semantics, group priority among the other groups, traffic shape, etc.

The application must be able to perform administration of the group, and thus inject application-specific context at important times in the life of the group. Administrative duties include: enable/disable the join of new endpoints, shutdown the group, renegotiate a QoS contract due to external stimuli, re-admit endpoints partitioned away by a transient failure, etc.. Partition management best reflects the need for policies dictated by the application. In some cases, the policy may be limited to a simple majority decision (e.g., a farm of unspecialized machines crunching numbers); in other cases, the policy may well discriminate between the role and location of the endpoints partitioned (e.g., machines associated with particular sensors).

The corollary is that middleware for RDDSs must be as policy neutral as possible; policies flow from the application to middleware, and in no circumstance should they be overridden by middleware.

Our RDDSs typically use a small number of communication paradigms and delivery semantics. We identify two points of aggregation: one for powerful, costly semantics (e.g., "all or none" delivery) and one for limited, low-overhead semantics (e.g., unreliable multicast). We require that middleware supports at least FIFO Atomic Broadcasts[2] and raw IP multicast, respectively. For the FIFO Atomic Broadcast to be predictable under failure, we are prepared to sacrifice throughput. As we count on the RDDS participating in fault management, we believe that

2. A reliable broadcast is a group multicast that all correct group participants deliver exactly once. A FIFO Atomic Broadcast is a reliable broadcast with total ordering and FIFO ordering with respect to traffic originated from the same source.

the RDDS will limit itself to use this service sparingly, and refrain from the idea of a "catch-all" reliable network service implemented in middleware.

The notion of membership is crucial to a RDDS. Middleware must supply a RDDS membership views whose properties—liveness vs. accuracy—are traded off based upon explicit indication from the RDDS. Typical values of liveness granularity range from 100 milliseconds to few seconds. Aggregates of failures, either cascades of failures or malicious denial of service attacks, must not prevent the RDDS from obtaining a new, timely membership view.

2.2. Requirements on implementation

The implementation of middleware for our RDDSs must be highly modular and layered. We must be able to add and replace modules to meet RDDS scenarios wherein software configurations may vary in space (e.g., embedded agents versus general purpose servers) and in time (e.g., an aircraft control system during take-off, cruise, and landing). Layering complements modularity by adding constraints on how modules combine and interact. We observe that middleware structured according to modularity and layering will naturally map to its own dependency graph (i.e., which functions depend upon which functions); such a graph is instrumental in assessing the impact of failures within a node (e.g., no more input memory buffers). Finally, engineering inspired by modularity and layering allows an evolutionary path towards more real-time, fault management, and security functionality.

Object-oriented technology is an obvious way to pursue modularity and layering.

Individual modules interact with the host system by utilizing system resources (e.g., processing, memory, bandwidth, etc.). Each of the modules must expose its own "knobs-and-dials" to enable QoS selections by higher-level modules and/or by a system administrator. It must be possible for a module to use a powerful interface to system resources, including verbs to partition resources on a per-channel basis (e.g., to guarantee forward progress of selected functions while denial-of-service attacks are in progress). Much like the QoS selections, the various choices with respect to resources (e.g., guaranteed vs. overbooked memory buffers) have to come from the context-rich top layers.

2.3. Requirements on the host environment

For RDDSs to count on sub-second responsiveness to failures, we require that real-time behaviors be provided throughout all layers— middleware layers as well as system layers and RDDSs. In fact, it is well-known that properties such as real-time guarantees and resilience to denial-of-service attacks are bottom-up properties. We assume that any predictability anomaly observed in the system layers, and, ultimately, in the hardware, can only be amplified by the middleware[3], not corrected.

Ethernet introduces anomalies through the exponential back-off of CSMA/CD. We detected less obvious anomalies while characterizing our middleware over Myrinet: the on-board firmware for Myrinet defines its own liveness mechanisms and granularity (i.e., it maps the topology once a second). We will not be able to push a RDDS over Myrinet to worst-case timings of hundreds of milliseconds unless we establish tight cooperation between the liveness mechanisms in the middleware and the equivalent mechanisms in the Myrinet firmware. With the emerging popularity of intelligent network controllers and programmable switches, the notion of "bottom" in bottom-up properties is more elusive than ever.

The reference real-time platform that we use to experiment with GIPC and RDDSs consists of the MK operating system—a microkernel-based system derived from Mach [2] and enhanced with a host of real-time features, including preemptability, migrating threads [8], a framework for scheduling policies and communication protocols [28] — and FDDI, Myrinet, and Ethernet connectivity.

3. The model

In the world that we want to model, several types of failures may strike: fail-stop and crash failures, send and receive omission failures, ordering failures, and link data integrity errors. For GIPC to be sufficiently general, identifying the scenario with weak hardware properties and the largest enumeration of failures is crucial. This scenario need not be the ideal target scenario for deployment, nor need it have bottom-up real-time properties. A group of PCs connected by an asynchronous and lossy link such as Ethernet serves well as a "weakest scenario" testbed.

This design center is well understood [23] in the literature. Theory proves the equivalence between distributed consensus and atomic broadcast in asynchronous systems. As our requirements prompt us to provide both, the former for membership and the latter for group communication, we decide to implement atomic broadcast and to resolve distributed consensus problems with atomic broadcasts.

