Implementing Multi-threaded CORBA Applications with Orbix and ACE

Douglas C. Schmidt

Washington University, St. Louis

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

schmidt@cs.wustl.edu

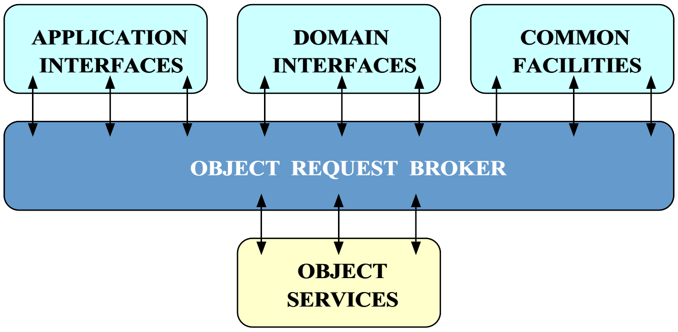
Outline

- Building multi-threaded distributed applications is hard
- To succeed, programmers must understand available tools, techniques, and patterns
- This tutorial examines how to build multi-threaded CORBA applications
  - Using Orbix and ACE
- It also presents several concurrency models
  1. *Thread-per-Request*
  2. *Thread Pool*
  3. *Thread-per-Object*

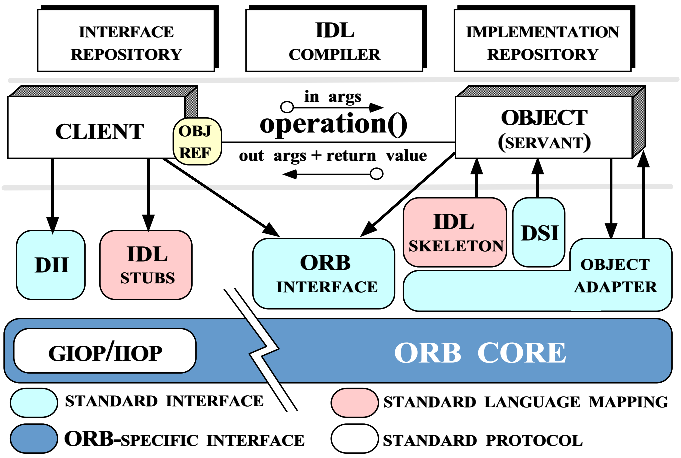
Overview of CORBA

- Simplifies application interworking
  - CORBA provides higher level integration than traditional "untyped TCP bytestreams"
- Provides a foundation for higher-level distributed object collaboration
  - *e.g.*, Windows OLE and the OMG Common Object Service Specification (COSS)
- Benefits for distributed programming similar to OO languages for non-distributed programming
  - *e.g.*, encapsulation, interface inheritance, and object-based exception handling

Overall CORBA Architecture



CORBA ORB Architecture



CORBA Quoter Example

- Ideally, to use a distributed service, we'd like it to look just like a non-distributed service:

```
int
main (int argc, char *argv[])
{
  // Use a factory to bind to any quoter.
  Quoter_var quoter = bind_quoter_service ();

  const char *stock_name = "ACME ORB Inc.";

  long value = quoter->get_quote (stock_name);
  cout << stock_name << " = " << value << endl;
}
```

- Unfortunately, life is harder when errors occur...

CORBA Quoter Interface

- We need to write a OMG IDL `interface` for our `Quoter` object

```
// Interface is similar to a C++ class.
interface Quoter
{
  exception Invalid_Stock {};

  long get_quote (in string stock_name)
    raises (Invalid_Stock);
};
```

- The use of OMG IDL promotes language independence, location transparency, and modularity

Motivation for Concurrency in CORBA

- Concurrent CORBA 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

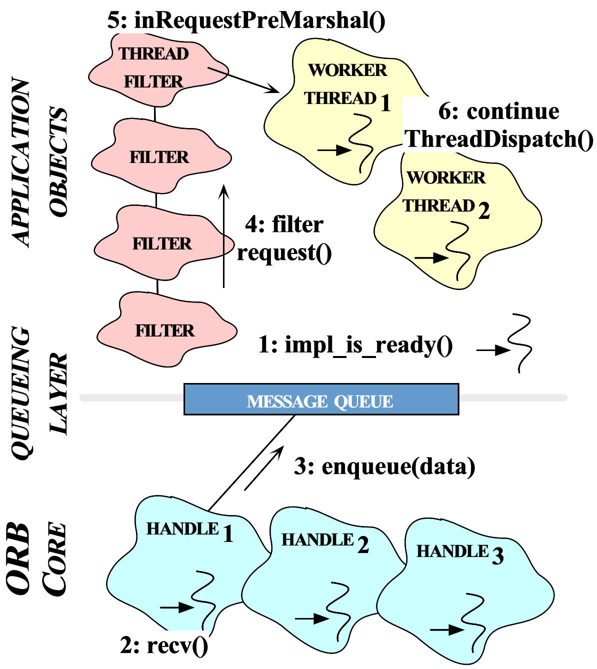
Overview of Orbix

- Orbix provides a thread-safe version of the standard Orbix CORBA libraries
- An application can choose to ignore threads and if it creates none, it need not be thread-safe
- Orbix is mostly backwardly compatible with non-threaded Orbix
  - Problems arise with event-loop integration mechanism
  - Performance is also an issue...

