Patterns and Frameworks for Concurrent Network Programming with ACE and C++

Douglas C. Schmidt

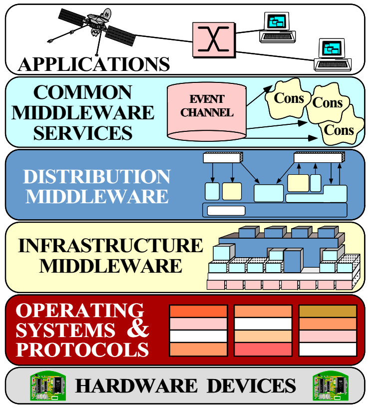
Associate Professor
schmidt@uci.edu
www.ece.uci.edu/~schmidt/

Elec. & Comp. Eng. Dept.
University of California, Irvine
(949) 824-1901

Sponsors

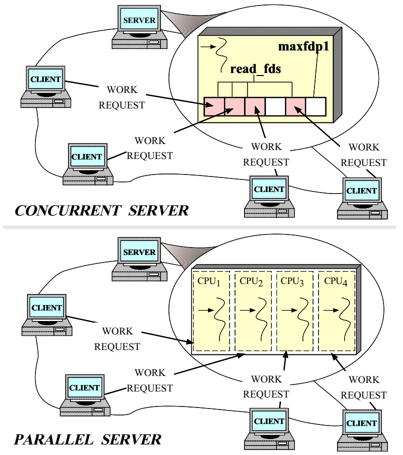
NSF, DARPA, ATD, BBN, Boeing, Cisco, Comverse, GDIS, Experian, Global MT, Hughes, Kodak, Krones, Lockheed, Lucent, Microsoft, Mitre, Motorola, NASA, Nokia, Nortel, OCI, Oresis, OTI, QNX, Raytheon, SAIC, Siemens SCR, Siemens MED, Siemens ZT, Sprint, Telcordia, USENIX

Roadmap to Levels of Middleware



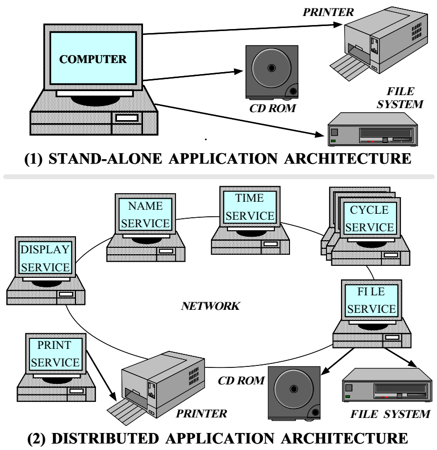
- **Observations**
  - Historically, apps built directly atop OS
  - Today, more and more apps built atop *middleware*
  - Middleware has several layers
- **General R&D challenges**
  - Performance optimizations
  - Quality of Service (QoS)
  - Software architecture & patterns

Motivation for Concurrency



- *Leverage hardware/software*
  - *e.g.*, multi-processors and OS thread support
- *Increase performance*
  - *e.g.*, overlap computation and communication
- *Improve response-time*
  - *e.g.*, GUIs and network servers
- *Simplify program structure*
  - *e.g.*, sync vs. async

Motivation for Distribution



- 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* concurrent networking applications is hard
  - *e.g.*, must address complex topics that are less problematic or not relevant for non-concurrent, stand-alone applications
- OO techniques and OO language features help to enhance software quality factors
  - Key OO techniques include *patterns* and *frameworks*
  - Key OO language features include *classes*, *inheritance*, *dynamic binding*, and *parameterized types*
  - Key software quality factors include *modularity*, *extensibility*, *portability*, *reusability*, and *correctness*

Caveats

- OO is *not* a panacea
  - However, when used properly it helps minimize “accidental” complexity and improve software quality factors
- Advanced OS features provide additional functionality and performance, *e.g.*,
  - *Multi-threading*
  - *Multi-processing*
  - *Synchronization*
  - *Shared memory*
  - *Explicit dynamic linking*
  - *Communication protocols and IPC mechanisms*

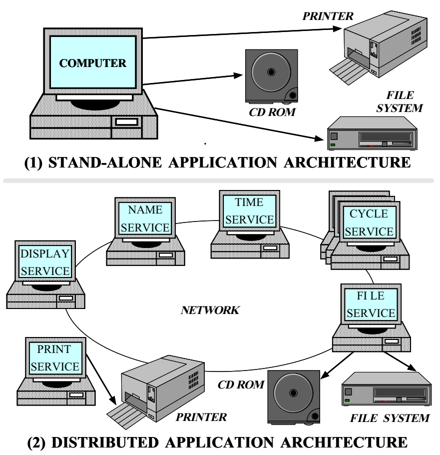
Tutorial Outline

- Brief overview of key OO networking and concurrency concepts and OS platform mechanisms
  - Emphasis is on *practical* solutions
- Examine several examples in detail
  - *Distributed Logger*
  - *Concurrent Web Server*
  - *Application-level Telecom Gateway*
  - *Call Center Manager Event Server*
- Discuss general concurrent programming strategies

Software Development Environment

- The topics discussed here are largely independent of OS, network, and programming language
  - Currently used successfully on UNIX/POSIX, Win32, and RTOS platforms, running on TCP/IP networks using C++
- Examples are illustrated using freely available ADAPTIVE Communication Environment (ACE) OO framework components
  - Although ACE is written in C++, the principles covered in this tutorial apply to other OO languages
  - *e.g.*, Java, Eiffel, Smalltalk, etc.
- In addition, other networks and backplanes can be used, as well

Sources of Complexity



- **Inherent complexity**
  - Latency
  - Reliability
  - Synchronization
  - Deadlock
- **Accidental Complexity**
  - Low-level APIs
  - Poor debugging tools
  - Algorithmic decomposition
  - Continuous re-invention

Sources of Inherent Complexity

Inherent complexity results from fundamental challenges, *e.g.*:

Concurrent programming

- Eliminating “race conditions”
- Deadlock avoidance
- Fair scheduling
- Performance optimization and tuning

Distributed programming

- Addressing the impact of latency
- Fault tolerance and high availability
- Load balancing and service partitioning
- Consistent ordering of distributed events

Sources of Accidental Complexity

Accidental complexity results from limitations with tools and techniques used to develop concurrent applications, *e.g.*,

- Lack of portable, reentrant, type-safe and extensible system call interfaces and component libraries
- Inadequate debugging support and lack of concurrent and distributed program analysis tools
- Widespread use of *algorithmic* decomposition
  - Fine for *explaining* concurrent programming concepts and algorithms but inadequate for *developing* large-scale concurrent network applications
- Continuous rediscovery and reinvention of core concepts and components

OO Contributions to Concurrent and Distributed Applications

Concurrent network programming is traditionally performed using low-level OS mechanisms, *e.g.*,

- *fork/exec*
- *Shared memory and semaphores*
- *Memory-mapped files*
- *Signals*
- *sockets/select*
- *Low-level thread APIs*

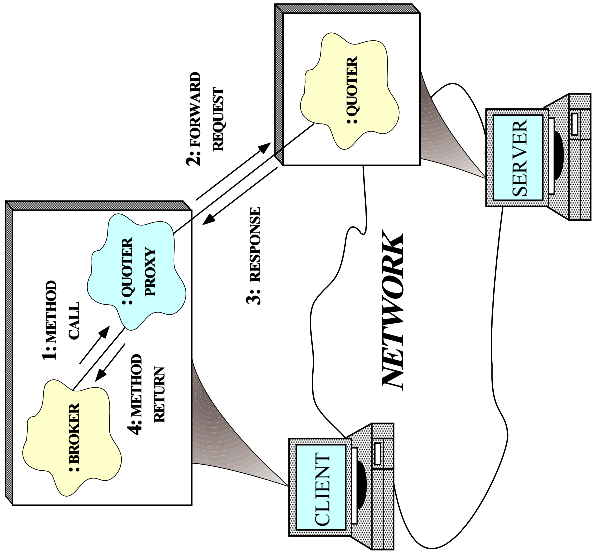
Patterns and *frameworks* elevate development level to focus on application concerns, *e.g.*,

- *Service functionality and policies*
- *Service configuration*
- *Concurrent event demultiplexing and event handler dispatching*
- *Service concurrency and synchronization*

Overview of Patterns

- 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

Example: the Proxy Pattern

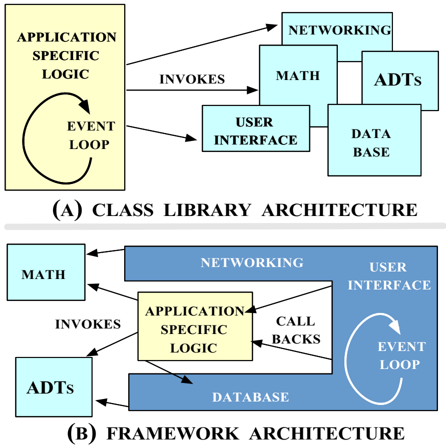


Intent: provide a surrogate for another object that controls access to it

Overview of Frameworks and Components

- 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"
- Frameworks facilitate reuse of successful communication software designs and implementations
  - Applications *inherit* from and *instantiate* framework components

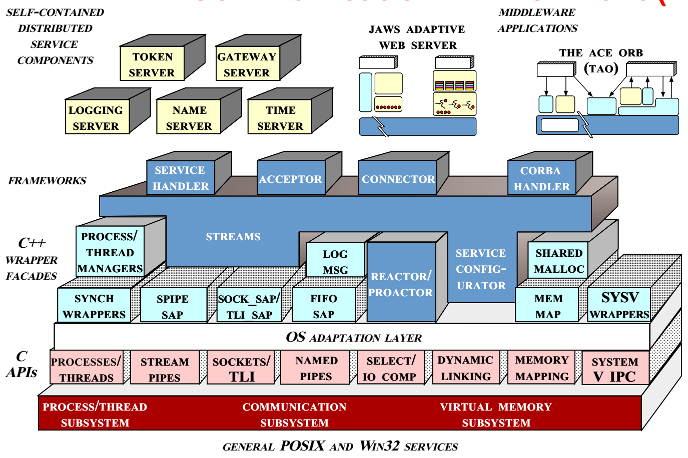
Class Libraries, Frameworks, and Components



Definition

- *Components*
  - Self-contained, "pluggable" ADTs
- *Frameworks*
  - Reusable, "semi-complete" applications
- *Patterns*
  - Problem, solution, context

The ADAPTIVE Communication Environment (ACE)

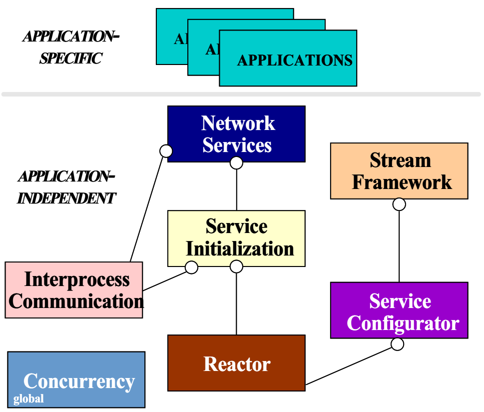


www.cs.wustl.edu/~schmidt/ACE.html

ACE Statistics

- ACE contains > 200,000 lines of C++
  - Over 25 person-years of effort
- Ported to UNIX, Win32, MVS, and RT/embedded platforms
  - *e.g.*, VxWorks, LynxOS, Chorus
- Large user community
  - ~schmidt/ACE-users.html
- Currently used by dozens of companies
  - Bellcore, BBN, Boeing, Ericsson, Hughes, Kodak, Lockheed, Lucent, Motorola, Nokia, Nortel, Raytheon, SAIC, Siemens, etc.
- Supported commercially by Riverace
  - www.riverace.com

Class Categories in ACE

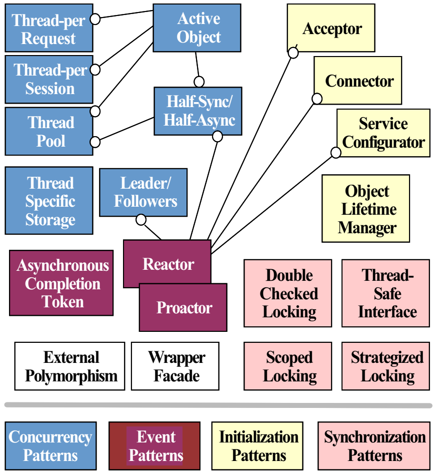


- ACE is structured as a "forest" of class categories
- This design permits fine-grained subsetting of ACE components
  - Subsetting helps minimize ACE's memory footprint

ACE Class Category Responsibilities

- `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 demuxing* and *event handler dispatching*
- `Service Configurator` automates *(re)configuration* 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

Patterns for Communication Middleware



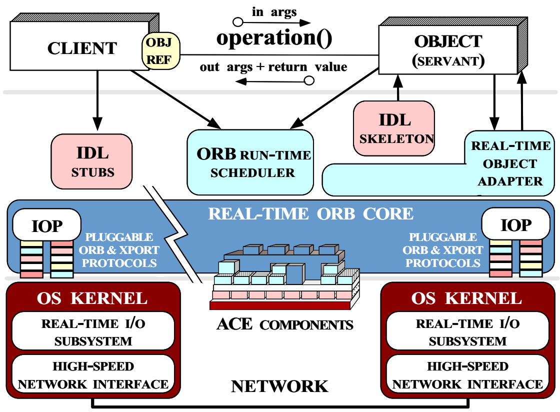
Observation

- *Failures rarely result from unknown scientific principles, but from failing to apply proven engineering practices and patterns*

Benefits of Patterns

- Facilitate design reuse
- Preserve crucial design information
- Guide design choices

The ACE ORB (TAO)



www.cs.wustl.edu/~schmidt/TAO.html

TAO Overview →

- A real-time, high-performance ORB
- Leverages ACE
  - Runs on POSIX, Win32, RTOSs

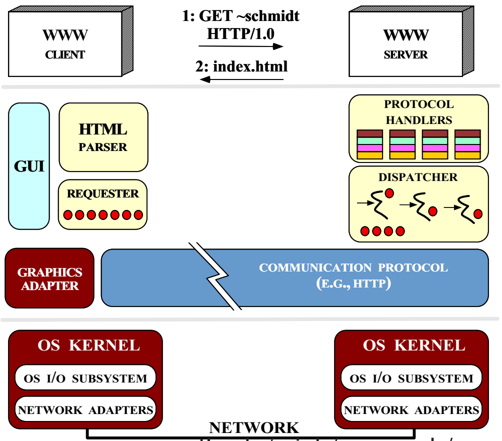
Related work →

- U. RI, Mitre
- QuO at BBN
- ARMADA at U. Mich.

TAO Statistics

- TAO order of magnitude
  - Core ORB > 200,000 LOC
  - IDL compiler > 130,000 LOC
  - CORBA Object Services > 150,000 LOC
  - Leverages ACE heavily
- Ported to UNIX, Win32, and RT/embedded platforms
  - *e.g.*, VxWorks, LynxOS, Chorus
- > 25 person-years of effort
- Currently used by many companies
  - *e.g.*, Boeing, BBN, Lockheed, Lucent, Motorola, Raytheon, SAIC, Siemens, etc.
- Supported commercially by OCI
  - www.ociweb.com

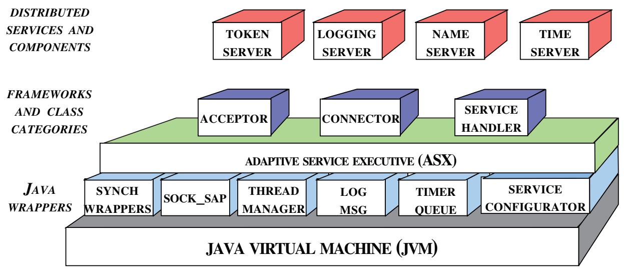
JAWS Adaptive Web Server



www.cs.wustl.edu/~jxh/research/

- **JAWS Overview**
  - A high-performance Web server
    - ∗ Flexible concurrency and dispatching mechanisms
  - Leverages the ACE framework
    - ∗ Ported to most OS platforms
  - Used commercially by Entera
    - ∗ www.entera.com

Java ACE



www.cs.wustl.edu/∼schmidt/JACE.html
www.cs.wustl.edu/∼schmidt/C++2java.html
www.cs.wustl.edu/∼schmidt/MedJava.ps.gz

Java ACE Overview

- A Java version of ACE
  - Used for medical imaging prototype

Key OO Design Concepts and Principles

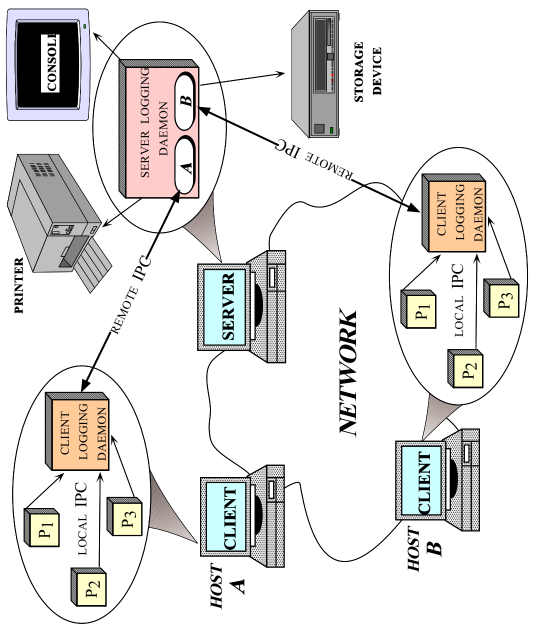
Key design concepts and design principles include:

- *Decomposition*
- *Abstraction*
- *Information Hiding*
- *Modularity*
- *Extensibility*
- *Virtual Machine Structuring*
- *Hierarchy*
- *Program Families and Subsets*

Main goal of these concepts and principles is to:

- Manage software system complexity
- Improve software quality factors
- Facilitate systematic reuse

Distributed Logging Service



Collects, formats, and outputs logging records forwarded from *client logging daemons* residing throughout a network or internetwork

Distributed Logging Service Programming API

The logging API is similar to `printf`, *e.g.*:

```
ACE_ERROR ((LM_ERROR, "(%t) fork failed"));
```

Generates on logging server host:

```
Oct 29 14:50:13 1992@tango.ics.uci.edu@2766@LM_ERROR@client
::(4) fork failed
```

and

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

generates on logging server host:

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

Conventional Logging Server Design

Typical algorithmic pseudo-code for distributed logging server:

```
void logging_server (void) {
  initialize acceptor 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!

Conventional Logging Server Implementation

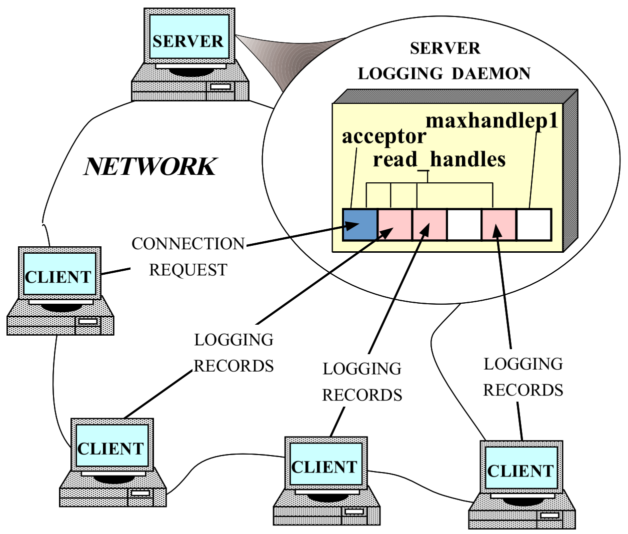
Note the excessive amount of detail required to program at the socket level...

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

typedef u_long COUNTER;
typedef int HANDLE;

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

// Acceptor-mode socket handle
static HANDLE acceptor;

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

// Set of currently active handles
static fd_set activity_handles;

// Scratch copy of activity_handles
static fd_set ready_handles;
```


