Object-Oriented Design Patterns for Network Programming in C++

Douglas C. Schmidt
Washington University, St. Louis

http://www.cs.wustl.edu/~schmidt/

schmidt@cs.wustl.edu

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 reconfiguration*
  - 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

- Object-oriented (OO) techniques and OO language features help to enhance concurrent software quality factors
  - Key OO techniques include *design 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. *Concurrent WWW client/server*
  2. *OO framework for layered active objects*
- Discuss general concurrent programming strategies

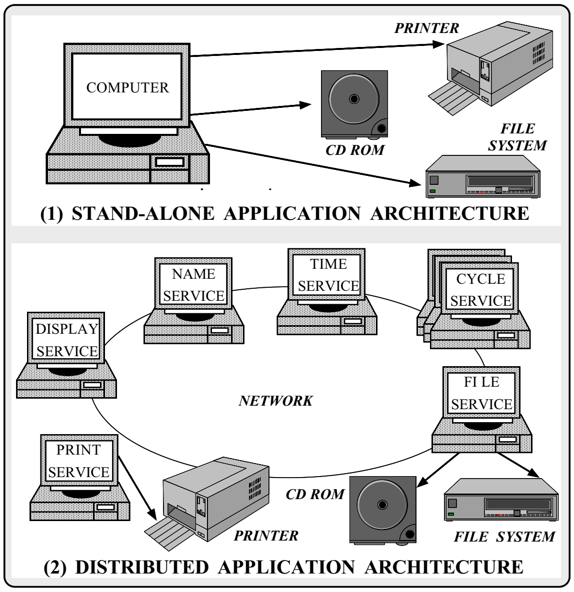
Software Development Environment

- The topics discussed here are largely independent of OS, network, and programming language
  - Currently used successfully on UNIX and Windows NT platforms, running on TCP/IP and IPX/SPX 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.

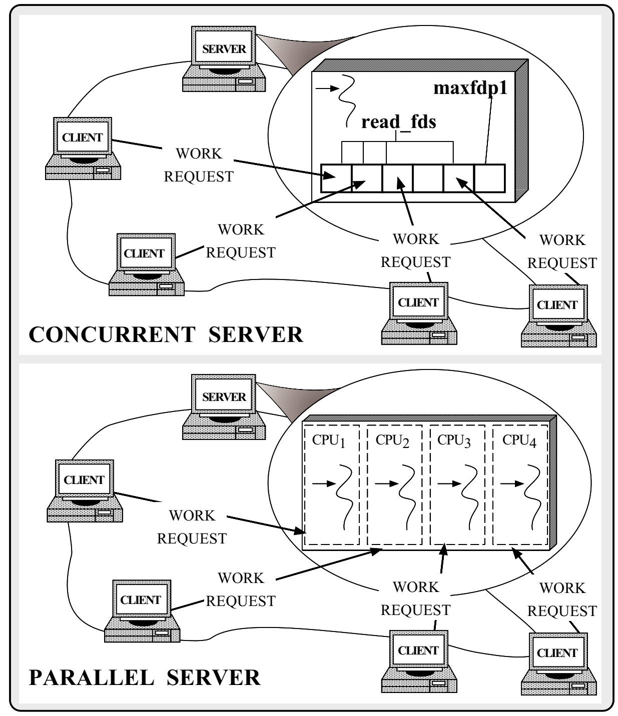
Definitions

- *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

Stand-alone vs. Distributed Application Architectures



Concurrency vs. Parallelism



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

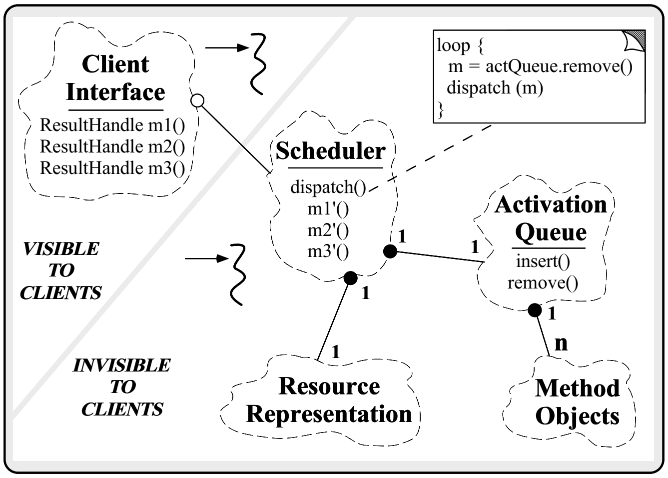
OO Contributions to Concurrent Applications

- UNIX 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 and Solaris threads*
- OO *design 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*

Design Patterns

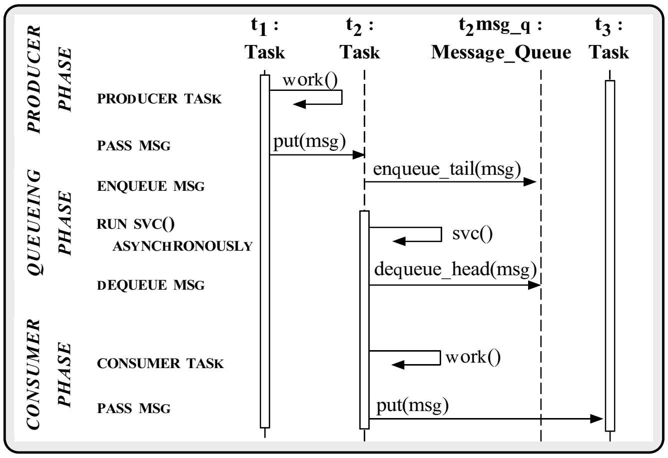
- Design patterns represent *solutions* to *problems* that arise when developing software within a particular *context*
  - *i.e.*, "Patterns == problem/solution pairs in a context"
- Patterns capture the static and dynamic *structure* and *collaboration* among key *participants* in software designs
  - They are particularly useful for articulating how and why to resolve *non-functional forces*
- Patterns facilitate reuse of successful software architectures and designs

Active Object Pattern



- *Intent*: decouples the thread of method execution from the thread of method invocation

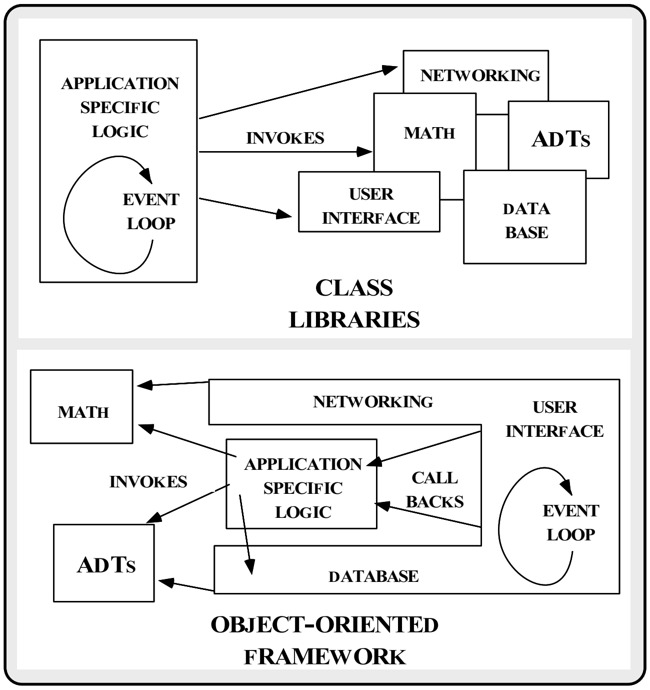
Collaboration in the Active Object Pattern



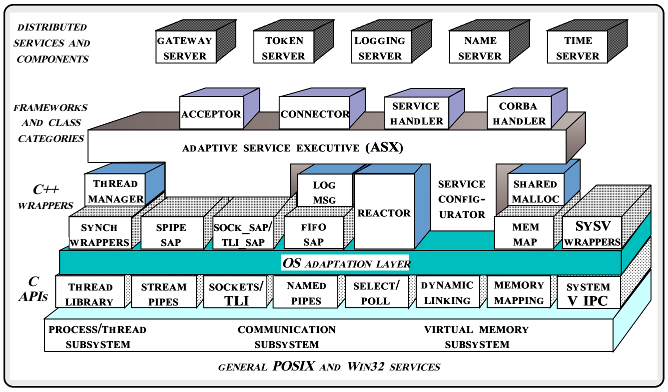
Frameworks

- 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

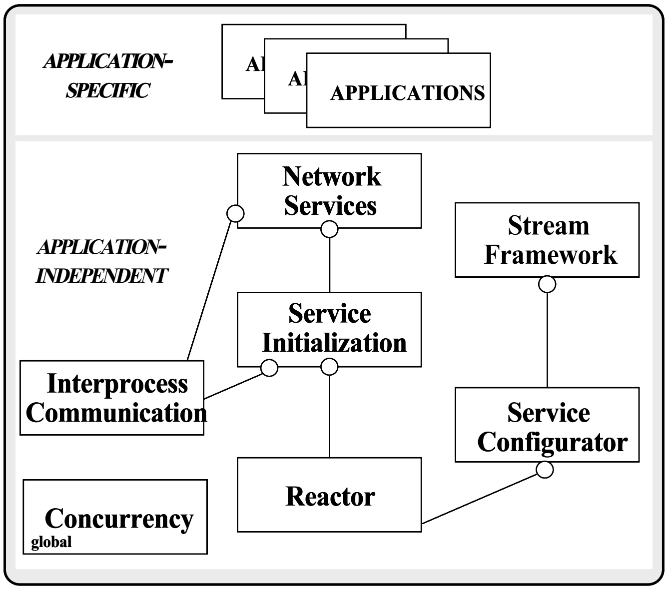


The ADAPTIVE Communication Environment (ACE)



- A set of C++ wrappers, class categories, and frameworks based on design patterns

Class Categories in ACE



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

Concurrency 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.

