Design and Performance of an Object-Oriented Framework for High-Speed Electronic Medical Imaging[*]

Irfan Pyarali Eastman Kodak Company
Timothy H. Harrison and Douglas C. Schmidt
Washington University

ABSTRACT: This paper describes the design and performance of an object-oriented communication framework being developed by the Health Imaging division of Eastman Kodak and the Electronic Radiology Laboratory at Washington University School of Medicine. The framework is designed to meet the demands of next-generation electronic medical imaging systems, which must transfer extremely large quantities of data efficiently and flexibly in a distributed environment. A novel aspect of this framework is its seamless integration of flexible high-level CORBA distributed object computing middleware with efficient low-level socket network programming mechanisms. In the paper, we outline the design goals and software architecture of our framework, describe how we resolved design challenges, and illustrate the performance of the framework over high-speed ATM networks.

*This research is supported in part by the Eastman Kodak Company Health Imaging division and the Electronic Radiology Laboratory at Washington University, St. Louis.

1. Introduction

The demand for distributed electronic medical imaging systems (EMISs) is pushed by technological advances and pulled by economic necessity [Blaine et al. 1994]. Recent advances in high-speed networks and hierarchical storage management provide the technological infrastructure needed to build large-scale distributed, performance-sensitive EMISs. Consolidating independent hospitals into integrated health care delivery systems to control costs provides the economic incentive for such systems.

Two key requirements for the communication infrastructure in a distributed EMIS are *flexibility* and *performance*. An EMIS must be flexible in order to transfer many types of message-oriented and stream-oriented data (such as HL7, DICOM, and domain-specific objects) across local and wide area networks. EMIS requirements for flexibility motivate the use of distributed object computing middleware such as CORBA [Object Management Group 1995] in the communication infrastructure. CORBA automates common network programming tasks (such as object selection, location, and activation, as well as parameter marshalling and framing), thereby enhancing application flexibility.

However, empirical studies [Schmidt et al. 1995.18; Gokhale & Schmidt 1996.7 & 1996.8] reveal that for bulk data transfer, the performance overhead of widely used CORBA implementations on high-speed ATM networks is 25% to 70% below that achievable using lower-level transport layer interfaces such as sockets or TLI. As high-speed networks like ATM, FDDI, and 100 Mbps Fast-Ethernet become ubiquitous, this performance overhead will force programmers to use lower-level mechanisms to achieve the necessary transfer rates, rather than adopting distributed object computing technologies. This is particularly problematic for performance-intensive application domains like medical imaging, where the use of low-level tools increases development effort and reduces system reliability and flexibility.

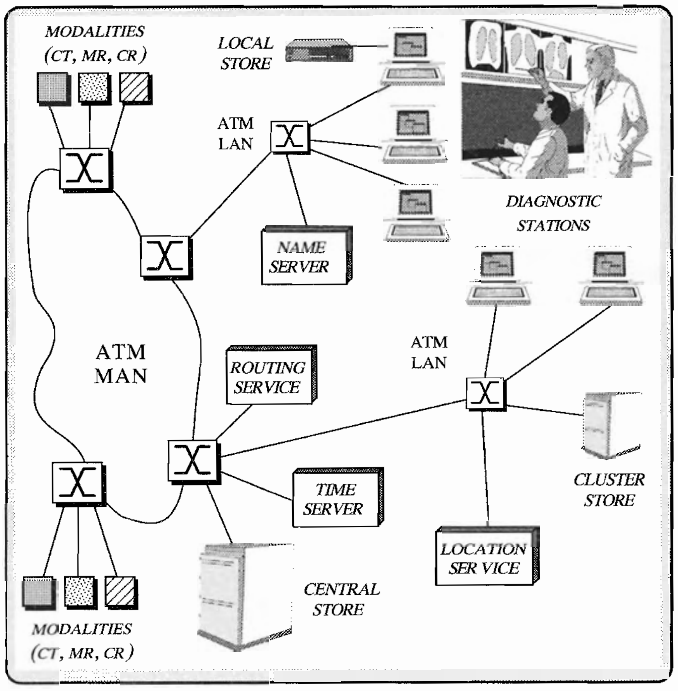

Figure 1. Topology of Distributed Objects in Project Spectrum.

To address this problem, we have developed an object-oriented communication software framework called *Blob Streaming*.[1] The Blob Streaming framework is designed to meet the requirements of next-generation electronic medical imaging systems (EMISs). Figure 1 illustrates the topology of our distributed EMIS environment [Blaine et al. 1994]. In this environment, various types of modalities (such as CT, MR, and CR) capture patient images and transfer them as Blobs to an appropriate storage management system (called a Blob Store). Radiologists use diagnostic workstations to retrieve these images for viewing and interpretation. In addition to medical images, next-generation EMISs must support multimedia Blobs such as video streams and audio diagnostic reports.

The Blob Streaming framework provides a uniform interface that enables EMIS developers to flexibly and efficiently operate on multiple types of Blobs located throughout a large-scale health delivery system. This framework combines the flexibility of high-level distributed object computing middleware (e.g., CORBA) with the efficiency of lower-level transport mechanisms (e.g., sockets).

1. Blob stands for "Binary Large OBject."

Developers of communication software for EMIS environments have traditionally had to choose between (1) high-performance, lower-level interfaces provided by sockets, or (2) less efficient, higher-level interfaces provided by communication frameworks like CORBA. Blob Streaming represents a midpoint in the solution space. It improves the correctness, programming simplicity, portability, and reusability of performance-sensitive EMIS communication software. Blob Streaming leverages the flexibility of CORBA, while its performance remains competitive with applications programmed at the socket level.

This paper is organized as follows: Section 2 motivates the design of the Blob Streaming framework, outlines the key design challenges, and describes how we resolved these challenges; Section 3 illustrates how the Blob Streaming framework has been used to build high-performance image transfer applications; Section 4 compares the performance of Blob Streaming with alternative C, C++, and CORBA approaches over a high-speed ATM network; Section 5 discusses recommendations based on our results; and Section 6 presents concluding remarks.

2. *Design of the Blob Streaming Framework*

2.1. *Blob Streaming Architecture*

The Blob Streaming framework is designed to minimize excessive layering to improve performance, while still allowing applications to be decoupled from communication details that are prone to change. This decoupling helps increase portability and enables transparent optimizations without altering public Blob Streaming interfaces. The shaded portion of Figure 2 illustrates the architecture of the Blob Streaming framework, which consists of the following layers:

- C++ wrapper layer: This layer uses an existing toolkit of C++ wrappers [Schmidt 1994.16] that shield applications from the details of the lower layer C library and OS system call mechanisms. These mechanisms include sockets and CORBA for interprocess communication, memory-mapped file wrappers for optimized secondary storage access, and event demultiplexing. The use of C++ wrappers provides strongly typed interfaces that simplify the development of Blob Streaming. For example, porting to alternative platforms requires no changes to Blob Streaming software because the Blob Streaming library does not directly access any OS specific interfaces. Currently, Blob Streaming is implemented on many versions of UNIX, as well as Win32 platforms.

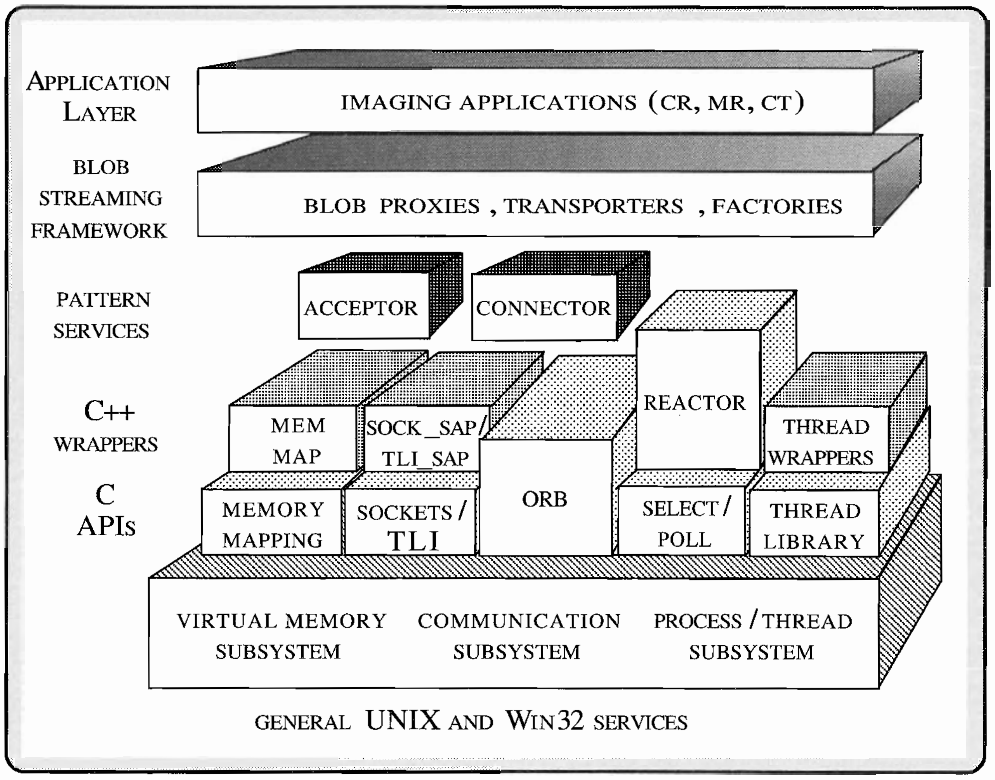

Figure 2. Layering Architecture of the Blob Streaming Framework.

- Common services layer: This layer uses an existing framework [Schmidt 1994.16] of strategic design patterns [Schmidt 1996.20] that enhance framework quality by providing reusable communication system components. For instance, Blob Streaming uses component implementations of the Acceptor and Connector patterns [Schmidt 1996.21] that decouple the passive and active initialization of services from the tasks performed once the services are initialized. Likewise, the component implementation of the Reactor pattern [Schmidt 1995.17] simplifies event-driven applications by associating event handler objects to the demultiplexing of events. The use of these patterns and components in the Blob Streaming framework leverages prior design efforts and reduces software development risks.

- Blob Streaming layer: This layer provides application developers with the Blob Streaming components that provide generic interfaces for high-speed Blob transfer. The main components include Blob *Proxies*, *Transporters*, and *Factories*:

- Proxies—which use the Bridge and Proxy patterns [Gamma et al. 1995] to represent location- and type-independent handles to Blobs. These patterns provide a surrogate that shields clients from knowledge of where the Blob resides, thereby making it easy to vary the location without affecting client code.
- Transporters—which use the Strategy pattern [Gamma et al. 1995] to represent location- and type-independent algorithms that perform optimal transfer of Blobs between sources and destinations. The Strategy pattern lets the algorithms vary independently from clients that use them.
- Factories—which use the Factory pattern [Gamma et al. 1995] to decouple Proxy creation from Proxy use. A Factory performs the work necessary to build a Proxy, such as using a location service to find the Blob within the EMIS.

A key design goal of the Blob Streaming layer is to provide operations that behave uniformly irrespective of where the Blob actually resides or what type of Blob is being transferred. For instance, Blob Store software that receives and stores MRI images to a database remains unchanged whether the source or destination of the MRI data is in memory, on a local file, in memory of a remote client, or on disk of a remote client.

