Implementing Multi-threaded CORBA Applications with ACE and TAO

Douglas C. Schmidt

Associate Professor
schmidt@uci.edu
www.eng.uci.edu/~schmidt/

Elec. & Comp. Eng. Dept.
University of California, Irvine
(949) 824-1901

Sponsors

NSF, DARPA, ATD, BBN, Boeing, Cisco, Comverse, GDIS, Experian, Global MT, Hughes, Kodak, Krones, Lockheed, Lucent, Microsoft, Mitre, Motorola, NASA, Nokia, Nortel, OCI, Oresis, OTI, Raytheon, SAIC, Siemens SCR, Siemens MED, Siemens ZT, Sprint, Telcordia, USENIX

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 ACE and TAO
- It also presents several concurrency models
  1. *Thread-per-Connection*
  2. *Thread Pool*

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

---

CORBA Quoter Example

```
int main (void)
{
  // Use a factory to bind
  // to a Quoter.
  Quoter_var quoter =
    bind_quoter_service ();

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

  CORBA::Long value =
    quoter->get_quote (name);
  cout << name << " = "
       << value << endl;
}
```

- Ideally, a distributed service should look just like a non-distributed service
- Unfortunately, life is harder when errors occur...

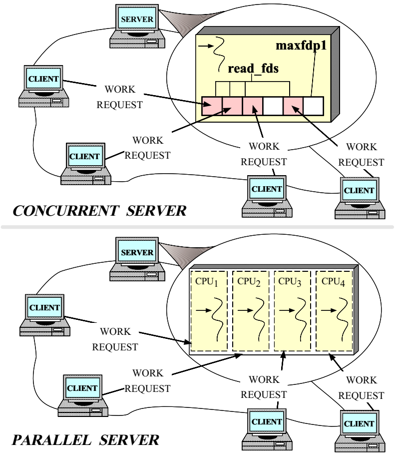
CORBA Quoter Interface

```
// IDL interface is like a C++
// class or Java interface.
interface Quoter
{
  exception Invalid_Stock {};

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

- We write an OMG IDL `interface` for our `Quoter`
  - Used by both clients and servers

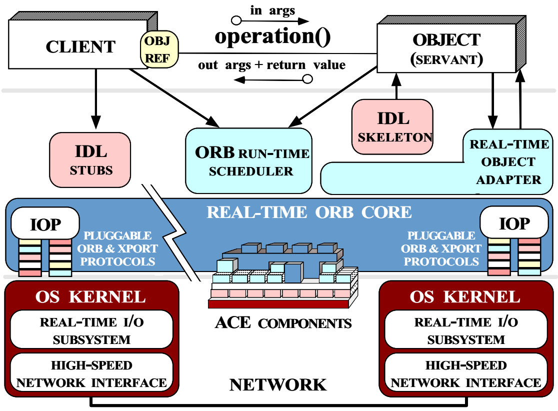
Using OMG IDL promotes *language/platform independence*, *location transparency*, *modularity*, and *robustness*

Motivation for Concurrency in CORBA



- *Leverage hardware/software*
  - *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.*, sync vs. async

Overview of The ACE ORB (TAO)



www.cs.wustl.edu/~schmidt/TAO.html

TAO Overview →

- An open-source, standards-based, real-time, high-performance CORBA ORB
- Runs on POSIX/UNIX, Win32, & RTOS platforms
  - *e.g.*, VxWorks, Chorus, LynxOS
- Leverages ACE

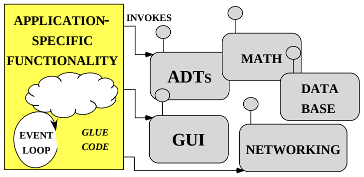
Threading in TAO