Main Event Loop of Logging Server

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

  // Loop forever performing logging
  // server processing.

  for (;;) {
    // struct assignment.
    ready_handles = activity_handles;

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

    // First receive pending logging records.
    handle_data ();

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


The `select`-based Logging Server Implementation



Serialize server processing at `select` demuxing level

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 lr;

  // 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 *) &lr) + nread,
               len - nread, 0);
  // Decode and print record.
  decode_log_record (&lr);
  if (write (out_h, lr.buf, lr.size) == -1)
    return -1;
  else return 0;
}
```


Initialize Acceptor Socket

```
static void initialize_acceptor (u_short port)
{
  struct sockaddr_in saddr;

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

  // Set up the address info. to become 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 (acceptor,
        (struct sockaddr *) &saddr,
        sizeof saddr);

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

  // Initialize handle sets.
  FD_ZERO (&ready_handles);
  FD_ZERO (&activity_handles);
  FD_SET (acceptor, &activity_handles);
  maxhp1 = acceptor + 1;
}
```


Handle Connection Acceptance

```
static void handle_connections (void)
{
  if (FD_ISSET (acceptor, &ready_handles)) {
    static struct timeval poll_tv = {0, 0};
    HANDLE h;

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

    do {
      // Beware of subtle bug(s) here...
      h = accept (acceptor, 0, 0);
      FD_SET (h, &activity_handles);

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


Handle Data Processing

```
static void handle_data (void) {
  // acceptor + 1 is the lowest client handle

  for (HANDLE h = acceptor + 1; h < maxhp1; h++)
    if (FD_ISSET (h, &ready_handles)) {
      ssize_t n = handle_log_record (h, 1);

      // Guaranteed not to block in this case!
      if (n > 0)
        ++request_count;
        // Count the # of logging records
      else if (n == 0) {
        // Handle connection shutdown.
        FD_CLR (h, &activity_handles);
        close (h);
        if (h + 1 == maxhp1) {
          // Skip past unused handles
          while (!FD_ISSET (--h,
                            &activity_handles))
            continue;
          maxhp1 = h + 1;
        }
      }
    }
}
```

Conventional Client Logging Daemon Implementation

The `main` method receives logging records from client applications and forwards them on to the logging server

```
int main (int argc, char *argv[])
{
  HANDLE stream = initialize_stream_endpoint
                    (argc > 1
                     ? atoi (argv[1])
                     : PORT);
  Log_Record lr;

  // Loop forever performing client
  // logging daemon processing.

  for (;;) {
    // ... get logging records from client
    //     application processes ...

    size_t size = htonl (lr.size);
    send (stream, &size, sizeof size);
    encode_log_record (&lr);
    send (stream, ((char *) &lr), sizeof lr);
  }
}
```


Client Connection Establishment

```
static HANDLE initialize_stream_endpoint
  (const char *host, u_short port)
{
  struct sockaddr_in saddr;

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

  // Set up the address info. to become client.
  memset ((void *) &saddr, 0, sizeof saddr);
  saddr.sin_family = AF_INET;
  saddr.sin_port = htons (port);
  hostent *hp = gethostbyname (host);
  memcpy ((void *) &saddr,
          htonl (hp->h_addr),
          hp->h_length);

  // Associate address with endpoint
  connect (stream,
           (struct sockaddr *) &saddr,
           sizeof saddr);
  return stream;
}
```


Limitations with Algorithmic Decomposition

Algorithmic decomposition tightly couples application-specific *functionality* and the following configuration-related characteristics:

- **Application Structure**
  - The number of services per process
  - Time when services are configured into a process
- **Communication and Demultiplexing Mechanisms**
  - The underlying IPC mechanisms that communicate with other participating clients and servers
  - Event demultiplexing and event handler dispatching mechanisms
- **Concurrency and Synchronization Model**
  - 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 impedes reusability, extensibility, and portability...
- In contrast, OO focuses on *application-specific* behavior, *e.g.*,

```
int Logging_Handler::handle_input (void)
{
  ssize_t n = handle_log_record (peer ().get_handle (),
                                 STDOUT);
  if (n > 0)
    ++request_count; // Count the # of logging records

  return n <= 0 ? -1 : 0;
}
```

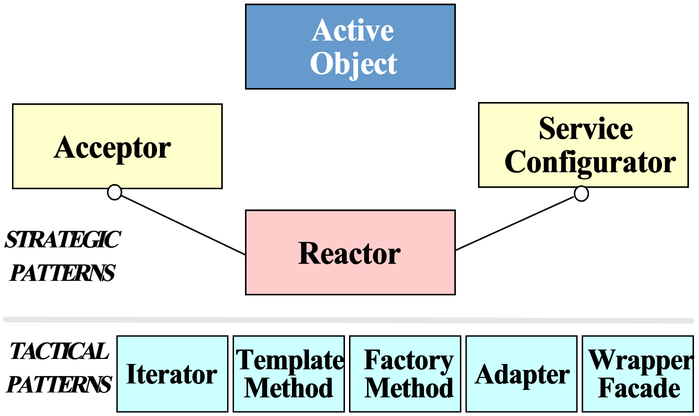
OO Contributions to Software

- *Patterns* facilitate the large-scale reuse of software architecture
  - Even when reuse of algorithms, detailed designs, and implementations is not feasible
- *Frameworks* achieve large-scale design and code reuse
  - In contrast, traditional techniques focus on the *functions* and *algorithms* that solve particular requirements
- Note that patterns and frameworks are not unique to OO!
  - But objects and classes are useful abstraction mechanisms

Why We Need Reusable Communication Middleware

- System call-level programming is wrong abstraction for application developers, *e.g.*,
  - *Too low-level* → error codes, endless reinvention
  - *Error-prone* → HANDLEs lack type-safety, thread cancellation woes
  - *Mechanisms do not scale* → Win32 TLS
  - *Steep learning curve* → Win32 Named Pipes
  - *Non-portable* → Win32 WinSock bugs
  - *Inefficient* → *i.e.*, tedious for humans
- GUI frameworks are inadequate for communication software, *e.g.*,
  - *Inefficient* → excessive use of virtual methods
  - *Lack of features* → minimal threading and synchronization mechanisms, no network services

Patterns in the Distributed Logger



- *Strategic* and *tactical* are relative to the *context* and *abstraction level*

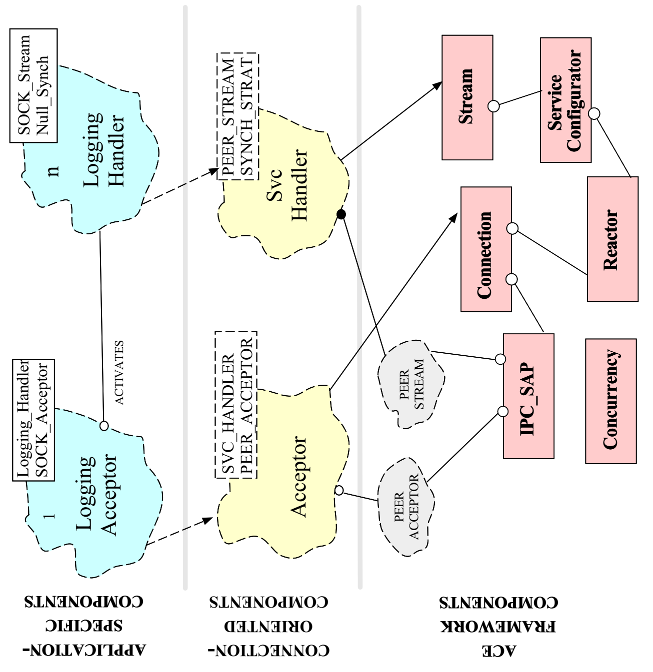
Summary of Pattern Intents

- *Wrapper Facade* → "Encapsulates the functions and data provided by existing non-OO APIs within more concise, robust, portable, maintainable, and cohesive OO class interfaces"
- *Reactor* → "Demultiplexes and dispatches requests that are delivered concurrently to an application by one or more clients"
- *Acceptor* → "Decouple the passive connection and initialization of a peer service in a distributed system from the processing performed once the peer service is connected and initialized"
- *Service Configurator* → "Decouples the implementation of services from the time when they are configured"
- *Active Object* → "Decouples method execution from method invocation to enhance concurrency and simplify synchronized access to an object that resides in its own thread of control"

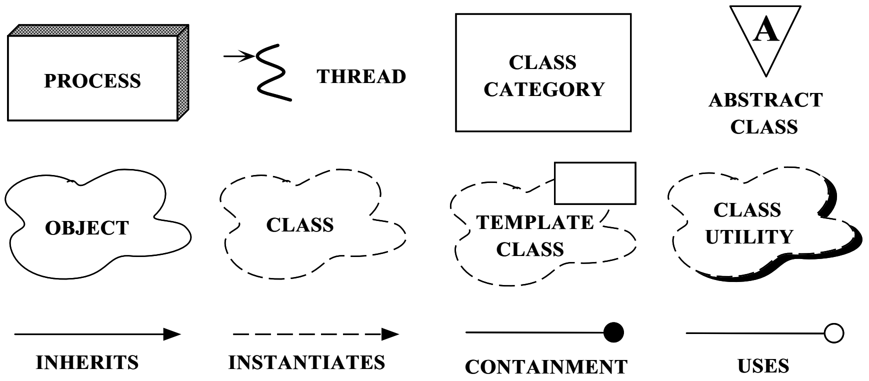
Components in the OO Logging Server

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

Class Diagram for OO Logging Server



Graphical Notation



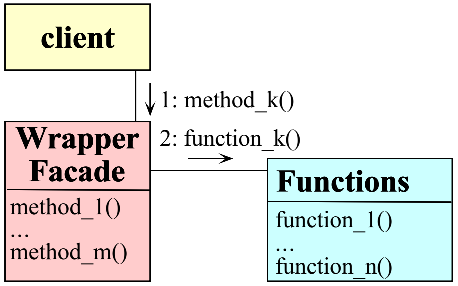
Addressing Robustness, Portability, and Maintainability Challenges

- Problem
  - Building distributed applications using low-level APIs is hard
- Forces
  - Low-level APIs are verbose, tedious, and error-prone to program
  - Low-level APIs are non-portable and non-maintainable
- Solution
  - Apply the *Wrapper Facade* pattern to encapsulate low-level functions and data structures

The Wrapper Facade Pattern

Intent

- *Encapsulates the functions and data provided by existing lower-level, non-OO APIs within more concise, robust, portable, maintainable, and cohesive higher-level OO class interfaces*

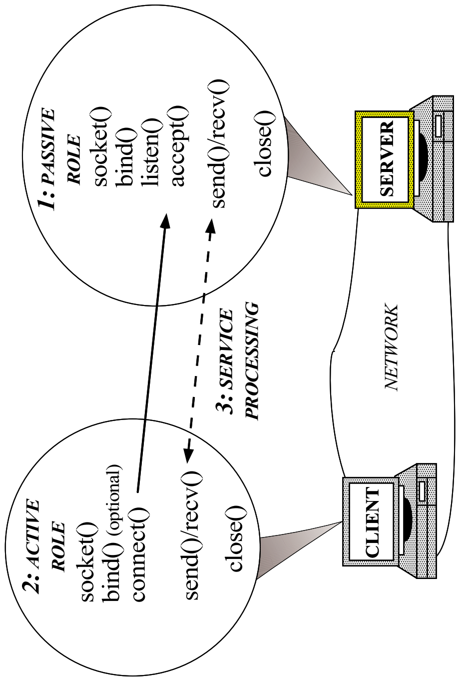


www.cs.wustl.edu/∼schmidt/wrapper-facade.ps.gz

Forces Resolved

- Avoid tedious, error-prone, and non-portable system APIs
- Create cohesive abstractions

Motivating the Wrapper Facade Pattern: the Socket API



Sockets are a common network programming API

Problem with Sockets: Lack of Type-safety

```
int buggy_echo_server (u_short port_num)
{ // Error checking omitted.
  sockaddr_in s_addr;
  int acceptor =
    socket (PF_UNIX, SOCK_DGRAM, 0);
  s_addr.sin_family = AF_INET;
  s_addr.sin_port = port_num;
  s_addr.sin_addr.s_addr = INADDR_ANY;
  bind (acceptor, (sockaddr *) &s_addr,
        sizeof s_addr);
  int handle = accept (acceptor, 0, 0);
  for (;;) {
    char buf[BUFSIZ];
    ssize_t n = read (acceptor, buf, sizeof buf);
    if (n <= 0) break;
    write (handle, buf, n);
  }
}
```

- I/O handles are not amenable to strong type checking at compile-time
- The adjacent code contains many subtle, common bugs

Problem with Sockets: Steep Learning Curve

Many socket/TLI API functions have complex semantics, *e.g.*:

- Multiple protocol families and address families
  - *e.g.*, TCP, UNIX domain, OSI, XNS, etc.
- Infrequently used features, *e.g.*:
  - Broadcasting/multicasting
  - Passing open file handles
  - Urgent data delivery and reception
  - Asynch I/O, non-blocking I/O, I/O-based and timer-based event multiplexing

Problem with Sockets: Portability

- Having multiple "standards," *i.e.*, sockets and TLI, makes portability difficult, *e.g.*,
  - May require conditional compilation
  - In addition, related functions are not included in POSIX standards
    - *e.g.*, `select`, `WaitForMultipleObjects`, and `poll`
- Portability between UNIX and Win32 Sockets is problematic, *e.g.*:
  - Header files
  - Error numbers
  - Handle vs. descriptor types
  - Shutdown semantics
  - I/O controls and socket options

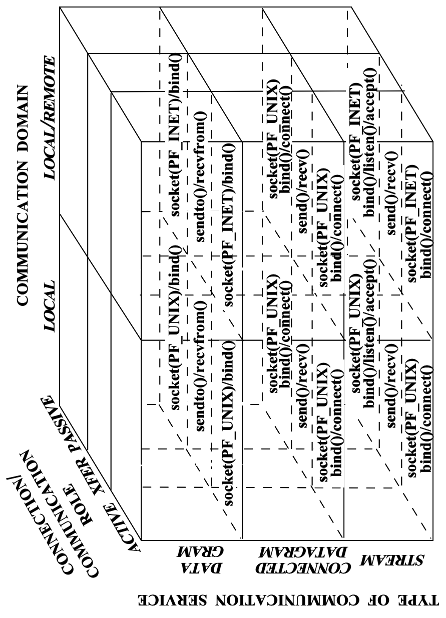
Problem with Sockets: Poorly Structured

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

Limitations

- Socket API is *linear* rather than *hierarchical*
- There is no consistency among names...
- Non-portable

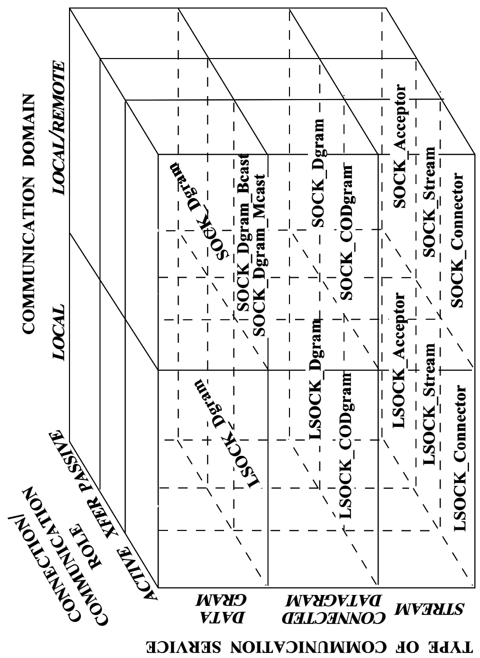
Socket Taxonomy



The Socket API can be classified along three dimensions

1. Connection role
2. Communication domain
3. Type of service

Solution: ACE Socket Wrapper Facades



The ACE C++ wrapper facades more explicitly model the key socket components using OO classes

ACE Socket Wrapper Factory Classes

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

  int connect
    (SOCK_Stream &new_sap,
     const INET_Addr &raddr,
     Time_Value *timeout,
     const INET_Addr &laddr);
  // ...
};
```

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

  SOCK_Acceptor (const INET_Addr &);
  int open (const INET_Addr &addr);
  int accept
    (SOCK_Stream &new_sap,
     INET_Addr *,
     Time_Value *);
  //...
};
```


ACE Socket Wrapper Facade Streaming and Addressing Classes

```
class SOCK_Stream : public SOCK {
public:
  // Trait.
  typedef INET_Addr PEER_ADDR;
  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 sendv_n (const iovec *iov,
                   int n);
  ssize_t recv_n (void *buf, int n);
  int close (void);
  // ...
};
```

```
class INET_Addr
  : public Addr
{
public:
  INET_Addr (u_short port,
             const char host[]);
  u_short get_port_number (void);
  UINT_32 get_ip_addr (void);
  // ...
};
```


Design Interlude: Motivating the Socket Wrapper Facade Structure

- 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.