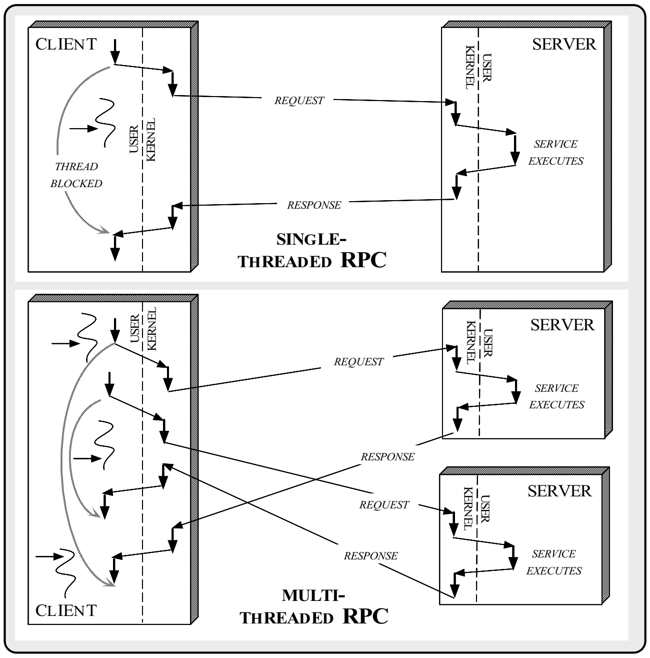
Lightweight Concurrency

- 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

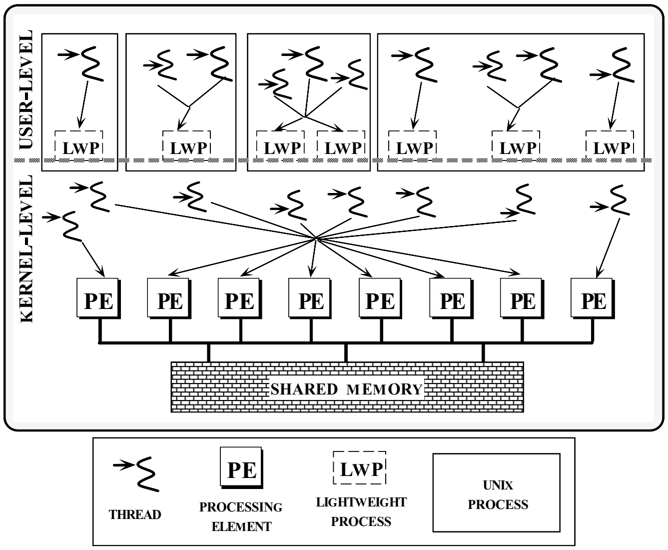
Single-threaded vs. Multi-threaded RPC



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...

Sun MP Model (cont'd)



- Application threads may be *bound* and/or *unbound*

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

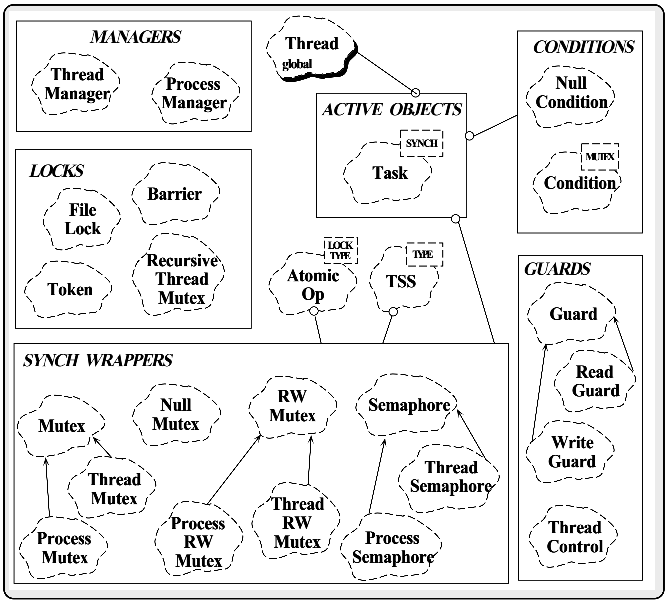
Common OS Synchronization Mechanisms

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

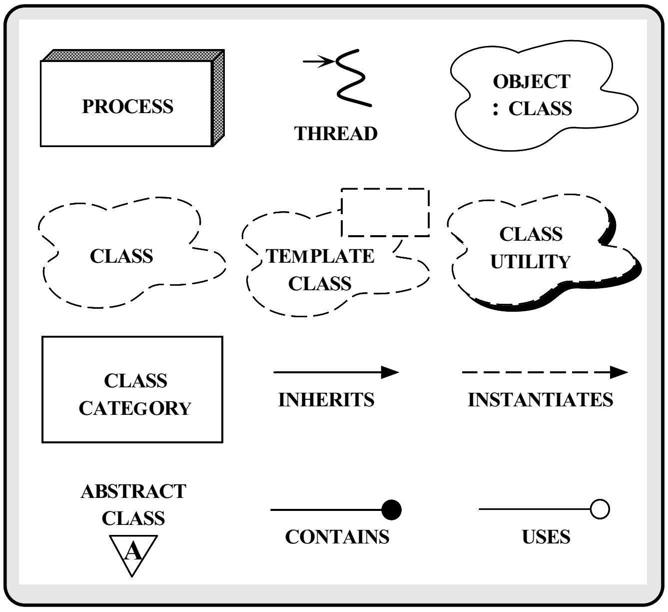
Additional ACE Synchronization Mechanism

1. *Guards*
   - An exception-safe scoped locking mechanism
2. *Barriers*
   - Allows threads to synchronize their completion
3. *Token*
   - Provides absolute 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



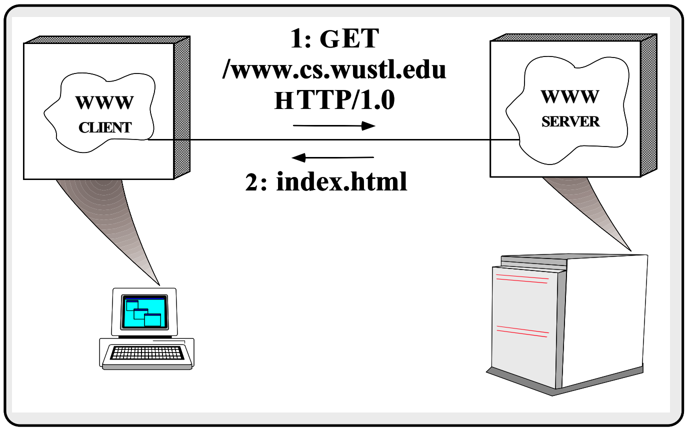
Graphical Notation



Concurrent WWW Client/Server Example

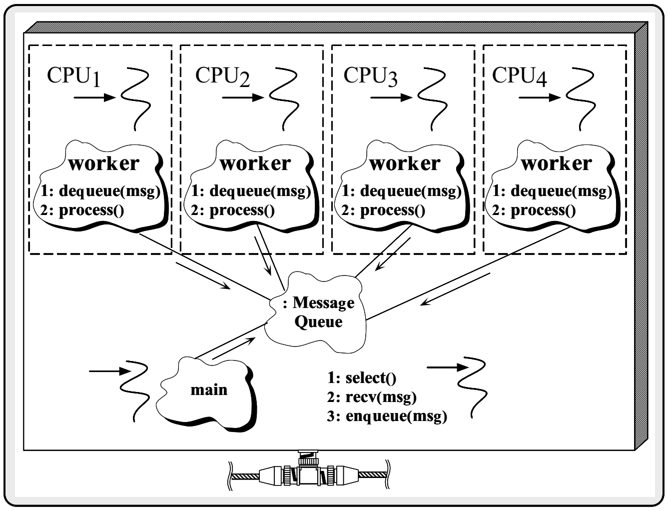
- The following example illustrates a concurrent OO architecture for a high-performance WWW client/server
- Key system requirements are:
  1. Robust implementation of HTTP 1.0 protocol
     - *i.e.*, resilient to incorrect or malicious WWW clients/servers
  2. Extensible for use with other protocols
     - *e.g.*, DICOM, HTTP 1.1, etc.
  3. Leverage multi-processor hardware and OS software
     - *e.g.*, Support various concurrency models

WWW Client/Server Architecture



- www.w3.org/pub/WWW/Protocols/HTTP/

Multi-threaded WWW Server Architecture



- Worker threads execute within one process

Pseudo-code for Concurrent WWW Server

- Pseudo-code for master server

```
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

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

OO Design Interlude

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

- A:
  - *Separation of concerns*
    - ▷ *Shield application from semantics of thread library* (user-level vs. kernel-level threads)
  - *Promotes more efficient use of multiple CPUs via load balancing*
  - *Enables transparent interpositioning*
  - *Makes it easier to shut down the system correctly*

- *Drawbacks*
  - Using a message queue may lead to greater *context switching* and *synchronization* overhead...

Thread Entry Point

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

```
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:

```
// Thread function prototype.
typedef void *(*THR_FUNC)(void *);
static const int NUM_THREADS = /* ... */;

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, 0, 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.
  while (thr_join (0, &t_id, (void **) 0) == 0)
    continue; // ...
}
```


Pseudo-code for recv_requests()

- *e.g.,*

```
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 object-oriented *framework* components and commonly recurring *design patterns*

Eliminating Race Conditions (Part 1 of 2)

- *Problem*
  - The concurrent server illustrated above contains "race conditions"
    - ▷ *e.g.*, auto-increment of global variable `request_count` is not serialized properly
- *Forces*
  - Modern shared memory multi-processors use *deep caches* and *weakly ordered* memory models
  - Access to shared data must be protected from corruption
- *Solution*
  - Use synchronization mechanisms

Basic Synchronization Mechanisms

- One approach to solve the serialization problem is to use OS mutual exclusion mechanisms explicitly, *e.g.*,

```
// SunOS 5.x, implicitly "unlocked".
mutex_t lock;
typedef u_long COUNTER;
COUNTER request_count;

void *worker (Message_Queue *msg_queue)
{
  // in function scope ...
    mutex_lock (&lock);
    ++request_count;
    mutex_unlock (&lock);
  // ...
}
```

- However, adding these `mutex_*` calls explicitly is *inelegant*, *obtrusive*, *error-prone*, and *non-portable*

C++ Wrappers for Synchronization

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

