A High-Performance Architcture for Distributed Object Computing

Douglas C. Schmidt

http://www.cs.wustl.edu/∼schmidt/

schmidt@cs.wustl.edu

Washington University, St. Louis

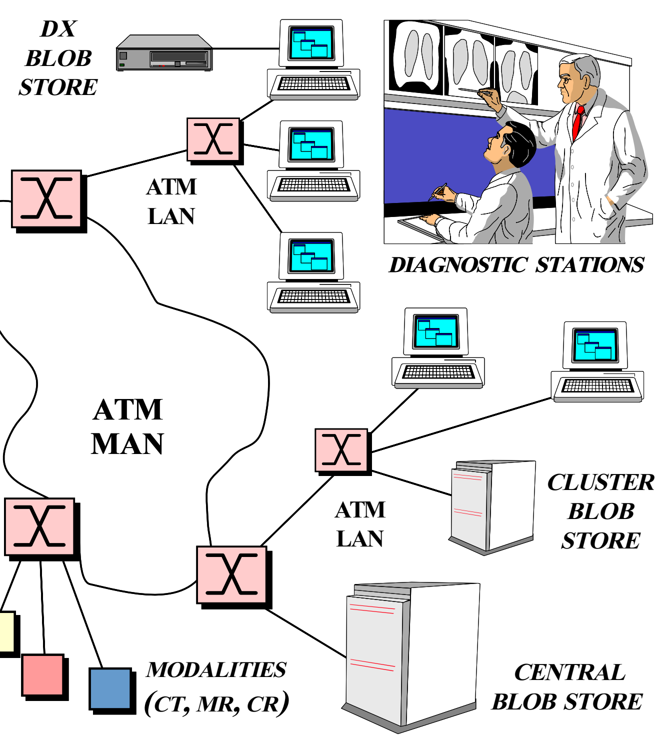
Introduction

- 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

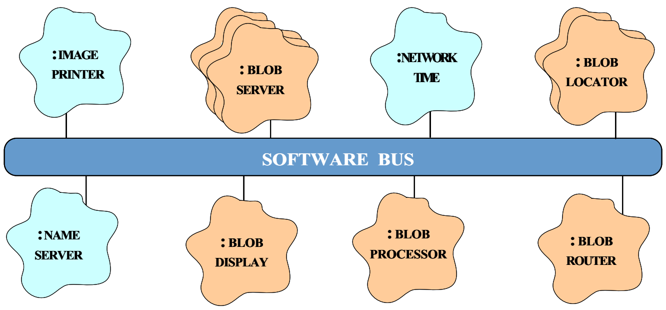
Outline of Talk

- Outline communication requirements of distributed medical imaging domain
- Compare performance of several network programming mechanisms:
  - Sockets
  - ACE C++ wrappers
  - Two CORBA implementations (ORBeline and Orbix)
- Discuss how to utilize distributed object computing frameworks efficiently and effectively
- Describe general principles for designing high-performance object-oriented network programming interfaces

Distributed Medical Imaging

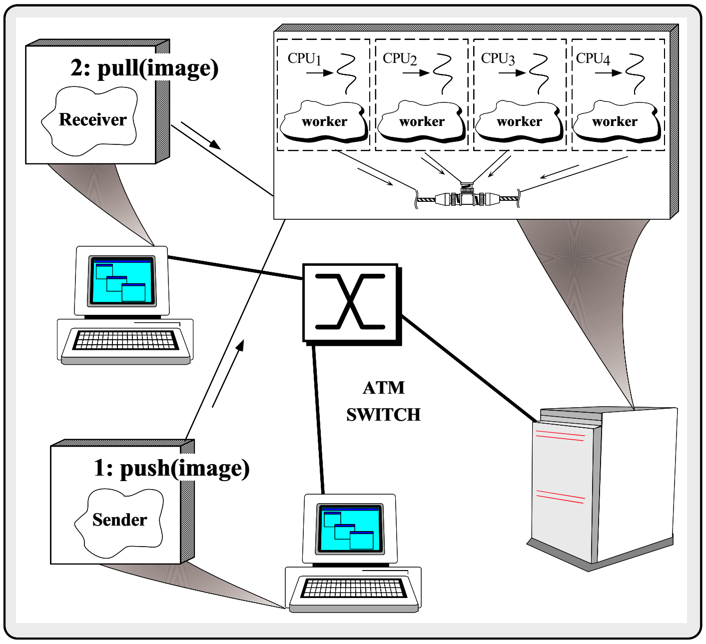


Distributed Objects in Medical Imaging Systems



- Image servers have the following responsibilities and requirements:
  * *Efficiently store/retrieve large medical images*
  * *Respond to queries from Image Locator Servers*
  * *Manage short-term and long-term image persistence*

