Design Patterns for Evolving System Software Components from unix to Windows nt

Douglas C. Schmidt

http://www.cs.wustl.edu/∼schmidt/

schmidt@cs.wustl.edu

Washington University, St. Louis

Motivation

- Developing *efficient*, *robust*, *extensible*, and *reusable* communication software is hard
- It is essential to understand successful techniques that have proven effective to solve common development challenges
- *Design patterns* and *frameworks* help to capture, articulate, and instantiate these successful techniques

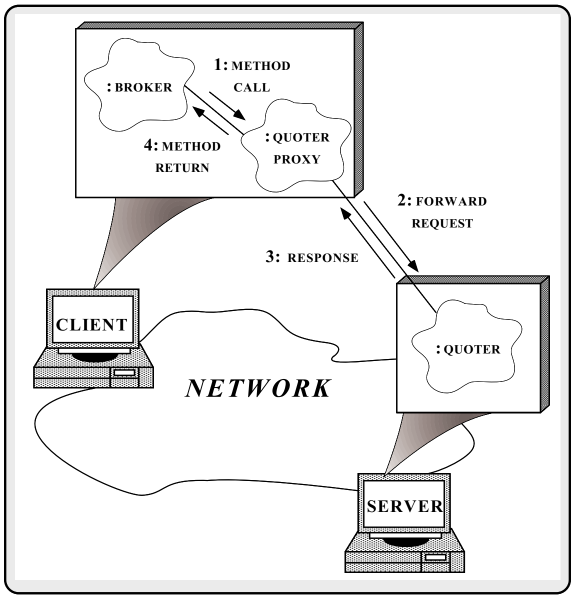
Observations

- Developers of communication software confront recurring challenges that are largely application-independent
  - *e.g.*, service initialization and distribution, error handling, flow control, event demultiplexing, concurrency control
- Successful developers resolve these challenges by applying appropriate *design patterns*
- However, these patterns have traditionally been either:
  1. *Locked inside heads of expert developers*
  2. *Buried in source code*

Design Patterns

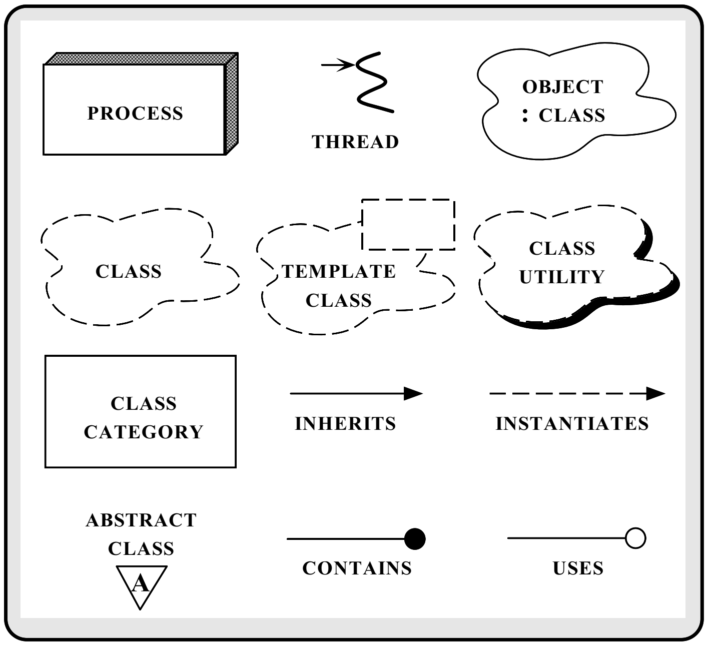
- Design patterns represent *solutions* to *problems* that arise when developing software within a particular *context*
  - *i.e.*, "Patterns == problem/solution pairs in a context"
- Patterns capture the static and dynamic *structure* and *collaboration* among key *participants* in software designs
  - They are particularly useful for articulating how and why to resolve *non-functional forces*
- Patterns facilitate reuse of successful software architectures and designs

Proxy Pattern



- *Intent*: provide a surrogate for another object that controls access to it

Graphical Notation



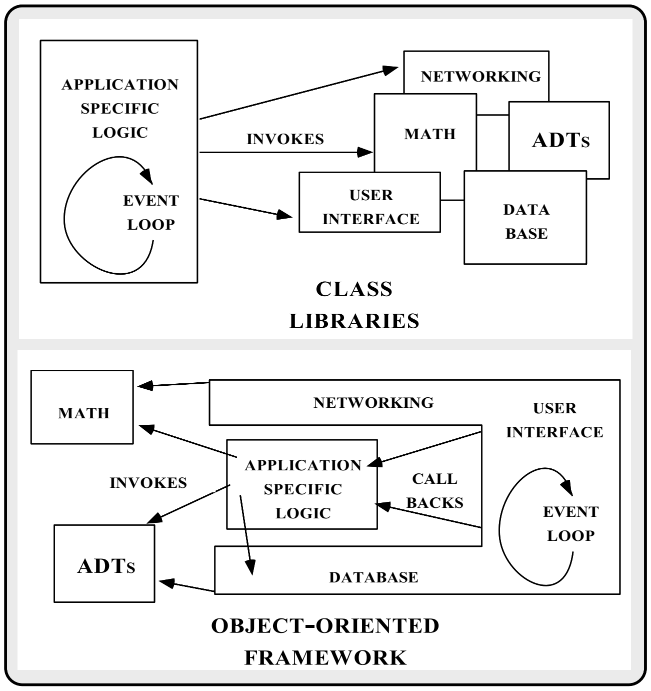
More Observations

- Reuse of patterns alone is not sufficient
  - Patterns enable reuse of architecture and design knowledge, but not code (directly)
- To be productive, developers must also reuse detailed designs, algorithms, interfaces, implementations, etc.
- Application *frameworks* are an effective way to achieve broad reuse of software

Frameworks

- A framework is:
  - "An integrated collection of components that collaborate to produce a reusable architecture for a family of related applications"
- Frameworks differ from conventional class libraries:
  1. Frameworks are "semi-complete" applications
  2. Frameworks address a particular application domain
  3. Frameworks provide "inversion of control"
- Typically, applications are developed by *inheriting* from and *instantiating* framework components

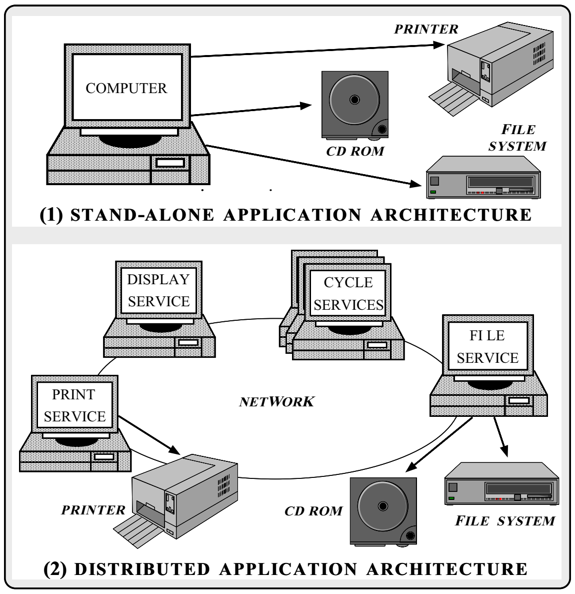
Differences Between Class Libraries and Frameworks