```
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 Thread_Mutex to Windows NT

- Win32 version of `Thread_Mutex`

```
class Thread_Mutex
{
public:
  Thread_Mutex (void) {
    InitializeCriticalSection (&lock_);
  }
  ~Thread_Mutex (void) {
    DeleteCriticalSection (&lock_);
  }
  int acquire (void) {
    EnterCriticalSection (&lock_); return 0;
  }
  int tryacquire (void) {
    TryEnterCriticalSection (&lock_); return 0;
  }
  int release (void) {
    LeaveCriticalSection (&lock_); return 0;
  }
private:
  CRITICAL_SECTION lock_; // Win32 locking mechanism.
  // ...
```


Using the C++ Thread_Mutex Wrapper

- Using the C++ wrapper helps improve portability and elegance:

```
Thread_Mutex lock;
typedef u_long COUNTER;
COUNTER request_count;

// ...
void *worker (Message_Queue *msg_queue)
{
    lock.acquire ();
    ++request_count;
    lock.release (); // Don't forget to call!

  // ...
}
```

- However, it does not solve the *obtrusiveness* 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

```
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*

Using the Guard Class

- Using the Guard class helps reduce errors:

```
Thread_Mutex lock;
typedef u_long COUNTER;
COUNTER request_count;

void *worker (Message_Queue *msg_queue) {
  // ...
    { Guard<Thread_Mutex> monitor (lock);
      ++request_count;
    }
  // ...
}
```

- However, using the Thread_Mutex and Guard classes is still overly obtrusive and subtle (*e.g.*, beware of elided braces)...

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

OO Design Interlude

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

- A: there are many locking mechanisms that benefit from 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 classes use the Wrapper and Adapter patterns to provide identical interfaces that facilitate parameterization

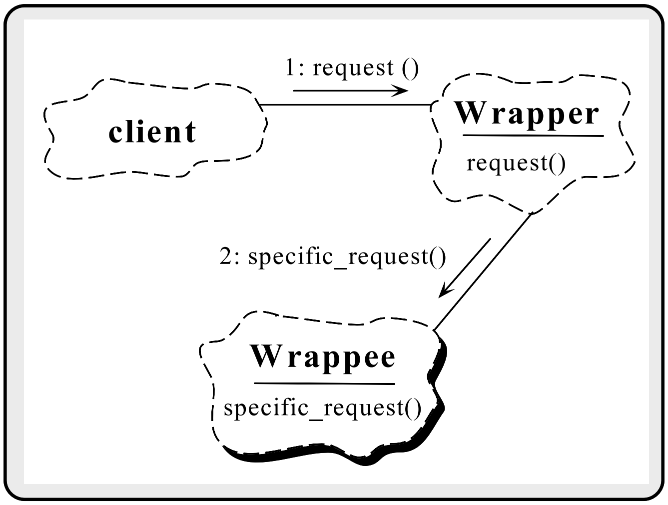
The Wrapper Pattern

- *Intent*
  - "Encapsulate low-level, stand-alone functions within type-safe, modular, and portable class interfaces"

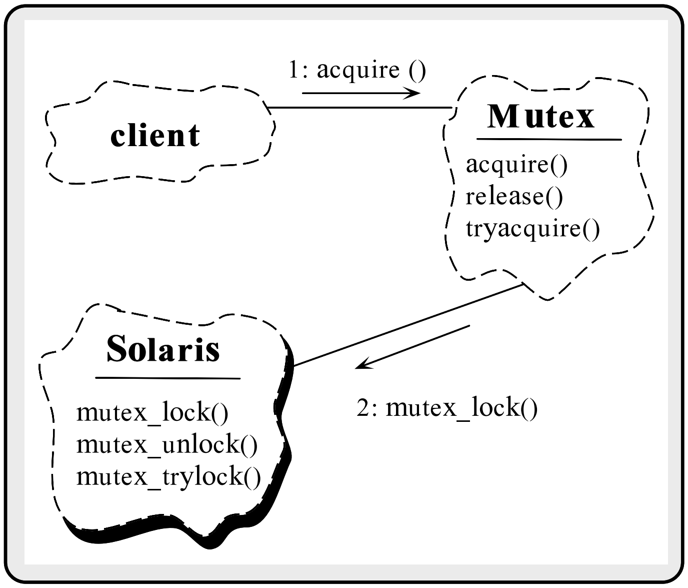
- This pattern resolves the following forces that arises when using native C-level OS APIs

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

Structure of the Wrapper Pattern



Using the Wrapper Pattern for Locking



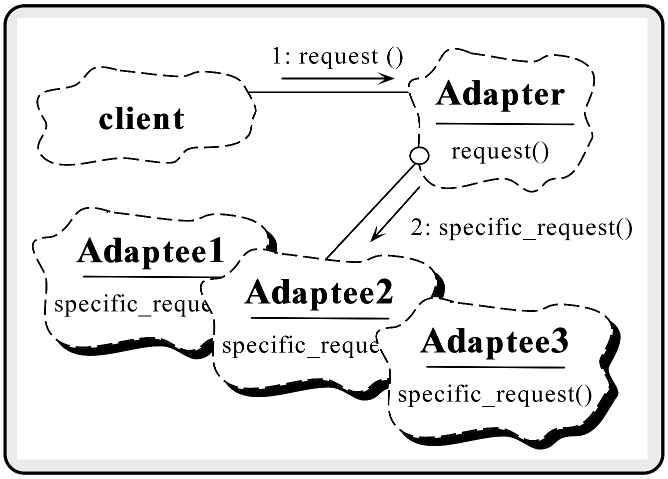
The Adapter Pattern

- *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

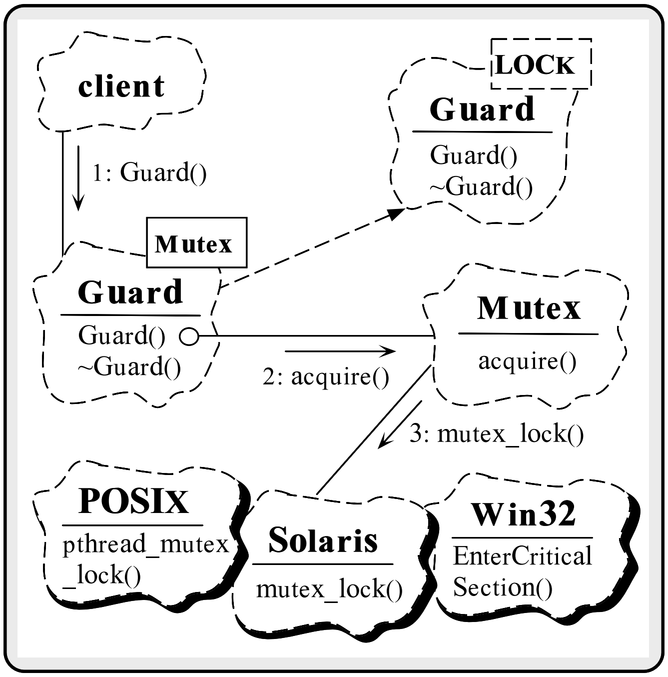
- This pattern resolves the following force that arises when using conventional OS interfaces

1. *How to provide an interface that expresses the similarities of seeminglying different OS mechanisms (such as locking or IPC)*

Structure of the Adapter Pattern



Using the Adapter Pattern for Locking



Transparently Parameterizing Synchonization Using C++

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

```
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_;
  }

  operator TYPE () {
    Guard<LOCK> m (lock_);
    return count_;
  }
  // Other arithmetic operations omitted...

private:
  LOCK lock_;
  TYPE  count_;
};
```


Thread-safe Version of Concurrent Server

- Using the `Atomic_Op` class, only one change is made to the code

```
#if defined (MT_SAFE)
typedef Atomic_Op<> COUNTER; // Note default parameters...
#else
typedef Atomic_Op<Null_Mutex> COUNTER;
#endif /* MT_SAFE */
COUNTER request_count;
```

- `request_count` is now serialized automatically

```
void *worker (Message_Queue *msg_queue)
{
    //...
    // Calls Atomic_Op::operator++(void)
    ++request_count;
    //...
}
```


Eliminating Race Conditions (Part 2 of 2)

- *Problem*
  - A naive implementation of `Message_Queue` 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 control
- *Solution*
  - Utilize a "Condition object"

Condition Object Overview

- `Condition` objects are used to "sleep/wait" until a particular condition involving shared data is signaled
  - Conditions 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 Object Usage

- A particular idiom is associated with acquiring resources via `Condition` objects

```
// Global variables
static Thread_Mutex lock; // Initially unlocked.
// Initially unlocked.
static Condition<Thread_Mutex> cond (lock);

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 Object Usage (cont'd)

- Another idiom is associated with releasing resources via `Condition` objects

```
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!

Condition Object Interface

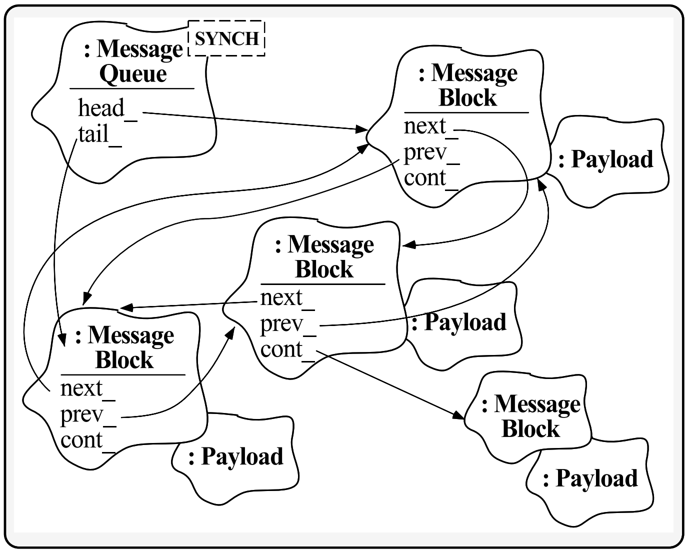
- In ACE, the `Condition` class is a wrapper for the native OS condition variable abstraction
  – *e.g.*, `cond_t` on SunOS 5.x, `pthread_cond_t` for POSIX, and a custom implementation on Win32