An Echo Server Written using ACE C++ Socket Wrapper Facades

```
int echo_server (u_short port_num)
{
  // Local server address.
  INET_Addr my_addr (port_num);

  // Initialize the acceptor mode server.
  SOCK_Acceptor acceptor (my_addr);

  // Data transfer object.
  SOCK_Stream new_stream;

  // Accept a new connection.
  acceptor.accept (new_stream);

  for (;;) {
    char buf[BUFSIZ];
    // Error caught at compile time!
    ssize_t n =
      acceptor.recv (buf, sizeof buf);
    new_stream.send_n (buf, n);
  }
}
```

Using the Wrapper Facade Pattern for the Logging Server

Note we haven't improved the overall design (yet)

```
// ... Same as before ...

// Acceptor-mode socket handle.
static SOCK_Acceptor acceptor;

// Set of currently active handles
static Handle_Set activity_handles;

// Scratch copy of activity_handles
static Handle_Set ready_handles;

static void initialize_acceptor (u_short port)
{
  // Set up address info. to become server.
  INET_Addr saddr (port);

  // Create a local endpoint of communication.
  acceptor.open (saddr);

  // Set the <SOCK_Acceptor> into non-blocking mode.
  acceptor.enable (NONBLOCK);

  activity_handles.set_bit (acceptor.get_handle ());
}
```

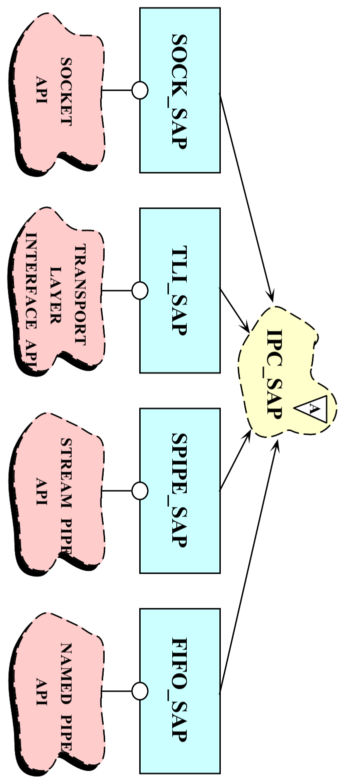

A Generic Version of the Echo Server

```
template <class ACCEPTOR>
int echo_server (u_short port)
{
  // Local server address (note traits).
  typename
  ACCEPTOR::PEER_ADDR my_addr (port);

  // Initialize the acceptor mode server.
  ACCEPTOR acceptor (my_addr);

  // Data transfer object (note traits).
  typename ACCEPTOR::PEER_STREAM stream;

  // Accept a new connection.
  acceptor.accept (stream);

  for (;;) {
    char buf[BUFSIZ];
    ssize_t n =
      stream.recv (buf, sizeof buf);
    stream.send_n (buf, n);
  }
}
```


Main Event Loop of Logging Server

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

  // Loop forever performing logging
  // server processing.

  for (;;) {
    // object assignment.
    ready_handles = activity_handles;

    // Wait for client I/O events.
    ACE_OS::select (int (maxhp1),
                    // calls operator fd_set *().
                    ready_handles);

    // First receive pending logging records.
    handle_data ();

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


Scope of the ACE IPC Wrapper Facades



www.cs.wustl.edu/~schmidt/IPC_SAP-92.ps.gz

OO Client Logging Daemon Implementation

```
int main (int argc, char *argv[])
{
  SOCK_Stream stream;
  SOCK_Connector con;    // Establish connection.
  con.connect (stream, INET_Addr (argc > 1
                       ? atoi (argv[1]) : PORT));
  Log_Record lr;

  // Loop forever performing client
  // logging daemon processing.
  for (;;) {
    // ... get logging records from client
    //     application processes ...
    UINT_32 size = lr.size;
    lr.size = htonl (lr.size);
    encode_log_record (&lr);
    iovec iov[2];
    iov[0].iov_len = sizeof (UINT_32);
    iov[0].iov_base = &lr.size;
    iov[1].iov_len = size;
    iov[1].iov_base = &lr;
    // Uses writev(2);
    stream.sendv_n (iov, 2);
  }
}
```


Handling Connections and Data Processing

```
static void handle_connections (void) {
  if (ready_handles.is_set (acceptor.get_handle ()))
    HANDLE h;
    SOCK_Stream str;

    // Handle all pending connection requests.
    while (acceptor.accept (str) != -1)
      activity_handles.set_bit (str.get_handle ());
  }
}

static void handle_data (void) {
  HANDLE h;
  Handle_Set_Iterator iter (ready_handles);

  while ((h = iter ()) != INVALID_HANDLE) {
    SOCK_Stream str (h);
    ssize_t n = handle_log_record (str, STDOUT);
    if (n > 0) // Count # of logging records.
      ++request_count;
    else if (n == 0) {
      // Handle connection shutdown.
      activity_handles.clr_bit (h);
      s.close ();
    }
}
```


Evaluating the Wrapper Facade Solution

Benefits

- More concise
- More robust
- More portable
- More maintainable
- More efficient

Liabilities

- Potentially more indirection
- Additional learning curve
- Still haven't solved the overall design problem
  - *i.e.*, the overall design is still based on step-wise refinement of functions

Receive and Process Logging Records

```
static ssize_t handle_log_record (SOCK_Stream s,
                                  HANDLE out_h) {
  UINT_32 len;
  Log_Record lr;

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

  ssize_t n = s.recv_n ((char *) &len, sizeof len);
  if (n <= 0) return n;

  len = ntohl (len); // Convert byte-ordering
  // Perform sanity check!
  if (len > sizeof (lr)) return -1;

  // The second recv then reads <len> bytes to
  // obtain the actual record.
  s.recv_n ((char *) &lr, sizeof lr);

  // Decode and print record.
  decode_log_record (&lr);
  if (ACE_OS::write (out_h, lr.buf, lr.size) == -1)
    return -1;
  else return 0;
}
```

ACE C++ Wrapper Facade Design Refactoring Principles

- Enforce typesafety at compile-time
- Allow controlled violations of typesafety
- Simplify for the common case
- Replace one-dimensional interfaces with hierarchical class categories
- Enhance portability with parameterized types
- Inline performance critical methods
- Define auxiliary classes to hide error-prone details

Enforce Typesafety at Compile-Time

Sockets cannot detect certain errors at compile-time, *e.g.*,

```
int acceptor = socket (PF_INET, SOCK_STREAM, 0);
// ...
bind (acceptor, ...); // Bind address.
listen (acceptor); // Make a acceptor-mode socket.
HANDLE n_sd = accept (acceptor, 0, 0);
// Error not detected until run-time.
read (acceptor, buf, sizeof buf);
```

ACE enforces type-safety at compile-time via *factories*, *e.g.*:

```
SOCK_Acceptor acceptor (port);

// Error: recv() not a method of SOCK_Acceptor.
acceptor.recv (buf, sizeof buf);
```


Allow Controlled Violations of Typesafety

Make it easy to use the C++ Socket wrapper facades correctly, hard to use it incorrectly, but not impossible to use it in ways the class designers did not anticipate

- *e.g.*, it may be necessary to retrieve the underlying socket handle:

```
SOCK_Acceptor acceptor;

// ...

Handle_Set ready_handles;

// ...

if (ready_handles.is_set (acceptor.get_handle ())
  ACE_OS::select (acceptor.get_handle () + 1, ready_handles);
```


Supply Default Parameters

```
SOCK_Connector (SOCK_Stream &new_stream,
                const Addr &remote_sap,
                ACE_Time_Value *timeout = 0,
                const Addr &local_sap = Addr::sap_any,
                int protocol_family = PF_INET,
                int protocol = 0);
```

The result is extremely concise for the common case:

```
SOCK_Stream stream;

// Compiler supplies default values.
SOCK_Connector con (stream, INET_Addr (port, host));
```

Define Parsimonious Interfaces

e.g., use LSOCK to pass socket handles:

```
LSOCK_Stream stream;
LSOCK_Acceptor acceptor ("/tmp/foo");

acceptor.accept (stream);
stream.send_handle (stream.get_handle ());
```

versus the less parsimonious BSD 4.3 socket code

```
LSOCK::send_handle (const HANDLE sd) const {
  u_char a[2]; iovec iov; msghdr send_msg;

  a[0] = 0xab, a[1] = 0xcd;
  iov.iov_base = (char *) a; iov.iov_len = sizeof a;
  send_msg.msg_iov = &iov; send_msg.msg_iovlen = 1;
  send_msg.msg_name = (char *) 0;
  send_msg.msg_namelen = 0;
  send_msg.msg_accrights = (char *) &sd;
  send_msg.msg_accrightslen = sizeof sd;
  return sendmsg (this->get_handle (), &send_msg, 0);
```

Note that SVR4 and BSD 4.4 APIs are different than BSD 4.3!

Combine Multiple Operations into One Operation

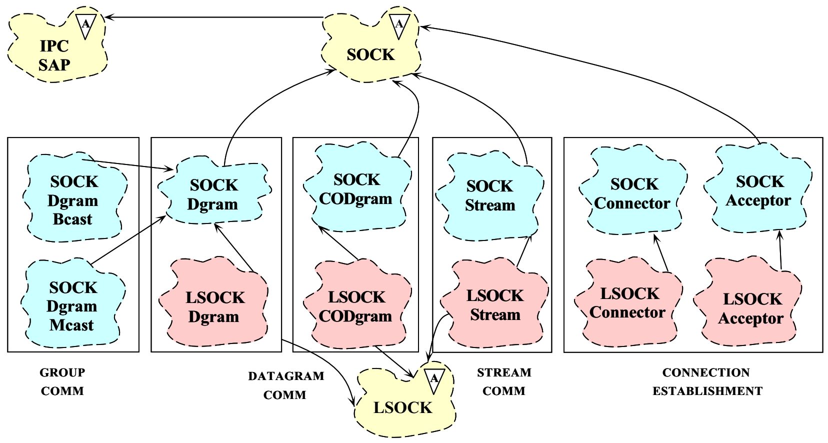
Creating a conventional acceptor-mode socket requires multiple calls:

```
int acceptor = socket (PF_INET, SOCK_STREAM, 0);
sockaddr_in addr;
memset (&addr, 0, sizeof addr);
addr.sin_family = AF_INET;
addr.sin_port = htons (port);
addr.sin_addr.s_addr = INADDR_ANY;
bind (acceptor, &addr, addr_len);
listen (acceptor);
// ...
```

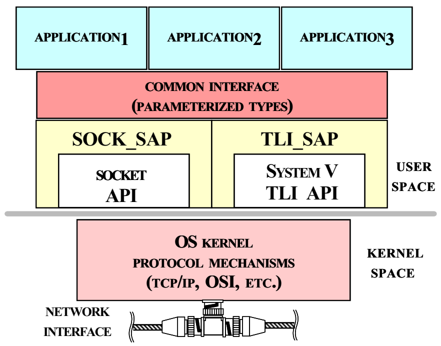
`SOCK_Acceptor` combines this into a single operation:

```
SOCK_Acceptor acceptor ((INET_Addr) port);
```


Create Hierarchical Class Categories



Enhance Portability with Parameterized Types




```
// Conditionally select IPC mechanism.
#if defined (USE_SOCKETS)
typedef SOCK_Acceptor PEER_ACCEPTOR;
#elif defined (USE_TLI)
typedef TLI_Acceptor PEER_ACCEPTOR;
#endif // USE_SOCKETS.

int main (void)
{
  // ...
  // Invoke with appropriate
  // network programming interface.
  echo_server<PEER_ACCEPTOR> (port);
}
```

Switching wholesale between sockets and TLI simply requires instantiating a different C++ wrapper

Inline Performance Critical Methods

Inlining is time and space efficient since key methods are very short:

```
class SOCK_Stream : public SOCK
{
public:
  ssize_t send (const void *buf, size_t n)
  {
    return ACE_OS::send (this->get_handle (), buf, n);
  }

  ssize_t recv (void *buf, size_t n)
  {
    return ACE_OS::recv (this->get_handle (), buf, n);
  }
};
```


Define Auxiliary Classes to Hide Error-Prone Details

Standard C socket addressing is awkward and error-prone

- *e.g.*, easy to neglect to zero-out a `sockaddr_in` or convert port numbers to network byte-order, etc.

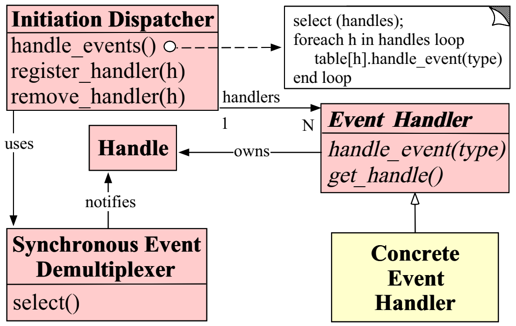
ACE C++ Socket wrapper facades define classes to handle details

```
class INET_Addr : public Addr { public:
  INET_Addr (u_short port, long ip_addr = 0) {
    memset (&this->inet_addr_, 0, sizeof this->inet_addr_);
    this->inet_addr_.sin_family = AF_INET;
    this->inet_addr_.sin_port = htons (port);
    memcpy (&this->inet_addr_.sin_addr, &ip_addr, sizeof ip_addr);
  }
  // ...
private:
  sockaddr_in inet_addr_;
};
```


Demultiplexing and Dispatching Events

- Problem
  - The logging server must process several different types of events simultaneously from different sources of events
- Forces
  - Multi-threading is not always available
  - Multi-threading is not always efficient
  - Multi-threading can be error-prone
  - Tightly coupling event demuxing with server-specific logic is inflexible
- Solution
  - Use the *Reactor* pattern to decouple event demuxing/dispatching from server-specific processing

The Reactor Pattern



www.cs.wustl.edu/∼schmidt/Reactor.ps.gz

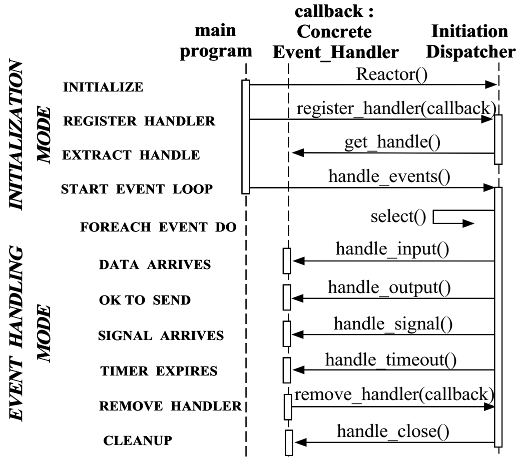
Intent

- *Demuxes & dispatches requests that are delievered concurrency to an application by one or more clients*

Forces Resolved

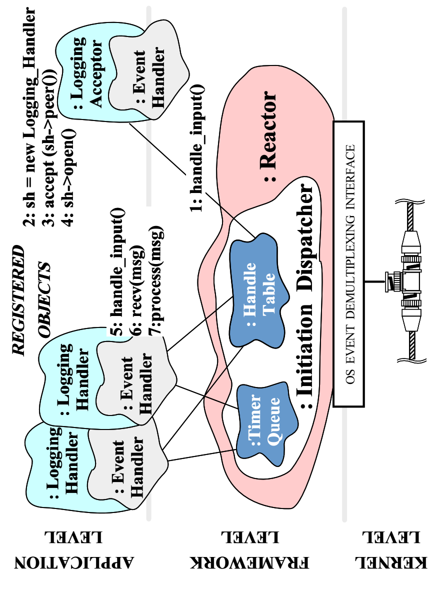
- Serially demux events *synchronously* & efficiently
- Extend applications without changing demuxing code

Collaboration in the Reactor Pattern



- Note *inversion of control*
- Also note how long-running event handlers can degrade quality of service since callbacks "steal" Reactor's thread of control...

Using the Reactor Pattern in the Logging Server



Benefits

- Straightforward to program
- Concurrency control is easy

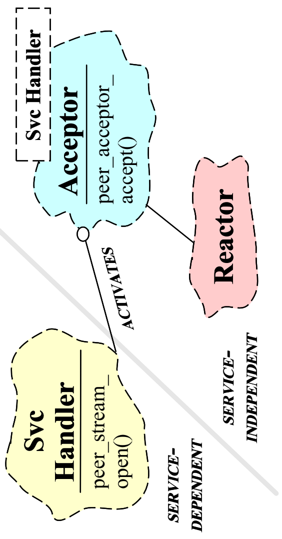
Liabilities

- Callbacks are "brittle"
- Can't leverage multi-processors

Addressing Acceptor Endpoint Connection and Initialization Challenges

- Problem
  - The *communication* protocol used between applications is often orthogonal to its *connection establishment* and *service handler initialization* protocols
- Forces
  - Low-level connection APIs are error-prone and non-portable
  - Separating *initialization* from *processing* increases software reuse
- Solution
  - Use the *Acceptor* pattern to decouple passive connection establishment and connection handler initialization from the subsequent logging protocol

The Acceptor Pattern



www.cs.wustl.edu/~schmidt/Acc-Con.ps.gz

Intent

- *Decouple the passive connection and initialization of a peer service in a distributed system from the processing performed once the peer service is connected and initialized*

Forces resolved

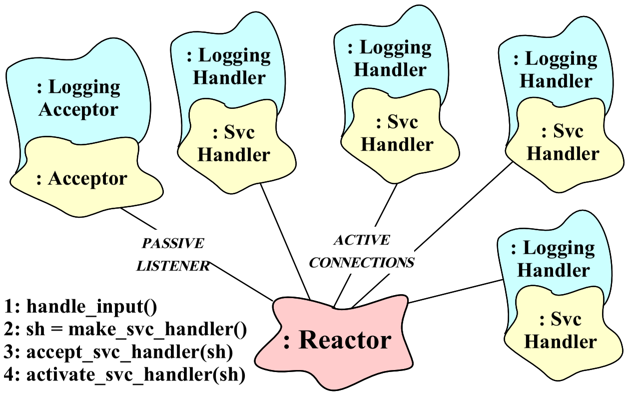
- Reuse passive connection setup and service initialization code
- Ensure that acceptor-mode handles aren't used to read/write data

`Acceptor` Class Public Interface

Note how service and IPC *aspects* are *strategized*!

```
template <class SVC_HANDLER, // Service aspect
          class PEER_ACCEPTOR> // IPC aspect
class Acceptor : public Service_Object {
  // Service_Object inherits from Event_Handler
public:
    // Initialization.
  virtual int open
    (const PEER_ACCEPTOR::PEER_ADDR &,
     Reactor * = Reactor::instance ());
    // Template Method or Strategy for creating,
    // connecting,  and activating SVC_HANDLER's.
  virtual int handle_input (HANDLE);

protected:
    // Factory method creates a service handler.
  virtual SVC_HANDLER *make_svc_handler (void);
    // Accept a new connection.
  virtual int accept_svc_handler (SVC_HANDLER *);
    // Activate a service handler.
  virtual int activate_svc_handler (SVC_HANDLER *);

private:
    // Acceptor IPC connection strategy.
  PEER_ACCEPTOR peer_acceptor_;
};
```


Using the Acceptor Pattern in the Logging Server

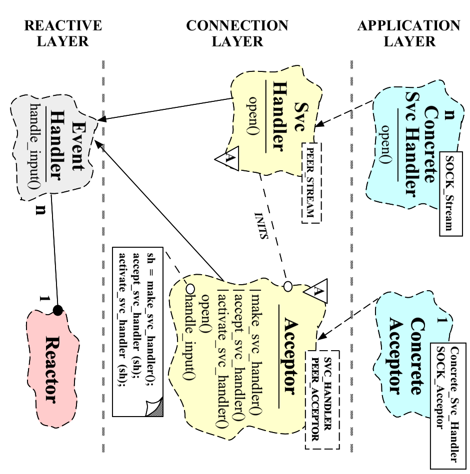


- The `Acceptor` is a *factory*
  - *i.e.*, it *creates*, *connects*, and *activates* a `Svc_Handler`
- There's typically one `Acceptor` per-service/per-port

`Acceptor` Class Implementation

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

// Template Method that creates, connects,
// and activates service handlers.

