Developing Efficient and Portable Communication Software with ACE and C++

Chris Gill and Douglas C. Schmidt

Distributed Object Computing Group
Computer Science Department, Washington University, St. Louis

cdgill@cs.wustl.edu
http://www.cs.wustl.edu/~schmidt/ACE-examples4.ps.gz

Problem: Software Evolution

- Communication software evolves over time
  - Requirements change
  - Platforms change
  - New design forces emerge
- It is essential to plan for inevitable change

Solution: Plan for Change Using Frameworks and Patterns

- Solution Approach
  - Identify sources of commonality and variability
  - Use patterns to identify reusable design artifacts
  - Use frameworks to “unify” variation in code artifacts

Sources of Variation in Communication Software

- Syntactic Variations
  - Unsupported non-essential APIs
  - Gratuitous differences in API
- Semantic Variations
  - Underlying platform differences
  - Framework must respect these differences
- Complex Variations
  - Unsupported essential portions of API
  - Emulation is necessary
ACE framework: Resolving Syntactic Variations

```c
int ACE_OS::fstat (ACE_HANDLE handle, 
    struct stat *stp) 
{ 
    #if defined (ACE_PSOS_LACKS_PHILE) 
    ACE_UNUSED_ARG (handle); 
    ACE_UNUSED_ARG (stp); 
    ACE_NOTSUP_RETURN (-1); 
    #elif defined (ACE_PSOS) 
    ACE_OSCALL_RETURN 
        (:fstat_f (handle, stp), int, -1); 
    #else 
    ACE_OSCALL_RETURN 
        (:fstat (handle, stp), int, -1); 
    #endif /* ACE_PSOS_LACKS_PHILE */
}
```

- **Examples**
  - **Unsupported**
    * Provide "no-op" definitions
  - **Syntax**
    * Re-map function parameters

ACE framework: Resolving Semantic Variations

```c
int ACE_OS::clock_gettime 
    (clockid_t clockid, struct timespec *ts) 
{ 
    #if defined (ACE_HAS_CLOCK_GETTIME) 
    ACESCALL_RETURN (:clock_gettime 
            (clockid, ts), int, -1); 
    #elif defined (ACE_PSOS) 
    ACE_UNUSED_ARG (clockid); 
    ACE_PSOS_Time_t pt; 
    int result = ACE_PSOS_Time_t::get_system_time (pt); 
    *ts = ACE_static_cast (struct timespec, pt); 
    return result; 
    #else 
    ACE_UNUSED_ARG (clockid); 
    ACE_UNUSED_ARG (ts); 
    ACE_NOTSUP_RETURN (-1); 
    #endif /* ACE_HAS_CLOCK_GETTIME */
}
```

- **Examples**
  - **Underlying differences**
    * Time in clock ticks
    * Ticks-per-second is board-dependent
  - **Framework must respect these differences**
    * Provide a consistent abstraction
    * Intermediate wrappers are useful for small, coherent abstractions

ACE framework: Resolving Complex Variations

```c
void *ACE_TSS_Emulation::tss_open 
    (void *ts_storage[ACE_TSS_KEYS_MAX]) 
{ 
    #if defined (ACE_PSOS) 
    u_long tss_base; 
    tss_base = (u_long) ts_storage; 
    t_setreg (0, PSOS_TASK_REG_TSS, tss_base); 
    void **tss_base_p = ts_storage; 
    for (u_int i = 0; 
        i < ACE_TSS_KEYS_MAX; 
        i++, tss_base_p) 
    *tss_base_p = 0; 
    return (void *) tss_base; 
    #elif defined (...) 
    // ...
}
```

- **Examples**
  - **Unsupported but essential portions of the API (e.g., thread-specific storage)**
    * Provided by POSIX, NT
    * Not provided by VxWorks, pSOS
  - **Emulation in user space is necessary**
    * Create a TSS emulation class
    * Provide platform-specific method implementations

Network Programming Alternatives

<table>
<thead>
<tr>
<th>DOC MIDDLEWARE</th>
<th>USER SPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCKETS AND TLI</td>
<td>TPI</td>
</tr>
<tr>
<td>OPEN/CLOSE/PUTMSG/GETMSG</td>
<td>NPI</td>
</tr>
<tr>
<td>STREAMS</td>
<td>DLPI</td>
</tr>
</tbody>
</table>

- **Communication software can be programmed at several levels of abstraction**
- **Different levels are appropriate for different tasks**
Navigating Through the Design Alternatives

Choosing the appropriate level of abstraction to program involves many factors

- **Performance**
  - Higher levels may be less efficient
- **Functionality**
  - Certain features, e.g., multicast, are not available at all levels
- **Ease of programming**
  - DOC middleware is typically easier to use
- **Portability**
  - The socket API is generally portable...