In an asynchronous environment, [16] states that even a single crash failure makes distributed consensus and atomic broadcast a problem impossible to solve. It turns out, however, that the asynchronous model in [16] does not apply directly to our design center. In fact, according to [16], we can insert an arbitrary delay between any two state

3. Some RDDSs on top of middleware may, however, be capable of adaptive algorithms which compensate for the anomalies.

transactions (i.e., an arbitrary delay between two messages from the network or an arbitrary delay between two instructions on the host). In real life, we know that each computing node is equipped with a highly reliable quartz crystal and has the notion of local time. Therefore, authors have circumvented the asynchronous model in [16] with the much friendlier timed asynchronous model. In this model, it is possible to introduce the notion of a timing (or performance) failure [11]: if we fail to reach a node within a prescribed interval, we begin to advertise a failure for the target node; we must be prepared, however, to deal with the consequences of "false" failures being detected. Erroneous decisions about failures may be reciprocated resulting in partitions of the group.

With a timed asynchronous model, we can solve the distributed consensus problem and we can perform atomic broadcasts. The membership that results is said to be live but not accurate: any failure is eventually detected (live), even though timeouts may trigger erroneous decisions (not accurate). This type of membership does meet our requirements (so long as valid group members share the same correct and erroneous decisions). The mechanism needed for fault detection—called a fault suspector—may be topology dependent.

We also require membership events to be totally ordered with respect to messages. Thus, we embrace the virtual synchrony model [3] with the specific goal of guaranteeing that the distance between virtual and real world is small and bounded.

4. The protocols

In this section we examine some fundamental design choices that shed light on the internal architecture of the GIPC middleware.

First, we distinguish between "physical" membership—the society of physical nodes—and "logical" membership—the society of endpoints exported via the process group abstraction (Figure 2). The fault suspector for physical membership is radically different from that for logical membership. While physical membership requires handshakes over the wire ("heartbeats"), logical membership does not and can be layered on top of physical membership, provided that physical membership can be configured to take the requirements of logical membership.

Second, we isolate the fault-suspector subsystem from the subsystem that provides reliability and ordering of data. Protocol choices may be different for these two subsystems to allow maximum efficiency for the given combination of hardware properties, failure types, and topology. For instance, while a neighbor surveillance protocol may be chosen for the fault suspector, a master/slave based algorithm

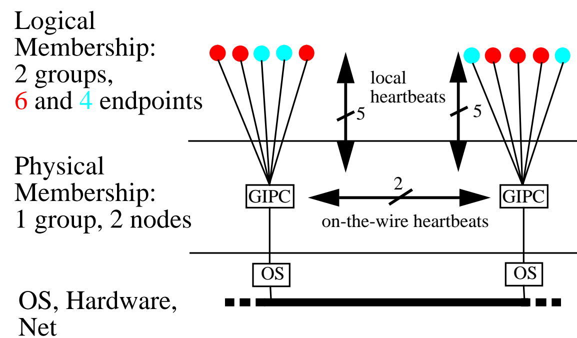


Figure 2: Sample cross-section showing logical and physical membership.

may the best choice for reliability and ordering. Protocols aside, application of different scheduling priorities and system resources to prioritize one subsystem versus the other (e.g., to make the fault suspector's signalling take place at higher priority than data traffic) can yield interesting empirical results. This decoupling comes at a price, however: traffic from one subsystem is much harder to piggyback on top of the traffic of another subsystem.

Third, we reflect the reduction of the consensus problems to atomic broadcast by identifying a layer, the reliability-ordering layer, which implements atomic broadcast and does not know whether it is manipulating membership views or application data. Membership views and application data are totally ordered as required by virtual synchrony.

Fourth, for the protocols that provide reliability and ordering, we must choose between a master/slave scheme and a rotating token scheme (although modularity won't preclude us from switching schemes or entertaining hybrid solutions such as [12]). The former is based on an "oracle" that supervises and legislates membership and ordering; in practice, slaves propose multicasts to the master, and the master actually performs the multicasts to the group via a two-phase commit. In the latter, control is uniformly spread among all participants, with a token that rotates through all sites and arbitrates group events. After some experimentation, we came to the conclusion that a master/slave scheme is better suited for real time and thus is what we implemented as default. Master/slave-based schemes allow higher concurrency (the master multicasts to the group while slaves unicast new multicast requests to the master) and inherently increase the responsiveness of process groups. Furthermore, the master is well positioned to understand relative merit among slaves and schedule the group activities accordingly; it is also able to apply group policies very quickly. With a master, most failures are very easy to deal with; on the other hand, the failure of the mas-

ter is more complex to address, as it requires an election process involving all slaves. In master/slave schemes, the number of slaves is limited by the occurrence of traffic implosion episodes at the master. With a token-based approach, the benefits and disadvantages tally differently. While the message volume is always under control, and there is no potential for implosion, the token represents a severe limit to concurrency. Group maintenance and repair are also much more complex due to the uniform distribution of state and responsibilities among members.

Fifth, protocol headers are fixed size and do not depend upon group population, since packets are addressed to the whole group and do not carry dependency lists or vector clocks[4]. Therefore RDDSs can assume a fixed overhead for headers and can carry their own speculations with respect to payload (e.g., will fragmentation occur or not) In case of multicast-capable hardware, the number of packets transmitted by any physical node is also independent from group population.

