Designing an Efficient & Scalable Server-side Asynchrony Model for CORBA

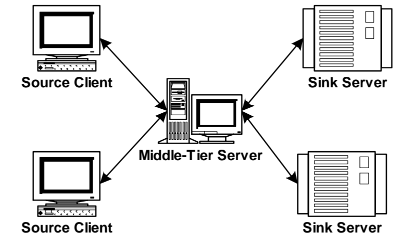
Darrell Brunsch, Carlos O'Ryan, & Douglas C. Schmidt
{brunsch,coryan,schmidt}@uci.edu
Department of Electrical & Computer Engineering
University of California, Irvine

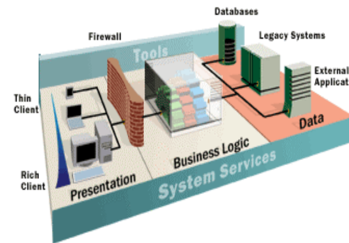
Wednesday, 13 June 2001

Research Sponsored by ATD, Cisco, DARPA, Raytheon, SAIC, & Siemens

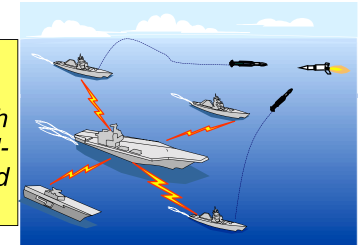
Motivation: Middle-Tier Servers

- In a multi-tier system, one or more "middle-tier" servers are placed between a *source client* & a *sink server*
  - A source client's two-way request may visit multiple middle-tier servers before it reaches its sink server
  - The result then flows in reverse through these intermediary servers before arriving back at the source client





Middle-tier servers are common in both business & real-time/ embedded systems

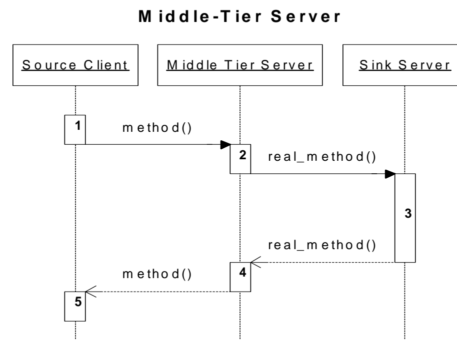


Challenges for Middle-Tier Servers

- Middle-tier servers must be highly scalable to avoid becoming a bottleneck when communicating with multiple source clients & sink servers

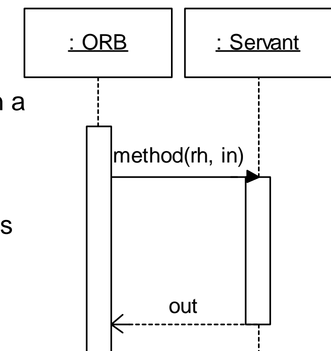
Typical middle-tier server steps
1. Client sends request
2. *Middle-tier processes the request & sends a new request to a sink server*
3. Sink server processes and returns data
4. *Middle-tier returns data to the client*
5. The client then processes the response data

- It's not scalable to dedicate a separate thread for each outstanding client request due to *thread creation, context switching, synchronization*, & *data movement* overhead



CORBA Limitations for Middle-Tier Servers

- It's hard to implement scalable & convenient middle-tier servers using standard CORBA
  - CORBA one-ways & DII/DSI are clearly inadequate
- Problems stem from the *tight coupling* between a server's receiving a request & returning a response *in the same activation record*
- This tight coupling limits a middle-tier server's ability to handle incoming requests & responses efficiently
  - *i.e.*, each request needs its own activation record
  - *This effectively restricts a request/ response pair to a single thread in standard CORBA*