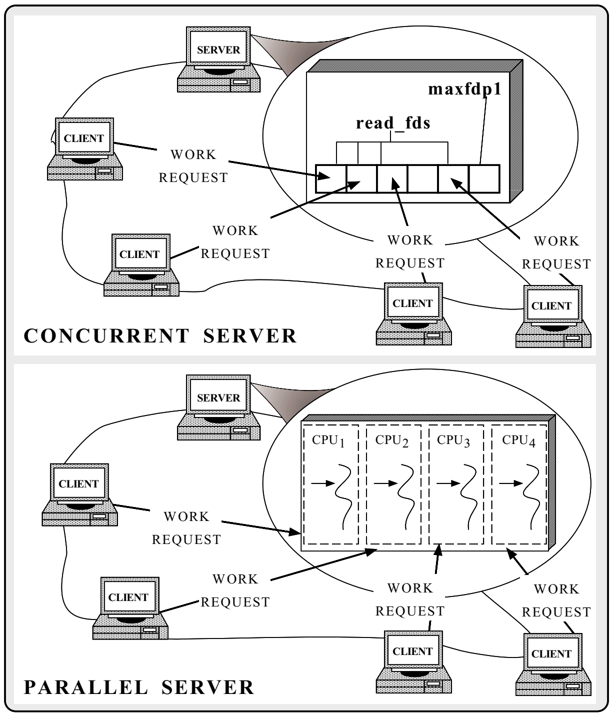
Tutorial Outline

- Outline key challenges for developing communication software
- Present the key reusable design patterns in a distributed medical imaging system
  - Both single-threaded and multi-threaded solutions are presented
- Discuss lessons learned from using patterns on production software systems

Stand-alone vs. Distributed Application Architectures



Concurrency vs. Parallelism



Sources of Complexity

- Distributed application development exhibits both *inherent* and *accidental* complexity
- *Inherent complexity* results from fundamental challenges, *e.g.*,
  - Distributed systems
    - ▷ *Latency*
    - ▷ *Error handling*
    - ▷ *Service partitioning and load balancing*
  - Concurrent systems
    - ▷ *Race conditions*
    - ▷ *Deadlock avoidance*
    - ▷ *Fair scheduling*
    - ▷ *Performance optimization and tuning*

Sources of Complexity (cont'd)

- *Accidental complexity* results from limitations with tools and techniques, *e.g.*,
  - Low-level tools
    - ▷ *e.g.*, Lack of type-secure, portable, re-entrant, and extensible system call interfaces and component libraries
  - Inadequate debugging support
  - Widespread use of *algorithmic* decomposition
    - ▷ Fine for *explaining* network programming concepts and algorithms but inadequate for *developing* large-scale distributed applications
  - Continuous rediscovery and reinvention of core concepts and components

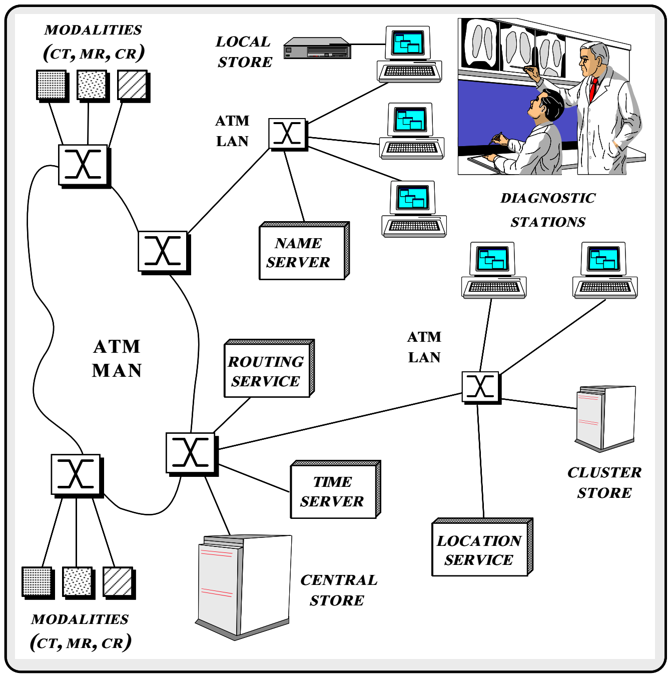
OO Contributions

- Concurrent and distributed programming has traditionally been performed using low-level OS mechanisms, *e.g.*,
  - *fork/exec*
  - *Shared memory*
  - *Signals*
  - *Sockets and select*
  - *POSIX pthreads, Solaris threads, Win32 threads*
- OO *design patterns* and *frameworks* elevate development to focus on application concerns, *e.g.*,
  - *Service functionality and policies*
  - *Service configuration*
  - *Concurrent event demultiplexing and event handler dispatching*
  - *Service concurrency and synchronization*

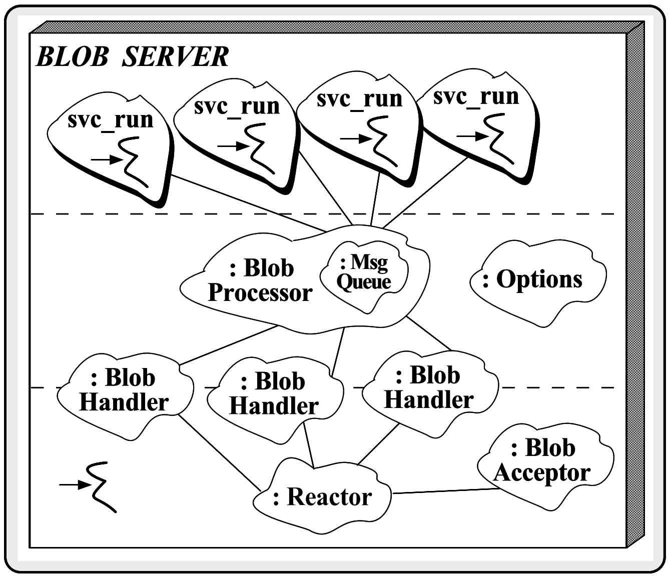
Distributed Medical Imaging Example

- This example illustrates the reusable *design patterns* and *framework* components used in an OO architecture for a *distributed medical imaging system*
- Application clients uses *Blob Servers* to store and retrieve medical images
- Clients and Servers communicate via a connection-oriented transport protocol
  - *e.g.*, TCP/IP, IPX/SPX, TP4

Distributed Electronic Medical Imaging Architecture

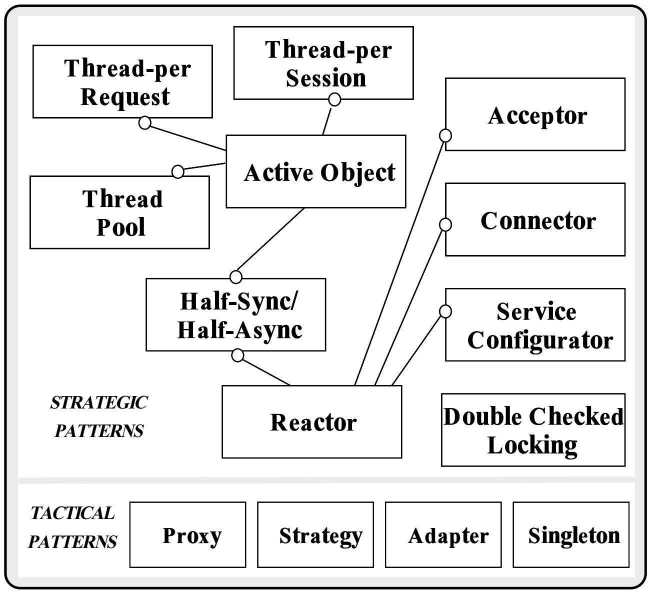


Architecture of the Blob Server



* *Manage short-term and long-term blob persistence*
* *Respond to queries from Blob Locators*

