An Object-Oriented Framework for High-Performance Electronic Medical Imaging

Douglas C. Schmidt
Tim Harrison
Department of Computer Science
Washington University, St. Louis, Missouri, 63130[1]
Irfan Pyarali
Kodak Health Imaging Systems
Dallas, Texas, 75240

This paper will appear in the *Software Technology Applied to Imaging and Multimedia Applications* mini-conference at the Symposium on Electronic Imaging in the International Symposia Photonics West 1996, SPIE, San Jose, California USA, January 27 - February 2, 1996.

Abstract

This paper describes the design and performance of an object-oriented communication framework being developed by Kodak Health Imaging Systems and the Electronic Radiology Laboratory at Washington University School of Medicine. The framework is designed to meet the demands of Project Spectrum, which is a large-scale distributed electronic medical imaging system. 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, illustrate the performance of the framework over ATM, and describe how we resolved design challenges we faced when developing an object-oriented communication framework for distributed medical imaging.

1 Introduction

The demand for distributed electronic medical imaging systems (EMISs) is driven by technological advances and economic necessity [1]. 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 [2] 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 [3] reveal that for bulk data transfer, the performance overhead of widely used CORBA implementations on high-speed ATM networks is 25% to 40% 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 may impede the adoption of distributed object computing technologies. This is particularly problematic for performance-sensitive application domains including medical imaging, where the use of low-level tools increases development effort and reduces system reliability and flexibility.

To address this problem, we have developed an object-oriented communication software framework called "Blob Streaming." In this context, the term Blob refers to a "binary large object." Common examples of Blobs in a contemporary EMIS include CR, MR, and CT images. 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). Developers of EMIS communication software 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 developed at the socket level.

This paper is organized as follows: Section 2 outlines the

[1]This research is supported in part by Kodak Health Imaging Systems and the Electronic Radiology Laboratory at Washington University, St. Louis.

main features and design goals of the the Blob Streaming framework, Section 3 describes the key design challenges and how we resolved them, Section 4 illustrates the performance of Blob Streaming and compares it with alternative approaches over a high-speed ATM network, and Section 5 presents concluding remarks.

2 Overview of the Blob Streaming Framework

2.1 Motivation

The Blob Streaming framework is designed to meet the requirements of next-generation electronic medical imaging systems (EMISs). Figure 1 illustrates the general topology of a distributed EMIS. In this environment, various types of modalities (such as CT, MR, and CR) capture patient images and transfer them to an appropriate Blob Store. Radiologists use diagnostic workstations to retrieve these images for viewing and interpretation.

To support a wide spectrum of radiological workflow efficiently, the EMIS must be flexible. For example, supporting the "batch" workflow of conventional radiology may require caching Blobs on local disks of diagnostic workstations. However, supporting more dynamic types of workflow, (such as "radiology-on-demand") requires high-speed network access to centralized Blob Stores. The Blob Streaming framework provides application developers with a uniform means of operating on multiple types of Blob data residing on multiple types of Blob Stores.

2.2 Blob Streaming design goals and features

The Blob Streaming framework is designed to meet two key EMIS requirements: *flexibility* and *performance*. It is hard to achieve these goals simultaneously. Software designs that improve abstraction (such as encapsulation and layering) often reduce performance. For instance, excessive function call layering reduces locality of reference and foils CPU instruction and data caching strategies. This can result in high bus and memory overhead, which is relatively expensive on modern RISC workstations [4].

This section outlines and evaluates the primary features and design goals implemented in the Blob Streaming framework. Section 3 explores how we resolved key design challenges that arose during our development.

2.2.1 Enhance framework abstraction

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 abstract away from the following tasks:

• **Abstracting away from Blob location:** Consider the case of presenting an MR image to a radiologist on a diagnostic workstation. In a large-scale distributed EMIS, the image can be located anywhere throughout the system. The location of the image is typically determined using name servers and locators. Once the image has been located, it must be transported to the radiologist's workstation for display. Before being displayed to the radiologist, the image can be processed (*e.g.*, magnified, rotated, and edge-enchanced) for optimal presentation.

The location of the image can vary significantly. The image may exist on disk of a remote Blob Store; it may exist in memory of a modality (such as an Ultrasound scanner); it may also exist on the radiologist workstation's local disk. To enhance the usability of the system the image must be presented to the radiologist quickly. Thus, the Blob Streaming framework is responsible for selecting the optimal transfer techniques for each of these cases. For instance, if the data is stored locally in a file, Blob Streaming memory maps the file, thereby avoiding excessive mode switches and read/write buffering.

