The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation

Ragunathan Rajkumar, **Mike Gagliardi** and **Lui Sha**
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
{rr,mjg,lrs@sei.cmu.edu}

Abstract

Distributed real-time systems[1] are becoming more pervasive in many domains including process control, discrete manufacturing, defense systems, air traffic control, and online monitoring systems in medicine. The construction of such systems, however, is impeded by the lack of simple yet powerful programming models and the lack of efficient, scalable, dependable and analyzable interfaces and their implementations. We argue that these issues need to be resolved with powerful application-level toolkits similar to that provided by ISIS [2]. In this paper, we consider the inter-process communication requirements which form a fundamental block in the construction of distributed real-time systems. We propose the real-time publisher/subscriber model, a variation of group-based programming and anonmyous communication techniques, as a model for distributed real-time inter-process communication which can address issues of programming ease, portability, scalability and analyzability. The model has been used successfully in building a software architecture for building upgradable real-time systems. We provide the programming interface, a detailed design and implementation details of this model along with some preliminary performance benchmarks. The results are encouraging in that the goals we seek look achievable.

1. Introduction

With the advent of high-performance networks such as ATM and upward-compatible 100Mbps network technologies, the cost of network bandwidth continues to drop steeply. From a hardware perspective, it is also often significantly cheaper to network multiple relatively cheap PCs and low-cost workstations to obtain an abundance of processing power. Unfortunately, these cost benefits have been slower in accruing to distributed real-time systems because of the still formidable challenges posed by integration issues, frequent lack of real-time support, lack of standardized interfaces, lack of good programming models, dependencies on specific communication protocols and networks, portability requirements and lack of trained personnel who perhaps need to be re-educated on the benefits and potentially major pitfalls of building working distributed real-time systems. We believe that the difficulty of programming distributed real-time systems with predictable timing behavior is in itself a significant bottleneck. Unfortunately, the problem is actually much harder because many other issues need to be addressed concurrently. These issues include the need to maintain portability, efficiency, scalability and dependability as the systems evolve or just become larger. Put together, these issues pose daunting challenges to the construction of predictable distributed real-time systems.

Three factors in particular seem to dominate the various phases of the life-cycle of developing and maintaining distributed real-time systems. First, the development, debugging and testing of distributed real-time systems is hindered by complexity; there are many degrees of freedom compared to uniprocessor systems which are conceptually easier to program. It would be extremely desirable to have a programming model which does not depend upon the underlying hardware architecture, whether it is a uniprocessor or a network of multiple processors. Secondly, systems change or evolve over time. Hardwiring any assumptions into processes, communication protocols and programming models can prove to be extremely costly as needs change. Finally, as distributed real-time systems grow larger, there is often a need to introduce new functionality by extending services and functions provided. Such extensions can become impossible if currently operational code needs to be rewritten to accommodate these changes, or sometimes even if the entire system just has to be taken down for installation. In other words, it would be desirable if a subsystem can be added online without having any downtime for the rest of the system.

1.1. Objectives for a Distributed Real-Time Systems Framework

Due to the many issues that need to be addressed simultaneously in building distributed real-time systems, we believe that a well-understood framework for building distributed real-time systems is essential. Specifically, the following objectives should be addressed by a framework for building such systems:

- *Ease of programming*: The framework must provide simple programming models and interfaces to the programmer. The implementation of this programming model is hidden from the application programmer and completely bears the responsibility of hiding the complexity of the underlying hardware (processor and net-

[1]This work is supported in part by the Office of Naval Research under contract N00014-92-J-1524 and in part by the Software Engineering Institute.

work) architecture, communication protocols and other configuration-dependent issues. Optimizations that are dictated by application performance requirements should still be done only at the implementation layer of the model. However, the programming model and interface exported to the programmer should be left untouched.

- *Portability*: The programming model should be portable across platforms and environments (at least at the source-code level). This objective basically implies that the model should not be bound to specific implementation-dependent features such as the availability of UDP/IP or a specific network medium.
- *Analyzability*: Since the system has real-time semantics, analyzability and predictability of the system's timing behavior cannot be sacrificed. It would be ideal if the programming model(s) and interface(s) come with schedulability models to facilitate scheduling analyses of applications using these interfaces.
- *Efficiency*: As long advocated by the real-time community, real-time systems are not "fast" systems but "predictable" systems from a timing point of view. Nevertheless, efficiency or performance is an important practical consideration in many applications. An elegant programming model whose implementation is unacceptably inefficient is unlikely to succeed in practice.
- *Scalability*: The natural appeal of distributed systems is that they can grow larger in order to provide added power and functionality as system needs grow over time. A programming model for these systems should be able to scale naturally. Similar to the ease of the programming requirement above, the responsibility for providing scalable performance and hiding programming complexity must lie in the implementation of the programming model.
- *Dependability*: As the system scales in size, components or subsystems will need to be changed because of considerations such as failure, preventive maintenance and upgrades. Since many real-time applications demand continuous and/or error-free operations, it is often critical in many cases that failure or maintenance problems do not cause damage to life and/or property.
- *Protection and enforcement*: Software errors in a new relatively untested module must ideally not bring the entire system down. Some form of protection and enforcement (such as denying a request for unauthorized access to a critical resource) can be very desirable.