Threading in Orbix

- Orbix uses threads internally, *e.g.*,
  * Listening for new connections
  * For each underlying network connection
  * Cleaning up after the other threads (reaper)
- Applications can create threads using the native threads package
  - *e.g.*, Solaris threads, Windows NT threads, POSIX Pthreads, etc.
- Locking within the libraries is a compromise between restrictions on concurrent execution and unacceptably high overhead
  - Orbix doesn't automatically synchronize access to application objects
  - Therefore, applications must synchronize access to their own objects

Orbix Thread Filter Architecture



Overview of Thread Filters

- To increase flexibility, Orbix uses a "Thread Filter" concurrency architecture
  - Thread Filters are a non-standard extension that use the "Chain of Responsibility" pattern
- Thread Filters largely shield the ORB and Object Adapter from the choice of concurrency model
  - *i.e.*, decouples demultiplexing from dispatching
- Various concurrency models can be created by using Thread Filters
  - *e.g.*, Thread-per-request, Thread Pool, Thread-per-Object

Using Thread Filters

- To process CORBA requests concurrently, create a filter containing the `inRequestPreMarshal` method
  - Orbix call this method before the request is processed
  - `inRequestPreMarshal` should return −1 to tell Orbix it will process the request concurrently
- Threads can be spawned according to the desired concurrency model

Example Thread Filter Code

- A thread filter that dispatches incoming calls

```
class TPR_Filter : public CORBA::ThreadFilter
{
public:
  // Intercept request and spawn thread.
  virtual int inRequestPreMarshal (CORBA::Request &);

  // Execute the request in a separate thread.
  static void *worker_thread (void *);
};
```

- Implements the Thread-per-Request model

```
TPR_Filter::inRequestPreMarshal (CORBA::Request &req);
{
  // Use Solaris thread creation function.
  thr_create (0, 0, TPR_Filter::worker_thread,
              (void *) &req, THR_DETACHED, 0);

  // Tell Orbix we'll dispatch request later
  return -1;
}
```


Thread Entry Point

- Once a thread is created, the CORBA request is passed to `worker_thread`
  - This calls `CORBA::Orbix.continueThreadDispatch`, which continues dispatching the request
  - Thus, the request is processed in a new thread of control
- The `worker_thread` code might look like this:

```
// Entry point where the new thread begins..

void *TPR_Filter::worker_thread (void *arg)
{
  CORBA::Request *req =
    static_cast <CORBA::Request *> (arg);
  CORBA::Orbix.continueThreadDispatch (*req);
  return 0;
}
```


Creating a Thread Filter

- A filter that inherits from `CORBA::ThreadFilter` will automatically be placed at the end of the "per-process filter chain"

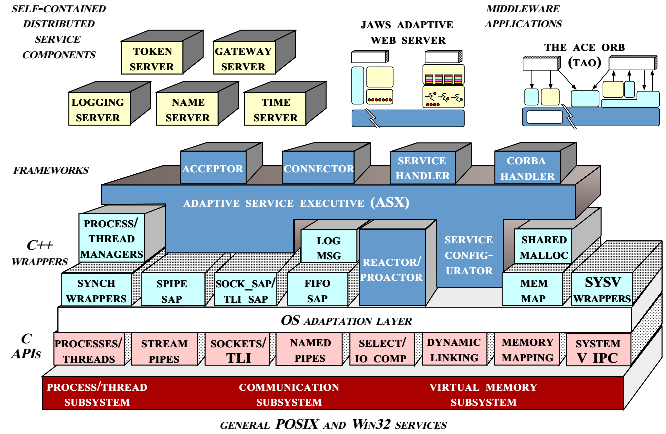
```
// Global object installs per-process Thread Filter.
TPR_Filter global_thread_dispatcher;
```

- Object creation causes filter insertion
- Note that there can only be a single per-process Thread Filter installed!

Overcoming Limitations with CORBA

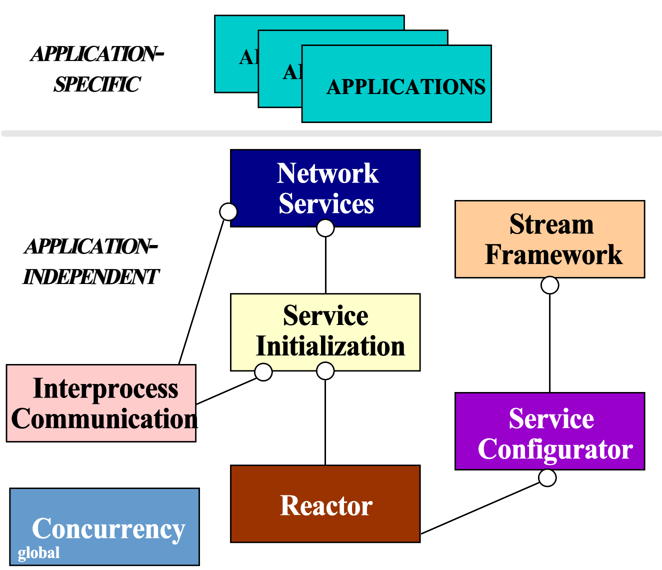
- *Problem*
  - CORBA primarily addresses "communication" topics
- *Forces*
  - Real world distributed applications need many other components
    - *e.g.*, concurrency control, layering, shared memory, event-loop integration, dynamic configuration, etc.
- *Solution*
  - Integrate CORBA with an OO communication framework

The ADAPTIVE Communication Environment (ACE)



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

Class Categories in ACE



Class Categories in ACE (cont'd)

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

Overview of ACE Concurrency

- ACE provides portable C++ threading and synchronization wrappers
- ACE classes we'll examine include:
  - *Thread Management*
    - `ACE_Thread_Manager` → encapsulates threads
  - *Synchronization*
    - `ACE_Thread_Mutex` and `ACE_RW_Mutex` → encapsulates mutexes
    - `ACE_Atomic_Op` → atomically perform arithmetic operations
    - `ACE_Guard` → automatically acquire/release locks
  - *Queueing*
    - `ACE_Message_Queue` → thread-safe message queue
    - `ACE_Message_Block` → enqueued/dequeued on message queues

Overview of ACE Concurrency (cont'd)

- Several `ACE_Thread_Manager` class methods are particularly interesting:
  - **spawn** → Create 1 new thread of control running **func**

```
int spawn (void *(*)(void *) func,
           void *arg,
           long flags,
           .....);
```

  - **spawn_n** → Create n new threads of control running **func**

```
int spawn_n (size_t n,
             void *(*)(void *) func,
             void *arg,
             long flags,
             ......);
```