```
template <class MUTEX>
class Condition
{
public:
    // Initialize the condition variable.
  Condition (const MUTEX&, int=USYNC_THREAD);
    // Implicitly destroy the condition variable.
  ~Condition (void);

    // Block on condition, or until abstime has
    // passed.  If abstime == 0 use blocking semantics.
  int wait (Time_Value *abstime = 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 MUTEX &mutex_; // Reference to mutex 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 **`m_blks`**
  – 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



The Message_Block Class

- The contents of a message are represented by a `Message_Block`

```
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 the `Message_Block` class
  - 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

OO Design Interlude

- Here's an example of the interfaces used in ACE
  - Note the use of the Adapter pattern to integrate third-party memory allocators

```
class Allocator {
  // ...
  virtual void *malloc (size_t nbytes) = 0;
  virtual void free (void *ptr) = 0;
};

template <class ALLOCATOR>
class Allocator_Adapter : public Allocator {
  // ...
  virtual void *malloc (size_t nbytes) {
    return allocator_.malloc (nbytes);
  }

  ALLOCATOR allocator_;
};

Allocator_Adapter<Shared_Alloc> sh_malloc;
Allocator_Adapter<New_Alloc> new_malloc;
Allocator_Adapter<Persist_Alloc> p_malloc;
Allocator_Adapter<TSS_Alloc> p_malloc;
```

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

```
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

```
private:
    // Routines that actually do the enqueueing and
    // dequeueing (do not hold locks).
  int enqueue_prio_i (Message_Block *, Time_Value *);
  int enqueue_tail_i (Message_Block *new_item);
  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_;
  SYNCH_STRATEGY::CONDITION not_empty_cond_;
  SYNCH_STRATEGY::CONDITION not_full_cond_;
};
```


The Message_Queue Class Implementation

- Uses ACE synchronization wrappers

```
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_;
}

template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::is_empty (void) const {
  Guard<SYNCH_STRATEGY::MUTEX> m (lock_);
  return is_empty_i ();
}

template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::is_full (void) const {
  Guard<SYNCH_STRATEGY::MUTEX> m (lock_);
  return is_full_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 inter-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...

```
// Queue new item at the end of the list.

template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::enqueue_tail
  (Message_Block *new_item, Time_Value *tv)
{
  Guard<SYNCH_STRATEGY::MUTEX> monitor (lock_);

  // 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_Queue<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 ();
}
```

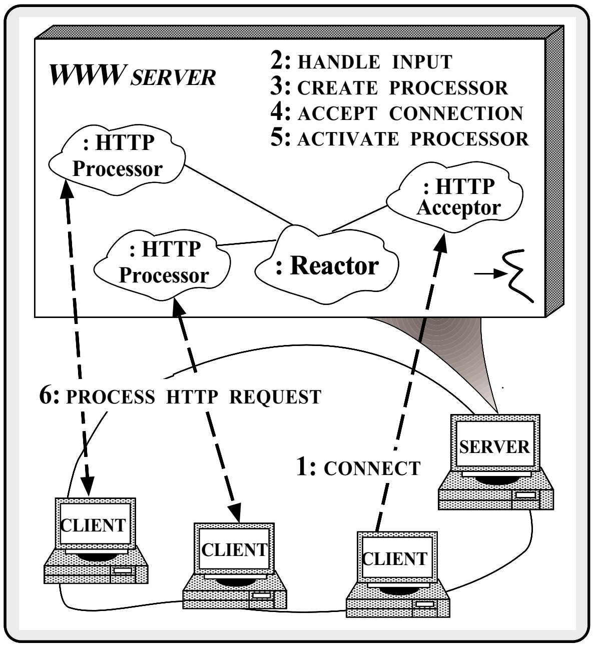

Overcoming Algorithmic Decomposition Limitations

- The previous slides illustrate low-level 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 higher-level ACE framework components and patterns that:
  1. *Increase reuse and extensibility e.g.,*
     - Decoupling solution from particular service, IPC and demultiplexing mechanisms
  2. *Improve the flexibility of concurrency control*

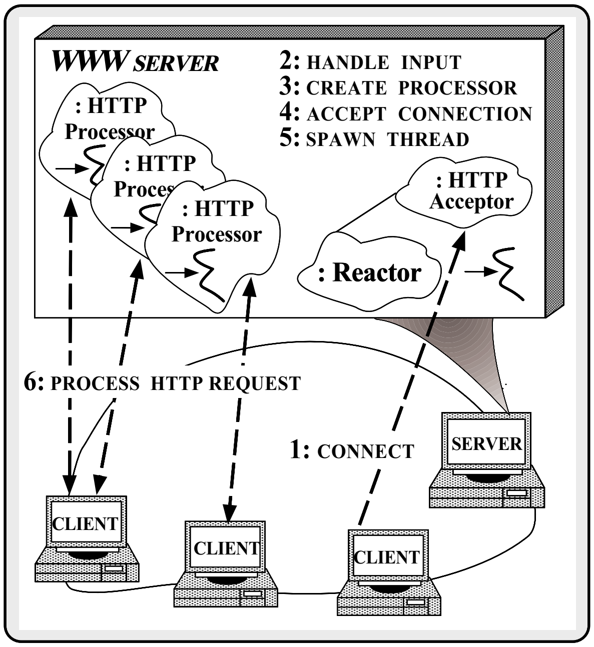
A Concurrent OO WWW Server

- The following example illustrates an OO solution to the concurrent WWW Server
  - The active objects are based on the ACE `Task` class
- There are several ways to structure concurrency in an WWW Server
  1. *Single-threaded*
  2. *Thread-per-request*
  3. *Thread-per-session*
  4. *Thread-pool*

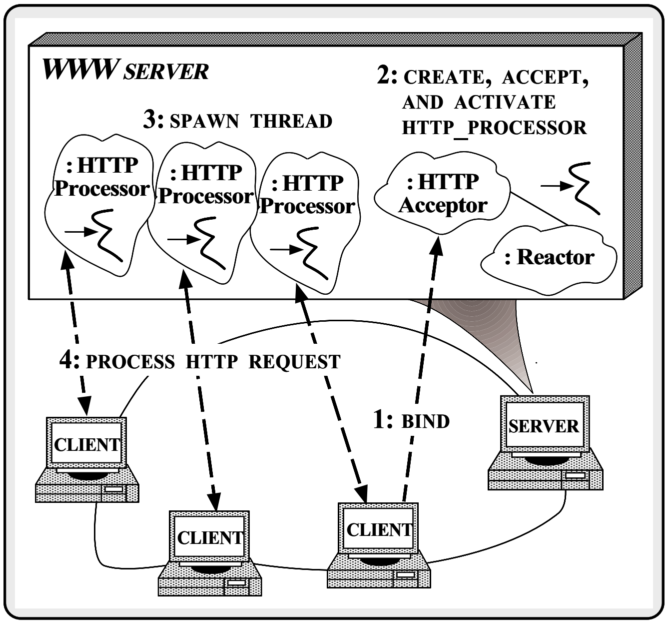
Single-threaded WWW Server Architecture



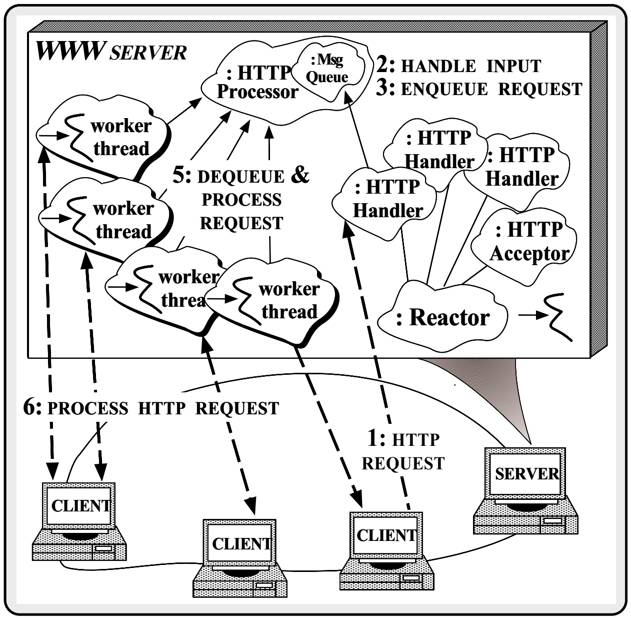
Thread-per-Request WWW Server Architecture



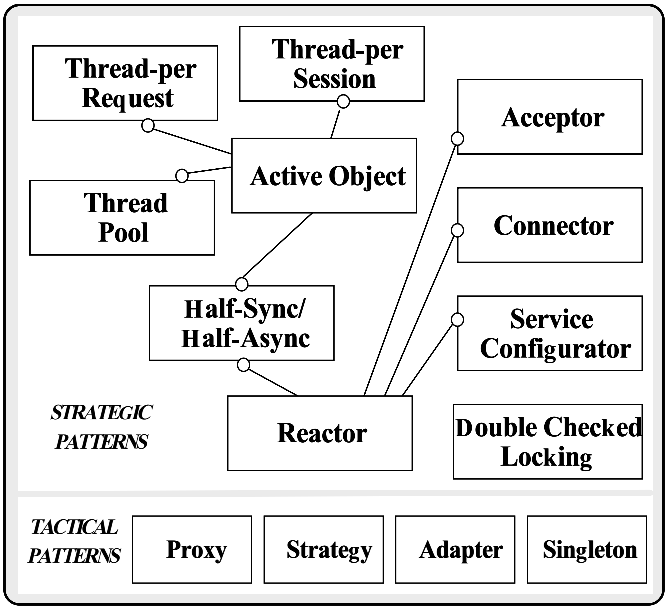
Thread-per-Session WWW Server Architecture



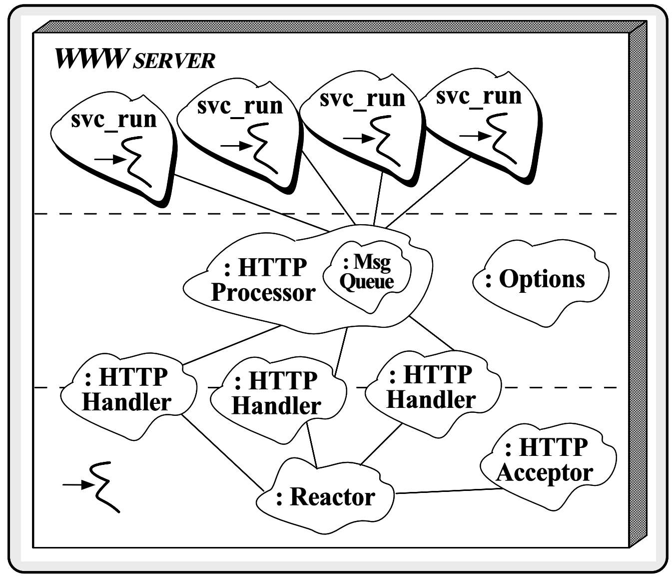
Thread-Pool WWW Server Architecture



Design Patterns in the WWW Client/Server



Architecture of the WWW Server

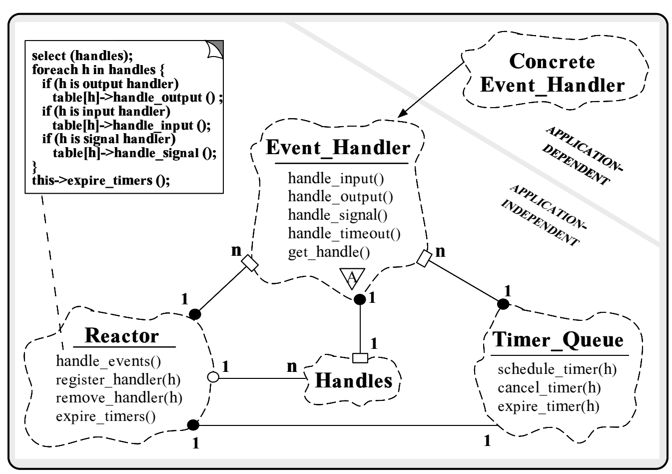


The Reactor Pattern

- *Intent*
  - "Decouples event demultiplexing and event handler dispatching from the services performed in response to events"

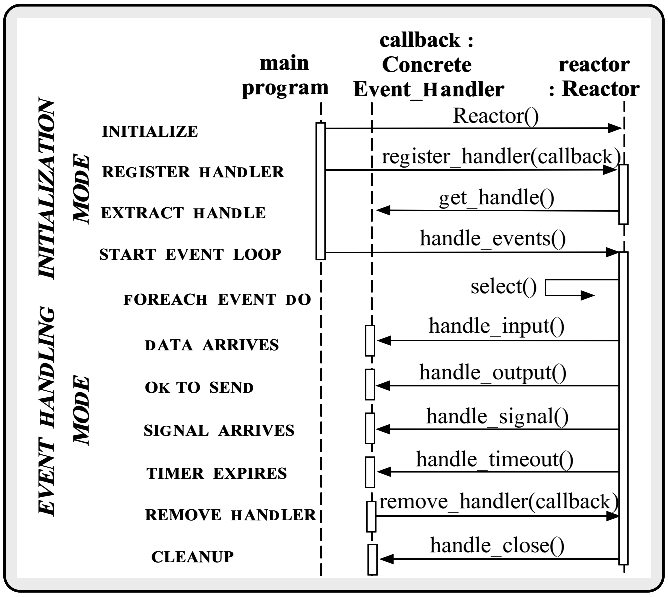
- This pattern resolves the following forces for event-driven software:
  - *How to demultiplex multiple types of events from multiple sources of events efficiently within a single thread of control*
  - *How to extend application behavior without requiring changes to the event dispatching framework*

Structure of the Reactor Pattern

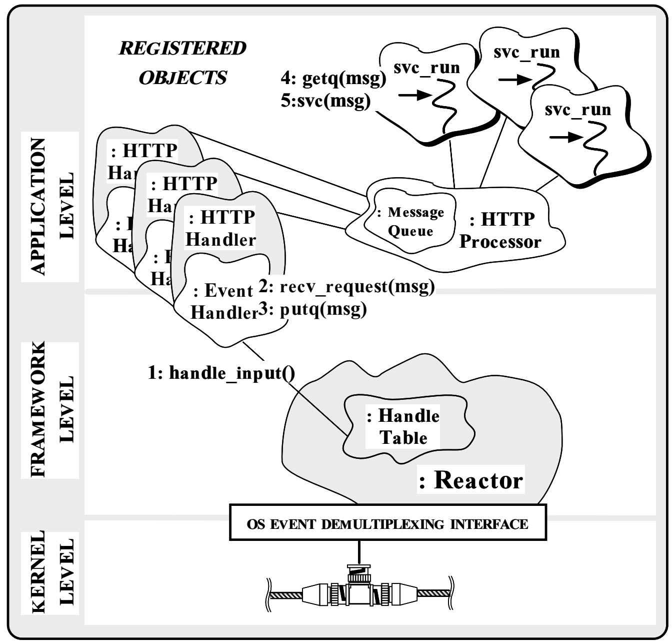


- Participants in the Reactor pattern

Collaboration in the Reactor Pattern



Using the Reactor in the WWW Server



The HTTP_Handler Public Interface

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