2.2. Resolving Design Challenges

Developing an enterprise-wide distributed EMIS is difficult. It requires a deep understanding of networking, databases, distributed systems, human/computer interfaces, radiological workflow, and hospital information systems. There are many technical challenges related to performance, functionality, high availability, information integrity, and security. Moreover, system requirements and the hardware/software environment change frequently.

To cope with complexity and inevitable changes, the software infrastructure of an EMIS must be flexible. In particular, developing large-scale distributed EMIS applications with low-level network programming tools like sockets is tedious, error-prone, and inflexible. Therefore, we designed Blob Streaming to elevate the level of programming for these applications. To accomplish this, we abstracted away from the following tasks and mechanisms in the Blob Streaming design:

- Common network programming tasks
- Blob location and storage mechanisms
- Blob type

- Blob transport mechanism
- Concurrency policies
- Multiple event loops
- Platform-specific OS mechanisms

This section describes the software design challenges we faced when developing the Blob Streaming framework for EMIS applications. The following explains how we resolved these challenges using object-oriented design techniques, design patterns, and C++ language features. Although the discussion centers around issues that arise when building medical imaging frameworks, the principles and patterns described below are representative of a wide range of bandwidth-intensive distributed object computing environments.

2.2.1. Abstracting Away from Common Network Programming Tasks

Many low-level programming tasks (such as object location and activation, parameter marshalling and framing) performed when building distributed applications are tedious and error-prone. The current version of Blob Streaming uses CORBA to automate these common low-level network programming tasks. The use of CORBA enabled us to concentrate on higher-level Blob Streaming issues (such as performance, reliability, and interface uniformity), rather than wrestling with low-level communication details. We used the following CORBA mechanisms to implement the Blob Streaming framework:

- Strongly-typed interfaces: In CORBA, all interfaces are defined using the CORBA interface definition language (IDL) [Object Management Group 1995]. A CORBA IDL compiler generates stubs and skeletons that translate IDL interface definitions into C++ classes. For instance, IDL interface definition in Figure 3 describes a `BlobTransporter` that is used internally by the framework to control Blob transfer from a server to a client. Client applications use the `BlobTransporter` to selectively request certain sections of a Blob. The ability to randomly access Blobs has several uses, including (1) the ability to efficiently access header information from a Blob, or (2) resuming an interrupted transaction without restarting from the beginning.

  The use of CORBA IDL interfaces allows the transmission of strongly-typed data across the network. Strong typing improves abstraction and eliminates errors common to socket-level programming. For instance, if the `send` and `recv` operations shown above were implemented over a socket connection, we would need to manually convert the typed information into a stream of untyped bytes. Moreover, the sender and receiver software for

```
interface BlobTransporter {
  // Timeout value representation.
  struct TimeValue { long sec; long usec; };

  // Transaction notification options.  These
  // options allow the framework to control blob
  // transfers acknowledgments.
  enum NotificationSemantics {
    SEND_NOTIFICATIONS,
    QUEUE_NOTIFICATIONS,
    IGNORE_NOTIFICATIONS
  };

  // A request to the server to send <length> bytes
  // of Blob data starting from <absoluteOffset>.
  // Since this can potentially be a long-duration
  // operation, a <timeout> can also be specified.
  // The <semantics> vary depending on the reliability
  // required.
  oneway void send (in long length,
                    in long absoluteOffset,
                    in boolean useTimeout,
                    in TimeValue timeout,
                    in NotificationSemantics semantics);

  // Informs the server to receive <length> bytes of
  // Blob data.  This data is copied to the Blob
  // starting at <absoluteOffset>.  Other options
  // are similar to send().
  oneway void recv (in long length,
                    in long absoluteOffset,
                    in boolean useTimeout,
                    in TimeValue timeout,
                    in NotificationSemantics semantics);
  // ... others omitted...
```

Figure 3. IDL Interface for Blob Transport.

parsing messages must be tightly coupled to ensure correctness. Since this provides many opportunities for errors, automating this process via CORBA significantly improves system robustness.

- Parameter marshalling and framing: CORBA IDL compilers automatically generate client-side stubs and server-side skeletons. These stubs and skeletons ensure correct byte ordering and linearization of all parameters sent via operation calls on CORBA interfaces over a network. For instance, the `send` and `recv` operations in the IDL `BlobTransporter` interface shown above pass various types of binary parameters. The IDL compiler maps these parameters into C++ data types such as `char` for the IDL `boolean` type and a C++ `struct` containing two `long` fields for the `TimeValue` parameter.

Marshalling the `BlobTransfer` parameters manually and then using sockets would require copying the parameter values into a transfer buffer and performing a `send`. We would also have to convert the representation of the `longs` from host-byte order to network-byte order. In addition, if the bytestream-oriented TCP/IP was used, we would be responsible for framing the data correctly at the receiver. Marshalling and framing are two tedious and error-prone aspects of network programming. By using CORBA, we did not need to implement these low-level operations.

- Object location and object activation: CORBA supports location transparency, i.e., services can be located anywhere in a distributed system. Therefore, objects accessed by clients can be remote, local (on the same host) or co-located (in the same address space). We used this feature of CORBA in the Blob Streaming framework to shield applications from the location of Blob Stores where a Blob of interest resides. Since CORBA interfaces are location independent, the framework invokes operations on Blob Stores without knowledge of where the server resides. As a result, applications that use Blob Streaming also have no dependencies on Blob Store locations.

  Blob Streaming also takes advantage of CORBA's activation services. Orbix can be configured such that if a request is received for a non-active server, the a server can be launched to process the request. This allows Blob Stores to be started by Orbix only when they're needed, thus conserving system resources.

2.2.2. *Abstracting Away from Blob Location and Storage*

The location of Blobs can vary significantly. Blobs may exist in the memory of a modality (such as an Ultrasound scanner), on the local disk of a radiologist's workstation, or in a remote Blob Store. To provide adequate reliability, availability, and performance a large-scale EMIS must support a range of Blob Stores. As shown in Figure 1, these include the following:

- Central Stores—which provide hierarchical storage management and support long-term archiving of Blobs.

- Cluster Stores—which cache Blobs within a cluster of diagnostic workstations in a local area network in order to increase system fault tolerance and decrease load on Central Blob Stores.
- Local Store—which cache Blobs on the local disk of a diagnostic (DX) workstation.
- Memory Stores—which cache Blobs in workstation memory.

In addition to the Blob Stores listed above, new implementations of Blob Stores can be created for more advanced data storage. For example, a *Database Store* might be designed to manage Blobs in a database (e.g., Oracle, Sybase, or ObjectStore) and an *Archival Store* can be implemented to maintain legacy data to comply with legal statutes on image persistence.

To enhance the system usability, images must be presented to the radiologist quickly. For instance, consider the case of presenting MR images to a radiologist on a diagnostic workstation. The Blob Streaming framework is responsible for selecting the optimal transfer technique for this task. If images are stored in files on the local Workstation Blob Store, Blob Streaming memory maps the files, thereby avoiding excessive mode switches and read/write buffering. If Blobs do not reside locally, they must be found using name servers and locators [Object Management Group 1994]. Once found, they must be transported to the radiologist's workstation for display and interpretation.

Before being displayed, however, Blobs may need to be processed (e.g., magnified, rotated, and edge-enhanced) for optimal presentation. Due to the wide range of stores that Blobs can reside, Blob Streaming allows application software that operates on Blobs to be developed independently of the Blob's location.

The component in Blob Streaming that facilitates location abstraction is the *Blob Proxy*. The Blob Proxy defines the interface visible to clients. All requests to the Blob Proxy are forwarded to the Slot object, which is the abstract class that defines the interface for implementation classes. Figure 4 shows the multiple specializations of the Slot class such as socket, memory, file, and database. This design is an example of the Proxy and Bridge patterns [Gamma et al. 1995], where the Blob interface is decoupled from its implementation so that the two can vary independently. Section 3.1 describes the Blob Proxy programming interface in greater detail.

The Blob Streaming framework can be extended by adding new Slot implementations. The separation of interface from implementation allows these extensions to be transparent to code that uses the Blob Streaming framework. This

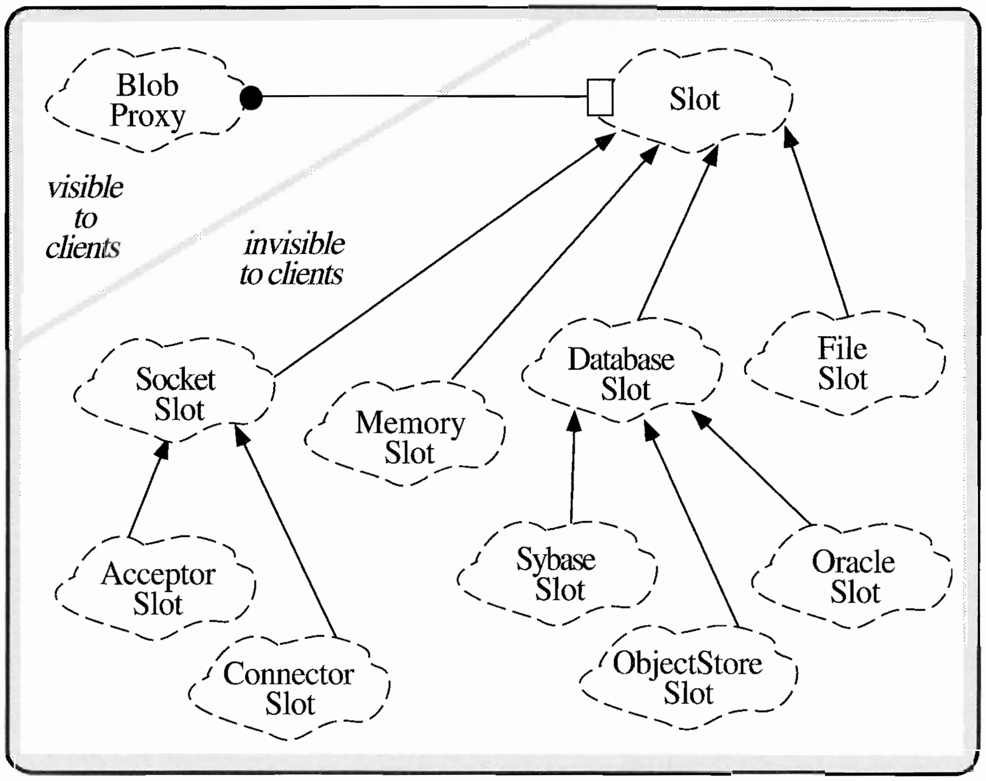

Figure 4. Blob Proxy and the Slot Hierarchy.

separation also enables the Blob Streaming framework to use the same Slot implementation instance for different Blob Proxies.[2]

The advantage of defining a uniform Blob Proxy interface is to *reduce software dependencies*. Using this generic interface, application software can be written to store and retrieve images from Blob Stores, rather than to files or databases directly. This shields existing software from changes in storage type. A disadvantage to this approach is the increased learning curve. For example, developers of Blob Servers who are familiar with a particular database must learn the Blob Store interface in order to use Blob Streaming. Therefore, as discussed in Section 3.1, the Blob Streaming interface was modeled after the UNIX file system interface, which provides a uniform set of operations (like `open`, `close`, `read`, `write`, `seek`, etc.) on various types of devices, files, and I/O streams.

2. This approach is used by some CORBA implementations like Orbix where multiple proxies use the same socket channel to communicate with a server. Some slots that take relatively long to setup (such as socket slots) can be cached internally to the library and can be reused by new Blob Proxies.