Common DOC Middleware Features

- DOC middleware “stub/skeleton compiler” support
  - Automatically generate code to perform presentation layer conversions
  - * e.g., network byte-ordering and parameter marshaling
- DOC middleware runtime support
  - Handle network addressing and remote service identification
  - Perform service registration, port monitoring, and service dispatching
  - Enforce authentication and security
  - Manage transport protocol selection and request delivery
  - Provide reliable operation delivery
  - Demultiplexing and dispatching
  - Concurrency and connection management

Overview of DOC Middleware

- Helps simplify many types of applications
- Lets developers work at higher levels of abstraction
- Examples include CORBA, DCOM, Java RMI, DCE, Sun RPC

DOC Middleware Limitations

- Some applications may need to access lower-level IPC mechanisms directly to meet certain requirements
  - * e.g., performance, functionality, portability, etc.
- Compared with direct use of sockets and TLI, DOC middleware may be less efficient due to
  - Presentation conversion processing and excessive data copying
  - Synchronous client-side and server-side stub behavior
  - Stop-and-wait flow control
  - Non-adaptive retransmission timer schemes
  - Non-optimized demultiplexing and concurrency models
**Standard APIs for Network IPC**

- Sockets and TLI allow access to lower-level IPC mechanisms, e.g.:
  - TCP/IP
  - XNS and Novell IPX NetWare protocols
  - UNIX domain sockets
  - OSI protocols

**Socket Taxonomy**

- The Socket API can be classified along three dimensions

---

**Problem with Sockets: Lack of Type-safety**

```c
int buggy_echo_server (u_short port_num)
{
    int n_fd;
    sockaddr_in s_addr;
    int s_fd = socket (PF_UNIX, SOCK_DGRAM, 0);
    s_addr.sin_family = AF_INET;
    s_addr.sin_port = port_num;
    s_addr.sin_addr.s_addr = INADDR_ANY;

    bind (s_fd, (sockaddr *) &s_addr, sizeof s_addr);
    n_fd = accept (s_fd, 0, 0);
    for (;;)
    {
        char buf[BUFSIZ];
        ssize_t n = read (s_fd, buf, sizeof buf);
        if (n <= 0) break;
        write (n_fd, buf, n);
    }
}
```

- I/O handles are not amenable to strong type checking at compile-time
- The adjacent code contains many subtle, common bugs

---

**Problem with Sockets: Steep Learning Curve**

Many socket/TLI API functions have complex semantics, e.g.:

- Multiple protocol families and address families
  - e.g., TCP, UNIX domain, OSI, XNS, etc.
- Infrequently used features, e.g.:
  - Broadcasting/multicasting
  - Passing open file handles
  - Urgent data delivery and reception
  - Asynch I/O, non-blocking I/O, I/O-based and timer-based event multiplexing
Problem with Sockets: Poorly Structured

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

- Note the socket API is linear rather than hierarchical

The ACE C++ IPC Wrapper Solution

- ACE provides C++ "wrappers" that encapsulate IPC programming
  - Interfaces like sockets and TLI
  - ACE provides C++ "wrappers" that encapsulate IPC programming

Intent and Structure of the Wrapper Facade Pattern

- Encapsulates low-level, stand-alone system mechanisms within type-safe, encapsulated, low-level, type-safe, modular, and portable class
- Forces Resolved
- Unifies and abstracts the API

Problem with Sockets: Portability

- Portable between UNIX and Windows
- I/O controls and socket options
- Shutdown semantics
- Handle vs. descriptor types
- Error numbers
- Header files
- Difficult, e.g., have multiple "standards", i.e., sockets and TLI

Thus, it's easy to mix and match system APIs.
The ACE C++ Socket Wrapper Class Structure

- Note how stand-alone functions are replaced by C++ class components.