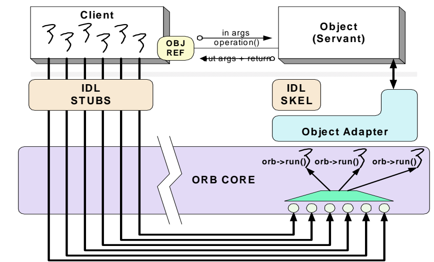


Design Characteristics of an Ideal Middle-tier Server Solution

- **Request throughput**
  - Provide high throughput for a client, *i.e.,* it should be able to handle a large number of requests per unit time, *e.g.,* per second or per "busy hour"
- **Latency/Jitter**
  - Minimize the request/ response processing delay (*latency*), as well as the variation of the delay (*jitter*)
- **Scalability**
  - Take advantage of multiple sink servers and handle many aggregate requests/responses
- **Portability**
  - Ideally, little or no changes & non-portable features should be required to implement a scalable solution
  - Clients should be completely unaware of middle-tier server existence
- **Simplicity**
  - Compared with existing designs, the solution should minimize the amount of work needed to implement scalable middle-tier server applications
  - Any ORB features required by the solution should be relatively easy to implement

Evaluating CORBA Server Concurrency Models

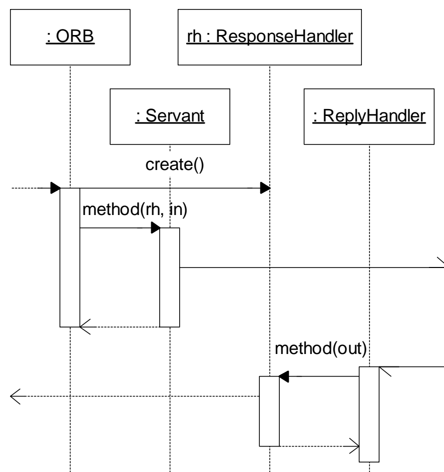
- There are a number of existing models for developing multi-tier servers:
  1. Single-threaded
  2. Nested upcalls & event loops
  3. Thread-per-request
  4. Static thread pools
  5. Dynamic thread pools
  6. Static thread pools with nested upcalls



- The single-threaded models (1 & 2) have the following characteristics
  - Low request throughput due to serialization
  - High latency/jitter due to serialization
  - Low scalability due to serialization
  - Good portability for #1
  - Good simplicity for simple use-cases
- The multi-threaded models (3–6) have the following characteristics
  - Good request throughput
  - Moderate-poor latency/jitter due to synchronization
  - Moderate scalability due to threading limits
  - Poor portability (except for ORBs compliant with RT-CORBA thread pools)
  - Good simplicity (if there's thread expertise)

Solution: Asynchronous Method Handling (AMH)

- AMH decouples the existing CORBA 1-to-1 association between
  1. An incoming request to the run-time stack and
  2. The activation record that received the request
- This design allows a server to return responses asynchronously, *without* incurring the overhead of multi-threading
- AMH is inspired by
  1. The CORBA asynchronous method invocation (AMI) model
  2. Continuations



Overview of SMI & AMI Models

Client ORB Server (1, 2)	Client ORB Poller Server (1, 2)	Client ORB reply handler Server (1, 2)
SMI model • The client invokes the operation & the ORB blocks • After the response is returned, the ORB returns control to the client application thread that invoked the operation	**AMI Polling Model** • The client invokes the operation & the call returns immediately • It later checks with the collocated Poller object to retrieve the response	**AMI Callback Model** • The client invokes the operation & the call returns immediately • The ORB later invokes the callback when the response arrives • Forms the basis for AMH

Proposed AMH Mapping

IDL:

```
interface Quoter {
  // A standard synchronous operation,
  // note that OMG IDL is not extended
  long get_quote (in string stock_name);
};
```

C++:

```
// Class implemented by apps
class My_AMH_Quoter
    : public POA_AMH_Quoter {
public:
 // ORB invokes this method, apps
 // implement Object behavior here
 virtual void get_quote (
     // ... the <rh> argument is
     // used to send response.  It
     // can be stored for later use
     AMH_QuoterResponseHandler_ptr rh,
     const char *stock_name);
};
```