Special effort must be devoted to keeping the virtual synchrony model in close touch with reality. We model each physical node as having only one ***failure domain***; this domain covers the address spaces for the individual group participants, GIPC itself, and the system layers that GIPC depends upon. We further specify the chosen master/slave scheme as one that provides safe delivery to failure domains via positive acknowledgments. For a message to be considered stable, and thus be disseminated to the various group participants, the message must have successfully reached the failure domain of each physical node involved in a given group. Once a message has reached the failure domain of a physical node, we trust GIPC to provide a fault tolerant dissemination path to the participants registered with the node. This may be accomplished, for instance, with support of node-local resources such as stable storage. With this definition of failure domain, we only transmit the acknowledgments among failure domains rather than among group endpoints[5].

The pervasive use of positive acknowledgments, either explicit or piggybacked, allows us to limit the number of messages waiting to become stable, and thus provides an upper bound on recovery actions. A situation where the master waits for positive acknowledgments from all slaves implies that the master has a precise knowledge of the membership. This, however, is not problematic, as we must track and expose membership views to participants in any case. A master awaiting positive acknowledgments might insist on resending the same data to slaves, just because the acknowledgment from one of the slaves has not yet arrived. Our solution is to implement message operation in a preemptive manner; thus, the above situation would persist only until a new membership view (which is likely to evict the slaves that do not reply) is proposed. The master will then resume from the membership change by re-sending old, unacknowledged data that has still some value under the new membership view.

A well-known problem [9] with positive acknowledgments is poor scalability. Whenever scalability matters, we are prepared to relax positive acknowledgments by splitting the set of all failure domains into a subset that is expected to produce positive acknowledgments and a subset that is expected to produce negative acknowledgments. Should the master fail, the former subset will be the one used for electing a new master within a bounded period of time.

Timely detection of partitions is also crucial for allowing protocols based on the virtual synchrony model to track reality. There is a function dedicated to “snoop” partitioned members and pass this information along to a RDDS throughout the API. This protocol is not entitled to apply any particular policy; rather, its mission is to alert the RDDS to act upon a partition.

Several algorithms in the protocols for membership, reliability, and ordering are prone to livelock situations. For instance, a failure that manifests itself with a hardware flicker can induce the middleware to continuously bid new membership views, and thus prevent any other protocol from making forward progress. We discourage livelocks by inserting filters (“skeptics”) above any object that can induce livelocks into higher layers[6]. A skeptic artificially limits the rate of status changes observed in the underlying objects.

5. The API

The API legislates how a RDDS connects to GIPC. We have defined an API for process groups and implemented it for the MK, UNIX (e.g., HP-UX, OSF/1), and Windows/NT operating systems. We anticipate working with multiple API specifications and implementations. The former are prompted by the need for different abstractions, whereas the latter are prompted by dependencies on the IPC mechanisms that a particular platform avails. Thus the modularity and layering guidelines must apply to the API component as well.

In the default API, a RDDS interacts with a process group by joining or leaving the group, sending and receiving data, receiving membership information, and negotiat-

4. As is typical of causal ordering protocols, for instance.

5. In much the same way as we transmit heartbeats only among physical nodes and not among logical endpoints.

6. It is appropriate to think of these filters as low-pass hardware filters implemented in software,

ing QoS contracts with GIPC. Such contracts may affect the whole group (e.g., type of membership and delivery semantics), or the individual endpoint which performs the operation (e.g, advertise traffic shape). The individual portion of the QoS contract is particularly relevant for endpoint liveness: the endpoint advertises the rate at which it will be signalling to GIPC, and should endpoint signalling exceed that range, GIPC will call for a performance failure. It will unilaterally terminate the endpoint, and supply the updated membership view to the surviving members of the group.

The operation of joining a group is not subject to real-time constraints; it is guaranteed, however, that joins do not jeopardize real-time guarantees of endpoints already in the group. There is a synchronous version of the operation of posting to a group (i.e., post and wait until the message becomes stable) and an asynchronous version (i.e., post returns immediately and an upcall confirms the message's stability).

On our real-time reference platform, the API has been implemented to support multiple threads and to use a real-time capable IPC mechanism based on migrating threads [8].

6. Implementing GIPC within CORDS

6.1. The toolkit

GIPC is built using CORDS[7], our object-based communication framework and toolkit derived from the University of Arizona's x-kernel technology [22]. Researchers using the x-kernel and CORDS have shown the practicality of creating a library of composable, "off-the-shelf" protocol objects—be it standard protocols or application-specific protocols. The modules in the library can be composed in various protocol graphs such as the one in Figure 3, and later re-used in other contexts. CORDS is highly portable; platform dependencies are entirely resolved within the framework (i.e., no impact on the protocols in the library).

CORDS protocol objects derive from the same root class, and are specialized with protocol specific state and functions. Introspection among protocol objects allows probing for features of neighboring protocol objects (e.g., multicast capability, MTU size, etc.). In the current version of CORDS, the edge connections among protocols shown in Figure 3 are static and established at compile time.

CORDS provides early demultiplexing and management of system resources (i.e., processing, memory, bandwidth) on the basis of individual communication channels or "paths" [29] throughout the protocol graph. A path can be thought of as the host-internal representation of an end-to-end flow; a path, however, is not bound to the use of specific communication paradigms such as TCP connections or IPv6 flows. Through the path abstraction, a CORDS user isolates sub-sets of traffic, selects system resources, and make the connection between the two. Early demultiplexing operates on input traffic and aggressively classifies packets among paths with pre-assigned resources or default resources. Paths can be statically or dynamically configured.