Image Server System Architecture



Motivation for CORBA

- Simplifies application interworking
  - CORBA provides higher level integration than traditional “untyped TCP bytestreams”
- Provides a foundation for higher-level distributed object 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 Overview

- CORBA specifies the following functions of an *Object Request Broker* (ORB)
  - *Interface Definition Language* (CORBA IDL)
  - *A mapping from CORBA IDL onto C, C++, and Smalltalk*
  - *An Interface Repository*
    * Contains meta-info that can be queried at run-time
  - *A Dynamic Invocation Interface*
    * Used to compose method requests at run-time
  - *A Basic Object Adaptor* (BOA)
    * Allows developers to integrate their objects with an ORB

CORBA Services

- CORBA provides the following mechanisms
  - *Parameter marshalling*
  - *Object location*
  - *Object activation*
  - *Replication and fault tolerance*
- COSS extends CORBA to provide services like
  - *Event services*
  - *Naming services*
  - *Transactions*
  - *Object lifecycle management*

Key Research Question

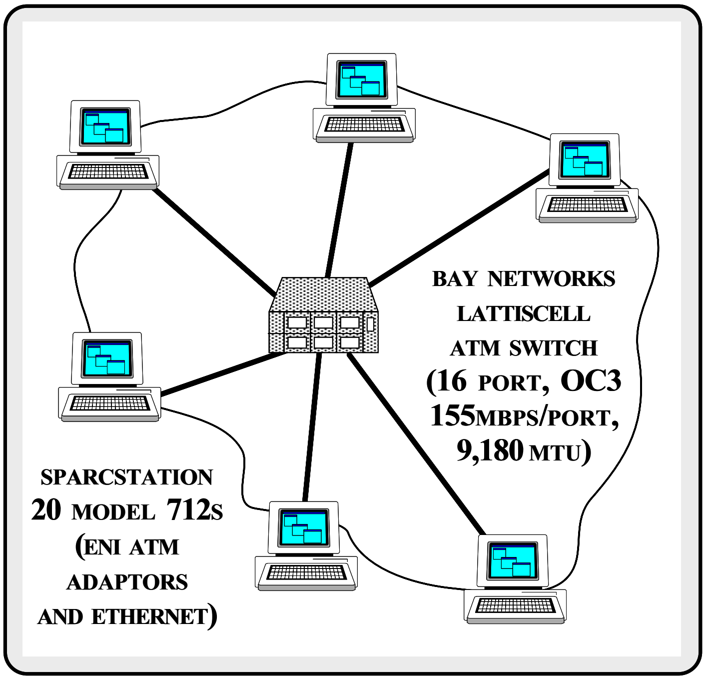
Can CORBA be used to transfer medical images efficiently over high-speed networks?

- Our goal was to determine this empirically *before* adopting the CORBA communication model wholesale

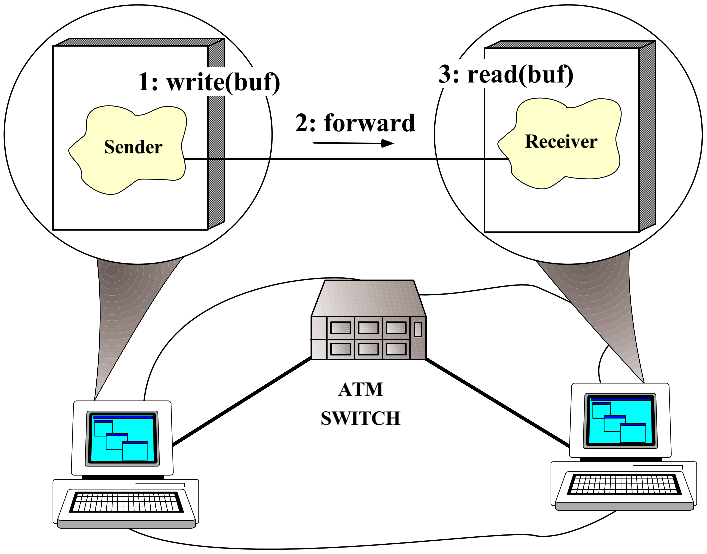
Performance Experiments

- Enhanced version of TTCP
  - TTCP measures end-to-end, oneway bulk data transfer
  - Enhanced version tests C, ACE C++ wrappers, and CORBA
- Parameters varied
  - 64 Mbytes of data buffers ranging from 1 Kbyte to 128 Kbyte (by powers of 2)
  - Socket queues were 8k (default) and 64k (maximum)
  - Networks were 155 Mbps ATM and 10 Mbps Ethernet
- Compiler was SunC++ 4.0.1 using highest optimization level