2.2.3. *Abstracting Away from Blob Type*

In addition to shielding application software from Blob location, the Blob Streaming framework abstracts away from Blob *type*. Therefore, a Blob Store that receives and stores MR images uses the same software to receive and store CT and CR images. The type of data being transferred is not directly exposed by the Blob Streaming interface.

The primary advantage of decoupling Blob type from Blob transfer is to *maximize software reuse* and *enhance interface uniformity*. In addition, our design allows meta-data (such as image identification information including patient name and examination data) to be separated and stored in a database. This decoupling allows image data (pixels) to be transported as fast as possible to the destination (e.g., using memory-mapped I/O and DMA). If an application requires access to the image's meta-data, complex queries can be performed on the database.[3]

The Blob Streaming framework is similar to the abstraction provided by an OS file system. The file system supports a variety of file formats. It is up to the application using the file to correctly interpret the file format. However, the type of abstraction offered by Blob Streaming is not available in other medical imaging toolkits (such as DICOM and HL7). Many such toolkits only transfer data formatted according to the protocol's specification. This becomes a problem when trying to extend a project to deal with new data types or when trying to optimize performance.

Another advantage of the Blob Streaming design is that it allows the integration of image processing and Blob transfer operations. Applications need not wait for an entire Blob to transfer before processing the data (e.g., compressing it as it is sent on the network and decompressing while being received). This technique is a form of Integrated Layer Processing (ILP) [Clark & Tennenhouse 1990], which has been used in high-speed communication protocol stacks. ILP optimizations can improve performance significantly by overlapping communication and computation, as well as reducing memory bus traffic.

2.2.4. *Abstracting Away from Blob Transport Mechanism*

Blob Streaming presently uses a combination of CORBA and TCP/IP as data transport mechanisms. CORBA is used for location and control operations, whereas TCP/IP is used for bulk data transfer. This design choice reflects a tradeoff between flexibility and efficiency. Blob Streaming leverages CORBA's

3. Consistency management between pixel store and database entries are considered outside the scope of the Blob Streaming framework. Certain image formats (e.g., DICOM) place meta-data as header information of the Blob. Since Blob Streaming treats all Blobs as untyped streams of data, images with integrated meta-data also can be transferred easily.

abstraction and flexibility, while still utilizing the efficiency of socket programming.

To shield applications from these low-level communication details, however, the public interface of Blob Streaming does not expose its internal transport mechanisms. This allows changes in the Blob Streaming architecture without affecting public interfaces. In particular, since CORBA is not visible to application programmers, different implementations of CORBA can be used (such as ORBeline, HP ORB Plus, or Sun NEO). Moreover, CORBA can be removed entirely and replaced with another mechanism (such as DCOM, DCE RPC, or Sun RPC). We chose CORBA since other parts of our EMIS use CORBA services like the COS Naming and Events [Object Management Group 1994]. Selecting a common distributed object computing framework reduced our training, maintenance, and software licensing costs.

Similarly, multiple transport mechanisms can be used to transfer bulk data efficiently. For instance, certain types of traffic (such as video and voice) can tolerate some degree of loss. In these cases, performance can be optimized by using a lightweight ATM protocol in place of TCP/IP. Since Blob Streaming provides a layer of abstraction over these details, optimizations can be performed without altering applications.

The primary advantages of decoupling the Blob Streaming public interface from its internal transport mechanisms are to *improve flexibility*, *increase portability*, and *enable transparent performance tuning*. Therefore, the framework can be tuned to use the best performing technology without affecting applications. For instance, subsequent versions of Blob Streaming could omit TCP/IP in favor of a strictly CORBA implementation if CORBA becomes performance-competitive with lower-level sockets programming. Likewise, CORBA could be replaced by DCOM and TCP/IP replaced by a lightweight ATM protocol. Figure 5 shows the communication layers currently used by Blob Streaming. In addition, it illustrates the different IPC and network layer choices that can be used as alternatives.

One disadvantage with Blob Streaming is performance overhead of the extra levels of abstraction. Although the cost of these abstractions can be reduced through optimizations such as C++ inlining, some overhead remains, as shown in Section 4. Another disadvantage is the increased complexity of the Blob Streaming internal design. In particular, connection management and synchronization are more complex. However, the complexity is not exposed to applications, which use the simple Blob Proxy interface provided by the Blob Streaming framework.

2.2.5. *Abstracting from Concurrency Policies*

Different applications require different types of operation invocation semantics from a framework. For instance, a multi-threaded server can simplify application

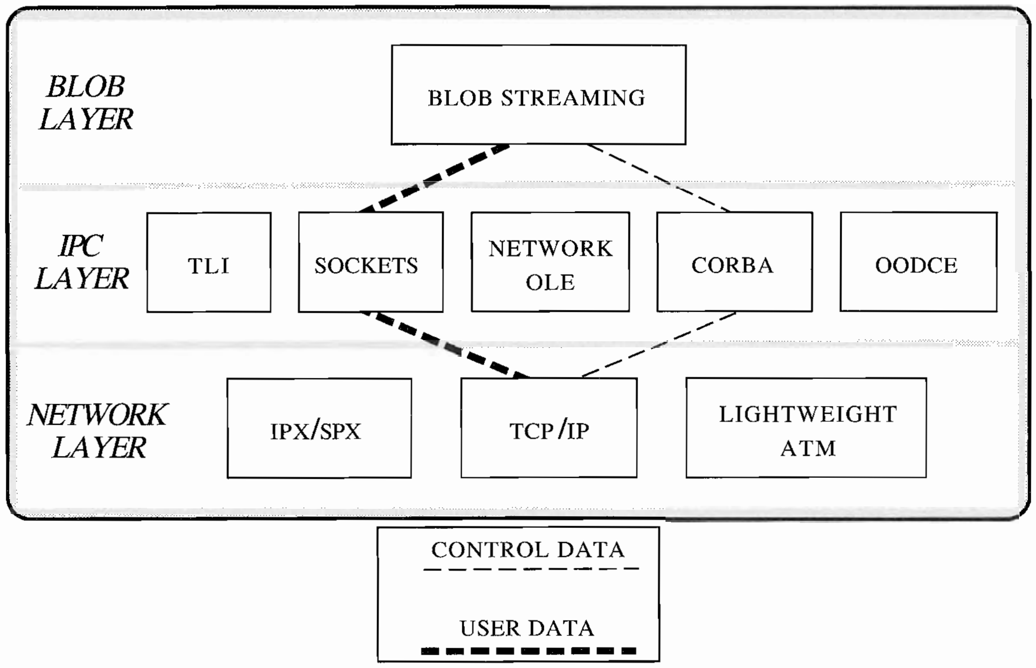

Figure 5. Communication Layers used by Blob Streaming.

software by using synchronous interfaces. Conversely, a single-threaded server that cannot afford to block on a single transaction needs an asynchronous interface to all long-duration operations. Similarly, client applications are frequently single-threaded and event-driven (e.g., GUIs), which cannot block indefinitely on synchronous calls.

On multi-threaded operating systems like Solaris 5.x [Eykholt et al. 1992] or Windows NT [Custer 1993], applications can use threads to simplify programming and take advantage of parallelism. A multi-threaded application can use synchronous interfaces for long-duration operations (such as large image transfers) since it will not block other threads. In contrast, single-threaded applications must be programmed carefully to avoid starving time-critical operations by blocking on long-duration operations.

Tightly coupling an application to a particular concurrency policy increases development effort if the concurrency policy changes (e.g., if a single-threaded application becomes multi-threaded or vice versa). It is hard to avoid this tight coupling because reusable frameworks and applications often must be developed without knowledge of the end system concurrency policies or hardware/software capabilities.

The Blob Streaming framework is designed to separate application software from dependencies on concurrency policies. Blob Streaming accomplishes this by providing uniform callback-driven interfaces to both synchronous and asynchronous operations. Switching between synchronous/return-value and asynchronous/callback interfaces can require modifications to application software. For instance, consider the case where a server implemented using multiple threads is ported to a platform that does not support threads. If the software run by the threads uses synchronous interfaces, many changes will be necessary to support asynchronous transactions in a single thread.

To improve portability and uniformity, the Blob Streaming framework supports a uniform callback interface for both synchronous and asynchronous operations. These callbacks indicate when an operation completes. For instance, a single-threaded application that needs to load a large image from a remote Blob Server performs an asynchronous Blob Streaming `read`, which does not block the application from handling GUI events. When the library completes the operation, the application is notified via a callback. Similarly, synchronous Blob Streaming operations also complete with callback notifications. The difference from asynchronous calls is that the callback has already been executed when the synchronous call returns.

The advantages of abstracting away from concurrency policies are *increased uniformity* and *increased flexibility of concurrency strategies*. For instance, the same software that is used asynchronously in a single-threaded application can be used synchronously in a multi-threaded application. Because both synchronous and asynchronous operations use callbacks, switching to new concurrency policies simply requires toggling a flag. Therefore, no application software will change. This flexibility is particularly useful for developers of reusable components who write software that can be used with a variety of concurrency strategies.

The disadvantage of this approach is that some developers may never want to program asynchronous operations. To some extent, the use of uniform interfaces increases the complexity of synchronous calls in order to eliminate dependency on a particular concurrency model. To address this issue, the Blob Streaming library offers wrappers around the synchronous callback operations to provide a synchronous/return-value API. This is illustrated in Section 3.4.

2.2.6. Abstracting Away from Event Loops

Complex EMIS applications must react to events from multiple sources. Common sources of EMIS events include DICOM toolkits, HL7 interface engines, GUI window events, and Blob Streaming transfers. Furthermore, the Blob Streaming library must integrate the processing of socket-level events, CORBA events, timer events, and signals.

Each of these sources of events (X Windows, CORBA, etc.) has its own event loop. If an application must react to all of these events, it cannot block indefinitely on any one event loop. One solution is to use a polling technique where the application uses a round-robin policy to check each event loop. A disadvantage to this approach is that it can lead to excessive overhead when there are no pending events.

An alternative approach is to combine the multiple event loops into a single waitable object. Blob Streaming uses ACE's Reactor [Schmidt 1995.17] to implement this technique. The Reactor provides a mechanism that integrates the event demultiplexing and event handler dispatching components of multiple frameworks. It presents applications with an object-oriented interface to lower-level OS event demultiplexing mechanisms that react to I/O handle events, timer events, and signal events. The `select`, `poll`, and `WaitForMultipleObjects` system calls are common examples of these demultiplexing mechanisms. This allows the application to block on the Reactor for all events, eliminating the overhead imposed by the polling technique.

Frameworks such as X windows or CORBA are generally driven by events from "waitable" I/O handles (also called descriptors). We will use a UNIX-centric naming policy and call these *select-based* objects. Some applications and frameworks also use waitable resources such as message queues, semaphores, and condition variables. We will call these *synchronization-based* objects.

An example of a synchronization-based object exists in MT-Orbix. The MT-Orbix library dedicates a thread to each network connection. This allows easy integration with third-party toolkits (such as the Tuxedo transaction monitor) that utilize System V message queues (which are synchronization-based). Requests that come over the connections are queued up in a thread-safe message queue. The main thread of control now waits on a conditional variable, rather than waiting in a demultiplexing operation (like `select` or `poll`) as it would in the single-threaded. After a new request is added to the message queue, the main thread is signaled, which then dequeues and processes the request.