Within the library of protocols available in CORDS[8], there are several packages that proved very useful while debugging, testing, and characterizing CORDS real-time set-ups like GIPC. First, the ORCHESTRA [13] fault injection package developed at the University of Michigan allows us to artificially create failure scenarios by filtering and sometimes altering input traffic. The protocols that compose this package can be seamlessly built into a GIPC protocol graph to exercise its protocol components, and to study their resilience to deadlocks or livelock situations. This package has greatly improved the robustness of our protocol implementations. Second, the LTS[9] clock synchronization package is a collection of protocols that provide an external clock synchronization service based on a probabilistic synchronization algorithm [10], and a clock amortization algorithm [27]. Unlike NTP, LTS guarantees an upper bound on the synchronization offset and defines failure semantics for those cases where the guarantees cannot be met. LTS proved useful in the off-line synchronization of events occurring at different nodes.

Protocols in CORDS can also take advantage of several utilities for exposing tunable parameters in a uniform fashion, for logging time-stamped events [6], and collecting statistics.

6.2. The GIPC protocol graph

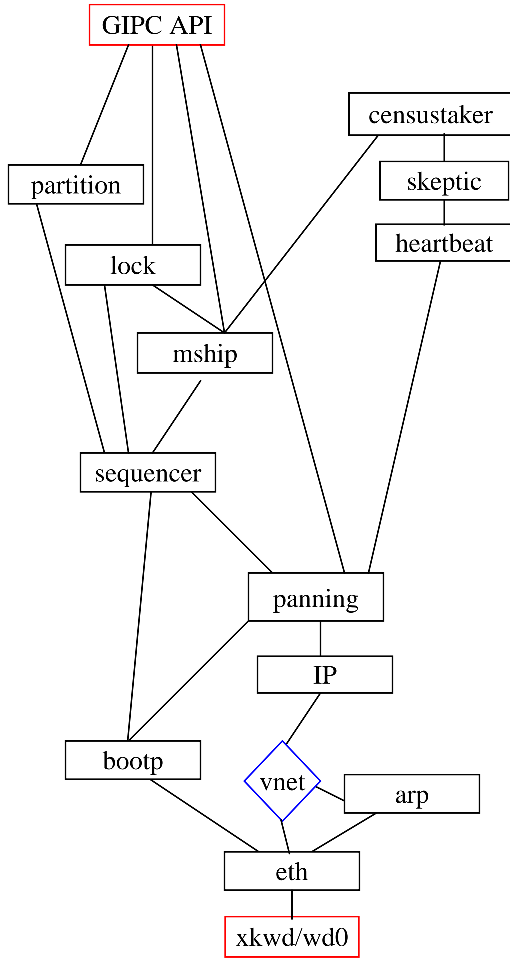
The flexibility of the CORDS framework greatly extends the reach of the GIPC implementation. We have anecdotal evidence that some of the following actions already helped ourselves in the evolutionary path towards real-time fault tolerance:

- protocol interposition (e.g., skeptic, fault injection)
- protocol replacement (e.g., sequencer vs. token)
- protocol addition (e.g., partition detector)
- protocol removal (e.g., heartbeat done in hardware)
- protocol reuse (e.g., panning protocol, IP multicast)

7. Communication Objects for Real-time Dependable Systems

8. About 60 protocols.

9. LTS' availability is limited to the MK system. Note that none of the protocols in GIPC depends upon LTS or any other distributed clock.



Protocol Name	Mission
GIPC API	provides the notion of multiple groups with multiple end-points per group per physical node. It also implements the endpoint liveness protocol and the verbs needed for QoS contract negotiation.
census-taker	maps the topology and promotes new membership views upon timeouts. When in a master/slave schema, it only runs on masters.
skeptic	implements a low-pass filter which limits status changes in underlying objects.
heartbeat	produces pseudo-isochronous signalling. When in a master/slave schema, it only run on slaves.
lock	abstracts a distributed lock.
partition	"snoops" partitions and forwards unreliable partition notifications to other subsystems that registered for this service.
mship	provides a database with the most recent membership view.
sequencer	realizes the reliability and ordering functions in a master/slave schema.
panning	filters out input packets that do not belong to a current membership view.
IP	Standard protocol (including multicast extensions)
BOOTP	Standard protocol
vnet	Handle multiple network adapters
ARP	Standard protocol
ETH	Standard protocol (also: FDDI, MYRINET)
xkwd/wd0	Bottom anchor (machine dependent)

Table 1: Protocol name and mission

Figure 3: The protocol objects constituent of GIPC and their organization.

Implementing the protocol graph described in Figure 3 was challenging, due to the rich interactions among protocols. The main differences between a protocol graph like GIPC and a linear one, such as UDP over IP over Ethernet, are co-dependency among protocols and co-dissemination of data and control. In GIPC, the membership protocol and the sequencer protocol are co-dependent. For some operations, one would like to layer the sequencer protocol above the membership protocol (i.e., the sequencer depends upon membership information to terminate an atomic broadcast—how many acks should I wait for); on the other hand, membership depends upon the sequencer protocol for achieving consensus on a new membership view (i.e., membership initiates atomic broadcasts on its own). Co-dissemination occurs when a message needs to be carbon-copied to more than one higher level protocol to notify that an event has occurred.

