Techniques for Object-Oriented Network Programming with C++

Douglas C. Schmidt

schmidt@cs.wustl.edu

Washington University, St. Louis

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

Motivation

- 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

- Developing *efficient*, *robust*, and *extensible* distributed applications is challenging
  - *e.g.*, must address complex topics that are less problematic or not relevant for non-distributed applications
- Object-oriented (OO) techniques and language features enhance distributed software quality factors
  - Key OO techniques → *design patterns* and *frameworks*
  - Key OO language features → *classes*, *inheritance*, *dynamic binding*, and *parameterized types*
  - Key software quality factors → *modularity*, *extensibility*, *portability*, *reusability*, and *correctness*

Tutorial Outline

- Outline key challenges for developing distributed applications
- Present a concurrent distributed application from the domain of enterprise medical imaging
- Compare and contrast an *algorithmic* and an *Object-Oriented* design and implementation of the application

Software Development Environment

- Note, the topics discussed here are largely independent of OS, network, and programming language
  - They are 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

Sources of Complexity

- Distributed application development exhibits both *inherent* and *accidental* complexity

- Examples of *Inherent* complexity
  - Addressing the impact of latency
  - Detecting and recovering from partial failures of networks and hosts
  - Load balancing and service partitioning

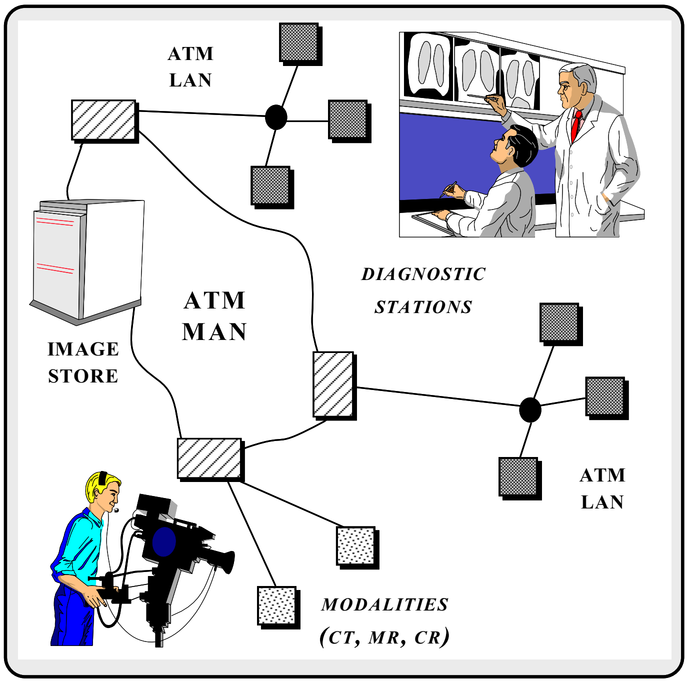
- Examples of *Accidental* complexity
  - Lack of type-secure, portable, re-entrant, and extensible system call interfaces and component libraries
  - Wide-spread use of *algorithmic* decomposition

Concurrent Network Server Example

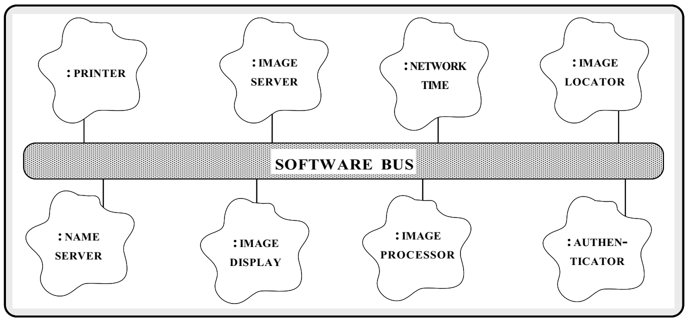
- The following example illustrates a concurrent OO architecture for medical Image Servers in an enterprise distributed health care delivery system

