Introduction to ACE

What is ACE?
- OO middleware framework that implements many core design patterns for concurrent communication software
- Targeted for developers of high-performance and real-time communication services and applications
- Simplifies and speeds development via reuse

Why Use Frameworks? Why Use ACE? Why Use Communication Middleware?
- How do we use ACE?
- The ADAPTIVE Communication Environment (ACE)
- Sources for more information

Why Use Frameworks?
- Proven solutions
  - Components
    * Self-contained, ready-to-use ADTs
  - Frameworks
    * Reusable, semi-complete applications
  - Patterns
    * Problem/solution pairs in a context
  - Architecture
    * Families of related patterns and components

Why Use ACE?
- Good example of a framework
- Provides many implementations of patterns
- Open-source
- It’s taking over the world ;-)
Why Use Communication Middleware?

- System call-level programming is wrong abstraction for application developers
  - Too low-level → error codes, endless reinvention
  - Error-prone → HANDLEs lack type-safety, thread cancellation woes
  - Mechanisms do not scale → Win32 TLS
  - Steep learning curve → Win32 Named Pipes
  - Non-portable → socket bugs
  - Inefficient → i.e., tedious for humans

- GUI frameworks are inadequate for communication software
  - Inefficient → excessive use of virtual methods
  - Lack of features → minimal threading and synchronization mechanisms, no network services

ACE Components

- **ACE Overview**
  - A concurrent OO networking framework
  - Available in C++ and Java
  - Ported to VxWorks, POSIX, and Win32

- **Related work**
  - x-Kernel
  - SysV STREAMS

ACE Components

- Interprocess communication
  - sockets, pipes, signal, message passing, shared memory

- Event demultiplexing
  - reactors, proactors

- Concurrency support
  - executions (threads) and synchronization

- Service configuration
  - dynamic linking

- Many other useful services
  - output (logging), sophisticated container classes, high-resolution timing and timers, singleton management

ACE Overview

- A concurrent OO networking framework
  - Available in C++ and Java
  - Ported to VxWorks, POSIX, and Win32

Related work

- x-Kernel
- SysV STREAMS
ACE Statistics

- ACE contains > 175,000 lines of C++
  - Over 20 person-years of effort
- Ported to UNIX, Win32, MVS, and embedded platforms
  - e.g., VxWorks, LynxOS, Chorus
- Large user community
  - www.cs.wustl.edu/~schmidt/ACE-users.html
- Currently used by dozens of companies
  - Bellcore, Boeing, Ericsson, Kodak, Lockheed, Lucent, Motorola, SAIC, Siemens, StorTek, etc.
- Supported commercially
  - www.riverace.com
  - www.ociweb.com

Patterns for Communication Middleware

- Observation
  - Failures rarely result from unknown scientific principles, but from failing to apply proven engineering practices and patterns
- Benefits of Patterns
  - Facilitate design reuse
  - Preserve crucial design information
  - Guide design choices

ACE Task Features

- Queueing
- Event demultiplexing
- Concurrency
- Dynamic linking

ACE Stream Features

- Layered service composition
- Synchronous and asynchronous messaging
- Dynamic configuration

Active Objects with ACE Tasks

ACE Stream Class Category
Network Programming Alternatives

- Communication software can be programmed at several levels of abstraction
- Different levels are appropriate for different tasks

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) {
    // Error checking omitted.
    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);

    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

- **Limitations**
  - Socket API is *linear* rather than *hierarchical*
  - There is no consistency among names...
  - Non-portable

Problem with Sockets: Portability

- **Having multiple “standards,” *i.e.*, sockets and TLI, makes portability difficult, *e.g.*,**
  - May require conditional compilation
  - In addition, related functions are not included in POSIX standards
    - *e.g.*, *select*, *WaitForMultipleObjects*, and *poll*

- **Portability between UNIX and Win32 Sockets is problematic, *e.g.*:**
  - Header files
  - Error numbers
  - Handle vs. descriptor types
  - Shutdown semantics
  - I/O controls and socket options

The ACE C++ IPC Wrapper Solution

- ACE provides C++ “wrapper facades” that encapsulate IPC programming interfaces like sockets and TLI
  - This is an example of the *Wrapper Facade Pattern*
Intent and Structure of the Wrapper Facade Pattern

- **Intent**
  - Encapsulates low-level, stand-alone system mechanisms within type-safe, modular, and portable class interfaces

- **Forces Resolved**
  - Avoid tedious, error-prone, and non-portable system APIs
  - Create cohesive abstractions

---

The ACE C++ Socket Wrapper Class Structure

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

---

SOCK_SAP Factory Class Interfaces

```cpp
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);
    // ...
};
```

---

SOCK_SAP Stream and Addressing Class Interfaces

```cpp
class SOCK_Stream : public SOCK
{
public:
    // Trait.
    typedef INET_Addr PEER_ADDR;
    typedef INET_Addr PEER_STREAM;

    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);
    // ...
};
```
**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.