Some of these requirements often seem and can be contradictory. For example, hard-wiring some assumptions may make analyzability easier. Scalability may sometimes conflict with both ease of programming and efficiency. Similarly, requiring portability or protection may sometimes affect efficiency adversely because of additional software layers and/or checks needed.

2. The Real-Time Publisher/Subscriber Communication Model

An inter-process communication (IPC) capability is a fundamental requirement of distributed real-time systems. All the goals listed in Section 1.1 have direct relevance to this communication capability. For example, it would be very desirable if the inter-process communication model is independent of uniprocessors or distributed systems, independent of specific communication protocols, independent of the network used, scales efficiently to a large number of nodes, works despite node or network failures, and fault(s) in some application module(s) do not bring down the entire communication layer. The IPC layer therefore constitutes a good test-candidate for the proposed framework. In the rest of this paper, we present the *Real-Time Publisher/Subscriber Communication* model for inter-process communication, along with its design and implementation which address many of these goals[2].

2.1. Related Work

The Real-Time Publisher/Subscriber communication model we describe is based on the group-based programming techniques [2] and the anonymous communication model [4]. Such a model in a general non-real-time context is also sometimes referred to as "blindcast" [6]. The model we propose is similar to yet different from these recently studied protocols in that we are not only driven by the need to maintain ease of programming, but also by the need to maintain analyzability, scalability and efficiency for real-time purposes.

2.2. The Communication Model

The Real-Time Publisher/Subscriber model associates logical handles to message types. Messages can be generated and received based only on these logical handles without any reference to the source(s) of the messages, destination(s) of the messages, and independent of the underlying communication protocols and networks in use. Once a logical handle has been created, sources (publishers) can send or publish messages with that handle. Sinks (subscribers) can subscribe to and receive all messages published with that handle. At any given time, the publishers need not know the subscribers and vice-versa. In fact, there may neither be any subscribers to a published message, nor any publishers on a logical handle subscribed to. Publishers and subscribers may also obtain or relinquish publication and subscription rights dynamically. It is also possible for an application thread to be both a publisher and a subscriber simultaneously. The logical handle itself can be as simple as a variable-length ascii string.

---

[2]In this paper, we ignore the dependability requirement but our current design and implementation have been carried out with this requirement in mind. Ongoing extensions to this design and implementation will allow the dependability requirement to be addressed as well.

In contrast to pure "blindcast" techniques, we also allow the publishers/subscribers to know *if needed* who the publishers/subscribers are on a certain message type, and the source of any received message. We allow this because additional publishers and/or subscribers have performance impact from a timing perspective, and it is impossible for a framework provider to judge all the needs of application-builders. However, by providing the information that applications may need, the framework can at least allow an application to make its own decisions. For example, the application could check the number of current publishers on a message type before allowing a new publication right to be requested. We believe that this concern of letting the application have access to performance-sensitive information and thereby have control over application performance is another significant issue that needs to be addressed by a framework for distributed real-time systems.

2.3. The Application Programming Interface

```
class DistTag_IPC {
 private:
  ...
 public:
  // create or destroy distribution tag
  Create_Distribution_Tag(tag_id_t tag, domain_t domain);
  Destroy_Distribution_Tag( tag_id_t tag );

  // get/release publication rights on distribution tag
  Get_Send_Access( tag_id_t tag );
  Release_Send_Access( tag_id_t tag );

  // get/release subscriber rights to distribution tag
  Subscribe( tag_id_t tag );
  Unsubscribe( tag_id_t tag );

  // publish on specified distribution tag
  Send_Message( tag_id_t tag, void *msg, int msglen );

  // receive a message on a subscribed tag:
  // returns the process id of the sender: specify time
  //  to wait and the node address of the sender is
  // returned in optional "out" parameter.
  Receive_Message( tag_id_t tag, void *msg, int *msglen,
          timeval_t *tout, in_addr_t *from_addr = NULL );

  // purge all input messages queued locally on tag
  Purge_Messages( tag_id_t tag );

  // is a message available on a specified tag?
  int Message_Available( tag_id_t tag );

  // query list of publishers/subscribers on a tag to
  // allow applications make own decisions.
  sender_list_t
     Get_Senders_On_Distribution_Tag( tag_id_t tag );
  receiver_list_t
     Get_Receivers_On_Distribution_Tag( tag_id_t tag );

  // Notes: In our implementation, tag_id_t is (char *)
  // and tag id's are specified as ascii strings.
  // "domain_t" used during tag creation can be
  // GLOBAL or LOCAL, but only GLOBAL is supported.
};
```

Figure 2-1: The Application Programming Interface for the RT Publisher/Subscriber Paradigm

The C++ application programming interface for our real-time publisher/subscriber model is given in Figure 2-1. An illustration of how exactly the API would be used is presented later in Figures 4-1 and 4-2. The notion of a *distribution tag* is key to our communication model. It is the logical handle by which all messages can be published and received. As its name implies, the tag represents how a message will be distributed. In this case, the tag contains information regarding the publishers and subscribers on that tag (including their node addresses, and communication-protocol dependent information such as port numbers, information which is not seen by the user). Hence, the API provides calls to create and destroy distribution tags. Once a distribution tag has been created, it can be used to obtain publication and/or subscription rights, to publish messages with that tag, and to receive messages with that tag. Associated with each distribution tag can also be a set of real-time attributes such as the priority at which message(s) with this tag will be transmitted.