Design Patterns in the Blob Server



Tactical Patterns

- Proxy
  - "Provide a surrogate or placeholder for another object to control access to it"
- Strategy
  - "Define a family of algorithms, encapsulate each one, and make them interchangeable"
- Adapter
  - "Convert the interface of a class into another interface client expects"
- Singleton
  - "Ensure a class only has one instance and provide a global point of access to it"

Concurrency Patterns

- *Reactor*
  - "Decouples event demultiplexing and event handler dispatching from application services performed in response to events"
- *Active Object*
  - "Decouples method execution from method invocation and simplifies synchronized access to shared resources by concurrent threads"
- *Half-Sync/Half-Async*
  - "Decouples synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency"
- *Double-Checked Locking Pattern*
  - "Ensures atomic initialization of objects and eliminates unnecessary locking overhead on each access"

Concurrency Architecture Patterns

- *Thread-per-Request*
  - "Allows each client request to run concurrently"
- *Thread-Pool*
  - "Allows up to N requests to execute concurrently"
- *Thread-per-Session*
  - "Allows each client session to run concurrently"

Service Initialization Patterns

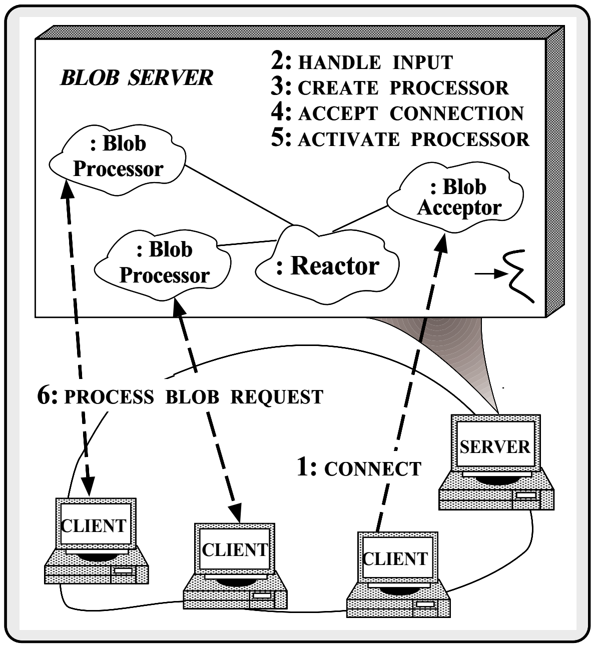
- *Connector*
  - "Decouples active connection establishment from the service performed once the connection is established"
- *Acceptor*
  - "Decouples passive connection establishment from the service performed once the connection is established"
- *Service Configurator*
  - "Decouples the behavior of network services from point in time at which services are configured into an application"

Concurrency Patterns in the Blob Server

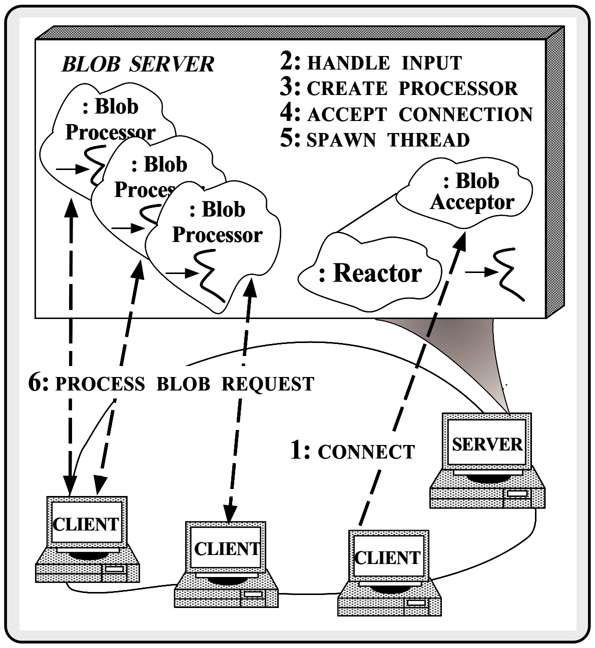
- The following example illustrates the *design patterns* and *framework components* in an OO implementation of a concurrent Blob Server
- There are various architectural patterns for structuring concurrency in a Blob Server

1. *Reactive*
2. *Thread-per-request*
3. *Thread-per-session*
4. *Thread-pool*

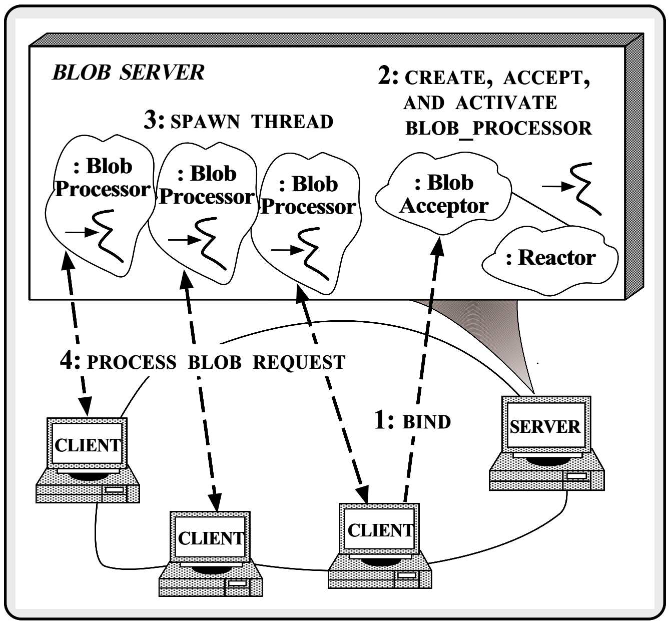
Reactive Blob Server Architecture



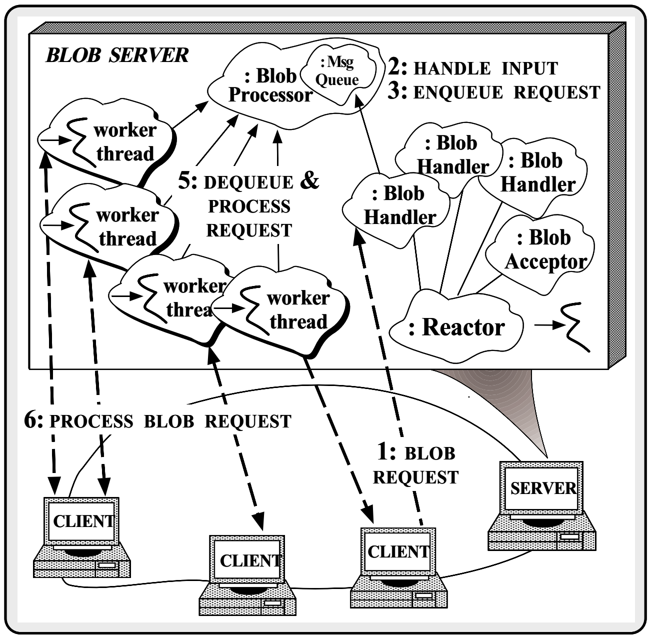
Thread-per-Request Blob Server Architecture



