Characterizing Group Communication Middleware for Real-time Distributed Systems*

L. M. Feeney, P. Bernadat, F. Travostino

The Open Group Research Institute
11 Cambridge Center, Cambridge, MA 02142
{imfeeney, bernadat, travos}@opengroup.org

Abstract

This paper presents our current work in characterizing the behavior of a real-time dependable distributed system, which must exhibit predictable behavior under load in the presence of partial failures. We focus on measuring the end-to-end properties of the middleware implementing the real-time process group service, specifically its membership and message latency. The paper also describes the tools and techniques we have developed, along with some of the practical issues that arise in instrumenting a real-time distributed system.

1 Introduction

A major focus of research at The Open Group Research Institute is the development of real-time dependable distributed systems. One of our goals is to provide our academic and industrial partners with configurable, reusable frameworks for research and further development. Target applications include factory automation, sensor monitoring and combat systems, which require predictable behavior, even in emergency conditions.

Under this program, we have implemented middleware for a real-time process group service (GIPC). In contrast to group communication services which emphasize throughput (e.g., ISIS[2]), GIPC is distinguished by its focus on real-time issues, configurability, and predictable behavior.

This paper describes our ongoing work characterizing GIPC behavior under load and in the presence of partial failures. Ensuring real-time guarantees at the application-level requires appropriate real-time behavior at all levels in the system. Moreover, because of our focus on configurability, we expose many tunable parameters within the system. Experience with how these parameters interact and how to select appropriate values for different application requirements is essential if we expect others to adopt our work.

Measuring a live real-time system presents issues that do not arise when working with a simulation. The instrumentation must be minimally intrusive, and must not be adversely affected by the load or failures whose effects we are trying to measure. Specialized hardware, such as a synchronized clock, dedicated CPU or bus snooper, can be used to address some of these issues. However, this becomes costly for more than a few nodes and runs counter to our preference for general-purpose, commodity hardware. Instead, our approach uses the real-time properties of the system to isolate the instrumentation.

Following a brief description of the system, we specify the high-level properties to be characterized and the variables on which they depend. We describe the tools and instrumentation we have developed and report on some preliminary measurements. Early measurements have confirmed some expected properties of the system and indicated some anomalies, emphasizing the importance of this sort of characterization. We note that our tests are often specifically constructed to measure one aspect of the system and do not generalize well. We speculate that a characterization framework is important for making these sorts of measurements.

2 Component Technologies

While our modular design and emphasis on configurability allow us to support many group communication paradigms and qualities of service, our current focus is on the powerful semantics of FIFO atomic broadcast and the virtual synchrony model[2]. GIPC provides FIFO atomic broadcast with safe delivery to the failure domain of the atomic broadcast engine of each group member. It also guarantees that group members order all messages and membership views with a global sequence number.

By implementing these semantics in a real-time environment, we have created a very strong base on which to build real-time dependable applications. The GIPC middleware is policy-neutral; the application is expected to participate in fault management and quality of service negotiation via the GIPC API.

*This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Rome Laboratory of the Air Force Materiel Command (AFMC).
Some of GIPC’s real-time properties, i.e. those based on reservation and prioritization, derive from its implementation within the CORDS (Communication Objects for Real-time Dependable Systems)[9] framework. CORDS is an object-oriented communication framework based on the University of Arizona x-kernel[7]. CORDS provides a framework for composing a graph (stack) of micro-protocol objects. Messages are shepherded through the protocol graph via object invocation.

CORDS’ most significant extension to the x-kernel framework is the path abstraction[10], which unifies resource (buffer and thread) reservation and allocation along each communication channel through the protocol graph. This mechanism provides for controlled interference among CORDS traffic. For high priority traffic, such as GIPC liveness messages, paths can be used to ensure that reserved buffers are available to receive the incoming packets and that high priority threads are waiting to shepherd them through the protocol graph.

The CORDS framework is implemented in the MK7 operating system, a portable real-time microkernel-based system derived from Mach 3.1 Key features include kernel preemption and fixed-priority scheduling.

3 Characterization Parameters

Our goal is to characterize the high-level properties of the system that are relevant to the real-time requirements of an application.