- Key system requirements are to support:
  1. Seamless electronic access to radiology expertise from any point in the system
  2. Immediate on-line access to medical images via advanced diagnostic workstations attached to high-speed ATM networks
  3. Teleradiology and remote consultation capabilities over wide-area networks

Medical Imaging Topology



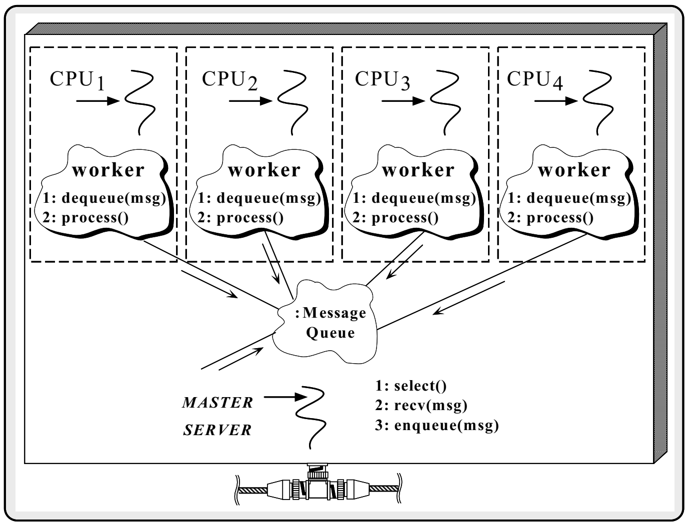
Concurrent Image Server Example



- Image Servers have the following responsibilities:

  * *Store/retrieve large medical images*
  * *Respond to queries from Image Locater Servers*
  * *Manage short-term and long-term image persistence*

Multi-threaded Image Server Architecture



- Worker threads execute within one process

Pseudo-code for Concurrent Image Server

- Pseudo-code for master server

  void master_server (**void**)
  {
      initialize listener endpoint and work queue
      spawn pool of worker threads
      foreach (*pending work request*) {
          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
  }

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 image request service.
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;

    // Identify and perform Image Server
    // request processing here...
  }
  return 0;
}
```

Master Server Driver Function

- The master driver function in the Image 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[]) {
  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_BOUND | THR_SUSPENDED, 0);

  // Initialize network device and recv work requests.
  recv_requests (msg_queue);

  // Resume all suspended threads (assumes contiguous id's)
  for (i = 0; i < NUM_THREADS; i++)
    thr_continue (t_id--);

  // 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(s)

    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 data into msg
            msg_queue.enqueue_tail (msg);
        }
    }
}
```


Limitations with the Image Server

- The algorithmic decomposition tightly couples application-specific *functionality* with various configuration-related characteristics, *e.g.*,
  - The image request handling service
  - The use of sockets and `select`
  - The number of services per process
  - The time when services are configured into a process
- There are *race conditions* in the code
- The solution is not portable since it hard-codes a dependency on SunOS 5.x threading mechanisms

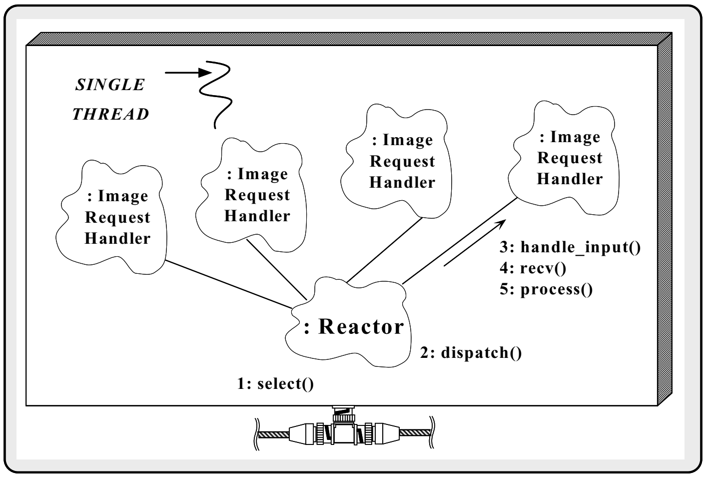
Eliminating Race Conditions in the Image Server

