Motivation for Concurrency

- Concurrent programming is increasing relevant to:
  - Leverage hardware/software advances
    * e.g., multi-processors and OS thread support
  - Increase performance
    * e.g., overlap computation and communication
  - Improve response-time
    * e.g., GUIs and network servers
  - Simplify program structure
    * e.g., synchronous vs. asynchronous network IPC

Motivation for Distribution

- Benefits of distributed computing:
  - Collaboration → connectivity and interworking
  - Performance → multi-processing and locality
  - Reliability and availability → replication
  - Scalability and portability → modularity
  - Extensibility → dynamic configuration and re-configuration
  - Cost effectiveness → open systems and resource sharing

Challenges and Solutions

- However, developing efficient, robust, and extensible concurrent networking applications is hard
  - e.g., must address complex topics that are less problematic or not relevant for non-concurrent, stand-alone applications

- OO techniques and OO language features help to enhance software quality factors
  - Key OO techniques include patterns and frameworks
  - Key OO language features include classes, inheritance, dynamic binding, and parameterized types
  - Key software quality factors include modularity, extensibility, portability, reusability, and correctness
Caveats

- **OO is not a panacea**
  - However, when used properly it helps minimize “accidental” complexity and improve software quality factors

- **Advanced OS features provide additional functionality and performance, e.g.,**
  - **Multi-threading**
  - **Multi-processing**
  - **Synchronization**
  - **Shared memory**
  - **Explicit dynamic linking**
  - **Communication protocols and IPC mechanisms**

Tutorial Outline

- **Outline key OO networking and concurrency concepts and OS platform mechanisms**
  - Emphasis is on practical solutions

- **Examine several examples in detail**
  1. **Distributed logger**
  2. **Concurrent WWW client/server**
  3. **Application-level Telecom Gateway**
  4. **OO framework for layered active objects**

- **Discuss general concurrent programming strategies**

Software Development Environment

- The topics discussed here are largely independent of OS, network, and programming language
  - Currently used successfully on UNIX/POSIX, Win32, and RTOS platforms, running on TCP/IP networks using C++

- Examples are illustrated using freely available ADAPTIVE Communication Environment (ACE) OO framework components
  - Although ACE is written in C++, the principles covered in this tutorial apply to other OO languages
    - e.g., Java, Eiffel, Smalltalk, etc.
  - In addition, other networks and backplanes can be used, as well

Definitions

- **Concurrency**
  - “Logically” simultaneous processing
  - Does not imply multiple processing elements

- **Parallelism**
  - “Physically” simultaneous processing
  - Involves multiple processing elements and/or independent device operations

- **Distribution**
  - Partition system/application into multiple components that can reside on different hosts
  - Implies message passing as primary IPC mechanism
Sources of Complexity

- Concurrent network application development exhibits both inherent and accidental complexity

**Inherent complexity** results from fundamental challenges

- **Concurrent programming**
  - Eliminating "race conditions"
  - Deadlock avoidance
  - Fair scheduling
  - Performance optimization and tuning

- **Distributed programming**
  - Addressing the impact of latency
  - Fault tolerance and high availability
  - Load balancing and service partitioning
  - Consistent ordering of distributed events

Sources of Complexity (cont’d)

- **Accidental complexity** results from limitations with tools and techniques used to develop concurrent applications, e.g.,
  - Lack of portable, reentrant, type-safe and extensible system call interfaces and component libraries
  - Inadequate debugging support and lack of concurrent and distributed program analysis tools
  - Widespread use of algorithmic decomposition
    - Fine for explaining concurrent programming concepts and algorithms but inadequate for developing large-scale concurrent network applications
  - Continuous rediscovery and reinvention of core concepts and components
**OO Contributions to Concurrent Applications**

- Concurrent network programming has traditionally been performed using low-level OS mechanisms, e.g.,
  - `fork/exec`
  - `Shared memory, mmap, and SysV semaphores`
  - `Signals`
  - `sockets/select`
  - `POSIX pthreads, Solaris threads, Win32 threads`

- **Patterns** and frameworks elevate development to focus on application concerns, e.g.,
  - **Service functionality and policies**
  - **Service configuration**
  - **Concurrent event demultiplexing and event handler dispatching**
  - **Service concurrency and synchronization**

**Patterns**

- Patterns represent *solutions to problems* that arise when developing software within a particular context
  - *i.e.*, “Patterns = problem/solution pairs in a context”

- Patterns capture the static and dynamic *structure and collaboration* among key participants in software designs
  - They are particularly useful for articulating how and why to resolve *non-functional forces*

- Patterns facilitate reuse of successful software architectures and designs

**Frameworks**

- A framework is:
  - “An integrated collection of components that collaborate to produce a reusable architecture for a family of related applications”

- Frameworks differ from conventional class libraries:
  1. Frameworks are “semi-complete” applications
  2. Frameworks address a particular application domain
  3. Frameworks provide “inversion of control”

- Typically, applications are developed by *inheriting* from and *instantiating* framework components
**Differences Between Class Libraries and Frameworks**

(A) CLASS LIBRARY ARCHITECTURE

1. Application Specific Logic
2. Networking
3. Math
4. ADTs
5. User Interface
6. Database

(B) FRAMEWORK ARCHITECTURE

1. Application Specific Logic
2. Networking
3. Math
4. ADTs
5. User Interface
6. Database

Why We Need Communication Middleware

- System call-level programming is wrong abstraction for application developers, *e.g.*,
  - 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 → Win32 WinSock bugs
  - Inefficient → i.e., tedious for humans

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

**The ADAPTIVE Communication Environment (ACE)**

- A set of C++ wrappers, class categories, and frameworks based on patterns
  - www.cs.wustl.edu/~schmidt/ACE.html

**ACE Statistics**

- Core ACE frameworks and components contain 175,000 lines of C++
  - > 20 person-years of effort

- Ported to UNIX, Win32, MVS, and embedded platforms

- Large user community (ACE-users.html)

- Currently used by dozens of companies
  - *e.g.*, Siemens, Motorola, Ericsson, Kodak, Bellcore, Boeing, SAIC, StorTek, etc.

- Supported commercially by Riverace
  - www.riverace.com/
Class Categories in ACE (cont’d)

- Responsibilities of each class category
  - IPC encapsulates local and/or remote IPC mechanisms
  - Service Initialization encapsulates active/passive connection establishment mechanisms
  - Concurrency encapsulates and extends multi-threading and synchronization mechanisms
  - Reactor performs event demultiplexing and event handler dispatching
  - Service Configurator automates configuration and reconfiguration by encapsulating explicit dynamic linking mechanisms
  - Stream Framework models and implements layers and partitions of hierarchically-integrated communication software
  - Network Services provides distributed naming, logging, locking, and routing services

The ACE ORB (TAO)

- Core TAO ORB contain ~50,000 lines of C++
  - Leverages ACE heavily
- > 10 person-years of effort
- Ported to UNIX, Win32, and embedded platforms
- Currently used by many companies
  - e.g., Siemens, Boeing, SAIC, Raytheon, etc.
- Supported commercially by OCI
  - wwwOCIweb.com/
JA WS Adaptive Web Server

A high-performance, cross-platform Web server built with ACE
- Used commercially by Entera

www.cs.wustl.edu/~jxh/research/

Java ACE

- A version of ACE written in Java
- Currently used for medical imaging prototype
  - www.cs.wustl.edu/~schmidt/JACE.html
  - www.cs.wustl.edu/~schmidt/C++2Java.html
  - www.cs.wustl.edu/~schmidt/MedJava.ps.gz

ACE-related Patterns

Concurrent Overview

- A thread of control is a single sequence of execution steps performed in one or more programs
  - One program → standalone systems
  - More than one program → distributed systems

- Traditional OS processes contain a single thread of control
  - This simplifies programming since a sequence of execution steps is protected from unwanted interference by other execution sequences.
Traditional Approaches to OS Concurrency

1. Device drivers and programs with signal handlers utilize a limited form of concurrency
   - e.g., asynchronous I/O
   - Note that concurrency encompasses more than multi-threading...

2. Many existing programs utilize OS processes to provide “coarse-grained” concurrency
   - e.g.,
     - Client/server database applications
     - Standard network daemons like UNIX `inetd`
   - Multiple OS processes may share memory via memory mapping or shared memory and use semaphores to coordinate execution
   - The OS kernel scheduler dictates process behavior

Evaluating Traditional OS Process-based Concurrency

- **Advantages**
  - *Easy to keep processes from interfering*
    - A process combines *security, protection, and robustness*

- **Disadvantages**
  1. *Complicated to program, e.g.,*
    - Signal handling may be tricky
    - Shared memory may be inconvenient
  2. *Inefficient*
    - The OS kernel is involved in synchronization and process management
    - Difficult to exert fine-grained control over scheduling and priorities

Modern OS Concurrency

- Modern OS platforms typically provide a standard set of APIs that handle
  1. Process/thread creation and destruction
  2. Various types of process/thread synchronization and mutual exclusion
  3. Asynchronous facilities for interrupting long-running processes/threads to report errors and control program behavior

- Once the underlying concepts are mastered, it’s relatively easy to learn different concurrency APIs
  - e.g., traditional UNIX process operations, Solaris threads, POSIX pthreads, WIN32 threads, Java threads, etc.

Lightweight Concurrency

- Modern OSs provide lightweight mechanisms that manage and synchronize multiple threads within a process
  - Some systems also allow threads to synchronize across multiple processes

- **Benefits of threads**
  1. *Relatively simple and efficient to create, control, synchronize, and collaborate*
    - Threads share many process resources by default
  2. *Improve performance by overlapping computation and communication*
    - Threads may also consume less resources than processes
  3. *Improve program structure*
    - e.g., compared with using asynchronous I/O
**Hardware and OS Concurrency Support**

- Most modern OS platforms provide kernel support for multi-threading
  - *e.g.*, SunOS multi-processing (MP) model
    - There are 4 primary abstractions
      1. *Processing elements* (hardware)
      2. *Kernel threads* (kernel)
      3. *Lightweight processes* (user/kernel)
      4. *Application threads* (user)
    - Sun MP thread semantics work for both uni-processors and multi-processors...

**Application Threads**

- Most process resources are equally accessible to all threads in a process, *e.g.*,
  - Virtual memory
  - User permissions and access control privileges
  - Open files
  - Signal handlers

- Each thread also contains unique information, *e.g.*
  - Identifier
  - Register set (*e.g.*, PC and SP)
  - Run-time stack
  - Signal mask
  - Priority
  - Thread-specific data (*e.g.*, `errno`)

- Note, there is generally no MMU protection for separate threads within a single process...
Kernel-level vs. User-level Threads

- Application and system characteristics influence the choice of user-level vs. kernel-level threading
- A high degree of "virtual" application concurrency implies user-level threads (i.e., unbound threads)
  - e.g., desktop windowing system on a uni-processor
- A high degree of "real" application parallelism implies lightweight processes (LWPs) (i.e., bound threads)
  - e.g., video-on-demand server or matrix multiplication on a multi-processor

Synchronization Mechanisms

- Threads share resources in a process address space
- Therefore, they must use synchronization mechanisms to coordinate their access to shared data
- Traditional OS synchronization mechanisms are very low-level, tedious to program, error-prone, and non-portable
- ACE encapsulates these mechanisms with higher-level patterns and classes

Common OS Synchronization Mechanisms

1. Mutual exclusion locks
   - Serialize thread access to a shared resource
2. Counting semaphores
   - Synchronize thread execution
3. Readers/writer locks
   - Serialize thread access to resources whose contents are searched more than changed
4. Condition variables
   - Used to block threads until shared data changes state
5. File locks
   - System-wide readers/write locks accessed by processes using filename

Additional ACE Synchronization Mechanism

1. Events
   - Gates and latches
2. Barriers
   - Allows threads to synchronize their completion
3. Token
   - Provides FIFO scheduling order and simplifies multi-threaded event loop integration
4. Task
   - Provides higher-level "active object" semantics for concurrent applications
5. Thread-specific storage
   - Low-overhead, contention-free storage
Concurrency Mechanisms in ACE

- www.cs.wustl.edu/~schmidt/Concurrency.ps.gz

Graphical Notation

Distributed Logging Service

- www.cs.wustl.edu/~schmidt/reactor-rules.ps.gz

Distributed Logging Service

- Server logging daemon
  - Collects, formats, and outputs logging records forwarded from client logging daemons residing throughout a network or internetwork
  
  The application interface is similar to printf

```c
ACE_ERROR ((LM_ERROR, "(1) fork failed"));  
// generates on server host


ACE_DEBUG ((LM_DEBUG,  
    "(2) sending to server %s", server_host));  
// generates on server host

::(6) sending to server bastille
```
Conventional Logging Server Design

- Typical algorithmic pseudo-code for the server daemon portion of the distributed logging service:

```c
void server_logging_daemon (void)
{
  initialize listener endpoint

  loop forever
  {
    wait for events
    handle data events
    handle connection events
  }
}
```

- The "grand mistake:"
  - Avoid the temptation to "step-wise refine" this algorithmically decomposed pseudo-code directly into the detailed design and implementation of the logging server!