The primary advantage of decoupling Blob location from Blob operations is to *reduce software dependencies*. Application software that operates on Blobs does not depend on the location of the data.

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

The primary advantage of decoupling Blob type from Blob transfer is to *maximize software reuse*. 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 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. Note that 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 image. Since Blob Streaming treats

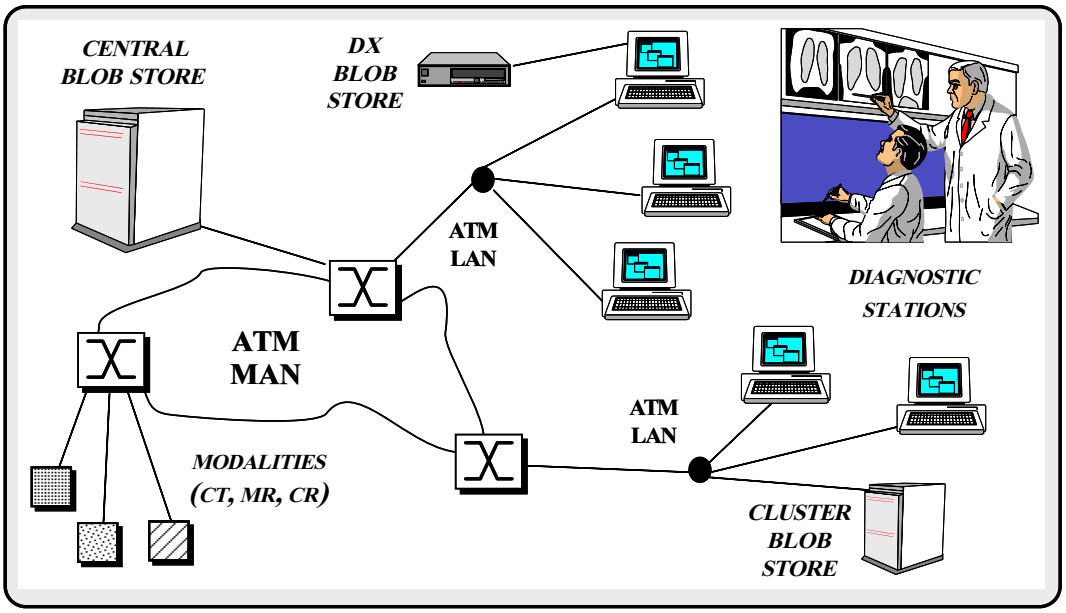

Figure 1: Topology of a distributed electronic medical imaging system.

all Blobs as untyped streams of data, images with integrated meta-data can be also be transferred easily.

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) [5], which has been used in high-speed communication protocol stacks. ILP optimizations significantly improve performance by overlapping communication and computation, as well as reducing memory bus traffic.

• **Abstracting away from Blob storage:** Blobs reside in "Blob Stores." 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 *Central Blob Stores* (which provide hierarchical storage management and support long-term archiving of Blobs), *Cluster Blob 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), *Workstation Blob Store* (which cache Blobs on the local disk of a diagnostic (DX) workstation), and *Memory Blob Stores* (which caches Blobs in workstation memory).

In the Blob Streaming framework the Blob Store interface supports operations to query for existing Blobs and to reserve space for creating new Blobs. The current Blob Streaming framework supports two types of Blob Stores: A File Store, which manages blobs on disk and a Memory Store, which manages Blobs in memory. 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.

The advantage of defining a uniform Blob Store interface is to *reduce software dependencies*. The Blob Streaming framework and its applications are decoupled from various types of storage (such as file, memory, and databases). Thus, application software can be written to store and retrieve images from Blob Stores, not 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 who are familiar with a particular database must learn the Blob Store interface to use Blob Streaming.

• **Abstracting away from OS-specific mechanisms:** The Blob Streaming framework shields applications from non-portable OS-specific features (such as event demultiplexing, threading, interprocess communication, and dynamic linking). 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 [6].

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 current version of Blob Streaming does not take advantage of native Windows NT mechanisms for overlapped I/O [7].