Our findings are consistent with other efforts [5] that have faced limitations of frameworks like the x-kernel when pushing protocol composition to its limits. Our solution was to extend the CORDS framework with more powerful flavors of existing semantics, or by adding new semantics (such as a publish/subscribe relation among protocols).

Managing system resources is extremely useful in GIPC. The non-interference (or, better, controllable interference) property among paths can be extended to individual process groups or functions within a group. The first and also simplest application of paths within GIPC was to isolate the fault suspector's traffic and have it use reserved network buffers and threads with some real-time scheduling attributes. This way the fault suspector path can be shielded from the node workload, and the chances of "heartbeats" falling behind, resulting in false detections, are decreased. More sophisticated uses of paths include implementing privileged process groups and expedited messages within a group. As a message is associated with a path, and thus with threads and scheduling attributes, messages with

higher priority naturally preempt messages with lower priority. Programming with resources is also a useful line of defense against cascades of failures that originate from lack of resources (e.g., no more network buffers)

7. Empirical results

Our standard real-time testbed consists of a pool of 100 Mhz Pentium PCs connected by 10 Mb/s Ethernet, Myrinet, and FDDI[10]. The PCs run GIPC and the real-time MK operating system.

We engaged up to 8 PCs in a GIPC process group, with one group endpoint on each of the PCs. The test driver that exercises GIPC, and thus simulates a RDDS, performs FIFO Atomic Broadcasts to the group from one of the endpoints; it uses the synchronous version of the GIPC API for posting to the group. The group is configured with a 300 milliseconds liveness granularity; this is the single largest contributor to the observed failure detection time of endpoints.

In characterizing the real-time properties of the system, we repeated the process of identifying outliers and correlating them with non-deterministic behaviors in GIPC, in the operating system, or in the underlying firmware/hardware. In each case, we were either able to fix bugs or tune parameters to eliminate the outliers, or we determined that they were the result of unpredictable behaviors of firmware/hardware. For our testing, it was far more practical to instrument CSMA/CD and understand the impact of collisions, if any, than to use link technologies with unexpected real-time limitations, such as Myrinet (Section 2.3). Therefore, we used Ethernet with an instrumented CSMA/CD, and in the load tests, we excluded a very small number of data points affected by unacceptably long CSMA/CD collision storms.

In the first set of measurements, we investigated the effect of packet loss on group post latency and membership view latency. We measured the minimum, average, and maximum latency of a group post operation on a node which was continuously posting, while packets on the network were artificially dropped at each of the physical nodes in the group (Figure 4). In this test, we do not consider any membership event.

Figure 5 shows the latency of a membership event (i.e., a PC is manually crashed) during the same regime of packet omission failures as in Figure 4. The membership event delivers a new, stable membership view; the latency includes two factors: the time for the failure detection (configured at 300 msecs) and the time for the atomic broadcast

10. FDDI connectivity is temporarily limited to 2 nodes, due to logistical problems

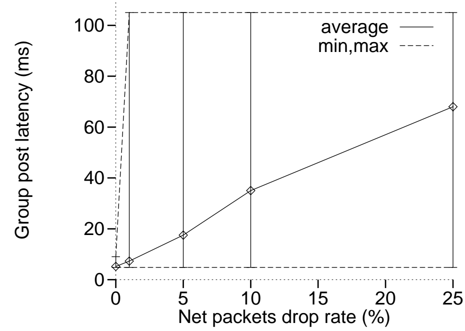


Figure 4: Group post latency vs. packet drop rate.

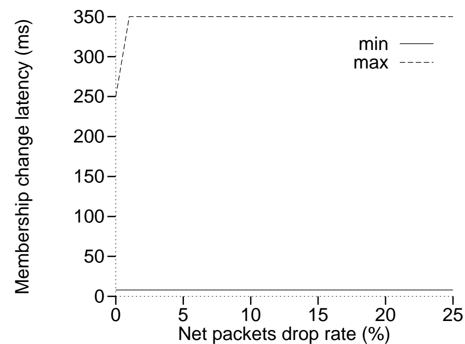


Figure 5: Membership view latency vs. packet drop rate.

that makes the proposed membership view stable (shown in Figure 4). Note that we are considering packet loss that is much greater than that observed during typical use, but might be observed in a partially damaged network or in one under attack.

In the following set of measurements, we have quantified the effects of network load on group post latency. Without using CORDS paths, we observe a severe degradation (Figure 6). CORDS paths and early demultiplexing,

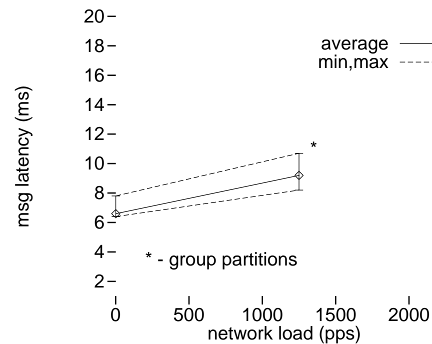


Figure 6: Group post latency vs. net load, without paths.

however, allow us to protect the group traffic from network load (Figure 7), and provide a better sample distribution. In both cases, the group eventually partitions due to interference from the competing load. However, the use of paths makes the system much more resistant. This result reaffirms the effectiveness of paths in a different environment than we used in [29].