Conventional Logging Server Implementation

- Note the excessive amount of detail required to program at the socket level...

```c
// Main program
static const int PORT = 10000;

typedef u_long COUNTER;
typedef int HANDLE;

// Counts the # of logging records processed
static COUNTER request_count;

// Passive-mode socket handle
static HANDLE listener;

// Highest active handle number, plus 1
static HANDLE maxhp1;

// Set of currently active handles
static fd_set read_handles;

// Scratch copy of read_handles
static fd_set tmp_handles;

int main (int argc, char *argv[])
{
  initialize_listener_endpoint
  {
    (argc > 1 ? atoi (argv[1]) : PORT);
  }

  for (; ; )
  {
    tmp_handles = read_handles; // struct assignment.

    // Wait for client I/O events
    select (maxhp1, &tmp_handles, 0, 0, 0);

    // First receive pending logging records
    handle_data ();

    // Then accept pending connections
    handle_connections ();
  }
}
```
// Initialize the passive-mode socket handle
static void initialize_listener_endpoint (u_short port) {
    struct sockaddr_in saddr;

    // Create a local endpoint of communication
    listener = 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 = htons (port);
    saddr.sin_addr.s_addr = htonl (INADDR_ANY);

    // Associate address with endpoint
    bind (listener, (struct sockaddr *)&saddr, sizeof saddr);

    // Make endpoint listen for connection requests
    listen (listener, 5);
}

// Receive pending logging records
static void handle_data (void) {
    // listener + 1 is the lowest client handle
    for (HANDLE h = listener + 1; h < maxhp1; h++)
        if (FD_ISSET (h, &tmp_handles)) {
            ssize_t n = handle_log_record (h, 1);

            if (n > 0)
                request_count++; // Count the # of logging records
            else if (n == 0) { // Handle connection shutdown.
                FD_CLR (h, &read_handles);
                close (h);
                if (h + 1 == maxhp1) {
                    // Skip past unused handles
                    while (!FD_ISSET (h, &read_handles))
                        continue;
                    maxhp1 = h + 1;
                }
            }
        }
}

// Receive and process logging records
static ssize_t handle_log_record (HANDLE in_h, HANDLE out_h) {
    ssize_t n;
    size_t len;
    Log_Record log_record;

    // The first recv reads the length (stored as a
    // fixed-size integer) of adjacent logging record.
    n = recv (in_h, (char *)&len, sizeof len, 0);
    if (n <= 0) return n;
    len = ntohl (len); // Convert byte-ordering

    // The second recv then reads LEN bytes to obtain the
    // actual record
    for (size_t nread = 0; nread < len; nread += n)
    n = recv (in_h, ((char *)&log_record) + nread,
               len - nread, 0);

    // Decode and print record.
    decode_log_record (&log_record);
    write (out_h, log_record.buf, log_record.size);
    return n;
}

// Check if any connection requests have arrived
static void handle_connections (void) {
    if (FD_ISSET (listener, &tmp_handles)) {
        static struct timeval poll_tv = {0, 0};
        HANDLE h;

        // Handle all pending connection requests
        // (note use of select's "polling" feature)
        do {
            h = accept (listener, 0, 0);
            FD_SET (h, &read_handles);

            // Grow max. socket handle if necessary.
            if (h >= maxhp1)
                maxhp1 = h + 1;
        } while (select (listener + 1, &tmp_handles,
                          0, 0, &poll_tv) == 1);
    }
}
Limitations with Algorithmic Decomposition Techniques

- Algorithmic decomposition tightly couples application-specific functionality and the following configuration-related characteristics:
  - **Structure**
    - The number of services per process
    - Time when services are configured into a process
  - **Communication Mechanisms**
    - The underlying IPC mechanisms that communicate with other participating clients and servers
    - Event demultiplexing and event handler dispatching mechanisms
  - **Concurrency Model**
    - The process and/or thread architecture that executes service(s) at run-time

Overcoming Limitations via OO

- The algorithmic decomposition illustrated above specifies many low-level details
  - Furthermore, the excessive coupling significantly complicates reusability, extensibility, and portability…

- In contrast, OO focuses on application-specific behavior, e.g.,

```c
int Logging_Handler::handle_input (void)
{
  ssize_t n = handle_log_record (peer ().get_handle (), STDOUT);
  if (n > 0)
    ++request_count; // Count the # of logging records
  return n <= 0 ? -1 : 0;
}
```

OO Contributions

- **Patterns** facilitate the large-scale reuse of software architecture
  - Even when reuse of algorithms, detailed designs, and implementations is not feasible

- **Frameworks** achieve large-scale design and code reuse
  - In contrast, traditional techniques focus on the functions and algorithms that solve particular requirements

- Note that patterns and frameworks are not unique to OO!
  - But objects are a useful abstraction mechanism

Patterns in the Distributed Logger

- Note that strategic and tactical are always relative to the context and abstraction level
Pattern Intents

- **Reactor pattern**
  - Decouple event demultiplexing and event handler dispatching from application services performed in response to events

- **Acceptor pattern**
  - Decouple the passive initialization of a service from the tasks performed once the service is initialized

- **Service Configurator pattern**
  - Decouple the behavior of network services from point in time at which services are configured into an application

- **Active Object pattern**
  - Decouple method invocation from method execution and simplifies synchronized access to shared resources by concurrent threads

OO Logging Server

- **OO server logging daemon decomposes into several modular components:**
  1. *Application-specific components*
     - Process logging records received from clients
  2. *Connection-oriented application components*
     - **SvcHandler**
       * Performs I/O-related tasks with clients
     - **Acceptor factory**
       * Passively accepts connection requests
       * Dynamically creates a **SvcHandler** object for each client and "activates" it
  3. *Application-independent ACE framework components*
     - Perform IPC, explicit dynamic linking, event demultiplexing, event handler dispatching, multi-threading, etc.

Demultiplexing and Dispatching Events

- **Problem**
  - The logging server must process several different types of events simultaneously

- **Forces**
  - Multi-threading is not always available
  - Multi-threading is not always efficient
  - Multi-threading can be error-prone
  - Tightly coupling general event processing with server-specific logic is inflexible

- **Solution**
  - Use the **Reactor** pattern to decouple generic event processing from server-specific processing
The Reactor Pattern

- **Intent**
  - “Decouple 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

- www.cs.wustl.edu/~schmidt/Reactor.ps.gz
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 enable flexible policies for creation, connection establishment, and concurrency
  4. How to ensure that a passive-mode handle is not accidentally used to read or write data

Structure of the Acceptor Pattern

Collaboration in the Acceptor Pattern

Using the Acceptor Pattern in the Logging Server

- Acceptor factory creates, connects, and activates a Svc_Handler
**Structure of the Acceptor Pattern in ACE**

- **Concrete Acceptor**
  - Open
- **Concrete Svc Handler**
  - Open
- **Event Handler**
  - Handle input
- **Reactor**
  - Handle input

**Acceptor Class Public Interface**

- A reusable template factory class that accepts connections from clients

```cpp
template <class SVC_HANDLER, // Service aspect
class PEER_ACCEPTOR>, // IPC aspect
class Acceptor : public Event_Handler {
    // Service_Object inherits from Event_Handler
    public:
        virtual int open (const PEER_ACCEPTOR::PEER_ADDR &,
                           Reactor * = Reactor::instance ());
    // Template Method or Strategy for creating.
    virtual int handle_input (HANDLE);
}
```

- Note how service and IPC aspects are strategized...

**Acceptor Class Protected and Private Interfaces**

- Only visible to the class and its subclasses

protected:

```cpp```
// Factory method that creates a service handler.
virtual SVC_HANDLER *make_svc_handler (void);

// Factory method that accepts a new connection.
virtual int accept_svc_handler (SVC_HANDLER *);

// Factory method that activates a service handler.
virtual int activate_svc_handler (SVC_HANDLER *);
```cpp```

private:

```cpp```
// Passive connection mechanism.
PEER_ACCEPTOR peer_acceptor_;
```cpp```

**Acceptor Class Implementation**

```cpp```
// Shorthand names.
define SH SVC_HANDLER
define PA PEER_ACCEPTOR

// Template Method Factory that creates, connects, and activates SVC_HANDLERs.
template <class SH, class PA> int
acceptor<SH, PA>::handle_input (HANDLE)
{
    // Factory Method that makes a service handler.
    SH *svc_handler = make_svc_handler ();
    // Accept the connection.
    accept_svc_handler (svc_handler);
    // Delegate control to the service handler.
    activate_svc_handler (svc_handler);
}
// Factory method for creating a service handler. 
// Can be overridden by subclasses to define new 
// allocation policies (such as Singletons, etc.). 

template <class SH, class PA> SH * 
Accept<SH, PA>::make_svc_handler (HANDLE) 
{ 
    return new SH; // Default behavior. 
} 

// Accept connections from clients (can be overridden). 

Accept<SH, PA>::accept_svc_handler (SH *svc_handler) 
{ 
    peer_acceptor_.accept (svc_handler->peer ()); 
} 

// Activate the service handler (can be overridden). 

Accept<SH, PA>::activate_svc_handler (SH *svc_handler) 
{ 
    if (svc_handler->open () == -1) 
        svc_handler->close (); 
} 

// Initialization. 

Accept<SH, PA>::open (const PA::PEER_ADDR &addr, Reactor *reactor) 
{ 
    // Forward initialization to concrete peer acceptor peer_acceptor_.open (addr); 
    // Register with Reactor. 
    reactor->register_handler 
        (this, Event_Handler::ACCEPT_MASK); 
}
**Svc_Handler implementation**

- By default, a Svc_Handler object is registered with the Reactor
  
  - This makes the service single-threaded and no other synchronization mechanisms are necessary

```c++
#define PS PEER_STREAM // Convenient short-hand.

template <class PS, class SYNCH_STRATEGY>
Svc_Handler<PS, SYNCH_STRATEGY>::Svc_Handler (Reactor *r): reactor_(r) {}

template <class PS, class SYNCH_STRATEGY> int
Svc_Handler<PS, SYNCH_STRATEGY>::open (void *)
{
    // Enable non-blocking I/O.
    peer ().enable (ACE_NONBLOCK);

    // Register handler with the Reactor.
    reactor_->register_handler
        (this, Event_Handler::READ_MASK);
}
```

---

**The Logging_Handler and Logging_Acceptor Classes**

- Templates implement application-specific logging server

```c++
// Performs I/O with client logging daemons.

class Logging_Handler :
    public Svc_Handler<SOCK_Acceptor::PEER_STREAM, NULL_SYNCH> {
public:
    // Recv and process remote logging records.
    virtual int handle_input (HANDLE);
};

// Logging_Handler factory.

class Logging_Acceptor :
    public Acceptor<Logging_Handler, SOCK_Acceptor> {
public:
    // Dynamic linking hooks.
    virtual int init (int argc, char *argv[]);
    virtual int fini (void);
};
```

---

**OO Design Interlude**

- Q: What are the SOCK_* classes and why are they used rather than using sockets directly?

- A: SOCK_* are “wrappers” that encapsulate network programming interfaces like sockets and TLI
  
  - This is an example of the “Wrapper pattern”
The **Wrapper Facade Pattern**

### Intent

Encapsulate low-level functions within type-safe, modular, and portable class interfaces.

This pattern resolves the following forces that arise when using native C-level OS APIs.

1. **Highly non-portable APIs**
   - There is no consistent naming among names, e.g., it gives no hints on how to use it correctly.
   - There is no consistency among names.
   - Highly non-portable.

2. **Error-prone and non-portable programming of low-level IPC mechanisms**
   - How to combine multiple related, but independent, functions into a single cohesive abstraction.
   - How to combine multiple related, but independent, functions into a single cohesive abstraction.

3. **How to avoid tedious, error-prone, and non-portable programming of low-level IPC mechanisms**
   - How to combine multiple related, but independent, functions into a single cohesive abstraction.

The pattern resolves the following forces.

- "Encapsulate low-level functions within type-safe, modular, and portable class interfaces."

---

**Socket Taxonomy**

**Socket Structure**

---

**Structure of the Wrapper Facade**

---
SOCK_SAP Factoy Class

Interfaces

class SOCK_Connector
{
public:
    // Traits
typedef INET_Addr PEER_ADDR;
typedef SOCK_Stream PEER_STREAM;

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

class SOCK_Acceptor : public SOCK
{
public:
    // Traits
typedef INET_Addr PEER_ADDR;
typedef SOCK_Stream PEER_STREAM;

    SOCK_Acceptor (const Addr &local_addr);
    int accept (SOCK_Stream &, Addr *, Time_Value *) const;
    // ...
};

SOCK_SAP Stream and Addressing Class Interfaces

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

    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 never used incorrectly...
    * e.g. you can't accidentally read or write on SOCK_Connectors or SOCK_Acceptors, etc.
OO Design Interlude

- Q: “How can you switch between different IPC mechanisms?”

- A: By parameterizing IPC Mechanisms with C++ Templates!