The MT-Orbix model adds a synchronization-based source of event demultiplexing to applications. For Win32 platforms, `WaitForMultipleObjects` can be used to wait on both select-based and synchronization-based objects. However, this is problematic for platforms (such as SVR4 UNIX) that do not provide a uniform model for demultiplexing synchronization-based events. The problem is relatively easy to solve if applications can use multiple threads. In this case, one or more threads could be dedicated to process select-based events, while other threads could be dedicated to process the synchronization-based events queued in the message queue.

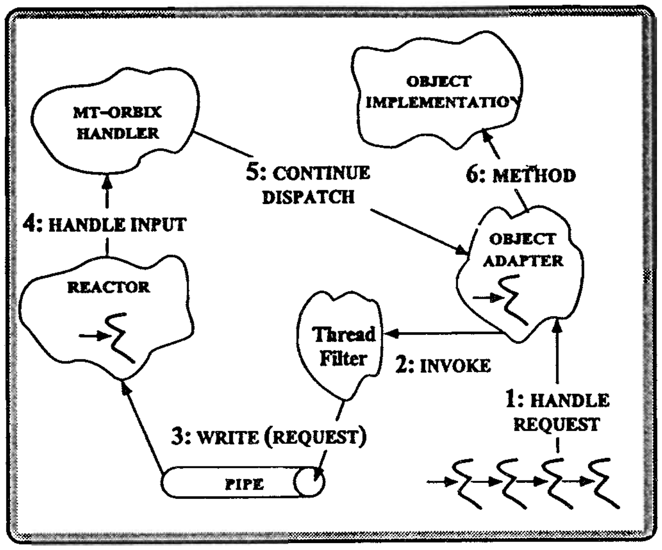

Figure 6. Participants and Collaborations integrating MT-Orbix into the Reactor.

However, many systems (including our EMIS applications) must deal with large amounts of legacy code that is not thread-safe. Therefore, it becomes essential that the select- and synchronization-based events be combined into one logical source. Figure 6 shows how we use the following components to adapt the MT-Orbix I/O handles into a single demultiplexing object:

- Reactor—The main thread is dedicated to handling the select-based events. This is done through the `select` demultiplexing operation. The purpose of the remaining components is to allow the `Reactor` to wait on MT-Orbix events as well as select-based events.

- Object Adapter—A separate thread is dedicated to handling the synchronization-based MT-Orbix events. This is done with MT-Orbix's `impl_is_ready` and operation marshalling filters. MT-Orbix uses a thread per network connection to receive incoming requests. When requests are given to the `Object Adapter`, it uses the `Thread Filter` to decide whether to process the event or not.

- Thread Filter—Orbix filters allow applications to access incoming CORBA requests from the message queue before upcalls are invoked on the appropriate objects. The `Thread Filter` notifies the `Reactor` via the `Pipe` when MT-Orbix events occur.

- Pipe—An intra-process communication channel (in this case, a `pipe`) is created for communication between the two event handlers. The reading end of the `pipe` is owned by the `Reactor` thread and is registered with the demultiplexing operation for input events. The `Object Adapter` thread owns the writing end of the `pipe`. When MT-Orbix events occur, the `Object Adapter` thread uses the `pipe` to communicate the event to the `Reactor` thread.

- MT-Orbix Handler—When the `Reactor` is notified of MT-Orbix events, it calls the `MT-Orbix Handler` to handle the request. The `MT-Orbix Handler` then uses the Orbix `Object Adapter` to continue the method dispatching.

- Object Implementation—Once MT-Orbix requests are received and processed by the various components discussed above, application level objects are finally called by the `Reactor` thread.

This design restores the simple model of "single threaded, single source of events" that our legacy applications require.

The advantage of integrating multiple event loops is that it allows developers to use Blob Streaming *while continuing to integrate with other frameworks*. For instance, an application developer building X-window applications can perform Blob Streaming operations without changing how the application interfaces with the event-loop. Since Blob Streaming uses the Reactor, the framework can be integrated with the necessary event-loop without affecting internal framework software or external framework interfaces.

The disadvantage to this approach is that the Reactor must be integrated with each new framework. This integration can be difficult if the framework does not provide adequate hooks into its internal event demultiplexing logic. Moreover, there is a performance penalty for this integration. For instance, the approach we used to integrate MT-Orbix with our single-threaded applications effectively eliminated concurrency within the event demultiplexing layer of imaging applications.

2.2.7. Abstracting Away from Platform-specific OS Mechanisms

As shown in Figure 2, the Blob Streaming framework shields applications from non-portable OS-specific features such as memory mapping, event demultiplexing, multi-threading, and interprocess communication. This, in turn, makes applications using the Blob Streaming interface portable across platforms *without* changing application communication software. The Blob Streaming framework has been ported to a variety of UNIX platforms, as well as Win32 platforms [Custer 1993].

The primary advantage of decoupling application software from OS-specific mechanisms is *cross-platform portability*. The primary disadvantage is that performance and functionality may be compromised to provide a generic OS interface. For example, the version of Blob Streaming described in this paper did not take advantage of native Windows NT asynchronous I/O mechanisms such as overlapped I/O or I/O completion ports [Schmidt & Stephenson 1995.19].

3. Blob Streaming Interfaces and Examples

This section describes the key components in the Blob Streaming framework and illustrates how to use these components to program synchronous and asynchronous Blob transfer applications. Our goal is to demonstrate the expressive power and simplicity of the framework.

3.1. Blob Proxy

Figure 7 shows the interface of the `BlobProxy` class, which includes methods like `open`, `close`, `read`, `write`, `size`, and `position`. These methods are similar to those provided by System V Release 4 (SVR4) UNIX for file I/O. SVR4 UNIX adapts a wide variety of disk and communication devices into a common set of I/O operations. Blob Streaming has the following notable differences from the SVR4 UNIX file system interfaces, however:

- Seamless Integration of Memory, Networking, and File I/O—The SVR4 UNIX I/O interfaces are not entirely uniform. For instance, a different set of calls is required to open a socket vs. opening a file. Likewise, SVR4 UNIX uses a different interface for memory-mapped file I/O and buffer-based network/file I/O. In contrast, Blob Streaming provides a uniform interface for all these forms of I/O. This makes it possible to abstract away from Blob location by removing inconsistencies and special cases in the I/O programming model.
- Object-oriented interfaces—Low-level network programming tools such as sockets do not provide sufficient type-checking since they utilize untyped I/O handles. It is disturbingly easy to misuse these interfaces in ways that can only be detected at run-time (such as trying to read or write data on a passive-mode listener socket used to accept connections). Unlike SVR4 UNIX, which provides these C-level system call interfaces, Blob Streaming provides C++ interfaces. The use of C++ enforces encapsulation and yields

a more modular, extensible, and less error-prone programming interface, without compromising performance.

The BlobProxy interface is designed so that operations can be invoked synchronously or asynchronously. Asynchronous invocation is useful for long-duration operations (such as open, send, and recv) that can run independently without blocking the main thread of control. Synchronous invocation is useful for (1) short-duration operations (such as size and type) that do not block the caller for long and (2) applications that spawn multiple threads to execute the calls without blocking the entire process.

The SynchOptions class gives users a single interface to specify the type of synchrony/asynchrony policy used for a call. This encapsulation simplifies the Blob Streaming interfaces and gives applications greater flexibility over the synchronization policies used by the application. For instance, applications can define a global instance of SynchOptions that is passed in to every Blob Streaming operation. In this way, applications can change the synchronization policy used by the entire application through a single SynchOptions instance. The SynchOptions interface is defined as follows:

```
class SynchOptions
{
  // Options flags for controlling synchronization.
  enum Options {
    NONBLOCK, // Use asynchronous invocation.
    BLOCK,    // Use synchronous invocation.
    TIMEOUT   // Use timed invocation.
  };

  SynchOptions
    (Options options,              // Synch policy.
     const TimeValue &timeout,     // Timeout duration.
     LocalReceiver *notifiee = 0); // Who to notify.

  // ...others omitted...
}
```

The Options enumeration records whether the call is to be made synchronously or asynchronously and whether it should be timed or not. If the TIMEOUT enumeral is enabled, the TimeValue is interpreted as specifying a timeout duration. Finally, if the call is performed asynchronously, the LocalReceiver pointer is used to specify an object whose receiveNotification method is called back when the asynchronous invocation completes.

```
class BlobProxy
{
public:
  // Open the Blob Proxy.
  void open (const SynchOptions &options);

  // Close the proxy down and release resources.
  void close (void);

  // Read <numBytes> from the Blob Proxy into the
  // <buffer>.
  void read (Buffer &buffer,
             size_t numBytes,
             const SynchOptions& options);

  // Write <numBytes> from the <buffer> to the
  // Blob Proxy.
  void write (const Buffer &buffer,
              size_t numBytes,
              const SynchOptions& options);

  // Size of data represented by the Blob Proxy.
  size_t size
    (const SynchOptions& options) const;

  // Type of data represented by the Blob Proxy.
  // Various types include pixel data or DICOM
  // image.
  BlobProxy::Type type
    (const SynchOptions& options) const;

  // Set/Get the position of the Blob Proxy
  // This allows the user to move to a
  // particular location in the Blob.
  void position (size_t offset,
                 BlobProxy::OffsetSetting whence,
                 const SynchOptions& options);
  size_t position
    (const SynchOptions& options) const;

  // ...others omitted...
```

Figure 7. BlobProxy Interface.

```
private:

  // A Blob Proxy can only be created
  // by a Blob Proxy Factory.
  BlobProxy (const BlobKey &key);
};
```

Figure 7. Continued.

3.2. *Blob Proxy Factory*

The Blob Proxy Factory is responsible for creating proxies to Blobs that may be remote or local. The Factory is also responsible for dynamically selecting and configuring the objects (such as Slots) needed to implement the Blob Proxy interface. This encapsulation of the responsibility and process of creating and composing implementation objects for the Blob Proxy isolates the user of the proxies from the implementation classes.

Figure 8 shows the interface of the `BlobProxyFactory` class, which has methods like `bind` and `route`. The `bind` method creates a Blob Proxy that is bound to a Blob. This is similar to the functionality provided by CORBA for creating a proxy to a remote object. The `route` method is used to create a new Blob of a given size. In this case, the factory is responsible for communicating with the appropriate Blob Store to reserve space for the new Blob. If the space is successfully reserved, a proxy is created to the new Blob and returned to the user.

3.3. *Blob Transporters*

The Blob Transporter is responsible for efficiently copying data from one Blob to another. The Blob Transporter implements algorithms that iterate over the source Blob and copy the data to the destination Blob. The `copy` methods of the Blob Transporter are similar to the algorithms provided by the C++ Standard Template Library (STL) [Stepanov & Lee 1994]. STL algorithms are completely generic and behave the same way irrespective of the types they work on. In contrast, the algorithms defined by the Transporter are optimized for different Blob locations. Since there are relatively few Blob locations types (memory, file, network, and database), it is feasible to explicitly optimize each type of Blob Transporter. For instance, a transporter can simply perform a `memcpy` when the source and destination of a copy are both in memory.

```
class BlobProxyFactory
{
public:

  // The factory creates a new Blob Proxy that
  // is bound to an existing Blob represented
  // by the <key>.
  static
  BlobProxy *bindBlob (const BlobKey &key,
                       const SynchOptions &options);

  // The factory creates a new Blob (represented
  // by <key>) of <size> bytes. It also creates
  // a Blob Proxy that is bound to the new Blob.
  static
  BlobProxy* routeBlob (const KBlobKey &key,
                        size_t size,
                        const SynchOptions &options);

  // ... others omitted...
};
```

Figure 8. Blob Proxy Factory Interface.

