Concurrent Object-Oriented Network Programming with C++

Douglas C. Schmidt

Washington University, St. Louis

http://www.cs.wustl.edu/~schmidt/

schmidt@cs.wustl.edu

Motivation for Distribution

- Benefits of distributed computing:
  - Collaboration → *connectivity* and *interworking*
  - Performance → *multi-processing* and *locality*
  - Reliability and availability → *replication*
  - Scalability and portability → *modularity*
  - Extensibility → *dynamic configuration and reconfiguration*
  - Cost effectiveness → *open systems* and *resource sharing*

Challenges and Solutions

- Developing *efficient*, *robust*, and *extensible* distributed applications is challenging
  - *e.g.*, must address complex topics that are less problematic or not relevant for non-distributed and non-concurrent applications
- Object-oriented (OO) techniques and C++ language features enhance distributed software quality factors
  - Key OO techniques → *design patterns* and *frameworks*
  - Key C++ language features → *classes*, *inheritance*, *dynamic binding*, and *parameterized types*
  - Key software quality factors → *modularity*, *extensibility*, *portability*, *reusability*, *reliability*, and *correctness*

Caveats

- OO and C++ are *not* a panacea
  - However, when used properly they help minimize "accidental" complexity and improve software quality
- Advanced OS features provide additional functionality and performance, *e.g.*,
  - *Multi-threading*
  - *Multi-processing*
  - *Synchronization*
  - *Explicit dynamic linking*
  - *Shared memory*
  - *Communication protocols and IPC mechanisms*

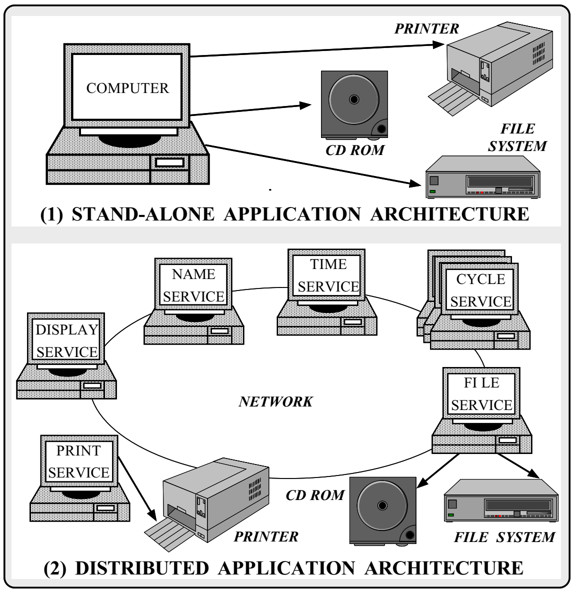
Tutorial Outline

- Outline key challenges for developing distributed applications
- Present an OO design and implementation of the following communication applications:
  1. *Distributed Logger*
  2. *Application-level Gateway*
- Both single-threaded and multi-threaded solutions are given

Software Development Environment

- The topics discussed here are largely independent of OS, network, and programming language
  - Currently being used on UNIX and Windows NT platforms, running on TCP/IP and IPX/SPX networks, written in C++
- Examples illustrated with freely available ADAPTIVE Communication Environment (ACE) OO toolkit
  - Although ACE is written in C++, the principles apply to other OO languages

Stand-alone vs. Distributed Application Architectures



Sources of Complexity

- Distributed application development exhibits both *inherent* and *accidental* complexity
- *Inherent complexity* results from fundamental challenges in the distributed application domain, *e.g.*,
  - Addressing the impact of latency
  - Detecting and recovering from partial failures of networks and hosts
  - Load balancing and service partitioning
  - Consistent ordering of distributed events

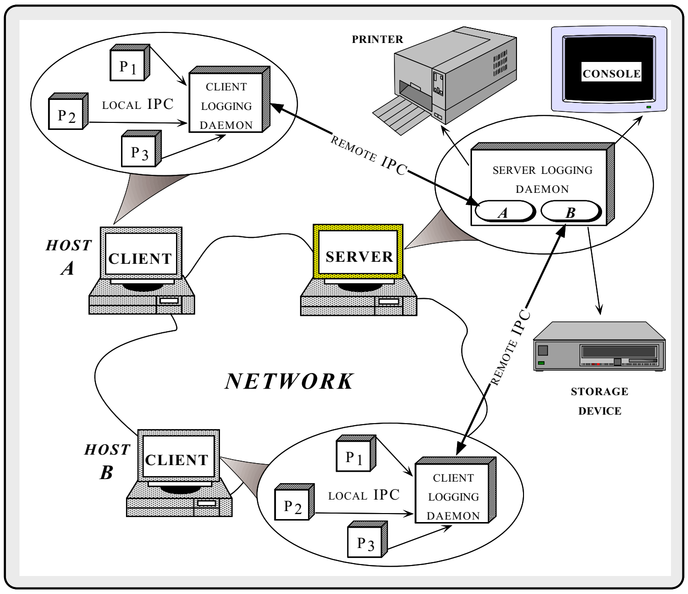
Sources of Complexity (cont'd)

- *Accidental complexity* results from limitations with tools and techniques used to develop distributed applications, *e.g.*,
  - Lack of type-safe, 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 to Distributed Application Development

- Distributed application implementors traditionally used low-level IPC and OS mechanisms, *e.g.*,
  - *sockets* → I/O descriptors, address formats, byte-ordering, signals, message buffering and framing
  - *select* → bitmasks, signals, descriptor counts, timeouts
  - *fork/exec* → signal handling, I/O descriptors
  - *POSIX pthreads and Solaris threads*
- OO *design patterns* and *frameworks* elevate focus onto on application requirements and policies, *e.g.*,
  - *Service functionality*
  - *Service configuration*
  - *Service concurrency*

Example Distributed Application



- *Distributed logging service*

Distributed Logging Service

- *Server logging daemon*
  - Collects, formats, and outputs logging records forwarded from *client logging daemons* residing throughout a network or internetwork
- The application interface is similar to `printf`

```
ACE_ERROR ((LM_ERROR, "(%t) can't fork in function spawn"));

// generates on server host

Oct 29 14:50:13 1992@crimee.ics.uci.edu@22766@7@client
::(4) can't fork in function spawn


ACE_DEBUG ((LM_DEBUG,
          "(%t) sending to server %s", server_host));

// generates on server host

Oct 29 14:50:28 1992@zola.ics.uci.edu@18352@2@drwho
::(6) sending to server bastille
```

Conventional Logging Server Design

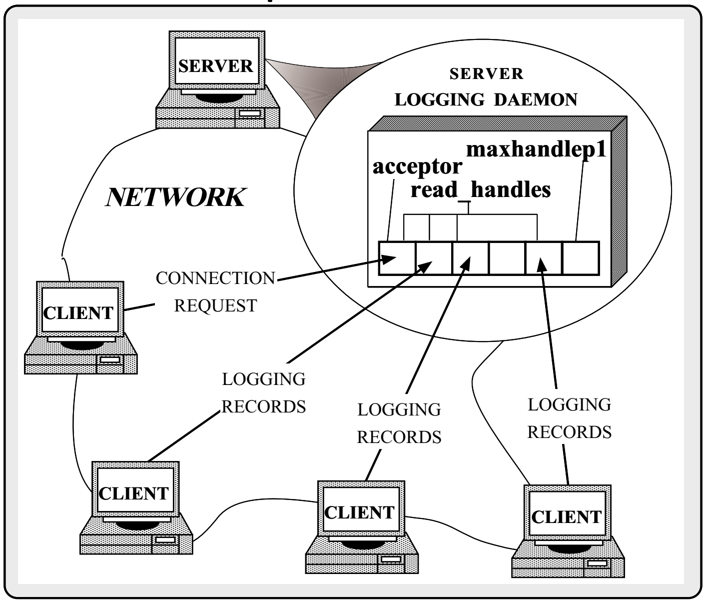
- Typical algorithmic pseudo-code for the server daemon portion of the distributed logging service:

void server_logging_daemon (**void**)
{
    initialize listener endpoint

    loop forever
    {
        wait for events
        handle data events
        handle connection events
    }
}

- The "grand mistake":
  - Avoid the temptation to "step-wise refine" this algorithmically decomposed pseudo-code directly into the detailed design and implementation of the logging server!

Select-based Logging Server Implementation



Conventional Logging Server Implementation

- Note the excessive amount of detail required to program at the socket level...

```
// Main program
static const int PORT = 10000;

typedef u_long COUNTER;
typedef int HANDLE;

// Counts the # of logging records processed
static COUNTER request_count;

// Passive-mode socket descriptor
static HANDLE listener;

// Highest active descriptor number, plus 1
static HANDLE maxhp1;

// Set of currently active descriptors
static fd_set read_handles;

// Scratch copy of read_handles
static fd_set tmp_handles;
```



```
// Run main event loop of server logging daemon.

int main (int argc, char *argv[])
{
  initialize_listener_endpoint
    (argc > 1 ? atoi (argv[1]) : PORT);

  // Loop forever performing logging server processing.

  for (;;) {
    tmp_handles = read_handles; // struct assignment.

    // Wait for client I/O events
    select (maxhp1, &tmp_handles, 0, 0, 0);

    // First receive pending logging records
    handle_data ();

    // Then accept pending connections
    handle_connections ();
  }
}
```

```
// Initialize the passive-mode socket descriptor

static void initialize_listener_endpoint (u_short port)
{
  struct sockaddr_in saddr;

  // Create a local endpoint of communication
  listener = socket (PF_INET, SOCK_STREAM, 0);

  // Set up the address information to become a server
  memset ((void *) &saddr, 0, sizeof saddr);
  saddr.sin_family = AF_INET;
  saddr.sin_port = htons (port);
  saddr.sin_addr.s_addr = htonl (INADDR_ANY);

  // Associate address with endpoint
  bind (listener, (struct sockaddr *) &saddr, sizeof saddr);

  // Make endpoint listen for connection requests
  listen (listener, 5);

  // Initialize descriptor sets
  FD_ZERO (&tmp_handles);
  FD_ZERO (&read_handles);
  FD_SET (listener, &read_handles);

  maxhp1 = listener + 1;
}
```



```
// Receive pending logging records

static void handle_data (void)
{
  // listener + 1 is the lowest client descriptor

  for (HANDLE h = listener + 1; h < maxhp1; h++)
    if (FD_ISSET (h, &tmp_handles)) {
      ssize_t n;

      // Guaranteed not to block in this case!
      if ((n = handle_log_record (h, 1)) > 0)
        ++request_count; // Count the # of logging records

      else if (n == 0) { // Handle connection shutdown.
        FD_CLR (h, &read_handles);
        close (h);

        if (h + 1 == maxhp1) {

          // Skip past unused descriptors

          while (!FD_ISSET (--h, &read_handles))
            continue;

          maxhp1 = h + 1;
        }
      }
    }
}
```



```
// Receive and process logging records

static ssize_t handle_log_record
  (HANDLE in_h, HANDLE out_h)
{
  ssize_t n;
  size_t len;
  Log_Record log_record;

  // The first recv reads the length (stored as a
  // fixed-size integer) of adjacent logging record.

  n = recv (in_h, (char *) &len, sizeof len, 0);

  if (n <= 0) return n;

  len = ntohl (len); // Convert byte-ordering

  // The second recv then reads LEN bytes to obtain the
  // actual record
  for (size_t nread = 0; nread < len; nread += n
    n = recv (in_h, ((char *) &log_record) + nread,
                  len - nread, 0);

  // Decode and print record.
  decode_log_record (&log_record);
  write (out_h, log_record.buf, log_record.size);
  return n;
}
```



```
// Check if any connection requests have arrived

static void handle_connections (void)
{
  if (FD_ISSET (listener, &tmp_handles)) {
    static struct timeval poll_tv = {0, 0};
    HANDLE h;

    // Handle all pending connection requests
    // (note use of select's "polling" feature)

    do {
      h = accept (listener, 0, 0);
      FD_SET (h, &read_handles);

      // Grow max. socket descriptor if necessary.
      if (h >= maxhp1)
        maxhp1 = h + 1;
    } while (select (listener + 1, &tmp_handles,
                  0, 0, &poll_tv) == 1);
}
```

Limitations with Algorithmic Decomposition Techniques

- Algorithmic decomposition tightly couples application-specific *functionality* and the following configuration-related characteristics:
  - **Structure**
    - ▷ The number of services per process
    - ▷ Time when services are configured into a process
  - **Communication Mechanisms**
    - ▷ The underlying IPC mechanisms that communicate with other participating clients and servers
    - ▷ Event demultiplexing and event handler dispatching mechanisms
  - **Execution Agents**
    - ▷ The process and/or thread architecture that executes service(s) at run-time

Overcoming Limitations via OO

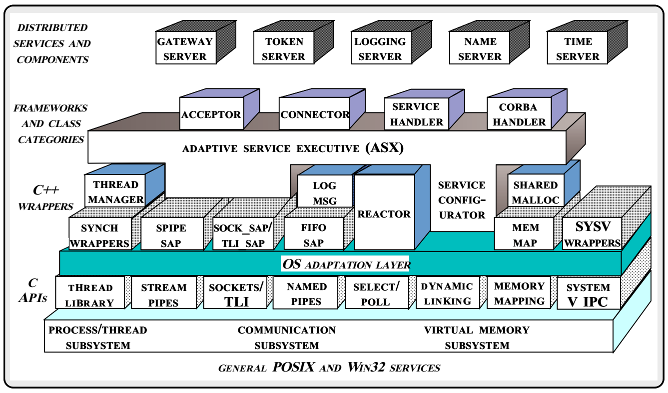
- The algorithmic decomposition illustrated above specifies *many* low-level details
  - Furthermore, the excessive coupling significantly complicates reusability, extensibility, and portability...
- In contrast, OO focuses on *application-specific* behavior, *e.g.*,

```
int Logging_Handler::svc (void)
{
  ssize_t n;

  n = handle_log_record (peer ().get_handle (), 1);
  if (n > 0)
    ++request_count; // Count the # of logging records
  return n <= 0 ? -1 : 0;
}
```


OO Contributions

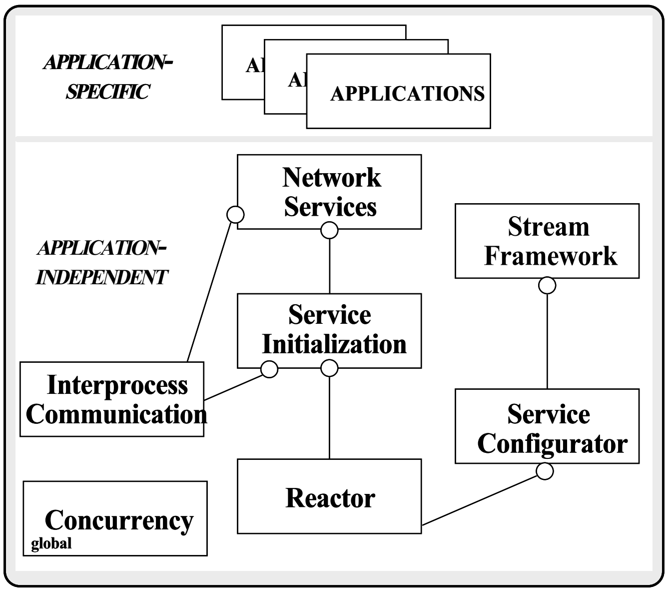
- *OO application frameworks* achieve large-scale design and code reuse
  - Frameworks emphasize the collaboration among societies of *classes* and *objects* in a domain
  - In contrast, traditional techniques focus on the *functions* and *algorithms* that solve particular requirements
- *OO design patterns* facilitate the large-scale reuse of software architecture
  - Even when reuse of algorithms, detailed designs, and implementations is not feasible

The ADAPTIVE Communication Environment (ACE)



- A set of C++ wrappers, class categories, and frameworks based on design patterns

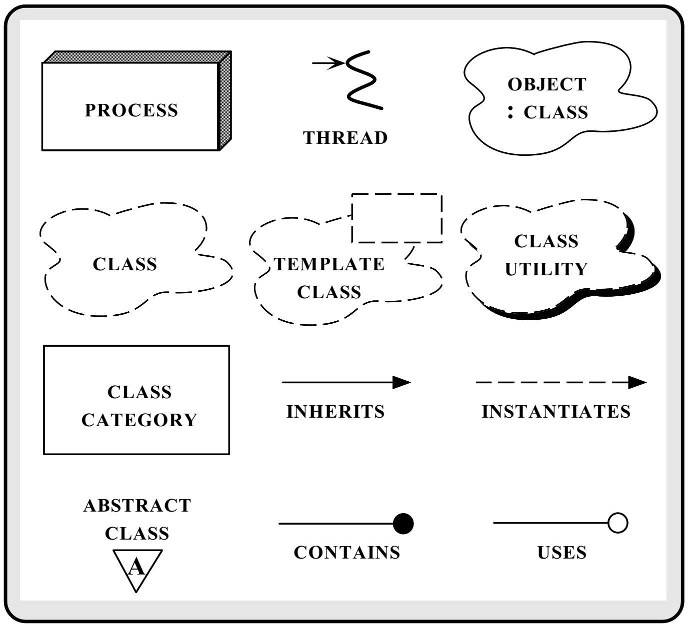
Class Categories in ACE



Class Categories in ACE (cont'd)

- Responsibilities of each class category
  - `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 demultiplexing* and *event handler dispatching*
  - `Service Configurator` automates *configuration* and *reconfiguration* 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