```c
#if defined (ACE_USE_SOCKETS)
typedef SOCK_Acceptor PEER_ACCEPTOR;
#elif defined (ACE_USE_TLI)
typedef TLI_Acceptor PEER_ACCEPTOR;
#endif /* ACE_USE_SOCKETS */

class Logging_Handler : public Svc_Handler<PEER_ACCEPTOR::PEER_STREAM,
   NULL_SYNCH>
   { /* ... */ };

class Logging_Acceptor : public Acceptor <Logging_Handler, PEER_ACCEPTOR>
   { /* ... */ };
```

Logging_Handler Implementation

- Implementation of the application-specific logging method

```c
// Callback routine that receives logging records.  
// This is the main code supplied by a developer!

int Logging_Handler::handle_input (HANDLE)
{  
   // Call existing function to recv  
   // logging record and print to stdout.  
   handle_log_record (peer ().get_handle (), STDOUT);
}
```

// Automatically called when a Logging_Acceptor object  
// is dynamically linked.

Logging_Acceptor::init (int argc, char *argv[])  
{  
   Get_Opt get_opt (argc, argv, "p", 0);  
   INET_Addr addr;
   for (int c; (c = get_opt ()) != -1; )  
      switch (c)  
      {  
         case 'p':  
            addr.set (atoi (getopt_optarg));  
            break;
         default:
            break;
      }  

      // Initialize endpoint and register with the Reactor  
      open (addr, Reactor::instance ());  
   }

// Automatically called when object is dynamically unlinked.

Logging_Acceptor::fini (void)
{  
   handle_close ();  
}
Putting the Pieces Together at Run-time

- **Problem**
  - Prematurely committing ourselves to a particular logging server configuration is inflexible and inefficient

- **Forces**
  - It is useful to build systems by “scripting” components
  - Certain design decisions can’t be made efficiently until run-time
  - It is a bad idea to force users to “pay” for components they do not use

- **Solution**
  - Use the Service Configurator pattern to assemble the desired logging server components dynamically

The Service Configurator Pattern

- **Intent**
  - “Decouple the behavior of services from the point in time at which these services are configured into an application”

- This pattern resolves the following forces for highly flexible communication software:
  - How to defer the selection of a particular type, or a particular implementation, of a service until very late in the design cycle
    - *i.e.*, at installation-time or run-time
  - How to build complete applications by scripting multiple independently developed services
  - How to reconfigure and control the behavior of the service at run-time

Structure of the Service Configurator Pattern

Collaboration in the Service Configurator Pattern
Using the Service Configurator Pattern for the Logging Server

**Dynamic Linking a Service**

- Application-specific factory function used to dynamically create a service

```c
extern "C" Service_Object *make_Logger (void);
Service_Object *make_Logger (void)
{
    return new Logging_Acceptor;
    // Framework automatically deletes memory.
}
```

- The `make_Logger` function provides a hook between an application-specific service and the application-independent ACE mechanisms
  - ACE handles all memory allocation and deallocation

**Service Configuration**

- The logging service is configured via scripting in a `svc.conf` file:

  ```bash
  % cat ./svc.conf
  # Dynamically configure the logging service
dynamic Logger Service_Object *
  logger:make_logger() "-p 2001"
  # Note, .dll or .so suffix added to "logger" automatically
  ```

- Generic event-loop to dynamically configure service daemons

```c
int main (int argc, char *argv[])
{
    // Initialize the daemon and configure services
    Service_Config::open (argc, argv);

    // Run forever, performing configured services
    Reactor::run_event_loop ();
    /* NOTREACHED */
}
```

```
* AWAITING EVENTS
  * PERFORM CALLBACK
  * CALL HANDLER/
  * Event_Handler::handle_input()

  * SHUTDOWN/
  * Service_Config::close()

  * START EVENT LOOP/
  * Reactor::run_event_loop()

  * IDLE

  * CONFIGURE/
  * Service_Config::process_directives()

  * INITIALIZED

```

- Note the separation of concerns between objects...
**Collaboration of Patterns in the Server Logging Daemon**

- **A**: Logging Acceptor
- **C**: Logging Handler
- **Service**: Service Config
- **Reactor**: Service Repository

**REGISTER SERVICE**

**START EVENT LOOP**

**CONNECTION EVENT**

**DATA EVENT**

**REGISTER HANDLER** FOR CLIENT I/O

**PROCESS LOGGING RECORD**

**STORE IN REPOSITORY**

**CONFIGURE**

**FOREACH SVC ENTRY DO**

**EXTRACT HANDLE**

**FOR EACH EVENT DO**

**STORE IN REPOSITORY**

**FOR EACH HANDLE**

**INITIALIZE SERVICE**

**DAEMON SHUTDOWN**

**FINI**

**LINK SERVICE**

**REMOVE HANDLER**

**C = new Logging_Handler**

**accept(C);**

**C->open(A)**

**register_handler(A)**

**run_event_loop()**

**get_handle()**

**insert()**

**Service_Config()**

**get_handle()**

**init(argc, argv)**

**handle_close()**

**remove_handler(A)**

**unlink_service()**

**allocate and activate object**

**client shutdown**

**daemon shutdown**

**unlink service**

**allocate and activate object**

**client shutdown**

**daemon shutdown**

**unlink service**

**Advantages of OO Logging Server**

- The OO architecture illustrated thus far decouples application-specific service functionality from:
  1. Time when a service is configured into a process
  2. The number of services per-process
  3. The type of IPC mechanism used
  4. The type of event demultiplexing mechanism used

- We can use the techniques discussed thus far to extend applications without:
  1. **Modifying, recompiling, and relinking existing code**
  2. **Terminating and restarting executing daemons**

- The remainder of the slides examine a set of techniques for decoupling functionality from concurrency mechanisms, as well

---

**Concurrent OO Logging Server**

- The structure of the server logging daemon can benefit from concurrent execution on a multi-processor platform

- This section examines ACE C++ classes and patterns that extend the logging server to incorporate concurrency
  - Note how most extensions require minimal changes to the existing OO architecture...

- This example also illustrates additional ACE components involving synchronization and multi-threading

- Thread-per-connection implementation
Pseudo-code for Concurrent Server

- Pseudo-code for multi-threaded Logging_Handler factory server logging daemon

```c
void handler_factory (void)
{
    initialize listener endpoint
    foreach (pending connection event) {
        accept connection
        spawn a thread to handle connection and run logger_handler() entry point
    }
}
```

- Pseudo-code for server logging daemon active object

```c
void logging_handler (void)
{
    foreach (incoming logging records from client)
        call handle_log_record()
    exit thread
}
```

Application-specific Logging Code

- The OO implementation localizes the application-specific part of the logging service in a single point, while leveraging off reusable ACE components

```c
int Thr_Logging_Handler::svc (void)
{
    while (handle_input () != -1)
    // Call existing function to recv logging record and print to stdout.
        continue;
    return 0;
}
```

Class Diagram for Concurrent OO Logging Server

Thr_Logging_Acceptor and Thr_Logging_Handler

- Template classes that create, connect, and activate a new thread to handle each client

```c
class Thr_Logging_Handler : public Logging_Handler
    // Inherits <handle_input>
{
public:
    // Override definition in the Svc_Handler
    // class (spawns a new thread!).
    virtual int open (void *)
    {
        // Process remote logging records.
        virtual int svc (void);
    }
};

class Thr_Logging_Acceptor :
    public Acceptor<Thr_Logging_Handler, SOCK_Acceptor>
{
    // Same as Logging_Acceptor...
};
```
ACE Tasks

- An ACE Task binds a separate thread of control together with an object's data and methods
  - Multiple active objects may execute in parallel in separate lightweight or heavyweight processes

- Task objects communicate by passing typed messages to other Tasks
  - Each Task maintains a queue of pending messages that it processes in priority order

- ACE Task are a low-level mechanism to support "active objects"

Task Class Public Interface

- C++ interface for message processing
  * Tasks can register with a Reactor
  * They can be dynamically linked
  * They can queue data
  * They can run as "active objects"

- e.g.,

```cpp
template <class SYNCH_STRATEGY>
class Task : public Service_Object {
public:
  // Initialization/termination hooks.
  virtual int open (void *args = 0) = 0;
  virtual int close (u_long flags = 0) = 0;

  // Hook to pass msg for immediate processing.
  virtual int put (Message_Block *,
                  Time_Value * = 0) = 0;

  // Hook run by daemon thread(s) for deferred processing.
  virtual int svc (void) = 0;

  // Turn task into an active object.
  int activate (long flags, int n_threads = 1);
```

 Task Inheritance Hierarchy

- Supports dynamically configured services
Task Class Protected Interface

- The following methods are mostly used within put and svc

```c
// Accessors to internal queue.
Message_Queue<SYNCH_STRATEGY> *msg_queue (void);
void msg_queue (Message_Queue<SYNCH_STRATEGY> *);

// Accessors to thread manager.
Thread_Manager *thr_mgr (void);
void thr_mgr (Thread_Manager *);

// Insert message into the message list.
int putq (Message_Block *, Time_Value *tv = 0);

// Extract the first message from the list (blocking).
int getq (Message_Block *mb, Time_Value *tv = 0);

// Hook into the underlying thread library.
static void svc_run (Task<SYNCH_STRATEGY> *);
```

OO Design Interlude

- Q: What is the svc_run() function and why is it a static method?

  - A: OS thread spawn APIs require a C-style function as the entry point into a thread

- The Stream class category encapsulates the svc_run function within the Task::activate method:

```c
template <class SYNCH_STRATEGY> int
Task<SYNCH_STRATEGY>::activate (long flags, int n_threads)
{
  if (thr_mgr () == NULL)
    thr_mgr (Thread_Manager::instance ());
  thr_mgr ()->spawn_n
    (n_threads, &Task<SYNCH_STRATEGY>::svc_run,
     (void *) this, flags);
}
```

OO Design Interlude (cont’d)

- Task::svc_run is static method used as the entry point to execute an instance of a service concurrently in its own thread

```c
template <class SYNCH_STRATEGY> void *
Task<SYNCH_STRATEGY>::svc_run (Task<SYNCH_STRATEGY> *t)
{
  // Thread added to thr_mgr()
  // automatically on entry...

  // Run service handler and record return value.
  void *status = (void *) t->svc ();
  tc.status (status);
  t->close (u_long (status));

  // Status becomes ‘return’ value of thread...
  return status;

  // Thread removed from thr_mgr()
  // automatically on return...
}
```

OO Design Interlude

- Q: “How can groups of collaborating threads be managed atomically?”

  - A: Develop a “thread manager” class

    - **Thread_Manager** is a collection class
      * It provides mechanisms for suspending and resuming groups of threads atomically
      * It implements barrier synchronization on thread exits

    - **Thread_Manager** also shields applications from incompatibilities between different OS thread libraries
      * It is integrated into ACE via the Task::activate method
The Active Object Pattern

- **Intent**
  - "Decouple 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 read and write operations on one endpoint that do not detract from the quality of service of other endpoints
  - How to serialize concurrent access to shared object state
  - How to simplify composition of independent services

Structure of the Active Object Pattern

Collaboration in the Active Object Pattern

ACE Support for Active Objects

- Can implement complete Active Object pattern or lighterweight subsets
Collaboration in ACE Active Objects

CONSUMER PHASE

<table>
<thead>
<tr>
<th>Task</th>
<th>Task</th>
<th>Message Queue</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENQUEUE</td>
<td>dequeue_head(msg)</td>
<td>dequeue_head(msg)</td>
<td>dequeue_head(msg)</td>
</tr>
<tr>
<td>RUN svc() ASYNCHRONOUSLY</td>
<td>SVC()()</td>
<td>SVC()()</td>
<td>SVC()()</td>
</tr>
<tr>
<td>PASS MSG</td>
<td>put(msg)</td>
<td>put(msg)</td>
<td>put(msg)</td>
</tr>
<tr>
<td>QUEUEING PHASE</td>
<td>enqueue_tail(msg)</td>
<td>enqueue_tail(msg)</td>
<td>enqueue_tail(msg)</td>
</tr>
<tr>
<td>PRODUCER TASK</td>
<td>work()</td>
<td>work()</td>
<td>work()</td>
</tr>
<tr>
<td>PASS MSG</td>
<td>put(msg)</td>
<td>put(msg)</td>
<td>put(msg)</td>
</tr>
<tr>
<td>PRODUCER TASK</td>
<td>put(msg)</td>
<td>put(msg)</td>
<td>put(msg)</td>
</tr>
</tbody>
</table>

Dynamically Reconfiguring the Logging Server

- The concurrent logger is reconfigured by changing the svc.conf file and sending SIGHUP signal to the server:

```
// Dynamically linked factory function that allocates a new threaded Logging Acceptor.
extern "C" Service_Object *make_LOGGER (void);
```

```
Service_Object *make_LOGGER (void) {
    return new Thr_Logging_Acceptor;
}
```

```
% cat ./svc.conf
# Dynamically configure the logging service
# dynamic Logger Service_Object *
# ./svcs/logger.dll:make_LOGGER () "-p 2010"
remove Logger
dynamic Logger Service_Object *
thr_logger:make_LOGGER () "-p 2010"
# .dll or .so suffix added to "thr_logger" automatically
```

Caveats

- The concurrent server logging daemon has several problems
  1. Output in the handle_log_record function is not serialized
  2. The auto-increment of global variable request_count is also not serialized

- Lack of serialization leads to errors on many shared memory multi-processor platforms...
  - Note that this problem is indicative of a large class of errors in concurrent programs...

- The following slides compare and contrast a series of techniques that address this problem

Explicit Synchronization Mechanisms

- One approach for serialization uses OS mutual exclusion mechanisms explicitly, e.g.,

```
// at file scope
mutex_t lock; // SunOS 5.x synchronization mechanism
```

```
// ...
handle_log_record (HANDLE in_h, HANDLE out_h) {
    // in method scope ...
    mutex_lock (&lock);
    write (out_h, log_record.buf, log_record.size);
    mutex_unlock (&lock);
    // ...
}
```

- However, adding these mutex calls explicitly is causes problems...
Problems Galore!

- Problems with explicit mutex_* calls:
  - Inelegant
    * "Impedance mismatch" with C/C++
  - Obtrusive
    * Must find and lock all uses of write
  - Error-prone
    * C++ exception handling and multiple method exit points cause subtle problems
    * Global mutexes may not be initialized correctly...
  - Non-portable
    * Hard-coded to Solaris 2.x
  - Inefficient
    * e.g., expensive for certain platforms/designs

C++ Wrappers for Synchronization

- To address portability problems, define a C++ wrapper:

```cpp
class Thread.Mutex
{
public:
  Thread.Mutex (void) {
    mutex_init (&lock_, USYNCH_THREAD, 0);
  }
  ~Thread.Mutex (void) {
    mutex_destroy (&lock_);
  }
  int acquire (void) {
    return mutex_lock (&lock_);
  }
  int tryacquire (void) {
    return mutex_trylock (&lock_);
  }
  int release (void) {
    return mutex_unlock (&lock_);
  }
private:
  mutex_t lock_; // SunOS 5.x serialization mechanism.
  void operator= (const Thread.Mutex &);
  Thread.Mutex (const Thread.Mutex &);
};
```

- Note, this mutual exclusion class interface is portable to other OS platforms

Porting Mutex to Windows NT

- WIN32 version of Mutex

```cpp
class Thread.Mutex
{
public:
  Thread.Mutex (void) {
    lock_ = CreateMutex (0, FALSE, 0);
  }
  ~Thread.Mutex (void) {
    CloseHandle (lock_);
  }
  int acquire (void) {
    return WaitForSingleObject (lock_, INFINITE);
  }
  int tryacquire (void) {
    return WaitForSingleObject (lock_, 0);
  }
  int release (void) {
    return ReleaseMutex (lock_);
  }
private:
  HANDLE lock_; // Win32 locking mechanism.
};
```

Using the C++ Mutex Wrapper

- Using C++ wrappers improves *portability* and *elegance*

```cpp
// at file scope
Thread.Mutex lock; // Implicitly "unlocked".