```
// Reusable base class.
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.
    Service_Config::reactor ()->register_handler
      (this, READ_MASK);

    // Register timeout in case client doesn't
    // send any HTTP requests.
    Service_Config::reactor ()->schedule_timer
      (this, 0, Time_Value (HTTP_CLIENT_TIMEOUT));
  }
```


The HTTP_Handler Protected Interface

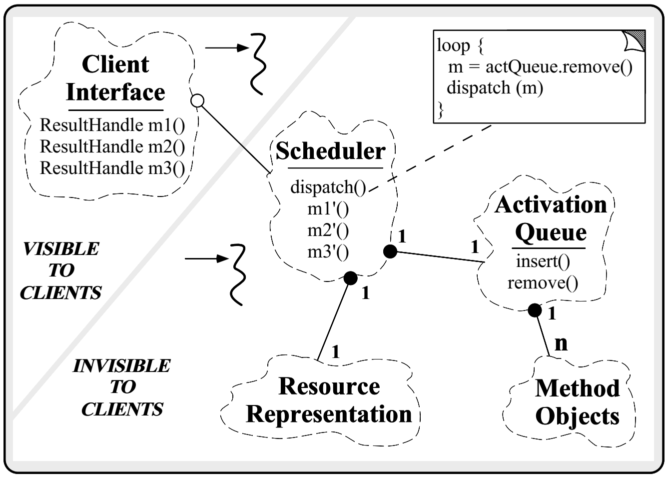
- The following methods are invoked by callbacks from the `Reactor`

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

The Active Object Pattern

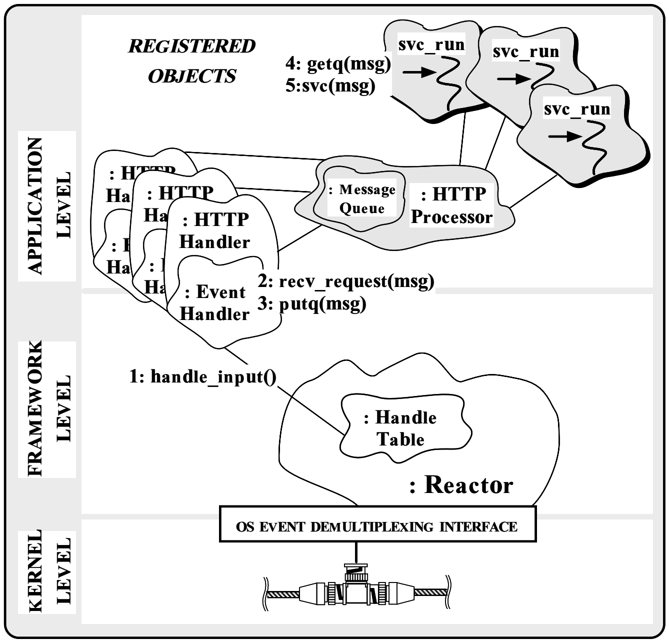
- *Intent*
  - "Decouples method execution from method invocation and simplifies synchronized access to shared resources by concurrent threads"
- This pattern resolves the following forces for concurrent communication software:
  - *How to allow blocking operations (such as read and write) to execute concurrently*
  - *How to serialize concurrent access to shared object state*
  - *How to simplify composition of independent services*

Structure of the Active Object Pattern

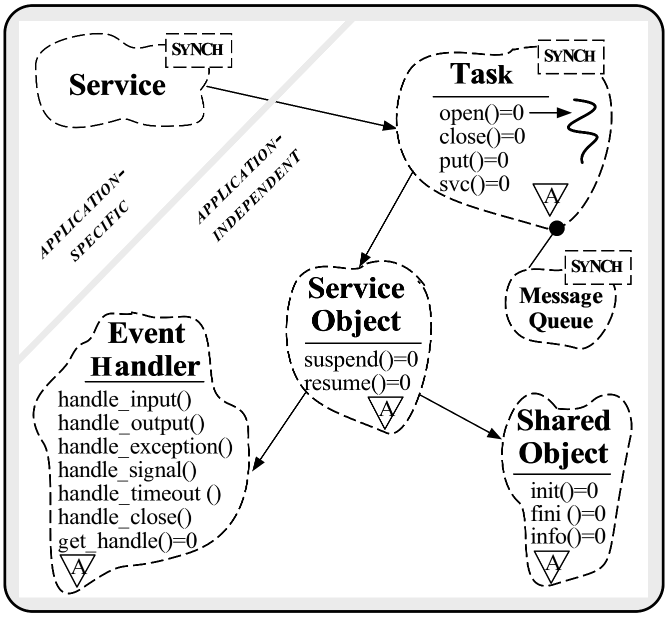


- *Intent*: decouples the thread of method execution from the thread of method invocation

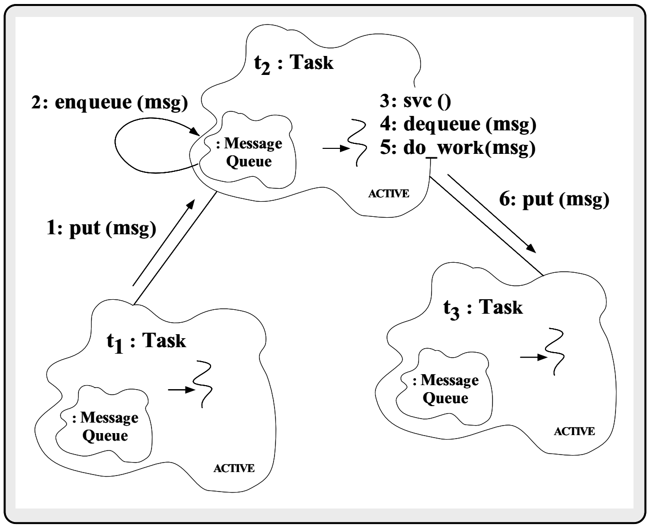
Using the Active Object Pattern in the WWW Server



Implementing the Active Object Pattern in ACE



Collaboration in ACE Active Objects



ACE 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"