C++:

```
// This class is implemented
// by the ORB
class AMH_QuoterResponseHandler
{
public:
 // Servers use this
 // method to send their
 // responses back to clients
 void get_quote
   (CORBA::Long return_value);
};
```

Programming C++ Servers with AMH & AMI

```
// Implement the get_quote()
// operation:
void My_AMH_Quoter::get_quote (
  AMH_QuoterResponseHandler_ptr rh,
  const char *stock_name)
{
  // We want to send AMI request
  // 1. Create the callback:
  My_Callback *cb =
    new My_Callback (rh);
  // 2. Activate the callback with
  //    the default POA
  AMI_Quoter_var callback =
    cb->_this ();

  // 3. Make the AMI request
  target_quoter_->sendc_get_quote
    (callback, stock_name);
}
```

```
// Implement the AMI ReplyHandler
class My_Callback : public
   POA_AMI_QuoterReplyHandler
{
public:
 // Save AMH response handler to
 // send the response later
 My_Callback
  (AMH_QuoterResponseHandler_ptr rh)
  : rh_ (AMH_QuoterResponseHandler
       ::_duplicate (rh)) {}

 // Callback operation, invoked by
 // ORB to send response to client
 // when sink server reply returns
 void get_quote (CORBA::Long retval)
 {
   rh_->get_quote (retval);
 }
private:
 AMH_QuoterResponseHandler_var rh_;
};
```

AMH Design Problems & Solutions

Problem	Solution
AMH violates most of the SMI assumptions regarding synchronization & concurrency optimizations	Use the *Strategy* pattern to encapsulate different algorithms & interchange them easily
How to ensure that only servants using AMH pay any penalties, such as additional dynamic memory allocators or footprint enlargement	Use the *Component Configurator* pattern to allow middleware or application developers to delay con-figuration decisions until run-time
How to leverage IDL compiler AMI stub generation for AMH skeleton generation	Use the *Visitor* pattern to represent operations that are performed & members of an object structure
How to minimize or remove all blocking I/O operations from the ORB	Support fully reactive & proactive I/O
How to handle multi-threading with AMH	Add a new **AMHCurrent** to represent all information normally contained in the thread activation

Evaluating AMH

- **Request throughput**
  - A middle-tier server can provide very high throughput by handling multiple incoming requests from a client asynchronously
- **Latency/Jitter**
  - When a request arrives, it's handled quickly & when the response returns from the sink server, a reply can be sent back immediately
  - Latency should be relatively low since no additional threads need be created to handle requests and wait for responses
  - However, more state is required than in the simple single-threaded case, resulting in more context stored on the heap
- **Scalability**
  - Scalability can be very high since the upcall for requests and callbacks on `ReplyHandler` objects need not block
  - Moreover, performance can be enhanced to take advantage of multiple CPUs by combining the AMI/AMH model with a thread pool
- **Portability**
  - AMH is not yet defined in a CORBA specification, nor is it widely implemented
- **Simplicity**
  - Server applications become more complicated if their code uses AMH & AMI
  - The ORB and IDL compiler also become more complicated because request lifetimes are decoupled from the lifetime of a servant upcall

Concluding Remarks

- Middle-tier servers need a scalable asynchronous programming model
  - The current AMI models don't suffice for middle-tier servers
- Our proposed asynchronous method handling (AMH) model supports efficient server-side asynchrony with relatively few changes to CORBA
  - AMH is similar to AMI, focusing on the server rather than the client
- Programming AMH applications requires more design decisions for server developers
  - However, performance gains should make the effort worthwhile
- An AMH implementation & performance results are forthcoming in TAO
  - **www.cs.wustl.edu/~schmidt/TAO.html**
- A paper on AMH is also available
  - **www.cs.wustl.edu/~schmidt/PDF/AMH.pdf**