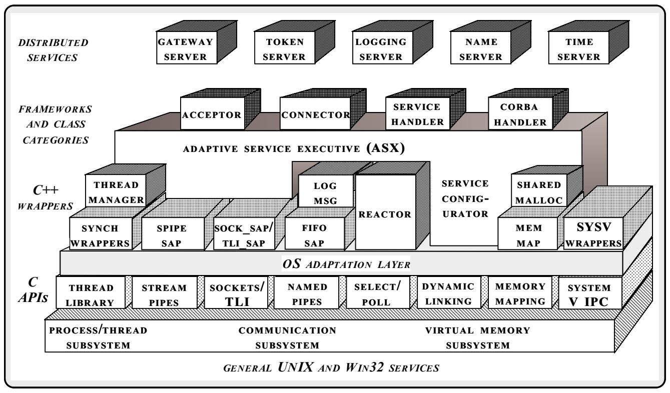
Thread-per-Session Blob Server Architecture



Thread-Pool Blob Server Architecture



The ADAPTIVE Communication Environment (ACE)

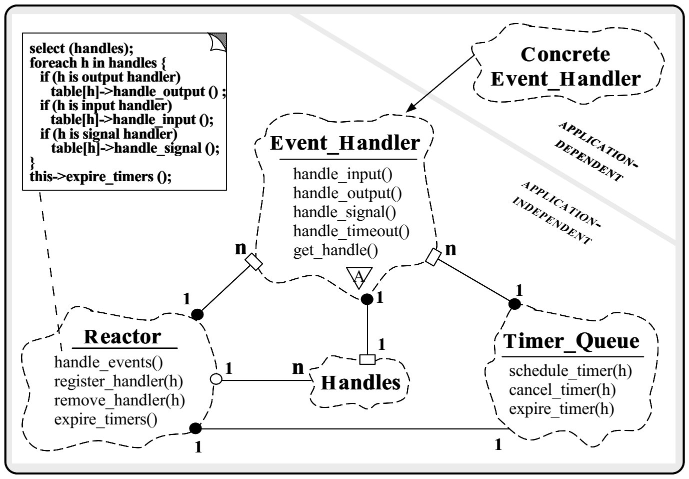


- A set of C++ wrappers and frameworks based on common design patterns

The Reactor Pattern

- *Intent*
  - "Decouples event demultiplexing and event handler dispatching from the services performed in response to events"
- This pattern resolves the following forces for event-driven software:
  - *How to demultiplex multiple types of events from multiple sources of events efficiently within a single thread of control*
  - *How to extend application behavior without requiring changes to the event dispatching framework*

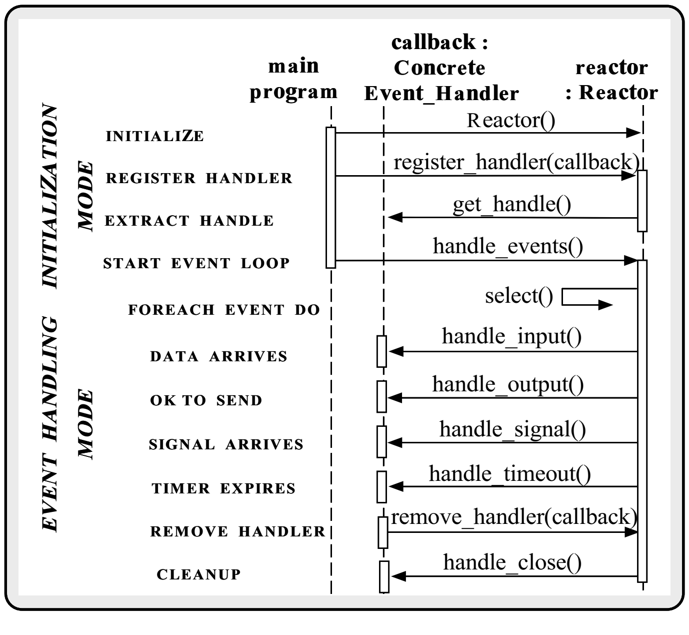
Structure of the Reactor Pattern



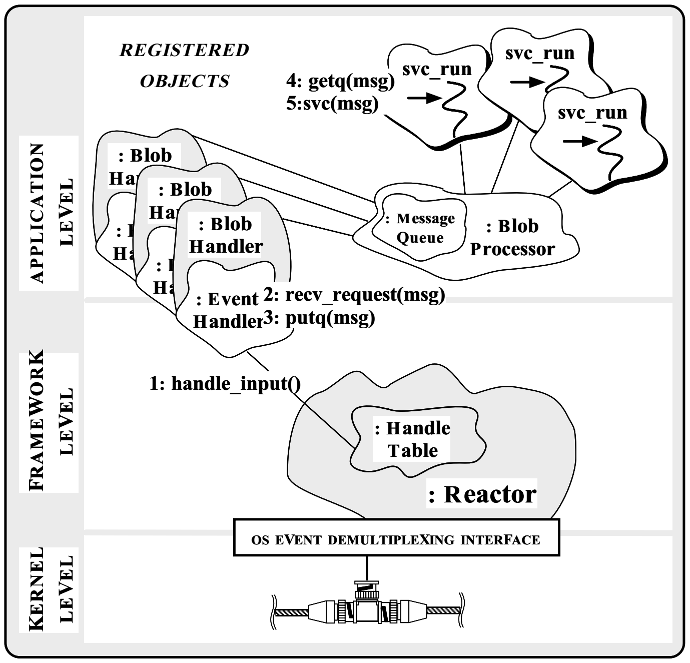
```
select (handles);
foreach h in handles {
  if (h is output handler)
    table[h]->handle_output () ;
  if (h is input handler)
    table[h]->handle_input ();
  if (h is signal handler)
    table[h]->handle_signal ();
}
this->expire_timers ();
```

- Participants in the Reactor pattern

Collaboration in the Reactor Pattern



Using the Reactor in the Blob Server



The Blob_Handler Interface

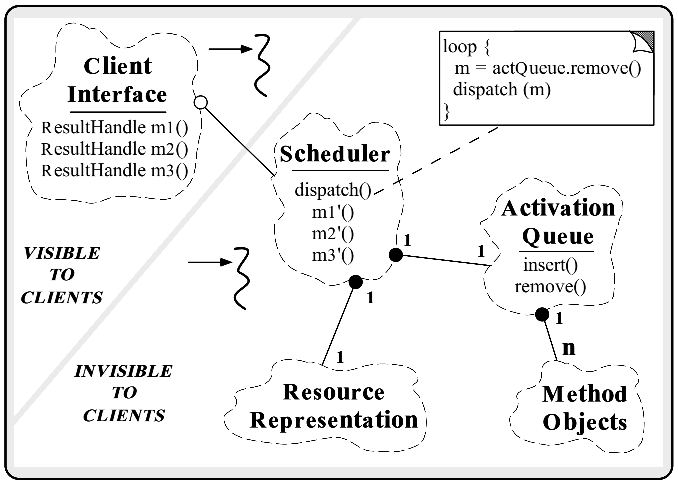
- The `Blob_Handler` is the Proxy for communicating with clients
  - Together with `Reactor`, it implements the asynchronous task portion of the Half-Sync/Half-Async pattern

```
// Reusable Svc Handler.
class Blob_Handler : public Event_Handler
{
public:
    // Entry point into Blob Handler.
  virtual int open (void) {
    // Register with Reactor to handle client input.
    Reactor::instance ()->register_handler
                           (this, READ_MASK);
  }

protected:
    // Notified by Reactor when client requests arrive.
  virtual int handle_input (void);

    // Receive and frame client requests.
  int recv_request (Message_Block &*);

  SOCK_Stream peer_stream_; // IPC endpoint.
};
```