2.4. Fault-Tolerant Clock Synchronization

Consider the problem of distributed clock synchronization [1]. Typically, clock slaves send a message to a clock master requesting the time, and the clock master responds with the time it has. The slaves use the master time and knowledge of the transmit time to update their local view of the globally synchronized clock. If the clock master dies, some other node must become clock master and all slaves must now re-direct their time requests to the new master. With the real-time publisher/subscriber communication model, two (or more) nodes can choose to act as cooperating clock masters. They obtain subscription rights to a distribution tag with logical name "Clock Master Request", and all slaves get publication rights to this tag. Slaves can also register the unique tags on which they expect to see their responses by sending registration messages to "Clock Master Request". A primary clock master response to a request from a slave would be to this tag (unique to that slave). In addition, the clock masters run an internal handshaking protocol (using a tag like "For Use Only by Masters") such that (say) at most one publishes responses to slaves' requests. If one master dies, another master can realize this because of (say) absence of heartbeat on "For Use Only by Masters" and start publishing responses to the slaves. A slave's behavior does not change: it continues to publish its requests to "Clock Master Request" and to subscribe to its individual response tag. The slave literally does not care where the clock master responses come from. However, it must be noted that the masters themselves must take great care to keep *their* clocks more tightly synchronized and to respond to outstanding slave requests when a master fails.

2.5. Upgrading a Process

The fault-tolerant clock synchronization model above can also be used to upgrade a process which is already running. All communications to the process's external environment must be using distribution tags. When this process needs to be upgraded, its replacement process is created, and it obtains publication rights and subscription rights to all the tags used by the process being replaced. Then, using a pre-defined handshaking protocol, the original process stops publishing its messages; simultaneously, the replacement process starts publishing the same messages and reacting to the messages it receives on its subscribed tags. As a result, as far as the rest of the environment is concerned, this process "looks" the same as the replaced process but can have enhanced functionality. This upgrade paradigm is at the heart of the Simplex architecture framework for building upgradable and dependable real-time systems [10] and is built upon our real-time publisher/subscriber communication model as just described.

3. The Design and Implementation of the RT Publisher/Subscriber Model

The key behind our design of the real-time publisher/subscriber communication model can be quickly summarized as follows. The generation and reception of messages happen in the "real-time loop" or the "steady-state path" and must therefore be as fast as possible. Conversely, the creation and destruction of tags, and getting publication or subscription rights are viewed as primarily non-real-time activities happening outside the real-time loop and can be very slow compared to actual message communications using tags. We refer to this as the *efficient steady-state path requirement*.

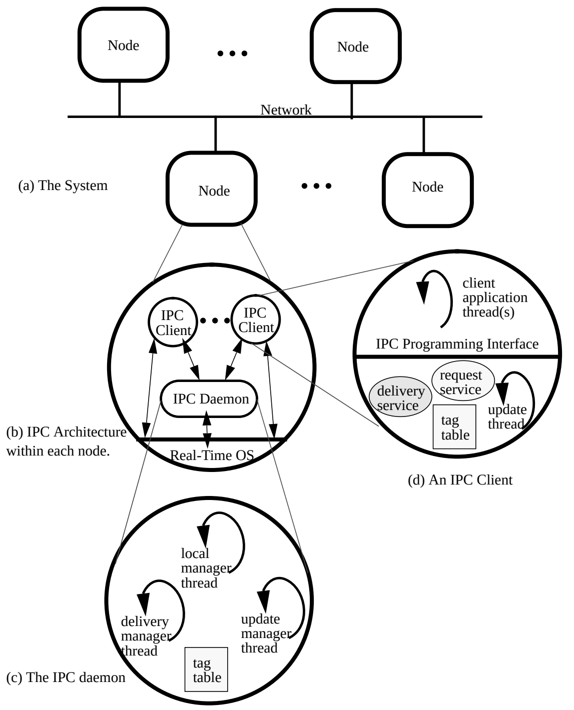

Figure 3-1: The Architecture of our Real-Time Publisher/Subscriber Communication Model.