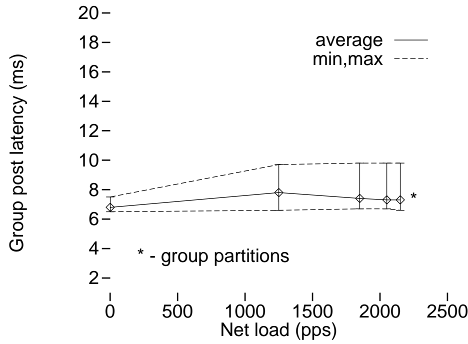


Figure 7: Group post latency vs. net load, with paths.

We have quantified the effect of system load on group post latency (Figure 8). The load is generated via AIMIII, a multi-user test suite that simulates a variable number of typical users for each of the PCs in the GIPC group. We also note that the AIMIII load does, in fact, compete with GIPC and its test driver; the benchmark runs more slowly when the GIPC test is running.

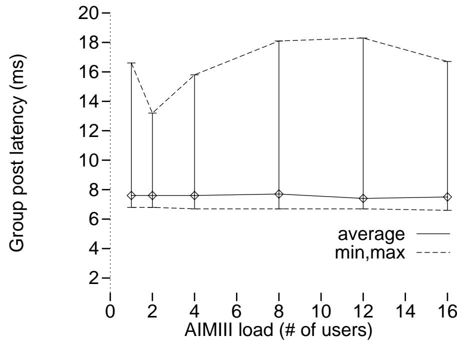


Figure 8: Group post latency vs. multi-user load.

Finally, we investigated the relationship between the group post latency and the number of PCs in the group. Positive acknowledgments are obviously responsible for the progressive loss in responsiveness; there is no evidence, however, of outliers. The switch between positive and negative acknowledgments anticipated in Section 4 will allow each RDDS to pursue its own trade-off of responsiveness vs. scalability (work in progress).

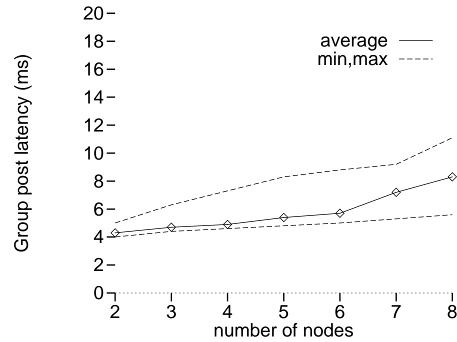


Figure 9: Group post latency vs. group size

8. Empirical results from use of a RDDS

We have developed a small-scale, dependable, real-time distributed process control application for a physical testbed (Figure 10).

Colored golf balls are dropped into a vertical tube, where they fall at gravity speed. At the end of the tube, flippers divert each ball to the left or right, depending on color. If no decision has been made or if the decision happens at the wrong time, the ball falls through the middle and is designated a failure.

One computer, the ‘‘control’’ node, is physically connected to the sensors and controllers on the apparatus. The timing and sorting decisions are performed by the other ‘‘compute’’ nodes. The control node and the compute nodes form a process group. The group members also have access to a distributed clock realized through LTS bounded-offset clock synchronization protocol (also implemented within CORDS).

Sensors along the tube record the initial velocity and reflectivity index (color) of the ball, which are read by the application on the control node. The application on the control node uses GIPC to make a reliable ordered broadcast of this data to the compute nodes.

The group members on the compute nodes use the reflectivity value to determine color and calculate the distributed clock time at which the golf ball will reach the bottom of the tube—a simple physics calculation suffices. At that time, they each send an unreliable message back to the control node, directing it to move a right or left flipper. If no message is received or is received at the wrong time, the ball drops through the tube, unsorted.

If we crash one of the compute nodes, the consensus process on the new group membership view will start and take precedence over any other traffic. We claim that there is an upper bound on the time it takes to terminate the con-

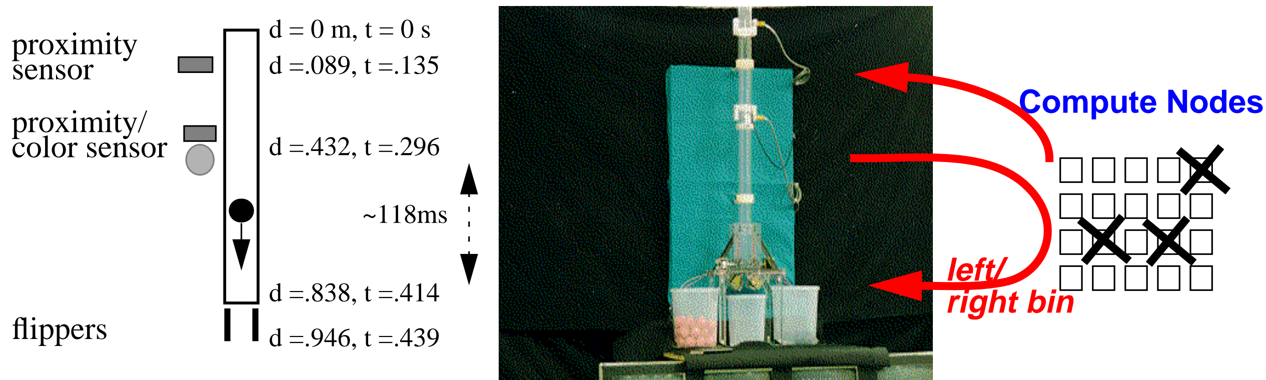


Figure 10: A distributed process control application.

sensus process with a new membership view. Furthermore, the distributed control application only works if the population in the latest view is greater than a pre-established limit: below this limit, the application suddenly stops with recognizable failure semantics.