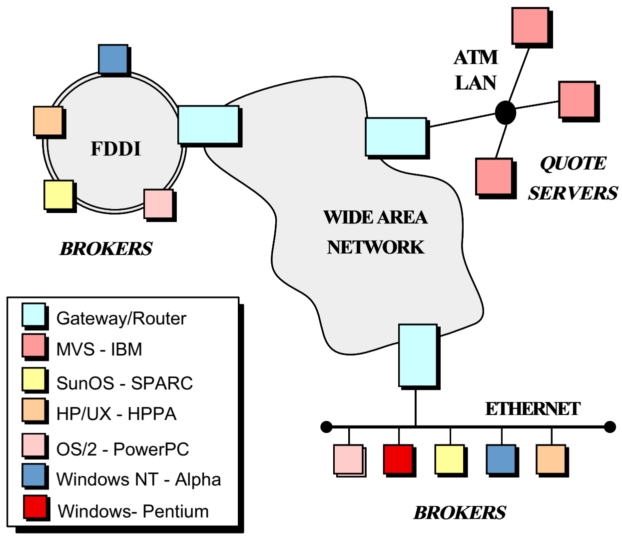
  - **wait** → Wait for all threads in a manager to terminate

```
int wait (void);
```


Orbix Examples

- Each example implements a concurrent CORBA stock quote service
  - Show how threads can be used on both the client and server side
- The server is implemented three different ways:
  1. *Thread-per-Request* → Every incoming CORBA request causes a new thread to be spawned to process it
  2. *Thread Pool* → A fixed number of threads are generated in the server at start-up to service all incoming requests
  3. *Thread-per-Object* → Each session created is assigned a thread to process requests for that session
- Note that clients are unaware which concurrency model is being used...

Stock Quoter Application



- Note the heterogeneity in this example

OMG IDL Definitions

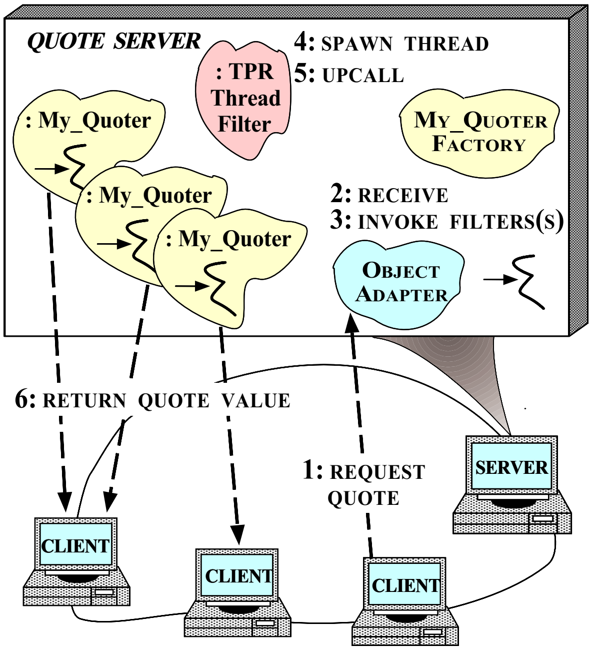
- The IDL definition is the same for all three server implementations:

```
// Define the interface for a stock quote server
module Stock
{
  exception Invalid_Stock {};

  // Interface is similar to a C++ class.
  interface Quoter : CosLifeCycle::LifeCycleObject {
    long get_quote (in string stock_name)
      raises (Invalid_Stock);
    // Inherits:
    // void remove () raises (NotRemovable);
  };

  // Manage the lifecycle of a Quoter object.
  interface Quoter_Factory : CosLifeCycle::GenericFactory {
    // Returns a new Quoter selected by name
    // e.g., "Dow Jones," "Reuters,", etc.
    // Inherits:
    // Object create_object (in Key k,
    //                       in Criteria criteria);
  };
};
```


Thread-per-Request Concurrency Architecture



Thread-per-Request Main Program

- In this scheme the server creates a single Quoter factory and waits in the Orbix event loop

```
typedef TIE_Quoter_Factory (My_Quoter_Factory)
        TIE_QUOTER_FACTORY;
class Quote_Database; // Singleton.

int main (void)
{
  try {
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, 0);
    ORB::BOA_var boa = orb->BOA_init (argc, argv, 0);

    // Create factory object.
    Quoter_Factory_var qf =
      new TIE_QUOTER_FACTORY (new My_Quoter_Factory);

    // Run CORBA event loop.
    boa->impl_is_ready ("Quoter_Factory");

    // Destructor of qf calls CORBA::release ();
  } catch (...) { /* handle exception ... */ }
}
```


Thread-per-Request Quoter Interface

- Implementation of the `Quoter` IDL interface

```
// Maintain count of requests.
typedef u_long COUNTER;

class My_Quoter
{
public:
  // Constructor.
  My_Quoter (const char *name);

  // Returns the current stock value.
  long get_quote (const char *stock_name);

private:
  // Maintain request count.
  static COUNTER req_count_;
};
```

Thread-per-Request Quoter Factory Interface

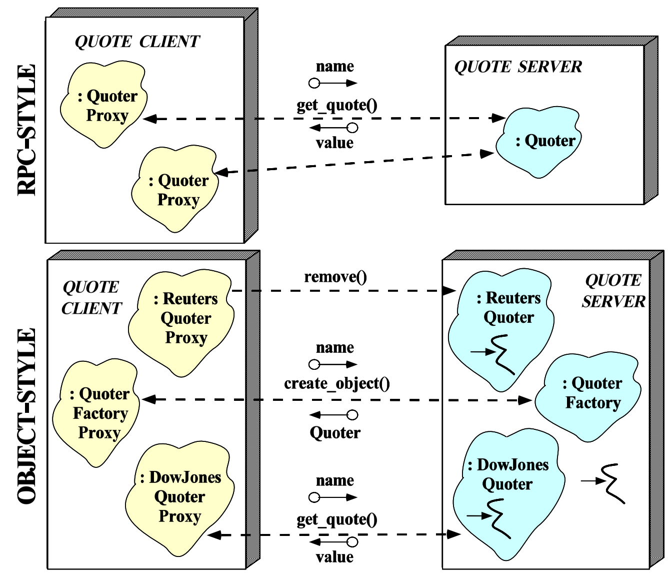
- Factory that manages a Quoter's lifecycle

```
class My_Quoter_Factory
{
public:
  // Constructor
  My_Quoter_Factory (void);

  // Factory method for creation.
  CORBA::Object_ptr create_object
    (const CosLifeCycle::Key &factory_key,
     const CosLifeCycle::Criteria &the_criteria);
};
```