// ...
handle_log_record (HANDLE in_h, HANDLE out_h)
{
  // in method scope ...
  lock.acquire ();
  write (out_h, log_record.buf, log_record.size);
  lock.release ();
  // ...
}
```

- However, this doesn't really solve the tedium or error-proneness problems
Automated Mutex Acquisition and Release

To ensure mutexes are locked and unlocked, we'll define a template class that acquires and releases a mutex automatically.

```cpp
template <class LOCK>
class Guard
{
public:
  Guard (LOCK &m): lock (m) { lock_.acquire (); }
  ~Guard (void) { lock_.release (); }
private:
  LOCK &lock_;
}
```

Guard uses the C++ idiom whereby a constructor acquires a resource and the destructor releases the resource.

Q: Why is Guard parameterized by the type of LOCK?

A: since there are many different flavors of locking that benefit from the Guard functionality, e.g.,

- Non-recursive vs recursive mutexes
- Intra-process vs inter-process mutexes
- Readers/Writer mutexes
- Solaris and System V semaphores
- File locks
- Null mutex

In ACE, all synchronization wrappers use the Adapter pattern to provide identical interfaces whenever possible to facilitate parameterization.

The Adapter Pattern

- Intent
  - "Convert the interface of a class into another interface client expects"
    
    - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces

- This pattern resolves the following force that arises when using conventional OS interfaces
  1. How to provide an interface that expresses the similarities of seemingly different OS mechanisms (such as locking or IPC)

Structure of the Adapter Pattern
Using the Adapter Pattern for Locking

A thread-safe handle_log_record()

Function

template <class LOCK = Thread_Mutex> ssize_t handle_log_record (HANDLE in_h, HANDLE out_h)
{
    // new code (beware of static initialization...)
    static LOCK lock;
    ssize_t n;
    size_t len;
    Log_Record log_record;

    n = recv (h, (char *) &len, sizeof len, 0);
    if (n != sizeof len) return -1;
    len = ntohl (len); // Convert byte-ordering

    for (size_t nread = 0; nread < len; nread += n)
    {
        n = recv (in_h, ((char *) &log_record) + nread, len - nread, 0);
        // Perform presentation layer conversions.
        decode (&log_record);
        // Automatically acquire mutex lock.
        Guard<LOCK> monitor (lock);
        write (out_h, log_record.buf, log_record.size);
        // Automatically release mutex lock.
    }
}

Remaining Caveats

- There is a race condition when incrementing the request_count variable

```c
int Logging_Handler::handle_input (void)
{
    ssize_t n = handle_log_record (peer ()->get_handle (), STDOUT);
    if (n > 0)
        ++request_count; // Count the # of logging records
    return n <= 0 ? -1 : 0;
}
```

- Solving this problem using the Mutex or Guard classes is still tedious, low-level, and error-prone

- A more elegant solution incorporates parameterized types, overloading, and the Decorator pattern

Transparently Parameterizing Synchronization Using C++

- The following C++ template class uses the “Decorator” pattern to define a set of atomic operations on a type parameter

```cpp
template <class LOCK = Thread_Mutex, class TYPE = u_long>
class Atomic.Op {
public:
    Atomic.Op (TYPE c = 0) { count_ = c; }
    TYPE operator++ (void) {
        Guard<LOCK> m (lock_); return ++count_; }
    void operator= (const Atomic.Op &ao) {
        if (this != &ao) { 
            Guard<LOCK> m (lock_); count_ = ao.count_; 
        }
    }
    operator TYPE () {
        Guard<LOCK> m (lock_);
        return count_; }
    // Other arithmetic operations omitted...
private:
    LOCK lock_;
    TYPE count_; }
```
Final Version of Concurrent Logging Server

- Using the AtomicOp class, only one change is made

```c
// At file scope.
typedef AtomicOp<> COUNTER; // Note default parameters...
COUNTER request_count;
```

- request_count is now serialized automatically

```c
for (; ; ++request_count) // AtomicOp::operator++
    handle_log_record (get_handle (), STDOUT);
```

- The original non-threaded version may be supported efficiently as follows:

```c
typedef AtomicOp<NullMutex> COUNTER; //...
for (; ; ++request_count)
    handle_log_record<NullMutex>
    (get_handle (), STDOUT);
```

Synchronization-aware Logging Classes

- A more sophisticated approach would add several new parameters to the Logging_Handler class

```c
template <class PEER_STREAM,
    class SYNCH_STRATEGY, class COUNTER>
class Logging_Handler
    : public Svc_Handler<PEER_STREAM, SYNCH_STRATEGY>
{
    public:
        Logging_Handler (void);
        // Process remote logging records.
        virtual int svc (void);
    protected:
        // Receive the logging record from a client.
        ssize_t handle_log_record (HANDLE out_h);
        // Lock used to serialize access to std output.
        static SYNCH_STRATEGY::MUTEX lock_; // Count the number of logging records that arrive.
        static COUNTER request_count_;
};
```

Thread-safe handle_log_record Method

```c
template <class PS, class LOCK, class COUNTER> ssize_t
Logging_Handler<PS, LOCK, COUNTER>::handle_log_record
    (HANDLE out_h)
{
    ssize_t n;
    size_t len;
    Log_Record log_record;

    ++request_count_; // Calls COUNTER::operator++().

    n = peer ().recv (&len, sizeof len);
    if (n != sizeof(len)) return -1;
    len = ntohl (len); // Convert byte-ordering
    peer ().recv_n (&log_record. len);
    // Perform presentation layer conversions
    log_record.decode ();
    // Automatically acquire mutex lock.
    Guard<LOCK> monitor (lock_);
    write (out_h, log_record.buf, log_record.size);
    // Automatically release mutex lock.
}
```

Using the Thread-safe handle_log_record() Method

- In order to use the thread-safe version, all we need to do is instantiate with AtomicOp

```c
typedef Logging_Handler<TLI_Stream,
    NULL_SYNCH, AtomicOp<> >
LOGGING_HANDLER;
```

- To obtain single-threaded behavior requires a simple change:

```c
typedef Logging_Handler<TLI_Stream,
    NULL_SYNCH, AtomicOp<NullMutex, u_long> >
LOGGING_HANDLER;
```
**Concurrent WWW Client/Server Example**

- The following example illustrates a concurrent OO architecture for a high-performance Web client/server.

- **Key system requirements are:**
  
  1. Robust implementation of HTTP protocol
     - *i.e.*, resilient to incorrect or malicious Web clients/servers
  
  2. Extensible for use with other protocols
     - *e.g.*, DICOM, HTTP 1.1, SFP
  
  3. Leverage multi-processor hardware and OS software
     - *e.g.*, support various concurrency models

---

**Pseudo-code for Concurrent WWW Server**

- **Pseudo-code for master server**

  ```c
  void master_server (void)
  {
    initialize work queue and
    listener endpoint at port 80
    spawn pool of worker threads
    foreach (pending work request from clients) {
      receive and queue request on work queue
    }
    exit process
  }
  ```

- **Pseudo-code for thread pool workers**

  ```c
  void worker (void)
  {
    foreach (work request on queue)
      dequeue and process request
    exit thread
  }
  ```

---

**General Web Client/Server Interactions**

**OO Design Interlude**

- **Q:** Why use a work queue to store messages, rather than directly reading from I/O handles?

  **A:**
  
  - Separation of concerns
  
  - Promotes more efficient use of multiple CPUs via load balancing
  
  - Enables transparent interpositioning and prioritization
  
  - Makes it easier to shut down the system correctly and portably

- **Drawbacks**

  - Using a message queue may lead to greater context switching and synchronization overhead...
  
  - Single point for bottlenecks
Thread Entry Point

- Each thread executes a function that serves as the “entry point” into a separate thread of control
  - Note algorithmic design...

```c
typedef u_long COUNTER;  // Track the number of requests
COUNTER request_count;   // At file scope.

// Entry point into the WWW HTTP 1.0 protocol.
void *worker (Message_Queue *msg_queue) {
    Message_Block *mb; // Message buffer.
    while (msg_queue->dequeue_head (mb)) > 0) {   // Keep track of number of requests.
        request_count++;
        // Print diagnostic
cout << "got new request " << OS::thr_self () << endl;
        // Identify and perform WWW Server
        // request processing here...
    }
    return 0;
```

Master Server Driver Function

- The master driver function in the WWW Server might be structured as follows:

```c
// Thread function prototype.
typedef void *(THR_FUNC)(void *);
int main (int argc, char *argv[]) {
    parse_args (argc, argv);
    Message_Queue msg_queue; // Queue client requests.

    // Spawn off NUM_THREADS to run in parallel.
    for (int i = 0; i < NUM_THREADS; i++)  
    thr_create (0, O, THR_FUNC (&worker),  
                 (void *) &msg_queue, THR_NEW_LWP, 0);

    // Initialize network device and
    // recv HTTP work requests.
    thr_create (0, 0, THR_FUNC (&recv_requests),  
               (void *) &msg_queue, THR_NEW_LWP, 0);

    // Wait for all threads to exit (BEWARE)!
    while (thr.join (O, &t_id, (void **) O) == 0) continue; // ...
}
```

Pseudo-code for recv_requests()

- e.g.,

```c
void recv_requests (Message_Queue *msg_queue) {
    initialize socket listener endpoint at port 80

    foreach (incoming request) {
        use select to wait for new connections or data  
        if (connection) establish connections using accept
        else if (data) {  
            use sockets calls to read HTTP requests into msg
            msg_queue.enqueue_tail (msg);
        }
    }
```

- The “grand mistake:”
  - Avoid the temptation to ‘step-wise refine’ this algorithmically decomposed pseudo-code directly into the detailed design and implementation of the WWW Server!

Limitations with the WWW Server

- The algorithmic decomposition tightly couples application-specific functionality with various configuration-related characteristics, e.g.,
  - The HTTP 1.0 protocol
  - The number of services per process
  - The time when services are configured into a process

- The solution is not portable since it hard-codes
  - SunOS 5.x threading
  - sockets and select

- There are race conditions in the code
Overcoming Limitations via OO

- The algorithmic decomposition illustrated above specifies too many low-level details
  - Furthermore, the excessive coupling complicates reusability, extensibility, and portability...

- In contrast, OO focuses on decoupling application-specific behavior from reusable application-independent mechanisms

- The OO approach described below uses reusable framework components and commonly recurring patterns

Eliminating Race Conditions

- **Problem**
  - A naive implementation of `MessageQueue` will lead to race conditions
    * e.g., when messages in different threads are enqueued and dequeued concurrently

- **Forces**
  - Producer/consumer concurrency is common, but requires careful attention to avoid overhead, deadlock, and proper concurrency control

- **Solution**
  - Utilize a “condition variables”

Condition Variable Overview

- Condition variables (CVs) are used to “sleep/wait” until a particular condition involving shared data is signaled
  - CVs may be arbitrarily complex C++ expressions
  - Sleeping is often more efficient than busy waiting...

- This allows more complex scheduling decisions, compared with a mutex
  - i.e., a mutex makes other threads wait, whereas a condition object allows a thread to make itself wait for a particular condition involving shared data

Condition Variable Usage

- A particular idiom is associated with acquiring resources via condition variables

```c
// Global variables
global static Thread_Mutex lock; // Initially unlocked. // Initially unlocked.
global static Condition_Thread_Mutex cond (lock);

// Acquire resources
void acquire_resources (void) {
    // Automatically acquire the lock.
    Guard <Thread_Mutex> monitor (lock);

    // Check condition (note the use of while)
    while (condition expression is not true)
        // Sleep if not expression is not true.
        cond.wait ();

    // Atomically modify shared information here...