- An application can choose to ignore threads and if it creates none, it need not be thread-safe
- TAO can be configured with various concurrency strategies:
  - Thread-per-Connection
  - Thread Pool
  - Thread-per-Endpoint
- TAO also provides many locking strategies
  - TAO doesn't automatically synchronize access to application objects
  - Therefore, applications must synchronize access to their own objects

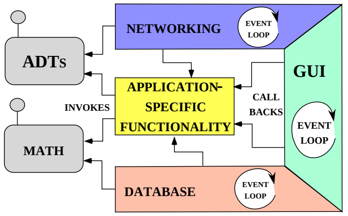
Overcoming Limitations with CORBA

- *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 ACE OO communication framework

Class Libraries, Frameworks, and Components



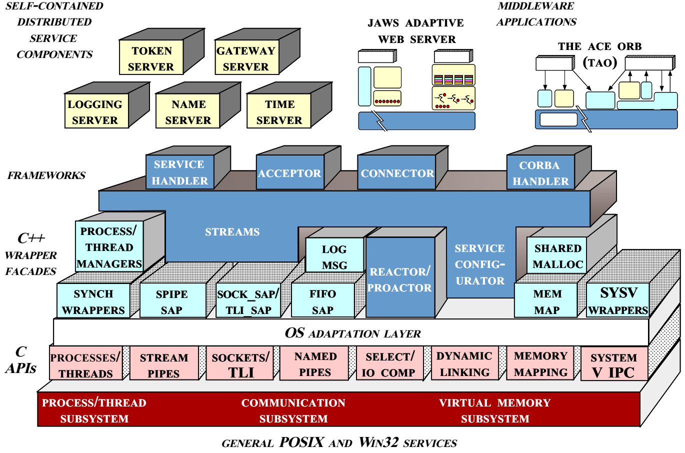
(A) CLASS LIBRARY ARCHITECTURE



(B) FRAMEWORK ARCHITECTURE

- **Proven solutions**
  - *Components*
    - Self-contained, “pluggable” ADTs
  - *Frameworks*
    - Reusable, “semi-complete” applications
  - *Patterns*
    - Problem/Solution/Context
  - *Architecture*
    - Families of related patterns and components

The ADAPTIVE Communication Environment (ACE)

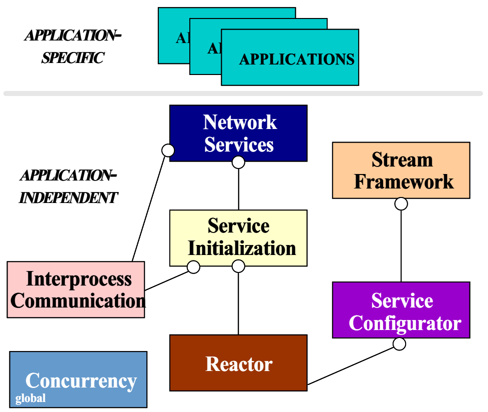


www.cs.wustl.edu/∼schmidt/ACE.html

ACE Statistics

- ACE contains $>$ 200,000 lines of C++
  - Over 25 person-years of effort
- Ported to UNIX, Win32, MVS, and RT/embedded platforms
  - *e.g.*, VxWorks, LynxOS, Chorus
- Large user community
  - ∼schmidt/ACE-users.html
- Currently used by dozens of companies
  - Bellcore, BBN, Boeing, Ericsson, Hughes, Kodak, Lockheed, Lucent, Motorola, Nokia, Nortel, Raytheon, SAIC, Siemens, etc.
- Supported commercially by Riverace
  - www.riverace.com

Class Categories in ACE



- ACE is structured as a “forest” of class categories
- This design permits fine-grained subsetting of ACE components
  - Subsetting helps minimize ACE’s memory footprint

ACE Class Category Responsibilities

- `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 demuxing* and *event handler dispatching*
- `Service Configurator` automates *(re)configuration* 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

---

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);
```