The overall architecture of the design of our communication model is presented in Figure 3-1. Application-level clients (processes) use the communication model by making calls listed in Figure 2-1 which are implemented by a library interface hiding the complexity of the underlying implementation. IPC daemons on various nodes communicate with one another keeping them all appraised of changes in distribution tag status: tag creation, tag deletion, addition/deletion of publication/subscription rights. (Recall that the current implementation does not yet tolerate node failures.) A client which creates/deletes tags and gets publish/subscribe rights does so by communicating with its closest IPC daemon. Needed distribution tag information is then sent back to the client from the IPC daemon, and is stored in a local tag table in the client's address space. Any changes to this tag information because of requests on local or remote nodes are notified to the client (by the same daemon which processed the original tag request) and are processed asynchronously by an *update thread* created and running in the library. Naturally, the local tag table is a shared resource among this update thread and the client thread(s) and must be protected *without* the danger of unbounded priority inversion.

3.1. The IPC Daemon

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

- *Local Manager*: A local manager thread responds to requests for tag creation / deletion and publication / subscription rights from clients on or in close proximity to this node. *Not* being part of the efficient steady-state path, this thread can run at low priority.
- *Update Manager*: The update manager thread communicates with its counterparts in the other IPC daemons to receive tag updates. Any local manager thread receiving a request which changes the status of a distribution tag sends the status update notification to the update managers on all the other nodes, and then updates the local daemon's copy of the tag table. The remote update manager threads update their respective copies of the tag table to reflect the change status. In addition, they may also notify any of their local clients which have a copy of that tag. These asynchronous notifications are handled by the update thread in the client library. A local manager thread can respond to its client only when all notifications to the remote update managers are complete. Hence, the update manager thread runs at a higher priority than the local manager thread.
- *Delivery Manager*: Strictly speaking, the delivery manager is not necessary. When a client needs to send a message with a valid publication right, the distribution tag will be in its local tag table in *its* address space and will contain the current publishers/subscribers to this tag. The client can therefore directly send a message to

each of the subscribers on this list via the library. However, if there are multiple receivers on a remote node, it is often wasteful to send many duplicate network messages to a single node. The delivery manager is intended to address this (potential) performance problem. The client just sends one copy of its message to a delivery manager on the remote node, and the delivery manager delivers the message to all receivers on its local node using locally efficient mechanisms. In a larger system, only the delivery manager needs to be running on each node and it acts more as a performance enhancement technique in that case. Since the delivery mechanism is directly in the steady-state path, this thread runs at the highest priority of the 3 threads. Actually, in order to avoid undesirable remote blocking effects [8], this thread may even run at priorities higher than many application threads.

3.2. The Client-Level Library

In the user-level library on the client side, three services acting as counterparts to the three threads in each IPC daemon are supported.

- *Local request service*: This library module in the client space translates client calls for tag status changes into a request that is sent to the local request manager of the closest IPC daemon. It then blocks until a successful or failed response is obtained. This service is executed as part of the calling client thread at its priority.
- *Delivery service*: This library module in the client gets activated when the client publishes a message. Distribution tag information from the local tag table is obtained, and if any receivers are on remote nodes, messages are sent to the delivery managers running on these remote nodes. At most one message is sent to any remote node since the remote delivery manager will perform any needed duplication to multiple local receivers. This service is also executed as part of the calling client thread at its priority.
- *Update service and thread*: The update service is performed by a separate user-transparent thread in the library. As mentioned earlier, this thread receives and processes notifications of tag status changes from its host IPC daemon. Any changes that it makes to the client tag table must be atomic. The atomicity can either be provided by semaphores supported by priority inheritance or by emulating the ceiling priority protocol [7, 8]. This thread should run at higher priority than all client threads in this process using the IPC services.

3.3. Sequence of Steps on Various Calls

In this subsection, we summarize the list of actions taken for different categories of calls in the API of Figure 2-1.

When a non-steady-state path request (tag creation/deletion request, publish/subscribe right request) is issued,

1. The client's local request service sends a request to the local manager of the IPC daemon (on the closest node) and blocks after setting up a timeout.
2. The local request manager checks to see if the tag status change can be made. If so, it sends an update status to the update manager threads of remote IPC daemons[3]. Then, it updates its local tag table.
3. The local request manager then sends a response back to the client which unblocks and checks the return value for success.
4. If there is no response from the IPC daemon (typically because it was not started), the client times out and detects the error status.

When the steady-state path request "publish message on a tag" is issued,

1. The calling thread atomically checks the local tag information for valid information.
2. If the information is valid, it sends copies of the message to all receivers on the local node (if any) and at most one message to each remote delivery manager whose node has at least one receiver for that tag.

When the steady-state path request "receive message on a tag" is issued,

1. The calling thread atomically checks the local tag information for valid information.
2. If valid, a specified timeout is set and the client thread blocks waiting for a message. If a message is already pending, it returns with the message. The process id of the sender process and its node address are also available on return.
3. If no message is received within the specified time, the client returns with an error.

When a query call "obtain senders/receivers on a tag" is issued,

1. The client's local tag table is atomically checked and if the information is available locally, it is returned.
2. If the information is not locally available, the request is forwarded to the nearest IPC daemon for obtaining the information.

3.4. Meeting the Goals of the Design

We discuss below how the above design achieves the design goals outlined in the beginning of Section 2.