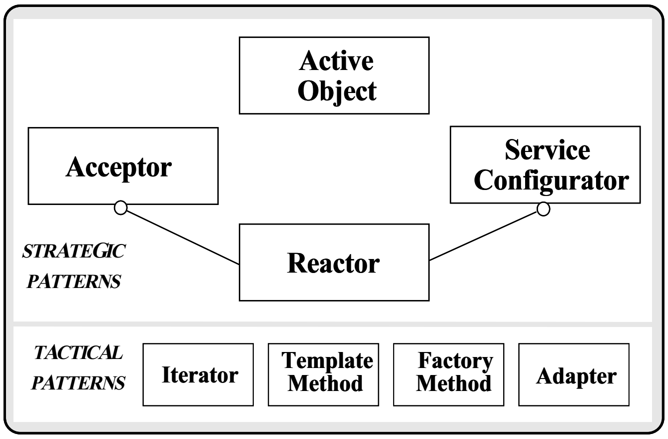
Graphical Notation



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

Design Patterns in the Distributed Logger



Design Patterns in the Distributed Logger (cont'd)

- *Reactor pattern*
  - Decouple event demultiplexing and event handler dispatching from application services performed in response to events
- *Acceptor pattern*
  - Decouple the passive initialization of a service from the tasks performed once the service is initialized

Design Patterns in the Distributed Logger (cont'd)

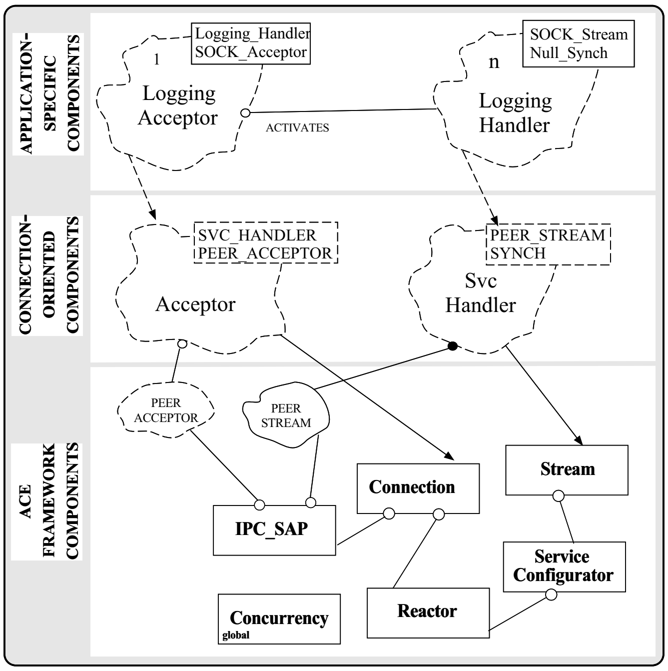
- *Service Configurator pattern*
  - Decouple the behavior of network services from point in time at which services are configured into an application
- *Active Object pattern*
  - Decouple method invocation from method execution and simplifies synchronized access to shared resources by concurrent threads

OO Logging Server

- The object-oriented server logging daemon is decomposed into several modular components that perform well-defined tasks:

1. *Application-specific components*
   - Process logging records received from clients
2. *Connection-oriented application components*
   - `Svc_Handler`
     - ▷ Performs I/O-related tasks with connected clients
   - `Acceptor` factory
     - ▷ Passively accepts connection requests from clients
     - ▷ Dynamically creates a `Svc_Handler` object for each client and "activates" it
3. *Application-independent ACE framework components*
   - Perform IPC, explicit dynamic linking, event demultiplexing, event handler dispatching, multi-threading, etc.

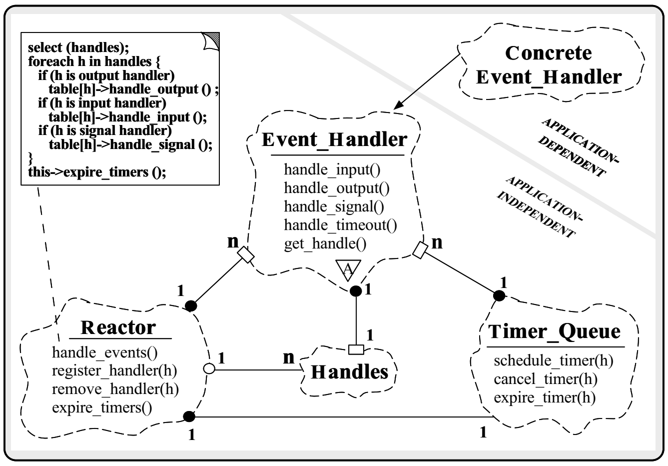
Class Diagram for OO Logging Server



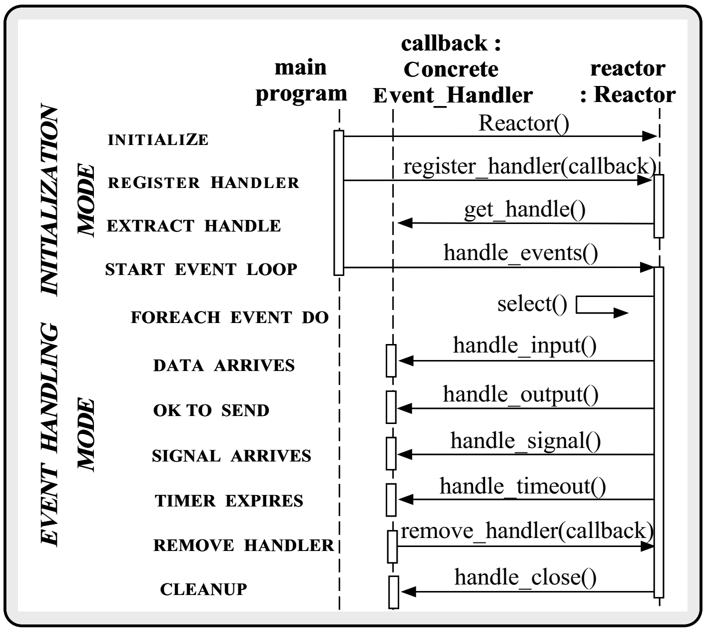
The Reactor Pattern

- *Intent*
  - "Decouple 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*

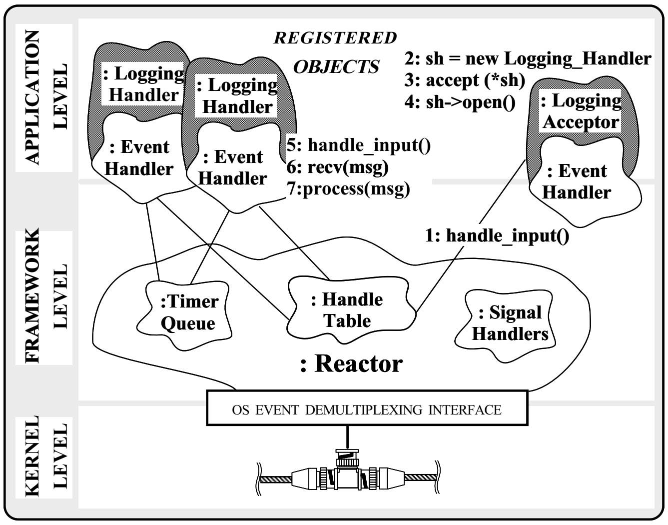
Structure of the Reactor Pattern



Collaboration in the Reactor Pattern



Using the Reactor Pattern in the Logging Server



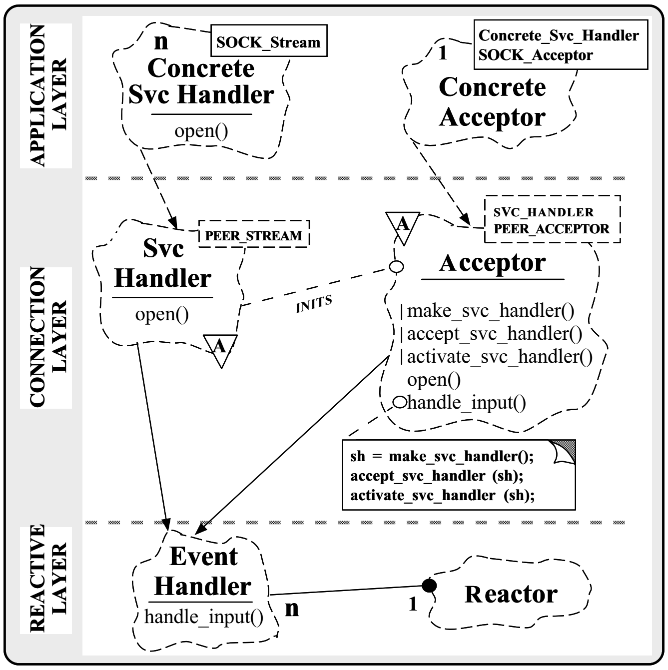
The Acceptor Pattern

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

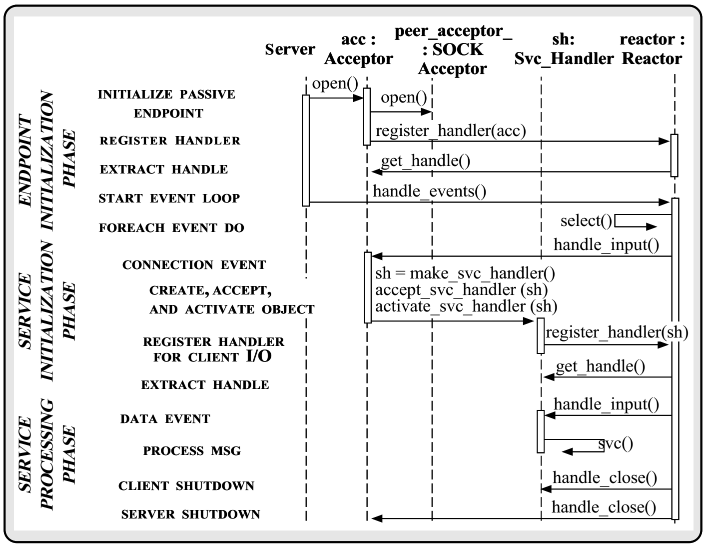
- This pattern resolves the following forces for network servers using interfaces like sockets or TLI:

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 enable flexible policies for creation, connection establishment, and concurrency*
4. *How to ensure that a passive-mode descriptor is not accidentally used to read or write data*

Structure of the Acceptor Pattern

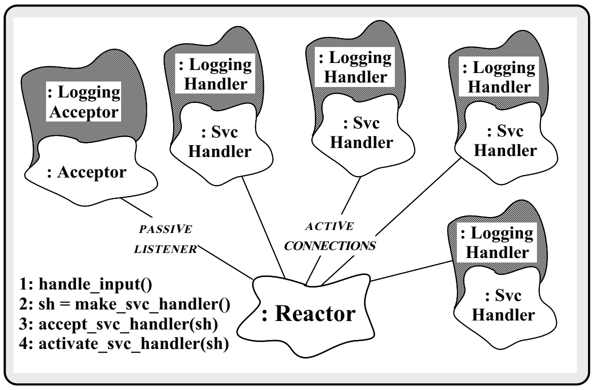


Collaboration in the Acceptor Pattern



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

Using the Acceptor Pattern in the Logging Server



Acceptor Class Public Interface

- A reusable template factory class that accepts connections from clients

```
template <class SVC_HANDLER, // Type of service
          class PEER_ACCEPTOR> // Accepts connections
class Acceptor : public Service_Object {
                        // Inherits from Event_Handler
public:
    // Initialization.
  virtual int open (const PEER_ACCEPTOR::PEER_ADDR &);

    // Template Method or Strategy for creating,
    // connecting,  and activating SVC_HANDLER's.
  virtual int handle_input (HANDLE);

    // Demultiplexing hooks.
  virtual HANDLE get_handle (void) const;
  virtual int handle_close (HANDLE, Reactor_Mask);
```


Acceptor Class Protected and Private Interfaces

- Only visible to the class and its subclasses

```
protected:
    // Factory method that creates a service handler.
  virtual SVC_HANDLER *make_svc_handler (void);

    // Factory method that accepts a new connection.
  virtual int accept_svc_handler (SVC_HANDLER *);

    // Factory method that activates a service handler.
  virtual int activate_svc_handler (SVC_HANDLER *);

private:
    // Passive connection mechanism.
  PEER_ACCEPTOR peer_acceptor_;
};
```


Acceptor Class Implementation

```
// Shorthand names.
#define SH SVC_HANDLER
#define PA PEER_ACCEPTOR

// Initialization.

template <class SH, class PA> int
Acceptor<SH, PA>::open (const PA::PEER_ADDR &addr)
{
  // Forward initialization to concrete peer acceptor
  peer_acceptor_.open (addr);

  // Register with Reactor.

  Service_Config::reactor()->register_handler
           (this, Event_Handler::READ_MASK);
}
```

```
// Template Method or Strategy Factory that creates,
//  connects, and activates new SVC_HANDLER objects.

template <class SH, class PA> int
Acceptor<SH, PA>::handle_input (HANDLE)
{
  // Factory Method that makes a service handler.

  SH *svc_handler = make_svc_handler ();

  // Accept the connection.

  accept_svc_handler (svc_handler);

  // Delegate control to the service handler.

  activate_svc_handler (svc_handler);
}
```



```
// Factory method for creating a service handler.
// Can be overridden by subclasses to define new
// allocation policies (such as Singletons, etc.).

template <class SH, class PA> SH *
Acceptor<SH, PA>::make_svc_handler (HANDLE)
{
  return new SH; // Default behavior.
}

// Accept connections from clients (can be overridden).

template <class SH, class PA> int
Acceptor<SH, PA>::accept_svc_handler (SH *svc_handler)
{
  peer_acceptor_.accept (*svc_handler);
}

// Activate the service handler (can be overridden).

template <class SH, class PA> int
Acceptor<SH, PA>::activate_svc_handler (SH *svc_handler)
{
  if (svc_handler->open () == -1)
    svc_handler->close ();
}
```



```
// Returns underlying I/O descriptor (called by Reactor).

template <class SH, class PA> HANDLE
Acceptor<SH, PA>::get_handle (void) const
{
  return peer_acceptor_.get_handle ();
}

// Perform termination activities when removed from Reactor.

template <class SH, class PA> int
Acceptor<SH, PA>::handle_close (HANDLE, Reactor_Mask)
{
  peer_acceptor_.close ();
}
```


Svc_Handler Class Public Interface

- Provides a generic interface for communication services that exchange data with a peer over a network connection

```
template <class PEER_STREAM, // Communication mechanism.
          class SYNCH> // Synchronization policy.
class Svc_Handler : public Task<SYNCH>
{
public:
  Svc_Handler (void); // Constructor.

    // Activate the client handler.
  virtual int open (void *);

    // Returns underlying PEER_STREAM.
  operator PEER_STREAM &();

    // Overloaded new operator that detects
    // when a Svc_Handler is allocated dynamically.
  void *operator new (size_t n);

    // Return underlying IPC mechanism.
  PEER_STREAM &peer (void);
```

Svc_Handler Class Protected Interface

- Contains the demultiplexing hooks and other implementation artifacts

```
protected:
    // Demultiplexing hooks inherited from Task.
  virtual int handle_close (HANDLE, Reactor_Mask);
  virtual HANDLE get_handle (void) const;
  virtual void set_handle (HANDLE);

private:
  PEER_STREAM peer_; // IPC mechanism.

    // Records if we're allocated dynamically.
  int is_dynamic_;

  virtual ~Svc_Handler (void);
};
```


Svc_Handler implementation

- By default, a `Svc_Handler` object is registered with the `Reactor`
  - This makes the service singled-threaded and no other synchronization mechanisms are necessary

```
#define PS PEER_STREAM // Convenient short-hand.

template <class PS, class SYNCH> int
Svc_Handler<PS, SYNCH>::open (void *)
{
  // Enable non-blocking I/O.
  peer ().enable (ACE_NONBLOCK);

  // Register handler with the Reactor.
  Service_Config::reactor()->register_handler
          (this, Event_Handler::READ_MASK);
}
```



```
// Extract the underlying I/O descriptor.

template <class PS, class SYNCH> HANDLE
Svc_Handler<PS, SYNCH>::get_handle (void) const
{
  // Forward request to underlying IPC mechanism.
  return peer ().get_handle ();
}

// Set the underlying I/O descriptor.

template <class PS, class SYNCH> void
Svc_Handler<PS, SYNCH>::set_handle (HANDLE h)
{
  peer ().set_handle (h);
}

// Return underlying IPC mechanism.

template <class PS, class SYNCH> PS &
Svc_Handler<PS, SYNCH>::peer (void)
{
  return peer_;
}
```



```
// Perform termination activities and
// deletes memory if we're allocated dynamically.

template <class PS, class SYNCH> int
Svc_Handler<PS, SYNCH>::handle_close
  (HANDLE, Reactor_Mask)
{
  peer ().close ();

  if (is_dynamic_)
    // This must be allocated dynamically!
    delete this;

  return 0;
}

// Extract the underlying PEER_STREAM (used by
// Acceptor::accept() and Connector::connect()).

template <class PS, class SYNCH>
Svc_Handler<PS, SYNCH>::operator PS &()
{
  return peer_;
}
```

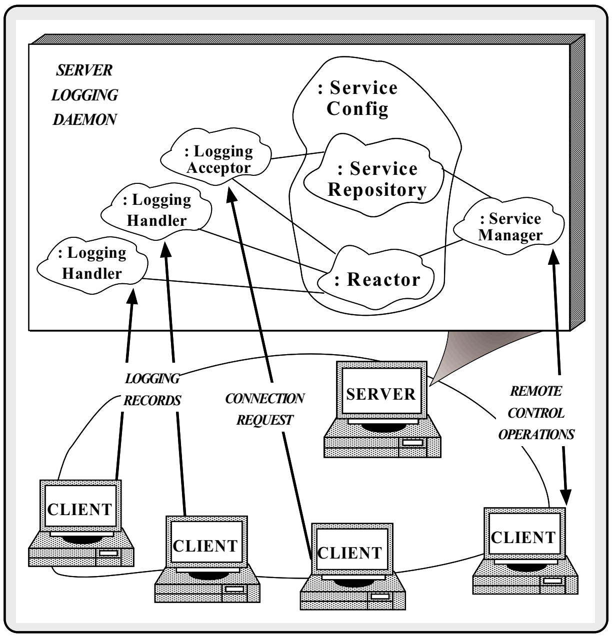
OO Design Interlude

- Q: "How can we determine if an object is allocated dynamically?"