The Active Object Pattern

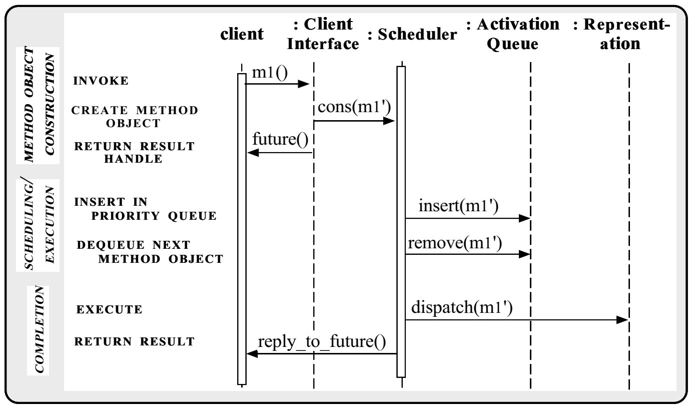
- *Intent*
  - "Decouples method execution from method invocation and simplifies synchronized access to shared resources by concurrent threads"
- This pattern resolves the following forces for concurrent communication software:
  - *How to allow blocking read and write operations on one endpoint that do not detract from the quality of service of other endpoints*
  - *How to simplify concurrent access to shared state*
  - *How to simplify composition of independent services*

Structure of the Active Object Pattern

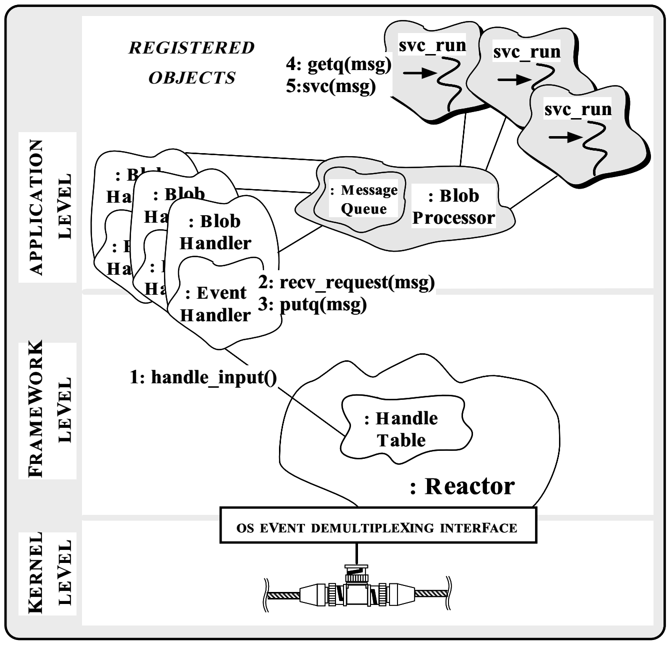


- The `Scheduler` determines the sequence that `Method Objects` are executed

Collaboration in the Active Object Pattern



Using the Active Object Pattern in the Blob Server



The Blob_Processor Class

- Processes Blob requests using the "Thread-Pool" concurrency model
  - Implement the synchronous task portion of the Half-Sync/Half-Async pattern

```
class Blob_Processor : public Task {
public:
    // Singleton access point.
  static Blob_Processor *instance (void);

    // Pass a request to the thread pool.
  virtual put (Message_Block *);

    // Event loop for the pool thread
  virtual int svc (int) {
    Message_Block *mb = 0; // Message buffer.

    // Wait for messages to arrive.
    for (;;) {
      getq (mb); // Inherited from class Task;
      // Identify and perform Blob Server
      // request processing here...

protected:
  Blob_Processor (void); // Constructor.
```

Using the Singleton Pattern

- The `Blob_Processor` is implemented as a Singleton that is created "on demand"

```
Blob_Processor *
Blob_Processor::instance (void) {
  // Beware race conditions!
  if (instance_ == 0) {
    instance_ = new Blob_Processor;
  }
  return instance_;
}
```

- Constructor creates the thread pool

```
Blob_Processor::Blob_Processor (void) {
  Thread_Manager::instance ()->spawn_n
    (num_threads, THR_FUNC (svc_run),
     (void *) this, THR_NEW_LWP);
}
```

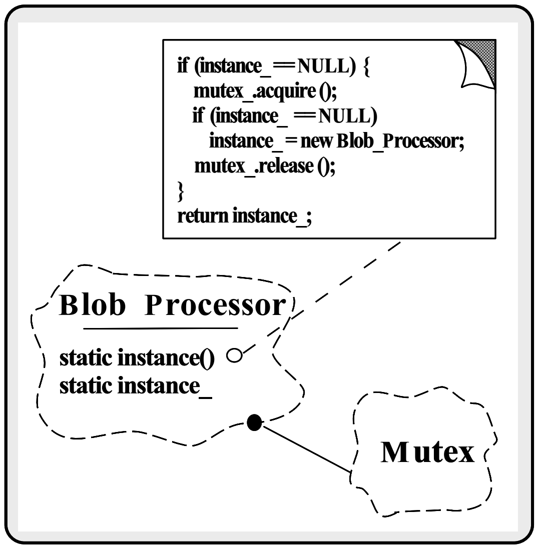
The Double-Checked Locking Pattern

- *Intent*
  - "Ensures atomic initialization of objects and eliminates unnecessary locking overhead on each access"

- This pattern resolves the following forces:
  1. *Ensures atomic initialization or access to objects, regardless of thread scheduling order*
  2. *Keeps locking overhead to a minimum*
     - *e.g.*, only lock on first access

- Note, this pattern assumes atomic memory access...