- The original Image Server uses a `Message_Queue` to queue `Message_Blocks`
  - The `worker` function running in each thread dequeues and processes these messages concurrently
- A naive implementation of `Message_Queue` will lead to race conditions
  - *e.g.*, when messages in different threads are enqueued and dequeued concurrently
- The solution described below requires the thread-safe ACE `Message_Queue` class

An OO Concurrent Image Server

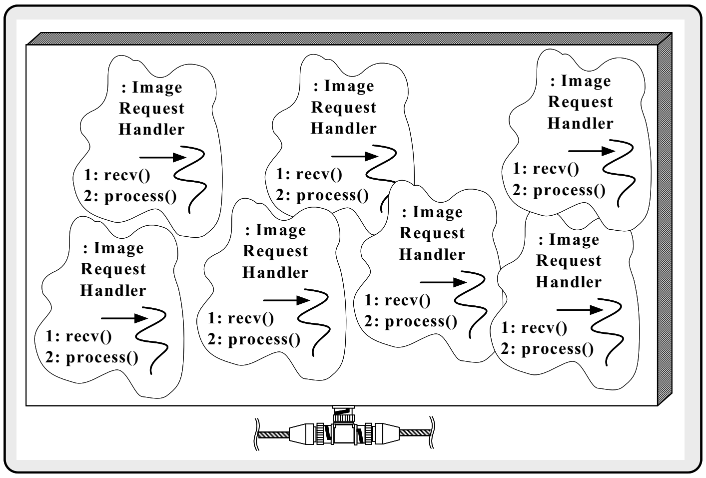
- The following example illustrates an OO solution to the concurrent Image Server
  - The active objects are based on the ACE `Task` class
- There are several ways to structure concurrency in an Image Server
  1. *Single-threaded, with all requests handled in one thread*
  2. *Multi-threaded, with all requests handled in separate threads*
  3. *Multi-threaded, with all requests handled by a thread pool*

(1) Single-threaded Image Server Architecture



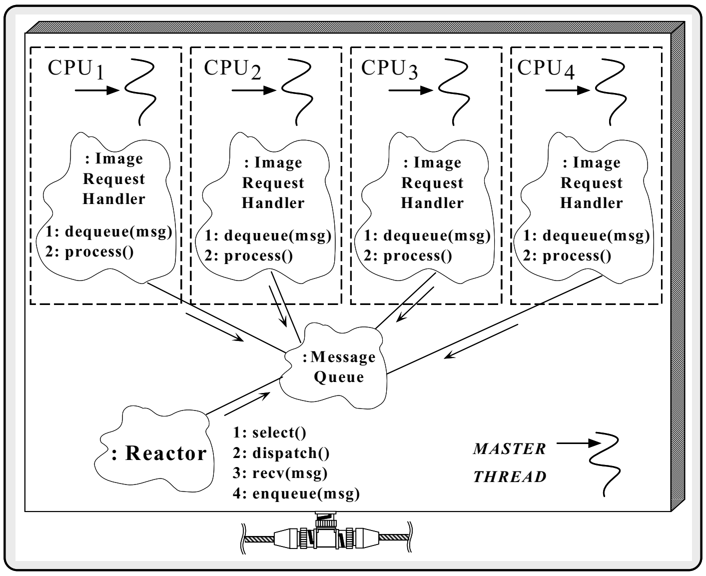
- Every handler processes one connection