Figure 9 shows the interface of the `BlobTransporter` class. Note that the `CopyTransporter` only implements static interfaces. State for copies in-progress is dynamically allocated by the copy routine and deleted when the operation completes. If the state for a copy operation was kept as instance data in a `CopyTransporter` instance, the instance would only be able to keep track of one in-progress copy. This would also force the user to create and manage multiple instances of `CopyTransporter` in order to execute multiple copy operations simultaneously.

3.4. Using the Blob Streaming Framework

The following discussion presents several use-cases that illustrate how to program synchronous and asynchronous applications using `BlobProxies`. The two examples in Figures 10 and 11 use Blob Streaming to copy images from a remote Blob Store to a local Blob Store. Blobs in the system are identified uniquely by `BlobKeys`. Both examples copy an image identified by `sourceKey` to an image identified by `destinationKey`.

```
class CopyTransporter
{
public:

  // Copy entire <source> Blob to
  // <destination> Blob.
  static
  void copy (BlobProxy *destinationProxy,
             BlobProxy *sourceProxy,
             const SynchOptions &options);

  // Copy <size> bytes from <source> Blob
  // to <destination> Blob.
  static
  void copy (BlobProxy *destinationProxy,
             BlobProxy *sourceProxy,
             size_t size,
             const SynchOptions &options);

  // ... others omitted...
};
```

Figure 9. Blob Transporter Interface.

A `destinationKey` is created by replicating the `sourceKey` and changing the host information in the `destinationKey` to the local host. Space is then reserved for the new image at the local `BlobStore` by calling `BlobStreamingFactory::routeBlob`. The copy options sets a timeout of 30 seconds for the copy operation. The `copy` operation will timeout if the operation does not complete in the specified time.

In the synchronous example, an exception is raised in the event of failure or timeout. In the asynchronous example, the `Replicator` class is notified of the result of the operation. Exceptions cannot be raised in the asynchronous example since the call to the `copy` method returns immediately without blocking the caller.

The primary difference between the two examples is the nature of the `copy` call. The first example shown in Figure 10 uses a synchronous, return-value based version of the `CopyTransporter::copy` method call. The second example shown in Figure 11 uses the asynchronous, callback based version of the `CopyTransporter::copy` method call.

```
// Retrieve to local store.
void copy (BlobKey sourceKey ) {

  // Create a key for the destination
  BlobKey destinationKey (sourceKey,
                          localHostName);

  // Allocate space on Blob Store for
  // destination Blob.
  BlobStreamFactory::routeBlob (destinationKey,
                                source->size ());

  // timeout after 30 seconds
  TimeValue timeout (30);
  SynchOptions copyOptions
    // Synchronous, timed invocation.
    (SynchOptions::TIMEOUT |
     SynchOptions::BLOCK,
     timeout); // Amount of time to block.

  // Synchronous copy of the Blob.
  try {
    CopyTransporter::copy (sourceKey,
                           destinationKey,
                           copyOptions);
  } catch (RecoverableException exc) {
    switch (exc.tag ()) {
    case ERROR_BLOB_COPY_FAILED:
      // report failure
      break;
    case ERROR_BLOB_COPY_TIMEOUT:
      // report timeout
      break;
    }
  }
  // report success
}
```

Figure 10. Synchronous, Return-value-based Copy Example.

```
class Replicator
  : public LocalReceiver
    // Defines the pure virtual
    // receiveNotification() method.
{
public:
  // Handles I/O completion.
  virtual bool receiveNotification
    (LocalNotification *notification);

  // Retrieve to local store.
  void copy (BlobKey sourceKey) {
    // Create a key for the destination
    BlobKey destinationKey (sourceKey,
                            localHostName);

    // Allocate space on Blob Store for
    // destination Blob.
    BlobStreamFactory::routeBlob (destinationKey,
                                  source->size ());

    // Timeout after 30 seconds.
    TimeValue timeout (30);
    SynchOptions copyOptions
      // Asynchronous, timed invocation.
      (SynchOptions::TIMEOUT |
       SynchOptions::NONBLOCK,
       timeout,       // Amount of time to wait.
       this);         // Notify this object (Replicator)
                      // upon completion of the copy.

    // Start an asynchronous copy.  On completion,
    // our receiveNotification() method is called.
    CopyTransporter::copy
      (sourceKey,      // copy from this Blob
       destinationKey, // to this Blob
       copyOptions);   // copy options
};
```

Figure 11. Asynchronous Callback-based Copy Example.

```
bool Replicator::receiveNotification
  (LocalNotification *notification)
{
  CopyTransporter::CopyNotification *
    copyNotification =
    (CopyTransporter::CopyNotification *)
    notification;

  switch (copyNotification->result ()) {
  case CopyTransporter::CopyNotification::SUCCEEDED:
    // report success
    break;
  case CopyTransporter::CopyNotification::FAILED:
    // report failure
    break;
  case CopyTransporter::CopyNotification::TIMEOUT:
    // report timeout
    break;
  default:
    return 0;
  }
};
```

Figure 12. Receiver of Copy Notifications.

Figure 12 illustrates the method that receives copy notifications for callback-based copies. The following changes to `Replicator::copy` are all that are required to make an asynchronous, callback-based operation like this:

```
SynchOptions copyOptions
  // Asynchronous, timed invocation.
  (SynchOptions::TIMEOUT |
   SynchOptions::NONBLOCK,
   timeout, // Amount of time to wait.
   this);   // Notify this object upon
            // completion of the copy.
```

into a synchronous, callback-based operation like this:

```
SynchOptions copyOptions
(SynchOptions::TIMEOUT, timeout);
```

If the thread executing the `copy` operation can afford to block without compromising the quality of service of other components of the application, the synchronous approach can be used. However, if the long-duration `copy` operation will affect other components of the application, the asynchronous approach can be used. This allows the application developer to develop the imaging replication module without becoming dependent on the concurrency model used for image replication. As a result, systems that use Blob Streaming are more portable than those written to use lower-level OS mechanisms directly.

4. *Performance of the Blob Streaming Framework*

Sections 2.2 and 3 motivate and outline the design and use of the Blob Streaming framework. Our design abstracts away from many low-level communication tasks to achieve the flexibility requirements of distributed EMISs. In practice, however, we recognized that the framework will not be widely used unless applications built using it meet their performance requirements.

This section describes performance tests of the Blob Streaming framework. The test scenario involved the point-to-point transfer of Blobs between a client and a server. In a large-scale EMIS, several types of bulk data transfers can place high loads on a communications framework. For instance, transferring a typical MR image study can include fifty 250 Kbyte images. Likewise, a CR image study can include several 500 Kbyte images. The tests performed on the Blob Streaming framework have been designed to mimic the behavior of transmitting studies such as these.

4.1. *Test Platform and Benchmarks*

The performance results in this section were collected using a Bay Networks LattisCell 10114 ATM switch connected to two dual-processor SPARCstation 20 Model 712s running SunOS 5.4. The LattisCell 10114 is a 16 Port, OC3 155Mbs/port switch. Each SPARCstation 20 contains two 70 Mhz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.4 TCP/IP protocol stack is implemented using an optimized version of the STREAMS communication framework [Ritchie 1984]. Each SPARCstation has 128 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to eight switched virtual connections per card.

Data for the experiment was produced and consumed by a client and server test application. The client represents a diagnostic workstation. The server application represents a Blob Store server. Various client and server parameters may be

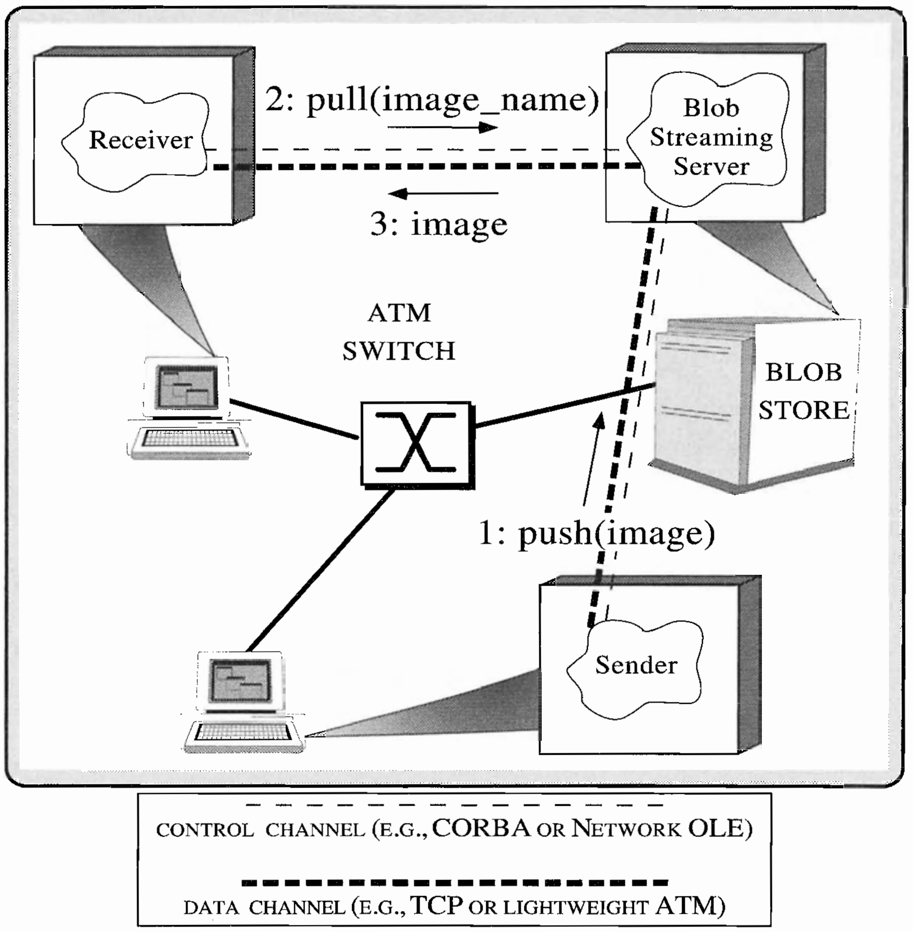

Figure 13. Push and Pull Models.

selected at run-time. These parameters include the size of the Blob being transferred and the size of the socket transmit and receive queues.

Our test environment is similar to the widely available `ttcp` benchmarking tool. However, our test application differs from `ttcp` since we implement a "request/response" model rather than the conventional `ttcp` "flooding" model. In our model, the client can request the server to send it data (the "pull" model) or move data to the server (the "push" model). This is different from `ttcp` because the data transmitter does not simply flood the receiver with a continuous unidirectional stream of bytes. The push and pull models implemented by our test application are illustrated in Figure 13 and described below.

- The push model: This model is representative of the use case where a modality stores data on a Blob Store. In addition, it can be used by a Blob

Store to pre-cache data to a workstation. The push model behaves as follows:

1. Negotiation—the client sends control data to the server characterizing the image being transferred from the client to the server (e.g., size and name of the image).
2. Transmission—the client then sends the image data.
3. Confirmation—the server sends a confirmation to the client when all the data is received. This acknowledgment is necessary to insure end-to-end reliability of the request/response transaction.

- The pull model: This model is representative of the use case where a workstation retrieves data from a Blob Store. The pull model behaves as follows:
  1. Negotiation—the client sends control data to the server characterizing the image the client wants from the server (size and name of the image).
  2. Transmission—the server then sends the image data. Once the client receives the data that was requested from the server, the request/response transaction is complete. Unlike the push model, the pull model does not require an extra acknowledgment, which improves performance, as shown in Figure 19.