Ease of programming is achieved by providing location transparency, an identical programming model for unicast or multicast, and an identical programming model for uniprocessors or distributed systems. This programming ease can be easily seen in the benchmark code of Section 4 where the code does not change as the system grows from one processor to multiple processors as well as from one receiver to multiple receivers.

Portability is achieved by maintaining protocol independence and keeping the interface constant. For example, our real-time communication model was first implemented on a single node using only UDP sockets. Then, for performance reasons, the underlying communication protocol was

---

[3] An acknowledgement-based delivery mechanism is recommended here. However, the current implementation just uses datagrams.

ported to POSIX message queues but the user-interface was unchanged. Finally, the interface remained constant as support for communication across nodes was added with only the addition of an *optional* new parameter to the `Receive_Message` call to identify the source address of a message. As will also be seen in the next section, POSIX message queues are still used locally on a node and UDP sockets are used across nodes, but all these details are hidden from the programmer at the interface.

Scalability is achieved by ensuring that there will not be significant bottlenecks when the size of the system grows. The implementation is supported by interacting IPC daemons running on various nodes, but as the number of nodes grows in the system, the communication between the daemons can become expensive. Hence, the current implementation is designed such that one daemon need not be running on every node. The daemons can be running only on an acceptable subset of the nodes in the system. Only the delivery manager needs to be replicated on every node.

Analyzability is achieved by assigning various threads appropriate scheduling priorities and by providing a schedulability model of how the IPC interface operates. This is the key component from a timing predictability point of view. There are many threads of execution in client space *and* in daemon space, and it is critical that these threads run at appropriate priorities. The distribution tag information is also stored as data shared among these threads in each process, and unbounded priority inversion must be avoided on access to this data. More discussion on this topic is provided in Section 3.6.

Efficiency is achieved by ensuring that the steady state path or the real-time path is as efficient as possible. Any message sending, reception or purge only uses information available locally within the client space. Remote deliveries involve network communication, and hence delivery to multiple receivers on the same node are accelerated by a delivery manager which does the local duplication.

Protection and enforcement is obtained by making sure that no process but the IPC-daemons on various nodes actually can change the global publisher/subscriber information. Clients can at worst corrupt only the locally stored information but this information is not accessible beyond the client's address space. Unfortunately, a misbehaving client can still try to flood the network or a remote delivery manager, and protection against such behavior can only be provided at a lower layer at the OS network interface.

3.5. Implementation Aspects

The current implementation of the above design uses the following:

- POSIX threads [5] are used both in the IPC daemon and the client, which are scheduled using fixed priorities.
- POSIX semaphores are used for protecting the tag table shared information in the IPC daemon.
- POSIX message queues are used for local delivery of messages within a node including direct delivery by a client to other local receivers, and indirect delivery to local receivers by a delivery manager receiving messages from a remote client.
- All client-generated requests to the IPC daemon and their corresponding responses use UDP sockets such that an IPC daemon need not be present on every node.
- All communications which cross node boundaries also currently use UDP sockets.
- Each message that is sent from a client is pre-pended with header information which includes the process id of the sender and its node address. This information can be used by higher-level handshaking protocols at the receiver(s).

Current system initialization requires that IPC daemons be started on all the networked nodes before any application-level interprocess communication using the real-time publisher/subscriber model can commence. The list of nodes in the system is read from a file (identical copies of which have been installed on all the nodes) so that each IPC daemon and library knows what other daemons to contact for tag updates and message delivery.

3.6. Schedulability Analysis

From a schedulability analysis point of view for real-time systems, the implementation of the RT publisher/subscriber model has many implications, almost all of which are relatively well-understood. The implementation presumes a fixed-priority-driven preemptive scheduling environment but the model is only a library interface not bound to any particular application. Various real-time applications can use this model in ways they see fit. However, for these applications to be able to analyze the impact of the real-time publisher/subscriber model they use, sufficient information regarding the model's internal architecture, its cost and priority inversion characteristics need to be known. The model's internal architectural design and implementation have been discussed in detail in the preceding sections. For example, the threads which provide various services or execute various modules are part of the design discussion as are their priority assignments. The cost of various IPC calls is provided in Section 4.

The techniques of [9, 3] can be applied for uniprocessor analysis and the notions of remote blocking, deferred execution penalty and global critical sections (if a remote client/server application architecture is used) of [8] can be used for distributed system analysis. These techniques basically need two pieces of information: the amount of execution time for segments of code and the worst-case duration of priority inversion. Priority inversion can result due to the atomic nature of *local* tag table accesses in each client and in the IPC daemon. However, note that these shared resources are only shared within the confines of each client and the IPC daemon process. Hence, it is relatively easy to identify the maximum duration of critical section accesses and the cost is bounded by measurements in the next section.

The relatively tricky aspect is that a tag status update can generate asynchronous notifications to many client threads, and hence the true value of preemption time from higher priority tasks and blocking time for a task must be based on the knowledge of the maximum number of clients (senders and receivers) on tags used by lower or higher priority tasks. However, results from real-time synchronization help determine these values. At any given priority level, at most one lower priority task from that node can issue a tag request (causing asynchronous notifications to be sent to other tasks) since this priority level can preempt subsequent lower priority requests and tasks. However, requests from higher priority tasks must be accounted for appropriately.