(2) Multi-threaded Image Server Architecture



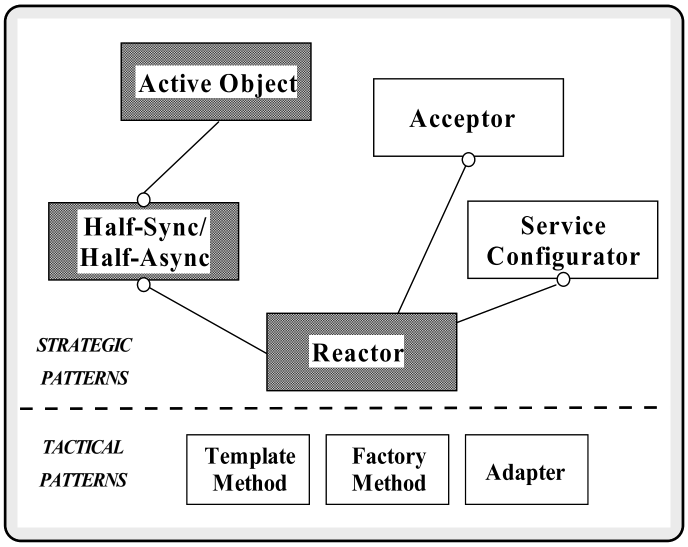
- Every handler processes one connection