We implemented and benchmarked the following versions of the test application for Blob transfers:

- C version: This version is implemented completely in C. It uses C socket calls to transfer and receive the data and control messages via TCP/IP. Figure 14 illustrates the design of this `ttcp` test.
- ACE C++ version: This version replaces all C socket calls in the applications with the C++ wrappers for sockets provided by the ACE network programming components [Schmidt 1994.16]. ACE encapsulates sockets with typesafe, portable, and efficient C++ interfaces. Figure 14 illustrates the design of this test, as well.
- CORBA version: The Orbix 1.3 implementation of CORBA was used. This version replaces all socket calls in the test applications with stubs and skeletons generated from a pair of CORBA interface definition language (IDL) specifications. One IDL specification uses a `sequence` parameter for the data buffer and the other uses a `string` parameter. Figure 15 illustrates the design of this test.
- Blob Streaming version: the Orbix implementation of CORBA was used to exchange control messages and C++ wrappers for sockets provided by ACE

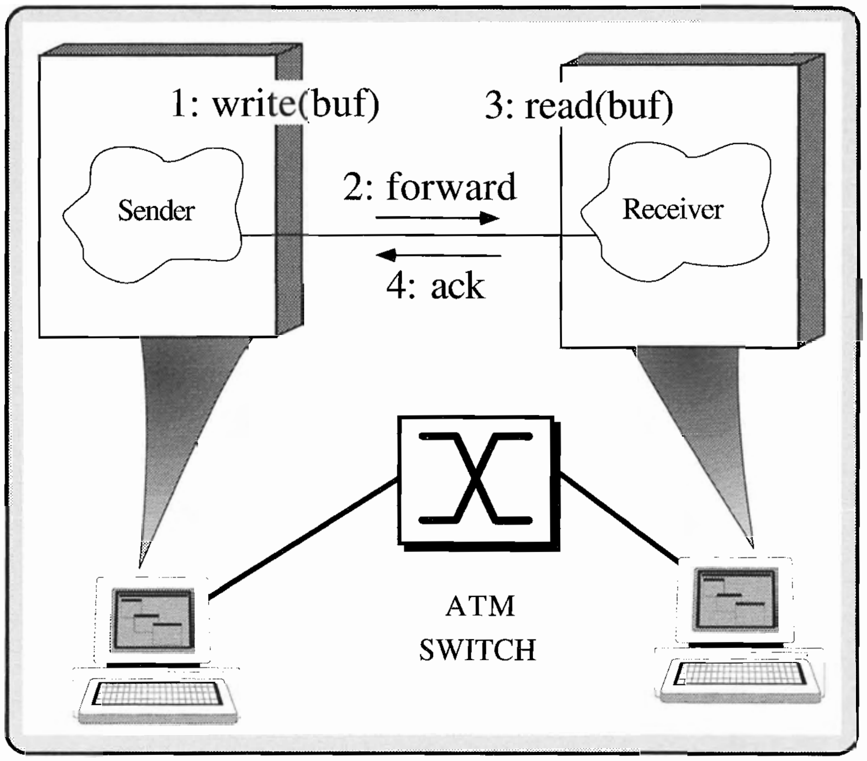

Figure 14. C and C++ `ttcp` Benchmarking Architecture.

were used for bulk data transfer. This is the only test that implements both the push and pull models. Figure 16 illustrates the design of this test for the push model and Figure 17 illustrates the design of the test for the pull model.

4.2. Performance Results

4.2.1. Throughput Results

We ran a series of tests that transferred 1 MB, 8 MB, 16 MB, and 32 MB of user data using TCP/IP over our ATM network testbed. Two different sizes for socket queues were used: 8 K (the default on SunOS 5.4) and 64 K (the maximum size supported by SunOS 5.4). Each test was run 20 times to account for performance variation due to transient load on the networks and hosts. The variance between runs was very low since the tests were conducted on an otherwise idle network.

- Push Model Throughput: Figure 18 shows that different versions of tests

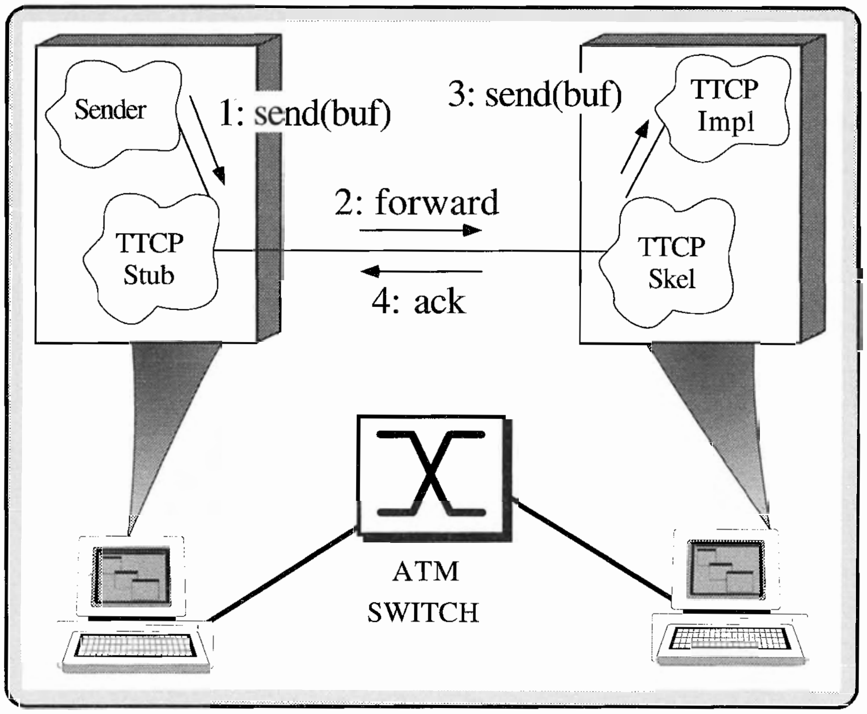

Figure 15. CORBA `ttcp` Benchmarking Architecture.

for Ethernet show much less variation, with the performance for all tests ranging from around 8 to 8.7 Mbps with 64 K socket queues. In addition, Figure 18 summarizes the performance results for all the push model benchmarks using 64 K and 8 K socket queues over a 155 Mbps ATM link.

The following describes the performance of each test program, using 64 K and 8 K socket queues:

- The C and ACE C++ wrapper versions of the tests obtained the highest throughput: 60 Mbps using 64 K socket queue. This indicates that the performance penalty for using the higher-level ACE C++ wrappers is insignificant and is comparable with using low-level C socket library calls directly.
- The Blob Streaming performance was slightly more than 80% of the C and C++ versions, reaching 50 Mbps with 64 K socket queues. The primary source of overhead in the Blob Streaming framework is explained in Section 4.2.2.
- The Orbix `sequence` version peaked at around 66% of the C and C++ versions, reaching 40 Mbps, whereas the Orbix `string` implementa-

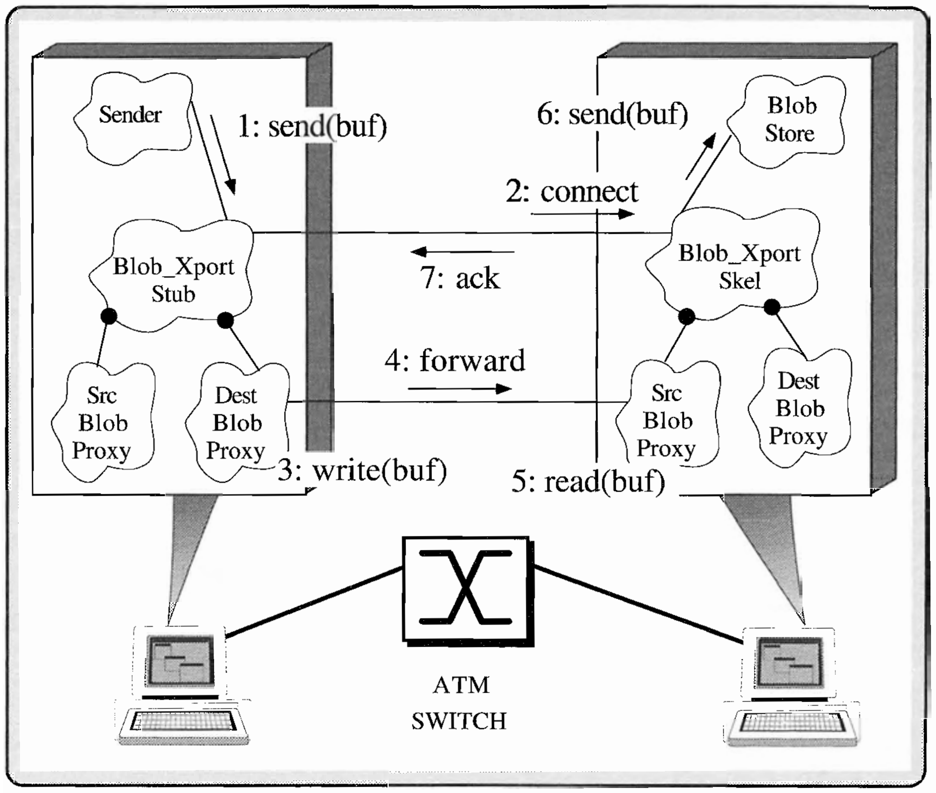

Figure 16. Blob Streaming `ttcp` Benchmarking Architecture (Push Model).

tion peaked at 33 Mbps (both using 64 K socket queues). The primary sources of overhead for the Orbix implementation of CORBA is explained in Section 4.2.2.

In addition to comparing the performance of the various transport mechanisms, Figure 18 also illustrates the generally low level of utilization of the ATM network. In particular, 60 Mbps represents only 40% of the 155 Mbps ATM link. This disparity between network channel speed and end-to-end application throughput is known as the *throughput preservation problem* [Schmidt & Suda 1993.15]. This problem occurs when only a portion of the available bandwidth is actually delivered to applications.