Using the Double-Checked Locking Pattern for the Blob Server



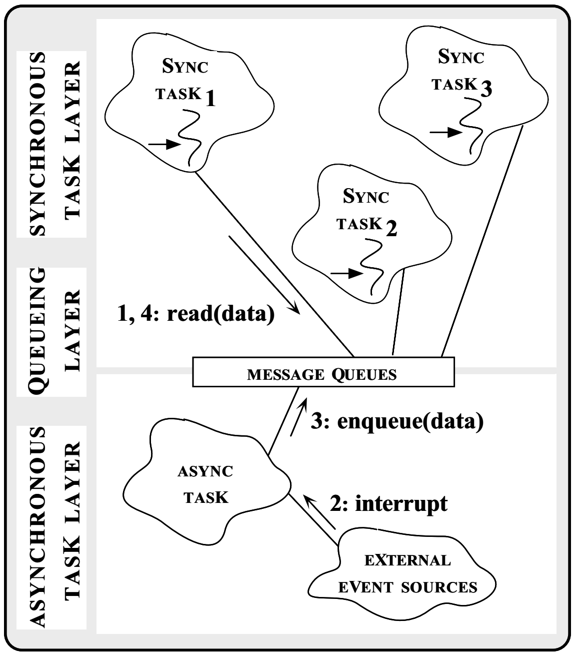
```
if (instance_==NULL) {
  mutex_.acquire ();
  if (instance_ ==NULL)
    instance_= new Blob_Processor;
  mutex_.release ();
}
return instance_;
```

Half-Sync/Half-Async Pattern

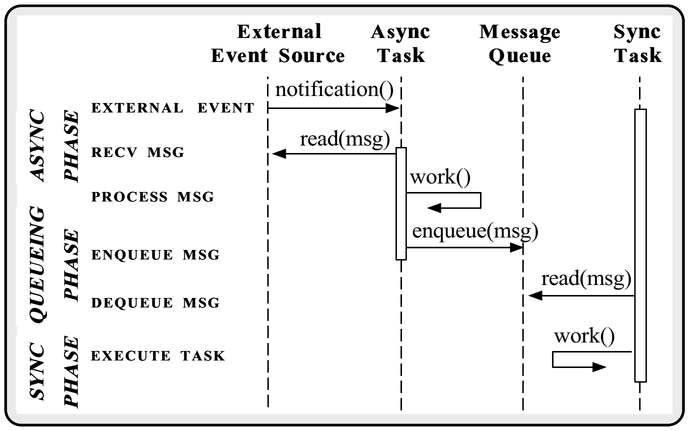
- *Intent*
  - "Decouples synchronous I/O from asynchronous I/O in a system to simplify programming effort without degrading execution efficiency"

- This pattern resolves the following forces for concurrent communication systems:
  - *How to simplify programming for higher-level communication tasks*
    - ▷ These are performed synchronously
  - *How to ensure efficient lower-level I/O communication tasks*
    - ▷ These are performed asynchronously

Structure of the Half-Sync/Half-Async Pattern

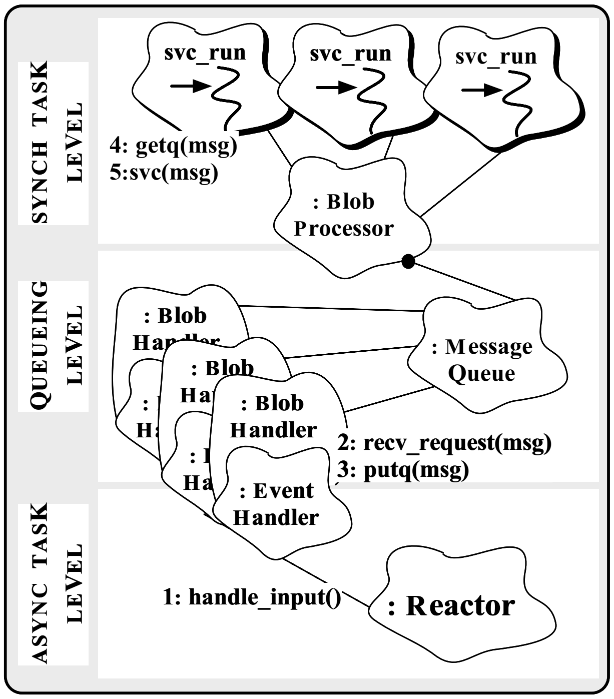


Collaborations in the Half-Sync/Half-Async Pattern



- This illustrates *input* processing (*output* processing is similar)

Using the Half-Sync/Half-Async Pattern in the Blob Server



Joining Async and Sync Tasks in the Blob Server

- The following methods form the boundary between the Async and Sync layers

```
int
Blob_Handler::handle_input (void)
{
  Message_Block *mb = 0;

  // Receive and frame message
  // (uses peer_stream_).
  recv_request (mb);

  // Insert message into the Queue.
  Blob_Processor::instance ()->put (mb);
}

// Task entry point.
Blob_Processor::put (Message_Block *msg)
{
  // Insert the message on the Message_Queue
  // (inherited from class Task).
  putq (msg);
}
```

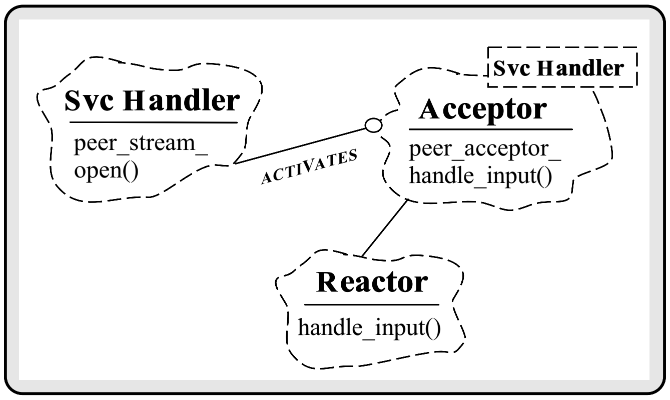
The Acceptor Pattern

- *Intent*
  - "Decouples passive initialization of a service from the tasks performed once the service is initialized"

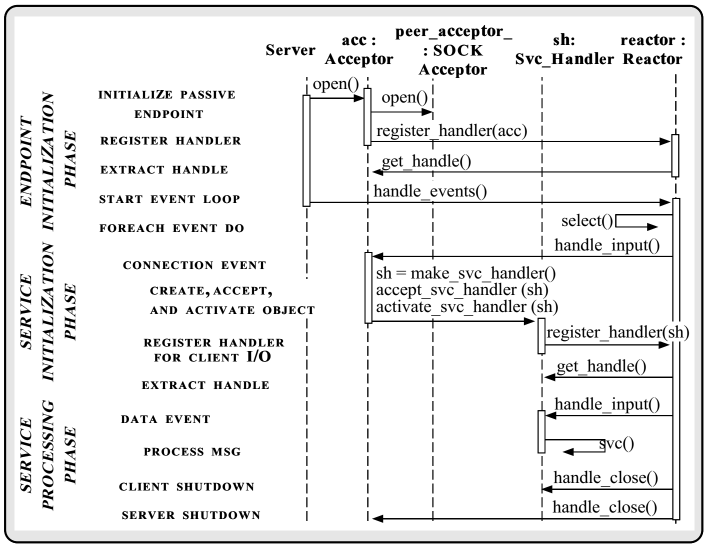
- This pattern resolves the following forces for network servers using interfaces like sockets or TLI:

1. *How to reuse passive connection establishment code for each new service*
2. *How to make the connection establishment code portable across platforms that may contain sockets but not TLI, or vice versa*
3. *How to ensure that a passive-mode descriptor is not accidentally used to read or write data*
4. *How to enable flexible policies for creation, connection establishment, and concurrency*

