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

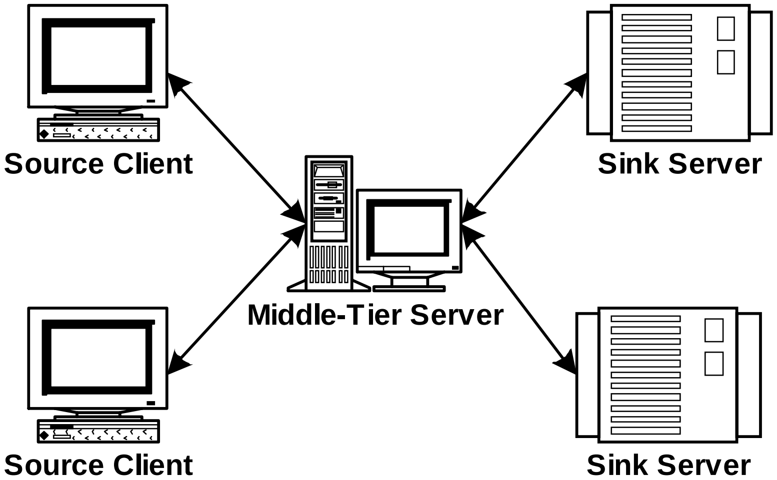
University of California, Irvine

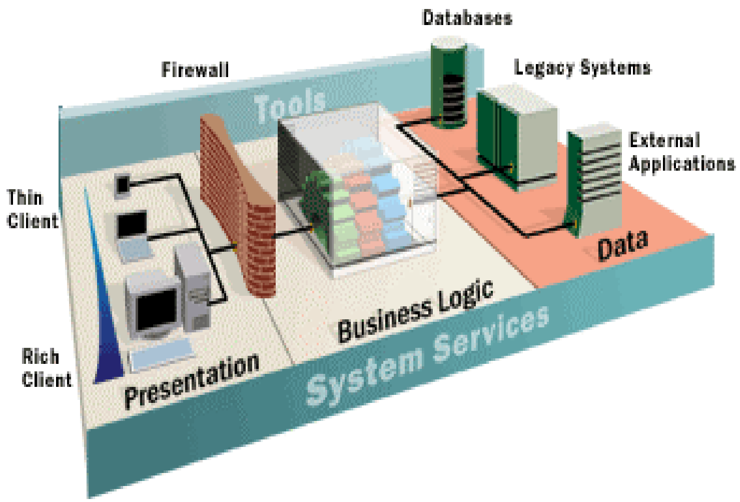
Wednesday, 18 July 2001

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

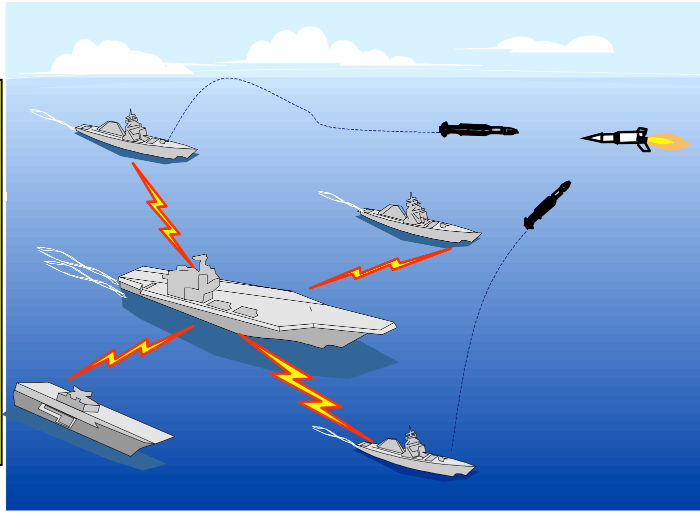
Motivation: Middle-Tier Servers

- 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



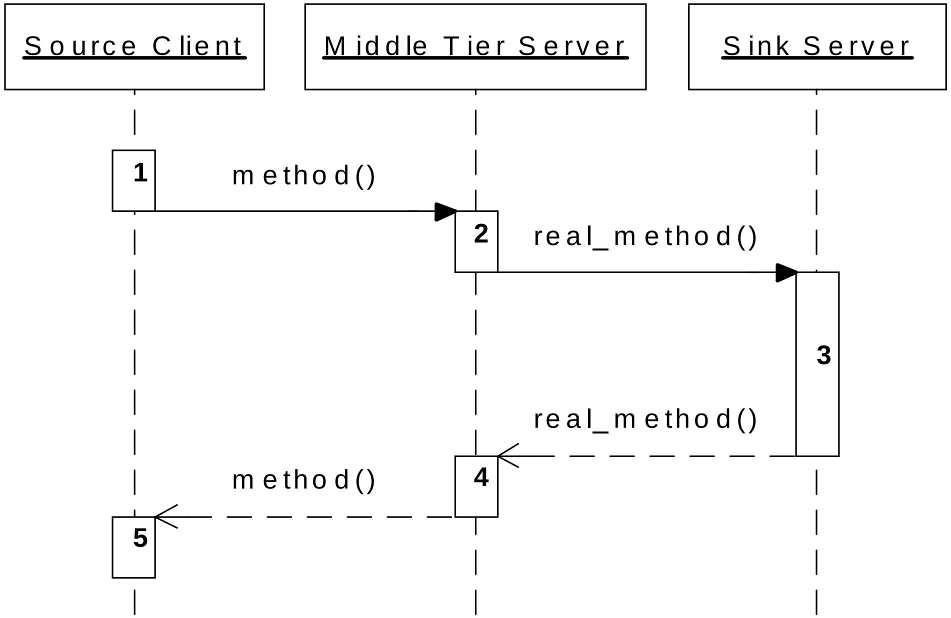
Challenges for Middle-Tier Servers

- Middle-tier servers must be highly scalable to avoid becoming a bottleneck when communicating with multiple source clients & sink servers