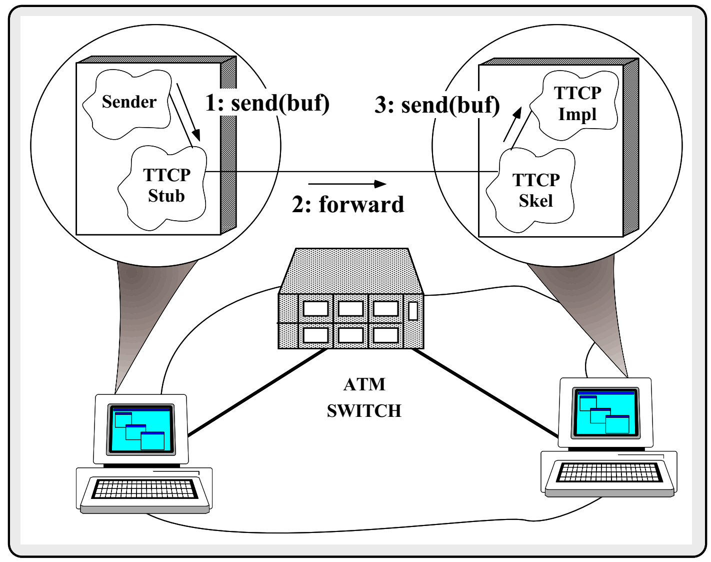
Network/Host Environment



TTCP Configuration for C and ACE C++ Wrappers



TTCP Configuration for CORBA Implementations



CORBA Implementations

- 2 implementations of TTCP using 2 versions of CORBA
  - IDL `string` and IDL `sequence`

```
typedef sequence<char> ttcp_sequence;

interface TTCP_Sequence
{
  oneway void send (in ttcp_sequence ttcp_seq);
};

interface TTCP_String
{
  oneway void send (in string ttcp_string);
};
```

  - Orbix 1.3 and ORBeline 1.2
    - Couldn't directly reuse source code since neither ORB supported same IDL → C++ mapping
    - Also, neither ORB supported CORBA 2.0

CORBA Sender Implementation

- Obtain reference to target objects via `_bind` factory:

```
// Use locator service to acquire bindings.
TTCP_String *t_str = TTCP_String::_bind ();
TTCP_Sequence *t_seq = TTCP_Sequence::_bind ();

// ...

// String transfer.

char *buffer = new char[buffer_size];
// Initialize data in char * buffer...

while (--buffers_sent >= 0)
  t_str->send (buffer);

// Sequence transfer.

ttcp_sequence sequence_buffer;
// Initialize data in TTCP_Sequence buffer...

while (--buffers_sent >= 0)
  t_seq->send (sequence_buffer);
```

CORBA Receiver Implementation

- Implementation class for IDL interface that inherits from automatically-generated CORBA skeleton class

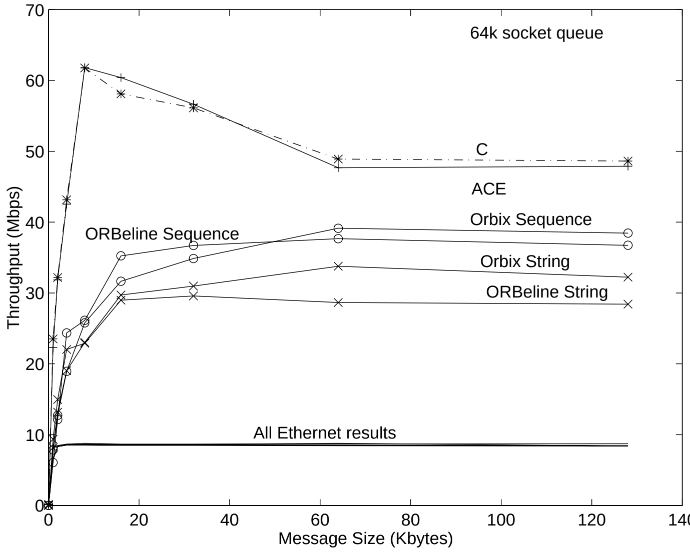
```
class TTCP_Sequence_i
  : virtual public TTCP_SequenceBOAImpl
{
public:
  TTCP_Sequence_i (void): nbytes_ (0) {}

  // Upcall invoked by the CORBA skeleton.
  virtual void send (const ttcp_sequence &ttcp_seq,
                     CORBA::Environment &IT_env)
  {
    this->nbytes_ += ttcp_seq._length;
  }
  // ...

private:
  // Keep track of bytes received.
  u_long nbytes_;
};
```