3.7. Object Hierarchy

The real-time publisher/subscriber IPC model has been implemented in C++ (the current implementation has about 7000 lines of code) and is based on an extensive and flexible object hierarchy. Basic classes encapsulate many needed OS functions such as timers, semaphores and threads to enable easy porting to other operating environments. Other library classes implement queues, hash-tables, protected buffers and communication protocols such as sockets and POSIX message queues. Above these is built the tables to hold the tag information and messaging classes for constructing and decoding messages sent among the IPC managers, and between the client and an IPC daemon. At the top level are the classes for instantiating the update manager, the delivery manager and the local manager on the daemon side, as well as the update thread, the local request service and the delivery service on the client side. The actual managers and services are provided by simple instantiations of these classes. Each of these classes can be ported to other platforms with different OSs and/or communication protocols without affecting the other layered classes.

The key parameters used to control real-time performance in this context are scheduling priorities and access to shared resources. Hence, all classes which relate to schedulable entities (i.e. threads) and access to shared resources (e.g. semaphores) are parameterized such that priorities and semaphore accesses are under the control of instantiating entities.

4. Performance of the Publisher/Subscriber Model

The real-time publisher/subscriber communication model described in the preceding sections has been implemented and preliminary benchmarks obtained on a network of i486DX-66MHz PCs running a POSIX-compliant real-time operating system LynxOS Version 2.2 with 32MB of memory. The network used was a dedicated 10Mbps ethernet but was very lightly loaded.

4.1. The Rationale Behind the Benchmarking

As discussed in Section 3.6, we attempted to measure the cost of each call that a client can make with the real-time publisher/subscriber model. We also tried to validate how far we met one of the design goals of efficiency compared to the traditional "build-from-scratch" communication facilities, despite the unoptimized nature of our prototype implementation and the optimized nature of the underlying UDP/IP and POSIX message queue communication primitives. In all measured cases, we published messages on a tag and measured the time it takes to get its response on another tag. This allows us to measure the elapsed time from a single point of reference, namely the sending end. For the sake of convenience, this is referred to as "sender-based IPC benchmarking". By sending multiple messages, sufficient elapsed time can be accumulated to eliminate any clock granularity problems.

"Sender-based IPC benchmarking" is conceptually simple in the case of either local or remote unicast: (a) send a message, (b) wait for a response, (c) repeat this several times, (d) measure the total time elapsed, (e) and compute the time elapsed per round-trip message. In the case of remote unicast, sender-based benchmarking also avoids the need for a tightly synchronized clock across nodes. With multicast in general and remote multicast in particular, the situation is trickier. If all multicast recipients respond with a unicast to the sender, the elapsed time contains a large mix of multicast and unicast messages. We measured this to use as a data point as well. However, if only one recipient responds with an *ack* message, then the total cost is dominated by the multicast and it is easy to identify the cost benefits of multicast. However, it is possible that the *ack* message returns concurrently before the multicast send has completed and the elapsed time measurement is distorted. We resolve this by multicasting a message to many receivers but only the lowest priority receiver responds with an *ack*. This ensures that the multicast send completes before the *ack* originates.

4.2. The Benchmark Code

The performance benchmark code on the sender and receiver sides for benchmarking is illustrative of the ease, power and compactness of using the real-time publisher/subscriber communication model. All benchmarks (local and remote unicast as well as local and remote multicast) are carried out with this code. The sender side is presented in Figure 4-1 and the receiver side is presented in Figure 4-2.

4.3. Performance of *vanilla* UDP/IP and POSIX message queues

It must be noted that the context-switching overhead from one process to another process is about 100 microseconds.

Figure 4-3 shows the control case that does *not* use the real-time publisher/subscriber model. There is one sender and one receiver. The sender sends a message and the

```
int benchmark_dist_tag_ipc( char *msg, int msglen,
                  int num_messages, int num_acks )
{
  DistTag_IPC ipc; timeval_t start, end;
  int i, j;

  // get send rights to receivers' tag
  // (tag is currently an ascii string)
  ipc.Get_Send_Access( RECEIVER_TAG );

  // get receive rights for getting acks back
  ipc.Subscribe( ACK_TAG );

  // get time-of-day before sending messages
  getCurrentTime( &start );

  // send out messages
  for ( i = 0; i < num_messages; i++ ) {
    // send message and wait for a response
    ipc.Send_Message( RECEIVER_TAG, msg, msglen );

    for ( j = 0; j < num_acks; j++ ) {
      // block on required acks
      ipc.Receive_Message(ACK_TAG, msg, msglen, NULL));
    }
  }

  // get time-of-day we stopped
  getCurrentTime( &end );

  // compute delay per message sent
  // and print it out
  ...
}
```