1. Membership latency is the interval between a membership change and the delivery of the new membership view to all endpoints. This involves detecting the membership change and notifying the group via atomic broadcast. In addition to voluntary membership changes initiated through the API, membership may change due to software failure, node failure, omission failure (dropped messages), network partition or performance failure (a slow node is evicted as if dead). Our experience shows that sub-second guarantees on membership latency are indeed feasible.

2. Message latency is the interval between the time an application posts a message (FIFO atomic broadcast) and the time it is received by all the other endpoints. The (synchronous) gipcPost API call does not return until the atomic broadcast has either successfully completed or returned a failure. Therefore, the time required for a gipcPost is a good approximation of the message latency.

3. Group stability is a measurement of the probability that a live node will be evicted from the group due to omission or performance failures.

Some issues involved and configuring and tuning the system include:

1. The overhead associated with maintaining group liveness is basically determined by the heartbeat rate (interval at which members send heartbeats). Membership latency derives from the tolerance (interval at which the master evicts members from which it has not received a heartbeat). The master will also evict a member after making some number of retransmissions while attempting to perform atomic broadcast, so this is another factor in membership latency. The ratio of tolerance to heartbeat rate is important to group stability. Clearly, there are complex tradeoffs among these parameters.

2. We must select priorities for the various CORDS paths, specifying the relative privilege of each type of CORDS traffic.

3. We have learned that we can not just consider relative priorities within the CORDS framework. We must also consider CORDS and GIPC interactions with both the kernel and the OS server.

We also measure the effects of load and partial failures on these properties. Specifically, we consider:

1. CPU load: We use AIMIII, a multi-user UNIX benchmark which simulates mixed user activity, as our load generator.


3. Packet Drop: The CORDS protocol suite includes a simple vdrop protocol, which drops some fraction of its incoming traffic.

4 Characterization Framework

We present some tools which we are using or plan to use in the characterization process. We have had success using LTS synchronized clocks for distributed measurements. Two other tools, DETAP and ORCHESTRA, appear to be potential building blocks for a characterization framework.

4.1 LTS

A distributed clock is necessary to measure intervals between events on separate nodes. LTS[6] is a bounded-offset external clock synchronization service based on probabilistic synchronization[4] and clock amortization[8]. The bounded-offset property of LTS ensures that the difference between the master clock and each of its clients will always be less than a predetermined constant. Thus the interval between events on different nodes can be measured with a known accuracy.
Because LTS requires stable message round trip times, a CORDS path is used to ensure that LTS traffic is privileged. An LTS client clock explicitly fails when it detects that it can no longer maintain its bounded-offset, so applications must be prepared to handle LTS clock failure. For a small number of machines with fairly good local clocks (worst case drift 100-200ppm), a guaranteed bound of 1 ms between two client clocks is feasible and requires about 8 messages per second per client.

4.2 DETAP

DETap is a tool for annotating system event logs with distributed timestamps.

ETAP (Event Trace and Analysis Package)[3] is the MK kernel instrumentation package. Code is instrumented with ETAP “probes”. When a probe is encountered, an event identifier, local timestamp, thread id, and 16 bytes of event-specific data are stored in a kernel log buffer. ETAP event tracing can be enabled for any combination of specific threads and event types as well as for interrupts and context switches. The log buffer can be read in the kernel debugger, or by an application that periodically writes the buffer to disk.

DETap uses ETAP and LTS to provide a distributed timestamp for each ETAP event. To avoid the overhead of reading the distributed timestamp for each event, we use semi-offline clock synchronization[1]. Each successful LTS communication is recorded as an ordinary ETAP event. Later, a script uses the LTS events in the ETAP log to recreate the distributed clock. This simulated distributed clock supplies the distributed timestamp for every event recorded while the LTS clock was valid.

Despite the complexity of instrumenting code for use with DETAP, the value of the LTS distributed clock for measuring membership latency makes it seem promising.

4.3 ORCHESTRA

ORCHESTRA[5] is a general, portable fault-injection toolkit being developed at the Real-Time Computing Lab at the University of Michigan, where it has been ported to CORDS.