    // Monitor destructor automatically releases lock.
}
```
**Condition Variable Usage (cont’d)**

- Another idiom is associated with releasing resources via condition variables

```cpp
void release_resources (void) {
    // Automatically acquire the lock.
    Guard<Thread_Mutex> monitor (lock);

    // Atomically modify shared information here...
    cond.signal (); // Could also use cond.broadcast()
    // monitor destructor automatically releases lock.
}
```

- Note how the use of the Guard idiom simplifies the solution
  - e.g., now we can’t forget to release the lock!

---

**Overview of Message Queue and Message Block Classes**

- A `Message_Queue` is composed of one or more `Message Blocks`
  - Similar to BSD `mbufs` or SVR4 STREAMS `mblks`
  - Goal is to enable efficient manipulation of arbitrarily-large message payloads without incurring unnecessary memory copying overhead

- `Message Blocks` are linked together by `prev_` and `next_` pointers

- A `Message Block` may also be linked to a chain of other `Message Blocks`

---

**Message Queue and Message Block Object Diagram**

- A `Message Queue` consists of one or more `Message Blocks`
  - Each `Message Block` contains a `Payload`
  - `Message Blocks` are linked together by `prev_` and `next_` pointers
  - `Message Queue` head and tail pointers
  - `Message Block` next and prev pointers
  - `Payload` objects

---

**Condition Variable Interface**

- In ACE, the `Condition_Thread_Mutex` class is a wrapper for the native OS condition variable abstraction
  - e.g., `cond_t` on SunOS 5.x, `pthread_cond_t` for POSIX, and a custom implementation on Win32

```cpp
class Condition_Thread_Mutex
public:
    // Initialize the condition variable.
    Condition_Thread_Mutex (const Thread_Mutex &);
    // Implicitly destroy the condition variable.
    ~Condition_Thread_Mutex (void);

    // Block on condition, or until time has passed. If time == 0 use blocking semantics.
    int wait (Time_Value /*time == 0*/) const;
    // Signal one waiting thread.
    int signal (void) const;
    // Signal *all* waiting threads.
    int broadcast (void) const;
private:
    cond_t cond_; // Solaris condition variable.
    const Thread_Mutex &mutex_; // Reference to mutex lock.
};
```
The Message Block Class

- The contents of a message are represented by a Message Block

```cpp
class Message_Block {
  friend class Message_Queue;
public:
  Message_Block (size_t size,
                  Message_Type type = MB_DATA,
                  Message_Block *cont = 0,
                  char *data = 0,
                  Allocator *alloc = 0);
  // ...
private:
  char *base_;  // Pointer to beginning of payload.
  Message_Block *next_; // Pointer to next message in the queue.
  Message_Block *prev_; // Pointer to previous message in the queue.
  Message_Block *cont_; // Pointer to next fragment in this message.
  // ...
};
```

OO Design Interlude

- Q: What is the Allocator object in the Message_Block constructor?

- A: It provides extensible mechanism to control how memory is allocated and deallocated
  - This makes it possible to switch memory management policies without modifying Message_Block
  - By default, the policy is to use new and delete, but it's easy to use other schemes, e.g.,
    * Shared memory
    * Persistent memory
    * Thread-specific memory
  - A similar technique is also used in the C++ Standard Template Library

The Message_Queue Class Public Interface

- A Message_Queue is a thread-safe queueing facility for Message Blocks
  - The bulk of the locking is performed in the public methods

```cpp
template <class SYNCH_STRATEGY>
class Message_Queue {
  public:
    // Default high and low water marks.
    enum { DEFAULT_LWM = 0, DEFAULT_HWM = 4096 };  
    // Initialize a Message_Queue.
    Message_Queue (size_t hwm = DEFAULT_HWM,
                    size_t lwm = DEFAULT_LWM);
    // Check if full or empty (hold locks)
    int is_empty (void) const;
    int is_full (void) const;
    // Enqueue and dequeue Message_Block *'s.
    int enqueue_prio (Message_Block *, Time_Value *);
    int enqueue_tail (Message_Block *, Time_Value *);
    int dequeue_head (Message_Block *, Time_Value *);
};
```
The Message Queue Class
Private Interface

- The bulk of the work is performed in the private methods

```cpp
private:
    // Routines that actually do the enqueueing and
    // dequeuing (do not hold locks).
    int enqueue_prio_i (Message_Block *, Time_Value *);
    int enqueue_tail_i (Message_Block *new_item, Time_Value *);
    int dequeue_head_i (Message_Block *first_item);
    // Check the boundary conditions (do not hold locks).
    int is_empty_i (void) const;
    int is_full_i (void) const;
    // ...
    // Parameterized types for synchronization
    // primitives that control concurrent access.
    // Note use of C++ "traits"
    SYNCH_STRATEGY::MUTEX lock_i;
    SYNCH_STRATEGY::CONDITION not_empty_cond_i;
    SYNCH_STRATEGY::CONDITION not_full_cond_i;
```

OO Design Interlude

- **Q:** How should locking be performed in an OO class?

- **A:** In general, the following general pattern is useful:
  - "Public functions should lock, private functions should not lock"
    - This also helps to avoid intra-class method deadlock...
  - This is actually a variant on a common OO pattern that "public functions should check, private functions should trust"
  - Naturally, there are exceptions to this rule...

The Message Queue Class
Implementation

- Uses ACE synchronization wrappers

```cpp
template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::is_empty_i (void) const {
    return cur_bytes_ <= 0 && cur_count_ <= 0;
}

template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::is_full_i (void) const {
    return cur_bytes_ > high_water_mark_i;
}

template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::enqueue_tail_i (Message_Block *new_item, Time_Value *tv) {
    Guard<SYNCH_STRATEGY::MUTEX> m (lock_i);
    return is_empty_i ();
}

// Queue new item at the end of the list.
template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::enqueue_tail_i (Message_Block *new_item, Time_Value *tv) {
    Guard<SYNCH_STRATEGY::MUTEX> m (lock_i);
    return is_full_i ();
}
```

// Wait while the queue is full.
while (is_full_i ()) {
    // Release the lock and wait for timeout, signal,
    // or space becoming available in the list.
    if (not_full_cond_.wait (tv) == -1)
        return -1;
}

// Actually enqueue the message at the end of the list.
enqueue_tail_i (new_item);

// Tell blocked threads that list has a new item!
not_empty_cond_.signal ();
```
// Dequeue the front item on the list and return it to the caller.

template <class SYNCH_STRATEGY> int Message_Que_UE<SYNCH_STRATEGY>::dequeue_head (Message_Block *first_item, Time_Value *tv)
{
    Guard<SYNCH_STRATEGY::MUTEX> monitor (lock_);
    // Wait while the queue is empty.
    while (is_empty_i ())
    {
        // Release the lock_ and wait for timeout, signal,
        // or a new message being placed in the list.
        if (not_empty_cond_.wait (tv) == -1)
            return -1;
    }
    // Actually dequeue the first message.
    dequeue_head_i (first_item);
    // Tell blocked threads that list is no longer full.
    not_full_cond_.signal ();
}

Overcoming Algorithmic Decomposition Limitations

- The previous slides illustrate tactical OO techniques, idioms, and patterns that:
  1. Reduce accidental complexity e.g.,
     - Automate synchronization acquisition and release (C++ constructor/destructor idiom)
     - Improve consistency of synchronization interface (Adapter and Wrapper patterns)
  2. Eliminate race conditions

- The next slides describe strategic patterns, frameworks, and components that:
  1. Increase reuse and extensibility e.g.,
     - Decoupling solution from particular service, IPC and demultiplexing mechanisms
  2. Improve the flexibility of concurrency control

Selecting the Server's Concurrency Architecture

- **Problem**
  - A very strategic design decision for high-performance Web servers is selecting an efficient concurrency architecture

- **Forces**
  - No single concurrency architecture is optimal
  - Key factors include OS/hardware platform and workload

- **Solution**
  - Understand key alternative concurrency patterns

Concurrency Patterns in the Web Server

- The following example illustrates the patterns and framework components in an OO implementation of a concurrent Web Server

- There are various architectural patterns for structuring concurrency in a Web Server
  1. Reactive
  2. Thread-per-request
  3. Thread-per-connection
  4. Synchronous Thread Pool
  5. Asynchronous Thread Pool
Web Server Software Architecture

- **Event Dispatcher**
  - Encapsulates Web server concurrency and dispatching strategies

- **HTTP Handlers**
  - Parses HTTP headers and processes requests

- **HTTP Acceptor**
  - Accepts connections and creates HTTP Handlers

Patterns in the Web Server Implementation

<table>
<thead>
<tr>
<th>TACTICAL PATTERNS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STRATEGIC PATTERNS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

- **Thread-per Request**
- **Thread Pool**
- **Accepter**
- **Connector**
- **Service Configurator**
- **Asynchronous Completion Token**
- **Reactor/Proactor**
- **Double Checked Locking**
- **State**
- **Strategy**
- **Abstract Factory**
- **Adapter**
- **Singleton**
Patterns in the WWW Client/Server (cont’d)

- The WWW Client/Server uses same patterns as distributed logger
  - i.e., Reactor, Service Configurator, Active Object, and Acceptor

- It also contains following patterns:
  - Connector
    - “Decouple the active initialization of a service from the tasks performed once the service is initialized”
  - Double-Checked Locking Optimization
    - “Ensures atomic initialization of objects and eliminates unnecessary locking overhead on each access”
  - Half-Sync/Half-Async
    - “Decouple synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency”

Architecture of Our WWW Server

An Integrated Reactive/Active Web Server

The HTTP_Handler Public Interface

- The HTTP_Handler is the Proxy for communicating with clients (e.g., WWW browsers like Netscape or IE)
  - It implements the asynchronous portion of Half-Sync/Half-Async pattern

```c
template <class PEER_ACCEPTOR>
class HTTP_Handler :
public Svc_Handler<PEER_ACCEPTOR::PEER_STREAM, NULL_SYNCH> {
public:
  // Entry point into HTTP_Handler, called by HTTP_Acceptor.
  virtual int open (void *)
  {
    // Register with Reactor to handle client input.
    Reactor::instance ()->register_handler
      (this, READ_MASK);

    // Register timeout in case client doesn’t
    // send any HTTP requests.
    Reactor::instance ()->schedule_timer
      (this, 0. Time_Value (HTTP_CLIENT_TIMEOUT));
  }
```
The HTTP Handler Protected Interface

- The following methods are invoked by callbacks from the Reactor

```cpp
protected:
    // Reactor notifies when client's timeout.
    virtual int handle_timeout (const Time_Value &, const void *)
    {
        // Remove from the Reactor.
        Reactor::instance ()->remove_handler
            (this, READ_MASK);
    }

    // Reactor notifies when client HTTP requests arrive.
    virtual int handle_input (HANDLE);
    
    // Receive/frame client HTTP requests (e.g., GET).
    int recv_request (Message_Block *);
};
```

Integrating Multi-threading

- Problem
  - Multi-threaded Web servers are needed since Reactive Web servers are often inefficient and non-robust

- Forces
  - Multi-threading can be very hard to program
  - No single multi-threading model is always optimal

- Solution
  - Use the Active Object pattern to allow multiple concurrent server operations in an OO-manner

Using the Active Object Pattern in the WWW Server

- Processes HTTP requests using the "Thread-Pool" concurrency model to implement the synchronous task portion of the Half-Sync/Half-Async pattern

```cpp
class HTTP_Processor : public Task<MT_SYNCH> {
    public:
        // Singleton access point.
        static HTTP_Processor *instance (void);

        virtual int put (Message_Block *, Time_Value *);

        virtual int svc (int)
        {
            Message_Block *mb = 0; // Message buffer.

            // Wait for messages to arrive.
            for (;;)
            {
                getq (mb); // Inherited from class Task;
                // Identify and perform HTTP Server
                // request processing here...
        }
```
Using the Singleton

- The HTTP Processor is implemented as a Singleton that is created "on demand"

```cpp
HTTP_Processor *
HTTP_Processor::instance (void)
{
    // Singleton access point.
    HTTP_Processor *
    if (instance_ == 0)
        instance_ = new HTTP_Processor;
    return instance_;}
```

_HTTP Processor/:/:instance (void)

- Beware of race conditions!

```cpp
HTTP_Processor::HTTP_Processor (void)
{
    // Inherited from class Task.
    activate (THR_NEW_LWP, num_threads);
}
```

Constructor creates the thread pool.

HTTP_Processor::HTTP_Processor (void)

- Use the Double-Checked Locking optimization pattern to minimize locking and ensure atomic initialization

```cpp
static instance() static instance
if  (instance_ == NULL)  {
    mutex_.acquire ();
    if  (instance_  == NULL)
        instance_ = new HTTP_Processor;
    mutex_.release ();}
return instance_;}
```

Subtle Concurrency Woes with the Singleton Pattern

- Problem

- The canonical Singleton implementation has subtle "bugs" in multi-threaded applications

- Forces

- Too much locking makes Singleton too slow...

- Too little locking makes Singleton unsafe...

- Solution

- Use the Double-Checked Locking optimization pattern to minimize locking and ensure atomic initialization

The Double-Checked Locking Optimization Pattern

- Intent

- "Ensures atomic initialization of objects and eliminates unnecessary locking overhead on each access"

- This pattern resolves the following forces:

1. Ensures atomic initialization or access to objects, regardless of thread scheduling order

2. Keeps locking overhead to a minimum

   - e.g., only lock on first access

- Note, this pattern assumes atomic memory access...

