The Performance of Object-Oriented Components for High-speed Network Programming

Douglas C. Schmidt
schmidt@cs.wustl.edu
Washington University, St. Louis

Introduction

- Distributed object computing (DOC) frameworks are well-suited for certain communication requirements and certain network environments
  - e.g., request/response or oneway messaging over low-speed Ethernet or Token Ring

- However, current DOC implementations exhibit high overhead for other types of requirements and environments
  - e.g., bandwidth-intensive and delay-sensitive streaming applications over high-speed ATM or FDDI

Outline

- Outline communication requirements of distributed medical imaging domain

- Compare performance of several network programming mechanisms:
  - Sockets
  - ACE C++ wrappers
  - CORBA (Orbix)
  - Blob Streaming

- Outline Blob Streaming Architecture and Related Patterns

- Evaluation and Recommendations
Distributed Objects in Medical Imaging Systems

- Blob Servers have the following responsibilities and requirements:
  * Efficiently store/retrieve large medical images (Blobs)
  * Respond to queries from Blob Locators
  * Manage short-term and long-term blob persistence

Motivation for Distributed Object Computing

- Simplify application development and interworking, e.g.,
  - CORBA provides higher level integration than traditional “untyped TCP byte streams”
  - ACE encapsulates lower-level networking and concurrency systems programming interfaces

- Provide a foundation for higher-level application collaboration
  - e.g., Windows OLE and the OMG Common Object Service Specification (COSS)

- Benefits for distributed programming similar to OO languages for non-distributed programming
  - e.g., encapsulation, interface inheritance, and object-based exception handling
**CORBA Components**

- The CORBA specification is comprised of several parts:
  1. An Object Request Broker (ORB)
  2. An Interface Definition Language (IDL)
  3. A Static Invocation Interface (SII)
  4. A Dynamic Invocation Interface (DII)
  5. A Dynamic Skeleton Interface (DSI)

- Other documents from OMG describe common object services built upon CORBA
  - *e.g.*, CORBAServices → *Event services, Name services, Lifecycle services*

**Motivation for CORBA and ACE on Project Spectrum**

- Two crucial issues for overall communication infrastructure *flexibility* and *performance*

- Flexibility motivates the use of a distributed object computing framework like CORBA to transport many formats of data
  - *e.g.*, HL7, DICOM, Blobs, domain objects, etc.

- Performance requires we transport this data as quickly as the current technology allows

**Key Research Question**

*Can CORBA and ACE be used to transfer medical images efficiently over high-speed networks?*

- Our goal was to determine this empirically *before* adopting distributed object computing wholesale
Performance Experiments

- Enhanced version of TTCP
  - TTCP measures end-to-end bulk data transfer with acknowledgements
  - Enhanced version tests C, ACE C++ wrappers, and CORBA, and Blob Streaming

- Parameters varied
  - 100 Mbytes of data transferred in various chunk sizes
  - Socket queues were 8k (default) and 64k (maximum)
  - Network was 155 Mbps ATM

- Compiler was SunC++ 4.0.1 using highest optimization level

TTCP Configuration for C and ACE C++ Wrappers

TTCP Configuration for CORBA Implementation
TTCP Configuration for Blob Streaming

Primary Sources of Overhead

- Data copying
- Demultiplexing
- Memory allocation
- Presentation layer formatting

Performance over ATM

High-Cost Functions

- C and ACE C++ Tests
  - Transferring 64 Mbytes with 1 Mbyte buffers

<table>
<thead>
<tr>
<th>Test</th>
<th>%Time</th>
<th>Calls</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>C sockets 93.9</td>
<td>112</td>
<td>write (sender) 3.6</td>
<td>110 read</td>
</tr>
<tr>
<td>C sockets 93.2</td>
<td>13,085</td>
<td>read (receiver) 4.5</td>
<td>3 getmsg</td>
</tr>
<tr>
<td>ACE C++ wrapper 94.4</td>
<td>112</td>
<td>write (sender) 3.2</td>
<td>110 read</td>
</tr>
<tr>
<td>ACE C++ wrapper 93.9</td>
<td>12,984</td>
<td>read (receiver) 5.6</td>
<td>3 getmsg</td>
</tr>
</tbody>
</table>
High-Cost Functions (cont’d)