ORCHESTRA uses a generic fault-injection protocol to do fault injection into multi-layer protocol stacks. The fault-injection protocol peeks at the headers in an incoming message to determine its destination and fate. Tcl scripts are used to describe protocol headers and do the fault-injection and analysis. This has proved extremely useful in testing and debugging CORDS/GIPC protocols. Because of the Tcl machinery, it is less useful for real-time characterization.

There is interest in enhancing ORCHESTRA for this purpose, leveraging the many benefits of the general toolkit. The fault-injection and udrop protocols are closely related. We plan to enhance udrop to support programmable patterns of loss using a control bitstream, rather than an interpreted script. We are also enhancing the udipitest protocol to support more complex programmable traffic generation.

5 Preliminary Measurements

These measurements were done using 100MHz DEC Celebris Pentium PCs on a private 10Mb Ethernet.

5.1 Membership Latency

To measure the membership latency in case of node failure, we use the LTS distributed clock to obtain the distributed time at which the new membership view is received at each node. Manually entering the kernel debugger disables the node, simulating a crash failure. The difference between the distributed time recorded at entry to the kernel debugger and the distributed time at which the membership view is reported at the remaining nodes is the membership latency. The precision of the measurement is constrained by bounded offset on the distributed clock (1ms). We have used this (highly manual) method for small number of trials and confirmed the feasibility of guaranteed membership latency as low as 150-300ms. In making these measurements, we also discovered and corrected problems arising from asymmetry between master and slave liveness parameters.

5.2 Message Latency

We measured the message latency, as approximated by the time for a synchronous gipcPost API.

![Figure 1: Message Latency (GIPC unprotected)](image)

In Fig. 1, GIPC traffic is directed to a path with priority equal to that of udipitest traffic. The average and maximum latency increase dramatically with load.
In Fig. 2, GIPC traffic is directed to a path with a priority greater than that of wdpipext traffic. The data actually falls into three distinct bands. The average latency is quite stable for all loads. In addition to the dense band of data points near the 4.7ms average, there are occasional data points at around 15ms and 50ms. We note that 10ms is the kernel alarm resolution and 30ms is CORDS’ internal heartbeat. The challenge is not only to fix this problem, but also to develop a toolkit for investigating such real-time anomalies.

5.3 Other experiments

We have also experimented with message latency under CPU load. The average and maximum message latency increase significantly as the load reaches about 10 simulated users, even though it is running with less priority than any GIPC activity. The anomaly may be caused in part by disk operations which originate from low priority threads, but nevertheless eventually generate interrupts. This highlights the need for tools which provide a window into activity at all levels of the system.

We used the vdrop protocol to measure the effect of packet drop on message latency. For drop rates of up to about 25%, we see a maximum message latency of 100ms, a value controlled by retransmission timeout. As the drop rates grow toward 50%, the group becomes highly unstable.

6 Lessons Learned

So far, we have learned three major lessons from this work.

The first lesson is that, overall, the characterization is very promising. While we are finding (and fixing) a number of specific anomalies, there seem to be few instances of wildly erratic behavior. Our experience emphasizes the importance of the characterization process. We do not trust simulations or microbenchmarks.

The second lesson is the importance of developing a characterization framework. The tests described above are often rather ad hoc. In some cases, thread priorities and other parameters had to be specifically configured so that the instrumentation itself did not fail under load. A common problem was recording the data, whether to screen or to disk, operations which (unlike the CORDS/GIPC middleware) require OS server intervention. Keeping data in kernel buffers can be impractical due to the level of manual intervention required and because GIPC and LTS treat entering the kernel debugger as a failure. Given our successes with privileged and non-privileged paths, we will investigate the possibility of using CORDS to retrieve data remotely.

More importantly, when we see anomalies such as the 15ms and 50ms measurements in the membership latency test, we don’t have tools for pursuing lower-level measurements in a systematic way. We have no general way of doing useful things like recording a protocol’s progress through its state machine, or determining where a message was delayed. We expect that DETAP, which can record events at every level of the system, will be helpful in this regard.

The third lesson is the importance of real-time as a bottom-up property affected by interactions at all levels of the system. A characterization framework will allow us to more effectively investigate anomalous behavior and to more easily integrate instrumentation into the overall real-time structure of the system.

References