Structure of the Acceptor Pattern

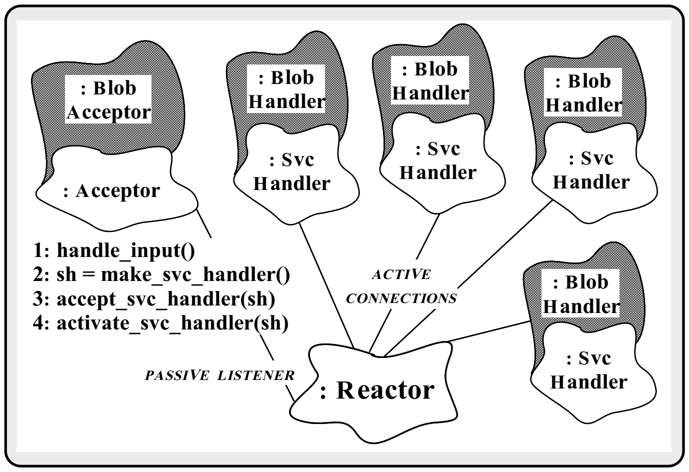


Collaboration in the Acceptor Pattern



- `Acceptor` is a factory that creates, connects, and activates a `Svc_Handler`

Using the Acceptor Pattern in the Blob Server



The Acceptor Class

- The `Acceptor` class implements the Acceptor pattern

```
// Reusable Factor
template <class SVC_HANDLER>
class Acceptor :
  public Service_Object // Subclass of Event_Handler.
{
public:
    // Notified by Reactor when clients connect.
  virtual int handle_input (void)
  {
    // The strategy for initializing a SVC_HANDLER.
    SVC_HANDLER *sh = new SVC_HANDLER;
    peer_acceptor_.accept (*sh);
    sh->open ();
  }
  // ...

protected:
    // IPC connection factory.
  SOCK_Acceptor peer_acceptor_;
}
```

The Blob_Acceptor Class Interface

- The `Blob_Acceptor` class accepts connections and initializes `Blob_Handlers`

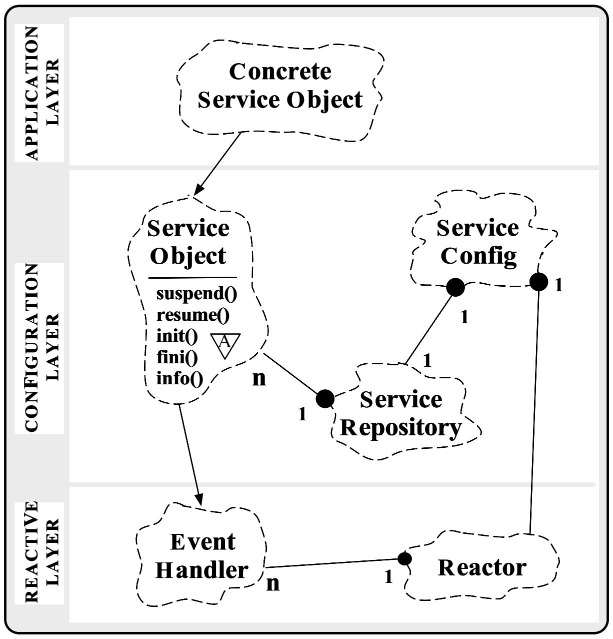
```
class Blob_Acceptor
  : public Acceptor<Blob_Handler>
  // Inherits handle_input() strategy from Acceptor.
{
public:
    // Called when Blob_Acceptor is dynamically linked.
  virtual int init (int argc, char *argv);

    // Called when Blob_Acceptor is dynamically unlinked.
  virtual int fini (void);
```

The Service Configurator Pattern

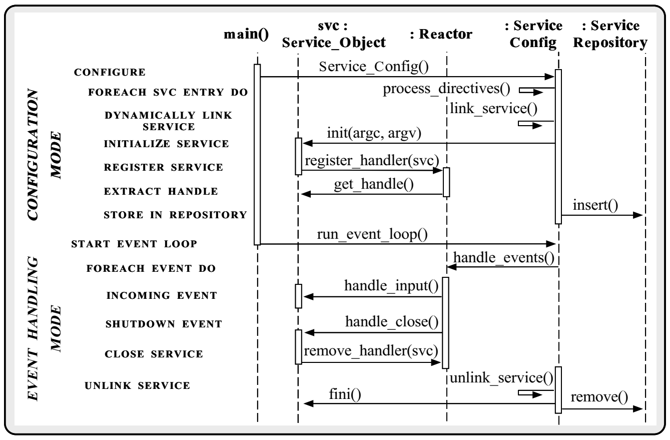
- *Intent*
  - "Decouples the behavior of communication services from the point in time at which these services are configured into an application or system"

- This pattern resolves the following forces for highly flexible communication software:
  - *How to defer the selection of a particular type, or a particular implementation, of a service until very late in the design cycle*
    - ▷ *i.e.*, at installation-time or run-time
  - *How to build complete applications by composing multiple independently developed services*
  - *How to optimize, reconfigure, and control the behavior of the service at run-time*

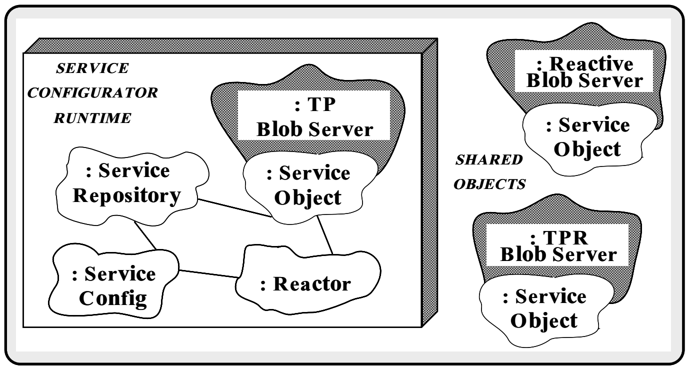
Structure of the Service Configurator Pattern



Collaboration in the Service Configurator Pattern



Using the Service Configurator Pattern in the Blob Server



- Existing service is based on Half-Sync/Half-Async pattern
- Other versions could be single-threaded or use other concurrency strategies...

The Blob_Acceptor Class Implementation

```
// Initialize service when dynamically linked.

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

  // Set the endpoint into listener mode.
  Acceptor::open (local_addr);

  // Initialize the communication endpoint.
  Reactor::instance ()->register_handler
                        (this, READ_MASK)
}

// Terminate service when dynamically unlinked.

int Blob_Acceptor::fini (void)
{
  // Unblock threads in the pool so they will
  // shutdown correctly.
  Blob_Processor::instance ()->close ();

  // Wait for all threads to exit.
  Thread_Manager::instance ()->wait ();
}
```

Configuring the Blob Server with the Service Configurator

- The concurrent Blob Server is configured and initialized via a configuration script

```
% cat ./svc.conf
dynamic TP_Blob_Server Service_Object *
        blob_server.dll:make_TP_Blob_Server()
        "-p $PORT -t $THREADS"
```

- Factory function that dynamically allocates a Half-Sync/Half-Async Blob_Server object

```
extern "C" Service_Object *make_TP_Blob_Server (void);

Service_Object *make_TP_Blob_Server (void)
{
  return new Blob_Acceptor;
  // ACE dynamically unlinks and deallocates this object.
}
```