(3) Multi-threaded Image Server Architecture



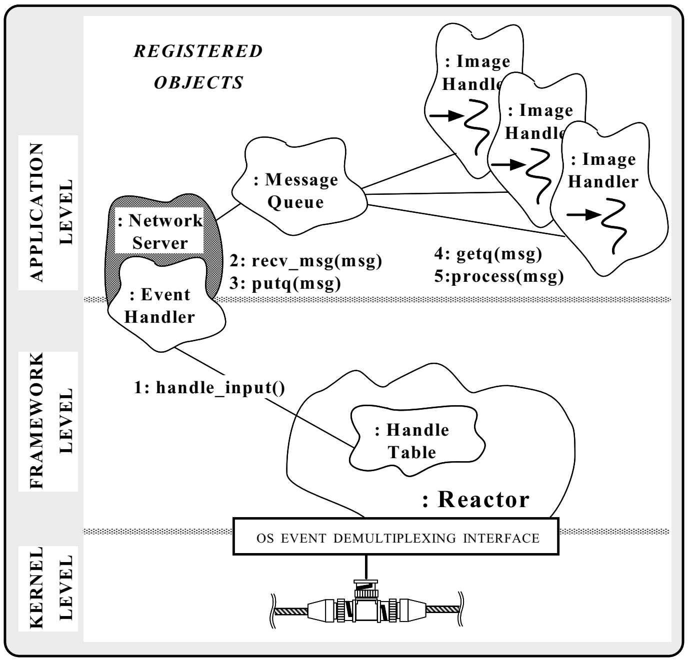
- Every handler processes one request

Design Patterns in the Image Server

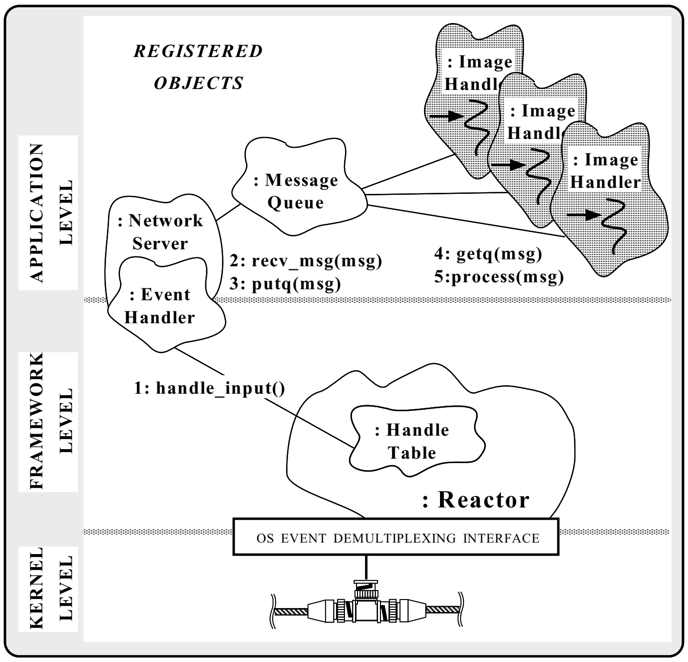


- The Image Server is based upon a system of design patterns

Using the Reactor for the Image Server



Using the Active Object Pattern for the Image Server



Using the Half-Sync/Half-Async Pattern for the Image Server

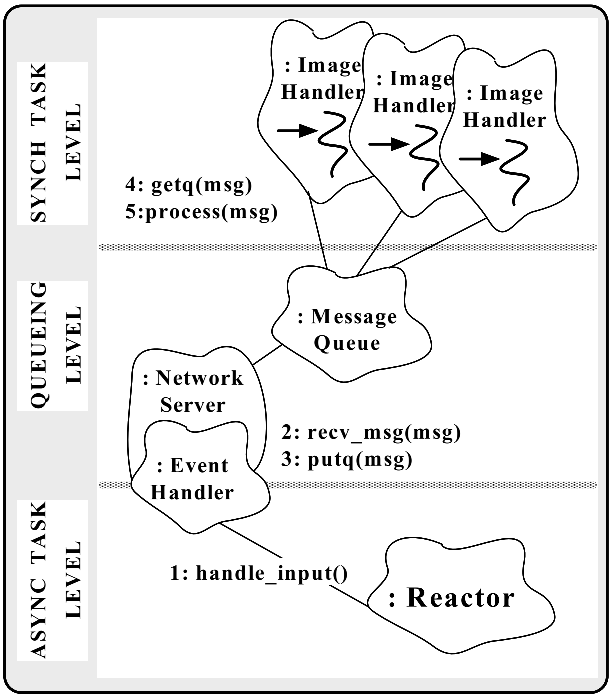


Image Server Public Interface

- The `Image_Server` class implements the service that processes image requests synchronously
  - To enhance reuse, the `Image_Server` is derived from a `Network_Server`

```
template <class PEER_ACCEPTOR> // Passive conn. factory
class Image_Server
  : public Network_Server<PEER_ACCEPTOR>
{
public:
    // Pass a message to the active object.
  virtual put (Message_Block *, Time_Value *);

    // Concurrent entry point into server thread.
  virtual int svc (int);
};
```


Network Server Public Interface

- `Network_Server` implements the asynchronous tasks in the Half-Sync/Half-Async pattern

```
// Reusable base class.
template <class PEER_ACCEPTOR> // Passive conn. factory
class Network_Server : public Task<MT_SYNCH>
{
public:
    // Dynamic linking hooks.
  virtual int init (int argc, char *argv);
  virtual int fini (void);

    // Pass a message to the active object.
  virtual put (Message_Block *, Time_Value *);

    // Accept connections and process from clients.
  virtual int handle_input (HANDLE);
```


Network Server Protected Interface

```
protected:
    // Parse the argc/argv arguments.
  int parse_args (int argc, char *argv[]);

    // Initialize network devices and connections.
  int init_endpoint (void);

    // Receive and frame an incoming message.
  int recv_message (PEER_ACCEPTOR::PEER_STREAM &,
                    Message_Block &*);

    // Acceptor factory for sockets.
  PEER_ACCEPTOR acceptor_;

    // Track # of requests.
  Atomic_Op<> request_count_;

    // # of threads.
  int num_threads_;

    // Listener port.
  u_short server_port_;
};
```