Using the Double-Checked Locking Optimization Pattern for the WWW Server

```cpp
if (instance_ == NULL) {
    mutex_acquire ();
    if (instance_ == NULL)
        instance_ = new HTTP_Processor;
    mutex_release ();
}
return instance_;
```
Integrating Reactive and Multi-threaded Layers

- **Problem**
  - Justifying the hybrid design of our Web server can be tricky

- **Forces**
  - Engineers are never satisfied with the status quo ;-)  
  - Substantial amount of time is spent re-discovering the intent of complex concurrent software design

- **Solution**
  - Use the Half-Sync/Half-Async pattern to explain and justify our Web server concurrency architecture

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)

Structure of the Half-Sync/Half-Async Pattern

Collaboration in the Half-Sync/Half-Async Pattern

- This illustrates input processing (output processing is similar)
Using the Half-Sync/Half-Async Pattern in the WWW Server

Joining Async and Sync Tasks in the WWW Server

- The following methods form the boundary between the Async and Sync layers

```cpp
template <class PA>
int HTTP_Handler<PA>::handle_input (HANDLE h)
{
    Message_Block mb = 0;
    // Try to receive and frame message.
    if (recv_request (mb) == HTTP_REQUEST_COMPLETE) {
        Reactor::instance ()->remove_handler
            (this, READ_MASK);
        Reactor::instance ()->cancel_timer (this);
        // Insert message into the Queue.
        HTTP_Processor<PA>::instance ()->put (mb);
    }
}

HTTP_Processor::put (Message_Block msg, Time_Value *timeout) {
    // Insert the message on the Message Queue
    // (inherited from class Task).
    putq (msg, timeout);
}
```

Optimizing Our Web Server for Asynchronous Operating Systems

- **Problem**
  - Synchronous multi-threaded solutions are not always the most efficient

- **Forces**
  - Purely asynchronous I/O is quite powerful on some OS platforms
    - e.g., Windows NT 4.x
  - Good designs should be adaptable to new contexts

- **Solution**
  - Use the Proactor pattern to maximize performance on Asynchronous OS platforms

The Proactor Pattern

- **Intent**
  - “Decouples asynchronous event demultiplexing and event handler completion dispatching from service(s) performed in response to events”

- This pattern resolves the following forces for asynchronous event-driven software:
  - How to demultiplex multiple types of events from multiple sources of events asynchronously and efficiently within a minimal number of threads
  - How to extend application behavior without requiring changes to the event dispatching framework
Structure of the Proactor Pattern

- www.cs.wustl.edu/~schmidt/proactor.ps.gz
Structuring Service Initialization

- **Problem**
  - The communication protocol used between clients and the Web server is often orthogonal to the initialization protocol

- **Forces**
  - Low-level connection establishment APIs are tedious, error-prone, and non-portable
  - Separating initialization from use can increase software reuse substantially

- **Solution**
  - Use the Acceptor and Connector patterns to decouple passive service initialization from runtime protocol

The HTTP_Acceptor Class Interface

- **The HTTP_Acceptor class** implements the Acceptor pattern
  - *i.e.*, it accepts connections initializes **HTTP Handlers**

```cpp
template <class PEER_ACCEPTOR>
class HTTP_Acceptor : public PEER_ACCEPTOR
{
  public:
    // Called when HTTP_Acceptor is dynamically linked.
    virtual int init (int argc, char *argv[]);
  
    // Called when HTTP_Acceptor is dynamically unlinked.
    virtual int fini (void);

    // ...
};
```

The HTTP_Acceptor Class Implementation

- **// Initialize service when dynamically linked.**

```cpp
template <class PA> int HTTP_Acceptor<PA>::init (int argc, char *argv[])
{
  Options::instance ()->parse_args (argc, argv);

  // Initialize the communication endpoint and register to accept connections.
  peer_acceptor ()->open
    (PA::PEER_ADDR (Options::instance ()->port ()). Reactor::instance ());
}
```

- **// Terminate service when dynamically unlinked.**

```cpp
template <class PA> int HTTP_Acceptor<PA>::fini (void)
{
  // Shutdown threads in the pool.
  HTTP_Processor<PA>::instance ()->msg_queue ()->deactivate ();

  // Wait for all threads to exit.
  HTTP_Processor<PA>::instance ()->thr_mgr ()->wait ();
}
```
Using the Service Configurator Pattern in the WWW Server

- Existing service is based on Half-Sync/Half-Async "'Thread pool'" pattern
  - Other versions could be single-threaded, could use other concurrency strategies, and other protocols

svc.conf

FILE
dynamic Web_Server Service_Object *
www_server:make_Web_Server() "-ORBport 2001"

- The concurrent WWW Server is configured and initialized via a configuration script

```bash
%/ cat ./svc.conf
dynamic Web_Server Service_Object *
  www_server:make_Web_Server()
  "-p $PORT -t $THREADS"
# .dll or .so suffix added to "www_server" automatically
```

- Factory function that dynamically allocates a Half-Sync/Half-Async WWW Server object

```c
extern "C" Service_Object *make_Web_Server (void);
Service_Object *make_Web_Server (void)
{
  return new HTTP_Acceptor<SOCK_Acceptor>;
  // ACE dynamically unlinks and deallocates this object.
}
```

The Connector Pattern

- Intent
  - "Decouple the active initialization of a service from the task performed once a service is initialized"

- This pattern resolves the following forces for network clients that use interfaces like sockets or TLI:
  1. How to reuse active 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 enable flexible policies for creation, connection establishment, and concurrency
  4. How to efficiently establish connections with large number of peers or over a long delay path
Structure of the Connector Pattern

- www.cs.wustl.edu/~schmidt/Acc-Con.ps.gz

Collaboration in the Connector Pattern

- Synchronous mode

Collaboration in the Connector Pattern

- Asynchronous mode

Structure of the Connector Pattern in ACE
Using the Connector Pattern in a WWW Client

- e.g., in the Netscape HTML parser

Connector Class Public Interface

- A reusable template factory class that establishes connections with clients

```cpp
template <class SVC_HANDLER> // Type of service
class PEER_CONNECTOR> // Connection factory
class Connector
    : public Service_Object
{
    public:
        // Initiate connection to Peer.
        virtual int connect (SVC_HANDLER svc_handler,
            const PEER_CONNECTOR::PEER_ADDR &,
            Synch_Options &synch_options);

        // Cancel a <svc_handler> that was
        // started asynchronously.
        virtual int cancel (SVC_HANDLER svc_handler);
};
```

OO Design Interlude

- Q: What is the Synch_Options class?
- A: This allows callers to define the synchrony/asynchrony policies, e.g.,

```cpp
class Synch_Options
{
    // Options flags for controlling synchronization.
    enum {
        USE_REACTOR = 1,
        USE_TIMEOUT = 2
    };

    Synch_Options(u_long options = 0,
        const Time_Value &timeout
        = Time_Value::zero,
        const void *arg = 0);

    // This is the default synchronous setting.
    static Synch_Options synch;
    // This is the default asynchronous setting.
    static Synch_Options async;
};
```

Connector Class Protected Interface

```cpp
protected:
    // Demultiplexing hooks.
    virtual int handle_output (HANDLE); // Success.
    virtual int handle_input (HANDLE); // Failure.
    virtual int handle_timeout (Time_Value & timeout);

    // Create and cleanup asynchronous connections...
    virtual int create_svc_tuple (SVC_HANDLER *,
        Synch_Options &);
    virtual Svc_Tuple *cleanup_svc_tuple (HANDLE);

    // Table that maps an I/O handle to a Svc_Tuple *.
    Map_Manager<HANDLE, Svc_Tuple *, Null_Mutex> handler_map_;

    // Factory that actively establishes connections.
    PEER_CONNECTOR connector_;
OO Design Interlude

- **Q:** “What is a good technique to implementing a handler map?”
  - e.g., to route messages or to map HANDLES to SVC_HANDLERS

- **A:** Use a Map_Manager collection class
  - ACE provides a Map_Manager collection that associates external ids with internal ids, e.g.,
    * External ids → HANDLE
    * Internal ids → set of Svc_Handlers
  - **Map_Manager** uses templates to enhance reuse

Map_Manager Class

- Synchronization mechanisms are parameterized...

```cpp
template <class EXT_ID, class INT_ID, class LOCK>
class Map_Manager
{
public:
  bool bind (EXT_ID, INT_ID *);
  bool unbind (EXT_ID);
  bool find (EXT_ID ex, INT_ID &in) {
    // Exception-safe code...
    Read_Guard<LOCK> monitor (lock_);
    // lock_read_acquire ();
    if (find_i (ex, in))
      return true;
    else
      return false;
    // lock_release ();
  }
private:
  LOCK lock_;
  bool locate_entry (EXT_ID, INT_ID &);
  ...};
```

Connector Class Implementation

```cpp
// Shorthand names.
#define SH SVC_HANDLER
#define PC PEER_CONNECTOR

// Initiate connection using specified blocking semantics.
template <class SH, class PC> int
Connector<SH, PC>::connect
(SH *sh,
 const PC::PEER_ADDR &r_addr,
 Synch_Options &options) {
 Time_Value *timeout = 0;
 int use_reactor = options[Synch_Options::USE.REACTOR];
 if (use_reactor) timeout = Time_Value::zerop;
 else
   timeout = options[Synch_Options::USE.TIMEOUT]?
   (Time_Value *) &options.timeout () : 0;

 // Use Peer_Connector factory to initiate connection.
 if (connector_.connect (*sh, r_addr, timeout) == -1) {
   // If the connection hasn't completed, then
   // register with the Reactor to call us back.
   if (use_reactor && errno == EWOULDBLOCK)
     create_svc_tuple (sh, options);
   else
     // Activate immediately if we are connected.
     sh->open ((void *) this);
 }
```
The OO Architecture of the JAWS Framework

Web Server Optimization Techniques

- Use lightweight concurrency
- Minimize locking
- Apply file caching and memory mapping
- Use “gather-write” mechanisms
- Minimize logging
- Pre-compute HTTP responses
- Avoid excessive time calls
- Optimize the transport interface

www.cs.wustl.edu/~jxh/research/
Application-level Gateway Example

- The next example explores the patterns and reusable framework components used in an OO architecture for application-level Gateways.

- Gateways route messages between Peers in a large-scale telecommunication system.

- Peers and Gateways are connected via TCP/IP.

OO Software Architecture of the Gateway

- Components in the Gateway behave as follows:
  1. *Gateway* parses configuration files that specify which Peers to connect with and which routes to use.
  2. *Channel.Connector* connects to Peers, then creates and activates *Channel* subclasses (*Input_Channel* or *Output_Channel*).
  3. Once connected, Peers send messages to the Gateway:
     - Messages are handled by an *Input_Channel*.
     - *Input_Channels* work as follows:
       (a) Receive and validate messages
       (b) Consult a *Routing_Table*
       (c) Forward messages to the appropriate Peer(s) via *Output_Channels*.

Physical Architecture of the Gateway
• The Gateway components are based upon a system of patterns.

Class Diagram for Single-Threaded Gateway

OO Gateway Architecture

• The Gateway is decomposed into components that are layered as follows:
  
1. *Application-specific components*
   
   - Channels route messages among Peers

2. *Connection-oriented application components*
   
   - `SvcHandler`
     
     * Performs I/O-related tasks with connected clients

   - `Connector` factory
     
     * Establishes new connections with clients

3. *Application-independent ACE framework components*
   
   - Perform IPC, explicit dynamic linking, event demultiplexing, event handler dispatching, multi-threading, etc.
Using the Connector Pattern for the Gateway

Specializing Connector and Svc_Handler

- Producing an application that meets Gateway requirements involves specializing ACE components
  - Connector → Channel_Connector
  - Svc_Handler → Channel → Input_Channel and Output_Channel

- Note that these new classes selectively override methods defined in the base classes
  - The Reactor automatically invokes these methods in response to I/O, signal, and timer events

Channel Class Public Interface

- Common methods and data for I/O Channels

```c++
// Determine the type of threading mechanism.
#if defined (ACE_USE_MT)
typedef MT_SYNCH SYNCH;
#else
typedef NULL_SYNCH SYNCH;
#endif

// This is the type of the Routing_Table.
typedef Routing_Table <Peer_Addr,
Routing_Entry,
SYNCH::MUTEX> ROUTING_TABLE;

class Channel
 : public Svc_Handler<SOCK_Stream, SYNCH>
{
public:
  // Initialize the handler (called by Connector).
  virtual int open (void * = 0);

  // Bind addressing info to Router.
  virtual int bind (const INET_Addr & , CONN_ID);
```
Q: What is the MT_SYNCH class and how does it work?

A: MT_SYNCH provides a thread-safe synchronization policy for a particular instantiation of a Svc_Handler:

- e.g., it ensures that any use of a Svc_Handler’s MessageQueue will be thread-safe
- Any Task that accesses shared state can use the “traits” in the MT_SYNCH

```cpp
class MT_SYNCH { public:
    typedef Thread_Mutex MUTEX;
    typedef Condition_Thread_Mutex CONDITION;
};
```

- Contrast with NULL_SYNCH

```cpp
class NULL_SYNCH { public:
    typedef Null_Mutex MUTEX;
    typedef Null_Condition_Thread_Mutex CONDITION;
};
```

#### Channel Class Protected Interface

- Common data for I/O Channels