SOCK_SAP Factory Class Interfaces

class SOCK_Connector
{
public:
  // Traits
  typedef INET.Addr PEER_ADDR;
  typedef SOCK_Stream PEER_STREAM;

  int connect (SOCK_Stream &new_sap, 
               const INET.Addr &raddr, 
               Time_Value /*timeout, 
               const INET.Addr &laddr); 
  // ...
};

class SOCK_Acceptor
{
public:
  // Traits
  typedef INET.Addr PEER_ADDR;
  typedef SOCK_Stream PEER_STREAM;

  int accept (SOCK_Stream &new_sap, 
              const INET.Addr &local_addr);
  int connect (SOCK_Acceptor &new_ac, 
               const INET.Addr &addr, 
               Time_Value /*timeout, 
               const INET.Addr &laddr); 
  // ...
};

SOCK_SAP Stream and Addressing Class Interfaces

class SOCK_Stream : public SOCK
{
public:
  // Traits
  typedef INET.Addr PEER_ADDR;

  ssize_t send (const void *buf, 
               int n);
  ssize_t recv (void *buf, 
               int n);
  ssize_t send_n (const void *buf, 
                  int n);
  ssize_t recv_n (void *buf, 
                  int n);
  int close (void); 
  // ...
};

class INET.Addr : public Addr
{
public:
  INET.Addr (u.short port_number, 
             const char host[]);
  u.short get_port_number (void);
  int32 get_ip_addr (void); 
  // ...
};

OO Design Interlude

Q: Why decouple the SOCK_Acceptor and the SOCK_Connector from SOCK_Stream?

A: For the same reasons that Acceptor and Connector are decoupled from Svc_Handler, e.g.,

- A SOCK_Stream is only responsible for data transfer
  - Regardless of whether the connection is established passively or actively
- This ensures that the SOCK* components are not used incorrectly...
  - e.g., you can't accidentally read or write on SOCK_Connectors or SOCK_Acceptors, etc.
int echo_server (u_short port_num)
{
    // Error handling omitted.
    INET_Addr my_addr (port_num);
    SOCK_Acceptor acceptor (my_addr);
    SOCK_Stream new_stream;

    acceptor.accept (new_stream);

    for (;;) {
        char buf[BUFSIZ];
        // Error caught at compile time!
        ssize_t n = acceptor.recv (buf, sizeof buf);
        new_stream.send_n (buf, n);
    }
}

template <class ACCEPTOR>
int echo_server (u_short port)
{
    // Local address of server (note use of traits).
    ACCEPTOR::PEER_ADDR my_addr (port);
    // Initialize the passive mode server.
    ACCEPTOR acceptor (my_addr);
    // Data transfer object (note use of traits).
    ACCEPTOR::PEER_STREAM stream;
    // Accept a new connection.
    acceptor.accept (stream);

    for (;;) {
        char buf[BUFSIZ];
        ssize_t n = stream.recv (buf, sizeof buf);
        stream.send_n (buf, n);
    }
}

---

- The following slides illustrate differences between using the Socket interface vs. the ACE C++ Socket wrappers
- The example is a simple client/server “network pipe” application that behaves as follows:
  1. Starts an iterative daemon at a well-known server port
  2. Client connects to the server and transmits its standard input to the server
  3. The server prints this data to its standard output
- The server portion of the “network pipe” application may actually run either locally or remotely..
Socket Client

```c
#define PORT_NUM 10000

int main (int argc, char *argv[]) {
  struct sockaddr_in saddr;
  struct hostent *hp;
  char *host = argc > 1 ? argv[1] : "tango.cs.wustl.edu";
  u_short port_num = argc > 2
    htons (argc > 2 ? atoi (argv[2]) : PORT_NUM);
  char buf[BUFSIZ];
  int s_fd;
  int w_bytes;
  int r_bytes;
  int n;

  /* Create a local endpoint of communication */
  s_fd = socket (PF_INET, SOCK_STREAM, 0);

  /* Determine IP address of the server */
  hp = gethostbyname (host);

  /* Set up the address information to contact the server */
  memset ((void *) &saddr, 0, sizeof saddr);
  saddr.sin_family = AF_INET;
  saddr.sin_port = port_num;
  memcpy (&saddr.sin_addr, hp->h_addr, hp->h_length);

  /* Establish connection with remote server */
  connect (s_fd, (struct sockaddr *) &saddr, sizeof saddr);

  /* Send data to server (correctly handles "incomplete writes" due to flow control) */
  while ((r_bytes = read (0, buf, sizeof buf)) > 0)
    for (w_bytes = 0; w_bytes < r_bytes; w_bytes += n)
      n = write (s_fd, buf + w_bytes, r_bytes - w_bytes);

  /* Explicitly close the connection */
  close (s_fd);
  return 0;
}
```