- A: Overload class-specific `operator new` and use thread-specific storage to associate the pointers, *e.g.*,

```
template <class PS, class SYNCH> void *
Svc_Handler<PS, SYNCH>::operator new (size_t n)
{
  void *allocated = ::new char[n];
  set_thread_specific (key_, allocated);
  return allocated;
}

template <class PS, class SYNCH>
Svc_Handler<PS, SYNCH>::Svc_Handler (void)
{
  void *allocated = get_thread_specific (key_);
  is_dynamic_ = allocated == this;
}
```


Object Diagram for OO Logging Server

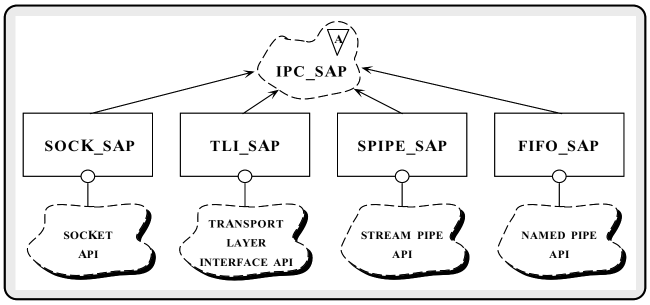


The Logging_Handler and Logging_Acceptor Classes

- These instantiated template classes implement application-specific server logging daemon functionality

```
// Performs I/O with client logging daemons.

class Logging_Handler :
  public Svc_Handler<SOCK_Stream, NULL_SYNCH> {
public:
    // Recv and process remote logging records.
  virtual int handle_input (HANDLE);
protected:
  char host_name_[MAXHOSTNAMELEN + 1]; // Client.
};

// Logging_Handler factory.

class Logging_Acceptor :
  public Acceptor<Logging_Handler, SOCK_Acceptor> {
public:
    // Dynamic linking hooks.
  virtual int init (int argc, char *argv[]);
  virtual int fini (void);
};
```


OO Design Interlude



- Q: *What are the SOCK_* classes and why are they used rather than using sockets directly?*

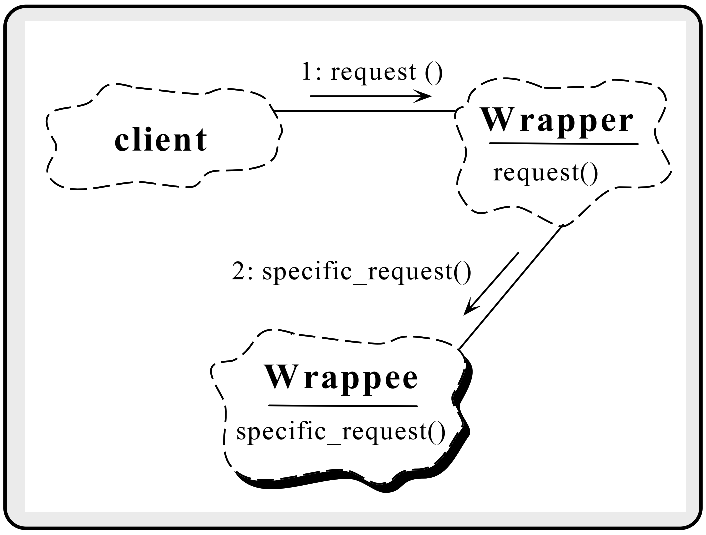
- A: `SOCK_*` are "wrappers" that encapsulate network programming interfaces like sockets and TLI
  - This is an example of the "Wrapper pattern"

The Wrapper Pattern

- *Intent*
  - "Encapsulate lower-level functions within type-safe, modular, and portable class interfaces"

- This pattern resolves the following forces that arise when using native C-level OS APIs
  1. *How to avoid tedious, error-prone, and non-portable programming of low-level IPC mechanisms*
  2. *How to combine multiple related, but independent, functions into a single cohesive abstraction*

Structure of the Wrapper Pattern



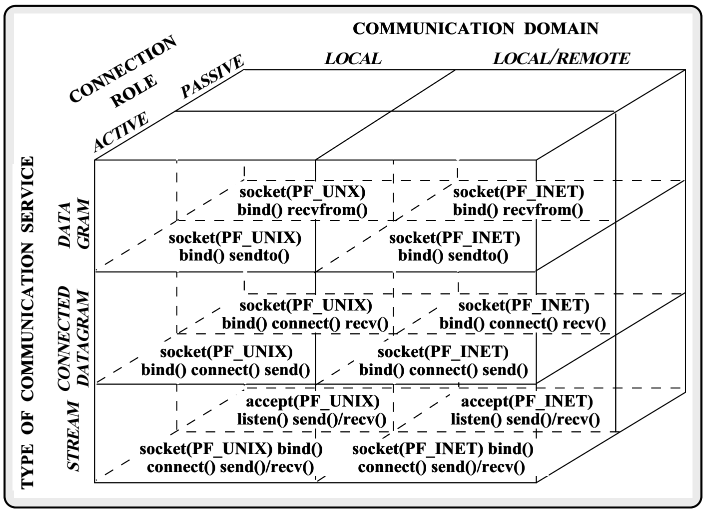
Socket Structure

socket()
bind()
connect()
listen()
accept()
read()
write()
readv()
writev()
recv()
send()
recvfrom()
sendto()
recvmsg()
sendmsg()
setsockopt()
getsockopt()
getpeername()
getsockname()
gethostbyname()
getservbyname()

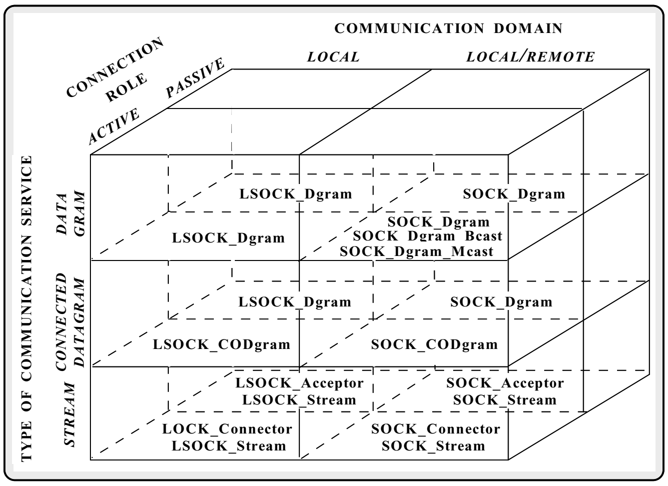
- Note that this API is *linear* rather than *hierarchical*
  - Thus, it gives no hints on how to use it correctly

- In addition, there is no consistency among names...

Socket Taxonomy



SOCK_SAP Class Structure



SOCK_SAP Factory Class Interfaces

```
class SOCK_Connector : public SOCK
{
public:
  // Traits
  typedef INET_Addr PEER_ADDR;
  typedef SOCK_Stream PEER_STREAM;

  int connect (SOCK_Stream &new_sap, const Addr &remote_addr,
               Time_Value *timeout, const Addr &local_addr);
  // ...
};

class SOCK_Acceptor : public SOCK
{
public:
  // Traits
  typedef INET_Addr PEER_ADDR;
  typedef SOCK_Stream PEER_STREAM;

  SOCK_Acceptor (const Addr &local_addr);

  int accept (SOCK_Stream &, Addr *, Time_Value *) const;
  //...
};
```


SOCK_SAP Stream and Addressing Class Interfaces

```
class SOCK_Stream : public SOCK
{
public:
  typedef INET_Addr PEER_ADDR; // Trait.

  ssize_t send (const void *buf, int n);
  ssize_t recv (void *buf, int n);
  ssize_t send_n (const void *buf, int n);
  ssize_t recv_n (void *buf, int n);
  ssize_t send_n (const void *buf, int n, Time_Value *);
  ssize_t recv_n (void *buf, int n, Time_Value *);
  int close (void);
  // ...
};

class INET_Addr : public Addr
{
public:
  INET_Addr (u_short port_number, const char host[]);
  u_short get_port_number (void);
  int32 get_ip_addr (void);
  // ...
};
```


OO Design Interlude

- Q: *Why decouple the SOCK_Acceptor and the SOCK_Connector from SOCK_Stream?*

- A: For the same reasons that Acceptor and Connector are decoupled from Svc_Handler, *e.g.*,

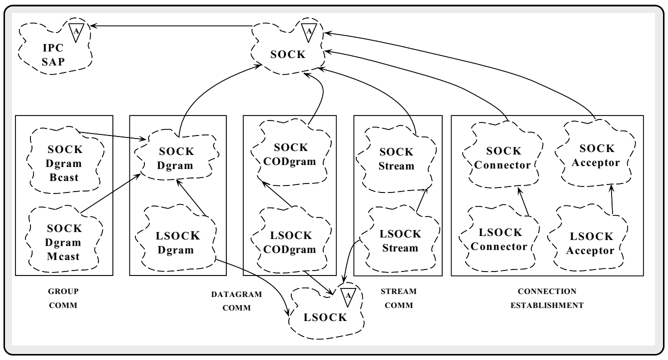
  – A `SOCK_Stream` is only responsible for data transfer

    ▷ Regardless of whether the connection is established passively or actively

  – This ensures that the `SOCK*` components are never used incorrectly...

    ▷ *e.g.*, you can't accidentally **`read`** or **`write`** on **`SOCK_Connectors`** or **`SOCK_Acceptors`**, etc.

SOCK_SAP Hierarchy



- Shared behavior is isolated in base classes
- Derived classes implement different communication services, communication domains, and connection roles

OO Design Interlude

- Q: "How can you switch between different IPC mechanisms?"
- A: By parameterizing IPC Mechanisms with C++ Templates!

```
#if defined (ACE_USE_SOCKETS)
typedef SOCK_Acceptor PEER_ACCEPTOR;
#elif defined (ACE_USE_TLI)
typedef TLI_Acceptor PEER_ACCEPTOR;
#endif /* ACE_USE_SOCKETS */

class Logging_Handler : public
  Svc_Handler<PEER_ACCEPTOR::PEER_STREAM,
              NULL_SYNCH>
    { /* ... /* };

class Logging_Acceptor : public
  Acceptor <Logging_Handler, PEER_ACCEPTOR>
    { /* ... */ };
```


Logging_Handler Implementation

- Implementation of the application-specific logging method

```
// Callback routine that receives logging records.
// This is the main code supplied by a developer!

template <class PS, class SYNCH> int
Logging_Handler<PS, SYNCH>::handle_input (HANDLE)
{
  // Call existing function to recv
  // logging record and print to stdout.
  handle_log_record (peer ().get_handle (), 1);
}
```



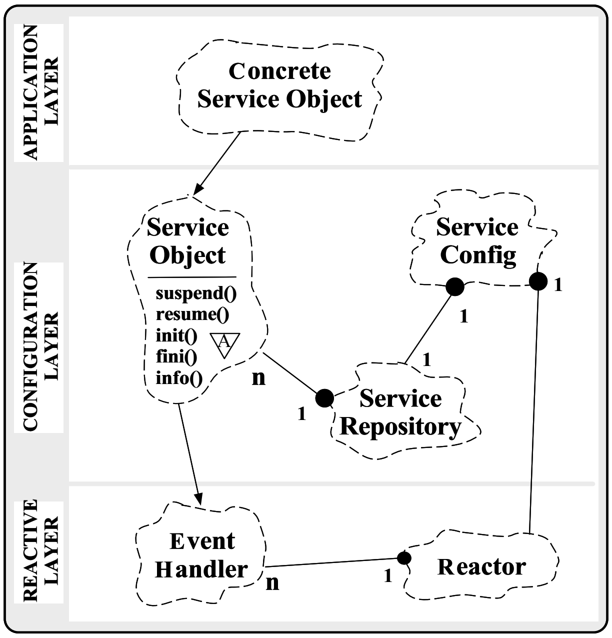
```
// Automatically called when a Logging_Acceptor object
// is dynamically linked.

Logging_Acceptor::init (int argc, char *argv[])
{
  Get_Opt get_opt (argc, argv, "p:", 0);
  INET_Addr addr;

  for (int c; (c = get_opt ()) != -1; )
     switch (c)
       {
       case 'p':
         addr.set (atoi (getopt.optarg));
         break;
       default:
         break;
       }

  // Initialize endpoint and register with the Reactor
  open (addr);
}

// Automatically called when object is dynamically unlinked.

Logging_Acceptor::fini (void)
{
  handle_close ();
}
```

The Service Configurator Pattern

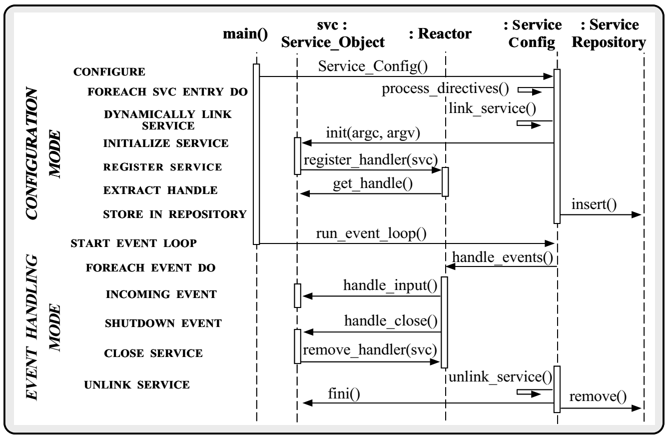
- *Intent*
  - "Decouple the behavior of network services from the point in time at which these services are configured into an application"

- 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 reconfigure and control the behavior of the service at run-time*

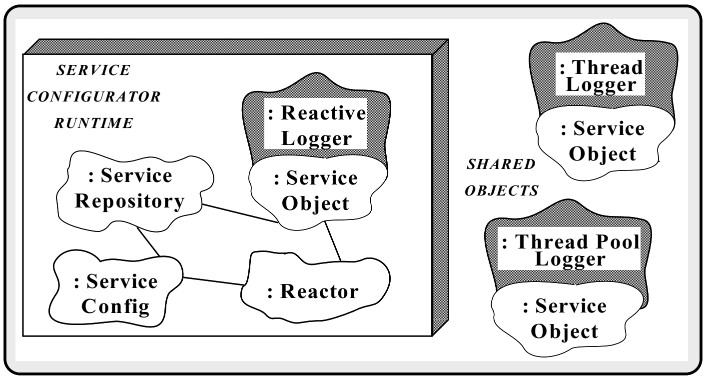
Structure of the Service Configurator Pattern



Collaboration in the Service Configurator Pattern



Using the Service Configurator Pattern for the Logging Server



- Existing service is single-threaded, other versions could be multi-threaded...

Dynamic Linking a Service

- Application-specific factory function used to dynamically create a service

```
// Dynamically linked factory function that allocates
// a new Logging_Acceptor object dynamically

extern "C" Service_Object *make_Logger (void);

Service_Object *
make_Logger (void)
{
  return new Logging_Acceptor;
  // Framework automatically deletes memory.
}
```

- The `make_Logger` function provides a *hook* between an *application-specific* service and the *application-independent* ACE mechanisms
  - ACE handles all memory allocation and deallocation

Service Configuration

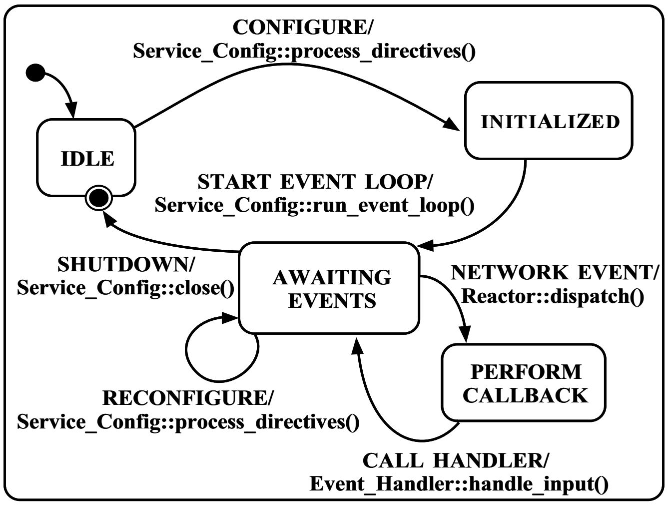
- The logging service is configured and initialized based upon the contents of a configuration script called `svc.conf`:

```
% cat ./svc.conf
# Dynamically configure the logging service
dynamic Logger Service_Object *
        logger.dll:make_Logger() "-p 2010"
```

- Generic event-loop to dynamically configure service daemons

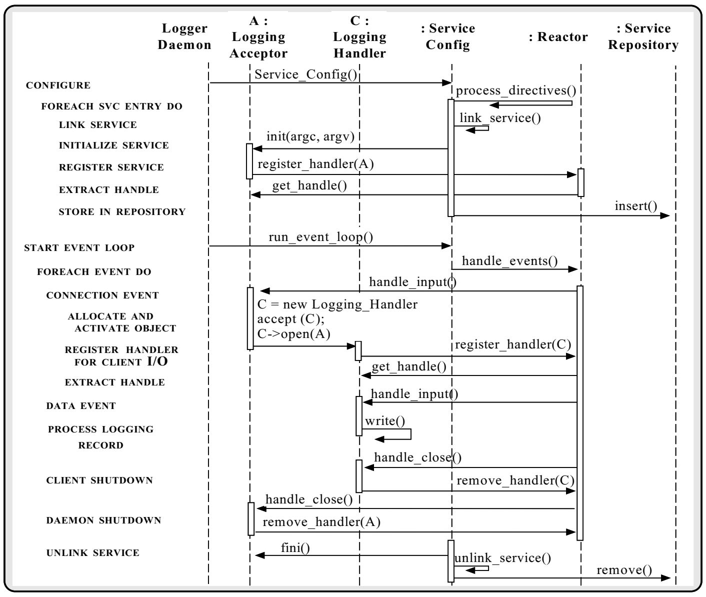
```
int
main (int argc, char *argv[])
{
  Service_Config daemon;
  // Initialize the daemon and configure services
  daemon.open (argc, argv);

  // Run forever, performing configured services
  daemon.run_reactor_event_loop ();
  /* NOTREACHED */
}
```


State-chart Diagram for the Service Configurator Pattern



- Note the separation of concerns between objects...