- We use the Factory Method pattern to make creation more abstract

RPC-style vs. Object-style Communication



Thread-per-Request Filter Interface

- The thread filter spawns a new thread for each request

```
class TPR_Filter : public CORBA::ThreadFilter
{
  // Intercept request and spawn thread
  virtual int inRequestPreMarshal (CORBA::Request &);

  // Execute the request in a separate thread
  static void *worker_thread (void *);
};
```

- We'll use the TIE approach to associate the CORBA interfaces with our implementation

```
DEF_TIE_Quoter_Factory (My_Quoter_Factory)
DEF_TIE_Quoter (My_Quoter)
```

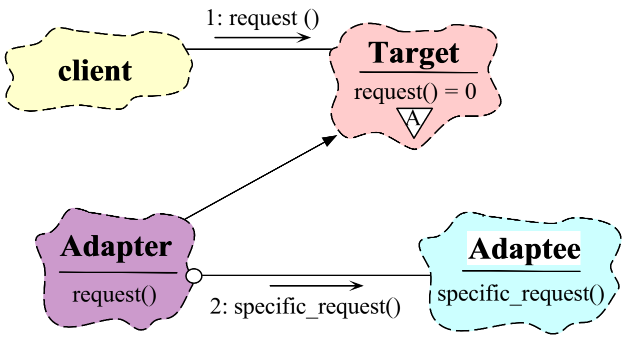

Associating Skeletons with Implementations

- *Problem*
  - How to associate the automatically generated IDL skeleton with our implementation code
- *Forces*
  - Inheritance is inflexible since it tightly couples the solution with the base classes
  - Virtual base classes are often incorrectly implemented and are potentially space inefficient
- *Solution*
  - Use the object form of the *Adapter* pattern

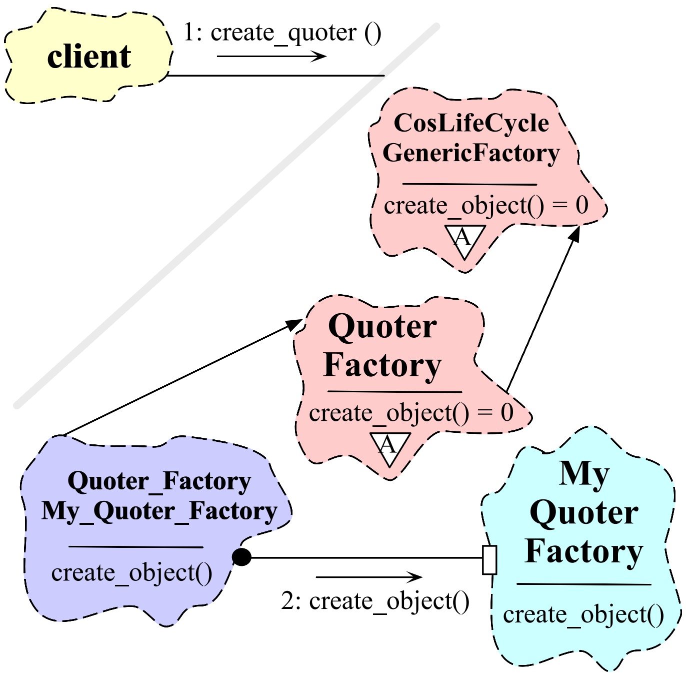
The Adapter Pattern

- *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 writing CORBA servers
  1. *How to associate the object implementation with the auto-generated IDL skeleton without requiring the use of inheritance*

Structure of the Object form of the Adapter Pattern



Using the Object form of the Adapter Pattern with Orbix TIE



Thread-per-Request Quoter Factory Implementation

- The factory controls the lifetime of a `Quoter`

```
typedef TIE_Quoter (My_Quoter) TIE_QUOTER;

CORBA::Object_ptr
My_Quoter_Factory::create_object
  (const CosLifeCycle::Key &factory_key,
   const CosLifeCycle::Criteria &the_criteria)
{
  My_Quoter_ptr q = new My_Quoter (factory_key.id);

  Quoter_ptr quoter = new TIE_QUOTER (q);

  // Be sure to duplicate the object reference!
  return quoter->_duplicate ();
}
```

Thread-per-Request Filter Implementation

- Every incoming request generates a new thread that runs "detached"
  - Detached threads terminate silently when the request is complete

```
int TPR_Filter::inRequestPreMarshal
    (CORBA::Request &req)
{
  ACE_Thread_Manager::instance ()->spawn
    (TPR_Filter::worker_thread, (void *) &req,
     THR_DETACHED | THR_NEW_LWP);

  // Tell Orbix we'll dispatch request later
  return -1;
}

void *TPR_Filter::worker_thread (void *arg)
{
  CORBA::Request *req =
    static_cast<CORBA::Request *> (arg);
  CORBA::Orbix.continueThreadDispatch (*req);
}

// Thread filter (automatically registered)...
TPR_Filter global_thread_dispatcher;
```


Thread-per-Request Quoter Implementation

- Implementation of thread-safe Quoter callback invoked by the CORBA skeleton

```
long
My_Quoter::get_quote (const char *stock_name)
{
  // Increment the request count (beware...).
  ++My_Quoter::req_count_;

  // Obtain stock price (beware...).
  long value = Quote_Database::instance ()->
             lookup_stock_price (stock_name);

  if (value == -1)
    // Skeleton handles exceptions.
    throw Stock::Invalid_Stock ();

  return value;
}
```