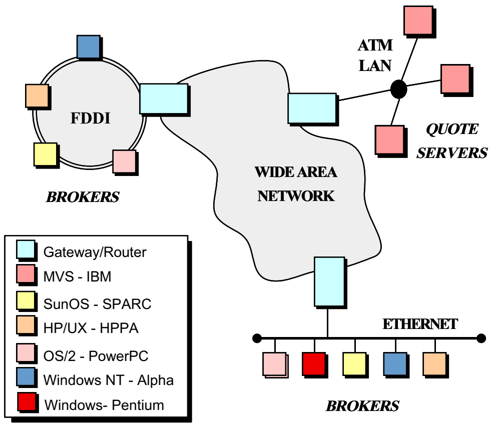
TAO Multi-threading Examples

- 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 in two different ways:

1. *Thread-per-Connection* → Every client connection 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

- Note that clients are unaware which concurrency model is being used...

CORBA Stock Quoter Application Example

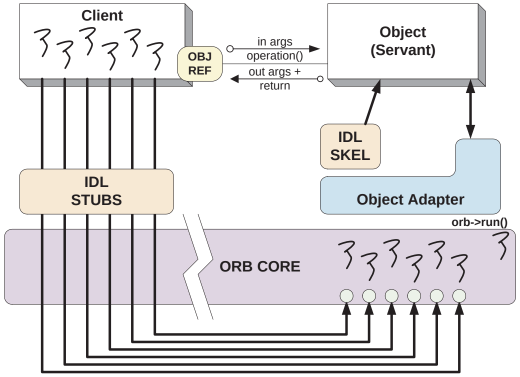


- The quote server(s) maintains the current stock prices
- Brokers access the quote server(s) via CORBA
- Note all the heterogeneity!

Simple OMG IDL Quoter Definition

```
module Stock {
  // Exceptions are similar to structs.
  exception Invalid_Stock {};
  exception Invalid_Factory {};

  // Interface is similar to a C++ class.
  interface Quoter {
    long get_quote (in string stock_name) raises (Invalid_Stock);
  };
  // A factory that creates Quoter objects.
  interface Quoter_Factory {
    // Factory Method that returns a new Quoter
    // selected by name e.g., "Dow Jones,"
    // "Reuters,", etc.
    Quoter create_quoter (in string quoter_service)
      raises (Invalid_Factory);
  };
};
```

TAO’s Thread-per-Connection Concurrency Architecture



Pros

- Simple to implement and efficient for long-duration requests

Cons

- Excessive overhead for short-duration requests
- Permits unbounded number of concurrent requests

Thread-per-Connection Main Program

The server creates a single Quoter factory and waits in ORB's event loop

```
int main (void)
{
  ORB_Manager orb_manager (argc, argv);
  const char *factory_name = "my quoter factory";

  // Create the servant, which registers with rootPOA
  // and Naming Service implicitly.
  My_Quoter_Factory factory (factory_name);

  // Block indefinitely waiting for incoming invocations
  // and dispatch upcalls.
  // After run() returns, the ORB has shutdown.
  orb_manager.run ();
}
```

The ORB's `svc.conf` file

```
static Resource_Factory "-ORBResources global -ORBReactorType select_mt"
static Server_Strategy_Factory "-ORBConcurrency thread-per-connection"
```

Thread-per-Connection: Quoter Interface

```
typedef u_long COUNTER; // Maintain request count.

// Implementation of the Quoter IDL interface
class My_Quoter :
  virtual public POA_Stock::Quoter,
  virtual public PortableServer::RefCountServantBase
{
public:
  // Constructor.
  My_Quoter (const char *name);

  // Returns the current stock value.
  long get_quote (const char *stock_name)
    throw (CORBA::SystemException, Quoter::InvalidStock);
private:
  // Maintain request count.
  static COUNTER req_count_;
};
```

Thread-per-Connection Quoter Implementation

```
// Implementation of multi-threaded Quoter callback invoked by
// the CORBA skeleton
long My_Quoter::get_quote (const char *stock_name)
    throw(CORBA::SystemException,Quoter::InvalidStock)
{
  // Increment the request count (beware...).
  ++My_Quoter::req_count_;

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

  // Skeleton handles exceptions.
  if (value == -1)
    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