Network Server Implementation

```
// Short-hand definitions.
#define PEER_ACCEPTOR PA

// Initialize server when dynamically linked.

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

  thr_mgr_ = new Thread_Manager;

  // Create all the threads (start them suspended).
  thr_mgr_->spawn_n (num_threads_,
                         THR_FUNC (svc_run),
                         (void *) this,
                         THR_BOUND | THR_SUSPENDED);
  // Initialize communication endpoint.
  init_endpoint ();

  // Resume all suspended threads.
  thr_mgr_->resume_all ();
  return 0;
}
```

```
template <class PA> int
Network_Server<PA>::init_endpoint (void)
{
  // Open up the passive-mode server.
  acceptor_.open (server_port_);

  // Register this object with the Reactor.
  Service_Config::reactor()->register_handler
    (this, Event_Handler::READ_MASK);
}

// Called when service is dynamically unlinked.

template <class PA> int
Network_Server<PA>::fini (void)
{
  // Unblock threads.
  msg_queue_->deactivate ();

  // Wait for all threads to exit.
  thr_msg_->wait ();

  delete thr_msg_;
}
```



```
// Called back by Reactor when events arrive from clients.
// This method implements the asynchronous portion of the
// Half-Sync/Half-Async pattern...

template <class PA> int
Network_Server<PA>::handle_input (HANDLE h)
{
  PA::PEER_STREAM stream;

  // Handle connection events.
  if (h == acceptor_.get_handle ()) {
    acceptor_.accept (stream);
    Service_Config::reactor()->register_handler
      (stream.get_handle (), this, Event_Handler::READ_MASK);
  }

  // Handle data events asynchronously
  else {
    Message_Block *mb = 0;

    stream.set_handle (h);

    // Receive and frame the message.
    recv_message (stream, mb);

    // Insert message into the Queue (this call forms
    // the boundary between the Async and Sync layers).
    putq (mb);
  }
}
```



```
// Pass a message to the active object.
template <class PA> int
Image_Server<PA>::put (Message_Block *msg,
                       Time_Value *tv)
{
  putq (msg, tv);
}

// Concurrent entry point into the service.  This
// method implements the synchronous part of the
// Half-Sync/Half-Async pattern.
template <class PA> int
Image_Server<PA>::svc (void) {
  Message_Block *mb = 0; // Message buffer.

  // Wait for messages to arrive.
  while (getq (mb)) != -1) {
    // Keep track of number of requests.
    ++request_count_;

    // Identify and perform Image Server
    // request processing here...
  }
  return 0;
}
```


Eliminating Race Conditions (Part 1 of 2)

- There is a subtle and pernicious problem with the concurrent server illustrated above:
  - The auto-increment of global variable `request_count` is not serialized properly
- Lack of serialization will lead to race conditions 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

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;

template <class PA> int
Image_Server<PA>::svc (void) {
  // 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

- Define a C++ wrapper to address portability and elegance problems:

```
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 release (void) { return mutex_unlock (&lock_); }

private:
  mutex_t lock_; // SunOS 5.x serialization mechanism.
};
```

- 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 (&this->lock_);
  }
  ~Thread_Mutex (void) {
    DeleteCriticalSection (&this->lock_);
  }
  int acquire (void) {
    EnterCriticalSection (&this->lock_);
    return 0;
  }
  int release (void) {
    LeaveCriticalSection (&this->lock_);
    return 0;
  }

private:
    // Win32 serialization mechanism.
  CRITICAL_SECTION lock_;
};
```


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;

template <class PA> int
Image_Server<PA>::svc (void) {
  // ...
    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) { this->lock_.acquire (); }
  ~Guard (void) { this->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;

template <class PA> int
Image_Server<PA>::svc (void) {
  // ...
    {
      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 C++ features such as parameterized types and overloading

OO Design Interlude

- 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 to Adapter pattern to provide identical interfaces whenever possible to facilitate parameterization

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) { this->count_ = c; }
  TYPE operator++ (void) {
    Guard<LOCK> m (this->lock_); return ++this->count_;
  }
  void operator= (const Atomic_Op &ao) {
    if (this != &ao) {
      Guard<LOCK> m (this->lock_); this->count_ = ao.count_;
    }
  }
  operator TYPE () {
    Guard<LOCK> m (this->lock_);
    return this->count_;
  }
  // Other arithmetic operations omitted...
private:
  LOCK lock_;
  TYPE  count_;
};
```