Collaboration of Patterns in the Server Logging Daemon



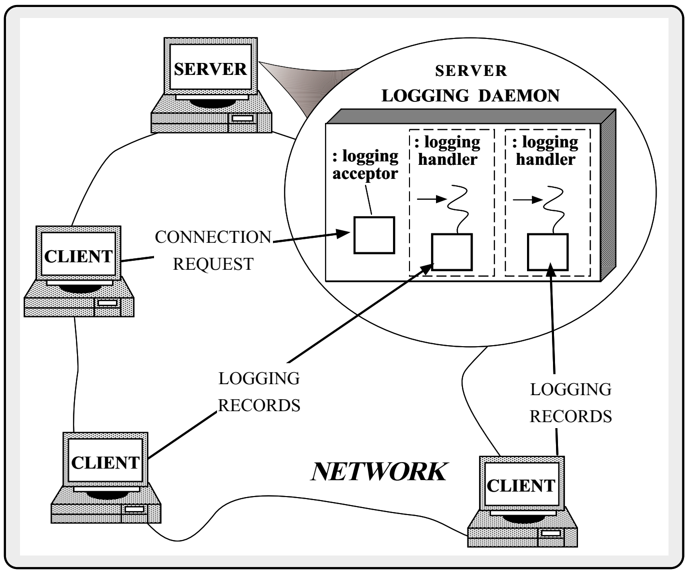
Advantages of OO Logging Server

- The OO architecture illustrated thus far decouples application-specific service functionality from:
  * Time when a service is configured into a process
  * The number of services per-process
  * The type of IPC mechanism used
  * The type of event demultiplexing mechanism used
- We can use the techniques discussed thus far to extend applications *without*:
  1. *Modifying*, *recompiling*, and *relinking* existing code
  2. *Terminating* and *restarting* executing daemons
- The remainder of the slides examine a set of techniques for decoupling functionality from *concurrency* mechanisms, as well

Concurrent OO Logging Server

- The structure of the server logging daemon can benefit from concurrent execution on a multi-processor platform
- This section examines ACE C++ classes and design patterns that extend the logging server to incorporate concurrency
  – Note how most extensions require minimal changes to the existing OO architecture...
- This example also illustrates additional ACE components involving synchronization and multi-threading

Concurrent OO Logging Server Architecture



- Thread-based implementation

Pseudo-code for Concurrent Server

- Pseudo-code for multi-threaded Logging_Handler factory server logging daemon

```
void handler_factory (void)
{
    initialize listener endpoint
    foreach (pending connection request) {
        accept request
        spawn a thread to handle request
        call logger_handler() active object
    }
}
```

- Pseudo-code for server logging daemon active object

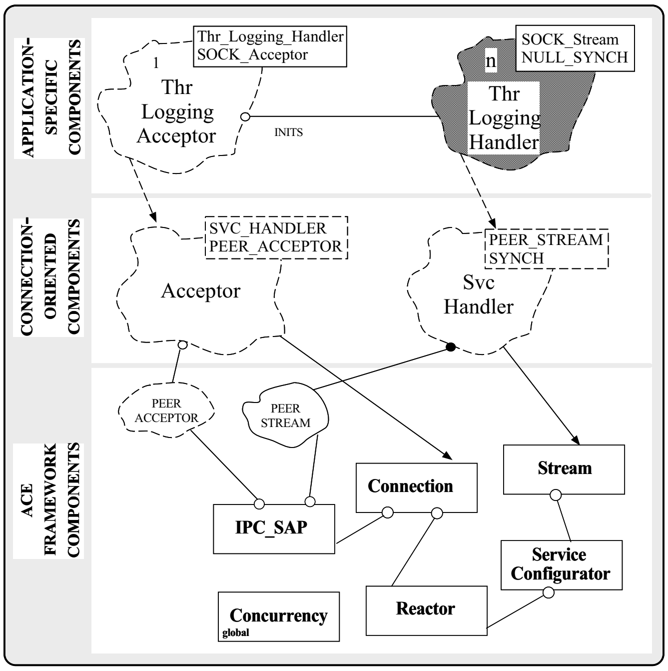
```
void logging_handler (void)
{
    foreach (incoming logging records from client)
        call handle_log_record()
    exit thread
}
```

Application-specific Logging Code

- The OO implementation localizes the application-specific part of the logging service in a single point, while leveraging off reusable ACE components

```
// Handle all logging records from a particular client
// (run in each slave thread).

int
Thr_Logging_Handler::svc (void)
{
  // Perform a "blocking" receive and process logging
  // records until the client closes down the connection.

  // Danger! race conditions...

  for (; ; ++request_count)
    handle_log_record (get_handle (), 1);

  /* NOTREACHED */
  return 0;
}
```

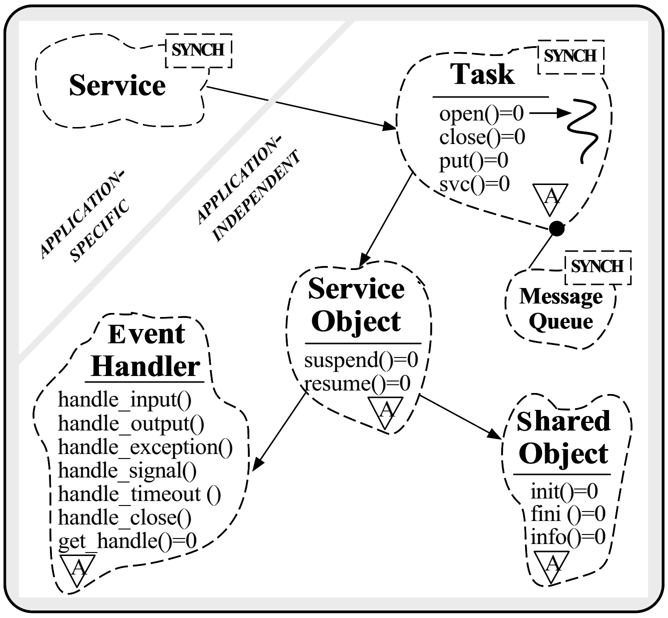

Class Diagram for Concurrent OO Logging Server



ACE Tasks

- An ACE `Task` binds a separate thread of control together with an object's data and methods
  - Multiple active objects may execute in parallel in separate lightweight or heavyweight processes
- `Task` objects communicate by passing typed messages to other `Tasks`
  - Each `Task` maintains a queue of pending messages that it processes in *priority order*
- ACE `Task` are a low-level mechanism to support "active objects"

Task Inheritance Hierarchy



- Supports dynamically configured services

Task Class Public Interface

- C++ interface for message processing

  * Tasks can register with a Reactor
  * They can be dynamically linked
  * They can queue data
  * They can run as "active objects"

- *e.g.*,

```
template <class SYNCH>
class Task : public Service_Object
{
public:
    // Initialization/termination routines.
  virtual int open (void *args = 0) = 0;
  virtual int close (u_long flags = 0) = 0;

    // Transfer msg to queue for immediate processing.
  virtual int put (Message_Block *, Time_Value * = 0) = 0;

    // Run by a daemon thread for deferred processing.
  virtual int svc (void) = 0;

    // Turn the task into an active object.
  int activate (long flags);
```


Task Class Protected Interface

- The following methods are mostly used within put and svc

```
  // Accessors to internal queue.
Message_Queue<SYNCH> *msg_queue (void);
void msg_queue (Message_Queue<SYNCH> *);

  // Accessors to thread manager.
Thread_Manager *thr_mgr (void);
void thr_mgr (Thread_Manager *);

  // Insert message into the message list.
int putq (Message_Block *, Time_Value *tv = 0);

  // Extract the first message from the list (blocking).
int getq (Message_Block *&mb, Time_Value *tv = 0);

  // Hook into the underlying thread library.
static void *svc_run (Task<SYNCH> *);
```


OO Design Interlude

- Q: *What is the svc_run() function and why is it a static method?*

- A: OS thread spawn APIs require a C-style function as the entry point into a thread

- The Stream class category encapsulates the svc_run function within the Task::activate method:

```
template <class SYNCH> int
Task<SYNCH>::activate (long flags, int n_threads)
{
  if (thr_mgr () == NULL)
    thr_mgr (Service_Config::thr_mgr ());

  thr_mgr ()->spawn_n
    (n_threads, &Task<SYNCH>::svc_run,
     (void *) this, flags);
}
```


OO Design Interlude (cont'd)

- Task::svc_run is static method used as the entry point to execute an instance of a service concurrently in its own thread

```
template <class SYNCH> void *
Task<SYNCH>::svc_run (Task<SYNCH> *t)
{
  Thread_Control tc (t->thr_mgr ()); // Record thread ID.

  // Run service handler and record return value.
  void *status = (void *) t->svc ();

  tc.status (status);
  t->close (u_long (status));

  // Status becomes 'return' value of thread...
  return status;
  // Thread removed from thr_mgr()
  // automatically on return...
}
```

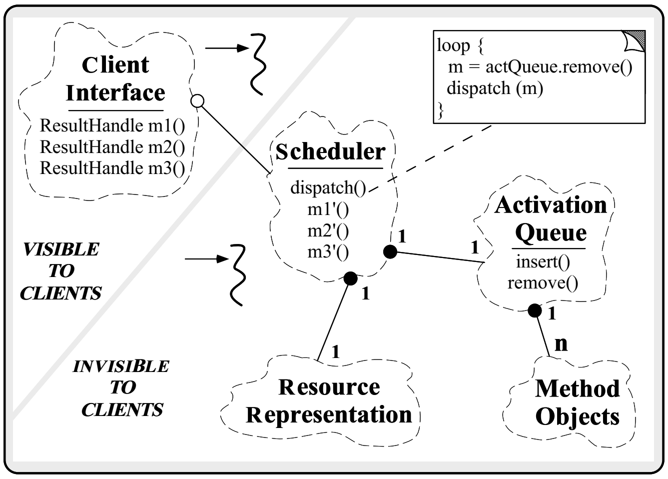
OO Design Interlude

- Q: "How can groups of collaborating threads be managed atomically?"
- A: Develop a "thread manager" class
  - Thread_Manager is a collection class
    - ▹ It provides mechanisms for *suspending* and *resuming* groups of threads atomically
    - ▹ It implements *barrier synchronization* on thread exits
  - Thread_Manager also shields applications from incompabitilities between different OS thread libraries
    - ▹ It is integrated into ACE via the Task::activate method

The Active Object Pattern

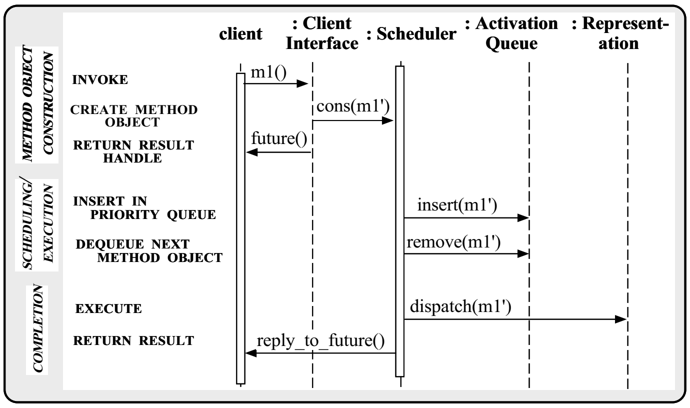
- *Intent*
  - "Decouple method execution from method invocation and simplifies synchronized access to shared resources by concurrent threads"
- This pattern resolves the following forces for concurrent communication software:
  - *How to allow blocking read and write operations on one endpoint that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

Structure of the Active Object Pattern

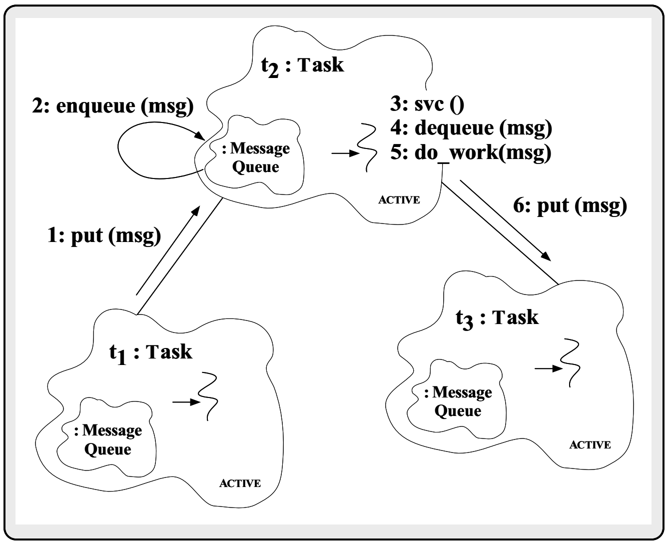


- The Scheduler is a "meta-object" that determines the sequence that Method Objects are executed

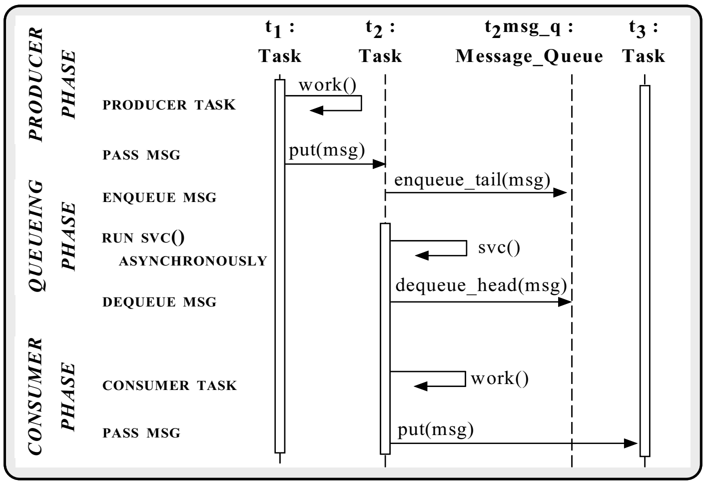
Collaboration in the Active Object Pattern



ACE Task Support for Active Objects



Collaboration in ACE Active Objects



Thr_Logging_Acceptor and Thr_Logging_Handler

- Template classes that create, connect, and activate a new thread to handle each client

```
class Thr_Logging_Handler
  : public Svc_Handler<SOCK_Stream, NULL_SYNCH>
{
public:
    // Override definition in the Svc_Handler class
    // (spawns a new thread!).
  virtual int open (void *);

    // Process remote logging records.
  virtual int svc (void);
};

class Thr_Logging_Acceptor :
  public Acceptor<Thr_Logging_Handler,
                  SOCK_Acceptor>
{
  // Same as Logging_Acceptor...
};
```



```
// Override definition in the Svc_Handler class
// (spawns a new thread in this case!).

int
Thr_Logging_Handler::open (void *)
{
  INET_Addr client_addr;

  peer ().get_local_addr (client_addr);
  strncpy (host_name_, client_addr.get_host_name (),
           MAXHOSTNAMELEN + 1);

  // Spawn a new thread to handle
  // logging records with the client.
  activate (THR_BOUND | THR_DETACHED);
}
```

```
// Process remote logging records.

int
Thr_Logging_Handler::svc (void)
{
  // Loop until the client terminates the connection.

  for (; ; ++request_count)
    // Call existing function to recv logging
    // record and print to stdout.

    handle_log_record (peer ().get_handle (), 1);

  /* NOTREACHED */
  return 0;
}

// Dynamically linked factory function that
// allocates a new threaded logging Acceptor object.

extern "C" Service_Object *make_Logger (void);

Service_Object *
make_Logger (void)
{
  return new Thr_Logging_Acceptor;
}
```


Dynamically Reconfiguring the Logging Server

- The concurrent logging service is configured and initialized by making a small change to the `svc.conf` file:

```
% cat ./svc.conf
# Dynamically configure the logging service
# dynamic Logger Service_Object *
#         /svcs/logger.dll:make_Logger() "-p 2010"
remove Logger
dynamic Logger Service_Object *
        thr_logger.dll:make_Logger() "-p 2010"
```

