Scalable and Efficient ORB Architecture for Asynchronous Messaging

Alexander B. Arulanthu

Masters Student &
Research Assistant for
Distributed Object computing

Computer Science Dept.
Washington University, St. Louis
www.cs.wustl.edu/~alex/

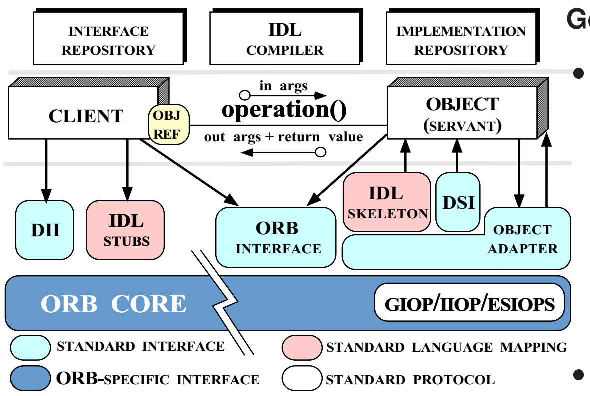
October 13, 1999

Advisor: Dr. Douglas C. Schmidt

Outline

- CORBA Introduction
- Limitations with Initial CORBA Model
- OMG Solution: CORBA Messaging Specification
- Our Work
- AMI Callback Overview
- ORB Architecture Enhancements
- Benchmarks
- Future Work
- Demo!
- Thanks!

Candidate Solution: CORBA



Goals of CORBA

- Simplify distribution by automating
  - Object location & activation
  - Parameter marshaling
  - Demultiplexing
  - Error handling
- Provide foundation for higher-level services

Limitations with Initial CORBA Model

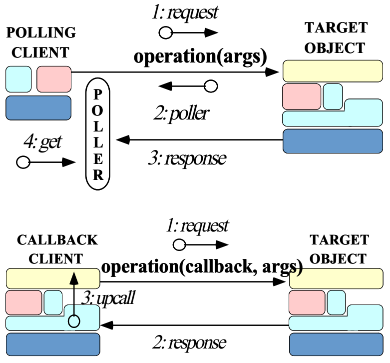
- Lack of support for QoS features
- Lack of support for asynchronous two-way invocations, which may
  1. Increase the number of client threads
     – *e.g.*, due to synchronous two-way communication
  2. Increase the end-to-end latency for multiple requests
     – *e.g.*, due to blocking on certain long-delay operations
  3. Decrease OS/network resource utilization
     – *e.g.*, inefficient support for bulk data transfers

Limitations with Workarounds for CORBA's Lack of Asynchrony

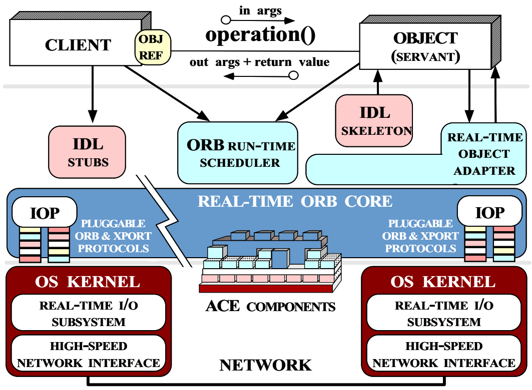
- *Synchronous method invocation* (SMI) multi-threading
  – Often non-portable, non-scalable, and inefficient
- Oneway operations
  – Best-effort semantics are unreliable
  – Requires *callback* objects
  – Applications must match callbacks with requests
- Deferred synchronous
  – Uses DII, thus *very* hard to program
  – Not type-safe

OMG Solution → CORBA Messaging Specification

- Defines *QoS Policies* for the ORB
  – Timeouts
  – Priority
- Specifies two *asynchronous method invocation* (AMI) models
  1. Poller model
  2. Callback model
- Standardizes *time-independent invocation* (TII) model
  – Used for store/forward routers

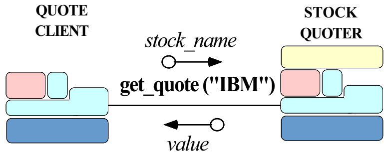


Our Work



- Implemented AMI Callback in TAO
- Enhanced ORB Architecture to support Connection Multiplexing
- Generalize Leader/Follower implementation for Connection Multiplexing
- Benchmarks