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);
}
```

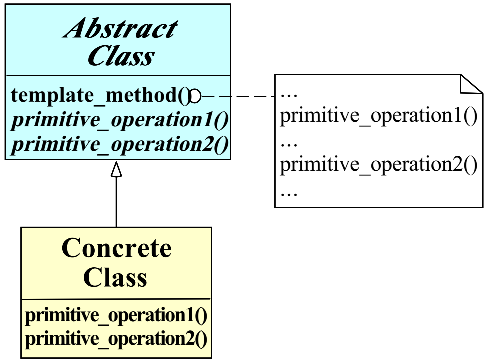

Structure of the Acceptor Pattern in ACE



Additional features of the ACE `Acceptor`

- Uses C++ parameterized types to *strategize* IPC and service *aspects*
- Uses Template Method pattern to strategize creation, connection establishment, and concurrency policies

The Template Method Pattern

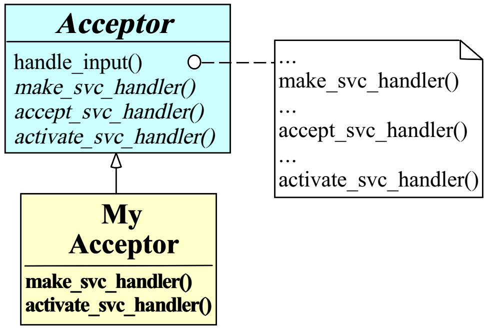


Intent

- *Define the skeleton of an algorithm in an operation, deferring some steps to subclasses*

Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* AW, '94

Using the Template Method Pattern in the ACE Acceptor Implementation



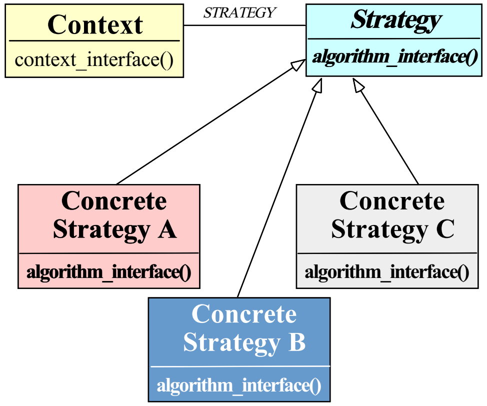
Benefits

- Straightforward to program via inheritance and dynamic binding

Liabilities

- Design is "brittle" and can cause "explosion" of subclasses due to "whitebox" design

The Strategy Pattern

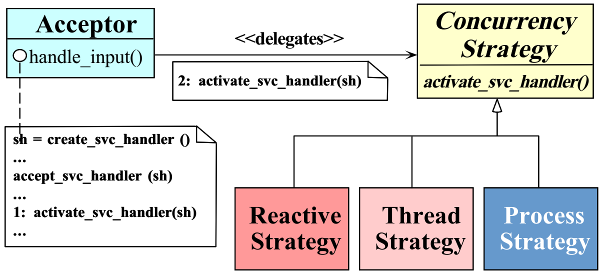


Intent

- *Define a family of algorithms, encapsulate each one, and make them interchangeable*

Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* AW, '94

Using the Strategy Pattern in the ACE Acceptor Implementation



Benefits

- More extensible due to "blackbox" design

Liabilities

- More complex and harder to develop initially

Svc_Handler Class Public Interface

Note how IPC and synchronization *aspects* are strategized

```
template <class PEER_STREAM, // IPC aspect
          class SYNCH_STRATEGY> // Synch aspect
class Svc_Handler : public Task<SYNCH_STRATEGY>
// Task is-a Service_Object,
// which is-an Event_Handler
{
public:
    // Constructor.
  Svc_Handler (Reactor * = Reactor::instance ());

    // Activate the handler (called by the Acceptor
.   // or Connector).
  virtual int open (void *);

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

  // ...
private:
  PEER_STREAM peer_; // IPC mechanism.

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


Acceptor Template Method Hook Implementations

Template method hooks can be overridden

```
// Factory method for creating a service handler.
template <class SH, class PA> SH *
Acceptor<SH, PA>::make_svc_handler (HANDLE) {
  return new SH; // Default behavior.
}

// Accept connections from clients.
template <class SH, class PA> int
Acceptor<SH, PA>::accept_svc_handler (SH *sh)
{
  peer_acceptor_.accept (sh->peer ());
}

// Activate the service handler.
template <class SH, class PA> int
Acceptor<SH, PA>::activate_svc_handler (SH *sh)
{
  if (sh->open () == -1)
    sh->close ();
}
```


Acceptor Initialization Implementation

Note how the PEER_ACCEPTOR's open method hides all the details associated with passively initializing communication endpoints

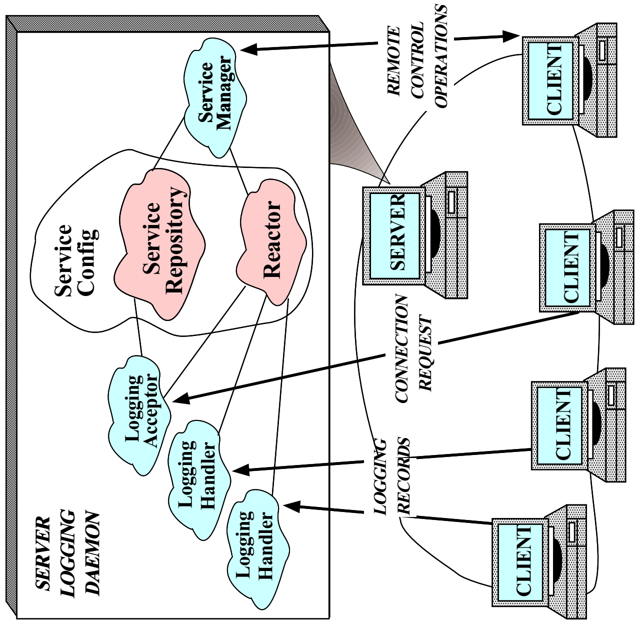
```
// Initialization.

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

  // Register with Reactor.
  reactor->register_handler
          (this, Event_Handler::ACCEPT_MASK);
}
```


Svc_Handler Implementation

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

template <class PS, class SYNCH_STRATEGY>
Svc_Handler<PS, SYNCH_STRATEGY>::Svc_Handler
  (Reactor *r): Service_Object (r) {}

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

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

- By default, a Svc_Handler object is registered with the Reactor
  - This makes the service "Reactive" so that no other synchronization mechanisms are necessary

Object Diagram for OO Logging Server



The `Logging_Handler` and `Logging_Acceptor` Classes

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

class Logging_Handler :
  public Svc_Handler<SOCK_Acceptor::PEER_STREAM,
                     // Trait!
                     NULL_SYNCH> {
public:
  // Recv and process remote logging records.
  virtual int handle_input (HANDLE);
};

// 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);
};
```


Design Interlude: Parameterizing IPC Mechanisms

- Q: *How can you switch between different IPC mechanisms?*
- A: By parameterizing IPC Mechanisms with C++ Templates, *e.g.*:

```
#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, // Trait!
              NULL_SYNCH>
    { /* ... /* };

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


`Logging_Handler` Input Method

Callback routine that receives logging records

```
int
Logging_Handler::handle_input (HANDLE)
{
  // Call existing function to recv
  // logging record and print to stdout.
  ssize_t n =
    handle_log_record (peer ().get_handle (),
                       STDOUT);
  if (n > 0)
    // Count the # of logging records
    ++request_count;
  return n <= 0 ? -1 : 0;
}
```

- Implementation of application-specific logging method
- This is the main code supplied by a developer!

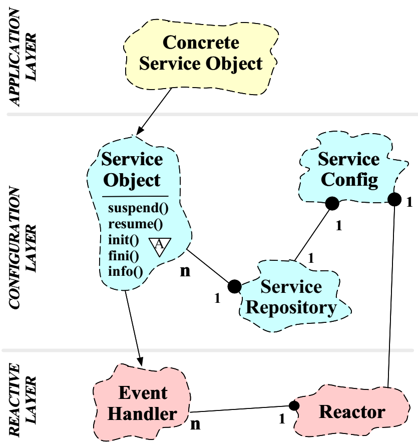
Logging_Acceptor Initialization and Termination

```
// Automatically called when a Logging_Acceptor
// object is linked dynamically.
Logging_Acceptor::init (int argc, char *argv[])
{
  Get_Opt get_opt (argc, argv, "p:", 0);
  INET_Addr addr (DEFAULT_PORT);

  for (int c; (c = get_opt ()) != -1; )
    switch (c) {
      case 'p':
        addr.set (atoi (getopt.optarg));
        break;
      default:
        break;
    }
  // Initialize endpoint and register with Reactor
  open (addr, Reactor::instance ());
}

// Automatically called when object is unlinked.
Logging_Acceptor::fini (void) { handle_close (); }
```


Putting the Pieces Together at Run-time

- Problem
  - Prematurely committing ourselves to a particular logging server configuration is inflexible and inefficient
- Forces
  - It is useful to build systems by "scripting" components
  - Certain design decisions can't be made efficiently until run-time
  - It is a bad idea to force users to "pay" for components they do not use
- Solution
  - Use the *Service Configurator* pattern to assemble the desired logging server components dynamically

The Service Configurator Pattern



Intent

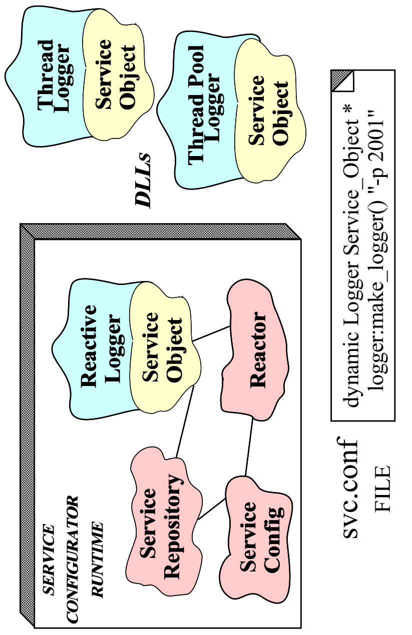
- *Decouples the implementation of services from the time when they are configured*

Forces Resolved

- Reduce resource utilization
- Support dynamic (re)configuration

www.cs.wustl.edu/~schmidt/Svc-Conf.ps.gz

Using the Service Configurator Pattern for the Logging Server



- The existing Logging Server service is single-threaded
- Other versions could be multi-threaded
- Note how we can script this via the `svc.conf` file

Dynamically Linking a Service

Dynamically linked factory function that allocates a new `Logging_Acceptor`

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

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

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

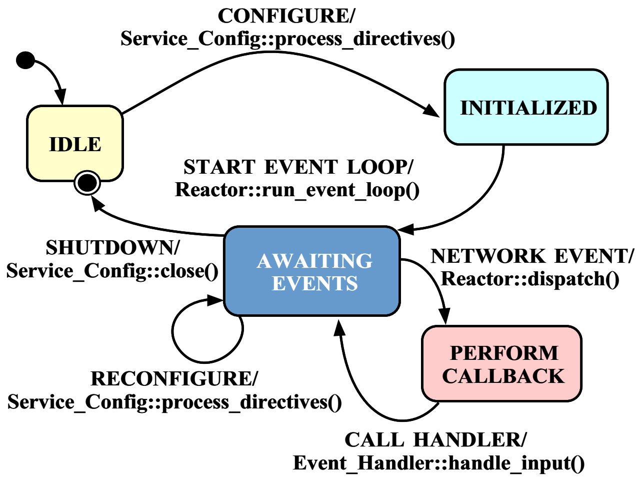
The logging service is configured via scripting in a `svc.conf` file:

```
% cat ./svc.conf
# Dynamically configure
# the logging service
dynamic Logger
Service_Object *
logger:make_Logger() "-p 2001"
# Note, .dll or .so suffix
# added to the logger
# automatically
```

Generic event-loop to dynamically configure service daemons

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

  // Run forever, performing
  // configured services
  Reactor::run_event_loop ();
  /* NOTREACHED */
}
```


State-chart for the Service Configurator Pattern



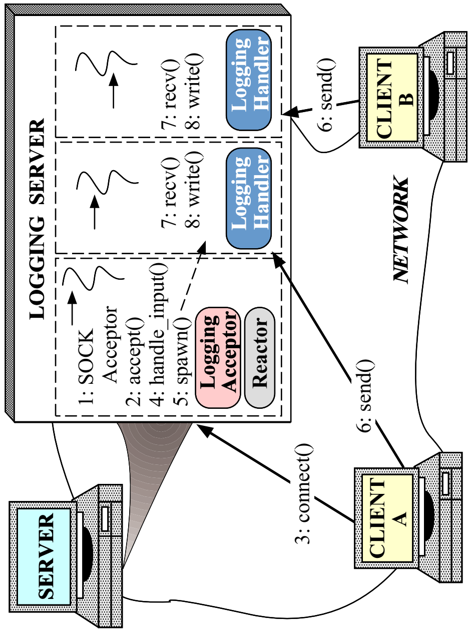
Advantages of OO Logging Server

- 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*:
  - *Modifying*, *recompiling*, and *relinking* existing code
  - *Terminating* and *restarting* executing daemons
- The remainder of the Logging Server slides examine a set of techniques for decoupling functionality from *concurrency* mechanisms, as well

Concurrent OO Logging Server

- The structure of the Logging Server can benefit from concurrent execution on a multi-processor platform
- This section examines ACE C++ classes and 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



Run's each client connection in a separate thread

Pseudo-code for Concurrent Server

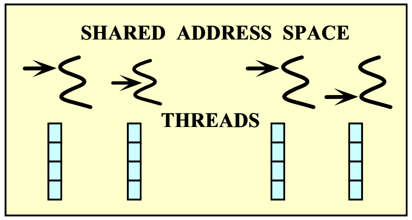
- Pseudo-code for multi-threaded `Logging_Handler` factory Logging Server

```
void handler_factory (void) {
  initialize acceptor endpoint
  foreach (pending connection event) {
           accept connection
           spawn a thread to handle connection and
           run logging_handler() entry point
  }
}
```

- Pseudo-code for `logging_handler` function

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


Concurrency Overview



PROCESS

- A thread is a sequence of instructions executed in one or more processes
  - *One process* → stand-alone systems
  - *More than one process* → distributed systems

Traditional OS processes contain a single thread of control

- This simplifies programming since a sequence of execution steps is protected from unwanted interference by other execution sequences...

Traditional Approaches to OS Concurrency

1. Device drivers and programs with signal handlers utilize a limited form of *concurrency*
   - *e.g.*, asynchronous I/O
   - Note that *concurrency* encompasses more than *multi-threading*...
2. Many existing programs utilize OS processes to provide "coarse-grained" concurrency
   - *e.g.*,
     - Client/server database applications
     - Standard network daemons like UNIX `inetd`
   - Multiple OS processes may share memory via memory mapping or shared memory and use semaphores to coordinate execution
   - The OS kernel scheduler dictates process behavior

Evaluating Traditional OS Process-based Concurrency

- Advantages
  - *Easy to keep processes from interfering*
    - A process combines *security*, *protection*, and *robustness*
- Disadvantages
  - *Complicated to program*, *e.g.*,
  - Signal handling may be tricky
  - Shared memory may be inconvenient
- *Inefficient*
  - The OS kernel is involved in synchronization and process management
  - Difficult to exert fine-grained control over scheduling and priorities

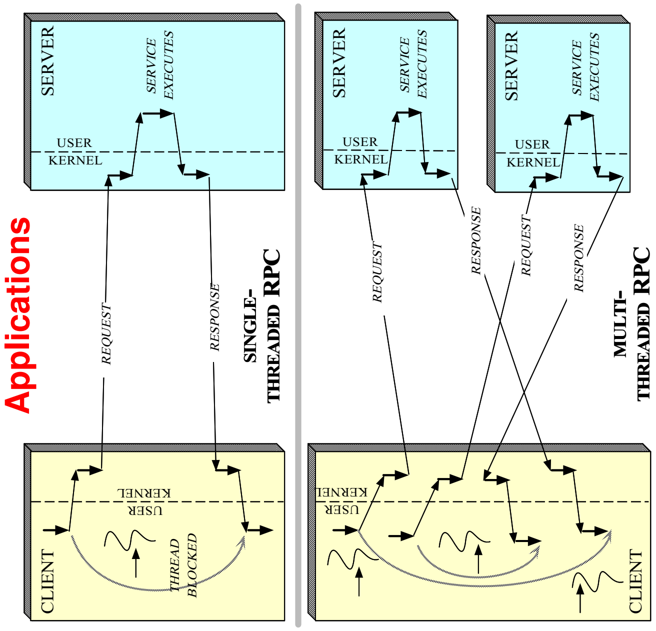
Modern OS Concurrency

- Modern OS platforms typically provide a standard set of APIs that handle
  - Process/thread creation and destruction
  - Various types of process/thread synchronization and mutual exclusion
  - Asynchronous facilities for interrupting long-running processes/threads to report errors and control program behavior
- Once the underlying concepts are mastered, it's relatively easy to learn different concurrency APIs
  - *e.g.*, traditional UNIX process operations, Solaris threads, POSIX pthreads, WIN32 threads, Java threads, etc.

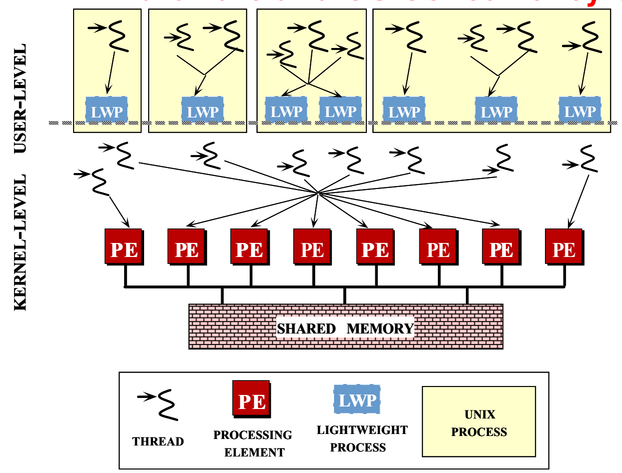
Lightweight Concurrency

- Modern operating systems provide lightweight mechanisms that manage and synchronize multiple threads *within* a process
  - Some systems also allow threads to synchronize *across* multiple processes
- Benefits of threads
  1. *Relatively simple and efficient to create, control, synchronize, and collaborate*
     - Threads share many process resources by default
  2. *Improve performance by overlapping computation and communication*
     - Threads may also consume less resources than processes
  3. *Improve program structure*
     - *e.g.*, compared with using asynchronous I/O

Example: Single-threaded vs. Multi-threaded Client/Server Applications



Hardware and OS Concurrency Support



Four typical abstractions

1. *Application threads*
2. *Lightweight processes*
3. *Kernel threads*
4. *Processing elements*

Application Threads

Most process resources are equally accessible to all threads in a process, *e.g.*,

- *Virtual memory*
- *User permissions and access control privileges*
- *Open files*
- *Signal handlers*

Each thread also contains unique information, *e.g.*,

- *Identifier*
- *Register set* (*e.g.*, PC and SP)
- *Run-time stack*
- *Signal mask*
- *Priority*
- *Thread-specific data* (*e.g.*, `errno`)

Note, there is no MMU protection for threads in a single process

Kernel-level vs. User-level Threads

- Application and system characteristics influence the choice of *user-level* vs. *kernel-level* threading
- A high degree of “virtual” application concurrency implies user-level threads (*i.e.*, unbound threads)
  - *e.g.*, desktop windowing system on a uni-processor
- A high degree of “real” application parallelism implies lightweight processes (LWPs) (*i.e.*, bound threads)
  - *e.g.*, video-on-demand server or matrix multiplication on a multi-processor

Overview of OS Synchronization Mechanisms

- Threads share resources in a process address space
- Therefore, they must use *synchronization mechanisms* to coordinate their access to shared data
- Traditional OS synchronization mechanisms are very low-level, tedious to program, error-prone, and non-portable
- ACE encapsulates these mechanisms with wrapper facades and higher-level patterns/components

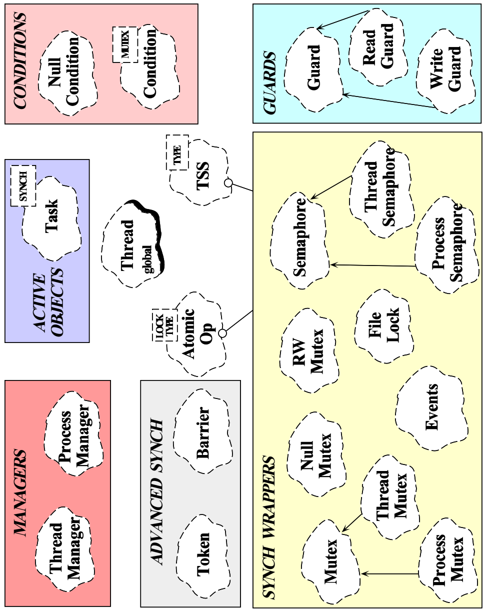
Common OS Synchronization Mechanisms

- Mutual exclusion (mutex) locks
  - Serialize thread access to a shared resource
- Counting semaphores
  - Synchronize thread execution
- Readers/writer (R/W) locks
  - Serialize resources that are searched more than changed
- Condition variables
  - Used to block threads until shared data changes state
- File locks
  - System-wide R/W locks accessed by processes

Additional ACE Synchronization Mechanism

- Events
  - *Gates* and *latches*
- Barriers
  - Allows threads to synchronize their completion
- Token
  - Provides FIFO scheduling order
- Task
  - Provides higher-level "active object" for concurrent applications
- Thread-specific storage
  - Low-overhead, contention-free storage

Concurrency Mechanisms in ACE

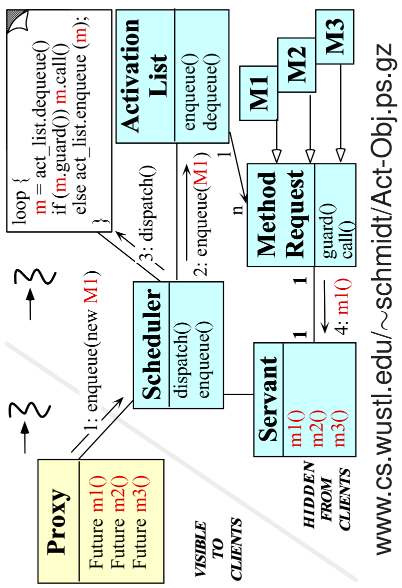


- All ACE Concurrency mechanisms are ported to all OS platforms
- www.cs.wustl.edu/~schmidt/Concurrency.ps.gz

Addressing Logger Server Concurrency Challenges

- Problem
  - Multi-threaded logging servers may be necessary when single-threaded reactive servers inefficient, non-scalable, or non-robust
- Forces
  - Multi-threading can be very hard to program
  - No single multi-threading model is always optimal
- Solution
  - Use the *Active Object* pattern to allow multiple concurrent logging server operations using an OO programming style

The Active Object Pattern



www.cs.wustl.edu/~schmidt/Act-Obj.ps.gz

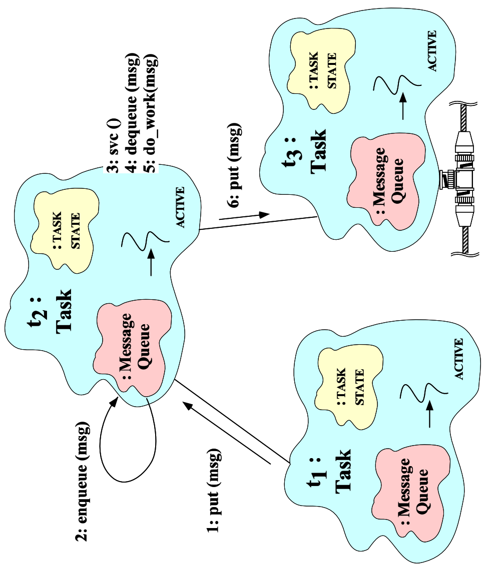
Intent

- *Decouples method execution from method invocation to enhance concurrency and simplify synchronized access to an object that resides in its own thread of control*

Forces Resolved

- Allow blocking operations
- Permit flexible concurrency strategies

ACE Support for Active Objects



ACE `Tasks` can be used to implement the complete Active Object pattern or lighterweight subsets

Application-specific Logging Code

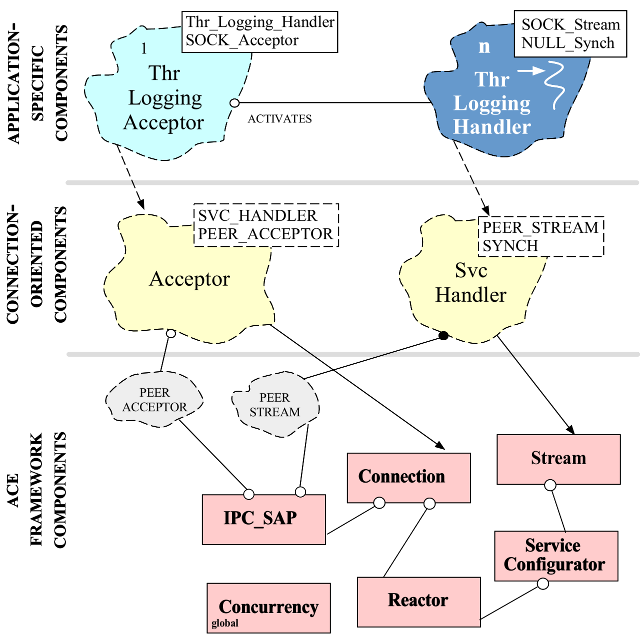
Process remote logging records by looping until the client terminates connection