CORBA Receiver Main

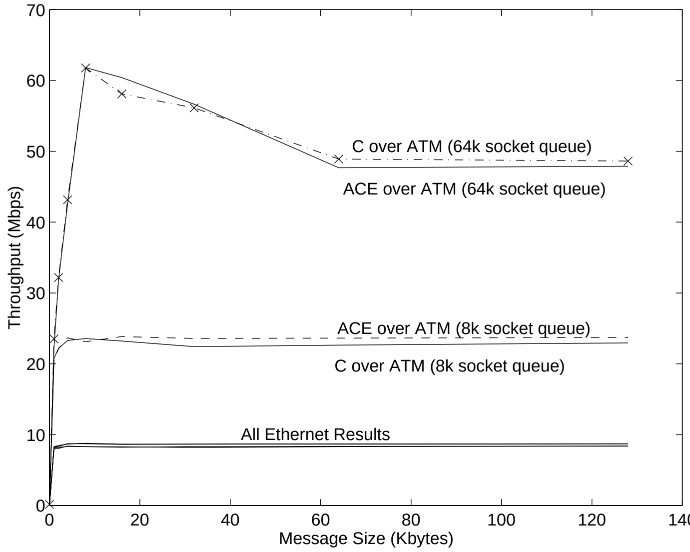
- Initializes object implementations and goes into CORBA event loop

```
int main (int argc, char *argv[])
{
  // Implements the Sequence object.
  TTCP_Sequence_i ttcp_sequence;

  // Implements the String object.
  TTCP_String_i ttcp_string;

  // Tell the ORB that the objects are active.
  CORBA::BOA::impl_is_ready ();

  /* NOTREACHED */
  return 0;
}
```

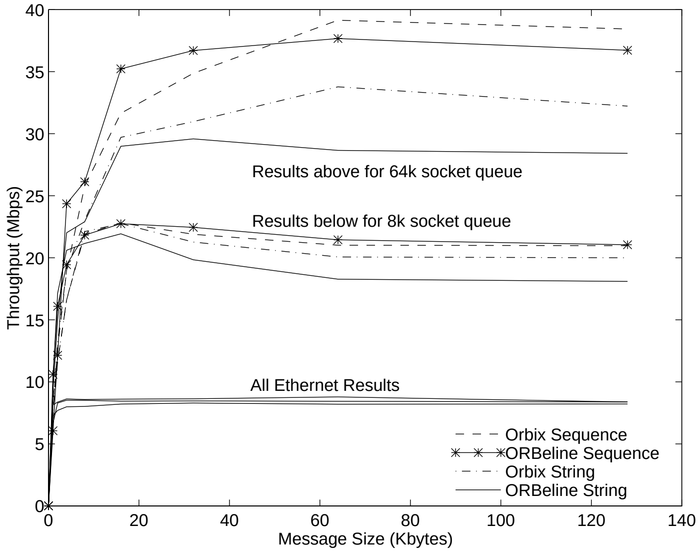

Performance over ATM and Ethernet



C and ACE Performance over ATM and Ethernet



Orbix and ORBeline Performance over ATM and Ethernet



Primary Sources of Overhead for CORBA

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

High-Cost Functions

- C and ACE C++ Tests
  - Transferring 64 Mbytes with 128 Kbyte buffers

Test	%Time	#Calls	msec/call	Name
C sockets (sender)	99.6	527	92.8	_write
C sockets (receiver)	99.3	7201	6.2	_read
ACE C++ wrapper (sender)	99.4	527	87.3	_write
ACE C++ wrapper (receiver)	99.6	7192	6.2	_read

High-Cost Functions (cont'd)

- Orbix String and Sequence Tests

Test	%Time	#Calls	msec/call	Name
Orbix Sequence	94.6	532	89.1	_write
(sender)	4.1	2121	1.0	memcpy
Orbix Sequence	92.7	7860	6.1	_read
(receiver)	4.8	2581	0.6	memcpy
Orbix String	89.0	532	85.6	_write
(sender)	4.6	2121	1.1	memcpy
	4.1	2700	0.7	strlen
Orbix String	86.3	7744	5.7	_read
(receiver)	5.5	6740	0.4	strlen
	4.5	2581	0.9	memcpy