Figure 4-1: Benchmarking Code on the "Sender" Side

```
run_receiver_in_loop( char *msg, int msglen, int ack )
{
  DistTag_IPC ipc;

  // create tags: no problem if already created
  ipc.Create_Distribution_Tag( RECEIVER_TAG );
  ipc.Create_Distribution_Tag( ACK_TAG );

  // get receive rights on RECEIVER_TAG
  ipc.Get_Send_Access( ACK_TAG );

  // get send rights on ACK_TAG
  ipc.Subscribe( RECEIVER_TAG );

  while ( True ) {
    // block for messages waiting forever
    ipc.Receive_Message( RECEIVER_TAG,msg,msglen,NULL);

    if ( ack ) {
       // send ack to "ack tag"
       ipc.Send_Message( ACK_TAG, msg, msglen );
    }
  }
}
```

Figure 4-2: Benchmarking Code on the "Receiver" Side

receiver returns a response using UDP/IP or POSIX message queues. Both sender and receiver are on the same node in columns 2 and 3 but are on different nodes in column 4. As can be seen, UDP protocol processing is costly relative to POSIX message queues but the latter work only within a node. This is the reason why we ported the local message delivery mechanism from UDP to POSIX message queues (without interface changes as intended).

Message Size (bytes)	Round-trip delay on same node using POSIX message queues (ms)	Round-trip delay on same node using UDP messages (ms)	Round-trip delay from different nodes using UDP messages (ms)
32	0.3191	1.488	2.200
64	0.3368	1.481	2.400
128	0.3552	1.556	2.762
256	0.4050	1.633	3.569
512	0.4899	1.657	5.190
1024	0.6654	1.856	8.355
2048	0.8808	2.604	13.200
4096	1.4306	3.501	21.980
8192	2.6490	6.013	39.700

Figure 4-3: Basic System IPC Costs

4.4. Publication/Subscription on a Single Node

Figure 4-4 presents the costs involved in multicast to multiple receivers on the same host node (when *all* receivers ack back to the sender) for a 256-byte message. A sender with lower priority than all the receivers (columns 2 and 3) performs better than having the sender with higher priority than all the receivers (columns 4 and 5) because the sender never blocks in the former case. Also, emulating the ceiling priority protocol (columns 2 and 5) to avoid the cost of semaphores to access the local tag table atomically (columns 3 and 4) helps to improve performance.

# of receivers	Low priority sender; *no* semaphores (ms)	Low priority sender; semaphores (ms)	High priority sender; semaphores (ms)	High priority sender; *no* semaphores (ms)
1	0.763	0.847	0.832	0.739
2	1.534	1.71	1.774	1.467
3	2.563	2.679	2.907	2.207
4	3.648	3.882	4.045	3.199
5	4.945	5.239	5.581	4.445
6	-	-	7.167	5.802

Figure 4-4: Round-Trip IPC Costs For Multicast on a single node.

Comparing the control case (Figure 4-3 column 2) with the case for 1 receiver, we see that our IPC model has roughly added at least 340 μs to a direct POSIX message queue send (0.4050 *vs* 0.739 ms for 256 byte messages) for a single receiver and this scales almost linearly as the number of receivers grows. We attribute this cost to the additional copying done by our implementation.

In the rest of this section, we use semaphores for critical section access so that the numbers are slightly higher than the performance obtained with emulating the ceiling priority protocol. Figure 4-5 presents the multicast cost on a single node when only the lowest priority receiver acks. It is surprising to see that a sender with higher priority than

# of receivers	Low priority sender (ms)	High priority sender (ms)
1	0.83	0.829
2	1.37	1.173
3	1.847	1.522
4	2.259	1.831
5	2.762	2.254
6	3.178	2.597
7	3.553	2.877
8	4.096	3.181

Figure 4-5: Round-Trip IPC Cost For Multicast on a single node.

the receivers performs better than a sender with lower priority than the receivers, the exact opposite of before! This is because of the tradeoff between increasing the number of context switches and letting one process block waiting for a message that has not yet arrived.

4.5. Subscription from a Remote Node

Figure 4-6 gives the costs incurred when multicast is performed to a remote node with 256-byte messages. Compare Figure 4-3 column 4, 256-byte message cost (3.569 ms) with Column 3, 1 receiver cost of Figure 4-6 (6.72ms). The added overhead of our library front-end and its sending a message to a delivery manager, which in turn delivers to local receivers has added an overhead of more than 3ms. However, with only two receivers on the remote node, two plain UDP messages sent across the network would have cost (3.569*2 = 7.138 ms) while the delivery manager is able to deliver two copies locally to accomplish the same function at 7.18ms, i.e. the performance crossover occurs only with 2 receivers on a remote node.

Additional receivers on the remote node cost only 0.5ms each (with a single ack), while direct duplicate communication would not only load the network but also cost an additional 3.569 ms. Even our unoptimized prototype is better than a straightforward implementation which sends network messages to each remote receiver, while at the same time providing a much simpler and more powerful interface that deals with both unicast and multicast simultaneously, whether local or remote. This is a significant observation with respect to our goals of ease of programming, scalability and even efficiency, despite the unoptimized state of the current implementation. Note that this enhancement is obtained by using UDP sockets across the network (the most efficient at that level) and POSIX message queues locally on the receiving side (the most efficient within a node as seen in Figure 4-3).