Since golf balls are mechanically fed into the tube at the rates of up to 4 balls per second, we assert that at most one ball will be dropped for each crash of the compute nodes, until we reach the threshold for the lowest membership acceptable.

As a demonstration, the application is convincing because the timing constraint and success or failure condition are extremely clear and obviously independent of any computational element in the system.

The demonstration[11] is also an excellent, though simplified, model of more realistic applications. The actual calculation is trivial and any live compute node can complete the control loop. In a more realistic application, the calculation may need to be divided among the compute nodes in some way. Or the algorithms chosen for the calculation may be based on the number of available compute nodes in the group. The real-time process group paradigm simplifies the development of these kinds of applications.

9. Related work

To our knowledge, Flaviu Cristian introduced the timed asynchronous model and the notion of performance failures [11]. These are the foundation of our GIPC work.

ISIS [3] and its successors Horus [4] and Ensemble are middleware for building fault tolerant services. They provide a host of delivery semantics and complex services for transparent replicas, partition healing, etc. They are not, however, specialized on real-time behaviors. Our approach is to experiment with real-time functionalities first, and make progress along richer semantics and scalability later on. We have adopted the virtual synchrony model; we are uninterested, however, in causal ordering and we only consider atomic broadcast for our set of real-time functionalities.

Transis [19] and Totem [20] are also middleware for fault tolerant services; they exhibit a stronger real-time behavior than ISIS and its successors. To our knowledge, however, there are no published results on their maintaining predictable behavior in the presence of unpredictable and potentially significant competing load. While Totem uses a rotating token to implement atomic broadcasts, GIPC uses a master/slave schema, which we thought to be more responsive in RDDS scenarios. GIPC's modularity, however, does not preclude from replacing the master/slave schema with Totem's protocol or hybrids.

The master/slave protocol that we have chosen for atomic broadcasts is similar to the one in Amoeba. Our implementation choices, however, are not consistent with the Amoeba code.

From Consul [21], we have taken the protocol framework (i.e., the x-kernel) and the goal of modularity. We have not imported the Consul protocols, though, because Psync would have brought more complexity that was actually needed in our context (we do not aim to have context graphs for recovery nor do we want to take advantage of causal ordering). Horus is also highly modular and inspired from earlier x-kernel work as well.

With the DELTA-4 XPA [26] project, we share the strong QoS connotation of the xAMp protocol suite.

The ARMADA project [1] at University of Michigan and the Cactus project [17] at University of Arizona have a

11. With a MPEG capable Web browser, it is possible to see the control application and the apparatus in operation at URL http://www.opengroup.org/RI/PubProjPgs/CORDS.htm.

research agenda similar to GIPC. Their real-time reference platform is also based on CORDS and MK; with them, we exchange technology in the form of CORDS protocols (e.g., GIPC, ORCHESTRA).

10. Conclusions and future work

We have described our experience in building middleware explicitly targeted to RDDSs, from the understanding of the requirements to the characterization of the resulting GIPC technology and its actual demonstration application. Preliminary empirical results show that the central focus on real-time properties pays off. It is also interesting to note that our work draws some of its strength from the CORDS development environment, its object-based nature, and its real-time properties, including controllable interference among communication channels.

Our experience argues that satisfying real-time requirements is more than a simple matter of adding small number of timeouts to a large legacy code base. On the other hand, we anticipate (and welcome) criticisms from those who believe that we have not done nearly enough to address, say, mission critical real-time scenarios. This is the intrinsic charm of the real-time discipline.

GIPC and CORDS are available on MK and, with more limited real-time properties, on UNIX, and Windows NT.

GIPC is an ongoing effort. We anticipate developments in the paradigms for composing multiple process groups, and new abstractions derived from process groups, like, for instance, group RPCs [8]. We are also interested in communication paradigms stronger and weaker than FIFO Atomic Broadcasts (e.g., FIFO Atomic Broadcasts with estimated physical clock ordering and scalable reliable multicast, respectively).

Acknowledgments

The authors would like to thank the reviewers for their insightful comments. Alexander Brown, Ray Clark, Saadat Dowlati, Yueying Fei, Michael Leibensperger, the SHAWS team, and many other colleagues at the Open Group Research Institute have greatly improved the GIPC technology and this paper. The authors are grateful to Doug Wells, the director of the real-time program, for his continued support of the CORDS project, and for envisioning the fault tolerant demonstration application described in Section 8.

The authors would also like to thank Prof. Farnam Jahanian and Scott Dawson at the University of Michigan for making their ORCHESTRA work available within CORDS, and for providing feedback on early snapshots of GIPC.

References

[1] T. Abdelzaher et al., “ARMADA Middleware Suite,” Proceedings of IEEE Workshop on Middleware for Distributed Real-time Systems and Services, pp 11-18, San Francisco, CA, December, 1997.

[2] M. Accetta et al., “Mach: A New Kernel Foundation for UNIX Development,” Proc. of the Summer Usenix Conference”, Atlanta, GA.