Socket Client (cont’d)

- Requires much less code (about 1/2 to 2/3 less)
- Provides greater clarity and less potential for errors
- Operates at no loss of efficiency

Note that the ACE C++ Socket wrapper example:

Complete example available at URL:

- www.cs.wustl.edu/~schmidt/IPC-SAP-92.ps.gz

Running the Network Pipe Program

Network Pipe with ACE C++ Socket Wrappers
ACEC/++Wrapper Tutorial

C++ Socket Wrapper Client

const short PORT_NUM = 10000;

int main(int argc, char* argv[]) {
  char* host = argv[1];
  short port_num = htons((short)atoi(argv[2]));
  INET_Addr server_addr(port_num, host);
  SOCK_Stream client_stream;
  SOCK_Connect connect;

  Establish the connection:
  connect = SOCK_Connect(client_stream, server_addr);
  if (connect == SOCKET_ERROR) {
    // error
  } else {
    // normal
    printf("Connected to %s on port %d\n", host, port_num);
  }

  // Send data to server (correctly handles incompletewrites)
  for (;;) {
    ssize_t bytes = read(0, buf, sizeof(buf));
    client_stream.send_n(buf, r_bytes);
  }

  // Explicitly close the connection
  client_stream.close();
  return 0;
}

ACEC/++Wrapper Tutorial

Socket Server

/* Close the connection */
{
  close(sockfd);
}

/* Create new endopints of communication */
{
  sockfd = socket(PF_INET, SOCK_STREAM, 0);
  if (sockfd < 0) {
    perror("socket");
    exit(EXIT_FAILURE);
  }

  struct sockaddr_in server_addr;
  server_addr.sin_family = AF_INET;
  server_addr.sin_port = htons(PORT_NUM);
  server_addr.sin_addr.s_addr = htonl(INADDR_ANY);

  if (bind(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
    perror("bind");
    exit(EXIT_FAILURE);
  }

  if (listen(sockfd, 5) < 0) {
    perror("listen");
    exit(EXIT_FAILURE);
  }
}

/* Perform the iterative server activities */
for (;;) {
  char buf[BUFSIZ];
  struct sockaddr_in cli_addr;
  int r_bytes, cli_addr_len = sizeof(cli_addr);
  struct hostent *hp;

  while ((r_bytes = read(sockfd, buf, BUFSIZ)) > 0) {
    write(1, buf, r_bytes);
  }

  close(sockfd);
}

ACEC/++Wrapper Tutorial

Socket Server (cont'd)

ACEC/++Wrapper Tutorial

C++ Socket Wrapper Client (cont'd)

ACEC/++Wrapper Tutorial

ACEC/++Wrapper Tutorial

ACEC/++Wrapper Tutorial

ACEC/++Wrapper Tutorial

ACEC/++Wrapper Tutorial

ACEC/++Wrapper Tutorial

ACEC/++Wrapper Tutorial

C++ Wrapper Socket Server

```c
const u_short PORT_NUM = 10000;

// SOCK_SAP Server.

int main (int argc, char *argv[])
{
    u_short port_num = argc == 1 ? PORT_NUM : atoi (argv[1]);

    // Create a server.
    SOCK_Acceptor acceptor ((INET_Addr) port_num);
    SOCK_Stream new_stream;
    INET_Addr cli_addr;
}
```