• **Abstracting from concurrency policies:** On multi-threaded operating systems like Solaris 5.x [8] or Windows NT [6], applications can use threads to simplify programming and take advantage of parallelism. Often, 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 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 vise versa). It is hard to avoid this tight coupling because reusable frameworks and applications may be developed without knowledge of the end system concurrency policies or hardware/software capabilities. The Blob Streaming framework is designed to decouple application software from dependency on concurrency policies. As discussed in Section 3.2, Blob Streaming accomplishes this by providing uniform callback-driven interfaces to both synchronous and asynchronous operations.

The advantage to this abstraction is increased flexibility and portability with respect to concurrency policies. The disadvantage to this approach is that the resulting uniform interface increases the complexity of synchronous calls in order to provide the needed concurrency independence.

• **Abstracting away from transport mechanism:** Blob Streaming presently uses a combination of CORBA and TCP/IP as data transport mechanisms. CORBA is used, primarily, for location and control operations, whereas TCP/IP is used for bulk data transfer. To shield applications from these low-level communication details, however, the public interface of Blob Streaming does not expose its internal transport mechanisms. In particular, CORBA is not visible to application programmers. This design allows transport mechanisms to be changed without affecting application software. For example, different implementations of CORBA can be used (such as ORBeline or Expersoft). Moreover, CORBA can be removed entirely and replaced with another mechanism (such as Network OLE, DCE RPC, or Sun RPC). We plan to optimize future versions of Blob Streaming to omit TCP/IP and use a lightweight transport protocol directly over ATM.

The primary advantages of decoupling the Blob Streaming public interface from its internal transport mechanisms are to *improve flexibility* and *enable transparent performance tuning*. The primary disadvantage is performance overhead of the extra level 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.

• **Abstracting from multiple event loops:** Complex EMIS applications must react to events from multiple sources such as DICOM toolkits, HL7 interface engines, GUI window events, and Blob Streaming transfers. Furthermore, the Blob Streaming library must handle socket level descriptor events, CORBA descriptor events, and timer events.

Frameworks such as X Windows and CORBA handle their respective events from their own event loops. In order for applications to use these tools efficiently, the multiple event loops must be integrated. We solved this problem by using ACE's Reactor [9] as the central event demultiplexor. The Reactor is an object-oriented interface to lower-level OS event demultiplexing operations (such as `select`, `poll`, and `WaitForMultipleObjects`) that react to descriptor events, timer events, and signal events. The Reactor provides a convenient solution to integrating the event demultiplexing and event handler dispatching components of multiple frameworks.

The advantage of integrating multiple event loops is that it allows developers to use Blob Streaming *while continuing to program 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 can be time consuming and tricky if the framework does not provide adequate hooks into its internal event demultiplexing logic.

2.2.2 Improve framework performance

As noted above, an EMIS must be flexible to handle many types of data and to adapt quickly to changes in requirements and in the hardware/software infrastructure. Distributed object computing middleware (such as CORBA) provides much of the flexibility required by an EMIS. However, current implementations of CORBA attain only one-half to two-thirds of the performance achievable by using lower-level mechanisms (like sockets) directly [3]. Achieving EMIS performance requirements is crucial because medical imaging is particularly bandwidth-intensive and delay-sensitive. Below, we outline our approach to this problem.

We address 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 since CORBA automatically generates code to marshall the parameters. Thus, for many types of inter-process communication, CORBA offers a powerful solution.

A disadvantage of CORBA is that current implementations incur significant performance overhead when used to transfer large amounts of data [3]. To avoid this overhead, Blob Streaming uses CORBA only as a "signaling mechanism" to negotiate TCP/IP connections for large transfers. This negotiation is a short-duration operation that exchanges a small amount of typed data and is therefore 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. To overcome these

inefficiencies, we use sockets to perform the bulk data transfers. Since Blob Streaming does not interpret the data it transfers, the untyped nature of socket-level data exchange is acceptable.

However, low-level network programming interfaces like sockets are hard to program because they have complex interfaces and are error prone. Our solution to this problem was to use C++ wrappers from the ACE toolkit [10] 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.

3 Resolving Design Challenges

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 and C++ language features.