High-Cost Functions (cont'd)

- ORBeline String and Sequence Tests

Test	%Time	#Calls	msec/call	Name
ORBeline Sequence	91.0	551	74.9	_write
(sender)	5.2	6413	0.4	memcpy
	1.8	1032	0.8	__sigaction
ORBeline Sequence	89.0	7568	5.8	_read
(receiver)	5.1	7222	0.3	memcpy
	3.3	1071	1.5	_poll
ORBeline String	83.8	551	83.9	_write
(sender)	5.4	920	3.2	strcpy
	4.3	5901	0.4	memcpy
	3.9	1728	1.2	strlen
	1.1	1032	0.6	__sigaction
ORBeline String	85.4	7827	5.5	_read
(receiver)	4.6	6710	0.3	memcpy
	4.2	1702	1.3	strlen
	2.8	1071	1.3	_poll

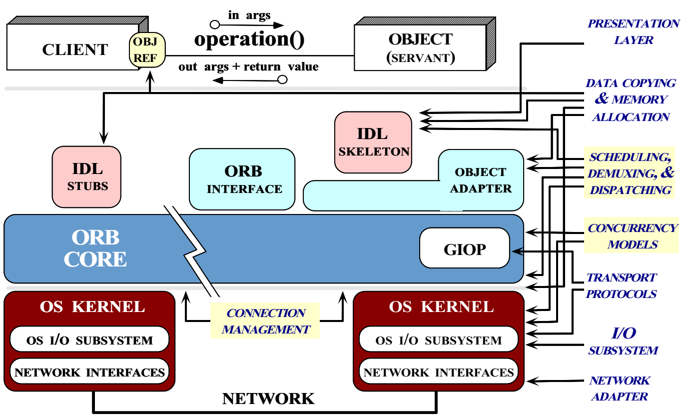
Evaluation and Recommendations

- 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 and copying
- 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:
  - Integrate DOC frameworks with OO encapsulation of network programming interfaces

Optimizations



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

- Key optimization points are illustrated above

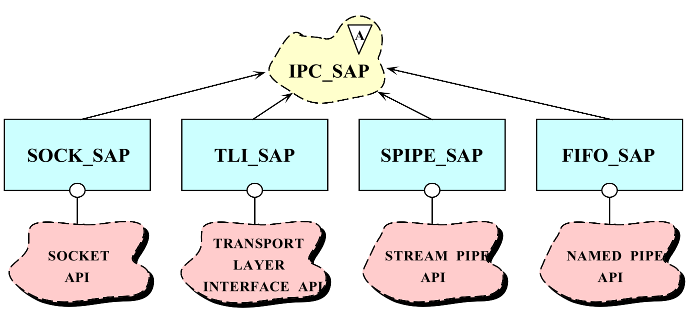
Network Programming Alternatives

- Developers of high-performance streaming applications traditionally had two alternatives:

  1. *Use higher-level, but less efficient network programming interfaces*
     – *e.g.*, DOC frameworks or RPC toolkits
  2. *Use lower-level, but more efficient network programming interfaces*
     – *e.g.*, sockets, TLI, Win32 named pipes

- ACE C++ wrappers represent a midpoint in the solution space
  – *e.g.*, improve *correctness*, *programming simplicity*, *portability*, and *reusability* without sacrificing *performance*

ACE C++ Wrappers



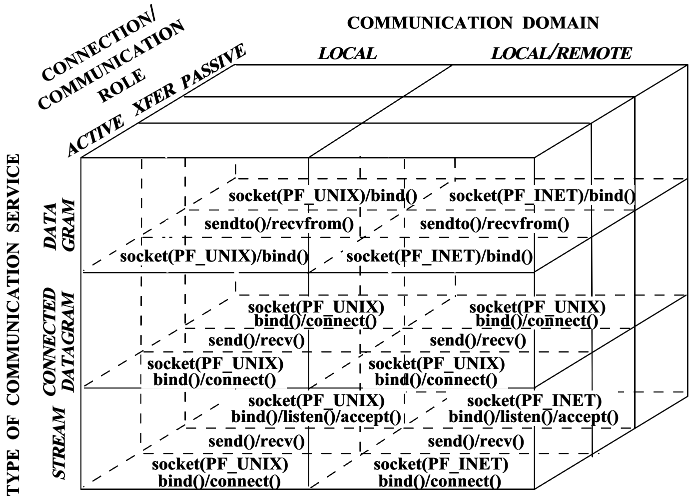
- `IPC_SAP` is an OO "middleware" API that encapsulates key network programming interfaces
- It makes programming at the transport layer much less tedious and error prone...