```
int
Thr_Logging_Handler::svc (void)
{
  while (handle_input () != -1)
    // Call existing function
    //  to recv logging record
    // and print to stdout.
    continue;

  return 0;
}
```

- The OO implementation localizes the application-specific part of the logging service in a single point, while leveraging off reusable ACE components
- Compare with original, which borrow's the Reactor thread

```
int
Logging_Handler::handle_input (void)
{
  handle_log_record
    (peer ().get_handle (),
     STDOUT);
  // ...
}
```

Class Diagram for Concurrent OO Logging Server



`Thr_Logging_Acceptor` and `Thr_Logging_Handler Interfaces`

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

```
class Thr_Logging_Handler
  : public Logging_Handler
  // Inherits <handle_input>
{
public:
    // Override definition in <Svc_Handler>
    // class to spawn a new thread!  This method
    // is called by the <Acceptor>.
  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...
};
```


`Thr_Logging_Handler` Implementation

Override definition in the `Svc_Handler` class to spawn a new thread

```
int
Thr_Logging_Handler::open (void *)
{
  // Spawn a new thread to handle
  // logging records with the client.
  activate (THR_DETACHED);
}
```

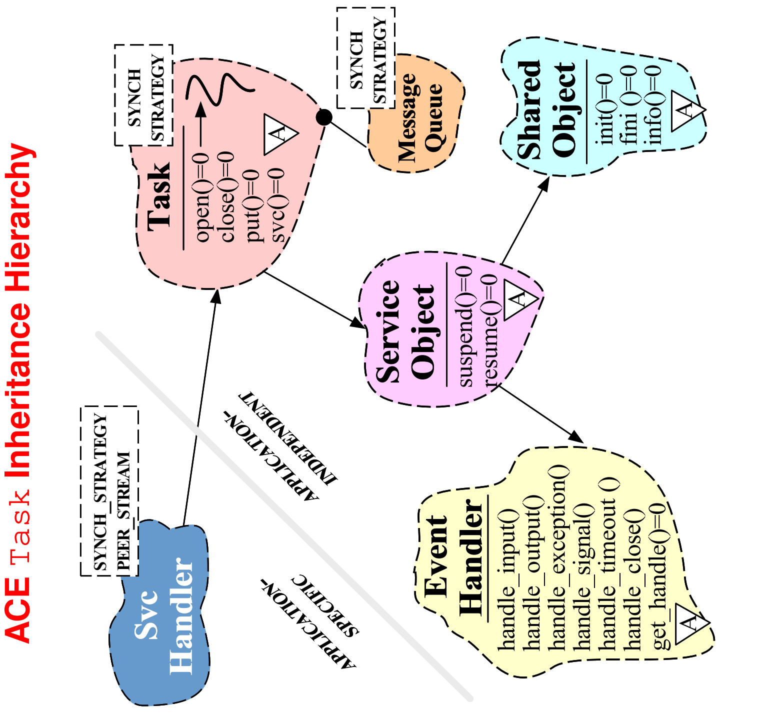
Process remote logging records by looping until client terminates connection

```
int
Thr_Logging_Handler::svc (void)
{
  while (handle_input () != -1)
    // Call existing function to recv
    // logging record and print to stdout.
    continue;
}
```


Design Interlude: 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` is a low-level mechanism to support "active objects"

ACE Task Inheritance Hierarchy



ACE `Task` Class Public Interface

```
template <class SYNCH_STRATEGY>
         // Synchronization aspect
class Task : public Service_Object {
public:
    // Initialization/termination hooks.
  virtual int open (void *args = 0) = 0;
  virtual int close (u_long = 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 task into active object.
  int activate (long flags, int threads = 1);
```

Features

1. `Tasks` can register with a `Reactor`
2. They can be dynamically linked
3. They can queue data
4. They can run as “active objects” in 1 or more threads

ACE `Task` Class Protected Interface

The following methods are mostly used within `put` and `svc`

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

    // 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_STRATEGY> *);
```


Design Interlude: Integrating Threading with C++

- 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_STRATEGY> int
Task<SYNCH_STRATEGY>::activate (long flags, int n_threads)
{
  if (thr_mgr () == NULL)
    thr_mgr (Thread_Manager::instance ());
  thr_mgr ()->spawn_n (n_threads,
                       &Task<SYNCH_STRATEGY>::svc_run,
                       (void *) this,
                       flags);
}
```

The `svc_run` Adapter Function

`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_STRATEGY> void *
Task<SYNCH_STRATEGY>::svc_run (Task<SYNCH_STRATEGY> *t)
{
  // Thread added to thr_mgr() automatically on entry.

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

  t->close (u_long (status));

  // Status becomes "return" value of thread...
  return status;

  // Thread removed from thr_mgr() automatically on return.
}
```


Design Interlude: Motivation for the ACE Thread Manager

- Q: "How can groups of collaborating threads be managed atomically?"
- A: Develop a "thread manager" class that:
  - Supports the notion of *thread groups*
  - Implements *barrier synchronization* on thread exits
  - Shields applications from incompatibilities between different OS thread libraries
    - *e.g.*, detached threads and thread joins

Dynamically Reconfiguring the Logging Server

The logging service is configured via scripting in a `svc.conf` file:

```
% cat ./svc.conf
# Dynamically reconfigure
# the logging service
remove Logger
dynamic Logger
Service_Object *
thr_logger:make_Logger()
           "-p 2002"
# .dll or .so suffix added to
# "thr_logger" automatically
```

Dynamically linked factory function that allocates a new threaded `Logging_Acceptor`

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

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

Logging service is reconfigured by changing the `svc.conf` file and sending SIGHUP signal to server

Caveats for the Concurrent Logging Server

- The concurrent Logging Server has several problems
  - Output in the `handle_log_record` function is not serialized
  - 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);
  if (ACE_OS::write (out_h, lr.buf, lr.size) == -1)
    return -1;
  mutex_unlock (&lock);
  // ...
}
```

- However, adding these `mutex` calls explicitly causes problems...

Problem: Explicit `mutex_*` Calls

- *Inelegant* → "Impedance mismatch" with C/C++
- *Obtrusive*
  - Must find and lock all uses of `write`
  - Can yield inheritance anomaly
- *Error-prone*
  - C++ exception handling and multiple method exit points
  - Thread mutexes won't work for separate processes
  - Global mutexes may not be initialized correctly
- *Non-portable* → Hard-coded to Solaris 2.x
- *Inefficient* → *e.g.*, expensive for certain platforms/designs

Solution: Synchronization Wrapper Facades

```
class Thread_Mutex
{
public:
  Thread_Mutex (void) {
    mutex_init (&lock_, USYNCH_THREAD, 0);
  }
  ~Thread_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:
  // SunOS 5.x serialization mechanism.
  mutex_t lock_;
  void operator= (const Thread_Mutex &);
  Thread_Mutex (const Thread_Mutex &);
};
```

Note how we prevent improper copying and assignment by using C++ access control specifiers

Porting `Thread_Mutex` to Windows NT

```
class Thread_Mutex
{
public:
  Thread_Mutex (void) {
    lock_ = CreateMutex (0, FALSE, 0);
  }
  ~Thread_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 Facade

- Using C++ wrapper facades improves *portability* and *elegance*

```
// at file scope.
Thread_Mutex lock; // Implicitly unlocked.

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

  lock.acquire ();
  if (ACE_OS::write (out_h, lr.buf, lr.size) == -1)
    return -1;
  lock.release ();
  // ...
```

- However, this doesn't really solve the *tedium* or *error-proneness* problems
  – www.cs.wustl.edu/~schmidt/ObjMan.ps.gz

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 (); }
  // ... other methods omitted ...

private:
  LOCK &lock_;
}
```

- `Guard` uses the *Scoped Locking* idiom whereby a *constructor acquires a resource* and the *destructor releases the resource*

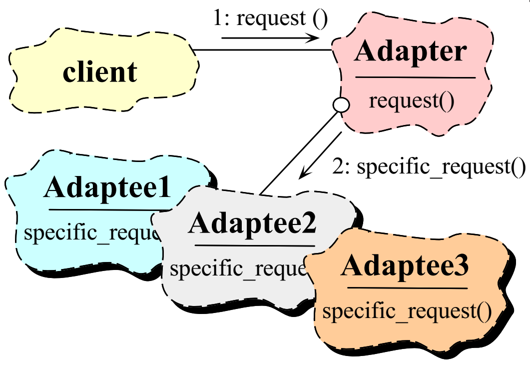
Thread-safe `handle_log_record`

```
template <class LOCK = Thread_Mutex> ssize_t
handle_log_record (HANDLE in, HANDLE out) {
  // beware static initialization...
  static LOCK lock;
  UINT_32 len;
  Log_Record lr;

  // The first recv reads the length (stored as a
  // fixed-size integer) of adjacent logging record.
  ssize_t n = s.recv_n ((char *) &len, sizeof len);
  if (n <= 0) return n;

  len = ntohl (len); // Convert byte-ordering
  // Perform sanity check!
  if (len > sizeof (lr)) return -1;

  // The second recv then reads <len> bytes to
  // obtain the actual record.
  s.recv_n ((char *) &lr, sizeof lr);

  // Decode and print record.
  decode_log_record (&lr);
  // Automatically acquire mutex lock.
  Guard<LOCK> guard (lock);
  if (ACE_OS::write (out, lr.buf, lr.size) == -1)
    return -1; // Automatically release mutex lock.
  return 0;
}
```


Design Interlude: Motivating the `Guard` Design

- Q: *Why is Guard parameterized by the type of LOCK*?
- A: since many different flavors of locking can benefit from the Scoped Locking protocol
  – *e.g.*, non-recursive vs. recursive mutexes, intra-process vs. inter-process mutexes, readers/writer mutexes, POSIX and System V semaphores, file locks, and the null mutex
- Q: *Why are templates used, as opposed to inheritance/polymorphism?*
- A: since they are more efficient and can reside in shared memory
- All ACE synchronization wrapper facades use the Adapter pattern to provide identical interfaces to facilitate parameterization

The Adapter Pattern



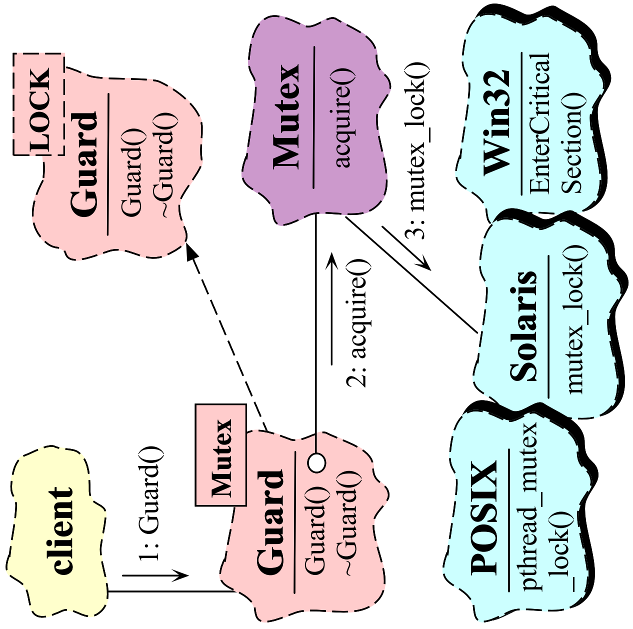
Intent

- *Convert the interface of a class into another interface client expects*

Force resolved:

- Provide an interface that captures similarities between different OS mechanisms, *e.g.*, locking or IPC

Using the Adapter Pattern for Locking



Remaining Caveats

```
int Logging_Handler::handle_input (void)
{
  ssize_t n = handle_log_record
    (peer ().get_handle (), STDOUT);
  if (n > 0)
    // Count # of logging records.
    ++request_count;
    // Danger, race condition!!!

  return n <= 0 ? -1 : 0;
}
```

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

www.cs.wustl.edu/~schmidt/Concurrency.ps.gz

Transparently Parameterizing Synchronization Using C++

Use the *Strategized Locking* pattern, C++ templates, and operator overloading to define "atomic operators"

```
template <class LOCK = Thread_Mutex,
          class TYPE = u_long>
class Atomic_Op {
public:
  Atomic_Op (TYPE c = 0) { count_ = c; }
  TYPE operator++ (void) {
    Guard<LOCK> guard (lock_); return ++count_;
  }
  operator TYPE () {
    Guard<LOCK> guard (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 (), STDOUT);
```

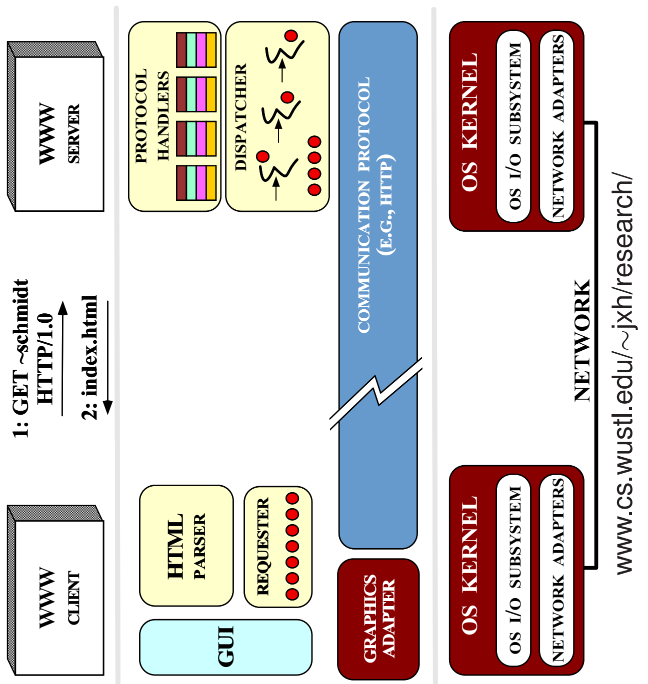
- The original non-threaded version may be supported efficiently as follows:

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


Concurrent Web Client/Server Example

- The following example illustrates a concurrent OO architecture for a high-performance Web client/server
- Key functional and non-functional system requirements are:
  - Robust implementation of HTTP 1.0 protocol
    - *i.e.*, resilient to incorrect or malicious Web clients/servers
  - Extensible for use with other protocols
    - *e.g.*, DICOM, HTTP 1.1, CORBA Simple Flow Protocol (SFP)
  - Leverage multi-processor hardware and OS software
    - *e.g.*, Support various concurrency patterns

General Web Client/Server Interactions



Pseudo-code for Concurrent Web Server

- Pseudo-code for master server

```
void master_server (void)
{
  initialize queue and acceptor at port 80
  spawn pool of worker threads
  foreach (pending work request from clients) {
    receive and queue request on queue
  }
  exit process
}
```

- Pseudo-code for thread pool workers

```
void worker (void)
{
  foreach (work request on queue)
    dequeue and process request
  exit thread
}
```

- As usual, make sure to avoid the "grand mistake"

Design Interlude: Motivating a Request Queue

- Q: *Why use a request queue to store messages, rather than directly reading from I/O handles?*
- A:
  - Promotes more efficient use of multiple CPUs via load balancing
  - Enables transparent interpositioning and prioritization
  - Makes it easier to shut down the server correctly and portably
  - Improves robustness to "denial of service" attacks
- *Drawbacks*
  - Using a message queue may lead to greater *context switching* and *synchronization* overhead...
  - Single point for bottlenecks

Thread Entry Point

```
typedef u_long COUNTER;
// Track the number of requests
COUNTER request_count; // At file scope.