- Sending a SIGHUP signal reconfigures the server logger daemon process
  - The original sequential version of the `Logger` service is dynamically unlinked and replaced with the new concurrent version

Caveats

- The concurrent server logging daemon has several problems
  1. Output in the `handle_log_record` function is not serialized
  2. The auto-increment of global variable `request_count` is also not serialized
- Lack of serialization leads to errors on many shared memory multi-processor platforms...
  - Note that this problem is indicative of a large class of errors in concurrent programs...
- The following slides compare and contrast a series of techniques that address this problem

Explicit Synchronization Mechanisms

- One approach for serialization uses OS mutual exclusion mechanisms explicitly, *e.g.*,

```
// at file scope
mutex_t lock; // SunOS 5.x synchronization mechanism

// ...
handle_log_record (HANDLE in_h, HANDLE out_h)
{
  // in method scope ...
  mutex_lock (&lock);
  write (out_h, log_record.buf, log_record.size);
  mutex_unlock (&lock);
  // ...
}
```

- However, adding these `mutex` calls explicitly is *inelegant*, *tedious*, *error-prone*, and *non-portable*

C++ Wrappers for Synchronization

- To address portability problems, define a C++ wrapper:

```
class Mutex
{
public:
  Mutex (void) {
    mutex_init (&lock_, USYNCH_THREAD, 0);
  }
  ~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 Mutex &);
  void Mutex (const Mutex &);
};
```

- Note, this mutual exclusion class interface is portable to other OS platforms

Porting Mutex to Windows NT

- WIN32 version of `Mutex`

```
class Mutex
{
public:
  Mutex (void) {
    lock_ = CreateMutex (0, FALSE, 0);
  }
  ~Mutex (void) {
    CloseHandle (lock_);
  }
  int acquire (void) {
    return WaitForSingleObject (lock_, INFINITE);
  }
  int tryacquire (void) {
    return WaitForSingleObject (lock_, 0);
  }
  int release (void) {
    return ReleaseMutex (lock_);
  }
private:
  HANDLE lock_; // Win32 locking mechanism.
  // ...
```


Using the C++ Mutex Wrapper

- Using C++ wrappers improves *portability* and *elegance*

```
// at file scope

Mutex lock; // Implicitly "unlocked".

// ...
handle_log_record (HANDLE in_h, HANDLE out_h)
{
  // in method scope ...

  lock.acquire ();
  write (out_h, log_record.buf, log_record.size);
  lock.release ();

  // ...
}
```

- However, this doesn't really solve the *tedium* 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*

OO Design Interlude

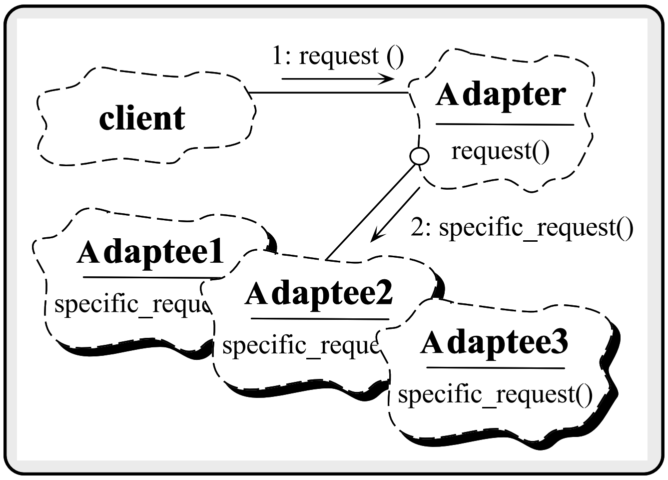
- Q: *Why is Guard parameterized by the type of LOCK?*
- A: since there are many different flavors of locking that benefit from the `Guard` functionality, *e.g.*,
  - * *Non-recursive vs recursive mutexes*
  - * *Intra-process vs inter-process mutexes*
  - * *Readers/writer mutexes*
  - * *Solaris and System V semaphores*
  - * *File locks*
  - * *Null mutex*
- In ACE, all synchronization wrappers use to Adapter pattern to provide identical interfaces whenever possible to facilitate parameterization

The Adapter Pattern

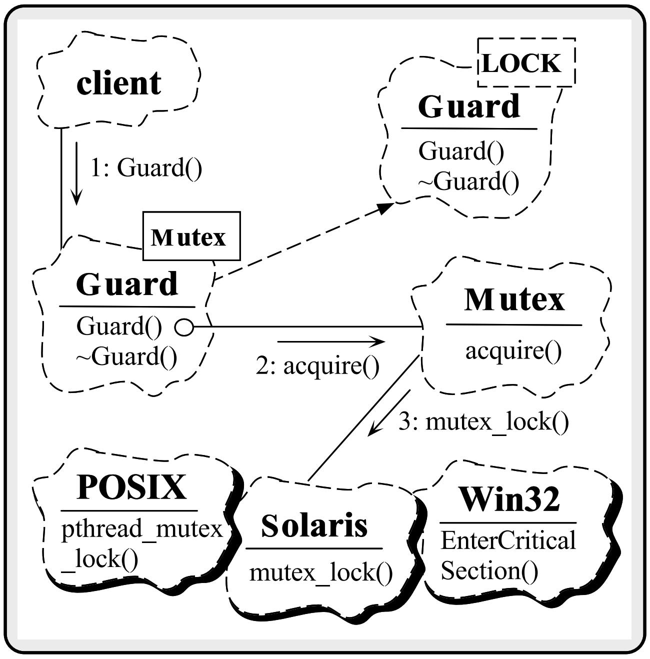
- *Intent*
  - "Convert the interface of a class into another interface client expects"
    - ▹ Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
- This pattern resolves the following force that arises when using conventional OS interfaces

1. *How to provide an interface that expresses the similarities of seemingly different OS mechanisms (such as locking or IPC)*

Structure of the Adapter Pattern



Using the Adapter Pattern for Locking



A thread-safe handle_log_record() Function

```
template <class LOCK = Mutex> ssize_t
handle_log_record (HANDLE in_h, HANDLE out_h)
{
  // new code (beware of static initialization...)
  static LOCK lock;
  ssize_t n;
  size_t len;
  Log_Record log_record;

  n = recv (h, (char *) &len, sizeof len, 0);

  if (n != sizeof len) return -1;
  len = ntohl (len); // Convert byte-ordering

  for (size_t nread = 0; nread < len; nread += n
    n = recv (in_h, ((char *) &log_record) + nread,
              len - nread, 0));
  // Perform presentation layer conversions.
  decode (&log_record);
  // Automatically acquire mutex lock.
  Guard<LOCK> monitor (lock);
  write (out_h, log_record.buf, log_record.size);
  // Automatically release mutex lock.
}
```


Remaining Caveats

- There is a race condition when incrementing the `request_count` variable

```
// Danger! race conditions...
for (; ; ++request_count)
  handle_log_record (get_handle (), 1);
```

- Solving this problem using the `Mutex` or `Guard` classes is still *tedious*, *low-level*, and *error-prone*

- A more elegant solution incorporates parameterized types, overloading, and the Decorator pattern

Transparently Parameterizing Synchronization Using C++

- The following C++ template class uses the "Decorator" pattern to define a set of atomic operations on a type parameter:

```
template <class LOCK = Mutex, class TYPE = u_long>
class Atomic_Op {
public:
  Atomic_Op (TYPE c = 0) { count_ = c; }
  TYPE operator++ (void) {
    Guard<LOCK> m (lock_); return ++count_;
  }
  void operator= (const Atomic_Op &ao) {
    if (this != &ao) {
      Guard<LOCK> m (lock_); count_ = ao.count_;
    }
  }
  operator TYPE () {
    Guard<LOCK> m (lock_);
    return count_;
  }
  // Other arithmetic operations omitted...
private:
  LOCK lock_;
  TYPE  count_;
};
```


Final Version of Concurrent Logging Server

- Using the `Atomic_Op` class, only one change is made

```
// At file scope.
typedef Atomic_Op<> COUNTER; // Note default parameters...
COUNTER request_count;
```

- `request_count` is now serialized automatically

```
for (; ; ++request_count) // Atomic_Op::operator++
  handle_log_record (get_handle (), 1);
```

- The original non-threaded version may be supported efficiently as follows:

```
typedef Atomic_Op<Null_Mutex> COUNTER;
//...
  for (; ; ++request_count)
    handle_log_record<Null_Mutex>
      (get_handle (), 1);
```

Synchronization-aware Logging Classes

- A more sophisticated approach would add several new parameters to the `Logging_Handler` class

```
template <class PEER_STREAM,
          class SYNCH, class COUNTER>
class Logging_Handler
  : public Svc_Handler<PEER_STREAM, SYNCH>
{
public:
  Logging_Handler (void);
    // Process remote logging records.
  virtual int svc (void);

protected:
    // Receive the logging record from a client.
  ssize_t handle_log_record (HANDLE out_h);
    // Lock used to serialize access to std output.
  static SYNCH::MUTEX lock_;
    // Count the number of logging records that arrive.
  static COUNTER request_count_;
    // Name of the host we are connected to.
  char host_name_[MAXHOSTNAMELEN + 1];
};
```


Thread-safe handle_log_record Method

```
template <class PS, class LOCK, class COUNTER> ssize_t
Logging_Handler<PS, LOCK, COUNTER>::handle_log_record
     (HANDLE out_h)
{
  ssize_t n;
  size_t len;
  Log_Record log_record;

  ++request_count_; // Calls COUNTER::operator++().

  n = peer ().recv (&len, sizeof len);

  if (n != sizeof len) return -1;
  len = ntohl (len); // Convert byte-ordering

  peer ().recv_n (&log_record, len);

  // Perform presentation layer conversions
  log_record.decode ();
  // Automatically acquire mutex lock.
  Guard<LOCK> monitor (lock_);
  write (out_h, log_record.buf, log_record.size);
  // Automatically release mutex lock.
}
```


Using the Thread-safe handle_log_record() Method

- In order to use the thread-safe version, all we need to do is instantiate with `Atomic_Op`

```
typedef Logging_Handler<TLI_Stream,
                        NULL_SYNCH,
                        Atomic_Op<> >
        LOGGING_HANDLER;
```

- To obtain single-threaded behavior requires a simple change:

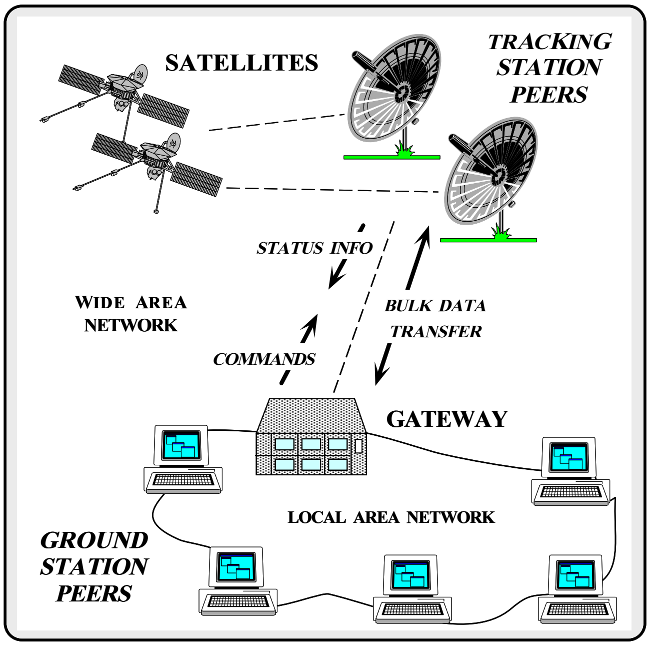
```
typedef Logging_Handler<TLI_Stream,
                        NULL_SYNCH,
                        Atomic_Op <Null_Mutex, u_long> >
        LOGGING_HANDLER;
```


Application-level Gateway Example

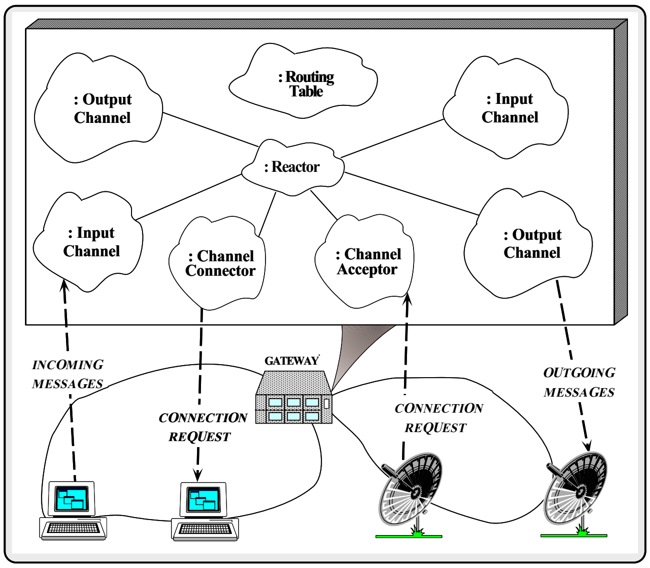
- The distributed logger is a relatively simple application
  - *e.g.*, it doesn't to deal with network *output* or queueing

- The next example explores the *design patterns* and *reusable framework* components used in an OO architecture for *application-level Gateways*
  - These Gateways route messages between Peers in a large-scale telecommunication system
  - Peers and Gateways are connected via TCP/IP

Physical Architecture of the Gateway



OO Software Architecture of the Gateway

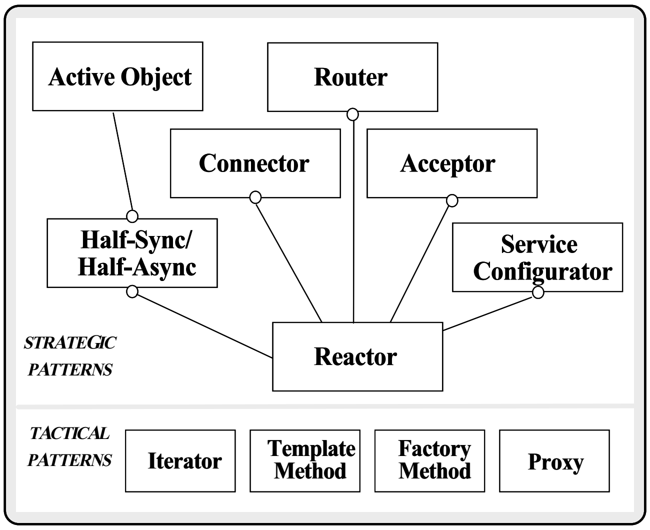


Gateway Behavior

- Components in the Gateway behave as follows:

1. `Gateway` parses configuration files that specify which Peers to connect with and which routes to use
2. `Channel_Connector` connects to Peers, then creates and activates the appropriate `Channel` subclasses (`Input_Channel` or `Output_Channel`)
3. Once connected, Peers send messages to the Gateway
   – Messages are handled by the appropriate `Input_Channel`
   – `Input_Channels` work as follows:
   (a) Receive and validate messages
   (b) Consult a `Routing_Table`
   (c) Forward messages to the appropriate Peer(s) via `Output_Channels`

Design Patterns in the Gateway

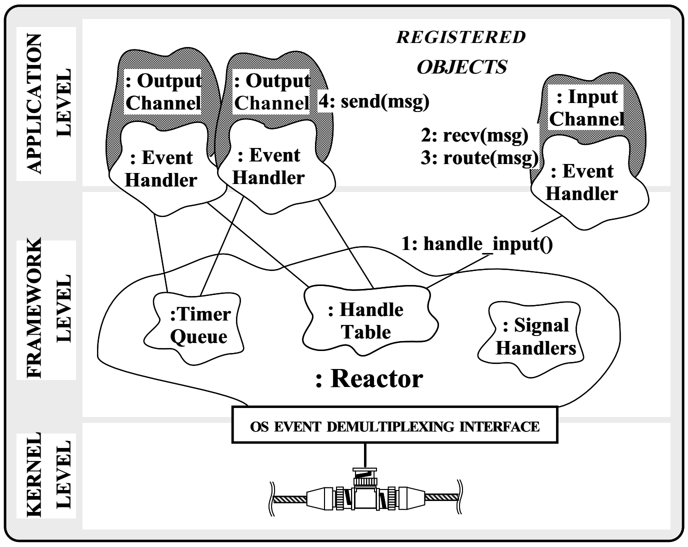


- The Gateway components are based upon a system of design patterns

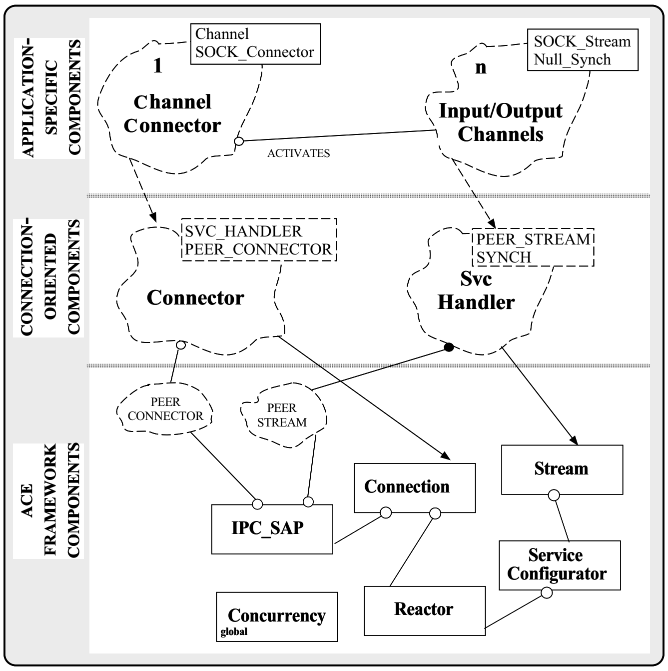
Design Patterns in the Gateway (cont'd)