// Perform the iterative server activities.

```c
for ( ; ; )
{
    char buf[BUFSIZ];

    // Create a new SOCK_Stream endpoint (note
    // automatic restart if errno == EINTR).
    acceptor.accept (new_stream, &cli_addr);

    printf ("client %s: ", cli_addr.get_host_name ());
    fflush (stdout);

    // Read data from client (terminate on error).
    for ( ; ; )
    {
        ssize_t r_bytes;
        r_bytes = new_stream.recv (buf, sizeof buf);
        if (r_bytes == 0) break;
        write (1, buf, r_bytes);
    }
    // Close new endpoint (listening
    // endpoint stays open).
    new_stream.close ();
}
```
Allow Controlled Violations of Typesafety

Make it easy to use the C++ Socket wrappers 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 handle:

```c
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);
```

Supply Default Parameters

```c
SOCK_Connector (SOCK_Stream &new_stream,
            const Addr &remote_sap,
            ACE_Time_Value *timeout = 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:

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

Define Parsimonious Interfaces

e.g., use LSOCK to pass socket handles:

```c
LSOCK_Stream stream;
LSOCK_Acceptor acceptor ("/tmp/foo");
acceptor.accept (stream);
stream.send_handle (stream.get_handle ());
```

versus

```c
LSOCK::send_handle (const HANDLE sd) const {
  u_char a[2]; iovec iov; msghdr send_msg;
  a[0] = 0; a[1] = 0xcc;
  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);
}
```

Combine Multiple Operations into One Operation

Creating a conventional passive-mode socket requires multiple calls:

```c
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:

```c
SOCK_Acceptor acceptor ((INET_Addr) port);
```
• Shared behavior is isolated in base classes
• Derived classes implement different communication services, communication domains, and connection roles

Enhance Portability with Parameterized Types (cont’d)

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

```c++
#include <stdio.h>

int main (void)
{
   // ... 

   if defined (USE_SOCKETS)
      typedef SOCK_Acceptor PEER_ACCEPTOR;
   #elif defined (USE_TLI)
      typedef TLI_Acceptor PEER_ACCEPTOR;
   #endif // USE_SOCKETS.

   // Invoke the echo_server with appropriate network programming interfaces.
   echo_server<PEER_ACCEPTOR> (port);
}
```

Inline Performance Critical Methods

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

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

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

ACE C++ Socket Wrappers define classes to handle these details

class INET_Addr : public Addr {
public:
    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_; // ...
};

Summary of ACE C++ Socket Wrapper Design Principles

- Domain analysis identifies and groups related classes of existing API behavior
  - Example subdomains include
    * Local context management and options, data transfer, connection/termination handling, etc.
    * Datagrams vs. streams
    * Local vs. remote addressing
    * Active vs. passive connection roles

- These relationships are directly reflected in the ACE C++ Socket wrapper inheritance hierarchy

Summary of ACE C++ Socket Wrapper Design Principles (cont’d)

- The ACE C++ Socket wrappers are designed to maximize reusability and sharing of components
  - Inheritance is used to factor out commonality and decouple variation
    - e.g.,
    * Push common services “upwards” in the inheritance hierarchy
    * Factor out variations in client/server portions of socket API
    * Decouple datagram vs. stream operations, local vs. remote, etc.
  - Inheritance also supports “functional subsetting”
    - e.g., passing open file handles...

- Performance improvements techniques include:
  - Inline functions are used to avoid additional function call penalties
  - Dynamic binding is used sparingly to reduce time/space overhead
    - i.e., it is eliminated for recv/send path

- Note the difference between the composition vs. decomposition/composition aspects in design complexity
  - i.e., ACE C++ Socket wrappers are primarily an exercise in composition since the basic components already exist
  - More complex OO designs involve both aspects...
    - e.g., the ACE Streams, Service Configurator, and Reactor frameworks, etc.
Concluding Remarks

- Defining C++ wrappers for native OS APIs simplifies the development of correct, portable, and extensible applications
  - C++ inline functions ensure that performance isn’t sacrificed
- ACE contains many C++ wrappers that encapsulate UNIX, Win32, and RTOS APIs interfaces
  - e.g., sockets, TLI, named pipes, STREAM pipes, etc.
- ACE can be integrated conveniently with CORBA and DCOM provide a flexible high-performance, real-time development framework