Eliminating Race Conditions

- *Problem*
  - The concurrent Quote server contains "race conditions" *e.g.*,
    - Auto-increment of static variable **req_count_** is not serialized properly
    - **Quote_Database** also may not be serialized...
- *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;

long
My_Quoter::get_quote (const char *stock_name)
{
  mutex_lock (&lock);
  // Increment the request count.
  ++My_Quoter::req_count_;

  // Obtain stock price.
  long value = Quote_Database::instance ()->
             lookup_stock_price (stock_name);
  if (value == -1)
    // Skeleton handles exceptions.
    throw Stock::Invalid_Stock ();
  mutex_unlock (&lock);
  return value;
}
```

Problems Galore!

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

C++ Wrappers for Synchronization

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

```
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;

long My_Quoter::get_quote (const char *stock_name)
{
  lock.acquire ();
  // Increment the request count.
  ++My_Quoter::req_count_;
  // Obtain stock price.
  long value = Quote_Database::instance ()->
               lookup_stock_price (stock_name);
  if (value == -1)
    // Skeleton handles exceptions.
    throw Stock::Invalid_Stock ();
  lock.release ();
  return value;
}
```

- 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;

long
My_Quoter::get_quote (const char *stock_name)
{
  Guard<Thread_Mutex> mon (lock);
  // Increment the request count.
  ++My_Quoter::req_count_;

  // Obtain stock price.
  long value = Quote_Database::instance ()->
               lookup_stock_price (stock_name);
  if (value == -1)
    // Skeleton handles exceptions.
    throw Stock::Invalid_Stock ();
  return value; // Destructor of mon release lock.
}
```

- However, using the Thread_Mutex and Guard classes is still overly obtrusive and subtle (may lock too much scope...)

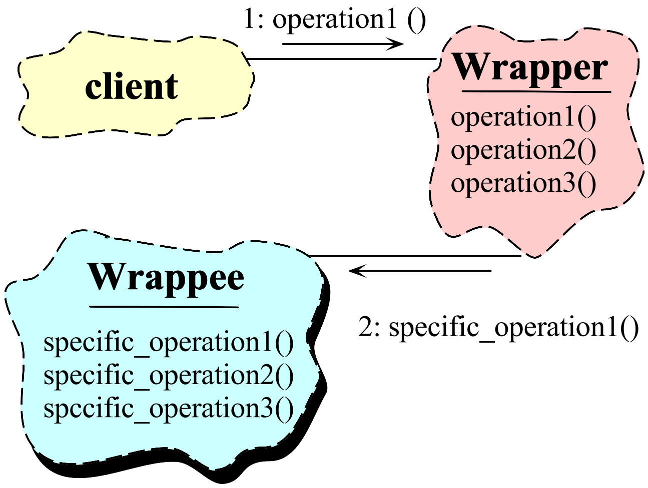
OO Design Interlude

- 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 Facade and Adapter patterns to provide identical interfaces that facilitate parameterization

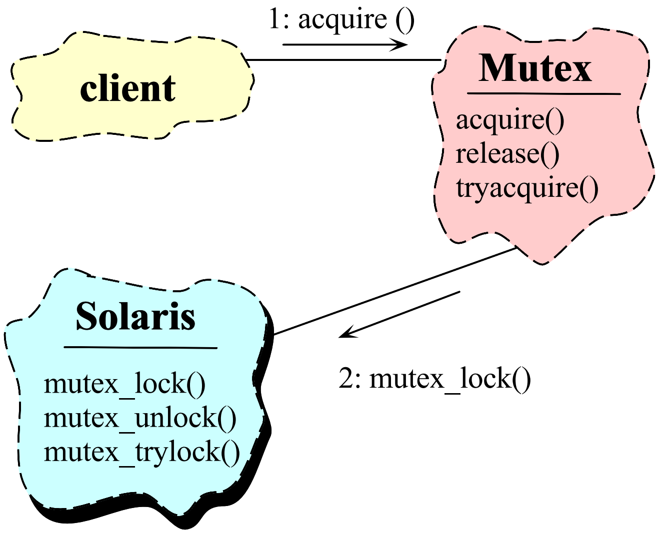
The Wrapper Facade Pattern

- *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*

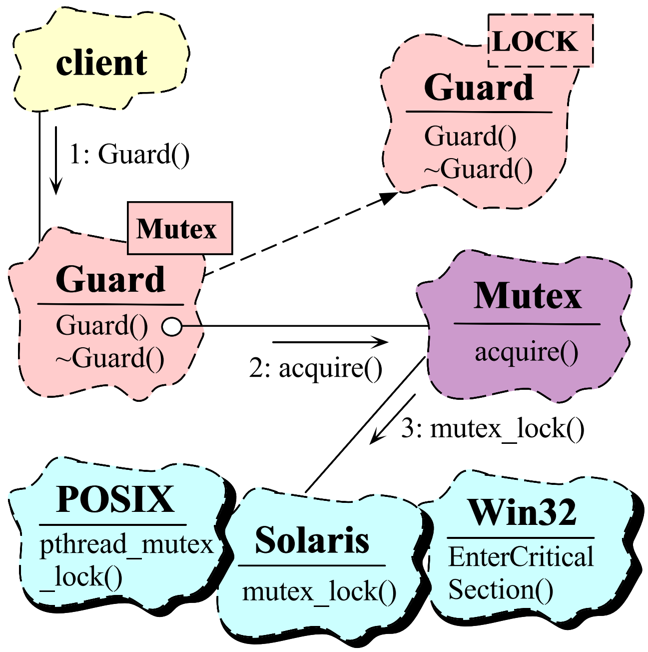
Structure of the Wrapper Facade Pattern



Using the Wrapper Facade Pattern for Locking



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 = ACE_Thread_Mutex,
          class TYPE = u_long>
class ACE_Atomic_Op {
public:
  ACE_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_;
};
```

Using ACE_Atomic_Op

- A few minor changes are made to the class header:

```
#if defined (MT_SAFE)
typedef ACE_Atomic_Op<> COUNTER; // Note default parameters
#else
typedef ACE_Atomic_Op<ACE_Null_Mutex> COUNTER;
#endif /* MT_SAFE */
```

- In addition, we add a lock, producing:

```
class My_Quoter
{
// ...
  // Serialize access to database.
  ACE_Thread_Mutex lock_;

  // Maintain request count.
  static COUNTER req_count_;
};
```


Thread-safe Version of Quote Server

- **req_count_** is now serialized automatically so only minimal scope is locked

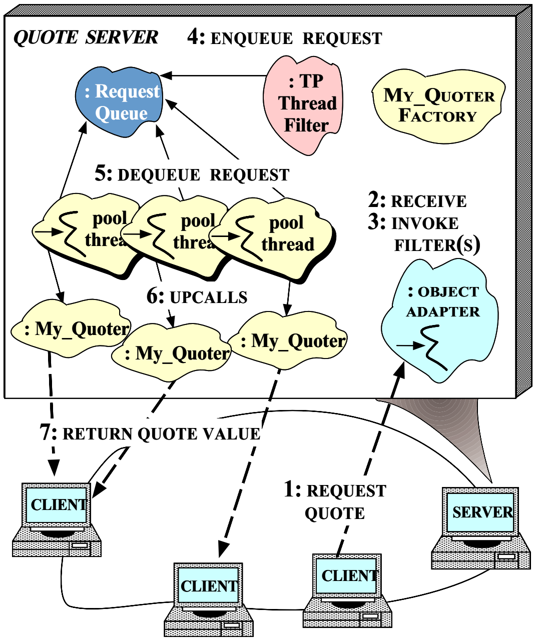
```
long
My_Quoter::get_quote (const char *stock_name)
{
  // Increment the request count by Calling
  // ACE_Atomic_Op::operator++(void)
  ++My_Quoter::req_count_;
  long value;
  {
    Guard<Thread_Mutex> mon (lock);

    // Obtain stock price.
    value = Quote_Database::instance ()->
            lookup_stock_price (stock_name);
    // Destructor of mon release lock.
  }
  if (value == -1)
    // Skeleton handles exceptions.
    throw Stock::Invalid_Stock ();
  return value;
}
```


Thread Pool

- This approach creates a thread pool to amortize the cost of dynamically creating threads
- In this scheme, before waiting for input the server code creates the following:
  1. A **Quoter_Factory** (as before)
  2. A pool of threads based upon the command line input
- Note the use of the ACE **spawn_n** method for spawning multiple pool threads

Thread Pool Concurrency Architecture



Thread Pool Main Program

```
const int DEFAULT_POOL_SIZE = 3;

// Thread filter (automatically registered)...
TP_Filter global_thread_dispatcher;

int main (int argc, char *argv[])
{
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, 0);
  ORB::BOA_var boa = orb->BOA_init (argc, argv, 0);

  // Initialize the factory implementation.
  Quoter_Factory_var qf =
    new TIE_Quoter_Factory (My_Quoter_Factory)
         (new My_Quoter_Factory);

  int pool_size = argc < 2 ? DEFAULT_POOL_SIZE
                           : atoi (argv[1]);

  // Create a thread pool.
  ACE_Thread_Manager::instance ()->spawn_n
    (pool_size,
     TP_Filter::pool_thread,
     (void *) &global_thread_dispatcher,
     THR_DETACHED | THR_NEW_LWP);

  // Wait for work to do....
  try {
    boa->impl_is_ready ("Quoter_Factory");
  } catch (...) { /* Handle exception... */ }
}
```


Thread Pool Filter Public Interface

- This approach uses an ACE `Message_Queue` to buffer requests

```
// Create a Thread Filter to dispatch incoming calls.

class TP_Filter : public CORBA::ThreadFilter
{
public:
  // Intercept request insert at end of req_queue_.
  virtual int inRequestPreMarshal (CORBA::Request &);

  // A pool thread executes this...
  static void *pool_thread (void *);

  // Called by our own pool threads.
  CORBA::Request *dequeue_head (void);
```


Thread Pool Filter Non-Public Interface

- Note the use of `ACE_Atomic_Op` to control access to counter...

```
protected:
  // Called by thread filter.
  void enqueue_tail (CORBA::Request *);

private:
  // Atomically increment request count.
  static ACE_Atomic_Op<ACE_Thread_Mutex,
                       u_long> req_count_;

  // Thread-safe message queue.
  ACE_Message_Queue<ACE_MT_SYNCH> req_queue_;
};
```


Thread Pool Filter Implementation

- The main Quoter implementation code is similar to the Thread-per-Request version
  - The differences are primarily in the thread and filter code and are highlighted below:

```
// static member function for thread entry point.

void *TP_Filter::pool_thread (void *arg)
{
  TP_Filter *tf = static_cast <TP_Filter *> (arg);

  // Loop forever, dequeueing new Requests,
  // and dispatching them....

  for (;;) {
    CORBA::Request *req = tf->dequeue_head ();

    // This call will perform the upcall,
    // send the reply (if any) and
    // delete the CORBA::Request for us...
    CORBA::Orbix.continueThreadDispatch (*req);
  }
  /* NOTREACHED */
}
```

Implementing the TP_Filter

- As requests come in they are inserted at the end of the queue

```
TP_Filter::inRequestPreMarshal (CORBA::Request &req)
{
  // Will block when "full".
  enqueue_tail (&req);

  // Tell Orbix we'll dispatch the request later..
  return -1;
}
```

- Meanwhile, all the threads wait for requests to arrive on the head of the message queue
  - If all the threads are busy, the queue keep growing
  - As always, flow control is an important concern...

Implementing the Request Queue

- The queue of CORBA Requests reuses the ACE thread-safe `Message_Queue`

```
void
TP_Filter::enqueue_tail (CORBA::Request *req)
{
  ACE_Message_Block *mb =
    new ACE_Message_Block ((char *) req);

  // Will block if queue is full.  This is where
  // we need to handle flow control policies...
  req_queue_.enqueue_tail (mb);
}