Thread-safe Version of Concurrent Server

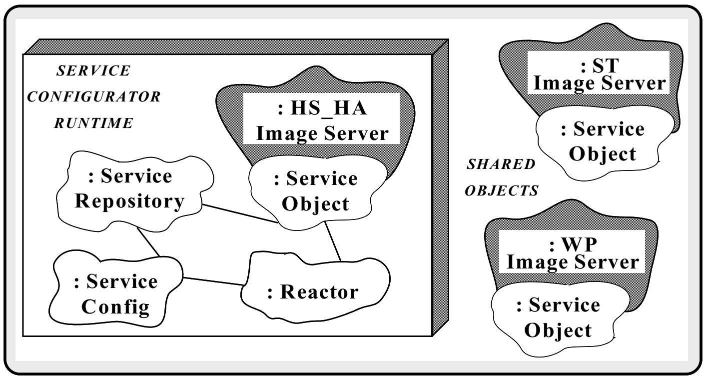
- Using the Atomic_Op class, only one change is made to the code

```
#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

```
template <class PA> int
Image_Server<PA>::svc (void) {
    //...
    // Calls Atomic_Op::operator++(void)
    ++request_count;
    //...
}
```


Using the Service Configurator Pattern in the Image Server



- Existing service is based on Half-Sync/Half-Async pattern, other versions could be single-threaded or use other concurrency strategies...

Image Server Configuration

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

```
% cat ./svc.conf
dynamic HS_HA_Image_Server Service_Object *
        /svcs/networkd.so:alloc_server() "-p 2112 -t 4"
```

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

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

Service_Object *alloc_server (void)
{
  return new Image_Server<SOCK_Acceptor>;
  // ASX 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

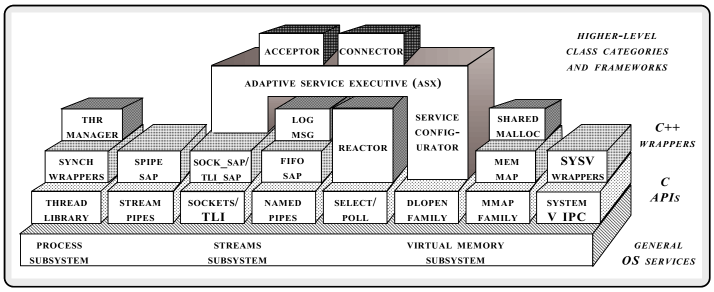
```
// Determine the communication mechanisms.

#if defined (ACE_USE_SOCKETS)
typedef SOCK_Stream PEER_STREAM;
typedef SOCK_Acceptor PEER_ACCEPTOR;
#elif defined (ACE_USE_TLI)
typedef TLI_Stream PEER_STREAM;
typedef TLI_Acceptor PEER_ACCEPTOR;
#endif

Service_Object *alloc_server (void)
{
  return new Image_Server<PEER_ACCEPTOR, PEER_STREAM>;
}
```

Main Program

- Dynamically configure and execute the network service

```
int main (int argc, char *argv[])
{
  // Initialize the daemon and
  // dynamically configure the service.

  Service_Config daemon (argc, argv);

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

  daemon.run_event_loop ();

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


The ADAPTIVE Communication Environment (ACE)



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

- C++ wrappers
  - *e.g.*, IPC_SAP, Synch, Mem_Map

- OO class categories and frameworks
  - *e.g.*, Reactor, Service Configurator, ADAPTIVE Service eXecutive (ASX)

Obtaining ACE

- The ADAPTIVE Communication Environment (ACE) is an OO toolkit designed according to key network programming patterns

- All source code for ACE is freely available
  - Anonymously ftp to `wuarchive.wustl.edu`
  - Transfer the files `/languages/c++/ACE/*.gz` and `gnu/ACE-documentation/*.gz`

- Mailing list
  - ace-users@cs.wustl.edu
  - ace-users-request@cs.wustl.edu

- WWW URL
  - http://www.cs.wustl.edu/~schmidt/