Limitations with Sockets

socket()
bind()
connect()
listen()
accept()
read()
write()
readv()
writev()
recv()
send()
recvfrom()
sendto()
recvmsg()
sendmsg()
setsockopt()
getsockopt()
getpeername()
getsockname()
gethostbyname()
getservbyname()

- The socket API is *one-dimensional* rather than *hierarchical*
  – Thus, it doesn't give any hints on how to use it correctly
- There is no consistency among names
- High potential for errors due to weak type-checking

Socket Taxonomy



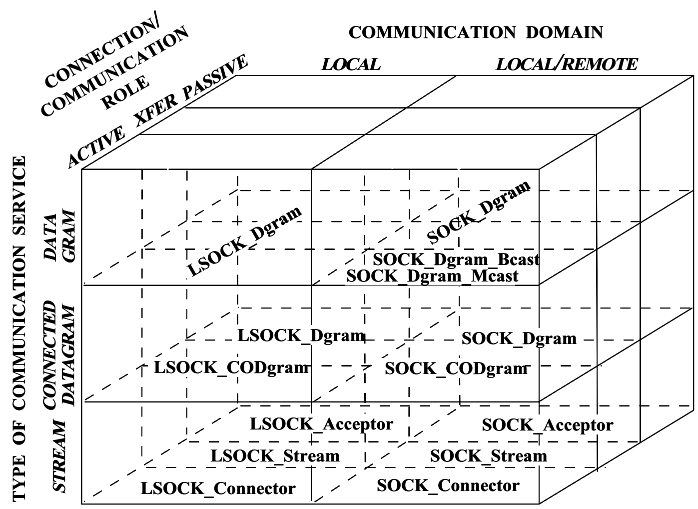
Common Socket Bugs

```
int echo_server (u_short port) {
  sockaddr_in s_addr;
  int s_sd, n_sd, len; // (1) uninitialized variable.
  char buf[BUFSIZ];
  // Create a local endpoint of communication.
  s_sd = socket (PF_UNIX, SOCK_DGRAM, 0);
  // Set up address information to become a server.
  // (2) forgot to "zero out" structure first...
  // (3) used the wrong address family ...
  s_addr.sin_family = AF_INET;
  // (4) forgot to use htons() on port...
  s_addr.sin_port = port;
  s_addr.sin_addr.s_addr = INADDR_ANY;
  bind (s_sd, &s_addr, sizeof s_addr) == -1)
  // Create a new endpoint of communication.
  // (5) can't accept() on a SOCK_DGRAM.
  // (6) Omitted a crucial set of parens...
  if (n_sd = accept (s_sd, &s_addr, &len) == -1) {
    int n;
    // (6) Omitted another set of parens...
    // (7) error to read from s_sd.
    while (n = read (s_sd, buf, sizeof buf) > 0)
      // (8) forgot to check for "short-writes".
      write (n_sd, buf, n);
    // Remainder omitted...
  }
}
```


SOCK_SAP Overview

- ACE SOCK_SAP socket wrappers are designed to improve:
  - *Correctness*
    - Operations are "type-safe"
  - *Ease of Learning/Ease of Use*
    - Combine related operations
    - Use default values for standard operations
  - *Portability/Reusability*
    - Map single OO API onto multiple underlying OS APIs
  - *Extensibility*
    - Developers may extend existing components via inheritance

SOCK_SAP Taxonomy



SOCK_SAP Class Interfaces

```
class SOCK_Connector : public SOCK
{
public:
  int connect (SOCK_Stream &new_sap, const Addr &remote_addr,
               int blocking);
  // ...

class SOCK_Acceptor : public SOCK
{
public:
  SOCK_Acceptor (const Addr &local_addr);
  int accept (SOCK_Stream &new_sap) const;
  //...

class SOCK_Stream : public SOCK
{
public:
  int send_n (const void *buf, int n);
  int recv (void *buf, int n, int &recvd);
  int close (void);
  // ...

class INET_Addr : public Addr
{
public:
  INET_Addr (u_short port, const char host[]);
  // ...
```