- It's not scalable to dedicate a separate thread for each outstanding client request due to *thread creation, context switching, synchronization*, & *data movement* overhead

Typical middle-tier server steps

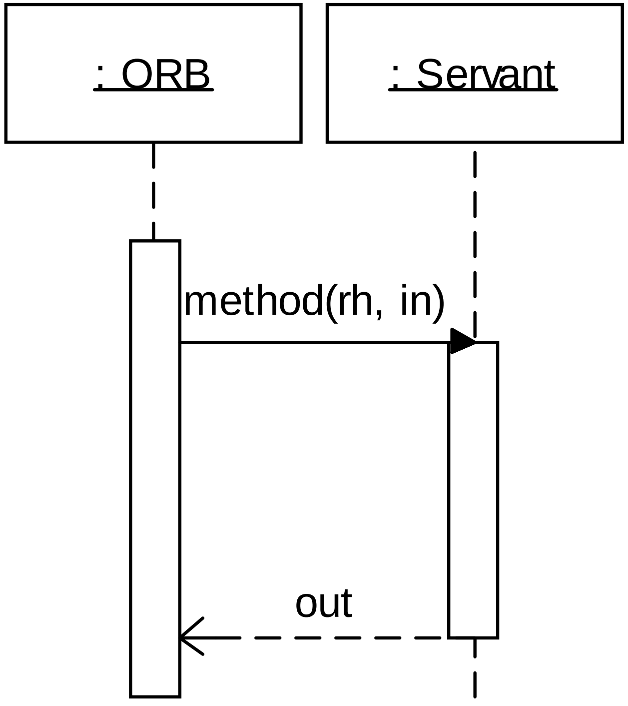
1. Client sends request
2. *Middle-tier processes the request & sends a new request to a sink server*
3. Sink server processes and returns data
4. *Middle-tier returns data to the client*
5. Which then processes the response data

Middle-Tier Server



CORBA Limitations for Middle-Tier Servers

- It's hard to implement scalable middle-tier servers using standard CORBA
- 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