- Orbix String and Sequence

<table>
<thead>
<tr>
<th>Test</th>
<th>%Time</th>
<th>Calls</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orbix Sequence</td>
<td>53.5</td>
<td>127</td>
<td>write</td>
</tr>
<tr>
<td>(sender)</td>
<td>35.1</td>
<td>223</td>
<td>read</td>
</tr>
<tr>
<td></td>
<td>7.3</td>
<td>1,108</td>
<td>memcpy</td>
</tr>
<tr>
<td>Orbix Sequence</td>
<td>85.6</td>
<td>12,846</td>
<td>read</td>
</tr>
<tr>
<td>(receiver)</td>
<td>12.4</td>
<td>1,064</td>
<td>memcpy</td>
</tr>
<tr>
<td>Orbix String</td>
<td>45.0</td>
<td>127</td>
<td>write</td>
</tr>
<tr>
<td>(sender)</td>
<td>35.1</td>
<td>223</td>
<td>read</td>
</tr>
<tr>
<td></td>
<td>10.8</td>
<td>1,315</td>
<td>strlen</td>
</tr>
<tr>
<td></td>
<td>6.0</td>
<td>1,108</td>
<td>memcpy</td>
</tr>
<tr>
<td>Orbix String</td>
<td>70.7</td>
<td>12,443</td>
<td>read</td>
</tr>
<tr>
<td>(receiver)</td>
<td>18.1</td>
<td>2,142</td>
<td>strlen</td>
</tr>
<tr>
<td></td>
<td>10.0</td>
<td>1,064</td>
<td>memcpy</td>
</tr>
</tbody>
</table>

High-Cost Functions (cont’d)

- Blob Streaming

<table>
<thead>
<tr>
<th>Test</th>
<th>%Time</th>
<th>Calls</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob Streaming</td>
<td>48.8</td>
<td>327</td>
<td>write</td>
</tr>
<tr>
<td>(sender)</td>
<td>44.8</td>
<td>232</td>
<td>read</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>2,055</td>
<td>memcpy</td>
</tr>
<tr>
<td>Blob Streaming</td>
<td>77.2</td>
<td>12,846</td>
<td>read</td>
</tr>
<tr>
<td>(receiver)</td>
<td>16.4</td>
<td>12,734</td>
<td>memcpy</td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>202</td>
<td>write</td>
</tr>
</tbody>
</table>

Overview of Blob Streaming

- Blob Streaming provides developers with a uniform interface for operations on multiple types of Binary Large Objects (BLOBs)

- Two primary goals
  1. Improved abstraction
     - Shield developers from knowledge of blob location (e.g., memory vs. “local” files vs. remote network)
  2. Maximize performance
     - Transport blobs as efficiently as current technology allows
Blob Streaming Architecture

- Blob Streaming components allow transparent use of resources through uniform blob interfaces.

- Blob Streaming support the following:
  - **Blob location**
    - *e.g.*, smart caches to decouple transfers from location algorithms
  - **Blob routing**
    - *e.g.*, context based routing
  - **Source and destination independent Blob transport**, *e.g.*
    - Store and retrieve from remote or local databases
    - Abstract operations like reads/writes may use local file reads/writes, or remote reads/writes via sockets

Blob Streaming Architecture

### Design Goals

- **Goal**: decouple application from OS platform
  - *e.g.*, applications can be shielded from fact that current version is implemented for UNIX
  - Thus, can port Blob Streaming to Windows NT or OS/2 without changing applications
  - Platform specific operations hidden behind abstract interfaces
    - *e.g.*, WIN32 WaitForMultipleObjects and UNIX select

- **Advantages**
  - Portability and extensibility

Blob Streaming Architecture

### Design Goals (cont’d)

- **Goal**: application independence from transport mechanism
  - Switch transports at any stage in the development without affecting application code
  - *Presently using CORBA and TCP/IP as transport mechanisms*
    - However, none of these mechanisms are exposed to programmers
    - *e.g.*, can use Network OLE
    - As transport technology improves, Blob Streaming can change without affecting applications
      - *e.g.*, “direct ATM”