AMI Callback Overview



Quoter IDL Interface:

```
module Stock {
 interface Quoter {
   // Two-way operation to
   // get current stock value.
   long get_quote
     (in string stock_name);
 };
 // ...
}
```

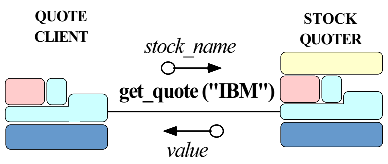
Implied-IDL:

```
module Stock {
 interface Quoter {
   // Two-way synchronous operation.
   long get_quote (in string stock_name);

   // Two-way asynchronous operation.
   void sendc_get_quote
      (Stock::AMI_QuoterHandler
       const char *stock);
  };
};

// ReplyHandler
module Stock {
  interface AMI_QuoterHandler
    : Messaging::ReplyHandler {
   public:
     // Callback method
     void get_quote (in long l);
   };
};
```

Example: Synchronous Client



IDL generated stub:

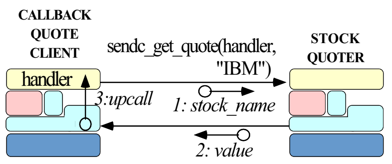
```
CORBA::ULong
Stock::Quoter::get_quote
  (const char *name)
{
  // setup connection
  // marshal
  // send request
  // get reply
  // demarshal
};
```

Application:

```
// NASDAQ abbreviations for ORB vendors.
static const char *stocks[] =
{
  "IONAY" // IONA Orbix
  "INPR"  // Inprise VisiBroker
  "IBM"   // IBM Component Broker
}
// Set the max number of ORB stocks.
static const int MAX_STOCKS = 3;

// Make synchronous two-way calls.
  for (int i = 0; i < MAX_STOCKS; i++) {
    CORBA::Long value =
      quoter_ref->get_quote (stocks[i]);
    cout << "Current value of "
         << stocks[i] << " stock: "
         << value << endl;
  }
```


Example: AMI Callback Client



Asynchronous stub:

```
void
Stock::Quoter::sendc_get_quote
       (AMI_QuoterHandler_ptr,
        const char *name)
{
 // setup connection
 // marshal
 // send request
 // return
};
```

Reply Handler Servant

```
class My_Async_Stock_Handler
 : public POA_Stock::AMI_QuoterHandler {
public:
 My_Async_Stock_Handler (const char *stock)
   : stock_ (CORBA::string_dup (stock))
 { }
 ~My_Async_Stock_Handler (void) { }

 virtual void get_quote (CORBA::Long value) {
   cout << stock_ << " stock: "
        << value << endl;

   // Decrement global reply count.
    reply_count--;
  }

private:
  CORBA::String_var stock_;
};
```


Example: AMI Callback Client (cont'd)

```
// Global reply count
int reply_received;

// Servants.
My_Async_Stock_Handler *
  handlers[MAX_STOCKS];

// Objrefs.
Stock::AMI_QuoterHandler_var
  handler_refs[MAX_STOCKS];

int i;

// Initialize ReplyHandler
// servants.
for (i = 0; i < MAX_STOCKS; i++)
 handlers[i] = new
 My_Async_Stock_Handler (stocks[i]);
```

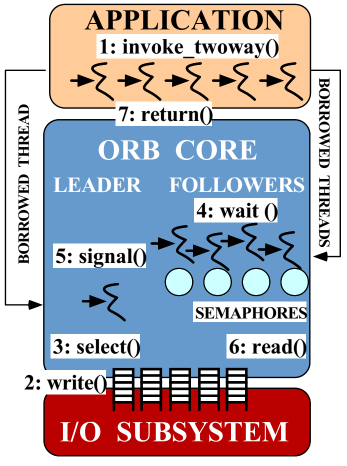
```
// Initialize ReplyHandler object refs.
for (i = 0; i < MAX_STOCKS; i++)
  handler_refs[i] = handlers[i]->_this ();

// Make asynchronous two-way calls using
// the callback model.
for (i = 0; i < MAX_STOCKS; i++)
  quoter_ref->sendc_get_quote
    (handler_refs[i],
     stocks[i]);

// Event loop to receive all replies
while (reply_count > 0)
  if (orb->work_pending ())
    orb->perform_work ();
```