The throughput preservation problem stems from operating system and protocol processing overhead (such as data movement, context switching, and synchronization [Clark & Tennenhouse 1990]). This throughput preservation problem is exacerbated by contemporary implementations of distributed object computing middleware like CORBA, which copy data

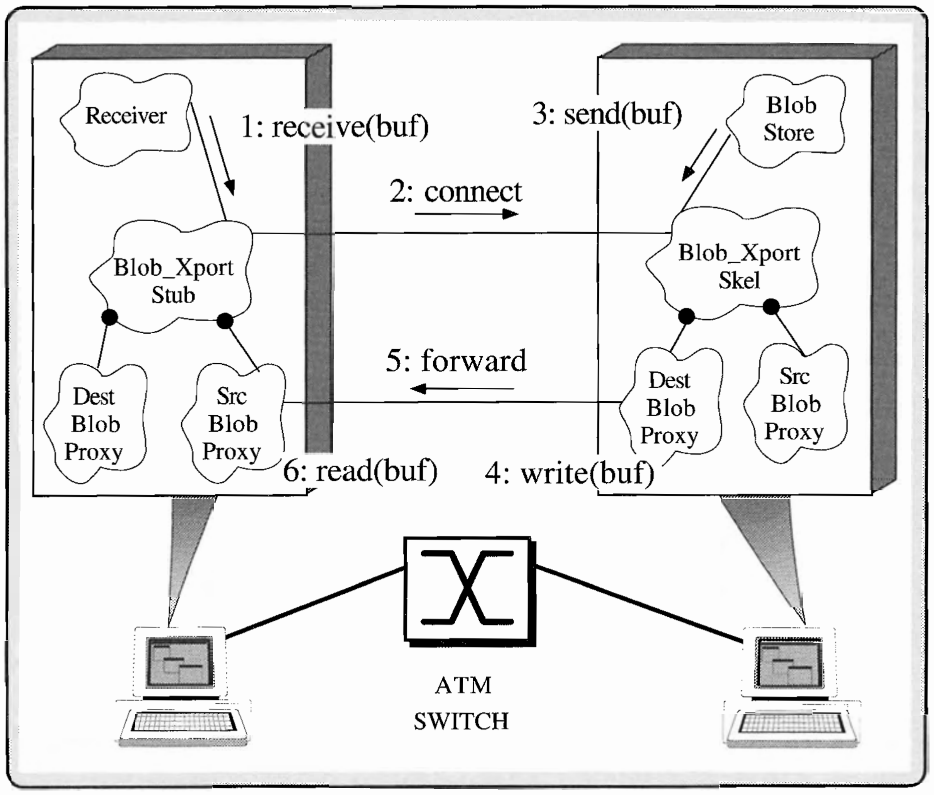

Figure 17. Blob Streaming ttcp Benchmarking Architecture (Pull Model).

multiple times during fragmentation/reassembly, marshalling, and demarshalling. Furthermore, the latency associated with the request-response protocol implemented by ttcp significantly reduced performance. An earlier implementation of ttcp [Schmidt et al. 1995.18] attained 90 Mbps over the same ATM testbed by using a "flooding" traffic generation model that did not use an end-to-end acknowledgment scheme.

Finally, Figure 18 illustrates the impact of socket queue size on throughput. Increasing the socket queue from 8 K to 64 K doubled performance from 28 Mbps to 60 Mbps. The reason for this is that larger socket queues increase the TCP window size [Modeklev et al. 1994], which allows the transmission of multiple TCP segments back-to-back.

These socket queue results demonstrate the importance of having hooks to manipulate underlying OS mechanisms (such as transport layer and socket layer options). It is important to note that the choice of socket queue size has more impact than the choice of communication model (i.e.,

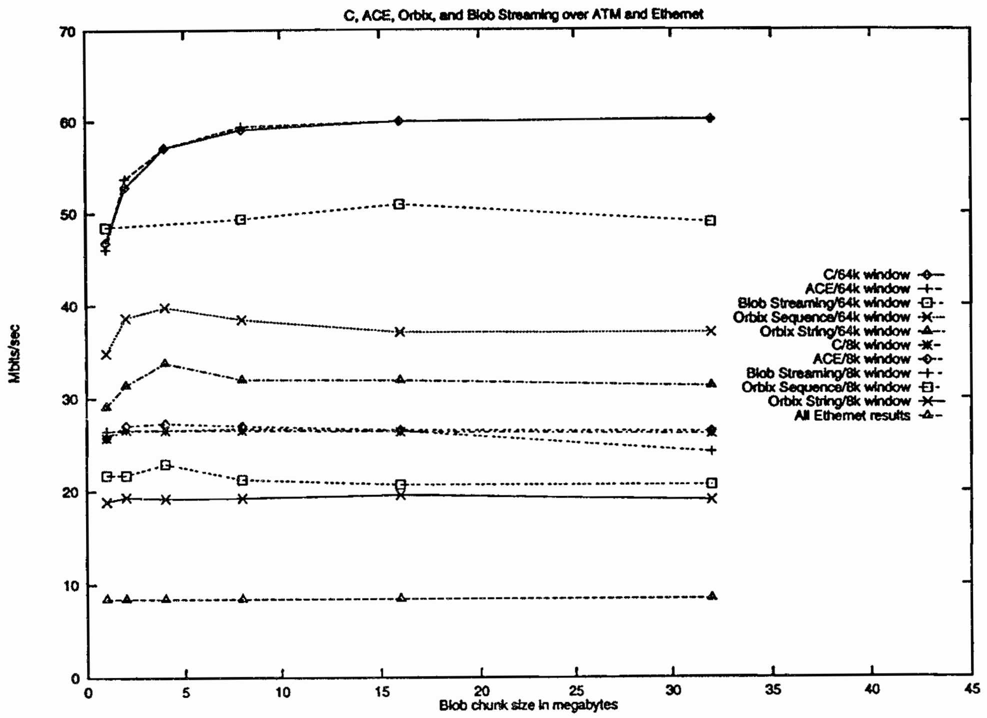

Figure 18. Push Model Performance Over Ethernet and ATM.

C/C++ vs. CORBA vs. Blob Streaming). In fact, the slowest communication model (CORBA) is faster with 64 K socket queues than the faster communication model (C/C++) with 8 K queues. Clearly, communication frameworks that do not offer these hooks to application developers are destined to perform poorly over high-speed networks.

- Pull Model Throughput: Figure 19 compares the performance of the pull model and the push model of the Blob Streaming versions of the tests.[4] For 64 K socket queue size, the pull model out-performed the push model by 15% to 20% for all sizes of data being transferred. This result illustrates the drawback of the push model, which must wait for an acknowledgment from the receiver in order to guarantee end-to-end delivery.

  Figure 19 also compares the two models with 8 K socket queue sizes. There is no appreciable difference in performance of the two mod-

4. Due to space constraints, the ACE, C, and CORBA pull model results are not shown – they exhibit similar performance curves, however.

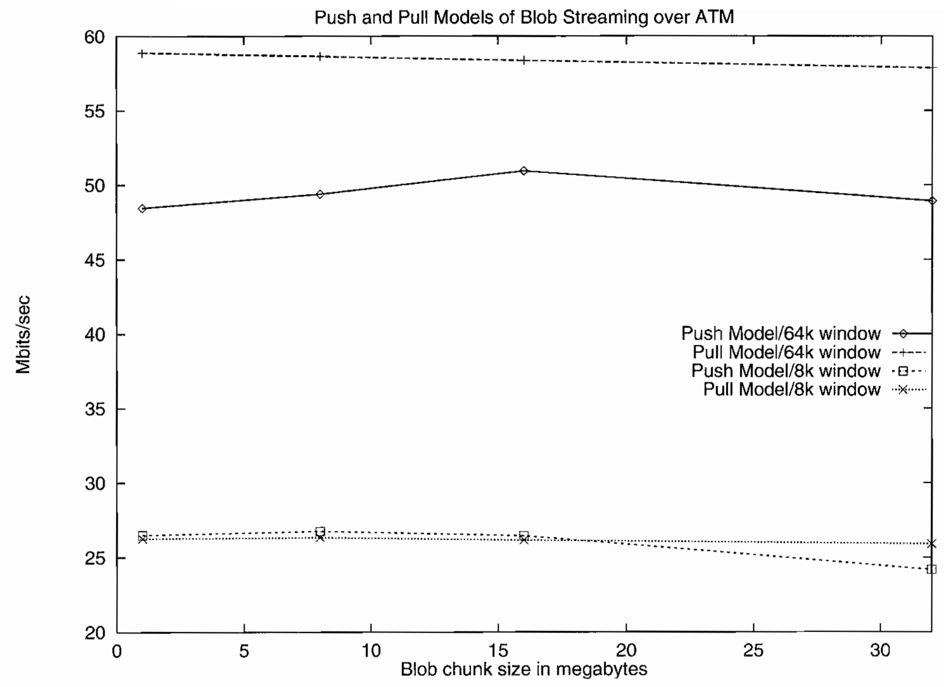

Figure 19. Pull Model vs. Push Model Performance Over ATM for Blob Streaming.

els with this socket queue size. This illustrates once again how important it is for ORBs to allow applications to tune the size of the underlying socket queues.

4.2.2. High Cost Functions

In order to explain the throughput results shown above, we used the `Quantify` execution profiler [P. Software 1995] to pinpoint the sources of overhead. The test applications were relinked using `Quantify`, which modified the object code to include monitoring instructions. Two related tools (`qxprof` and `qv`) were then used to display and measure the amount of time spent in functions during program execution. Table 1 lists the functions where the most time was spent sending and receiving 1 Mbytes of user data and using 64 K socket queues. The results show the push model experiment repeated 100 times.

- High cost operations for C and ACE C++: The high cost operations for C and ACE C++ wrapper versions are nearly identical. The sender spent 94% of the time in the `write` system call sending data to the receiver. About 3% of the time was spent in receiving acks from the receiver. The receiver

Test	%Time	#Calls	Name
C sockets (sender)	93.9	112	`write`
	3.6	110	`read`
C sockets (receiver)	93.2	13,085	`read`
	4.5	102	`write`
ACE C++ wrapper (sender)	94.4	112	`write`
	3.2	110	`read`
ACE C++ wrapper (receiver)	93.9	12,984	`read`
	5.6	102	`write`
Orbix Sequence (sender)	53.5	127	`write`
	35.1	223	`read`
	7.3	1,108	`memcpy`
Orbix Sequence (receiver)	84.6	12,846	`read`
	12.4	1,064	`memcpy`
	3.2	101	`write`
Orbix String (sender)	45.0	127	`write`
	35.1	223	`read`
	10.8	1,315	`strlen`
	6.0	1,108	`memcpy`
Orbix String (receiver)	70.7	12,443	`read`
	16.1	2,142	`strlen`
	10.0	1,064	`memcpy`
	3.0	101	`write`
Blob Streaming (sender)	48.8	327	`write`
	44.8	232	`read`
	1.3	2,055	`memcpy`
Blob Streaming (receiver)	77.2	12,546	`read`
	16.4	12,734	`memcpy`
	1.4	102	`write`

Table 1. High Cost Functions for `Push` Model Blob Streaming Tests.

spent 93% of the time in the `read` system call receiving data from the sender. About 1.5% of the time was spent in sending acks from the sender.

The sender approximately made 100 `write` system calls (once per iteration) to send the data and approximately 100 `read` system calls (once per iteration) to receive the ack. The receiver made approximately 13,000 `read` system calls (130 times per iteration) to receive the data and approximately 100 `write` system calls (once per iteration) to send the ack. The excessive amounts of `reads` results from fragmentation of the data into packets of 9,180 bytes, which is the maximum transmission unit (MTU) size of the ATM network.

- High cost operations for Orbix: Two different implementations of Orbix were profiled. The first version uses a `sequence` parameter for the data buffer and the other uses a `string` parameter. Both the sender and the receiver spent a considerable amount of time in copying data (6-12% of the time was spent in `memcpy`), slowing down the performance of the system. The decrease in performance compared to the C and ACE wrappers versions causes the sender to wait longer to receive the ack from the receiver. This is indicated by the time spent in the `read` system call. These experiments are similar to the ones in [Schmidt et al. 1995.18] and details about the behavior is explained in that paper.

  The Orbix implementation differs from the C and ACE implementations in the number of `read` system calls made to receive an ack. Orbix implementations make two `read` system calls per-ack compared to one call by the C and ACE versions. This is because Orbix uses the "header followed by the data" protocol. The first `read` system call reads the fixed size header and the subsequent `read` system call reads the variable size payload. This protocol is not necessary in the C and ACE versions since the only type of information sent to the sender is an ack.