3.1 Automating 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. 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 extensively to implement the Blob Streaming framework:

• **Strongly-typed interfaces:** In CORBA, all interfaces are defined using the CORBA interface definition language (IDL) [2]. A CORBA IDL compiler generates stubs and skeletons that translate IDL interface definitions into C++ classes. For instance, the following IDL interface definition describes a `BlobTransporter` that is used internally by the framework to control Blob transfer from a server to a client:

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

  // Transaction notification options.  These
  // options allow the framework to control blob
  // transfers acknowledgements.
  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...
```

Clients use the `BlobTransporter` to selectively request certain sections of a Blob. The ability to randomly access Blobs is useful when only the header information from the Blob is required or when a disrupted transaction must be restarted.

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 convert the typed information manually into a stream of untyped bytes. Moreover, the sender and receiver software for 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 using sockets would require copying the parameter values into a transfer buffer and then doing 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, where services can be located anywhere in a distributed system. Objects accessed may be remote, local (on the same host) or co-located (in the same address space). We used this feature in the Blob Streaming framework to shield applications from the location of Blob Stores where a Blob of interest resides. The Blob Streaming framework, however, violates this transparency to transfer the Blob efficiently. This violation occurs internally to the framework, however, and is not exposed to developers.

Applications need not know the location of a Blob Store to perform operations on their Blobs. The framework uses a Blob Location service to locate the appropriate Blob Store server, marshalls the request, and sends it to the server. If the Blob Store server is not running when a client sends the request, the ORB will automatically start the server.

3.2 Supporting uniform interfaces

The uniformity of features in the Blob Streaming framework was inspired by the System V Release 4 (SVR4) UNIX file system. SVR4 UNIX adapts a wide variety of disk and communication devices into a common set of operations (such as `open`, `close`, `read`, `write`, and `seek`). Unlike UNIX, however (which is implemented in C and provides C-level system call interfaces), Blob Streaming is implemented in C++. The use of C++ enforces encapsulation and leads to a more modular, extensible, and less error-prone programming interface.

In general, uniform interfaces are easier to work with than those containing many inconsistencies and special cases. With Blob Streaming, we found it useful to provide application developers with a programming interface whose operations behave uniformly irrespective of where the Blob actually resides or what type of Blob is being transfered. Therefore, Blob Store software that receives and stores MRI images to a database remains unchanged whether the MRI data is in memory, on a local file, in memory of a remote client, or on disk of a remote client.

Support for asynchronous and synchronous operations is another example of uniform interfaces in the Blob Streaming framework. Different applications require different types of operation invocation semantics from a framework. For instance, a multi-threaded server can simplify application software by using synchronous interfaces. Conversely, a single-threaded server that cannot afford to block on any one 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.

Switching between synchronous/return-value and asynchronous/callback interfaces requires changes to application software. Consider the case where a server that is implemented using multiple threads needs to be ported to a platform that does not support threads. If the software run by the threads uses synchronous interfaces, many changes are required to support asynchronous transactions through a single thread. To address this problem, 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 server performs an asynchronous Blob Streaming read that 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, when the synchronous call returns, the callback has already been executed. There are two advantages to this approach: (1) increased uniformity and (2) increased flexibility of concurrency strategies. 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 concurrency policies is merely toggling a flag. Therefore, no application software will change. This flexibility is particularly useful for developers of reuseable components who write software that can be used with a variety of concurrency strategies.