// Entry point into the Web HTTP 1.0 protocol,
// which runs in each thread in the thread pool.
void *worker (Message_Queue *msg_queue)
{
  Message_Block *mb; // Message buffer.

  while (msg_queue->dequeue_head (mb)) > 0) {
    // Keep track of number of requests.
    ++request_count;

    // Print diagnostic
    cout << "got new request" << OS::thr_self ()
         << endl;

    // Identify and perform Web Server
    // request processing here...
  }
  return 0;
}
```


Master Server Driver Function

```
// Thread function prototype.
typedef void *(*THR_FUNC)(void *);

int main (int argc, char *argv[]) {
  parse_args (argc, argv);
  Message_Queue msg_queue; // Queue client requests.

  // Spawn off NUM_THREADS to run in parallel.
  for (int i = 0; i < NUM_THREADS; i++)
    thr_create (0, 0,
                THR_FUNC (&worker),
                  (void *) &msg_queue,
                    THR_BOUND, 0);

  // Initialize network device and
  // recv HTTP work requests.
  thr_create (0, 0,
              THR_FUNC (&recv_requests),
                (void *) &msg_queue,
                  THR_BOUND, 0);

  // Wait for all threads to exit (BEWARE)!
  while (thr_join (0, &t_id, (void **) 0) == 0)
    continue; // ...
}
```


Pseudo-code for `recv_requests`

```
void recv_requests (Message_Queue *msg_queue)
{
  initialize socket acceptor at port 80

  foreach (incoming request})
  {
    use select to wait for new
      connections or data
    if (connection)
      establish connections using accept()
    else if (data) {
      use sockets calls to
        read() HTTP requests into msg
        msg_queue.enqueue_tail (msg);
    }
  }
}
```

This is the "supplier" thread

Limitations with the Web Server

- The algorithmic decomposition tightly couples application-specific *functionality* with various configuration-related characteristics, *e.g.*,
- The HTTP 1.0 protocol
- The number of services per process
- The time when services are configured into a process
- The solution is not portable since it hard-codes
  - SunOS 5.x threading
  - sockets and `select`
- There are *race conditions* in the code

Overcoming Limitations via OO

- The algorithmic decomposition illustrated above specifies too many low-level details
  - Furthermore, the excessive coupling complicates reusability, extensibility, and portability...
- In contrast, OO focuses on decoupling *application-specific* behavior from reusable *application-independent* mechanisms
- The OO approach described below uses reusable *framework* components and commonly recurring *patterns*

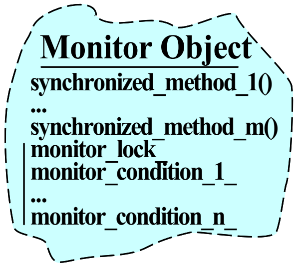
Eliminating Race Conditions

- Problem
  - A naive implementation of `Message_Queue` will lead to race conditions
    - *e.g.*, when messages in different threads are enqueued and dequeued concurrently
- Forces
  - Producer/consumer concurrency is common, but requires careful attention to avoid overhead, deadlock, and proper control
- Solution
  - Utilize the *Monitor Object* pattern and *condition variables*

The Monitor Object Pattern

Intent

- *Synchronizes method execution to ensure only one method runs within an object at a time. It also allows an object's methods to cooperatively schedule their execution sequences.*



~schmidt/monitor.ps.gz

Forces Resolved

- Synchronization corresponds to methods
- Objects, not clients, are responsible for synchronization
- Cooperative method scheduling

Overview of Condition Variables

- Condition variables (CVs) are used to “sleep/wait” until a particular condition involving shared data is signaled
  - CVs can wait on arbitrarily complex C++ expressions
  - Sleeping is often more efficient than busy waiting...
- This allows more complex scheduling decisions, compared with a mutex
  - *i.e.*, a mutex makes *other* threads wait, whereas a condition variable allows a thread to make *itself* wait for a particular condition involving shared data

Condition Variable Usage Patterns

```
// Initially unlocked.
static Thread_Mutex lock;
static Condition_Thread_Mutex
       cond (lock);

// synchronized
void acquire_resources (void) {
  // Automatically acquire lock.
  Guard<Thread_Mutex> guard (lock);

  // Check condition in loop
  while (cond. expression false)
    // Sleep.
    cond.wait ();

  // Atomically modify shared
  // information.

  // Destructor releases lock.
}
```

Note how the use of the Guard idiom simplifies the solution since we can’t forget to release the lock!

```
// synchronized
void release_resources (void) {
  // Automatically acquire lock.
  Guard<Thread_Mutex> guard (lock);

  // Atomically modify shared
  // information...

  cond.signal ();
  // Could use cond.broadcast()

  // guard automatically
  // releases lock.
}
```

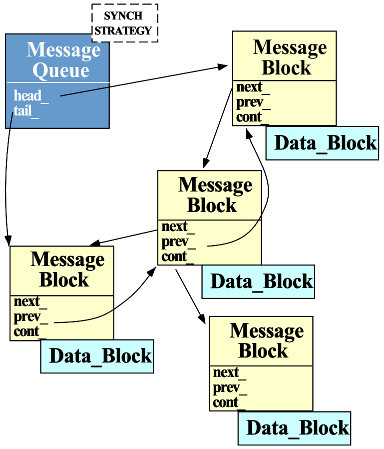

ACE Condition Variable Interface

```
class Condition_Thread_Mutex
public:
    // Initialize the CV.
  Condition_Thread_Mutex
    (const Thread_Mutex &);
    // Implicitly destroy the CV.
  ~Condition_Thread_Mutex (void);
    // Block on condition, or until
    // time passes.  If time == 0 block.
  int wait (Time_Value *time = 0) const;
    // Signal one waiting thread.
  int signal (void) const;
    // Signal *all* waiting threads.
  int broadcast (void) const;
private:
  cond_t cond_; // Solaris CV.
  const Thread_Mutex &mutex_;
};
```

In ACE, the `Condition_Thread_Mutex` class is a wrapper for the native OS condition variable abstraction

- *e.g.*, `cond_t` on SunOS 5.x, `pthread_cond_t` for POSIX, and a custom implementation on Win32

Overview of `Message_Queue` and `Message_Block`

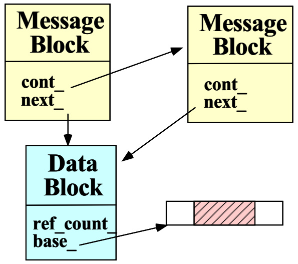


- A `Message_Queue` is a doubly-linked list of `Message_Blocks`
  - Efficiently handles arbitrarily-large message payloads
- A `Message_Block` is a *Composite*
  - Similar to BSD `mbufs` or SVR4 STREAMS `m_blks`
- Design parameterizes synchronization and allocation aspects

The Message_Block Class

```
class Message_Block {
public:
  Message_Block
   (size_t size,
    Message_Type type = MB_DATA,
    Message_Block *cont = 0,
    char *data = 0,
    u_long priority = 0,
    Allocator *alloc = 0);
  // ...
private:
  Data_Block *data_block_;
    // Pointer to payload.
  Message_Block *next_;
    // Pointer to next message.
  Message_Block *prev_;
    // Pointer to previous message.
  Message_Block *cont_;
    // Pointer to next ``fragment.''
    // ...
};
```

- Message_Block and Data_Block hide messaging implementations from clients
- *e.g.*, reference counting can be added transparently



Design Interlude: Motivating the ACE Allocator

- Q: *What is the Allocator object in the Message_Block constructor?*
- A: It provides an extensible mechanism to control how memory is allocated and deallocated
  - This makes it possible to switch memory management policies *without* modifying Message_Block
  - By default, the policy is to use new and delete, but it's easy to use other schemes, *e.g.*,
    - Heap memory
    - Shared memory
    - Persistent memory
    - Thread-specific memory
  - A similar technique is also used in the C++ Standard Template Library

Applying the External Polymorphism Pattern to ACE Memory Allocators

Intent

- *Allow classes that are not related by inheritance and/or have no virtual methods to be treated polymorphically*

www.cs.wustl.edu/~schmidt/C++-EP.ps.gz

```
class Allocator {
  // ...
  virtual void *malloc (size_t nbytes) = 0;
  virtual void free (void *ptr) = 0;
};

template <class ALLOCATOR>
class Allocator_Adapter : public Allocator {
  // ...
  virtual void *malloc (size_t nbytes)
  { return allocator_.malloc (nbytes); }
  ALLOCATOR allocator_;
};

Allocator_Adapter<Shared_Alloc> sh_malloc;
Allocator_Adapter<New_Alloc> new_malloc;
Allocator_Adapter<Persist_Alloc> p_malloc;
Allocator_Adapter<TSS_Alloc> tss_malloc;
```


Message_Queue Class Public Interface

Note how the synchronization aspect can be strategized!

```
template <class SYNCH_STRATEGY = MT_SYNCH>
         // Synchronization aspect
class Message_Queue
{
public:
    // Default high and low water marks.
  enum {
    DEFAULT_LWM = 0,
    DEFAULT_HWM = 4096
  };

    // Initialize a Message_Queue.
  Message_Queue (size_t hwm = DEFAULT_HWM,
                 size_t lwm = DEFAULT_LWM);

    // Check if full or empty (hold locks)
  int is_empty (void) const;
  int is_full (void) const;

    // Enqueue and dequeue Message_Block *'s.
  int enqueue_prio (Message_Block *, Time_Value *);
  int enqueue_tail (Message_Block *, Time_Value *);
  int dequeue_head (Message_Block *&, Time_Value *);
```

Design Interlude: Parameterizing Synchronization Strategies

- Q: *What is* `MT_SYNCH` *and how does it work?*
- A: `MT_SYNCH` provides a thread-safe synchronization strategy for a `Svc_Handler`
  - *e.g.*, it ensures that a `Svc_Handler`'s `Message_Queue` is thread-safe
  - Any `Task` that accesses shared state can use the `MT_SYNCH` *traits*

Note the use of *traits*:

```
struct MT_SYNCH {
  typedef Thread_Mutex MUTEX;
  typedef
    Condition_Thread_Mutex
    COND;
};

struct NULL_SYNCH {
  typedef Null_Mutex MUTEX;
  typedef
    Null_Condition_Thread_Mutex
    COND;
};
```


Message_Queue Class Private Interface

```
private:
    // Check boundary conditions & don't hold locks.
  int is_empty_i (void) const;
  int is_full_i (void) const;

    // Routines that actually do the enqueueing and
    // dequeueing & don't hold locks.
  int enqueue_prio_i (Message_Block *, Time_Value *);
  int enqueue_tail_i (Message_Block *new_item,
                      Time_Value *);
  int dequeue_head_i (Message_Block *&first_item);

    // ...

    // Parameterized types for synchronization
    // primitives that control concurrent access.
    // Note use of C++ traits
  typename SYNCH_STRATEGY::MUTEX     lock_;
  typename SYNCH_STRATEGY::COND not_empty_cond_;
  typename SYNCH_STRATEGY::COND not_full_cond_;

  size_t high_water_mark_;
  size_t low_water_mark_;
  size_t cur_bytes_;
  size_t cur_count_;
};
```


Design Interlude: Tips for Intra-class Locking

- Q: *How should locking be performed in an OO class?*
- A: Apply the *Thread-Safe Interface* pattern:
  - "Interface functions should lock and do no work – implementation functions should do the work and not lock "
    - ∗ This pattern helps to avoid intra-class method deadlock
  - This is actually a variant on a common OO pattern that "public functions should check, private functions should trust"
    - ∗ Naturally, there are exceptions to this rule...
  - This pattern avoids the following surprises
    - ∗ Unnecessary overhead from recursive mutexes
    - ∗ Deadlock if recursive mutexes aren't used
- www.cs.wustl.edu/∼schmidt/locking-patterns.ps.gz

Message_Queue Class Implementation

```
template <class SYNCH_STRATEGY>
Message_Queue<SYNCH_STRATEGY>::Message_Queue
  (size_t hwm, size_t lwm)
  : not_empty_cond_ (lock_), not_full_cond_ (lock_),
    ... {}

template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::is_empty_i (void) const
{ return cur_bytes_ == 0 && cur_count_ == 0; }

template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::is_full_i (void) const
{ return cur_bytes_ > high_water_mark_; }

template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::is_empty (void) const
{
  Guard<SYNCH_STRATEGY::MUTEX> m (lock_);
  return is_empty_i ();
}

template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::is_full (void) const
{
  Guard<SYNCH_STRATEGY::MUTEX> m (lock_);
  return is_full_i ();
}
```

Enqueueing/Dequeueing Messages on a Message_Queue

```
template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::
enqueue_tail (Message_Block *new_item,
              Time_Value *tv) {
  Guard<SYNCH_STRATEGY::MUTEX>
    guard (lock_);
  // Wait while the queue is full.
  while (is_full_i ()) {
    // Release the <lock_> and wait
    // for timeout, signal, or space
    // to become available in the list.
    if (not_full_cond_.wait (tv) == -1)
      return -1;
  }
  // Actually enqueue the message at
  // the end of the list.
  enqueue_tail_i (new_item);

  // Tell blocked threads that
  // list has a new item!
  not_empty_cond_.signal ();
}
```

```
template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::
dequeue_head (Message_Block *&first_item,
              Time_Value *tv) {
  Guard<SYNCH_STRATEGY::MUTEX>
    guard (lock_);
  // Wait while the queue is empty.
  while (is_empty_i ()) {
    // Release lock_ and wait for timeout,
    // signal, or a new message being
    // placed in the list.
    if (not_empty_cond_.wait (tv) == -1)
      return -1;
  }
  // Actually dequeue the first message.
  dequeue_head_i (first_item);