  Figure 18 illustrated that the performance of the Orbix `sequence` results consistently performed around 6 to 7 Mbps higher than the `string`. This difference in performance is due to the C++ mapping for strings in the CORBA IDL specification. The client-side stubs that perform parameter marshalling for remote calls must obtain the length of the string being sent. This is accomplished via calls to `strlen`, which add significant overhead to the `string` version. However, the IDL-to-C++ mapping of the `sequence` provides length fields in addition to the data.

  To illustrate the difference, consider the following IDL definition of a `sequence` and its corresponding C++ mapping:

```
// IDL definition
typedef sequence<char> char_sequence;
oneway void push (in char_sequence data_seq,
                  in string data_string);

// C++ mapping
struct char_sequence {
  u_long _maximum;
  u_long _length;
  char *_buffer;
};

void push (const char_sequence &data_seq,
           const char *data_string);
```

The _length field is explicitly set by the application allowing client-side stub to know the size of the _buffer. Thus, data_string requires a strlen; data_seq does not.

- High cost operations for Blob Streaming: Compared with the C, ACE, and Orbix implementations, the Blob Streaming sender implementation performs a higher number of write calls. As shown in Table 1, Blob Streaming makes three write system calls per iteration, whereas the C, ACE, and Orbix versions only make one call. The first call by Blob Streaming sends the control information, the second call is for the data, and the third is for a request for the ack. The control information cannot be bundled with the data as Blob Streaming uses different channels for control and data messages. All the other versions use the same channel for control and data messages.

  The Quantify analysis of the Blob Streaming implementation revealed that the receiver spent 16.4% of the time in memcpy. Upon closer inspection, we found our implementation was making an extra copy of the data received from clients. A single extra copy reduced the performance of Blob Streaming and the sender has to wait longer to receive an ack from the receiver.

  One way to reduce this overhead is to have the application preallocate the buffer space before passing into the Blob Stream receiver. Once we remove the extra data copy from the receiver, we expect the results to perform roughly the same as the C and ACE C++ wrapper versions. In particular, although the sender makes three times more calls to write, we expect the overhead is due to the extra data copying on the receiver, rather than the additional mode switching on the sender.

5. Evaluations and Recommendations

When developing large frameworks such as Blob Streaming, the greatest challenge is designing for future changes in requirements and environments. The framework must be able to adapt to the ever-changing needs of the customer it is built for. Blob Streaming chose CORBA as a tool to help the framework meet these demands. The following two sections discuss our recommendations to others facing similar challenges.

5.1. Designing Object-Oriented Communication Frameworks

Based on our performance experiments and our experience using the Blob Streaming framework, our evaluations and recommendations for developing object-oriented communication frameworks for high-performance bulk data delivery systems include the following:

- Develop flexible tools—The framework must be able to deal with new types of data and new transport protocols and networks. If the tools used to build the framework cannot adapt to changing needs, the framework will not be flexible either. This was one of our motivations for using CORBA.
- Know the performance requirements—Meeting the performance requirements of bandwidth-intensive and delay-sensitive applications is essential before the framework will be adopted widely. Furthermore it is important to evaluate tools based on empirical measurements rather than adopting a particular communication model or implementation unconditionally. Our performance requirements motivated the combination of CORBA with lower-level transport mechanisms to achieve the performance benefits of sockets.
- Make the system easy to use—The learning curve of using a new framework must be as small as possible. This inspired us to simplify the Blob Streaming interfaces by modeling after the UNIX file I/O interfaces and including abstractions such as `SynchOptions` and the stateless `CopyTransporter`. Whenever possible, leverage well known designs and idioms that will help decrease the learning curve for the framework users.
- Decouple concurrency policies—The framework should try to avoid making concurrency policy decisions. Applications using the framework should not have to be single- or multi-threaded. The framework must, however,

provide mechanisms that allow the framework to work correctly in a multi-threading and multi-processing environment. Blob Streaming addresses this need by supporting uniform callback interfaces for both synchronous and asynchronous operations.

- Design with portability in mind—Portability requirements of the framework must be addressed in the early phase of design. This helps the designers and developers make reasonable assumptions about the OS level services available. Blob Streaming uses the ACE toolkit [Schmidt et al. 1994.16] to remove dependencies from OS-specific system call mechanisms.
- Design for new technologies—Networks have experienced a tremendous growth in the last few years. There is no reason to doubt that this trend will continue for many more years. Prototypes of gigabit network are already being developed [Parulkar et al. 1995]. Next generation frameworks must be able to adapt to new technologies such as higher speed networks and new transport protocols.
- Do not assume event-loop ownership—The framework should not assume ownership of the event-loop. Applications using the framework will typically be dealing with multiple sources of input like GUI events and CORBA events. Blob Streaming uses the ACE Reactor [Schmidt 1995.17] as a single demultiplexing object to encapsulate these multiple sources of events.

5.2. Using CORBA Effectively

CORBA offers many advantages for developing complex distributed systems since it automates many common network programming tasks such as object selection, location, and activation, as well as parameter marshalling and framing. However, a major disadvantage of CORBA is that current implementations incur significant performance overhead when used to transfer large amounts of data [Schmidt 1995.18].

We addressed the performance problems of CORBA by integrating it with sockets. Our approach uses CORBA for control messages and sockets for bulk data transfer. This two-tiered design leverages CORBA's extensibility and socket's efficiency. CORBA is particularly useful for short-duration, request/response operations that exchange richly typed data.

Modifying or extending the type of information exchanged between applications is also straightforward using CORBA since it automatically generates code to marshall the parameters. Thus, for many types of inter-process communication, CORBA offers a powerful solution. TCP/IP endpoint negotiations in Blob Streaming are performed using CORBA messages. These negotiations usually contain small amounts of richly typed data, and therefore are well suited for CORBA.

The poor performance of CORBA bulk data transfer is a result of existing implementations that fail to optimize common sources of overhead. This overhead stems primarily from inefficient presentation layer conversions, data copying, memory management, and inefficient receiver-side demultiplexing and dispatching operations. This overhead is often masked on low-speed networks like Ethernet and Token Ring. On high-speed networks like ATM or FDDI, however, this overhead becomes a significant factor limiting communication performance [DoVan et al. 1995]. To overcome these inefficiencies, we use sockets to setup point-to-point TCP connections and transmit bulk data efficiently across the connections. Since Blob Streaming does not interpret the data it transfers, the untyped nature of socket-level data exchange is acceptable.

Low-level network programming interfaces like sockets are hard to program because they have complex interfaces and are prone to subtle programming errors. Our solution to this problem was to use C++ wrappers from the ACE toolkit [Schmidt 1994] to encapsulate the C interfaces. ACE provides a rich set of efficient, reusable C++ wrappers, class categories, and frameworks that perform common communication software tasks (such as event demultiplexing, event handler dispatching, connection establishment, message routing, dynamic configuration of application services, and concurrency control).

It is important to note that ACE does not offer all the services of CORBA (such as object selection, location, activation, and parameter marshalling). Therefore, CORBA provides important value as a higher-level distributed object computing framework.

6. Concluding Remarks

We are currently deploying the Blob Streaming framework in a production distributed electronic medical imaging system being developed as part of Project Spectrum at the Electronic Radiology Lab (ERL) at the Washington University School of Medicine and BJC Health System, in collaboration with industrial partners Kodak Health Imaging Systems, IBM/ISSC, and Southwestern Bell Corporation. BJC is one of the nation's largest integrated health delivery systems, representing an alliance of health care partners in Missouri and southern Illinois.

Distributed electronic medical imaging systems like Project Spectrum require high-performance bulk data communication. The Blob Streaming framework described in this paper uses sockets to achieve high performance and uses CORBA to provide the flexibility needed for distributed electronic medical imaging systems. Blob Streaming allows application code to be developed independent of Blob location, Blob type, and Blob storage. These abstractions allow image processing algorithms to be reused for many types and locations of Blobs. In addition, Blob Streaming is designed to allow flexibility across platforms by abstracting from OS-specific mechanisms, concurrency policies, and event loops.

References

1. G. Blaine, M. Boyd, and S. Crider, Project Spectrum: Scalable Bandwidth for the BJC Health System, *HIMSS, Health Care Communications*, pages 71–81, 1994.
2. D. D. Clark and D. L. Tennenhouse, Architectural Considerations for a New Generation of Protocols, *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, Philadelphia, PA, pages 200–208, ACM, Sept. 1990.
3. H. Custer, *Inside Windows NT.* Redmond, Washington, Microsoft Press, 1993.
4. M. DoVan, L. Humphrey, G. Cox, and C. Ravin, Initial Experience with Asynchronous Transfer Mode for Use in a Medical Imaging Network, *Journal of Digital Imaging*, vol. 8, pages 43–48, February 1995.
5. J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, Beyond Multiprocessing... Multithreading the SunOS Kernel, *Proceedings of the Summer USENIX Conference*, San Antonio, Texas, June 1992.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1995.
7. A. Gokhale and D. C. Schmidt, Measuring the Performance of Communication Middleware on High-Speed Networks, *Proceedings of SIGCOMM '96*, Stanford, CA, ACM, August 1996.
8. A. Gokhale and D. C. Schmidt, Perfomance of the CORBA Dynamic Invocation Interface and Internet Inter-ORB Protocol over High-Speed ATM Networks, *Proceedings of GLOBECOM '96*, London, England, IEEE, November 1996.
9. K. Modeklev, E. Klovning, and O. Kure, TCP/IP Behavior in a High-Speed Local ATM Network Environment, *Proceedings of the 19th Conference on Local Computer Networks*, Minneapolis, MN, pages 176–185, IEEE, Oct. 1994.
10. Object Management Group, *CORBAServices: Common Object Services Specification, Revised Edition*, 95-3-31 ed., Mar. 1994.
11. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.
12. P. Software, *Quantify User's Guide*, 1995.
13. G. Parulkar, D. C. Schmidt, and J. S. Turner, a^It^Pm: a Strategy for Integrating IP with ATM, *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, ACM, September 1995.
14. D. Ritchie, A Stream Input-Output System, *AT& T Bell Labs Technical Journal*, vol. 63, pages 311–324, Oct. 1984.
15. D. C. Schmidt and T. Suda, Transport System Architecture Services for High-Performance Communications Systems, *IEEE Journal on Selected Areas in Communication*, vol. 11, pages 489–506, May 1993.
16. D. C. Schmidt, ACE: an Object-Oriented Framework for Developing Distributed Applications, *Proceedings of the 6th USENIX C++ Technical Conference*, Cambridge, Massachusetts, USENIX Association, April 1994.

17. D. C. Schmidt, Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching, *Pattern Languages of Program Design*, J. O. Coplien and D. C. Schmidt, eds., Reading, MA: Addison-Wesley, 1995.
18. D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, Object-Oriented Components for High-speed Network Programming, *Proceedings of the 1st Conference on Object-Oriented Technologies and Systems*, Monterey, CA, USENIX, June 1995.
19. D. C. Schmidt and P. Stephenson, Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms, *Proceedings of the 9th European Conference on Object-Oriented Programming*, Aarhus, Denmark, ACM, August 1995.
20. D. C. Schmidt, A Family of Design Patterns for Application-level Gateways, *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, S. P. Berczuk, ed., Wiley and Sons, 1996.
21. D. C. Schmidt, A Family of Design Patterns For Flexibly Configuring Network Services in Distributed Systems, *International Conference on Configurable Distributed Systems*, May 6–8 1996.
22. A. Stepanov and M. Lee, The Standard Template Library, Tech. Rep. HPL-94-34, Hewlett-Packard Laboratories, April 1994.