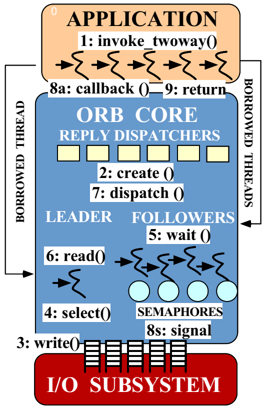

Problem: Who Processes an Asynchronous Reply

- SMI Architecture
  - Calling thread blocks for reply
  - ORB can use it to process reply
- AMI Architecture
  - No thread waits for asynchronous reply
- Forces
  - Preserve optimizations for SMI case



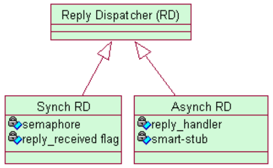
Leader/Follower

Solution → Reply Dispatching Strategy



New Leader/Follower

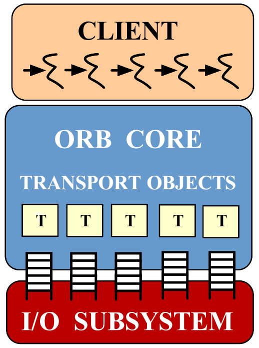
1. Current Leader thread reads reply
2. Invokes Reply Dispatcher



Strategize Reply Dispatchers

- Synch Dispatcher stores reply and wakeup waiting thread
- Asynch Dispatcher uses *smart-stub* to invoke the callback

Problem: Connection Utilization



Previous Architecture

- One outstanding request per-connection
- Reserved connections for hard real-time systems

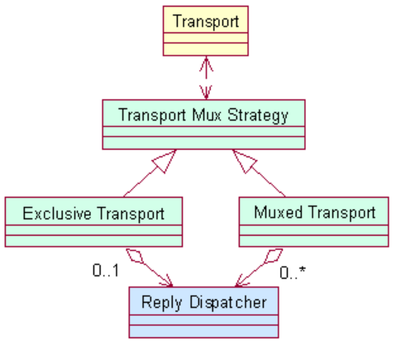
Suitability for AMI

- Request per-connection inefficient for AMI
- Allow multiple outstanding requests on a connection

Forces

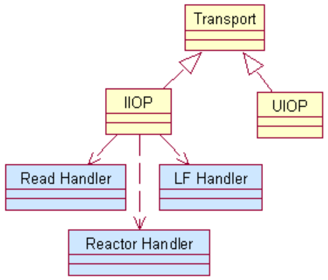
- Preserve existing optimizations, support connection multiplexing, be configurable

Solution → Transport Mux Strategy



- Strategy pattern
- Service configurator
- Exclusive Transport
  - Single `Reply Dispatcher` at a time
- Muxed Transport
  - Multiple `Reply Dispatchers` at a time

Problem: Wait Mechanisms



Forces

- Support AMI and SMI, preserve optimizations, be configurable

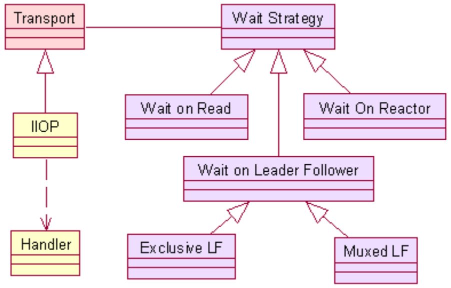
Wait mechanisms in TAO →

- Wait-on-Read
  - Optimized for hard real-time, no nested upcalls
- Wait-on-Reactor
  - Supports nested upcalls, single-threaded
- Wait-on-Leader/Follower
  - Supports nested upcalls, multi-threaded