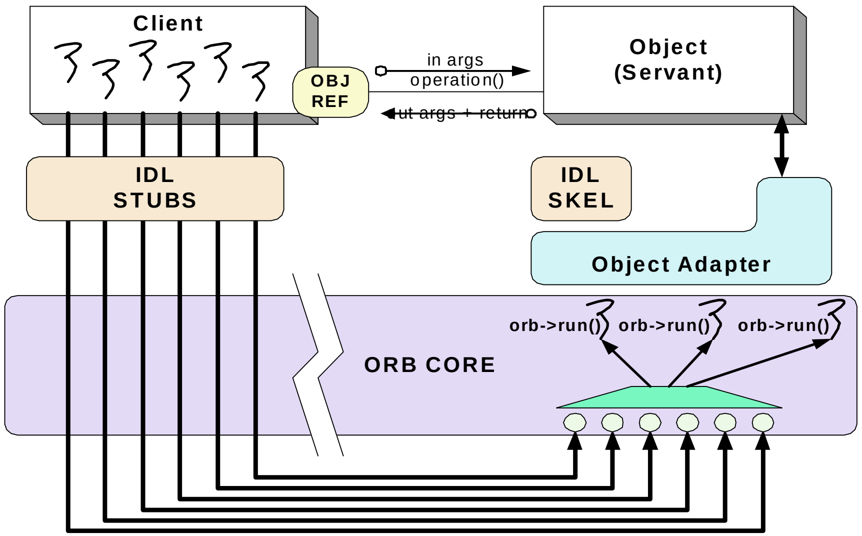


Design Characteristics of an Ideal Middle-tier Server Solution

- **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 and non-portable features should be required to implement a scalable solution
- **Simplicity**
  - Compared with existing designs, the solution should minimize the amount of work needed to implement scalable middle-tier servers
  - Any ORB features required by the solution should be 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

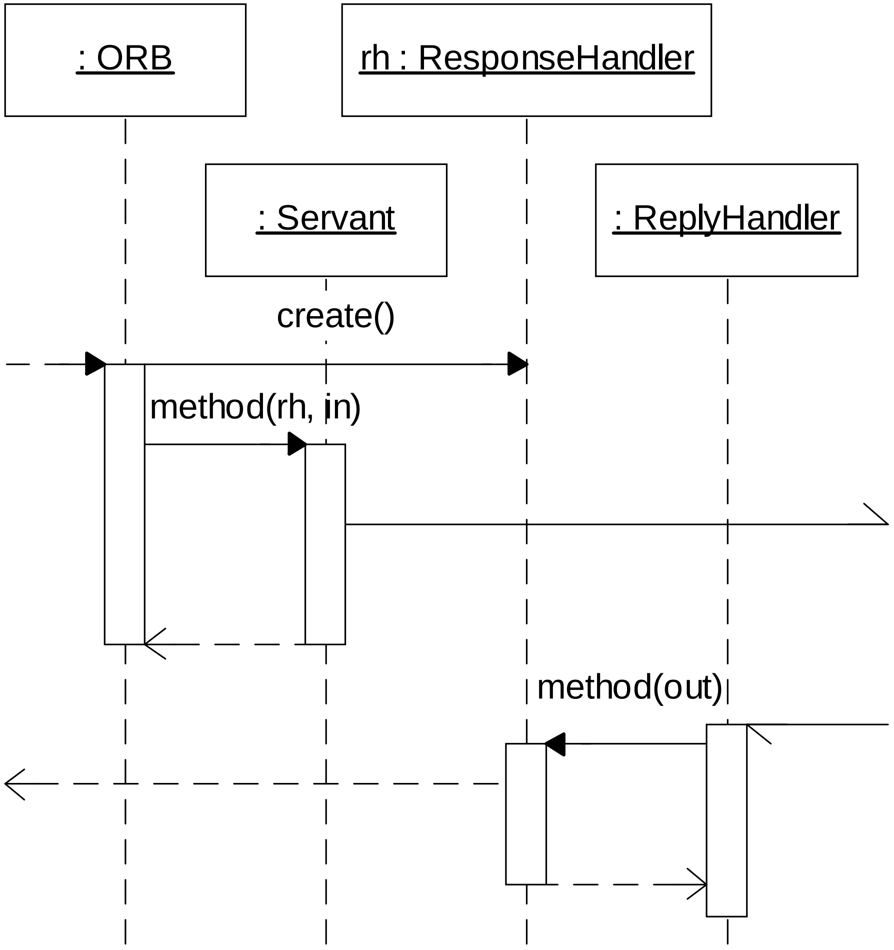


- The single-threaded models (1 & 2) have the following weaknesses
  - Low request throughput due to serialization
  - High latency/jitter due to serialization
  - Low scalability due to serialization
  - Good portability for #1
  - Good simplicity for simple use-cases