SOCK_SAP Revision of Echo Server

```
template <class ACCEPTOR, class STREAM, class ADDR>
int echo_server (u_short port)
{
  // Local address of server.
  ADDR s_addr (port);
  // Remote address object.
  ADDR addr;

  // Initialize the passive mode server.
  ACCEPTOR acceptor (s_addr);

  // Data transfer object.
  STREAM stream;

  // Accept a new connection.
  if (acceptor.accept (stream, &addr) != -1) {
    char buf[BUFSIZ];
    for (size_t n; stream.recv (buf, sizeof buf, n) > 0;)
      if (stream.send_n (buf, n) != n)
        // Remainder omitted.
  }
}
// ...
echo_server<SOCK_Acceptor, SOCK_Stream, INET_Addr> (port);
```


ACE Wrapper Design Principles

- The following principles applied throughout the ACE wrappers:
  - *Enforce typesafety at compile-time*
  - *Allow controlled violations of typesafety*
  - *Simplify for the common case*
  - *Replace one-dimensional interfaces with hierarchical class categories*
  - *Enhance portability with parameterized types*
  - *Inline performance critical methods*
  - *Define auxiliary classes to hide error-prone details*

Enforce Typesafety at Compile-Time

- Sockets cannot detect certain errors at compile-time, *e.g.*,

```
int s_sd = socket (PF_INET, SOCK_STREAM, 0);
// ...
bind (s_sd, ...); // Bind address.
listen (s_sd); // Make a passive-mode socket.

// Error not detected until run-time.
read (s_sd, buf, sizeof buf);
```

- ACE enforces typesafety at compile-time via *factories*, *e.g.*,

```
SOCK_Acceptor acceptor (port);

// Error: recv() not a method of SOCK_Acceptor.
acceptor.recv (buf, sizeof buf);
```

Allow Controlled Violations of Typesafety

- *Make it easy to use SOCK_SAP correctly, hard to use it incorrectly, but not impossible to use it in ways the class designers did not anticipate*

- *e.g.*, it may be necessary to retrieve the underlying socket descriptor

```
fd_set rd_sds;

FD_ZERO (&rd_sds);

FD_SET (acceptor.get_handle (), &rd_sds);

select (acceptor.get_handle () + 1, &rd_sds, 0, 0, 0);
```


Simplify for the Common Case

- *Supply default parameters for common method arguments*

```
SOCK_Connector (SOCK_Stream &new_stream,
                const Addr &remote_sap,
                int blocking_semantics = 0,
                const Addr &local_sap = Addr::sap_any,
                int protocol_family = PF_INET,
                int protocol = 0);
```

- The result is extremely concise for the common case:

```
SOCK_Stream stream;
// Compiler supplies default values.
SOCK_Connector con (stream, INET_Addr (port, host));
```


Simplify for the Common Case (cont'd)

- *Define parsimonious interfaces*
  - *e.g.*, use LSOCK to pass socket descriptors:

```
LSOCK_Stream stream;
LSOCK_Acceptor acceptor ("/tmp/foo");

acceptor.accept (stream);
stream.send_handle (stream.get_handle ());
```

  - versus

```
LSOCK::send_handle (const HANDLE sd) const {
  u_char a[2];
  iovec  iov;
  msghdr send_msg;

  a[0] = 0xab, a[1] = 0xcd;
  iov.iov_base = (char *) a; iov.iov_len = sizeof a;
  send_msg.msg_iov = &iov; send_msg.msg_iovlen = 1;
  send_msg.msg_name = (char *) 0;
  send_msg.msg_namelen = 0;
  send_msg.msg_accrights = (char *) &sd;
  send_msg.msg_accrightslen = sizeof sd;
  return sendmsg (this->get_handle (), &send_msg, 0);
```


Simplify for the Common Case (cont'd)

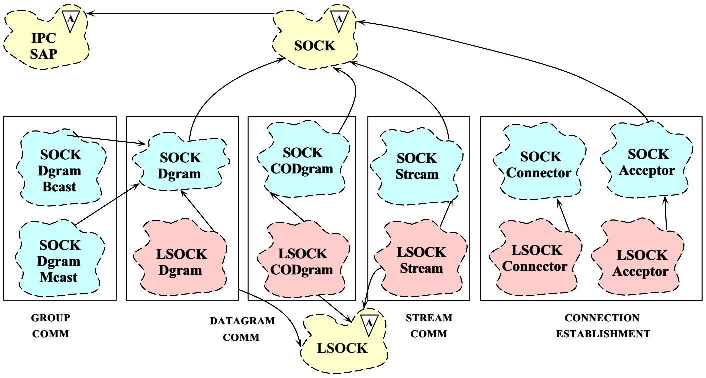
- *Combine multiple operations into a single operation*
  - *e.g.*, creating a conventional passive-mode socket requires multiple calls:

```
int s_sd = socket (PF_INET, SOCK_STREAM, 0);
sockaddr_in addr;
memset (&addr, 0, sizeof addr);
addr.sin_family = AF_INET;
addr.sin_port = htons (port);
addr.sin_addr.s_addr = INADDR_ANY;
bind (s_sd, &addr, addr_len);
listen (s_sd);
// ...
```