[3] K.P.Birman, R.Cooper et al. “Isis - A Distributed Programming Environment”, User Guide and Reference Manual, Cornell University, Ithaca, NY, Jun 1990.

[4] Ken Birman, “Building Secure and Reliable Network Applications,” ISBN 1-884777-29-5, Manning Publications Co.

[5] N. Bhatti, R. Schlichting, “A System for Constructing Configurable High-Level Protocols,” Proceedings of SIGCOMM ‘95, August 28 1995, Cambridge, MA, USA.

[6] J. CaraDonna, “The Event Trace Analysis Package: ETAP User’s Guide,”OSF Research Institute TR, February, 1995.

[7] D.R.Cheriton and D.Skeen. “Understanding the Limitations of Causally and Totally Ordered Communication”, Proc. Fourteenth Symposium on Operating System Principles, pp44-57, Asheville, NC, Dec 1993.

[8] R.K. Clark, E.D. Jensen, F.D. Reynolds, An Architectural Overview of the Alpha Real-Time Distributed Kernel,” Proc. USENIX Workshop on Micro-kernels and Other Kernel Architectures, pp 127-146, Seattle, WA, April 1992.

[9] F. Cristian, R. De Beijer, and S. Mishra, A Performance Comparison of Asynchronous Atomic Broadcast Protocols. Distributed Systems Engineering, Vol. 1, No. 4 (June 1994), 177--201.

[10] F. Cristian. Probabilistic Clock Synchronization. In Distributed Computing, 3:146--158, 1989.

[11] F. Cristian, “Understanding fault-tolerant distributed systems,” Communications of ACM, 34(2):56-78, Feb 1991.

[12] F. Cristian, S. Mishra, “The Pinwheel Asynchronous Atomic Broadcast Protocols,” UCSD TR.

[13] S. Dawson, F. Jahanian, T. Mitton, “Fault Injection Experiments on Real-Time Protocols Using ORCHESTRA,” Proceedings of IEEE High-Assurance Systems Engineering Workshop, October 21 1996, Niagara on the Lake, Ontario, Canada.

[14] L. Feeney. LTS: Algorithms and Implementation. The Open Group Research Institute TR, to be published.

[15] L. Feeney, P. Bernadat, and F. Travostino, “Characterizing Group Communication Middleware for a Real-time Distributed System,” WIP report for the Real-Time Systems Symposium, December,1997, San Francisco, CA.

[16] M.J. Fischer, N.A.Lynch and M.S.Paterson. “Impossibility of distributed consensus with one faulty process”, Journal of the ACM, 32(2):374-382, Apr 1985.

[17] M.A. Hiltunen, X. Han, R.D. Schlichting, “Real-Time Issues in Cactus,” Proceedings of IEEE Workshop on Middleware for Distributed Real-time Systems and Services, pp 214-221, San Francisco, CA, December, 1997.

[18] M.F.Kaashoek. “Group Communication in Distributed Computer Systems”, PhD Thesis, Vrije Universiteit, Amsterdam 1992.

[19] D. Malkhi, “Multicast Communication for High Availability,” Ph.D. diss., Hebrew University of Jerusalem, 1994.

[20] L.E. Moser et al., “Totem: A Fault-Tolerant Multicast Group Communication System,” Communications of the ACM 39:4 (April 1996):54-63.

[21] S.Mishra, L.L.Peterson and R.D.Schlichting. “Experience with Modularity in Consul”, Software Practice and Experience, Vol 23(10), 1059-1075, Oct 1993.

[22] S.O’Malley and L.L.Peterson. “A Dynamic Network Architecture”, ACM Transactions on Computer Systems, 10(2), May, 1992.

[23] Sape Mulender, Vassos Hadzilacos, Sam Toueg. “Distributed Systems - Chapter 5 - Fault -Tolerant Broadcast and Related problems” pp 97-145. Addison-Wesley, 1993

[24] M. Russinovich, Z. Segall, D. Siewiroek, “Application Transparent Fault Management in Fault Tolerant Mach,” Proceedings of the 23rd Int. Conf. on Fault-Tolerant Computing (FTCS-23), June 10 1993, Toulouse, France.

[25] L. Rodrigues, E. Siegel, P. Verissimo, “A Replication-Transparent Remote Invocation Protocol,” Proceedings of the 13th Symposium on Reliable Distributed Systems, October 25-27 1994, Dana point, CA.

[26] L.Rodrigues and P.Verissimo, “xAMP: a Multi-primitive Group Communications Service”, Proceedings of the 11th Symposium on Reliable Distributed Systems, Houston, TX, October 1992.

[27] F. Schmuck, F. Cristian. Continuous Clock Amortization Need Not Affect the Precision of a Clock Synchronization Algorithm. In Proceedings of the 9th Annual Symposium on Principles of Distributed Computing, 1990.

[28] F.Travostino, F.Reynolds. “An O-O Communication Subsystem for Real-Time Distributed Mach”, Proceedings of the 1994 IEEE Workshop on Object-Oriented Real-Time Dependable Systems, Irvine, Ca, October 1994.

[29] F.Travostino, E. Menze and F.Reynolds, “Paths: Programming with System Resources in Support of Real-time Distributed Applications”, Proceedings of the 2nd IEEE Workshop on Object-Oriented Real-Time Dependable Systems, Feb 1-2 1996, Laguna Beach, CA.

[30] Quorum. http://www.ito.darpa.mil/research/quorum