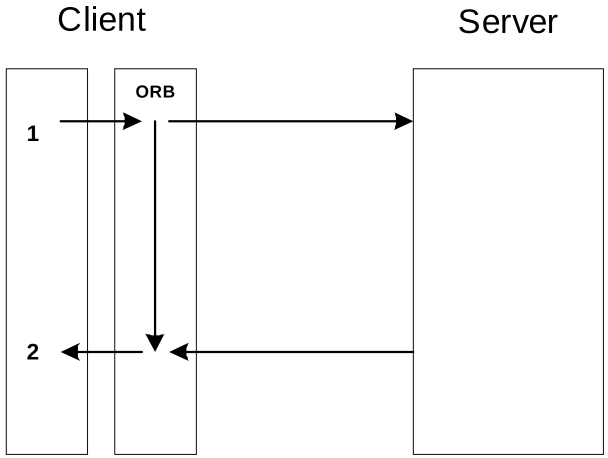
- The multi-threaded models (3–6) have the following weaknesses
  - Good request throughput
  - Moderate-poor latency/jitter due to synchronization
  - Moderate scalability due to threading limits
  - Poor portability (except for ORBs compliant with RT-CORBA thread pools)
  - Good simplicity (if there's thread expertise)

Solution: Asynchronous Method Handling (AMH)

- AMH decouples the existing CORBA 1-to-1 association of an incoming request to the run-time stack originating from 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)
  2. Continuations

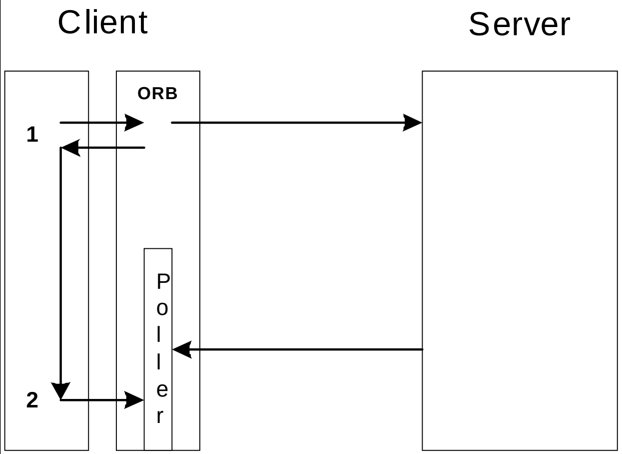


Overview of SMI & AMI Models



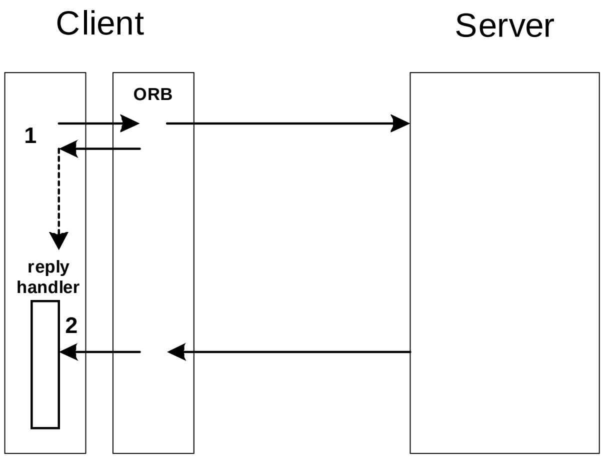
SMI model

- 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

Proposed AMH Mapping

IDL:

```
interface Quoter {
  // A standard synchronous operation.
  // 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:
 // Applications use this
 // method to send their
 // responses
 void get_quote
   (CORBA::Long return_value);
};
```

Programming Servers with AMH & AMI

C++:

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

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

C++:

```
// Implement the AMI ReplyHandler
class My_Callback : public
   POA_AMI_QuoterReplyHandler
{
public:
 // Save AMH response handler to
 // send the response later
 My_Callback (AMH_Quoter_ptr h)
  : handler_
      (AMH_Quoter::_duplicate(h))
  {}

 // Callback operation, invoked by
 // the ORB when the nested reply
 // shows up
 void get_quote (CORBA::Long retval)
 {
   handler_->get_quote (retval);
 }
};
```

Evaluating AMH

- **Request throughput**
  - Middle-tier servers can provide very high throughput by handling multiple incoming requests from clients asynchronously
- **Latency/Jitter**
  - When a request arrives, it is 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 implemented by many ORBs
- **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 CORBA
  - AMH is similar in spirit to AMI, it just focuses on the server, rather than client.
- Programming AMH applications requires more design decisions
- An implementation & performance results will be forthcoming in TAO
  - **`www.cs.wustl.edu/~schmidt/TAO.html`**