The disadvantage to this approach is that some developers may never want to program asynchronous operations. To address this issue, the Blob Streaming library offers wrappers around the synchronous callback operations to provide a synchronous/return-value API.

4 Blob Streaming Performance

Sections 2 and 3 outline and motivate the design of the Blob Streaming framework. This 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.

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 [11]. Each SPARCstation has 128 Mbytes of

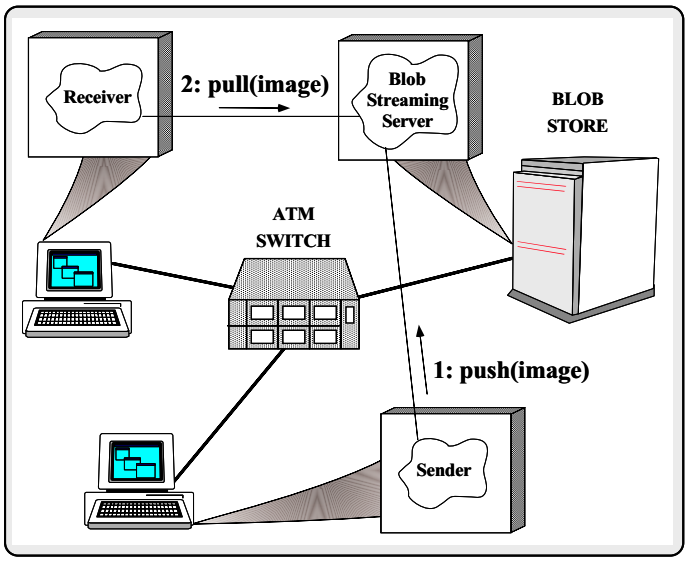

Figure 2: Push and pull models.

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 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 the test applications implement a "transaction" 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 merely flood the receiver with a continuous unidirectional stream of bytes.

The push and pull transaction models implemented by our test application are describe below:

• **The push model:** The push 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 precache data to a workstation. The push transaction model behaves as follows: (1) the client sends control data to the server characterizing the image being transferred from the client to the server (size and name of the image), (2) the client then sends the image data, and (3) the server sends a confirmation to the client on receiving all the data. This acknowledgement is necessary to insure end-to-end reliability of the transaction.

• **The pull model:** The pull model is representative of the use case where a workstation retrieves data from a Blob Store. The pull transaction model behaves as follows: (1) the client sends control data to the server characterizing the image the client wants from the server (size and name of the image) and (2) the server then sends the image data. Note that the pull model does not require an extra acknowledgement. Once the client receives the data that was requested from the server, the transaction is complete.

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

- **C version** – this is implemented completely in C. It uses C socket calls to transfer and receive the data and control messages via TCP/IP.
- **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 [10]. ACE encapsulates sockets with typesafe, portable, and efficient C++ interfaces.
- **CORBA version** – the Orbix implementation of CORBA was used: version 1.3 of Orbix from IONA Technologies. 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.
- **Blob Streaming version** – the Orbix implementation of CORBA was used for exchanging control messages and C++ wrappers for sockets provided by ACE were used for bulk data transfer.

All these versions test the push model. Due to space limitations, we have not included performance data for the pull model.

4.2 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. Previous test have shown that different versions of `ttcp` 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. Therefore, the Ethernet benchmarks were not included in these tests.

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 otherwise idle networks.

Figure 3 summarizes the performance results for all the benchmarks using 64 K and 8 K socket queues over a 155 Mbps ATM link. 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

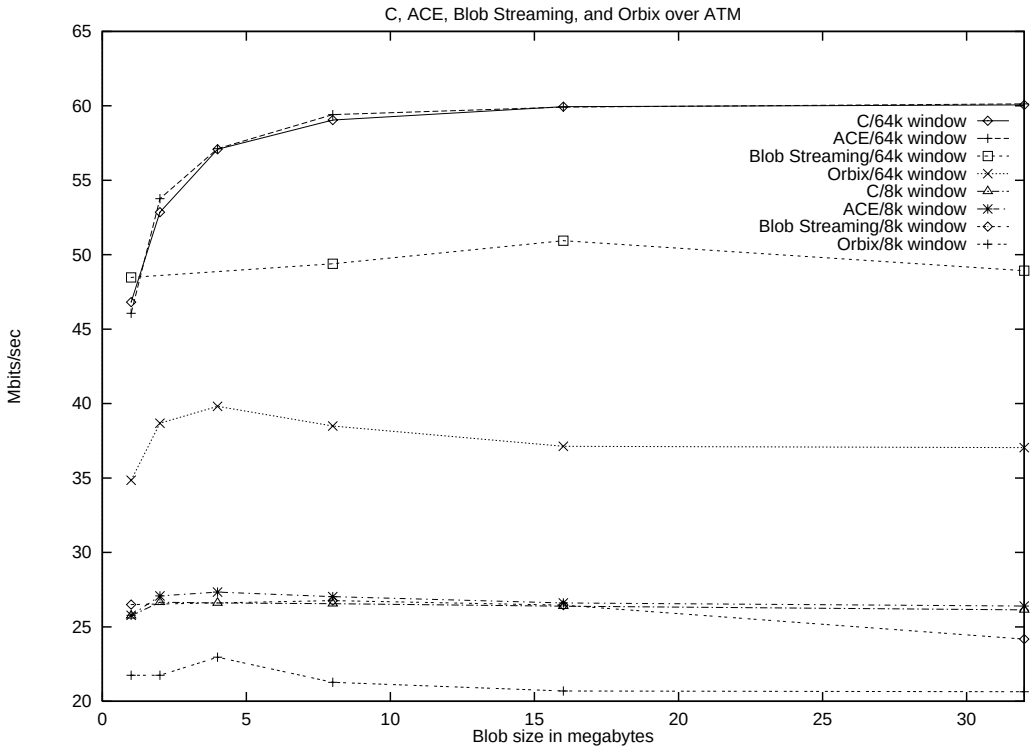

Figure 3: C, ACE C++, Blob Streaming, and Orbix performance over ATM.

for using the higher-level ACE C++ wrappers is insignificant, compared 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.[2] However, the Orbix CORBA versions peaked at around 66% of the C and C++ versions, reaching 40 Mbps with 64 K socket queues.