CORBA::Request *TP_Filter::dequeue_head (void)
{
  ACE_Message_Block *mb;

  // Will block if queue is empty.
  req_queue_.dequeue_head (mb);

  CORBA::Request *req =
    static_cast <CORBA::Request *> (mb->base ());
  delete mb;
  return req;
}
```

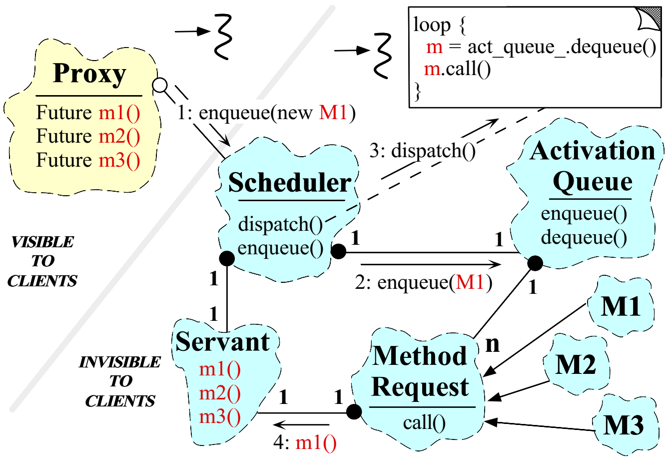

Thread-per-Object

- The third example of using threads is the most complicated
  - An ACE thread-safe `Message_Queue` is used, as with the Thread Pool
  - However, each object has its own thread and its own queue
    - ∗ Rather than one queue of incoming requests per server...
- Like Thread-per-Request, no threads are started in advance
  - However, the implementation differs...
- This is a classic example of the "Active Object" pattern
  - In this case, each object maintains a different client "session"

The Active Object Pattern

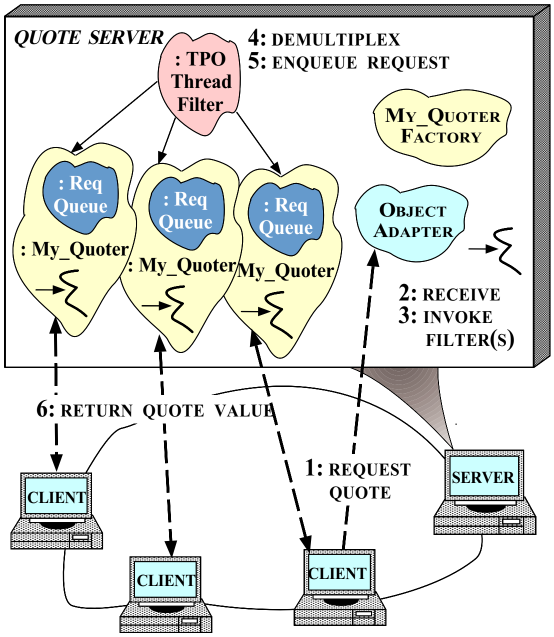
- *Intent*
  - "Decouples method execution from method invocation and simplifies synchronized access to shared resources by concurrent threads"
- This pattern resolves the following forces for concurrent communication software:
  - *How to allow blocking operations (such as read and write) to execute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

Active Object Pattern



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

Thread-per-Object Concurrency Architecture



Thread-per-Object Quoter Public Interface

- Each `Quoter` active object has its own thread of control

```
class My_Quoter
{
public:
  // Constructor
  My_Quoter (const char *name);

  // Returns the current stock value.
  virtual long get_quote (const char *stock_name);

  // A thread executes this per-active object.
  static void *object_thread (void *);

  // Thread filter uses this to queue the Request
  virtual void enqueue_tail (CORBA::Request *);
```


Thread-per-Object Quoter Non-Public Interface

- Each `Quoter` active object has a `Message_Queue`

```
protected:
  // Queue of pending requests called by our thread.
  CORBA::Request *dequeue_head (void);

private:
  // Atomically increment request count.
  static ACE_Atomic_Op<ACE_Thread_Mutex,
                       u_long> req_count_;

  // Thread-safe message queue.
  ACE_Message_Queue<ACE_MT_SYNCH> req_queue_;
};
```

Thread-per-Object Constructor

- The contructor spawns a separate thread of control
- This thread runs the event loop of the active object

```
My_Quoter::My_Quoter (const char *name)
{
  // Activate a new thread for the Quoter object.
  ACE_Thread_Manager::instance ()->spawn
    (My_Quoter::object_thread,
     this, // Get My_Quoter.
     THR_DETACHED | THR_NEW_LWP);
}
```


Thread-per-Object Quoter Factory Implementation

- Static entry point method:

```
void *My_Quoter::object_thread (void *arg)
{
  My_Quoter_ptr quoter =
    static_cast<My_Quoter_ptr> (arg);

  // Loop forever, receiving new Requests,
  // and dispatching them....

  for (;;)
    {
      CORBA::Request *req = quoter->dequeue_head ();

      // This call will perform the upcall,
      // send the reply (if any) and
      // delete the Request for us...
      CORBA::Orbix.continueThreadDispatch (*req);
    }
  /* NOTREACHED */
  return 0;
}
```


Thread-per-Object Quoter Factory Implementation

- Threads are created by the `My_Quoter` constructor
  - Note the reuse of components from Thread Pool

```
CORBA::Object_ptr
My_Quoter_Factory::create_object
  (const CosLifeCycle::Key &factory_key,
   const CosLifeCycle::Criteria &the_criteria)
{
  My_Quoter_ptr q = new My_Quoter (factory_key.id);

  Quoter_ptr quoter = new TIE_QUOTER (q);

  // Be sure to duplicate the object reference!
  return quoter->_duplicate ();
}
```


Thread-per-Object Filter

- The filter must figure out which object an incoming request references
- This allows the filter to queue the request in the right active object
- Note that Orbix does the demultiplexing for us automatically!
- However, we must make sure we ignore everything that isn't a `Quoter` (such as the `Quoter_Factory` requests or others)

Thread-per-Object Filter Implementation

- This filter implementation is more complex

```
TPO_Filter::inRequestPreMarshal (CORBA::Request &req)
{
  CORBA::Object_ptr obj = req.target ();

  // Ensure its a Quoter (could be Quoter_Factory).
  CORBA::Environment env;
  Quoter_ptr quoter = Quoter::_narrow (obj, env);

  if (env) // Must be the Quoter_Factory or other...
           // continue the work on THIS thread.
    return 1; // tell Orbix to continue normally.