---

**ACE C++ Wrapper `echo_server`**

```cpp
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);
    }
}
```

---

**A Generic Version of the Echo Server**

```cpp
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);
    }
}
```

---

**Socket vs. ACE C++ Socket Wrapper Example**

- 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...
Network Pipe with Sockets

1: PASSIVE ROLE
socket()
bind() (optional)
connect()
send()/recv()
close()

2: ACTIVE ROLE
socket()
bind()
listen()
accept()
send()/recv()
close()

3: SERVICE PROCESSING

Network Pipe with ACE C++ Socket Wrappers

1: PASSIVE ROLE
SOCK_Acceptor
SOCK_Stream
send()/recv()
close()

2: ACTIVE ROLE
SOCK_Connector
SOCK_Stream
send()/recv()
close()

3: SERVICE PROCESSING

Running the Network Pipe Program

- e.g.,
  % ./server &
  % echo "hello world" | ./client localhost
client localhost.cs.wustl.edu:
  hello world

- Note that the ACE C++ Socket wrapper example:
  - 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

- Complete example available at URL:
  - www.cs.wustl.edu/~schmidt/IPC_SAP-92.ps.gz
C++ Socket Wrapper Client (cont'd)

// Send data to server (correctly handles // "incomplete writes").

for (;;) {
    ssize_t r_bytes = read (0, buf, sizeof buf);
    cli_stream.send_n (buf, r_bytes);
}

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

Socket Client (cont'd)

/* 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 = write (s_fd, buf + w_bytes, r_bytes - w_by

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

Socket Server

#define PORT_NUM 10000
int
main (int argc, char *argv[])
{
    u_short port_num =
        htons (argc > 1 ? atoi (argv[1]) : PORT_NUM);
    struct sockaddr_in saddr;
    int s_fd, n_fd;

    /* Create a local endpoint of communication */
    s_fd = socket (PF_INET, SOCK_STREAM, 0);
    /* Set up the address information to
     * become a server */
    memset ((void *) &saddr, 0, sizeof saddr);
    saddr.sin_family = AF_INET;
    saddr.sin_port = port_num;
    saddr.sin_addr.s_addr = INADDR_ANY;

    /* Associate address with endpoint */
    bind (s_fd, (struct sockaddr *) &saddr,
        sizeof saddr);
    /* Make endpoint listen for service requests */
    listen (s_fd, 5);

    /* Start accepting client requests */
    while (1)
    {
        n_fd = accept (s_fd, (struct sockaddr *) &saddr,
            (socklen_t *) &saddr_len);
        cli_stream.send_n (buf, r_bytes);
    }

    /* Close the server socket */
    close (s_fd);
    return 0;
}

C++ Socket Wrapper Client

const u_short PORT_NUM = 10000;
int main (int argc, char *argv[])
{
    char buf[BUFSIZ];
    char *host = argc > 1 ? argv[1] : "ics.uci.edu";
    u_short port_num =
        htons (argc > 2 ? atoi (argv[2]) : PORT_NUM);

    INET_Addr server_addr (port_num, host);
    SOCK_Stream cli_stream;
    SOCK_Connector connector.

    // Establish the connection with server.
    connector.connect (cli_stream, server_addr);

    // 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 = write (s_fd, buf + w_bytes, r_bytes - w_bytes));

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

Copyright ©1997-2000 Dept. of Computer Science, Washington University
/* Performs the iterative server activities */
for (;;) {
    char buf[BUFSIZ];
    struct sockaddr_in cli_addr;
    struct hostent *hp;
    int r_bytes, cli_addr_len = sizeof cli_addr;
    /* Create a new endpoint of communication */
    while ((n_fd = accept (s_fd, (struct sockaddr *)&cli_addr, &cli_addr_len)) == -1
      && errno == EINTR) continue;
    if (n_fd == -1) continue;
    hp = gethostbyaddr ((char *) &cli_addr.sin_addr, cli_addr_len, AF_INET);
    printf("client %s: ", hp->h_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 ();
}

Copyright ©1997-2000 Dept. of Computer Science, Washington University
Enforce Typesafety at Compile-Time

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

```c
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 type-safety at compile-time via factories, e.g.:

```c
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 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

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_accrightslen = (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);
```

---

Create Hierarchical Class Categories

- Shared behavior is isolated in base classes
- Derived classes implement different communication services, communication domains, and connection roles

---

Enhance Portability with Parameterized Types

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

```c
// Conditionally select IPC mechanism.
#if defined (USE_SOCKETS)
typedef SOCK_Acceptor PEER_ACCEPTOR;
#elif defined (USE_TLI)
typedef TLI_Acceptor PEER_ACCEPTOR;
#endif // USE_SOCKETS.

int main (void)
{
    // ...
    // 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:

```cpp
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:

```cpp
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

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

- 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.

Lessons Learned Building ACE

- Good components, frameworks, and software architectures take time to develop
- Reuse-in-the-large works best when:
  - The marketplace is competitive
  - The domain is complex
  - Building middleware in-house costs too much
  - Corporate culture is supportive
- Produce reusable components by generalizing from working applications
  - i.e., don’t build components in isolation
- The best components (and systems research) come from solving real problems

For More Information

- Overview in http://www.cs.wustl.edu/~schmidt/ACE.html
- Header files in ~levine/ACE_wrappers/ace, tests in tests/
- man pages in /project/adaptive/ACE_wrappers/man, e.g.,
  % man -M /project/adaptive/ACE_wrappers/man \ ACE_SOCK_Connector
  - The man pages are generated from the header files.
  - The html pages are generated from the man pages.
- Mail list: email subscribe ace-users to majordomo@cs.wustl.edu
- Newsgroup: comp.soft-sys.ace