```
// SunOS 5.x, implicitly "unlocked".
mutex_t lock;

long
My_Quoter::get_quote (const char *stock_name)
    throw(CORBA::SystemException,Quoter::InvalidStock)
{
  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)
    throw(CORBA::SystemException,Quoter::InvalidStock)
{
  lock.acquire ();
  ++My_Quoter::req_count_; // Increment the request 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)
    throw(CORBA::SystemException,Quoter::InvalidStock)
{
  Guard<Thread_Mutex> mon (lock);
  ++My_Quoter::req_count_; // Increment the request 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

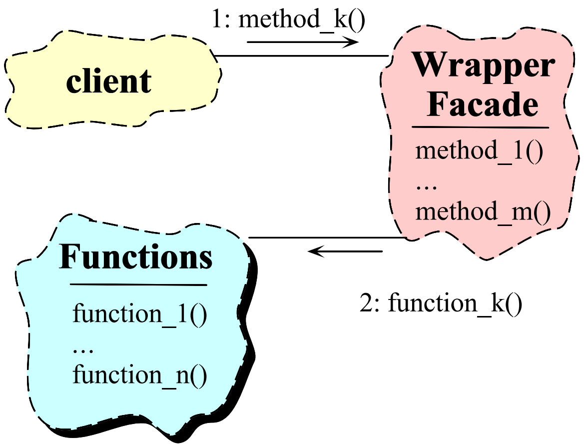
---

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



1: method_k()

client

Wrapper Facade

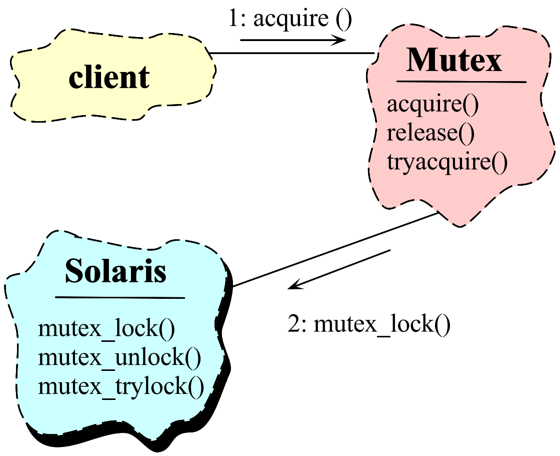
method_1()
...
method_m()

Functions

function_1()
...
function_n()

2: function_k()

Using the Wrapper Facade Pattern for Locking



1: acquire ()

client

Mutex

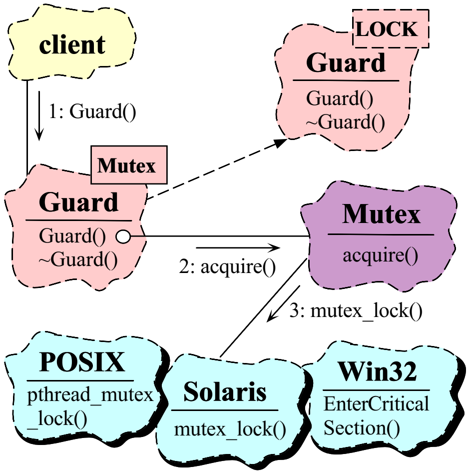
acquire()
release()
tryacquire()

Solaris

mutex_lock()
mutex_unlock()
mutex_trylock()

2: mutex_lock()

Using the Adapter Pattern for Locking



Parameterizing Synchronization Via 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 : virtual public POA_Stock::Quoter,
                  virtual public PortableServer::RefCountServantBase
{
// ...
  // 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