- The Gateway uses the same system of patterns as the distributed logger
  - *i.e.*, Reactor, Service Configurator, Active Object, and Acceptor
- It also contains following patterns:
  - *Connector pattern*
    - ▷ Decouple the active initialization of a service from the tasks performed once the service is initialized
  - *Router pattern*
    - ▷ Decouple multiple sources of input from multiple sources of output to prevent blocking in a single-thread of control
  - *Half-Sync/Half-Async*
    - ▷ Decouple synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency

Using the Reactor Pattern for the Single-Threaded Gateway



Class Diagram for Single-Threaded Gateway



OO Gateway Architecture

- The Gateway is decomposed into components that are layered as follows:

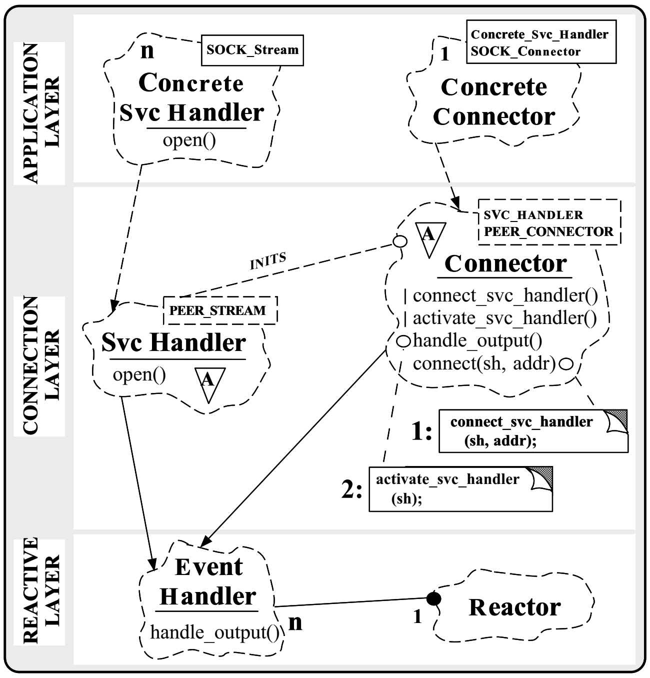
1. *Application-specific components*
   - `Channels` route messages among Peers
2. *Connection-oriented application components*
   - `Svc_Handler`
     - ▷ Performs I/O-related tasks with connected clients
   - `Connector` factory
     - ▷ Establishes new connections with clients
     - ▷ Dynamically creates a `Svc_Handler` object for each client and "activates" it
3. *Application-independent ACE framework components*
   - Perform IPC, explicit dynamic linking, event demultiplexing, event handler dispatching, multithreading, etc.

The Connector Pattern

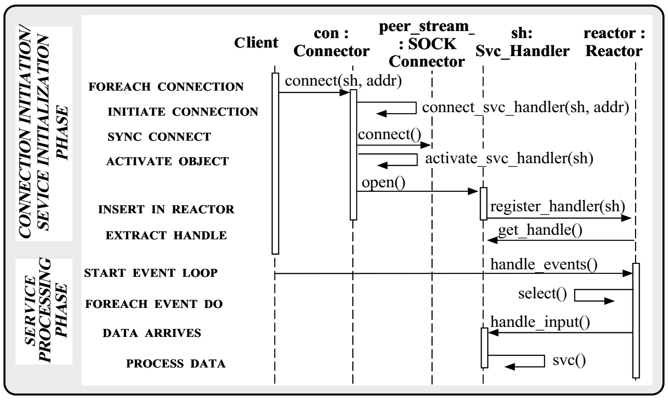
- *Intent*
  - "Decouple the active initialization of a service from the task performed once a service is initialized"
- This pattern resolves the following forces for network clients that use interfaces like sockets or TLI:

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 policies for creation, connection establishment, and concurrency*
4. *How to efficiently establish connections with large number of peers or over a long delay path*

Structure of the Connector Pattern

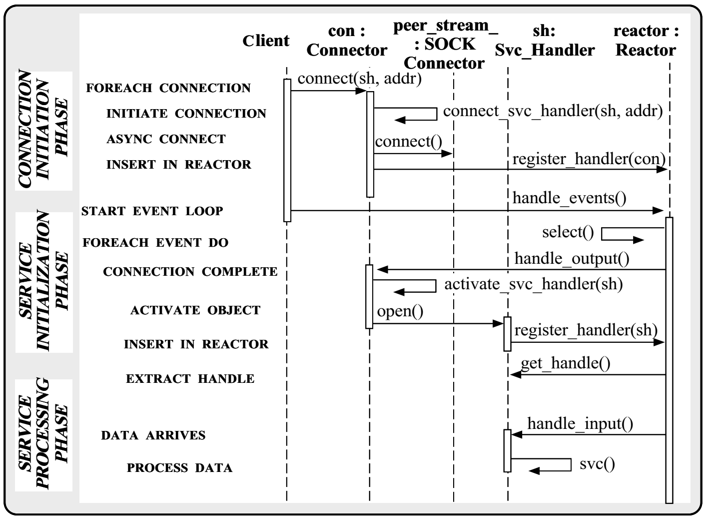


Collaboration in the Connector Pattern



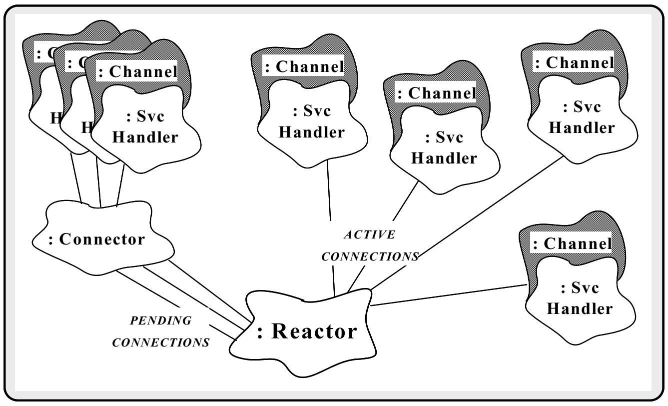
- Synchronous mode

Collaboration in the Connector Pattern



- Asynchronous mode

Using the Connector Pattern for the Gateway



Connector Class Public Interface

- A reusable template factory class that establishes connections with clients

```
template <class SVC_HANDLER, // Type of service
          class PEER_CONNECTOR> // Connection factory
class Connector
  : public Service_Object
{
public:
    // Initiate connection to Peer.
  virtual int connect (SVC_HANDLER *svc_handler,
                       const PEER_CONNECTOR::PEER_ADDR &,
                       Synch_Options &synch_options);

    // Cancel a <svc_handler> that was
    // started asynchronously.
  virtual int cancel (SVC_HANDLER *svc_handler);
```


OO Design Interlude

- Q: *What is the Synch_Options class?*

- A: This allows callers to define the synchrony/asynchrony policies, *e.g.*,

```
class Synch_Options
{
  // Options flags for controlling synchronization.
  enum {
    USE_REACTOR = 1,
    USE_TIMEOUT = 2
  };

  Synch_Options (u_long options = 0,
                 const Time_Value &timeout
                   = Time_Value::zero,
                 const void *arg = 0);

    // This is the default synchronous setting.
  static Synch_Options synch;
    // This is the default asynchronous setting.
  static Synch_Options asynch;
};
```


Connector Class Protected Interface

```
protected:
    // Demultiplexing hooks.
  virtual int handle_output (HANDLE); // Success.
  virtual int handle_input (HANDLE); // Failure.
  virtual int handle_timeout (Time_Value &, const void *);

    // Create and cleanup asynchronous connections...
  virtual int create_svc_tuple (SVC_HANDLER *,
                                Synch_Options &);
  virtual Svc_Tuple *cleanup_svc_tuple (HANDLE);

    // Table that maps an I/O handle to a Svc_Tuple *.
  Map_Manager<HANDLE, Svc_Tuple *, Null_Mutex>
    handler_map_;

    // Factory that actively establishes connections.
  PEER_CONNECTOR connector_;
};
```

OO Design Interlude

- Q: "What is a good technique to implementing a handler map?"
  - *e.g.*, to route messages or to map HANDLEs to SVC_HANDLERs

- A: Use a `Map_Manager` collection class
  - ACE provides a `Map_Manager` collection that associates *external ids* with *internal ids*, *e.g.*,
    - ▹ External ids → routing ID or HANDLE
    - ▹ Internal ids → set of `Channel` * or `Svc_Handler`
  - `Map_Manager` uses templates to enhance reuse

Map_Manager Class

- Synchronization mechanisms are parameterized...

```
template <class EXT_ID, class INT_ID, class LOCK>
class Map_Manager
{
public:
  bool bind (EXT_ID, INT_ID *);
  bool unbind (EXT_ID);

  bool find (EXT_ID ex, INT_ID &in) {
    // Exception-safe code...
    Read_Guard<LOCK> monitor (lock_);
    // lock_.read_acquire ();
    if (locate_entry (ex, in))
      return true;
    else
      return false;
    // lock_.release ();
 }

private:
  LOCK lock_;
  bool locate_entry (EXT_ID, INT_ID &);
  // ...
};
```


Connector Class Implementation

```
// Shorthand names.
#define SH SVC_HANDLER
#define PC PEER_CONNECTOR

// Initiate connection using specified blocking semantics.
template <class SH, class PC> int
Connector<SH, PC>::connect
  (SH *sh, const PC::PEER_ADDR &r_addr, Synch_Options &options)
{
  Time_Value *timeout = 0;
  int use_reactor = options[Synch_Options::USE_REACTOR];

  if (use_reactor) timeout = Time_Value::zerop;
  else
    timeout = options[Synch_Options::USE_TIMEOUT]
      ? (Time_Value *) &options.timeout () : 0;

  // Use Peer_Connector factory to initiate connection.
  if (connector_.connect (*sh, r_addr, timeout) == -1)
  {
    // If the connection hasn't completed, then
    // register with the Reactor to call us back.
    if (use_reactor && errno == EWOULDBLOCK)
      create_svc_tuple (sh, options);
  } else
    // Activate immediately if we are connected.
    sh->open ((void *) this);
}
```



```
// Register a Svc_Handler that is in the
// process of connecting.

template <class SH, class PC> int
Connector<SH, PC>::create_svc_tuple
  (SH *sh, Synch_Options &options)
{
  // Register for both "read" and "write" events.
  Service_Config::reactor ()->register_handler
      (sh->get_handle (),
       Event_Handler::READ_MASK |
       Event_Handler::WRITE_MASK);

  Svc_Tuple *st = new Svc_Tuple (sh, options.arg ());

  if (options[Synch_Options::USE_TIMEOUT])
    // Register timeout with Reactor.
    int id = Service_Config::reactor ()->schedule_timer
               (this, (const void *) st,
                options.timeout ());
    st->id (id);

  // Map the HANDLE to the Svc_Handler.
  handler_map_.bind (sh->get_handle (), st);
}
```

```
// Cleanup asynchronous connections...

template <class SH, class PC> Svc_Tuple *
Connector<SH, PC>::cleanup_svc_tuple (HANDLE h)
{
  Svc_Tuple *st;

  // Locate the Svc_Tuple based on the handle;
  handler_map_.find (h, st);

  // Remove SH from Reactor's Timer_Queue.
  Service_Config::reactor ()->cancel_timer (st->id ());

  // Remove HANDLE from Reactor.
  Service_Config::reactor ()->remove_handler (h,
    Event_Handler::RWE_MASK | Event_Handler::DONT_CALL);

  // Remove HANDLE from the map.
  handler_map_.unbind (h);
  return st;
}
```

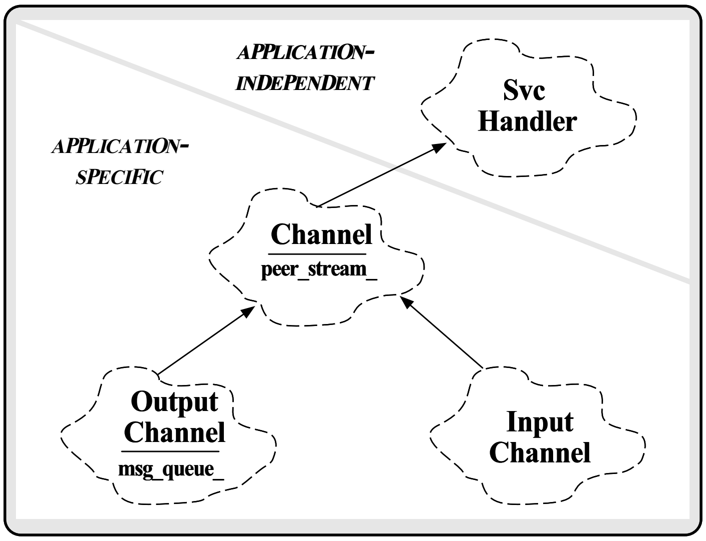


```
// Finalize a successful connection (called by Reactor).

template <class SH, class PC> int
Connector<SH, PC>::handle_output (HANDLE h) {
  Svc_Tuple *st = cleanup_svc_tuple (h);

  // Transfer I/O handle to SVC_HANDLE *.
  st->svc_handler ()->set_handle (h);

  // Delegate control to the service handler.
  sh->open ((void *) this);
}

// Handle connection errors.

template <class SH, class PC> int
Connector<SH, PC>::handle_input (HANDLE h) {
  Svc_Tuple *st = cleanup_svc_tuple (h);
}

// Handle connection timeouts.

template <class SH, class PC> int
Connector<SH, PC>::handle_timeout
  (Time_Value &time, const void *arg) {
  Svc_Tuple *st = (Svc_Tuple *) arg;
  st = cleanup_svc_tuple
        (st->svc_handler ()->get_handle ());
  // Forward "magic cookie"...
  st->svc_handler ()->handle_timeout (tv, st->arg ());
}
```


Specializing Connector and Svc_Handler

- Producing an application that meets Gateway requirements involves *specializing* ACE components
  - Connector → Channel_Connector
  - Svc_Handler → Channel → Input_Channel and Output_Channel
- Note that these new classes selectively override methods defined in the base classes
  - The Reactor automatically invokes these methods in response to I/O, signal, and timer events

Channel Inheritance Hierarchy



Channel Class Public Interface

- Common methods and data for I/O Channels

```
// Determine the type of threading mechanism.
#if defined (ACE_USE_MT)
typedef MT_SYNCH SYNCH;
#else
typedef NULL_SYNCH SYNCH;
#endif /* ACE_USE_MT */

// This is the type of the Routing_Table.
typedef Routing_Table <Peer_Addr,
                       Routing_Entry,
                       SYNCH::MUTEX> ROUTING_TABLE;

class Channel
  : public Svc_Handler<SOCK_Stream, SYNCH>
{
public:
    // Initialize the handler (called by Connector).
  virtual int open (void * = 0);

    // Bind addressing info to Router.
  virtual int bind (const INET_Addr &, CONN_ID);
```


OO Design Interlude

- Q: *What is the MT_SYNCH class and how does it work?*

- A: *MT_SYNCH provides a thread-safe synchronization policy for a particular instantiation of a Svc_Handler*
  - e.g., it ensures that any use of a `Svc_Handler`'s `Message_Queue` will be thread-safe
  - Any `Task` that accesses shared state can use the "traits" in the `MT_SYNCH`

```
class MT_SYNCH { public:
  typedef Mutex MUTEX;
  typedef Condition<Mutex> CONDITION;
};
```

  - Contrast with `NULL_SYNCH`

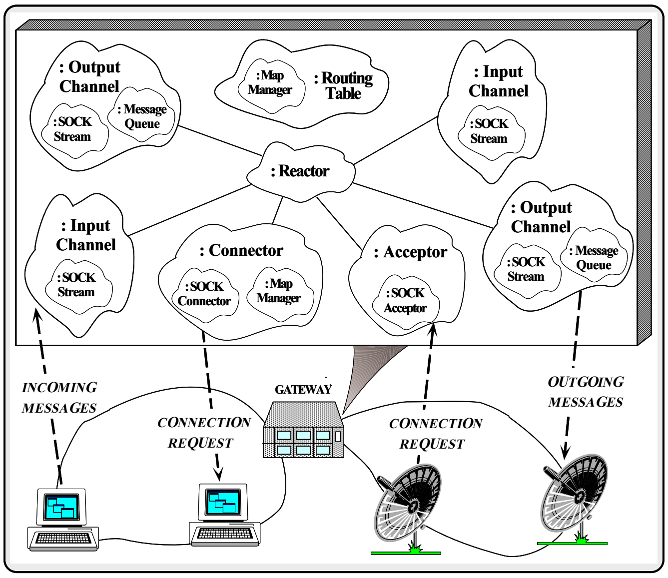
```
class NULL_SYNCH { public:
  typedef Null_Mutex MUTEX;
  typedef Null_Condition<Null_Mutex> CONDITION;
};
```


Channel Class Protected Interface

- Common data for I/O Channels

```
protected:
    // Reconnect Channel if connection terminates.
  virtual int handle_close (HANDLE, Reactor_Mask);

    // Address of peer.
  INET_Addr addr_;

    // The assigned connection ID of this Channel.
  CONN_ID id_;
};
```