```
template <class SYNCH>
class Task : public Service_Object
{
public:
  Task (Thread_Manager * = 0, Message_Queue<SYNCH> * = 0);
    // Initialization/termination routines.
  virtual int open (void *args = 0) = 0;
  virtual int close (u_long flags = 0) = 0;

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

    // Run by a daemon thread for deferred processing.
  virtual int svc (void) = 0;

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


Task Class Public Interface (cont'd)

- The following methods are mostly used within the `put` and `svc` hooks

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

  // 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> *);
```


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:

```
template <class SYNCH> int
Task<SYNCH>::activate (long flags, int n_threads)
{
  if (thr_mgr () == NULL)
    thr_mgr (Service_Config::thr_mgr ());

  thr_mgr ()->spawn_n
    (n_threads, &Task<SYNCH>::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

```
template <class SYNCH> void *
Task<SYNCH>::svc_run (Task<SYNCH> *t)
{
  Thread_Control tc (t->thr_mgr ()); // Record thread ID.

  // 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 incompabitilities between different OS thread libraries

    - ▷ It is integrated into ACE via the `Task::activate` method

The HTTP_Processor Class

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

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

    // Pass a request to the thread pool.
  virtual put (Message_Block *, Time_Value *);

    // Entry point into a pool thread.
  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

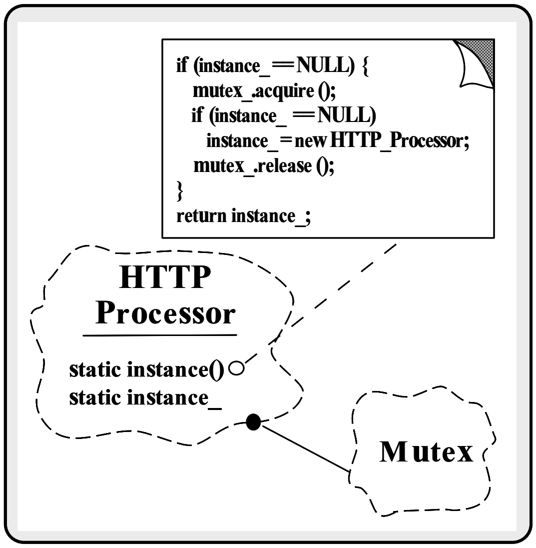
- The `HTTP_Processor` is implemented as a Singleton that is created "on demand"

```
// Singleton access point.

HTTP_Processor *
HTTP_Processor::instance (void)
{
  // Beware of race conditions!
  if (instance_ == 0) {
    instance_ = new HTTP_Processor;
  }
  return instance_;
}

// Constructor creates the thread pool.

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

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...

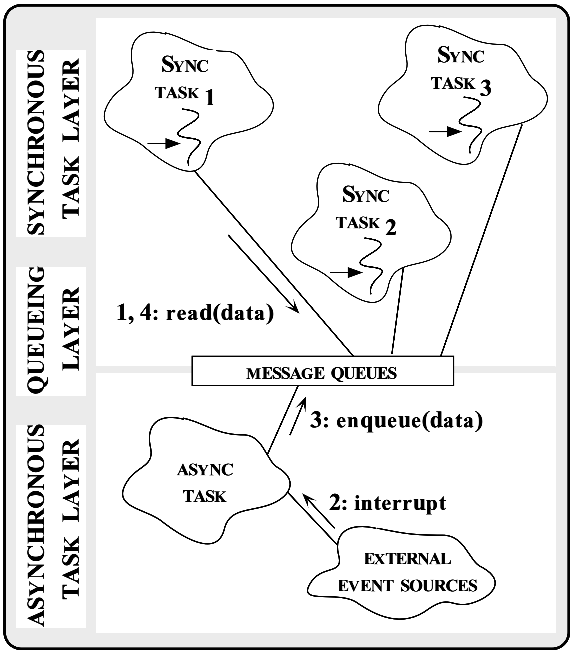
Using the Double-Checked Locking Optimization Pattern for the WWW Server



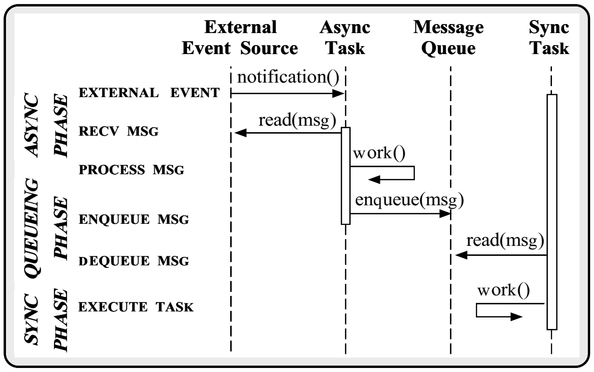
Half-Sync/Half-Async Pattern

- *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

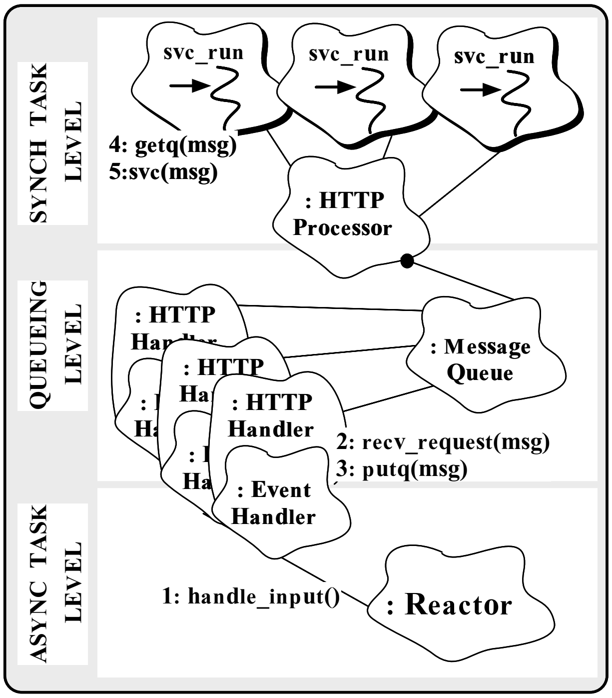


Collaboration in the Half-Sync/Half-Async Pattern



- 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

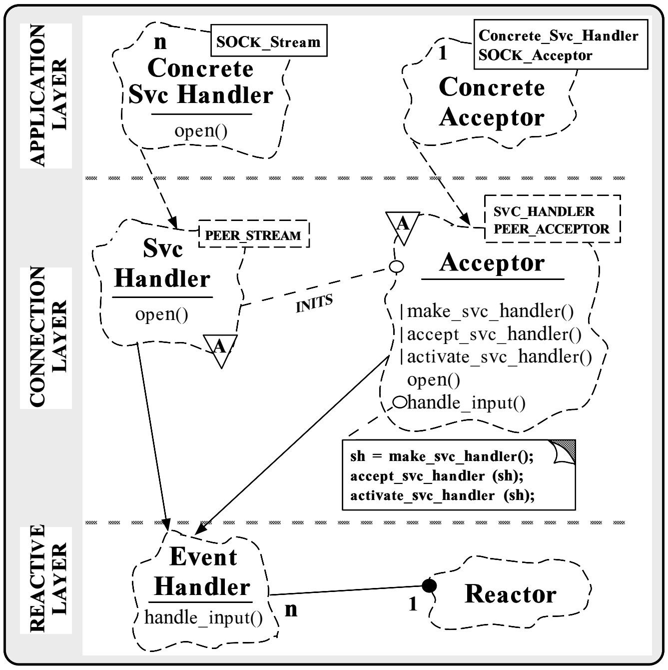
```
template <class PEER_ACCEPTOR> int
HTTP_Handler<PEER_ACCEPTOR>::handle_input (HANDLE h)
{
  Message_Block *mb = 0;

  // Try to receive and frame message.
  if (recv_request (mb) == HTTP_REQUEST_COMPLETE) {
    Service_Config::reactor ()->remove_handler
      (this, READ_MASK);
    Service_Config::reactor ()->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);
}
```


The Acceptor Pattern

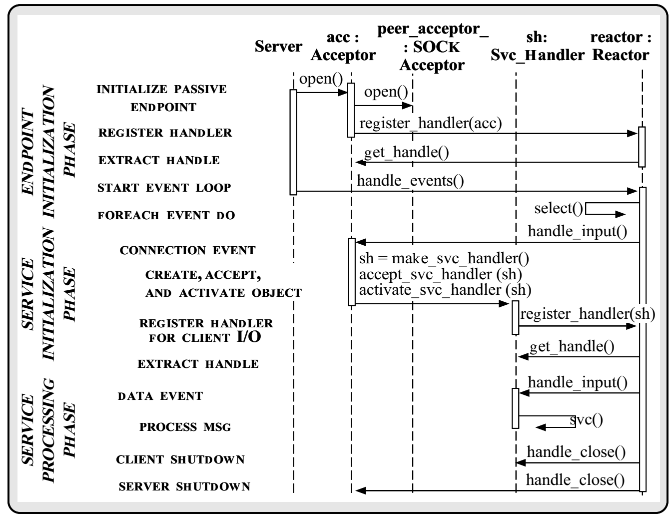
- *Intent*
  - "Decouple the passive initialization of a service from the tasks performed once the service is initialized"
- This pattern resolves the following forces for network servers using interfaces like sockets or TLI:

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 descriptor is not accidentally used to read or write data*

Structure of the Acceptor Pattern

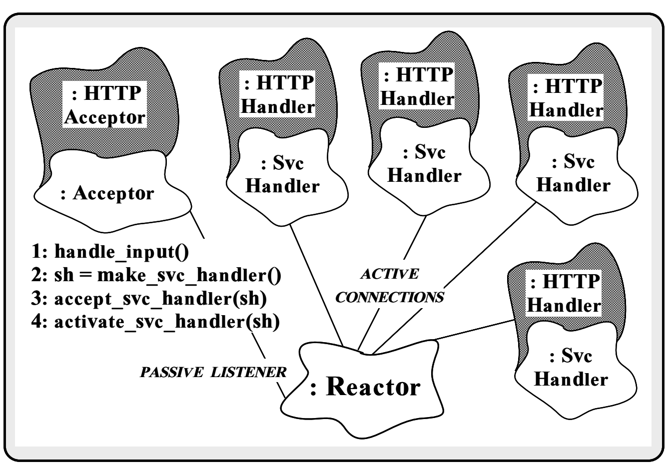


Collaboration in the Acceptor Pattern



- Acceptor factory creates, connects, and activates a Svc_Handler

Using the Acceptor Pattern in the WWW Server



The HTTP_Acceptor Class Interface

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

```
template <class PEER_ACCEPTOR>
class HTTP_Acceptor : public
                            // This is a ‘‘trait.’’
  Acceptor<HTTP_Handler<PEER_ACCEPTOR::PEER_STREAM>,
           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);

  // ...
};
```

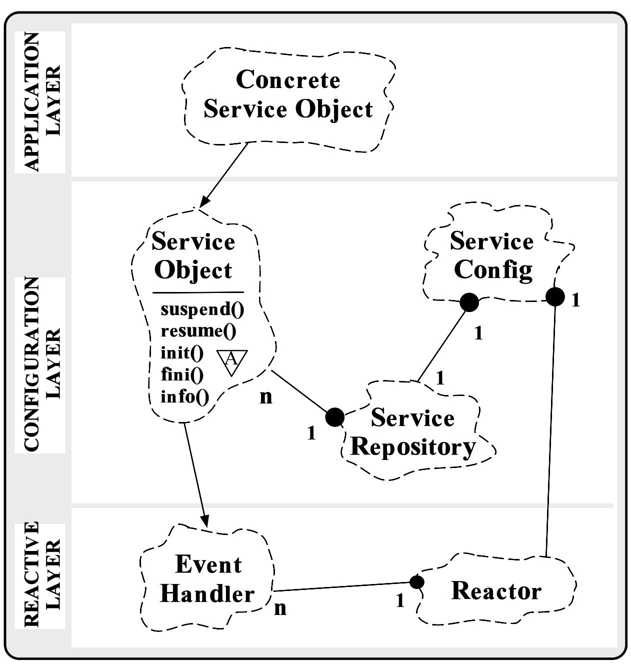
The HTTP_Acceptor Class Implementation

```
// Initialize service when dynamically linked.

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

  // Initialize the communication endpoint.
  peer_acceptor ().open
    (PA::PEER_ADDR (Options::instance ()->port ()));

  // Register to accept connections.
  Service_Config::reactor ()->register_handler
    (this, READ_MASK);
}

// Terminate service when dynamically unlinked.

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 ();
}
```

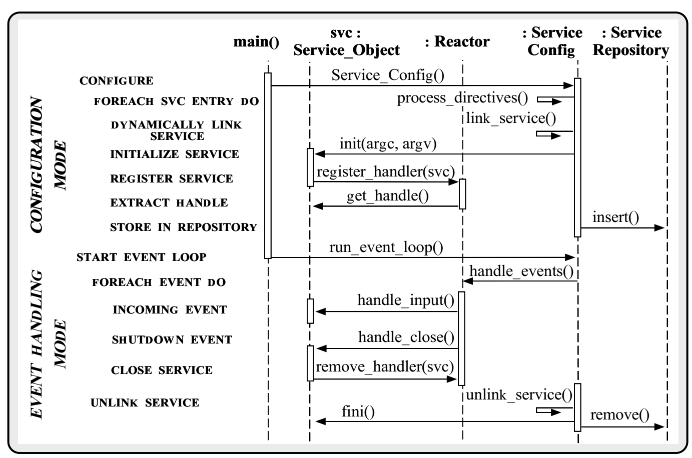

The Service Configurator Pattern

- *Intent*
  - "Decouples the behavior of network services from the point in time at which these services are configured into an application"
- This pattern resolves the following forces for network daemons:
  - *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 composing multiple independently developed services*
  - *How to reconfigure and control the behavior of the service at run-time*

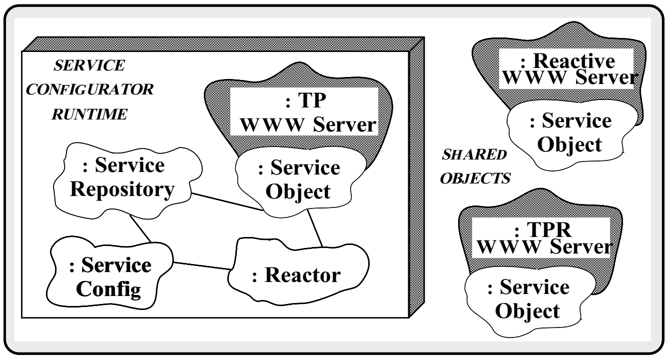
Structure of the Service Configurator Pattern



Collaboration in the Service Configurator Pattern



Using the Service Configurator Pattern 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

Service Configurator Implementation in C++

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

```
% cat ./svc.conf
dynamic TP_WWW_Server Service_Object *
        www_server.dll:make_TP_WWW_Server()
        "-p $PORT -t $THREADS"
```

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

```
extern "C" Service_Object *make_TP_WWW_Server (void);

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


Parameterizing IPC Mechanisms with C++ Templates

- To switch between a socket-based service and a TLI-based service, simply instantiate with a different C++ wrapper

```
// Determine the communication mechanisms.

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

Service_Object *make_TP_WWW_Server (void)
{
  return new HTTP_Acceptor<PEER_ACCEPTOR>;
}
```


Main Program for WWW Server

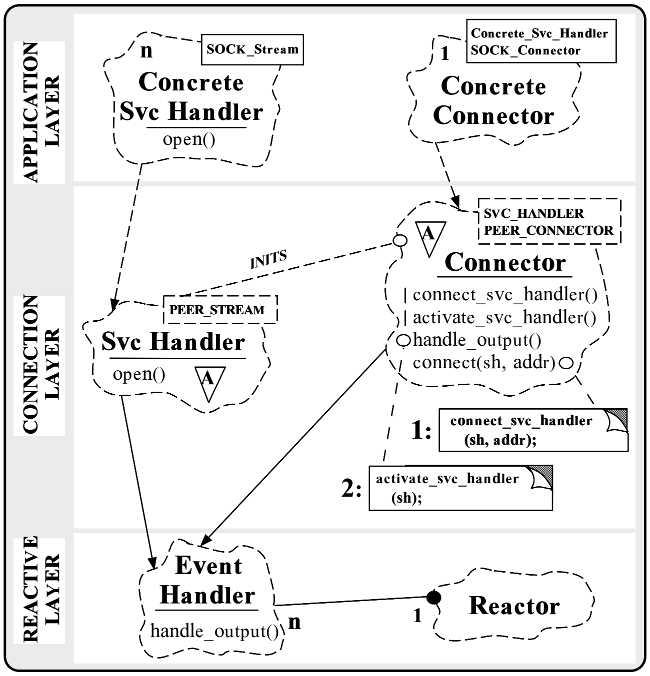
- Dynamically configure and execute the WWW Server
  - Note that this is totally generic!

```
int main (int argc, char *argv[])
{
  Service_Config daemon;

  // Initialize the daemon and dynamically
  // configure the service.
  daemon.open (argc, argv);

  // Loop forever, running services and handling
  // reconfigurations.