  - SOCK_Acceptor combines this into a single operation:

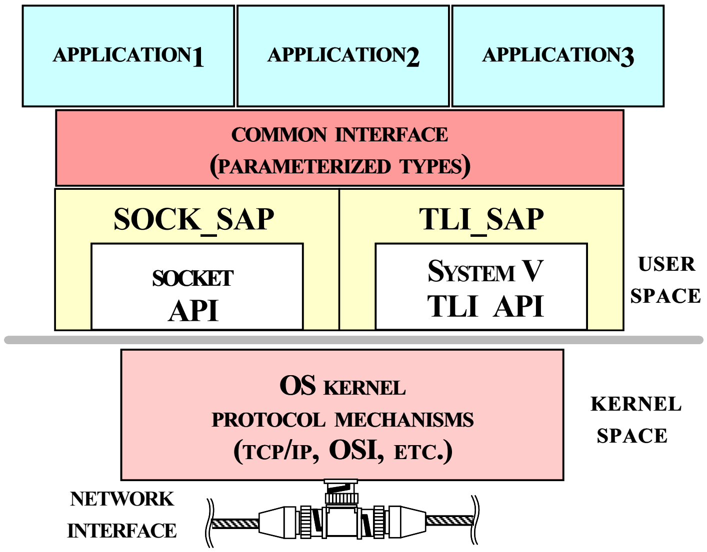
```
SOCK_Acceptor acceptor (INET_Addr (port));
```

Replace One-Dimensional Interfaces with Hierarchical Class Categories



- Classes are organized hierarchically
  - Shared behavior and state is concentrated in base classes
  - Derived classes implement different communication services

Enhance Portability with Parameterized Types



Enhance Portability with Parameterized Types (cont'd)

- Switching wholesale between sockets and TLI simply requires instantiating a different C++ wrapper, *e.g.*,

```
// Conditionally select IPC mechanism.
#if defined (USE_SOCKETS)
typedef SOCK_Stream STREAM;
typedef SOCK_Acceptor ACCEPTOR;

#elif defined (USE_TLI)
typedef TLI_Stream STREAM;
typedef TLI_Acceptor ACCEPTOR;
#endif // USE_SOCKETS.

typedef INET_Addr ADDR;

int main (void)
{
  // ...

  // Invoke the echo_server with appropriate
  // network programming interfaces.
  echo_server<ACCEPTOR, STREAM, ADDR> (port);
}
```


Inline Performance Critical Methods

- Inlining is time and space efficient since key methods are very short:

```
class SOCK_Stream : public SOCK
{
public:
  ssize_t send (const void *buf, size_t n)
  {
    return write (this->get_handle (), buf, n);
  }

  ssize_t recv (void *buf, size_t n)
  {
    return read (this->get_handle (), buf, n);
  }
};
```

- Use `write`/`read` rather than `send`/`recv`

Define Auxiliary Classes to Hide Error-Prone Details

- Standard C socket addressing is awkward and error-prone
  - *e.g.*, easy to neglect to zero-out a `sockaddr_in` or convert port numbers to network byte-order, etc.

- IPC_SAP defines addressing classes to handle these details

```
class INET_Addr : public Addr {
public:
  INET_Addr::INET_Addr (u_short port, long ip_addr = 0) {
    memset (&this->inet_addr_, 0, sizeof this->inet_addr_);
    this->inet_addr_.sin_family = AF_INET;
    this->inet_addr_.sin_port = htons (port);
    memcpy (&this->inet_addr_.sin_addr,
            &ip_addr, sizeof ip_addr);
  }
private:
  sockaddr_in inet_addr_;
};
```


Concluding Remarks

- Defining C++ wrappers for existing OS APIs simplifies the development of correct, portable, and extensible applications
  - C++ inline functions ensure that performance isn't sacrificed

- ACE SOCK_SAP is an example of applying C++ wrappers to standard UNIX and Windows NT network programming interfaces
  - *e.g.*, sockets, TLI, named pipes, STREAM pipes, etc.

- ACE wrappers can be integrated conveniently with CORBA to provide a flexible, high-performance network programming mechanism

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 `wuarchive.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
  - http://www.cs.wustl.edu/~schmidt/