Main Program for Blob Server

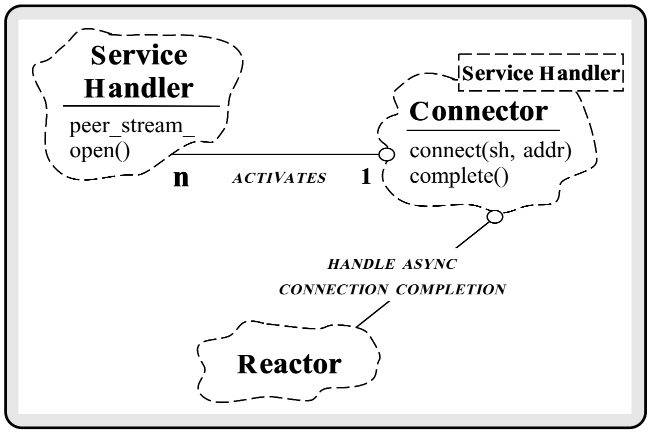
- Dynamically configure and execute the Blob Server
  - Note that this is totally generic!

```
int main (int argc, char *argv[])
{
  Service_Config daemon;

  // Initialize the daemon and dynamically
  // configure the service.
  daemon.open (argc, argv);

  // Loop forever, running services and handling
  // reconfigurations.

  daemon.run_event_loop ();

  /* NOTREACHED */
}
```

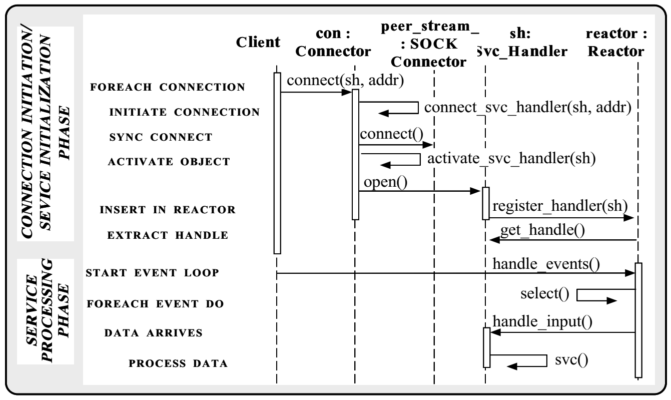
The Connector Pattern

- *Intent*
  - "Decouples active initialization of a service from the task performed once a service is initialized"
- This pattern resolves the following forces for network clients that use interfaces like sockets or TLI:
  1. *How to reuse active connection establishment code for each new service*
  2. *How to make the connection establishment code portable across platforms that may contain sockets but not TLI, or vice versa*
  3. *How to enable flexible service concurrency policies*
  4. *How to actively establish connections with large number of peers efficiently*

Structure of the Connector Pattern

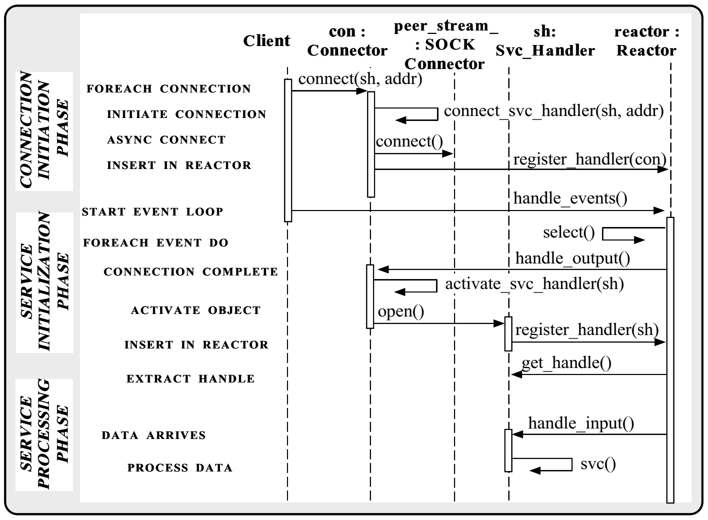


Collaboration in the Connector Pattern



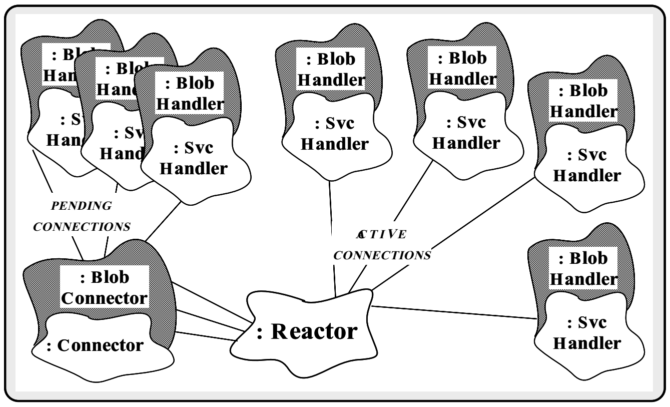
- Synchronous mode

Collaboration in the Connector Pattern



- Asynchronous mode

Using the Connector in the Blob Clients



Benefits of Design Patterns

- *Design patterns enable large-scale reuse of software architectures*
- *Patterns explicitly capture expert knowledge and design tradeoffs*
- *Patterns help improve developer communication*
- *Patterns help ease the transition to object-oriented technology*

Drawbacks to Design Patterns

- *Patterns do not lead to direct code reuse*
- *Patterns are deceptively simple*
- *Teams may suffer from pattern overload*
- *Patterns are validated by experience and discussion rather than by automated testing*
- *Integrating patterns into a software development process is a human-intensive activity*

Suggestions for Using Patterns Effectively

- *Do not recast everything as a pattern*
  - Instead, develop strategic domain patterns and reuse existing tactical patterns
- *Institutionalize rewards for developing patterns*
- *Directly involve pattern authors with application developers and domain experts*
- *Clearly document when patterns apply and do not apply*
- *Manage expectations carefully*

Books and Magazines on Patterns

- *Books*
  - Gamma et al., "Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Wesley, Reading, MA, 1994.
  - "Pattern Languages of Program Design," editors James O. Coplien and Douglas C. Schmidt, Addison-Wesley, Reading, MA, 1995
- *Special Issues in Journals*
  - "Theory and Practice of Object Systems" (guest editor: Stephen P. Berczuk)
  - "Communications of the ACM" (guest editors: Douglas C. Schmidt, Ralph Johnson, and Mohamed Fayad)
- *Magazines*
  - C++ Report and Journal of Object-Oriented Programming, columns by Coplien, Vlissides, and De Souza

Conferences and Workshops on Patterns

- 1st EuroPLoP
  - July 10–14, 1996, Kloster Irsee, Germany
- 3rd Pattern Languages of Programs Conference
  - September 4–6, 1996, Monticello, Illinois, USA
- Relevant WWW URLs

http://www.cs.wustl.edu/~schmidt/jointPLoP–96.html/
http://st-www.cs.uiuc.edu/users/patterns/patterns.html

Obtaining ACE

- The ADAPTIVE Communication Environment (ACE) is an OO toolkit designed according to key network programming patterns
- All source code for ACE is freely available
  - Anonymously ftp to `wuarchive.wustl.edu`
  - Transfer the files `/languages/c++/ACE/*.gz` and `gnu/ACE-documentation/*.gz`
- Mailing lists

* ace-users@cs.wustl.edu
* ace-users-request@cs.wustl.edu
* ace-announce@cs.wustl.edu
* ace-announce-request@cs.wustl.edu

- WWW URL
  - http://www.cs.wustl.edu/~schmidt/ACE.html