```cpp
protected:
    // Reconnect Channel if connection terminates.
    virtual int handle_close (HANDLE, Reactor_Mask);
    // Address of peer.
    INET_Addr addr;
    // The assigned connection ID of this Channel.
    CONN_ID id;
};
```

#### Detailed OO Architecture of the Gateway

- Handle input processing and routing of messages from Peers

```cpp
class Input_Channel : public Channel {
    public:
        Input_Channel (void);
    protected:
        // Receive and process Peer messages.
        virtual int handle_input (HANDLE);
        // Receive a message from a Peer.
        virtual int recv_peer (Message_Block *);
        // Action that routes a message from a Peer.
        int route_message (Message_Block *);
        // Keep track of message fragment.
        Message_Block *msg_frag;
};
```
**Output_Channel Interface**

- Handle output processing of messages sent to Peers

```cpp
class Output_Channel : public Channel {
public:
    Output_Channel (void);

    // Send a message to a Gateway (may be queued).
    virtual int put (Message_Block *, Time_Value * = 0);

protected:
    // Perform a non-blocking put().
    int nonblk_put (Message_Block *mb);

    // Finish sending a message when flow control abates.
    virtual int handle_output (HANDLE);

    // Send a message to a Peer.
    virtual int send_peer (Message_Block *);
};
```

**Channel_Connector Class Interface**

- A Concrete factory class that behaves as follows:

1. Establishes connections with Peers to produce Channels
2. Activates Channels, which then do the work

```cpp
class Channel_Connector : public
    Connector <Channel, // Type of service
    SOCK_Connector> // Connection factory
{
public:
    // Initiate (or reinitiate) a connection on Channel.
    int initiate_connection (Channel *);
};
```

- Channel_Connector also ensures reliability by restarting failed connections

**Channel_Connector Implementation**

- Initiate (or reinitiate) a connection to the Channel

```cpp
int Channel_Connector::initiate_connection (Channel *channel) {
    // Use asynchronous connections...
    if (connect (channel, channel->addr (),
        Sync_Options::asynch) == -1) {
        if (errno != EWOULDBLOCK)
            // Reschedule ourselves to try to connect again.
            Reactor::instance ()->schedule_timer
                (channel, 0, channel->timeout ());
        else
            return -1; // Failure.
    } else
        // We're connected.
        return 0;
}
```

**The Router Pattern**

- **Intent**
  - "Decouple multiple sources of input from multiple sources of output to prevent blocking"

- The Router pattern resolves the following forces for connection-oriented routers:
  - **How to prevent misbehaving connections from disrupting the quality of service for well-behaved connections**
  - **How to allow different concurrency strategies for Input and Output Channels**
Collaboration in the Router Pattern

START EVENT LOOP
FOREACH EVENT DO
DATA EVENT
RECV MSG
ROUTE MSG
FIND DESTINATIONS
SEND MSG (QUEUE IF FLOW CONTROLLED)
FLOW CONTROL ABATES
DEQUEUE AND SEND MSG (REQUEUE IF FLOW CONTROLLED)

main() : Routing Table : Output Channel : Input Channel reactor : Reactor
handle_events()
select()
handle_input()
recur peer()

input() : Input Channel : Output Channel
recv_peer()
schedule_wakeup()

output() : Input Channel : Output Channel
recv_peer()
schedule_wakeup()

// Receive input message from Peer and route the message.
int Input_Channel::handle_input (HANDLE)
{
    Message_Block *route_addr = 0;

    // Try to get the next message.
    if ((n = recv_peer (route_addr)) <= 0) {
        if (errno == EWOULDBLOCK) return 0;
        else return n;
    } 
    else
        route_message (route_addr);
}

// Send a message to a Peer (queue if necessary).
int Output_Channel::put (Message_Block *mb, Time_Value *)
{
    if (msg_queue->is_empty ( ) )
        // Try to send the message *without* blocking!
        nonblk_put (mb);
    else
        // Messages are queued due to flow control.
        msg_queue->enqueue_tail (mb, Time_Value::zerop);
}
// Route message from a Peer.
int
Input_Channel::route_messages (Message_Block *route_addr)
{
    // Determine destination address.
    CONN_ID route_id = *(CONN_ID *) route_addr->rd_ptr();
    const Message_Block *const data = route_addr->cont();
    Routing_Entry re = 0;
    // Determine route.
    Routing_Table::instance ()->find (route_id, re);
    // Initialize iterator over destination(s).
    SetIterator<Channel *> si (re->destinations ());
    // Multicast message.
    for (Channel *out_ch;
        si.next (out_ch) != -1;
        si.advance ()) {
        Message_Block *newmsg = data->duplicate ();
        if (out_ch->put (newmsg) == -1) // Drop message.
            newmsg->release (); // Decrement reference count.
    } delete route_addr;
}

// unique connection id that denotes a Channel.
typedef short CONN_ID;

// Peer address is used to identify the
// source/destination of a Peer message.
class Peer.Addr {
    public:
        CONN_ID conn_id_; // Unique connection id.
        u_char logical_id_; // Logical ID.
        u_char payload_; // Payload type.
    };

    // Fixed sized header.
class Peer_Header { public: /* ... */; }

    // Variable sized message (edu may be
    // between 0 and MAX_MSG_SIZE).
class Peer_Message {
    public:
        // The maximum size of a message.
        enum { MAX_PAYLOAD_SIZE = 1024 }; // Fixed-sized header portion.
        char sdu_[MAX_PAYLOAD_SIZE]; // Message payload.
    };

    // Pseudo-code for receiving framed message
    // (using non-blocking I/O).
int
Input_Channel::recv_peer (Message_Block *&route_addr)
{
    if (msg_frag_.is empty) {
        msg_frag_ = new Message_Block;
        receive fixed-sized header into msg_frag._
        if (errors occur)
            cleanup
        else
            determine size of variable-sized msg_frag._
    } else
        determine how much of msg_frag_ to skip
    perform non-blocking recv of payload into msg_frag_.
    if (entire message is now received) {
        route_addr = new Message_Block (sizeof(Peer.Addr).
            msgfrag_)
        Peer.Addr addr (id (), msgfrag_->routing_id_. 0);
        route_addr->copy (&addr, sizeof(Peer.Addr));
        return to caller and reset msgfrag_.
    } else if (only part of message is received)
        return errno = EWOULDBLOCK
    else if (fatal error occurs)
        cleanup
}
**OO Design Interlude**

- **Q**: How can a flow controlled Output_Channel know when to proceed again without polling or blocking?

- **A**: Use the Event_Handler::handle_output notification scheme of the Reactor
  
  - *i.e.*, via the Reactor’s methods `schedule_wakeup` and `cancel_wakeup`

- This provides cooperative multi-tasking within a single thread of control
  
  - The Reactor calls back to the handle_output method when the Channel is able to transmit again

---

// Simple implementation...

```c
int Output_Channel::send_peer (Message_Block *mb)
{
    ssize_t n;
    size_t len = mb->length ();

    // Try to send the message.
    n = peer ()->send (mb->rd_ptr (), len);

    if (n <= 0)
        return errno == EWOULDBLOCK ? 0 : n;
    else if (n < len)
        // Skip over the part we did send.
        mb->rd_ptr (n);
    else /* if (n == length) */ {
        delete mb; // Decrement reference count.
        errno = 0;
    }

    return n;
}
```

---

// Finish sending a message when flow control conditions abate. This method is automatically called by the Reactor.

```c
int Output_Channel::handle_output (HANDLE)
{
    Message_Block *mb = 0;

    // Take the first message off the queue.
    msg_queue_->dequeue_head
        (mb, Time_Value::zerop);

    if (nonblk_put (mb) != -1
        || errno != EWOULDBLOCK) {
        // If we succeed in writing msg out completely
        // (and as a result there are no more msgs
        // on the MessageQueue), then tell the Reactor
        // not to notify us anymore.

        if (msg_queue_->is_empty ()
            Reactor::instance ()->cancel_wakeup
                (this, Event_Handler::WRITE_MASK);
        }
    }
```
The Gateway Class

- This class integrates other application-specific and application-independent components

Gateway Class Public Interface

- Since Gateway inherits from Service_Object it may be dynamically (re)configured into a process at run-time

```cpp
// Parameterized by the type of I/O channels.
template <class INPUT_CHANNEL, // Input policies
class OUTPUT_CHANNEL> // Output policies
class Gateway
    : public Service_Object
    {
    public:
        // Perform initialization.
        virtual int init (int argc, char *argv[]);
        // Perform termination.
        virtual int fini (void);
    }
```

Gateway Class Private Interface

```cpp
protected:
    // Parse the channel table configuration file.
    int parse_cc_config_file (void);
    // Parse the routing table configuration file.
    int parse_rt_config_file (void);
    // Initiate connections to the Peers.
    int initiate_connections (void);
    // Table that maps Connection IDs to Channel *'s.
    Map_Manager<CONN_ID, Channel *, Null_Mutex> config_table_;
};
```

// Convenient short-hands.
#define IC INPUT_CHANNEL
#define OC OUTPUT_CHANNEL

// Pseudo-code for initializing the Gateway (called automatically on startup).
```cpp
template <class IC, class OC>
Gateway<IC, OC>::init (int argc, char *argv[])
{
    // Parse command-line arguments.
    parse_args (argc, argv);
    // Parse and build the connection configuration.
    parse_cc_config_file ();
    // Parse and build the routing table.
    parse_rt_config_file ();
    // Initiate connections with the Peers.
    initiate_connections ();
    return 0;
}
```
Configuration and Gateway
Routing

- The Gateway decouples the connection topology from the peer routing topology

**Configuration Files**

- The following config file specifies the connection topology among the Gateway and its Peers

<table>
<thead>
<tr>
<th>Conn ID</th>
<th>Hostname</th>
<th>Port</th>
<th>Direction</th>
<th>Max Retry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>peer1</td>
<td>10002</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>peer2</td>
<td>10002</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>peer3</td>
<td>10002</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>peer4</td>
<td>10002</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>peer5</td>
<td>10002</td>
<td>0</td>
<td>32</td>
</tr>
</tbody>
</table>

- The following config file specifies the routing topology among the Gateway and its Peers

<table>
<thead>
<tr>
<th>Conn ID</th>
<th>Logical ID</th>
<th>Payload</th>
<th>Destinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>30</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>09</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>13</td>
<td>1, 3</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>
// Initiate connections with the Peers.

int Gateway::initiate_connections (void)
{
    // Example of the Iterator pattern.
    Map <CONN_ID, Channel *, Null_Mutex> cti (connection_table);
    // Iterate through connection table
    // and initiate all channels.
    for (const Map_Entry <CONN_ID, Channel *> & me = 0;
        cti.next (me) != 0;
        cti.advance ()
    )
    
        // Initiate non-blocking connect.
        Channel *channel = me->int_id_;
        // Iterate through connection table
        // and initiate all channels.
        for (const Map_Entry <CONN_ID, Channel *> & me = 0;
            cti.next (me) != 0;
            cti.advance ()
        )
        
            // Initiate non-blocking connect.
            Channel *channel = me->int_id_;
            
            // Example of the Iterator pattern.
            Map <CONN_ID, Channel *, Null_Mutex> cti (connection_table);
            // Iterate through connection table
            // and initiate all channels.
            for (const Map_Entry <CONN_ID, Channel *> & me = 0;
                cti.next (me) != 0;
                cti.advance ()
            )
            
                // Initiate non-blocking connect.
                Channel *channel = me->int_id_;
                // Example of the Service Configurator pattern.
                int main (int argc, char *argv[])
                {
                    // Initialize the daemon and
                    // dynamically configure services.
                    Service_Config::open (argc, argv);
                    // Run forever, performing configured services.
                    Reactor::run_event_loop ();
                    /* NOTREACHED */
                }

Dynamically Configuring Services into an Application

- Main program is generic

Using the Service Configurator Pattern for the Gateway

- Replace the single-threaded Gateway with a multi-threaded Gateway

Dynamic Linking a Gateway Service

- Service configuration file

  % cat ./svc.conf
  static Svc_Manager "-p 5150"
  dynamic Gateway_Service Service_Object *
      Gateway::make_Gateway () "-d"
  # .dll or .so suffix added to "logger" automatically

- Application-specific factory function used to dynamically link a service

  // Dynamically linked factory function that allocates
  // a new single-threaded Gateway object.
  extern "C" Service_Object *make_Gateway (void);
  Service_Object *
  make_Gateway (void)
  {
      return new Gateway<Input_Channel, Output_Channel>;
      // ACE automatically deletes memory.
  }
Concurrency Strategies for Patterns

- The Acceptor and Connector patterns do not constrain the concurrency strategies of a Svc_Handler.

- There are three common choices:
  1. Run service in same thread of control
  2. Run service in a separate thread
  3. Run service in a separate process

- Observe how OO techniques push this decision to the “edges” of the design
  - This greatly increases reuse, flexibility, and performance tuning

Collaboration in the Active Object-based Gateway Routing

Using the Active Object Pattern for the Gateway

Using the Half-Sync/Half-Async Pattern in the Gateway
Thr_Output_Channel Class Interface

- New subclass of Channel uses the Active Object pattern for the Output_Channel
  - Uses multi-threading and synchronous I/O (rather than non-blocking I/O) to transmit message to Peers
  - Transparently improve performance on a multi-processor platform and simplify design

```cpp
#define ACE_USE_MT
#include "Channel.h"

class Thr_Output_Channel : public Output_Channel {
public:
  // Initialize the object and spawn a new thread.
  virtual int open (void /* */) {
    // Send a message to a peer.
    virtual int put (Message_Block *, Time_Value * = 0); 
    // Transmit peer messages within separate thread.
    virtual int svc (void);
  }
};
```

Thr_Output_Channel Class Implementation

- The multi-threaded version of open is slightly different since it spawns a new thread to become an active object!