Problems →

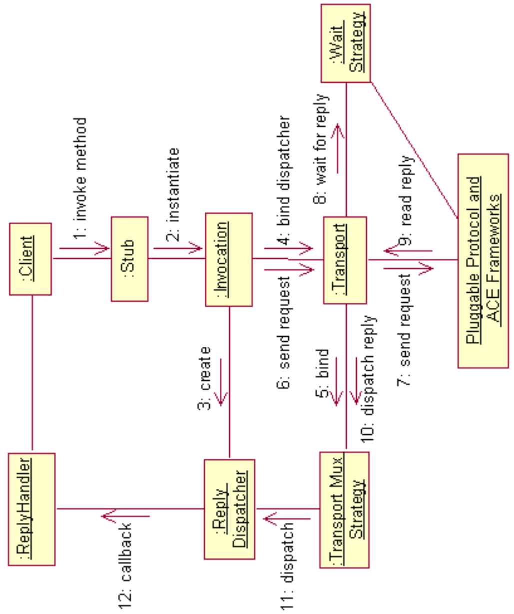
- Implemented in Connection Handlers
- Hard to scale to new protocols
- Hard to integrate AMI

Solution → Wait Strategy

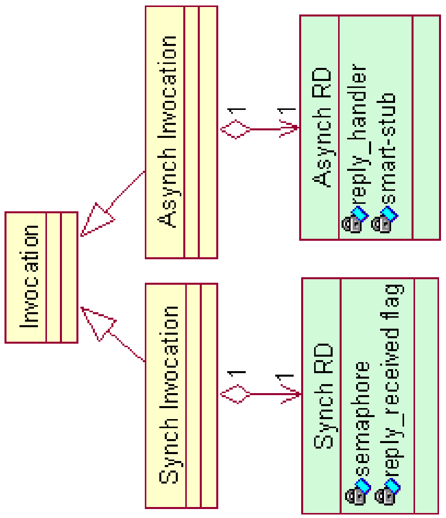


- Apply *Strategy* pattern
- Decouple from Connection Handlers
- Preserve existing strategies
- Integrate `Leader/Follower Wait Strategy` and `Muxed Transport`

Putting the Components Together in TAO

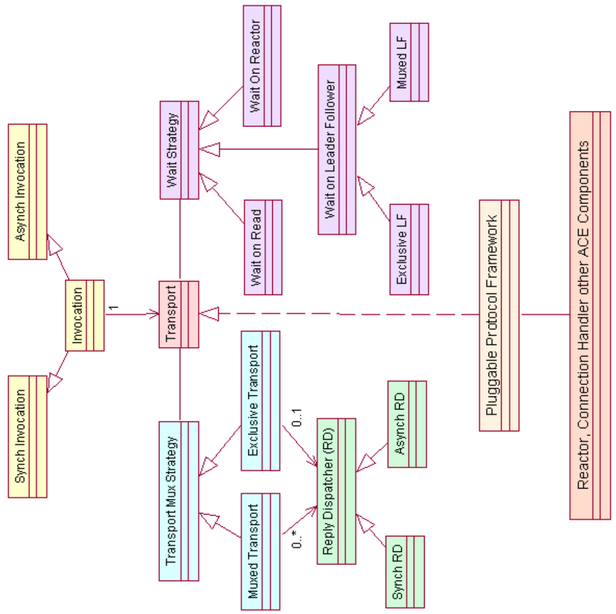


Invocation Facade

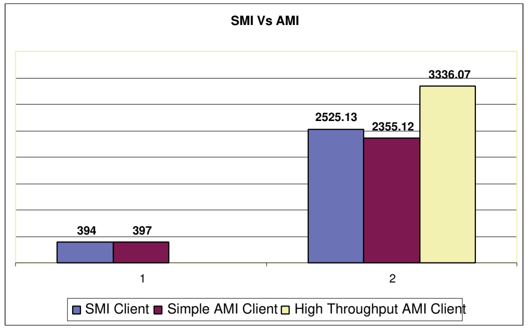


- Synch Dispatcher *on stack*
- Asynch Dispatcher *on heap*

AMI-enabled TAO ORB Architecture



Benchmarks



Client Applications

- Simple SMI Client
- Simple AMI Client
- High Performance AMI Client

Future Work

- Exception delivery in Callback
- Rest of the Messaging Spec.
  - Poller Model
  - Time Independant Invocations
  - QoS Policies
- Real-time Specification

Thanks

- Thanks Carlos for the excellent guidance!
- Michael Kircher for help in the IDL compiler changes
- Thanks Dr.Schmidt !
- Irfan for all the help
- DOC Group
- Friends