  // Tell blocked threads that list
  // is no longer full.
  if (cur_bytes_ <= low_water_mark_)
    not_full_cond_.signal ();
}
```


Overcoming Algorithmic Decomposition Limitations

- Previous slides illustrate *tactical* techniques and patterns that:
  - *Reduce accidental complexity e.g.*,
    - Automate synchronization acquisition and release (Scoped Locking idiom)
    - Improve synchronization mechanisms (Adapter, Wrapper Facade, Monitor Object, Thread-Safe Interface, Strategized Locking patterns)
  - *Eliminate race conditions*
- Next, we describe *strategic* patterns, frameworks, and components to:
  - *Increase reuse and extensibility e.g.*,
    - Decoupling service, IPC, and demultiplexing
  - *Improve the flexibility of concurrency control*

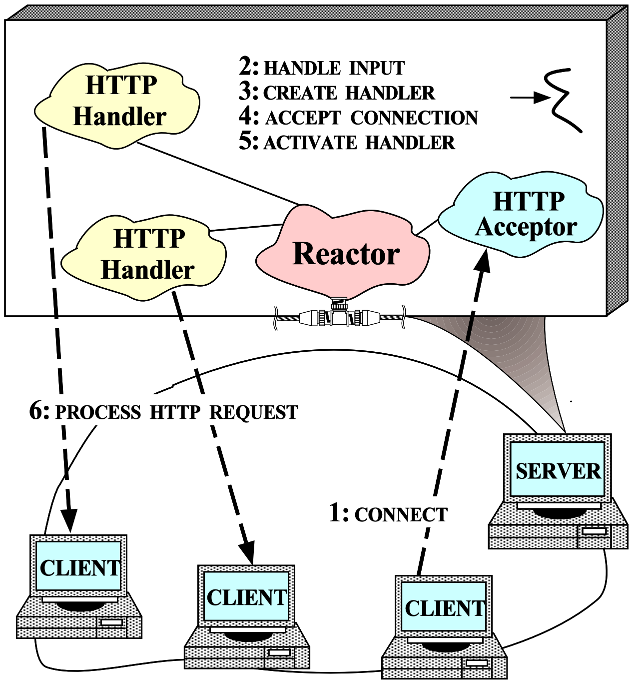
Selecting the Server's Concurrency Architecture

- Problem
  - A very strategic design decision for high-performance Web servers is selecting an efficient *concurrency architecture*
- Forces
  - No single concurrency architecture is optimal
  - Key factors include OS/hardware platform and workload
- Solution
  - Understand key alternative *concurrency* patterns

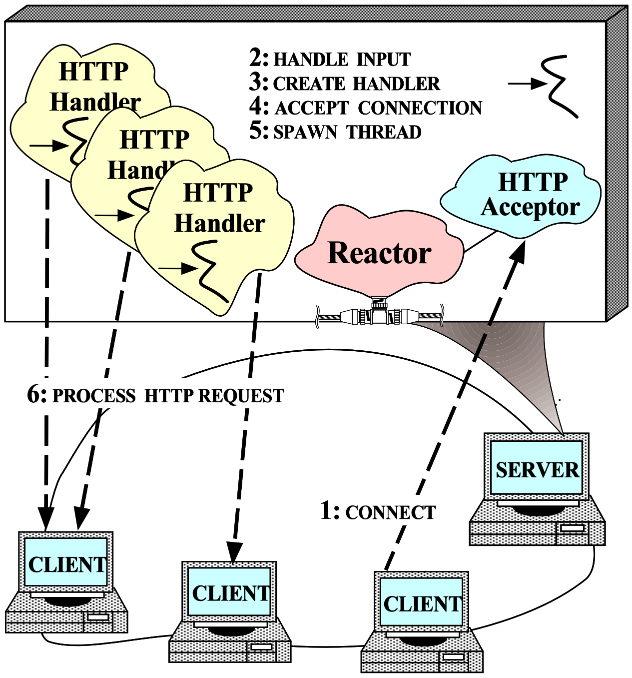
Concurrency Patterns in the Web Server

- The following example illustrates the *patterns* and *framework components* in an OO implementation of a concurrent Web Server
- There are various architectural patterns for structuring concurrency in a Web Server
  - *Reactive*
  - *Thread-per-request*
  - *Thread-per-connection*
  - *Synchronous Thread Pool*
    - Leader/Followers Thread Pool
    - Half-Sync/Half-Async Thread Pool
  - *Asynchronous Thread Pool*

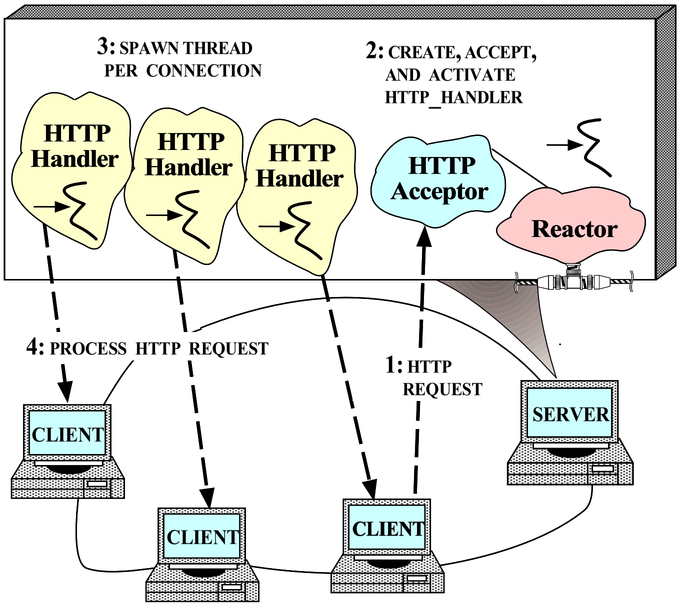
Reactive Web Server



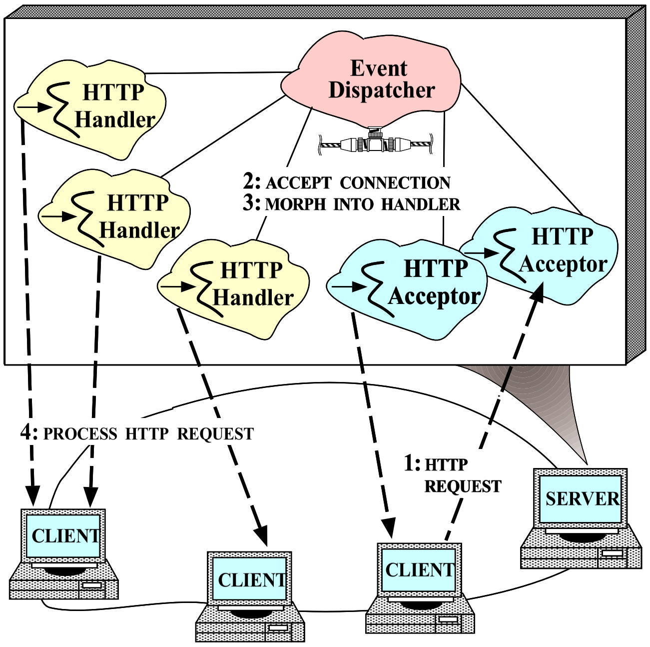
Thread-per-Request Web Server



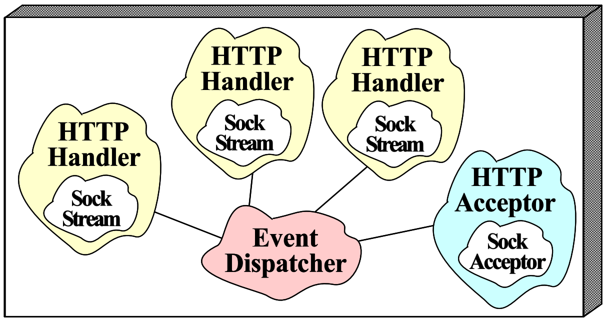
Thread-per-Connection Web Server



Leader/Followers Synchronous Thread Pool Web Server

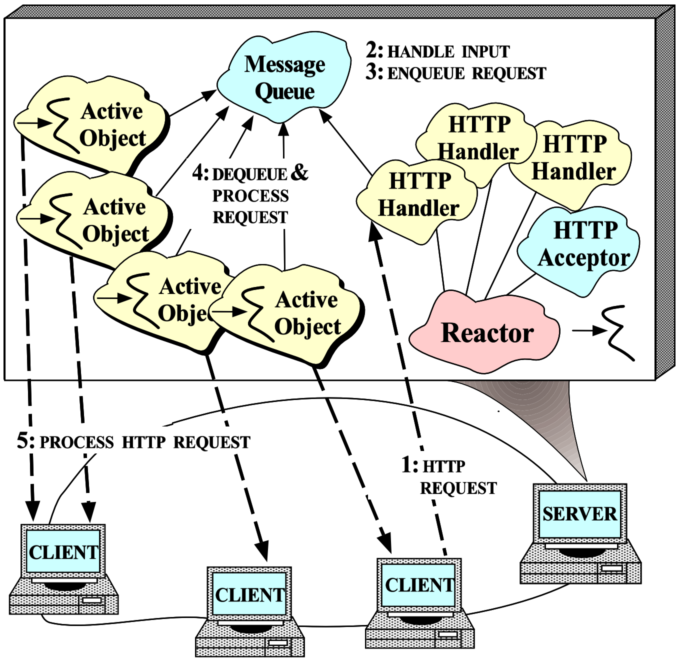


Web Server Software Architecture

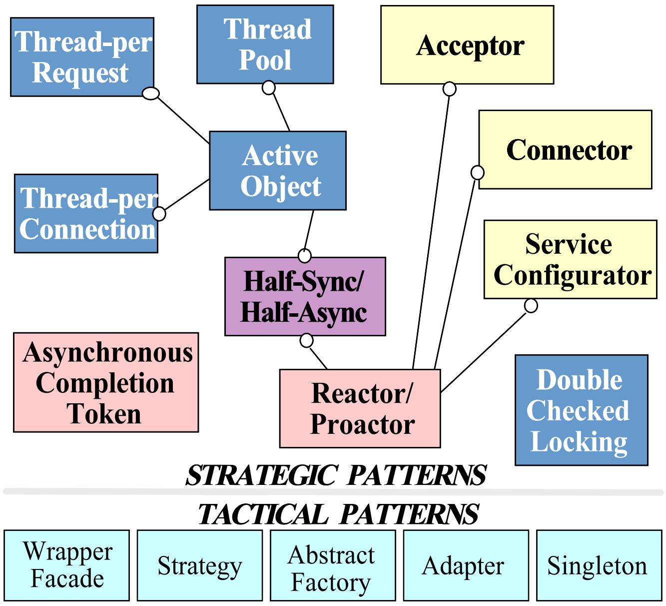


- Event Dispatcher
  - Encapsulates Web server concurrency and dispatching strategies
- HTTP Handlers
  - Parses HTTP headers and processes requests
- HTTP Acceptor
  - Accepts connections and creates HTTP Handlers

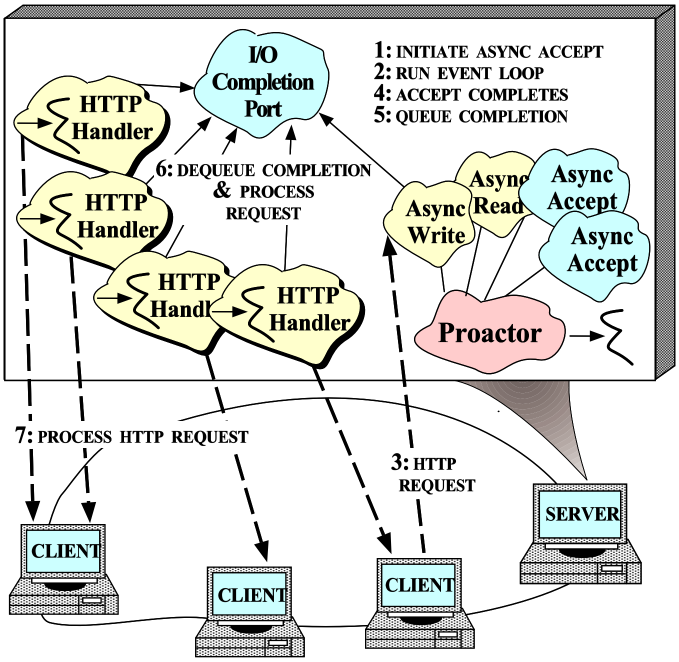
Half-Sync/Half-Async Synchronous Thread Pool Web Server



Patterns in the Web Server Implementation



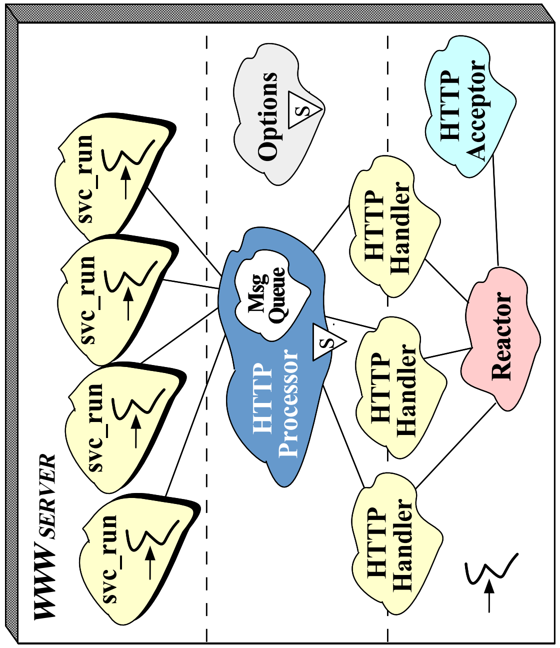
Asynchronous Thread Pool Web Server



Patterns in the Web Client/Server (cont'd)

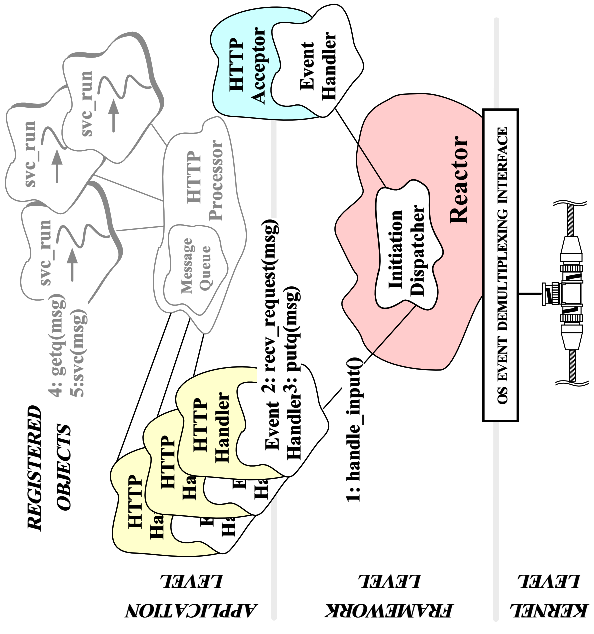
- The Web Client/Server uses same patterns as distributed logger
  - *i.e.*, Reactor, Service Configurator, Active Object, and Acceptor
- It also contains patterns with the following intents:
  - *Connector* → "Decouple the active connection and initialization of a peer service in a distributed system from the processing performed once the peer service is connected and initialized"
  - *Double-Checked Locking Optimization* → "Allows atomic initialization, regardless of initialization order, and eliminates subsequent locking overhead"
  - *Half-Sync/Half-Async* → "Decouples synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency"

Architecture of Our Web Server



www.cs.wustl.edu/~schmidt/HPL.ps.gz

An Integrated Reactive/Active Web Server



We're focusing on the Reactive layer here

`HTTP_Handler` Public Interface

```
template <class PEER_ACCEPTOR>
class HTTP_Handler : public
  Svc_Handler<PEER_ACCEPTOR::PEER_STREAM,
              NULL_SYNCH> {
public:
    // Entry point into HTTP_Handler,
    // called by HTTP_Acceptor.
  virtual int open (void *)
  {
    // Register with Reactor
    // to handle input.
    reactor ()->register_handler
      (this, Event_Handler::READ_MASK);
    // Register timeout in case client
    // doesn't send any HTTP requests.
    reactor ()->schedule_timer
      (this, 0, Time_Value (CLIENT_TIMEOUT));
  }
```

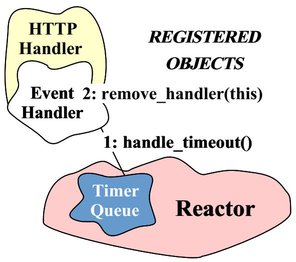
The `HTTP_Handler` is the Proxy for communicating with clients (e.g., Web browsers like Netscape or *i.e.*,)

- It implements the asynchronous portion of Half-Sync/Half-Async pattern

`HTTP_Handler` Protected Interface

```
protected:
  // Reactor dispatches this
  // method when clients timeout.
  virtual int handle_timeout
    (const Time_Value &, const void *)
  {
    // Remove from the Reactor.
    reactor ()->remove_handler
      (this, Event_Handler::READ_MASK);
  }
  // Reactor dispatches this method
  // when HTTP requests arrive.
  virtual int handle_input (HANDLE);

    // Receive/frame client HTTP
    // requests (e.g., GET).
  int recv_request (Message_Block *&);
};
```

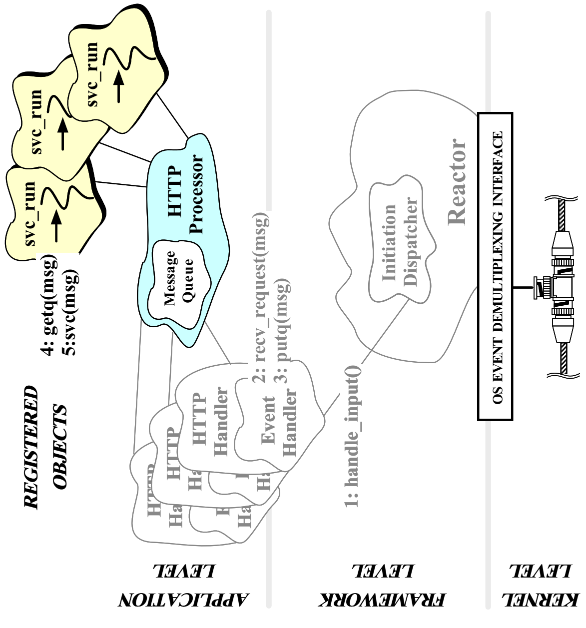
These methods are invoked by callbacks from the `Reactor`



Integrating Multi-threading

- Problem
  - Multi-threaded Web servers are needed since Reactive Web servers are often inefficient and non-robust
- Forces
  - Multi-threading can be very hard to program
  - No single multi-threading model is always optimal
- Solution
  - Use the *Active Object* pattern to allow multiple concurrent server operations in an OO-manner

Using the Active Object Pattern in the Web Server



We're focusing on the Active Object layer here

The `HTTP_Processor` Class

```
class HTTP_Processor
  : public Task<MT_SYNCH> {
private: HTTP_Processor (void);
public:
    // Singleton access point.
  static HTTP_Processor *instance (void);
    // Pass a request to the thread pool.
  virtual int put (Message_Block *,
                   Time_Value *);
    // Entry point into a pool thread.
  virtual int svc (void)
  {
    Message_Block *mb = 0;

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

- Processes HTTP requests using the "Thread-Pool" concurrency model
- This method implements the synchronous task portion of the Half-Sync/Half-Async pattern

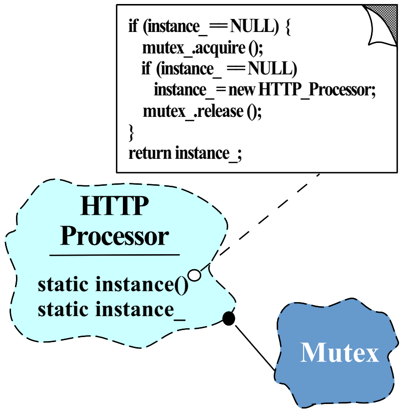
Using the Singleton Pattern

```
// Singleton access point.

HTTP_Processor *
HTTP_Processor::instance (void)
{
  // Beware of race conditions!
  if (instance_ == 0)
    // Create the Singleton "on-demand."
    instance_ = new HTTP_Processor;

  return instance_;
}

// Constructor creates the thread pool.

HTTP_Processor::HTTP_Processor (void)
{
  // Inherited from class Task.
  activate (THR_BOUND,
            Options::instance ()->threads ());
}
```


Subtle Concurrency Woes with the Singleton Pattern

- Problem
  - The canonical Singleton implementation has subtle "bugs" in multi-threaded applications
- Forces
  - Too much locking makes Singleton too slow...
  - Too little locking makes Singleton unsafe...
- Solution
  - Use the *Double-Checked Locking* optimization pattern to minimize locking **and** ensure atomic initialization

The Double-Checked Locking Optimization Pattern



www.cs.wustl.edu/~schmidt/DC-Locking.ps.gz

Intent

- *Allows atomic initialization, regardless of initialization order, and eliminates subsequent locking overhead*

Forces Resolved:

- Ensures atomic object initialization
- Minimizes locking overhead

Caveat!

- This pattern assumes *atomic* memory access

The ACE Singleton Template

```
template <class TYPE, class LOCK>
class Singleton : public Cleanup {
public:
  static TYPE *instance (void) {
    // Memory barrier could go here...
    if (singleton_ == 0) {
      Guard<LOCK> guard (Object_Manager::
        get_singleton_lock ());
      if (singleton_ == 0)
        singleton_ = new Singleton<TYPE>;
        // Memory barrier could go here...
      Object_Manager::at_exit (singleton_);
    }
    return singleton_->instance_;
  }
  virtual void cleanup (void *param = 0);
protected:
  Singleton (void);
  TYPE instance_;
  static Singleton<TYPE, LOCK> *
    singleton_;
};
```

Features

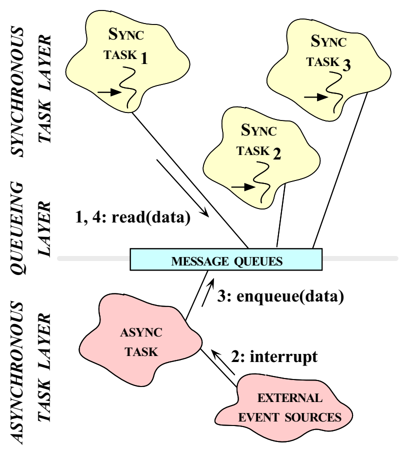
- Turns any class into a singleton
- Automates Double-Checked Locking Optimization
- Ensures automatic cleanup when process exits

www.cs.wustl.edu/~schmidt/ObjMan.ps.gz

Integrating Reactive and Multi-threaded Layers

- Problem
  - Justifying the hybrid design of our Web server can be tricky
- Forces
  - Engineers are never satisfied with the status quo ;-)
  - Substantial amount of time is spent re-discovering the *intent* of complex concurrent software design
- Solution
  - Use the *Half-Sync/Half-Async* pattern to explain and justify our Web server concurrency architecture

The Half-Sync/Half-Async Pattern



Intent

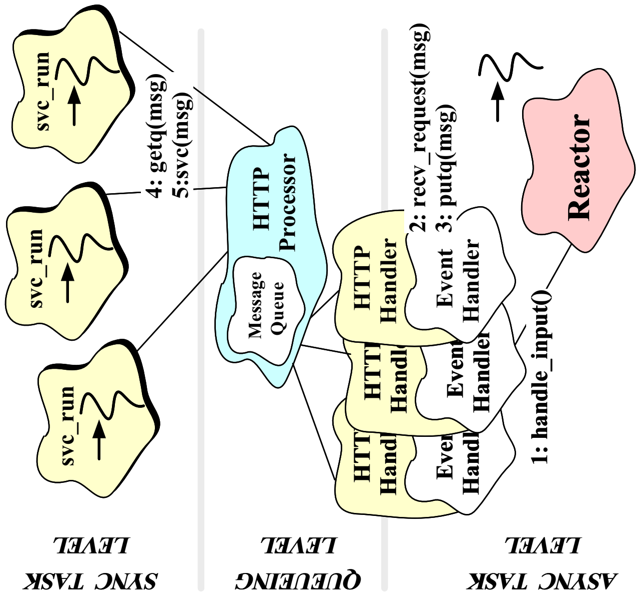
- *Decouples synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency*

Forces Resolved:

- Simplify programming
- Ensure efficient I/O

www.cs.wustl.edu/~schmidt/PLoP-96.ps.gz

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



Joining Async and Sync Tasks in the Web Server

```
// The following methods form the boundary
// between the Async and Sync layers.

template <class PA> int
HTTP_Handler<PA>::handle_input (HANDLE h)
{
  Message_Block *mb = 0;

  // Try to receive and frame message.
  if (recv_request (mb) == HTTP_REQUEST_COMPLETE) {
    reactor ()->remove_handler
      (this, Event_Handler::READ_MASK);
    reactor ()->cancel_timer (this);
    // Insert message into the Queue.
    HTTP_Processor<PA>::instance ()->put (mb);
  }
}

HTTP_Processor::put (Message_Block *msg,
                     Time_Value *timeout) {
  // Insert the message on the Message_Queue
  // (inherited from class Task).
  putq (msg, timeout);
}
```