```cpp
int
Thr_Output_Channel::open (void *)
{
  // Become an active object by spawning a
  // new thread to transmit messages to Peers.
  activate (THR_NEW_LWP | THR_DETACHED);
}

// Queue up a message for transmission (must not block
// since all Input_Channels are single-threaded).

```cpp
int
Thr_Output_Channel::put (Message_Block *mb, Time_Value *)
{
  // Perform non-blocking enqueue.
  msg_queue_->enqueue_tail (mb, Time_Value::zerop);
}
```

```cpp
// Transmit messages to the peer (note simplification
// resulting from threads...)

```cpp
int
Thr_Output_Channel::svc (void)
{
  Message_Block *mb = 0;

  // Since this method runs in its own thread it
  // is OK to block on output.
  while (msg_queue_->dequeue_head (mb) != -1)
    send_peer (mb);

  return 0;
}
```
Dynamic Linking a Gateway Service

- Service configuration file

```bash
% cat ./svc.conf
remove Gateway_Service
dynamic Gateway_Service Service_Object *
thr_Gateway:make_Gateway () "-d"
# .dll or .so suffix added to "thr_Gateway" automatically
```

- Application-specific factory function used to dynamically link a service

```c
// Dynamically linked factory function that allocates a new multi-threaded Gateway object.
extern "C" Service_Object *make_Gateway (void);
Service_Object *
make_Gateway (void)
{
    return new Gateway<Input_Context, Thr_Output_Context>;
    // ACE automatically deletes memory.
}
```

ACE Streams

- An ACE Stream allows flexible configuration of layered processing modules

- It is an implementation of the Pipes and Filters architectural pattern
  - This pattern provides a structure for systems that process a stream of data
  - Each processing step is encapsulated in a filter component
  - Data is passed through pipes between adjacent filters, which can be re-combined

Call Center Manager Example

Implementing a Stream in ACE

- A Stream contains a stack of Modules
- Each Module contains two Tasks
  - i.e., a read Task and a write Task
- Each Task contains a Message Queue and a pointer to a Thread Manager
Stream Class Category

Alternative Invocation Methods

Alternative Concurrency Models

ACE Stream Example: Parallel I/O Copy

Illustrates an implementation of the classic “bounded buffer” problem

The program copies stdin to stdout via the use of a multi-threaded Stream

In this example, the “read” Task is always ignored since the data flow is unidirectional
Producer and Consumer Object Interactions

5: write()
4: svc()
3: put()
2: svc()
1: read()

Producer Interface

• e.g.,

// typedef short-hands for the templates.
typedef Stream<MT_SYNCH> MT_Stream;
typedef Module<MT_SYNCH> MT_Module;
typedef Task<MT_SYNCH> MT_Task;

// Define the Producer interface.
class Producer : public MT_Task
{
    public:
        // Initialize Producer.
        virtual int open (void *)
        {
            // activate() is inherited from class Task.
            activate (THR_NEW_LWP);
        }

        // Read data from stdin and pass to consumer.
        virtual int svc (void);
    // ...
};

Consumer Class Interface

• e.g.,

// Define the Consumer interface.
class Consumer : public MT_Task
{
    public:
        // Initialize Consumer.
        virtual int open (void *)
        {
            // activate() is inherited from class Task.
            activate (THR_NEW_LWP);
        }

        // Enqueue the message on the Message_Queue for subsequent processing in svc().
        virtual int put (Message_Block*, Time_Value* = 0)
        {
            // putq() is inherited from class Task.
            return putq (mb, tv);
        }

        // Receive message from producer and print to stdout.
        virtual int svc (void);
    // ...
};
Evaluation of the Stream Class

Category

- Structuring active objects via a Stream allows "interpositioning"
  - Similar to adding a filter in a UNIX pipeline

- New functionality may be added by "pushing" a new processing Module onto a Stream,
  e.g.,

  ```
  stream.push (new MT_Module ("Consumer", new Consumer))
  stream.push (new MT_Module ("Filter", new Filter));
  stream.push (new MT_Module ("Producer", new Producer));
  ```

Concurrency Strategies

- Developing correct, efficient, and robust concurrent applications is challenging

- Below, we examine a number of strategies that addresses challenges related to the following:
  - Concurrency control
  - Library design
  - Thread creation
  - Deadlock and starvation avoidance
General Threading Guidelines

- A threaded program should not arbitrarily enter non-threaded (i.e., "unsafe") code
- Threaded code may refer to unsafe code only from the main thread
  - e.g., beware of errno problems
- Use reentrant OS library routines ("r") rather than non-reentrant routines
- Beware of thread global process operations
  - e.g., file I/O
- Make sure that main terminates via thr_exit(3T) rather than exit(2) or "falling off the end"

Thread Creation Strategies

- Use threads for independent jobs that must maintain state for the life of the job
- Don’t spawn new threads for very short jobs
- Use threads to take advantage of CPU concurrency
- Only use "bound" threads when absolutely necessary
- If possible, tell the threads library how many threads are expected to be active simultaneously
  - e.g., use thr_setconcurrency

General Locking Guidelines

- Don’t hold locks across long duration operations (e.g., I/O) that can impact performance
  - Use “Tokens” instead…
- Beware of holding non-recursive mutexes when calling a method outside a class
  - The method may reenter the module and deadlock
- Don’t lock at too small of a level of granularity
- Make sure that threads obey the global lock hierarchy
  - But this is easier said than done…

Locking Alternatives

- Code locking
  - Associate locks with body of functions
    - Typically performed using bracketed mutex locks
    - Often called a monitor
- Data locking
  - Associate locks with data structures and/or objects
  - Permits a more fine-grained style of locking
- Data locking allows more concurrency than code locking, but may incur higher overhead
**Single-lock Strategy**

- One way to simplify locking is use a single, application-wide mutex lock

- Each thread must acquire the lock before running and release it upon completion

- The advantage is that most legacy code doesn’t require changes

- The disadvantage is that parallelism is eliminated
  - Moreover, interactive response time may degrade if the lock isn’t released periodically

**Passive Object Strategy**

- A more OO locking strategy is to use a “Passive Object”
  - Also known as a “monitor”

- Passive Object synchronization mechanisms allow concurrent method invocations
  - Either eliminate access to shared data or use synchronization objects
  - Hide locking mechanisms behind method interfaces
    - Therefore, modules should not export data directly

- Advantage is transparency

- Disadvantages are increased overhead from excessive locking and lack of control over method invocation order

**Active Object Strategy**

- Each task is modeled as an active object that maintains its own thread of control

- Messages sent to an object are queued up and processed asynchronously with respect to the caller
  - i.e., the order of execution may differ from the order of invocation

- This approach is more suitable to message passing-based concurrency

- The ACE Task class implements this approach

**Invariants**

- In general, an invariant is a condition that is always true

- For concurrent programs, an invariant is a condition that is always true when an associated lock is not held
  - However, when the lock is held the invariant may be false
  - When the code releases the lock, the invariant must be re-established

- e.g., enqueueing and dequeueing messages in the MessageQueue class
Run-time Stack Problems

- Most threads libraries contain restrictions on stack usage
  - The initial thread gets the “real” process stack, whose size is only limited by the stacksize limit
  - All other threads get a fixed-size stack
    * Each thread stack is allocated off the heap and its size is fixed at startup time

- Therefore, be aware of “stack smashes” when debugging multi-threaded code
  - Overly small stacks lead to bizarre bugs, e.g.,
    * Functions that weren’t called appear in backtraces
    * Functions have strange arguments

Avoiding Deadlock in OO Frameworks

- Deadlock can occur due to properties of OO frameworks, e.g.,
  - Callbacks
  - Inter-class method calls

- There are several solutions
  - Release locks before performing callbacks
    * Every time locks are reacquired it may be necessary to reevaluate the state of the object
  - Make private “helper” methods that assume locks are held when called by methods at higher levels
  - Use a Token or a Recursive Mutex

Deadlock

- Permanent blocking by a set of threads that are competing for a set of resources

- Caused by “circular waiting,” e.g.,
  - A thread trying to reacquire a lock it already holds
  - Two threads trying to acquire resources held by the other
    * e.g., $T_1$ and $T_2$ acquire locks $L_1$ and $L_2$ in opposite order

- One solution is to establish a global ordering of lock acquisition (i.e., a lock hierarchy)
  - May be at odds with encapsulation...

Recursive Mutex

- Not all thread libraries support recursive mutexes
  - Here is portable implementation available in ACE:
    ```cpp
    class Recursive_Thread_Mutex
    {
    public:
        // Initialize a recursive mutex.
        Recursive_Thread_Mutex (void);
        // Implicitly release a recursive mutex.
        ~Recursive_Thread_Mutex (void);
        // Acquire a recursive mutex.
        int acquire (void) const;
        // Conditionally acquire a recursive mutex.
        int tryacquire (void) const;
        // Releases a recursive mutex.
        int release (void) const;
    
    private:
        Thread_Mutex nesting_mutex_;
        Condition_Thread_Mutex mutex_available_;
        thread_t owner_id_;
        int nesting_level_;
    };
    ```
Avoiding Starvation

- Starvation occurs when a thread never acquires a mutex even though another thread periodically releases it.

- The order of scheduling is often undefined.

- This problem may be solved via:
  - Use of "voluntary pre-emption" mechanisms
    - e.g., `thr_yield()` or `Sleep`
  - Using a "Token" that strictly orders acquisition and release.

Drawbacks to Multi-threading

- **Performance overhead**
  - Some applications do not benefit directly from threads
  - Synchronization is not free
  - Threads should be created for processing that lasts at least several 1,000 instructions.

- **Correctness**
  - Threads are not well protected against interference from other threads.
  - Concurrency control issues are often tricky.
  - Many legacy libraries are not thread-safe.

- **Development effort**
  - Developers often lack experience.
  - Debugging is complicated (lack of tools).
Lessons Learned using OO Patterns

- **Benefits of patterns**
  - Enable large-scale reuse of software architectures
  - Improve development team communication
  - Help transcend language-centric viewpoints

- **Drawbacks of patterns**
  - Do not lead to direct code reuse
  - Can be deceptively simple
  - Teams may suffer from pattern overload

Lessons Learned using OO Frameworks

- **Benefits of frameworks**
  - Enable direct reuse of code (cf patterns)
  - Facilitate larger amounts of reuse than stand-alone functions or individual classes

- **Drawbacks of frameworks**
  - High initial learning curve
    - Many classes, many levels of abstraction
  - The “inversion of control” for reactive dispatching may be non-intuitive
  - Verification and validation of generic components is hard

Lessons Learned using C++

- **Benefits of C++**
  - Classes and namespaces modularize the system architecture
  - Inheritance and dynamic binding decouple application policies from reusable mechanisms
  - Parameterized types decouple the reliance on particular types of synchronization methods or network IPC interfaces

- **Drawbacks of C++**
  - Many language features are not widely implemented
  - Development environments are primitive
  - Language has many dark corners and sharp edges

Software Principles for Distributed Applications

1. **Use patterns and frameworks to separate policies from mechanisms**
   - Enhance reuse of common concurrent programming components

2. **Decouple service functionality from configuration-related mechanisms**
   - Improve flexibility and performance

3. **Utilize OO class abstractions, inheritance, dynamic binding, and parameterized types**
   - Improve extensibility and modularity
Software Principles for Distributed Applications (cont’d)

1. Use advanced OS mechanisms to enhance performance and functionality
   - e.g., implicit and explicit dynamic linking and multi-threading

2. Perform commonality/variability analysis
   - Identify uniform interfaces for variable components and support pluggability of variation

Conferences and Workshops on Patterns

- Pattern Language of Programs Conferences
  - September, 1999, Monticello, Illinois, USA
  - st-www.cs.uiuc.edu/users/patterns/patterns.html

- The European Pattern Languages of Programming conference
  - July, 1999, Kloster Irsee, Germany
  - www.cs.wustl.edu/~schmidt/patterns.html

- USENIX COOTS
  - May 3–7, 1999, San Diego, CA
  - www.usenix.org/events/coots99/

Patterns and Frameworks Literature

- Books
  - Gamma et al., “Design Patterns: Elements of Reusable Object-Oriented Software” AW, 1994
  - Siemens, Pattern-Oriented Software Architecture, Wiley and Sons, 1996

- Special Issues in Journals
  - October ’96 CACM (guest editors: Douglas C. Schmidt, Ralph Johnson, and Mohamed Fayad)
  - October ’97 CACM (guest editors: Douglas C. Schmidt and Mohamed Fayad)

- Magazines
  - C++ Report and JOOP, columns by Coplien, Vlissides, Vinoski, Schmidt, and Martin

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
  - www.cs.wustl.edu/~schmidt/ACE.html

- Mailing lists
  * ace-users@cs.wustl.edu
  * ace-users-request@cs.wustl.edu
  * ace-announce@cs.wustl.edu
  * ace-announce-request@cs.wustl.edu

- Newsgroup
  - comp.soft-sys.ace
Concluding Remarks

- Developers of communication software confront recurring challenges that are largely application-independent
  - e.g., service initialization and distribution, error handling, flow control, event demultiplexing, concurrency control

- Successful developers resolve these challenges by applying appropriate patterns to create communication frameworks

- Frameworks are an effective way to achieve broad reuse of software