- **Advantages**
  - Portability, extensibility, and performance tuning

Design Patterns in Blob Streaming

- **Blob Streaming is based upon a system of design patterns**

---

### Design Patterns

![Design Patterns Diagram]

**TACTICAL PATTERNS**
- Factory
- Strategy
- Bridge
- Iterator
- Adapter

**STRATEGIC PATTERNS**
- Connector
- Acceptor
- Service Configurator
- Service Stream
- Reactor
- Thread Specific Storage
- External Polymorphism
The Reactor Pattern

- **Intent**
  - An object behavioral pattern that decouples event demultiplexing and event handler dispatching from the services performed in response to events

- This pattern resolves the following forces for event-driven software:
  - How to demultiplex multiple types of events from multiple sources of events efficiently within a single thread of control
  - How to extend application behavior without requiring changes to the event dispatching framework

- Participants in the Reactor pattern
The Active Object Pattern

- **Intent**
  - Decouples method execution from method invocation and simplifies synchronized access to shared resources by concurrent threads

- This pattern resolves the following forces for concurrent communication software:
  - How to allow blocking operations (such as read and write) to execute concurrently
  - How to simplify concurrent access to shared state

Structure of the Active Object Pattern in ACE

Collaboration in ACE Active Objects

Using the Active Object Pattern for Blob Streaming
**Half-Sync/Half-Async Pattern**

- **Intent**
  - An architectural pattern that decouples synchronous I/O from asynchronous I/O in a system to simplify programming effort without degrading execution efficiency

- This pattern resolves the following forces for concurrent communication systems:
  - *How to simplify programming for higher-level communication tasks*
    - These are performed synchronously (via Active Objects)
  - *How to ensure efficient lower-level I/O communication tasks*
    - These are performed asynchronously (via the Reactor)

---

**Collaborations in the Half-Sync/Half-Async Pattern**

- This illustrates *input processing (output processing is similar)*

---

**Using the Half-Sync/Half-Async Pattern for Blob Streaming**

---

**Structure of the Half-Sync/Half-Async Pattern**

---
The Acceptor Pattern

- **Intent**
  - Decouple the passive initialization of a service from the tasks performed once the service is initialized.

- This pattern resolves the following forces for network servers using interfaces like sockets or TLI:
  1. *How to reuse passive connection establishment code for each new service*
  2. *How to make the connection establishment code portable across platforms that may contain sockets but not TLI, or vice versa*
  3. *How to ensure that a passive-mode descriptor is not accidentally used to read or write data*
  4. *How to enable flexible policies for creation, connection establishment, and concurrency*

Collaboration in the Acceptor Pattern

- Acceptor factory creates, connects, and activates a Svc_Handler.
Evaluation and Recommendations

- Understand communication requirements and network/host environments

- Measure performance empirically before adopting a communication model
  - Low-speed networks often hide performance overhead

- Insist CORBA implementors provide hooks to manipulate options
  - e.g., setting socket queue size with ORBeline was hard

- Increase size of socket queues to largest value supported by OS

- Tune the size of the transmitted data buffers to match MTU of the network

Evaluation and Recommendations (cont’d)

- Use IDL sequences rather than IDL strings to avoid unnecessary data access (i.e. strlen)

- Use write/read rather than send/recv on SVR4 platforms

- Long-term solution:
  - Optimize DOC frameworks
  - Add streaming support to CORBA specification

- Near-term solution for CORBA overhead on high-speed networks:
  - e.g., Blob Streaming integrates CORBA with ACE

Optimizations

- To be effective for use with performance-critical applications over high-speed networks, CORBA implementations must be optimized

Obtaining ACE

- The ADAPTIVE Communication Environment (ACE) is an OO toolkit designed according to key network programming patterns

- All source code for ACE is freely available
  - Anonymously ftp to warchive.wustl.edu
  - Transfer the files /languages/c++/ACE/* .gz and gnu/ACE-documentation/* .gz

- Mailing list
  - ace-users@cs.wustl.edu
  - ace-users-request@cs.wustl.edu

- WWW URL