  daemon.run_reactor_event_loop ();
  /* NOTREACHED */
}
```

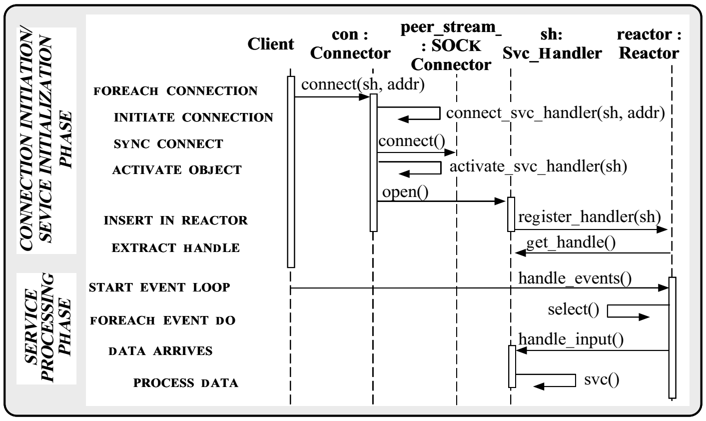
The Connector Pattern

- *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

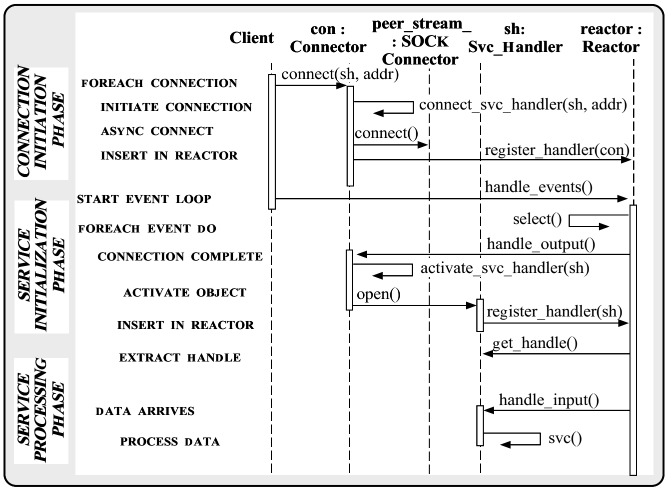


Collaboration in the Connector Pattern



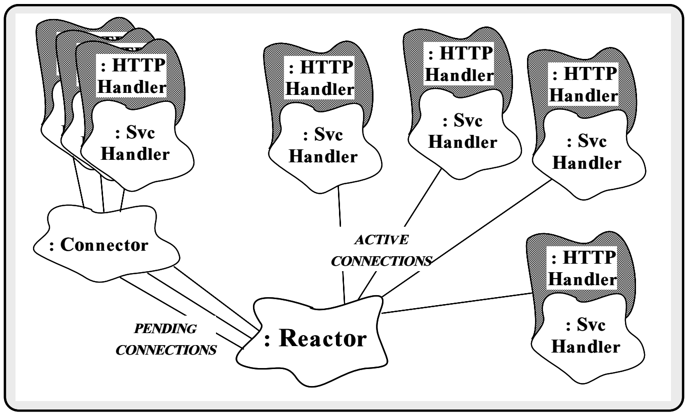
- Synchronous mode

Collaboration in the Connector Pattern



- Asynchronous mode

Using the Connector Pattern in a WWW Client



- *e.g.*, in the Netscape HTML parser

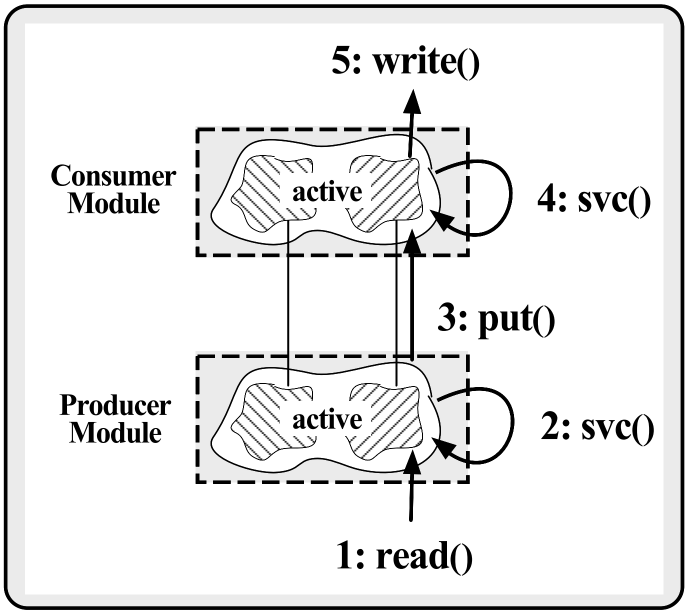
ACE Stream Example: Parallel I/O Copy

- Illustrates an implementation of the classic "bounded buffer" problem
  - The program copies stdin to stdout via the use of a multi-threaded `Stream`
- A `Stream` is an implementation of the "Layer Service Composition" pattern
  - The intent is to allow flexible configuration of layered processing modules

Implementing a Stream in ACE

- A `Stream` contains a stack of `Modules`
- Each `Module` contains two `Tasks`
  - In this example, the "read" `Task` is always ignored since the data flow is uni-directional
- Each `Task` contains a `Message_Queue` and a pointer to a `Thread_Manager`

Producer and Consumer Object Interactions



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



```
// Run in a separate thread.

int
Producer::svc (void)
{
  for (int n; ; ) {
    // Allocate a new message.
    Message_Block *mb = new Message_Block (BUFSIZ);

    // Keep reading stdin, until we reach EOF.

    if ((n = read (0, mb->rd_ptr (), mb->size ())) <= 0)
    {
      // Send a shutdown message to other thread and exit.
      mb->length (0);
      this->put_next (mb);
      break;
    }
    else
    {
      mb->wr_ptr (n); // Adjust write pointer.

      // Send the message to the other thread.
      this->put_next (mb);
    }
  }
  return 0;
}
```


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);
};
```



```
// The consumer dequeues a message from the Message_Queue,
// writes the message to the stderr stream, and deletes
// the message.  The Consumer sends a 0-sized message to
// inform the consumer to stop reading and exit.

int
Consumer::svc (void)
{
  Message_Block *mb = 0;

  // Keep looping, reading a message out of the queue,
  // until we get a message with a length == 0,
  // which informs us to quit.

  for (;;)
    {
      int result = getq (mb);

      if (result == -1) break;
      int length = mb->length ();

      if (length > 0)
        write (1, mb->rd_ptr (), length);

      delete mb;

      if (length == 0) break;
    }
  return 0;
}
```

Main Driver Function

- *e.g.*,

```
int main (int argc, char *argv[])
{
  // Control hierachically-related active objects.
  MT_Stream stream;

  // Create Producer and Consumer Modules and push
  // them onto the Stream.  All processing is then
 // performed in the Stream.

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

  // Barrier synchronization: wait for the threads,
  // to exit, then exit ourselves.
  Service_Config::thr_mgr ()->wait ();
  return 0;
}
```

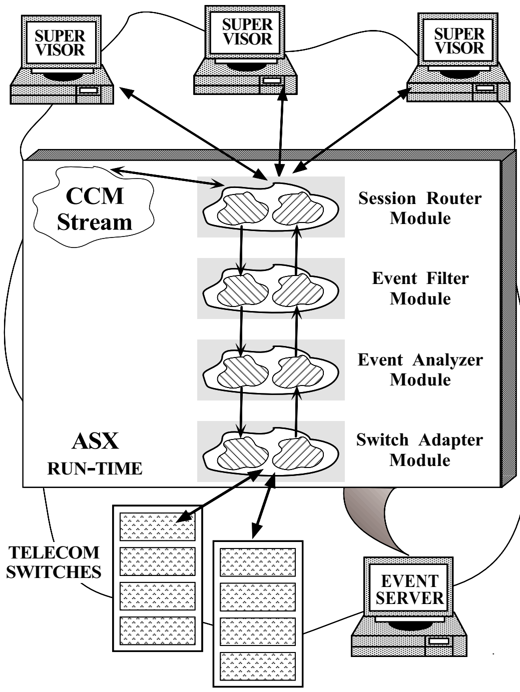

Evaluation of the Stream Class Category

- 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));
```


Call Center Manager Example



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"
- A passive object contains synchonization mechanisms that allow multiple method invocations to execute concurrently
  - 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 `Message_Queue` 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

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...

Avoiding Deadlock in OO Frameworks

- Deadlock can occur due to properties of OO frameworks, *e.g.*,
  - *Callbacks*
  - *Intra-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

Recursive Mutex

- Not all thread libraries support recursive mutexes
  - Here is portable implementation available in ACE:

```
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_;
};
```

```
// Acquire a recursive mutex (increments the nesting
// level and don't deadlock if owner of the mutex calls
// this method more than once).

Recursive_Thread_Mutex::acquire (void) const
{
  thread_t t_id = Thread::self ();

  Guard<Thread_Mutex> mon (nesting_mutex_);

  // If there's no contention, grab mutex.
  if (nesting_level_ == 0) {
    owner_id_ = t_id;
    nesting_level_ = 1;
  } else if (t_id == owner_id_)
    // If we already own the mutex, then
    // increment nesting level and proceed.
    nesting_level_++;
  else {
    // Wait until nesting level drops
    // to zero, then acquire the mutex.
    while (nesting_level_ > 0)
      mutex_available_.wait ();

    // Note that at this point
    // the nesting_mutex_ is held...

    owner_id_ = t_id;
    nesting_level_ = 1;
  }
  return 0;
```



```
// Releases a recursive mutex.

Recursive_Thread_Mutex::release (void) const
{
  thread_t t_id = Thread::self ();

  // Automatically acquire mutex.
  Guard<Thread_Mutex> mon (nesting_mutex_);

  nesting_level_--;

  if (nesting_level_ == 0) {
    // This may not be strictly necessary, but
    // it does put the mutex into a known state...
    owner_id_ = OS::NULL_thread;

    // Inform waiters that the mutex is free.
    mutex_available_.signal ();
  }
  return 0;
}

Recursive_Thread_Mutex::Recursive_Thread_Mutex (void)
  : nesting_level_ (0),
    owner_id_ (OS::NULL_thread),
    mutex_available_ (nesting_mutex_)
{
}
```


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 Design 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 flow of control for reactive dispatching is 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

Obtaining ACE

- The ADAPTIVE Communication Environment (ACE) is an OO toolkit designed according to key network programming patterns
- All source code for ACE is freely available
  - Anonymously ftp to `wuarchive.wustl.edu`
  - Transfer the files `/languages/c++/ACE/*.gz`
- Mailing lists

  * ace-users@cs.wustl.edu
  * ace-users-request@cs.wustl.edu
  * ace-announce@cs.wustl.edu
  * ace-announce-request@cs.wustl.edu
- WWW URL
  - http://www.cs.wustl.edu/~schmidt/ACE.html

Patterns Literature

- *Books*
  - Gamma et al., "Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Wesley, 1994
  - *Pattern Languages of Program Design* series by Addison-Wesley, 1995 and 1996
  - Siemens, *Pattern-Oriented Software Architecture*, Wiley and Sons, 1996
- *Special Issues in Journals*
  - Dec. '96 "Theory and Practice of Object Systems" (guest editor: Stephen P. Berczuk)
  - October '96 "Communications of the ACM" (guest editors: Douglas C. Schmidt, Ralph Johnson, and Mohamed Fayad)
- *Magazines*
  - C++ Report and Journal of Object-Oriented Programming, columns by Coplien, Vlissides, and Martin