In addition to comparing the performance of the various transport mechanisms, Figure 3 also illustrates the generally low level of utilization of the ATM network. In particular, 60 Mbps represents only 40 percent 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* [12]. 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 [5]). This throughput preservation problem is exacerbated by contemporary implementations of distributed object computing middleware like CORBA, which copy data 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` [3] 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 3 illustrates the impact of socket queue size on throughput. Larger socket queues increase the TCP window size [13], which allows the transmission of multiple TCP segments back-to-back. Increasing the socket queue from 8 K to 64 K doubled performance from 28 Mbps to 60 Mbps. These results demonstrate the importance of having hooks to manipulate underlying OS mechanisms (such as transport layer and socket layer options). Communication frameworks that do not offer these hooks to application developers are destined to perform poorly over high-speed networks.

5 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.

The primary objective of Project Spectrum is to link the stand-alone heterogeneous computer systems of 15 acute care facilities, as well as over 5,500 physicians, in the BJC system into a single integrated network. Key system requirements

[2] Subsequent code profiling revealed that the Blob Streaming receiver was performing an unnecessary data copy. With this copy removed, the performance of Blob Streaming should be comparable to the C and ACE C++ wrapper versions.

are to support seamless electronic access to clinical expertise from any point in the system. Other requirements call for immediate, on-line access to information via advanced clinical workstations attached to high-speed networks, teleradiology and remote consultation capabilities, and practice management support tools.

Distributed electronic medical imaging systems like Project Spectrum require high-performance bulk data communication. Existing implementations of higher-level distributed object computing middleware like CORBA do not provide adequate performance for bulk data transfer due to data copying, demultiplexing, and memory management overhead. 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 [14].

The Blob Streaming framework described in this paper provides more efficient data transfer than using CORBA as the sole bulk data transport mechanism. Blob Streaming uses CORBA as a control mechanism to negotiate endpoints of TCP/IP communication in a location-independent manner. The lower-level C++ wrappers for sockets are then used to establish point-to-point TCP connections and transmit bulk data efficiently across the connections. This strategy builds on the strengths of both CORBA and sockets. It shields application developers from lower-level details of sockets without incurring the performance overhead associated with using a CORBA-only solution.

Blob Streaming uses sockets to achieve the performance of lower-level networking tools 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 concurrency OS-specific mechanisms, concurrency policies, and event loops.

Thanks to Chris Tarr of ObjectSpace for his contributions to the design of Blob Streaming.

References

[1] G. Blaine, M. Boyd, and S. Crider, "Project Spectrum: Scalable Bandwidth for the BJC Health System," *HIMSS, Health Care Communications*, pp. 71–81, 1994.

[2] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 (draft) ed., May 1995.

[3] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the Conference on Object-Oriented Technologies*, (Monterey, CA), USENIX, June 1995.

[4] P. Druschel, M. B. Abbott, M. Pagels, and L. L. Peterson, "Network subsystem design," *IEEE Network (Special Issue on End-System Support for High Speed Networks)*, vol. 7, July 1993.

[5] D. D. Clark and D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 200–208, ACM, Sept. 1990.

[6] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[7] D. C. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms," in *Proceedings of the* 9^{th} *European Conference on Object-Oriented Programming*, (Aarhus, Denmark), ACM, August 1995.

[8] 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," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[9] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[10] D. C. Schmidt, "ASX: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[11] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

[12] D. C. Schmidt and T. Suda, "Transport System Architecture Services for High-Performance Communications Systems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 489–506, May 1993.

[13] K. Modeklev, E. Klovning, and O. Kure, "TCP/IP Behavior in a High-Speed Local ATM Network Environment," in *Proceedings of the* 19^{th} *Conference on Local Computer Networks*, (Minneapolis, MN), pp. 176–185, IEEE, Oct. 1994.

[14] 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, pp. 43–48, February 1995.