# of receivers	All receivers acknowledge (ms)	Only low priority receiver acks (ms)
1	6.68	6.72
2	8.58	7.18
3	10.57	7.54
4	12.88	7.98
5	15.28	8.55
6	17.86	9.15
7	18.32	9.43
8	21.52	-

Figure 4-6: Round-Trip IPC Costs For Multicast to multiple receivers on one *remote* node.

4.6. Subscription from Multiple Remote Nodes

Figure 4-7 presents the costs involved for 256-byte messages when there can be upto 3 receivers on each of 3 remote nodes "laurel", "hardy" and "chaplin". Messages are initiated from the node "shemp". *All* receivers must ack to the sender. As can be seen, if there is one receiver on each of the 3 remote nodes, the round-trip cost is around 12.5 ms. Also, the costs do not scale linearly because of some pipelining concurrency that happens naturally across nodes. This is clearly seen in rows 3 and 4 where there are 1 or 2 receivers on a remote host "laurel", but the total difference in cost is almost negligible. This is because the responses from the receivers on "laurel" happen in parallel with the messages sent to receivers on hosts "hardy" and "chaplin".

# of receivers on host “laurel”	# of receivers on host “hardy”	# of receivers on host “chaplin”	Round-trip delay per published message on “shemp” (ms)
1	0	0	6.74
1	1	0	8.55
1	1	1	12.5
2	1	1	12.68
2	2	1	15.28
2	2	2	17.79
3	2	2	20.17
3	3	2	22.78

Figure 4-7: Round-Trip IPC Costs For Multicast to multiple recipients on many remote nodes.

4.7. Lessons Learned

We have gained some valuable experience and insights into the communication model with our prototype implementation. As mentioned earlier, this communication model has been successfully used in implementing inter-processor communication in the Simplex architecture [10]. Due to the expense of UDP sockets in local node communications, as per Figure 4-3, we re-implemented the interface using POSIX message queues. Since the interface could be re-implemented without any changes, it demonstrated that the goals of interface portability and performance optimization are not mutually exclusive.

The performance of the current implementation needs to be further optimized, however. We pay a performance penalty because of additional message copying necessitated by the need to insert and then delete message headers for use by the communication model. Currently, tag information is also stored in a hash-table and the hashing cost is relatively high. We therefore plan to eliminate strings for tags from being hashed once a tag has been created. However, despite these current inefficiencies, it is encouraging from Section 4.5 to see that the current implementation can actually do better for multicast to remote nodes and the improvement actually grows larger as the number of receivers increases.

Another alternative to consider is the use of shared memory segments to store the tag table information for sharing among clients and the IPC-daemon on a node. While this does not affect the steady state path significantly, it can make the implementation much simpler. However, two disadvantages would arise. First, with a single piece of shared memory having all the information, protection and failure issues need to be addressed. Secondly, as the system grows, IPC daemons must run on all nodes to update the shared memory information.

5. Concluding Remarks

The construction of distributed real-time systems poses many challenges because of the concurrent need to satisfy many attributes including ease of programming, portability, scalability, analyzability, efficiency and protection. We have sought to identify how many of these attributes can be dealt with in the context of inter-process communications, a fundamental component of any distributed system. We have identified the real-time publisher/subscriber model as being an excellent candidate in this domain. We also presented a programming interface which is quite simple based on the notion of a distribution tag. The interface has been tested with the Simplex architecture, and has remained stable with needed optimizations done without affecting the interface. We also provided a detailed design of the model, an implementation and benchmarks (of Section 4.5 in particular) to show that many attributes can indeed be satisfied.

Despite these positive results, much work remains to be done. The current prototype does not tolerate processor failures even though the current design and code is structured such that these changes can be incorporated relatively easily. In addition, redundant networks need to be supported to tolerate network failure. Finally, support for flexible yet real-time transfer of state between new and existing IPC daemons in different states is critical.

References

1. F. Cristian. "A probabilistic approach to distributed clock synchronization". *Distributed Computing 3* (1989), 146-158.

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

3. Klein, M. H., Ralya, T., Pollak, B., Obenza, R. and Harbour, M. G. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems.* Kluwer Academic Publishers, 1993. ISBN 0-7923-9361-9.

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

5. *IEEE Standard P1003.4 (Real-time extensions to POSIX).* IEEE, 345 East 47th St., New York, NY 10017, 1991.

6. Meyers, B. C. IEEE POSIX P1003.21: Requirements for Distributed Real-time Communications. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, October, 1994.

7. Rajkumar, R., Sha, L., Lehoczky, J.P. "An Experimental Investigation of Synchronization Protocols". *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software* (May 1988).

8. Rajkumar, R. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, 1991. ISBN 0-7923-9211-6.

9. Sha, L., Rajkumar, R. and Lehoczky, J. P. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". *IEEE Transactions on Computers* (September 1990), 1175-1185.

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

Table of Contents

List of Figures