Detailed OO Architecture of the Gateway



Input_Channel Interface

- Handle input processing and routing of messages from Peers

```
class Input_Channel : public Channel
{
public:
  Input_Channel (void);

protected:
    // Receive and process Peer messages.
  virtual int handle_input (HANDLE);

    // Receive a message from a Peer.
  virtual int recv_peer (Message_Block *&);

    // Action that routes a message from a Peer.
  int route_message (Message_Block *);

    // Keep track of message fragment.
  Message_Block *msg_frag_;
};
```


Output_Channel Interface

- Handle output processing of messages sent to Peers

```
class Output_Channel : public Channel
{
public:
  Output_Channel (void);

    // Send a message to a Gateway (may be queued).
  virtual int put (Message_Block *, Time_Value * = 0);

protected:
    // Perform a non-blocking put().
  int nonblk_put (Message_Block *mb);

    // Finish sending a message when flow control abates.
  virtual int handle_output (HANDLE);

    // Send a message to a Peer.
  virtual int send_peer (Message_Block *);
};
```


Channel_Connector Class Interface

- A Concrete factory class that behaves as follows:

  1. Establishes connections with Peers to produce `Channels`

  2. Activates `Channels`, which then do the work

```
class Channel_Connector : public
  Connector <Channel, // Type of service
             SOCK_Connector> // Connection factory
{
public:
  // Initiate (or reinitiate) a connection on Channel.
  int initiate_connection (Channel *);
}
```

- `Channel_Connector` also ensures reliability by restarting failed connections

Channel_Connector Implementation

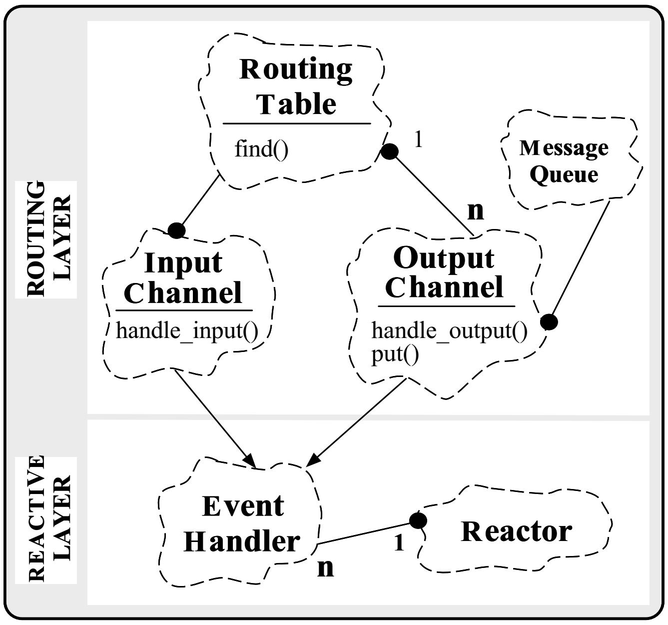
- Initiate (or reinitiate) a connection to the Channel

```
int
Channel_Connector::initiate_connection (Channel *channel)
{
  // Use asynchronous connections...
  if (connect (channel, channel->addr (),
                   Synch_Options::asynch) == -1) {
    if (errno == EWOULDBLOCK)
      return -1; // We're connecting asynchronously.
    else
      // Failure, reschedule ourselves to try again later.
      Service_Config::reactor ()->schedule_timer
        (channel, 0, channel->timeout ());
  }
  else
    // We're connected.
    return 0;
}
```

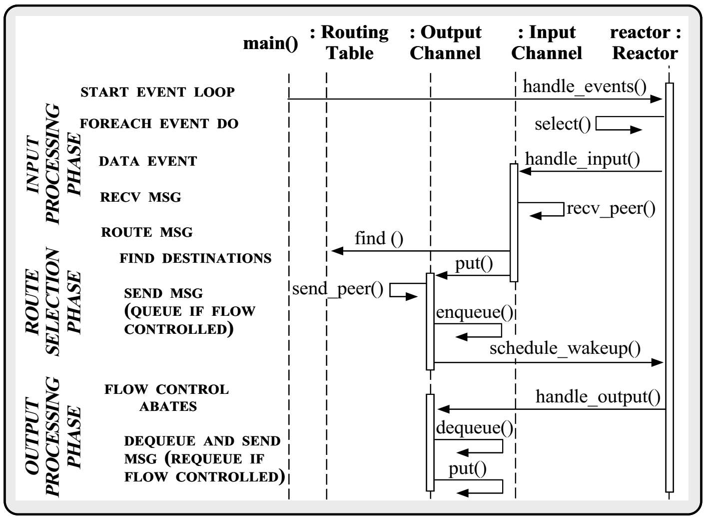
The Router Pattern

- *Intent*
  - "Decouple multiple sources of input from multiple sources of output to prevent blocking"
- The Router pattern resolves the following forces for connection-oriented routers:
  - *How to prevent misbehaving connections from disrupting the quality of service for well-behaved connections*
  - *How to allow different concurrency strategies for Input and Output Channels*

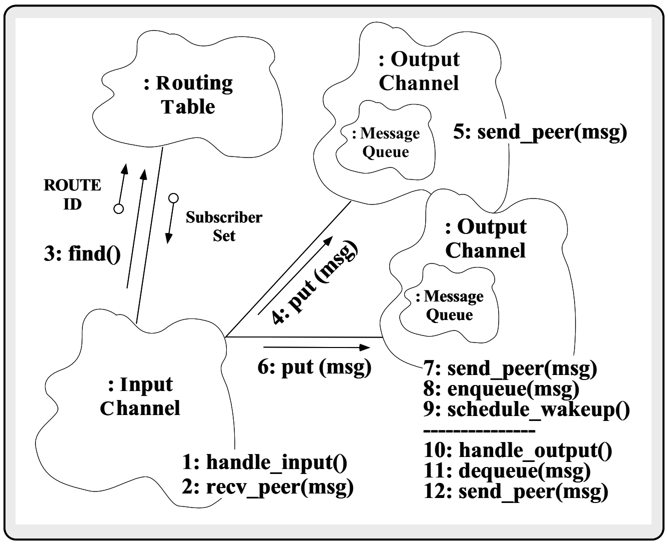
Structure of the Router Pattern



Collaboration in the Router Pattern



Collaboration in Single-threaded Gateway Routing

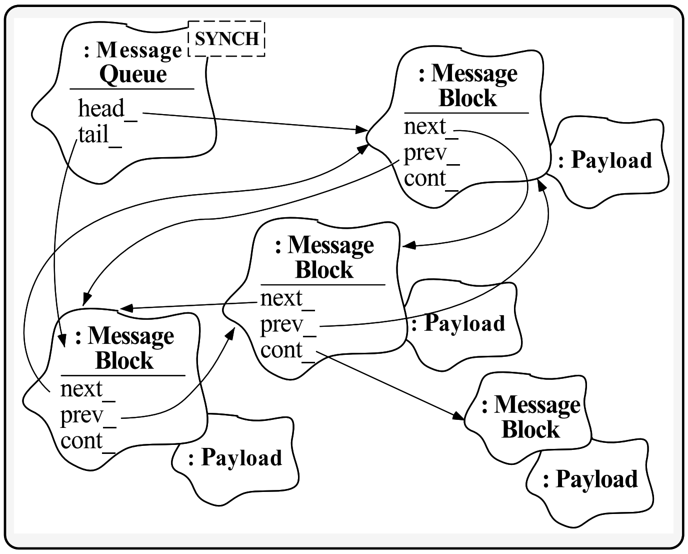


```
// Receive input message from Peer and route
// the message.

int
Input_Channel::handle_input (HANDLE)
{
  Message_Block *route_addr = 0;

  // Try to get the next message.
  if ((n = recv_peer (route_addr)) <= 0) {
    if (errno == EWOULDBLOCK) return 0;
    else return n;
  }
  else
    route_message (route_addr);
}

// Send a message to a Peer (queue if necessary).

int
Output_Channel::put (Message_Block *mb, Time_Value *)
{
  if (msg_queue_->is_empty ())
    // Try to send the message *without* blocking!
    nonblk_put (mb);
  else
    // Messages are queued due to flow control.
    msg_queue_->enqueue_tail (mb, Time_Value::zerop);
}
```



```
// Route message from a Peer.

int
Input_Channel::route_messages (Message_Block *route_addr)
{
  // Determine destination address.
  CONN_ID route_id = *(CONN_ID *) route_addr->rd_ptr ();

  const Message_Block *const data = route_addr->cont ();
  Routing_Entry *re = 0;

  // Determine route.
  Routing_Table::instance ()->find (route_id, re);

  // Initialize iterator over destination(s).
  Set_Iterator<Channel *> si (re->destinations ());

  // Multicast message.
  for (Channel *out_ch;
       si.next (out_ch) != -1;
       si.advance ()) {
    Message_Block *newmsg = data->duplicate ();
    if (out_ch->put (newmsg) == -1) // Drop message.
      newmsg->release (); // Decrement reference count.
  }
  delete route_addr;
}
```


Message_Queue and Message_Block Object Diagram



Peer_Message

```
// unique connection id that denotes a Channel.
typedef short CONN_ID;

// Peer address is used to identify the
// source/destination of a Peer message.
class Peer_Addr {
public:
  CONN_ID conn_id_; // Unique connection id.
  u_char logical_id_; // Logical ID.
  u_char payload_; // Payload type.
};

// Fixed sized header.
class Peer_Header { public: /* ... */ };

// Variable-sized message (sdu_ may be
// between 0 and MAX_MSG_SIZE).

class Peer_Message {
public:
    // The maximum size of a message.
  enum { MAX_PAYLOAD_SIZE = 1024 };
  Peer_Header header_; // Fixed-sized header portion.
  char sdu_[MAX_PAYLOAD_SIZE]; // Message payload.
};
```

OO Design Interlude

- Q: *What should happen if put() fails?*
  - *e.g.*, if a queue becomes full?
- A: The answer depends on whether the error handling policy is different for each router object or the same...
  - Bridge pattern: *give reasonable default, but allow substitution via subclassing*
- A related design issue deals with avoiding output blocking if a Peer connection becomes flow controlled


```
// Pseudo-code for receiving framed message
// (using non-blocking I/O).

int
Input_Channel::recv_peer (Message_Block *&route_addr)
{
  if (msg_frag_ is empty) {
    msg_frag_ = new Message_Block;
    receive fixed-sized header into msg_frag_
    if (errors occur)
      cleanup
    else
      determine size of variable-sized msg_frag_
  }
  else
    determine how much of msg_frag_ to skip

  perform non-blocking recv of payload into msg_frag_
  if (entire message is now received) {
    route_addr = new Message_Block (sizeof (Peer_Addr),
                                    msg_frag_)
    Peer_Addr addr (id (), msg_frag_->routing_id_, 0);
    route_addr->copy (&addr, sizeof (Peer_Addr));
    return to caller and reset msg_frag_
  }
  else if (only part of message is received)
    return errno = EWOULDBLOCK
  else if (fatal error occurs)
    cleanup
}
```


OO Design Interlude

- Q: *How can a flow controlled Output_Channel know when to proceed again without polling or blocking?*
- A: *Use the Event_Handler::handle_output notification scheme of the Reactor*
  - *i.e.*, via the `Reactor`'s methods `schedule_wakeup` and `cancel_wakeup`
- This provides cooperative multi-tasking within a single thread of control
  - The `Reactor` calls back to the `handle_output` method when the `Channel` is able to transmit again

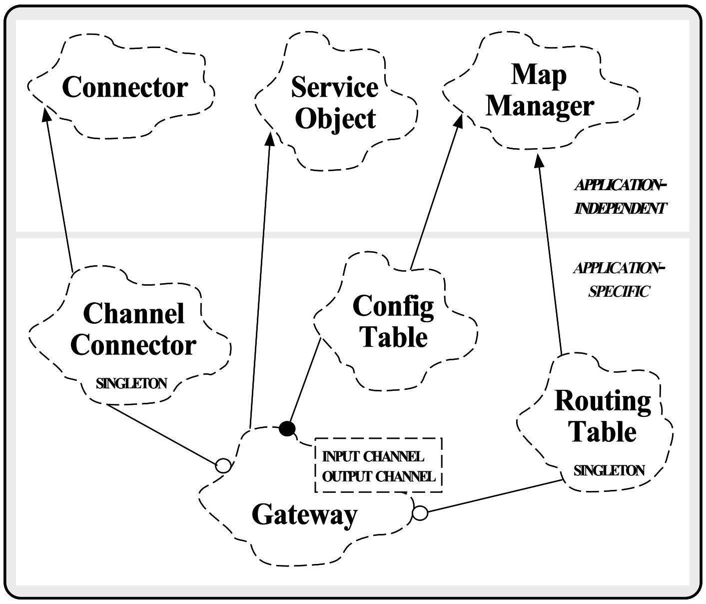

```
// Perform a non-blocking put() of message MB.

int Output_Channel::nonblk_put (Message_Block *mb)
{
  // Try to send the message using non-blocking I/O
  if (send_peer (mb) != -1
      && errno == EWOULDBLOCK)
  {
    // Queue in *front* of the list to preserve order.
    msg_queue_->enqueue_head (mb, Time_Value::zerop);

    // Tell Reactor to call us back when we can send again.

    Service_Config::reactor ()->schedule_wakeup
      (this, Event_Handler::WRITE_MASK);
  }
}
```

```
// Simple implementation...

int
Output_Channel::send_peer (Message_Block *mb)
{
  ssize_t n;
  size_t len = mb->length ();

  // Try to send the message.
  n = peer ().send (mb->rd_ptr (), len);

  if (n <= 0)
    return errno == EWOULDBLOCK ? 0 : n;
  else if (n < len)
    // Skip over the part we did send.
    mb->rd_ptr (n);
  else /* if (n == length) */ {
    delete mb; // Decrement reference count.
    errno = 0;
  }
  return n;
}
```



```
// Finish sending a message when flow control
// conditions abate.  This method is automatically
// called by the Reactor.

int
Output_Channel::handle_output (HANDLE)
{
  Message_Block *mb = 0;

  // Take the first message off the queue.
  msg_queue_->dequeue_head
                  (mb, Time_Value::zerop);
  if (nonblk_put (mb) != -1
      || errno != EWOULDBLOCK) {
    // If we succeed in writing msg out completely
    // (and as a result there are no more msgs
    // on the Message_Queue), then tell the Reactor
    // not to notify us anymore.

    if (msg_queue_->is_empty ()
      Service_Config::reactor ()->cancel_wakeup
        (this, Event_Handler::WRITE_MASK);
  }
}
```


The Gateway Class



- This class integrates other application-specific and application-independent components

Gateway Class Public Interface

- Since `Gateway` inherits from `Service_Object` it may be dynamically (re)configured into a process at run-time

```
// Parameterized by the type of I/O channels.
template <class INPUT_CHANNEL, // Input policies
          class OUTPUT_CHANNEL> // Output policies
class Gateway
  : public Service_Object
{
public:

    // Perform initialization.
  virtual int init (int argc, char *argv[]);

    // Perform termination.
  virtual int fini (void);
```

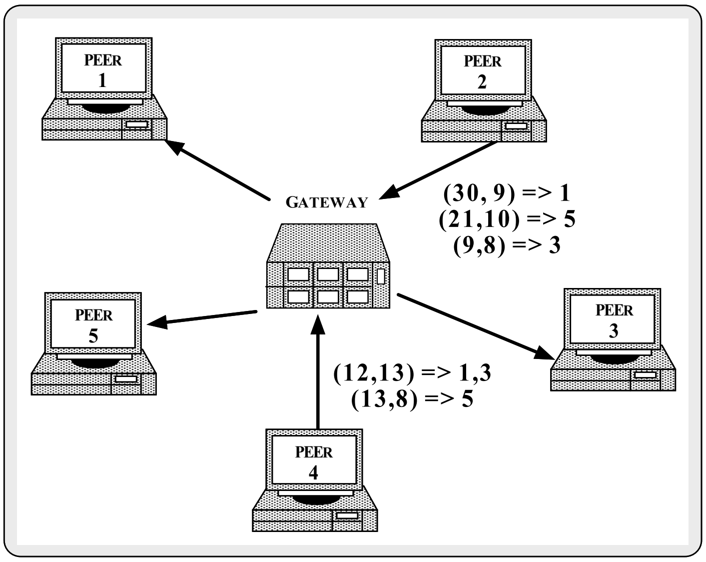
Gateway Class Private Interface

```
protected:
    // Parse the channel table configuration file.
  int parse_cc_config_file (void);

    // Parse the routing table configuration file.
  int parse_rt_config_file (void);

    // Initiate connections to the Peers.
  int initiate_connections (void);

  // Table that maps Connection IDs to Channel *'s.
  Map_Manager<CONN_ID, Channel *, Null_Mutex>
    config_table_;
};
```



```
// Convenient short-hands.
#define IC INPUT_CHANNEL
#define OC OUTPUT_CHANNEL

// Pseudo-code for initializing the Gateway (called
// automatically on startup).

template <class IC, class OC>
Gateway<IC, OC>::init (int argc, char *argv[])
{
  // Parse command-line arguments.
  parse_args (argc, argv);

  // Parse and build the connection configuration.
  parse_cc_config_file ();

  // Parse and build the routing table.
  parse_rt_config_file ();

  // Initiate connections with the Peers.
  initiate_connections ();
  return 0;
}
```