  // Get the My_Quoter object.
  My_Quoter_ptr my_quoter =
    static_cast <My_Quoter_ptr> (DEREF (quoter));

  // Pass the request to the per object thread.
  my_quoter->enqueue_tail (&req);

  // Tell Orbix we will dispatch the request later..
  return -1;
```


Client Application

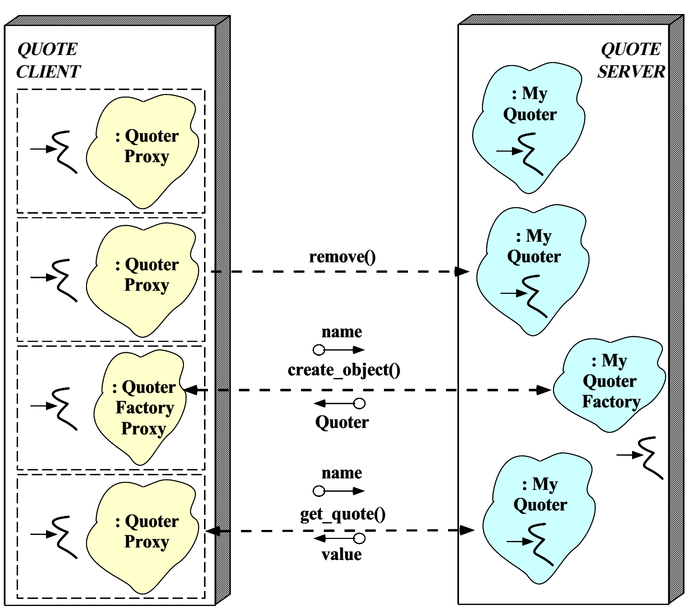
- The client works with any server concurrency model

- The client obtains a `Quoter_Factory` object reference, spawns n threads, and obtains a `Quoter` object reference per-thread

- Each thread queries the `Quoter` 100 times looking up the value of the ACME ORBs stock

- The main routine then waits for the threads to terminate

Client/Server Structure



Client Code

- The entry point function that does a remote invocation to get a stock quote from the server
  - This executes in one or more threads

```
static void *get_quotes (void *arg)
{
  Quoter_Factory_ptr factory =
    static_cast<Quoter_Factory_ptr> (arg);

  CosLifeCycle::Key key =
    Options::instance ()->key ();
  Quoter_var quoter = Stock::Quoter::_narrow
    (factory->create_object (key));

  if (!CORBA::is_nil (quoter)) {
    for (int i = 0; i < 100; i++) {
      try {
        long value = quoter->get_quote ("ACME ORBs");
        cout << "value = " << value << endl;
      } catch (...) { /* Handle exception */ }
    }
    quoter->remove ();
  }
}
```

Main Client Program

- Client spawns threads to run the `get_quotes` function and waits for threads to exit

```
int main (int argc, char *argv[])
{
  Options::instance ()->parse_args (argc, argv);

  try {
    // Create a remote Quoter_Factory.

    // Narrow to Quoter_Factory interface.
    Quoter_Factory_var factory =
      bind_service<Quoter_Factory> ("My_Quoter_Factory",
                                    argc, argv);

    // Create client threads.
    ACE_Thread_Manager::instance ()->spawn_n
      (Options::instance ()->threads (), get_quotes,
       (void *) factory, THR_DETACHED | THR_NEW_LWP);

    // Wait for the client threads to exit
    ACE_Thread_Manager::instance ()->wait ();
  } catch (...) { /* ... */ }
}
```


Obtaining an Object Reference

- Obtain an object reference

```
template <class T> T *
bind_service (const char *name,
              int argc, char *argv[]) {
  static CosNaming::NamingContext_ptr name_context = 0;
  CORBA::Object_var obj;
  CosNaming::Name svc_name;
  svc_name.length (1); svc_name[0].id = name;

  // ``First time in'' check.
  if (name_context == 0) {
    // Get reference to name service.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, 0);

    obj = orb->resolve_initial_references ("NameService");

    name_context = CosNaming::NamingContext::_narrow (obj);
  }

  // Find object reference in the name service.
  obj = name_context->resolve (svc_name);

  // Narrow to the T interface and away we go!
  return T::_narrow (obj);
}
```


Evaluating the Concurrency Models

- Thread-per-Request
  - *Advantages*
    - Simple to implement
    - Permits fine-grain load balancing
    - Most useful for long-duration requests
  - *Disadvantages*
    - Excessive overhead for short-duration requests
    - Permits unbounded number of concurrent requests
    - Application responsible for concurrency control

Evaluating the Concurrency Models (cont'd)

- Thread Pool
  - *Advantages*
    - Bounds the number of concurrent requests
    - Scales nicely for multi-processor platforms
    - Permits load balancing
  - *Disadvantages*
    - Applications must handle concurrency control
    - Potential for Deadlock

Evaluating the Concurrency Models (cont'd)

- Thread-per-Object
  - *Advantages*
    - May simplify concurrency control when re-working single-threaded code
  - *Disadvantages*
    - Does not support load balancing
    - Potential for deadlock on nested callbacks

Concluding Remarks

- Orbix supports several threading models
  - Performance may determine model choice
- ACE provides key building blocks for simplifying concurrent application code
  - www.cs.wustl.edu/~schmidt/ACE.html
- More information on CORBA can be obtained at
  - www.cs.wustl.edu/~schmidt/corba.html
- C++ Report columns written with Steve Vinoski
  - www.cs.wustl.edu/~schmidt/report-doc.html