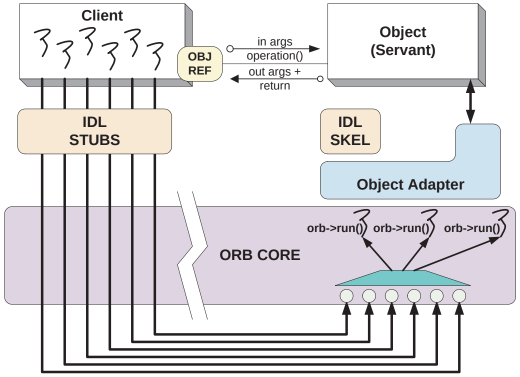
```
long
My_Quoter::get_quote (const char *stock_name)
{
  // Increment the request count by calling
  // ACE_Atomic_Op::operator++(void) decorator.
  ++My_Quoter::req_count_;

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

---

TAO's Thread Pool Concurrency Architecture



Pros

- Bounds the number of concurrent requests
- Scales nicely for multi-processor platforms, *e.g.*, permits load balancing

Cons

- Potential for Deadlock

Thread Pool Main Program

```
const int DEFAULT_POOL_SIZE = 8;

int main (int argc, char *argv[])
{
  try {
    ORB_Manager orb_manager (argc, argv);

    const char *factory_name = "my quoter factory";

    // Create the servant, which registers with
    // the rootPOA and Naming Service implicitly.
    My_Quoter_Factory factory (factory_name);

    // ...
```

Thread Pool Main Program (cont'd)

```
    int pool_size = argc < 2 ? DEFAULT_POOL_SIZE
                             : atoi (argv[1]);
    // Create a thread pool.
    ACE_Thread_Manager::instance ()->spawn_n
      (pool_size, &run_orb, (void *) orb_manager.orb (),
       THR_DETACHED | THR_NEW_LWP);

    // Block indefinitely waiting for other threads to exit.
    ACE_Thread_Manager::instance ()->wait ();

    // After run() returns, the ORB has shutdown.
    } catch (...) { /* handle exception ... */ }
}
```

Thread Pool Configuration

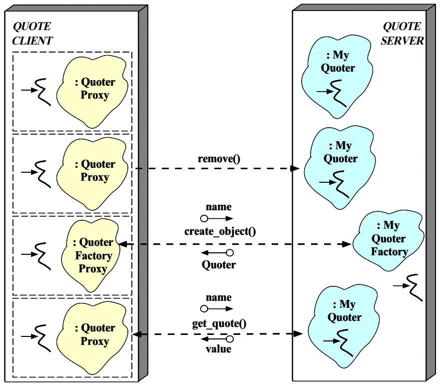
The `run_orb` adapter function

```
void run_orb (void *arg)
{
  try {
    CORBA::ORB_ptr orb =
      ACE_reinterpret_cast (CORBA::ORB_ptr, arg);

    // Block indefinitely waiting for incoming
    // invocations and dispatch upcalls.
    orb->run ();

    // After run() returns, the ORB has shutdown.
  } catch (...) { /* handle exception ... */ }
}
```

The ORB's `svc.conf` file

```
static Resource_Factory "-ORBReactorType tp"
```

Client/Server Structure



- 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 to obtain the value of ACME ORB's stock
- `main()` then waits for threads to terminate

Client Code

The entry point function that does a remote invocation to get a stock quote from the server

```
// This method 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 {
    // 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 via the Naming Service

```
static CORBA::ORB_ptr orb;
extern CosNaming::NamingContext_ptr name_context;

template <class T> typename T::_ptr_type /* trait */
bind_service (const char *n, int argc, char *argv[])
{
  CORBA::Object_var obj; // "First time" check.
  if (CORBA::is_nil (name_context)) {
    // Get reference to name service.
    orb = CORBA::ORB_init (argc, argv, 0);
    obj = orb->resolve_initial_references ("NameService");
    name_context = CosNaming::NamingContext::_narrow (obj);
    if (CORBA::is_nil (name_context)) return 0;
  }
  CosNaming::Name svc_name;
  svc_name.length (1); svc_name[0].id = n;
  svc_name[0].kind = "object impl";
  // 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);
}
```

Concluding Remarks

- TAO 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