Configuration and Gateway Routing



Configuration Files

- The Gateway decouples the connection topology from the peer routing topology
  - The following config file specifies the connection topology among the Gateway and its Peers

```
# Conn ID  Hostname         Port   Direction  Max Retry
# -------  --------         ----   ---------  ---------
    1      peer1            10002  O          32
    2      peer2            10002  I          32
    3      peer3            10002  O          32
    4      peer4            10002  I          32
    5      peer5            10002  O          32
```

  - The following config file specifies the routing topology among the Gateway and its Peers

```
# Conn ID  Logical ID  Payload  Destinations
# -------  ----------  -------  ------------
  2        30          9        1
  2        21          10       5
  2        09          8        3
  4        12          13       1,3
  4        13          8        5
```

```
// Parse the cc_config_file and
// build the connection table.

template <class IC, class OC>
Gateway<IC, OC>::parse_cc_config_file (void)
{
  CC_Entry entry;
  cc_file.open (cc_filename);

  // Example of the Builder Pattern.

  while (cc_file.read_line (entry) {
    Channel *ch;

    // Locate/create routing table entry.
    if (entry.direction_ == 'O')
      ch = new OC;
    else
      ch = new IC;

    // Set up the peer address.
    INET_Addr addr (entry.port_, entry.host_);
    ch->bind (addr, entry.conn_id_);
    ch->max_timeout (entry.max_retry_delay_);
    config_table_.bind (entry.conn_id_, ch);
  }
}
```



```
// Parse the rt_config_file and
// build the routing table.

template <class IC, class OC>
Gateway<IC, OC>::parse_rt_config_file (void)
{
  RT_Entry entry;
  rt_file.open (cc_filename);

  // Example of the Builder Pattern.

  while (cc_file.read_line (entry) {
    Routing_Entry *re = new Routing_Entry;
    Peer_Addr peer_addr (entry.conn_id, entry.logical_id_);
    Set<Channel *> *channel_set = new Set<Channel *>;

    // Example of the Iterator pattern.
    foreach destination_id in entry.total_destinations_ {
      Channel *ch;
      if (config_table_.find (destination_id, ch);
        channel_set->insert (ch);
    }

    // Attach set of destination channels to routing entry.
    re->destinations (channel_set);

    // Bind with routing table, keyed by peer address.
    routing_table.bind (peer_addr, re);
  }
}
```



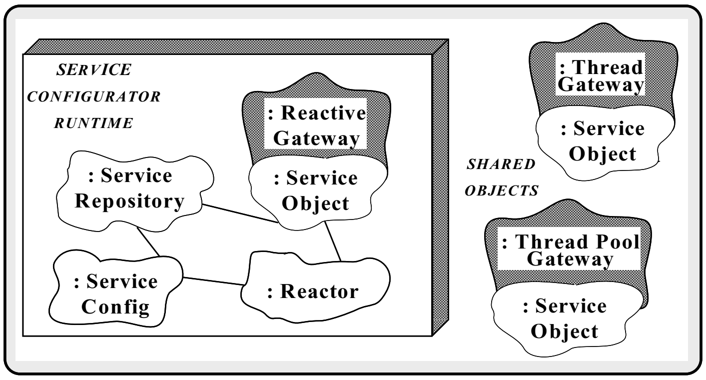
```
// Initiate connections with the Peers.

int Gateway<IC, OC>::initiate_connections (void)
{
  // Example of the Iterator pattern.
  Map_Iterator<CONN_ID, Channel *, Null_Mutex>
    cti (connection_table_);

  // Iterate through connection table
  // and initiate all channels.

  for (const Map_Entry <CONN_ID, Channel *> *me = 0;
       cti.next (me) != 0;
       cti.advance ()) {
    Channel *channel = me->int_id_;

    // Initiate non-blocking connect.
    Channel_Connector::instance ()->
      initiate_connection (channel);
  }
  return 0;
}
```


Dynamically Configuring Services into an Application

- Main program is generic

```
// Example of the Service Configurator pattern.

int main (int argc, char *argv[])
{
  Service_Config daemon;
  // Initialize the daemon and
  // dynamically configure services.
  daemon.open (argc, argv);

  // Run forever, performing configured services.

  daemon.run_reactor_event_loop ();

  /* NOTREACHED */
}
```

Using the Service Configurator Pattern for the Gateway



- Replace the single-threaded Gateway with a multi-threaded Gateway

Dynamic Linking a Gateway Service

- Service configuration file

```
% cat ./svc.conf
static Svc_Manager "-p 5150"
dynamic Gateway_Service Service_Object *
        Gateway.dll:make_Gateway () "-d"
```

- Application-specific factory function used to dynamically link a service

```
// Dynamically linked factory function that allocates
// a new single-threaded Gateway object.

extern "C" Service_Object *make_Gateway (void);

Service_Object *
make_Gateway (void)
{
  return new Gateway<Input_Channel, Output_Channel>;
  // ACE automatically deletes memory.
}
```

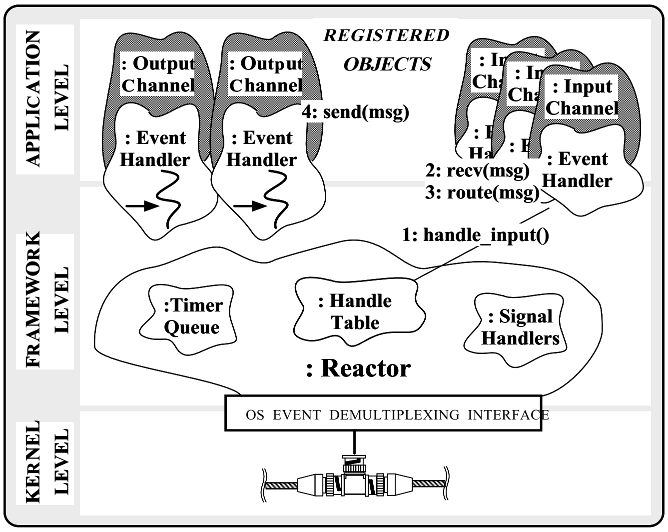

Concurrency Strategies for Patterns

- The Acceptor and Connector patterns do not constrain the concurrency strategies of a `Svc_Handler`

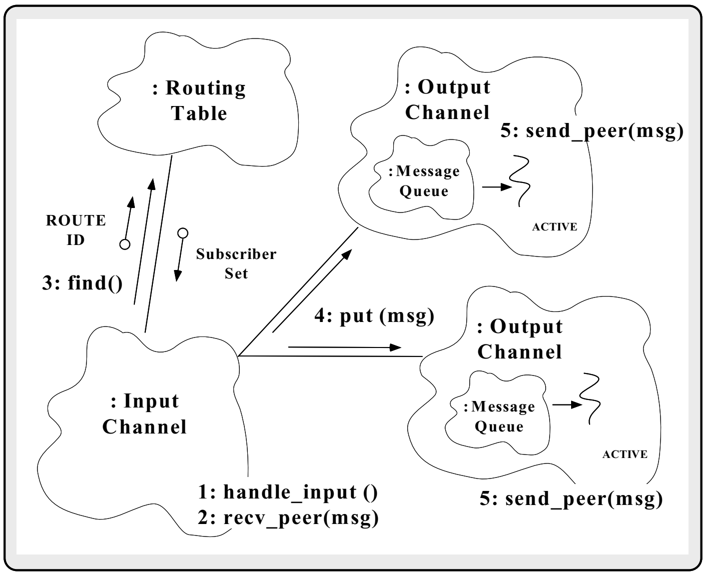
- There are three common choices:
  1. *Run service in same thread of control*
  2. *Run service in a separate thread*
  3. *Run service in a separate process*

- Observe how OO techniques push this decision to the "edges" of the design
  - This greatly increases reuse, flexibility, and performance tuning

Using the Active Object Pattern for the Multi-threaded Gateway



Collaboration in the Active Object-based Gateway Routing

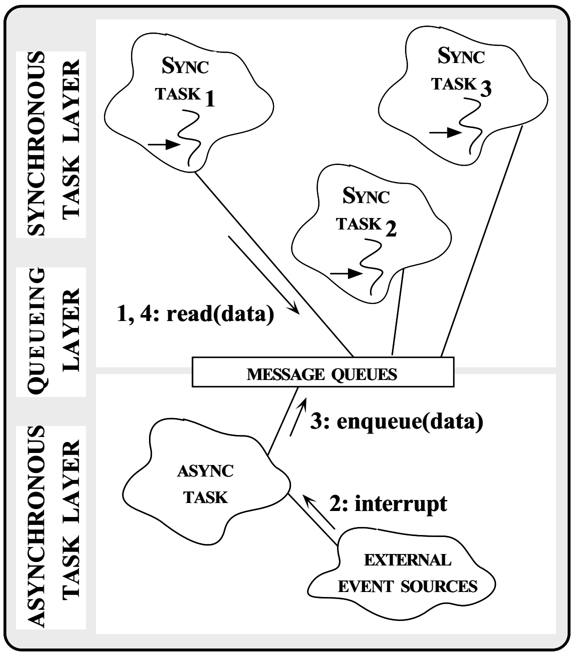


Half-Sync/Half-Async Pattern

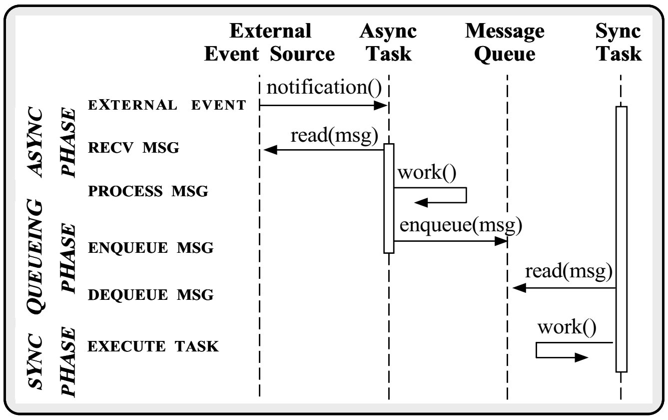
- *Intent*
  - "Decouple synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency"

- This pattern resolves the following forces for concurrent communication systems:
  - *How to simplify programming for higher-level communication tasks*
    - ▹ These are performed synchronously (via Active Objects)
  - *How to ensure efficient lower-level I/O communication tasks*
    - ▹ These are performed asynchronously (via Reactor)

Structure of the Half-Sync/Half-Async Pattern

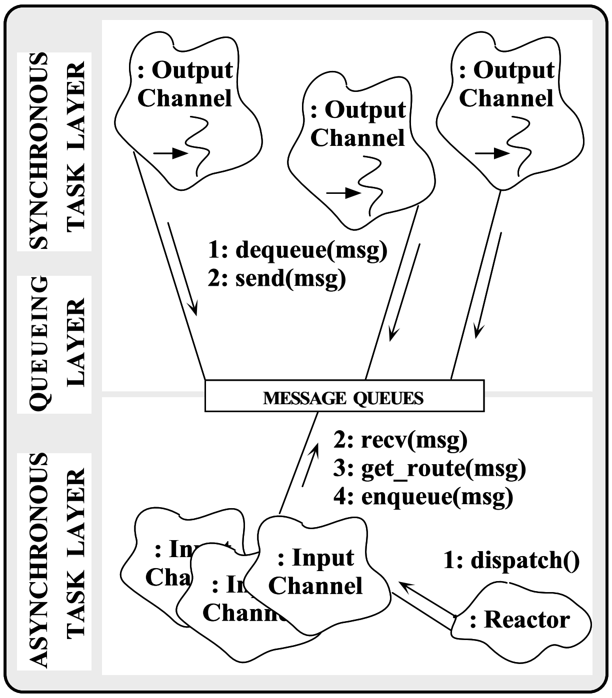


Collaborations in the Half-Sync/Half-Async Pattern

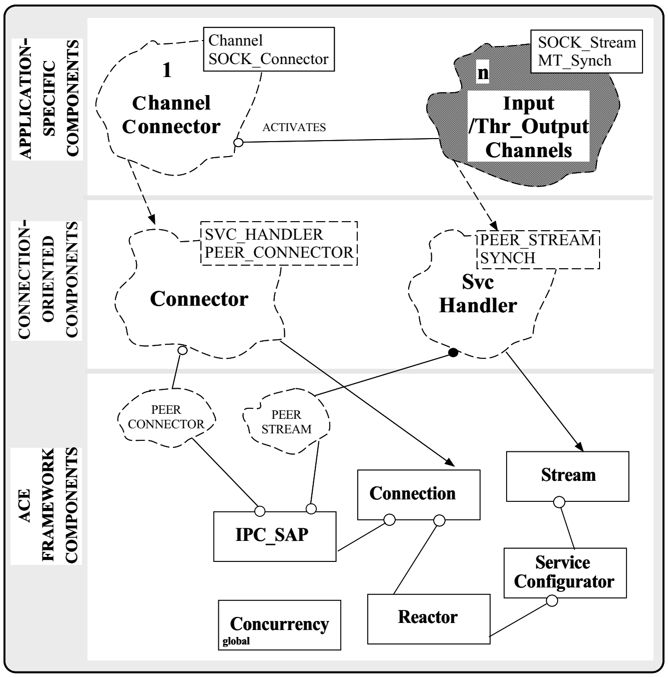


- This illustrates *input* processing (*output* processing is similar)

Using the Half-Sync/Half-Async Pattern in the Gateway



Class Diagram for Multi-Threaded Gateway



Thr_Output_Channel Class Interface

- New subclass of `Channel` uses the Active Object pattern for the `Output_Channel`
  - Uses multi-threading and synchronous I/O (rather than non-blocking I/O) to transmit message to Peers
  - Transparently improve performance on a multi-processor platform and simplify design

```
#define ACE_USE_MT
#include "Channel.h"

class Thr_Output_Channel : public Output_Channel
{
public:
    // Initialize the object and spawn a new thread.
  virtual int open (void *);

    // Send a message to a peer.
  virtual int put (Message_Block *, Time_Value * = 0);

    // Transmit peer messages within separate thread.
  virtual int svc (void);
};
```


Thr_Output_Channel Class Implementation

- The multi-threaded version of `open` is slightly different since it spawns a new thread to become an active object!

```
// Override definition in the Output_Channel class.

int
Thr_Output_Channel::open (void *)
{
  // Become an active object by spawning a
  // new thread to transmit messages to Peers.

  activate (THR_NEW_LWP | THR_DETACHED);
}
```

- `activate` is a pre-defined method on class `Task`

```
// Queue up a message for transmission (must not block
// since all Input_Channels are single-threaded).

int
Thr_Output_Channel::put (Message_Block *mb, Time_Value *)
{
  // Perform non-blocking enqueue.
  msg_queue_->enqueue_tail (mb, Time_Value::zerop);
}

// Transmit messages to the peer (note simplification
// resulting from threads...)

int
Thr_Output_Channel::svc (void)
{
  Message_Block *mb = 0;

  // Since this method runs in its own thread it
  // is OK to block on output.

  while (msg_queue_->dequeue_head (mb) != -1)
    send_peer (mb);

  return 0;
}
```


Dynamic Linking a Gateway Service

- Service configuration file

```
% cat ./svc.conf
remove Gateway_Service
dynamic Gateway_Service Service_Object *
        thr_Gateway.dll:make_Gateway () "-d"
```

- Application-specific factory function used to dynamically link a service

```
// Dynamically linked factory function that allocates
// a new multi-threaded Gateway object.

extern "C" Service_Object *make_Gateway (void);

Service_Object *
make_Gateway (void)
{
  return new Gateway<Input_Channel, Thr_Output_Channel>;
  // ACE automatically deletes memory.
}
```


Lessons Learned using OO Design Patterns

- *Benefits of patterns*
  - *Enable large-scale reuse of software architectures*
  - *Improve development team communication*
  - *Help transcend language-centric viewpoints*
- *Drawbacks of patterns*
  - *Do not lead to direct code reuse*
  - *Can be deceptively simple*
  - *Teams may suffer from pattern overload*

Lessons Learned using OO Frameworks

- *Benefits of frameworks*
  - Enable direct reuse of code (*cf* patterns)
  - Facilitate larger amounts of reuse than stand-alone functions or individual classes
- *Drawbacks of frameworks*
  - High initial learning curve
    - ▹ Many classes, many levels of abstraction
  - The flow of control for reactive dispatching is non-intuitive

Lessons Learned using C++

- *Benefits of C++*
  - *Classes* and *namespaces* modularize the system architecture
  - *Inheritance* and *dynamic binding* decouple application *policies* from reusable *mechanisms*
  - *Parameterized types* decouple the reliance on particular types of synchronization methods or network IPC interfaces
- *Drawbacks of C++*
  - Many language features are not widely implemented
  - Development environments are primitive
  - Language has many dark corners and sharp edges

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

Patterns Literature

- *Books*
  - Gamma et al., "Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Wesley, 1994
  - *Pattern Languages of Program Design* series by Addison-Wesley, 1995 and 1996
  - Siemens, *Pattern-Oriented Software Architecture*, Wiley and Sons, 1996
- *Special Issues in Journals*
  - Dec. '96 "Theory and Practice of Object Systems" (guest editor: Stephen P. Berczuk)
  - October '96 "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 Martin

Relevant Conferences and Workshops

- Joint *Pattern Languages of Programs* Conferences
  - 3rd PLoP
    - ▷ September 4–6, 1996, Monticello, Illinois, USA
  - 1st EuroPLoP
    - ▷ July 10–14, 1996, Kloster Irsee, Germany
  - http://www.cs.wustl.edu/~schmidt/jointPLoP–96.html/
- USENIX COOTS
  - June 17–21, 1996, Toronto, Canada
  - http://www.cs.wustl.edu/~schmidt/COOTS–96.html/