Optimizing Our Web Server for Asynchronous Operating Systems

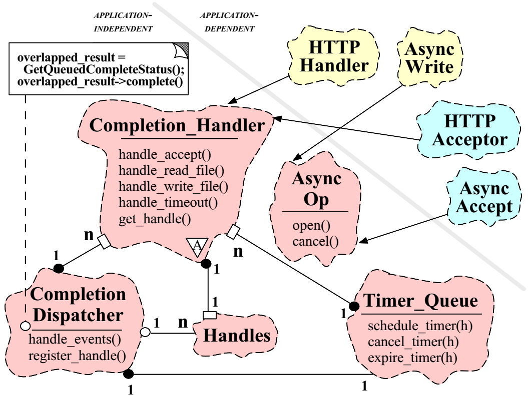
- Problem
  - Synchronous multi-threaded solutions are not always the most efficient
- Forces
  - Purely asynchronous I/O is quite powerful on some OS platforms
    - *e.g.*, Windows NT 4.x or UNIX w/`aio_*` calls
  - Good designs should be adaptable to new contexts
- Solution
  - Use the *Proactor* pattern to maximize performance on Asynchronous OS platforms

The Proactor Pattern

Intent

- *Demultiplexes and dispatches service requests that are triggered by the completion of asynchronous operations*

Resolves same forces as Reactor

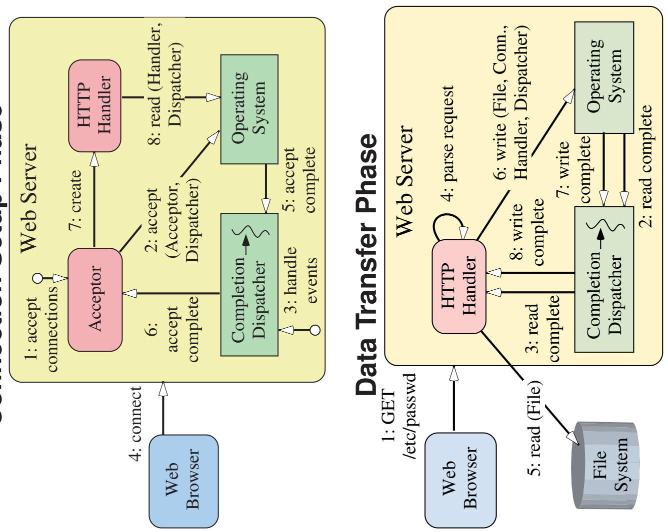


www.cs.wustl.edu/~schmidt/proactor.ps.gz

Using the Proactor for the Web Server

Connection Setup Phase

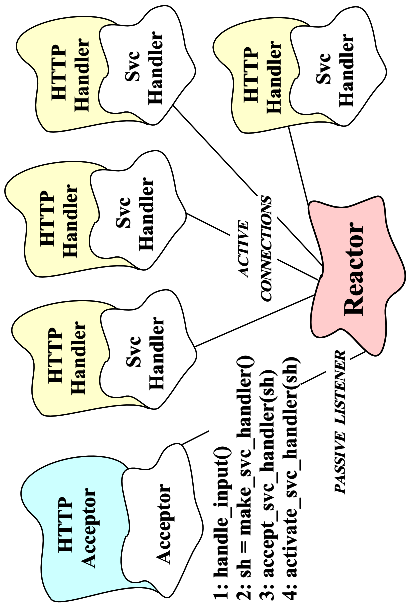
Data Transfer Phase



Structuring Service Initialization

- Problem
  - The *communication protocol* used between clients and the Web server is often orthogonal to the *initialization protocol*
- Forces
  - Low-level connection establishment APIs are tedious, error-prone, and non-portable
  - Separating *initialization* from *use* can increase software reuse substantially
- Solution
  - Use the *Acceptor* and *Connector* patterns to decouple passive service initialization from run-time protocol

Using the Acceptor Pattern in the Web Server



The `HTTP_Acceptor` is a *factory* that *creates*, *connects*, and *activates* an `HTTP_Handler`

`HTTP_Acceptor` Class Interface

```
template <class PEER_ACCEPTOR>
class HTTP_Acceptor : public Acceptor<
  HTTP_Handler<
   PEER_ACCEPTOR::PEER_STREAM>,
   // Note use of a "trait".
   PEER_ACCEPTOR>
{
public:
    // Called when HTTP_Acceptor is
    // dynamically linked.
  virtual int init (int argc, char *argv[]);

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

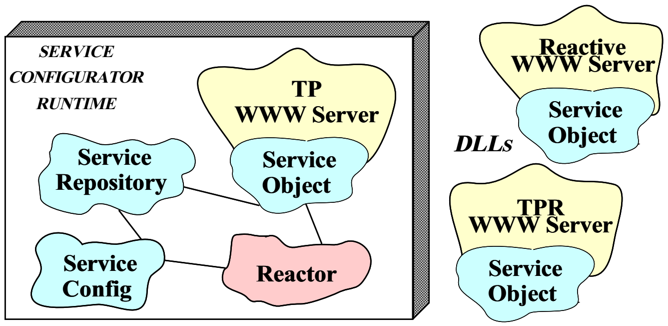
The `HTTP_Acceptor` class implements the Acceptor pattern

- *i.e.*, it accepts connections/initializes `HTTP_Handlers`

`HTTP_Acceptor` Class Implementation

```
// Initialize service when dynamically linked.

template <class PA> int
HTTP_Acceptor<PA>::init (int argc, char *argv[])
{
  Options::instance ()->parse_args (argc, argv);

  // Initialize the communication endpoint and
  // register to accept connections.
  peer_acceptor ().open
    (PA::PEER_ADDR (Options::instance ()->port ()),
     Reactor::instance ());
}

// Terminate service when dynamically unlinked.

template <class PA> int
HTTP_Acceptor<PA>::fini (void)
{
  // Shutdown threads in the pool.
  HTTP_Processor<PA>::instance ()->
    msg_queue ()->deactivate ();

  // Wait for all threads to exit.
  HTTP_Processor<PA>::instance ()->
    thr_mgr ()->wait ();
}
```


Using the Service Configurator Pattern in the Web Server



svc.conf FILE

```
dynamic Web_Server Service_Object *
  www_server:make_Web_Server() "-port 80"
```

Service Configurator Implementation in C++

The concurrent Web Server is configured and initialized via a configuration script

```
% cat ./svc.conf
dynamic Web_Server
  Service_Object *
  web_server:make_Web_Server()
  "-p 80 -t $THREADS"
# .dll or .so suffix added to
#  "web_server" automatically
```

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

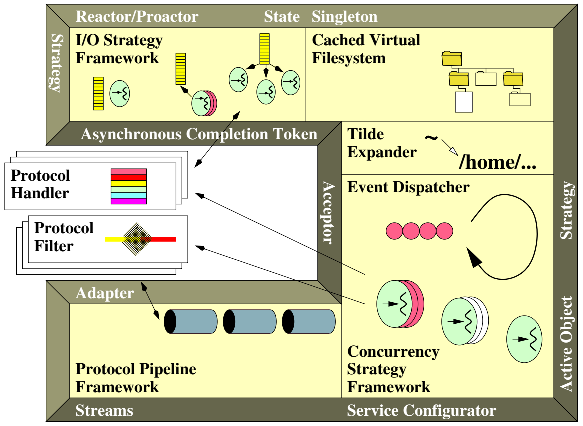
```
extern "C" Service_Object *
make_Web_Server (void);

Service_Object *
make_Web_Server (void)
{
  return new
    HTTP_Acceptor<SOCK_Acceptor>;
  // ACE dynamically unlinks and
  // deallocates this object.
}
```


Main Program for the Web Server

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

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

  Reactor::run_event_loop ();
  /* NOTREACHED */
}
```

- The `main` function is totally generic!
- Dynamically configure & execute Web Server
- Make any application "Web-enabled"

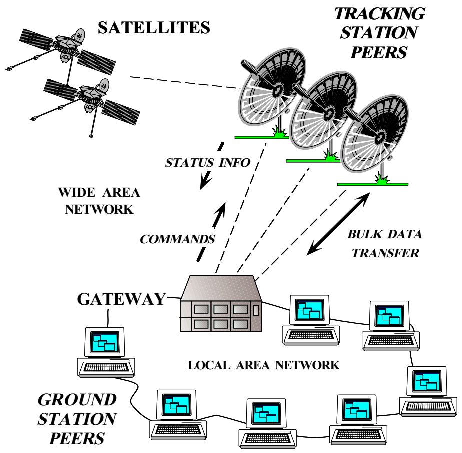
Optimizing the JAWS Framework



www.cs.wustl.edu/~jxh/research/

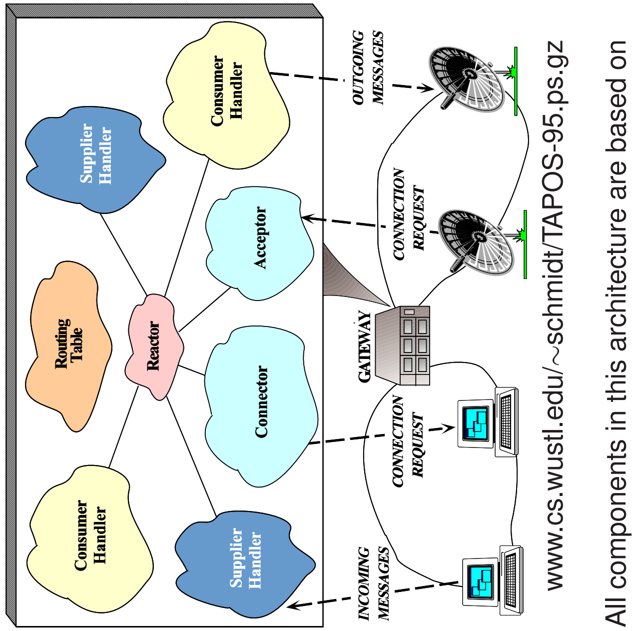
- Use lightweight concurrency
- Minimize locking
- Apply file caching and memory mapping
- Use "gather-write" mechanisms
- Minimize logging
- Pre-compute HTTP responses
- Avoid excessive `time` calls
- Optimize the transport interface

Application-level Telecom Gateway Example



- This example explores the *patterns* and *reusable framework* components for an *application-level Gateway*
- The Gateway routes messages between Peers
- Gateway and Peers are connected via TCP/IP

OO Software Architecture of the Gateway



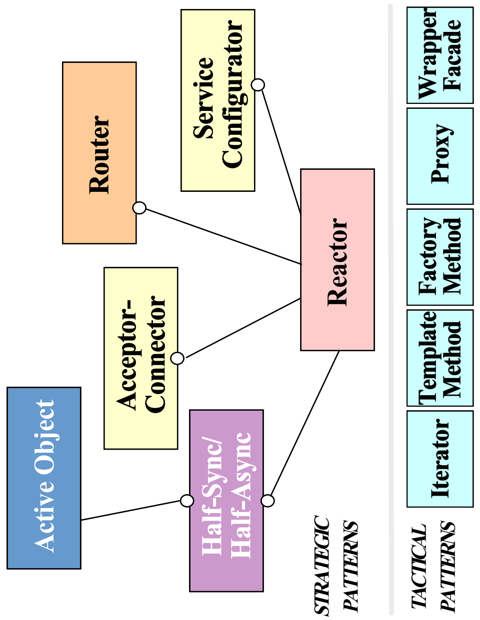
www.cs.wustl.edu/~schmidt/TAPOS-95.ps.gz

All components in this architecture are based on patterns from ACE

Gateway Behavior

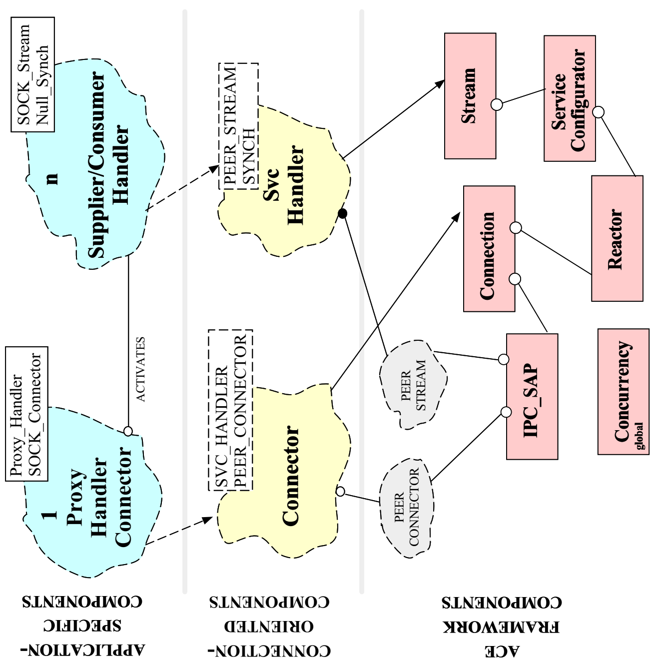
- Components in the Gateway behave as follows:
  1. `Gateway` parses configuration files that specify which Peers to connect with and which routes to use
  2. `Proxy_Handler_Connector` connects to Peers, then creates and activates `Proxy_Handler` subclasses (`Supplier_Handler` or `Consumer_Handler`)
  3. Once connected, Peers send messages to the Gateway
     – Messages are handled by an `Supplier_Handler`
     – `Supplier_Handlers` work as follows:
       * Receive and validate messages
       * Consult a `Routing_Table`
       * Forward messages to the appropriate Peer(s) via `Consumer_Handlers`

Patterns in the Gateway



The Gateway components are based upon a common *pattern language*

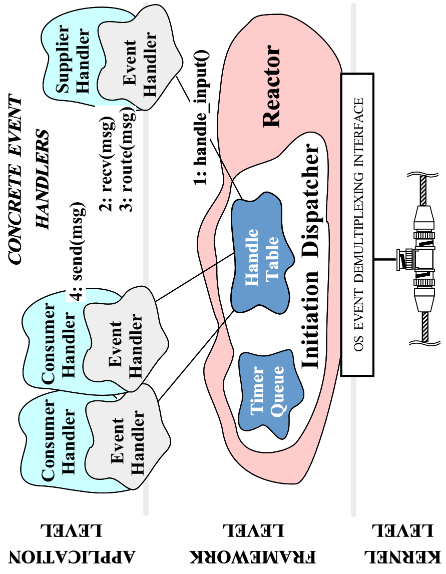
Class Diagram for Single-Threaded Gateway



OO Gateway Architecture

- *Application-specific components*
  - `Proxy_Handler`s route messages among Peers
- *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
- *Application-independent ACE framework components*
  - Perform IPC, explicit dynamic linking, event demultiplexing, event handler dispatching, multi-threading, etc.

Using the Reactor Pattern for the Gateway



Benefits

- Straightforward to program
- Concurrency control is trivial

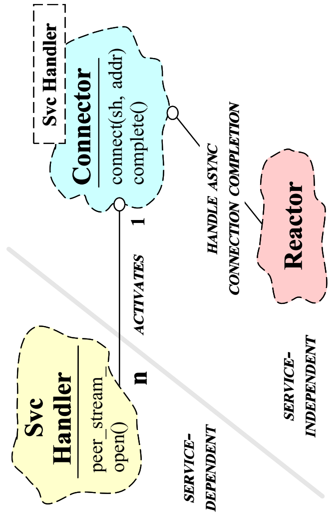
Liabilities

- Design is "brittle"
- Can't leverage multi-processors

Addressing Active Endpoint Connection and Initialization Challenges

- Problem
  - Application *communication* protocols are often orthogonal to their *connection establishment* and *service initialization* protocols
- Forces
  - Low-level connection APIs are error-prone and non-portable
  - Separating *initialization* from *processing* increases software reuse
  - Asynchronous connections are important over long-delay paths
- Solution
  - Use the *Acceptor-Connector* pattern to decouple connection and initialization protocols from the Gateway routing protocol

The Connector Pattern



www.cs.wustl.edu/~schmidt/Acc-Con.ps.gz

Intent

- *Decouple the active connection and initialization of a peer service in a distributed system from the processing performed once the peer service is connected and initialized*

Forces Resolved:

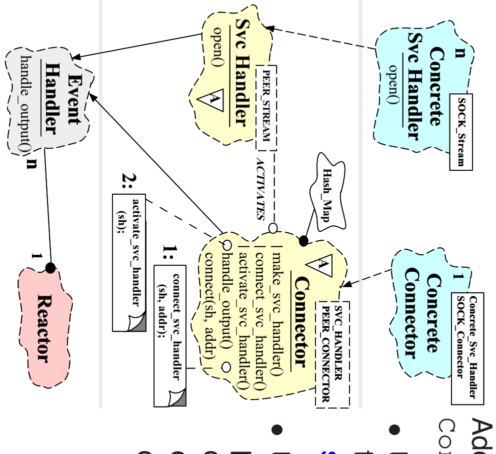
- Reuse connection code
- Efficiently setup connections with many peers or over long delay paths

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 &r_addr,
     Synch_Options &synch_options);

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


Structure of the Connector Pattern in ACE



Additional features of the ACE Connector

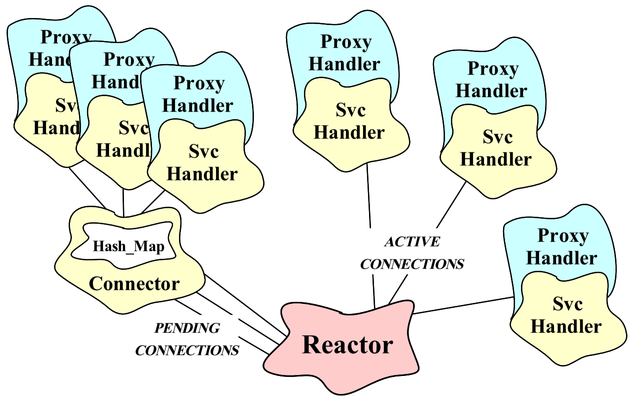
- Uses C++ parameterized types to *strategize* IPC and *service aspects*
- Uses Template Method pattern to strategize creation, connection establishment, and concurrency policies

Design Interlude: Motivation for the Synch_Options Class

- 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 *act = 0);
    // This is the default synchronous setting.
  static Synch_Options synch;
    // This is the default asynchronous setting.
  static Synch_Options asynch;
};
```


Using the Connector Pattern for the Gateway



- The Connector is a *factory*
  - *i.e.*, it *connects* and *activates* a Svc_Handler
- There's typically 1 Connector per-service

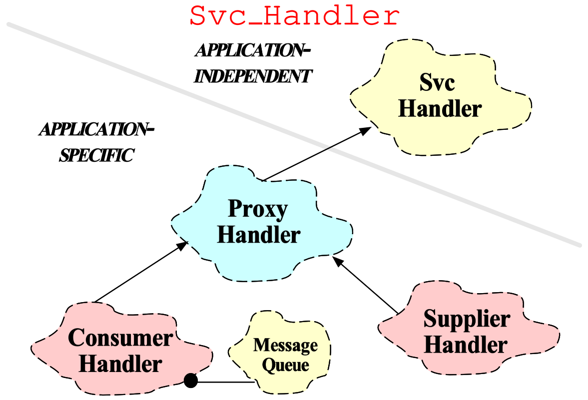
Connector Hook Method Implementations

```
template <class SH, class PC> SH *
Connector<SH, PC>::make_svc_handler (void) {
  return new SH;
}

template <class SH, class PC> int
Connector<SH, PC>::connect_svc_handler (SH &*sh,
           const PEER_CONNECTOR::PEER_ADDR &addr,
           ACE_Time_Value *timeout) {
  // Peer_Connector factory initiates connection.
  if (connector_.connect (sh, 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> for <sh> and return -1
  } else
    // Activate immediately if we're connected.
    activate_svc_handler (sh);
}

template <class SH, class PC> int
Connector<SH, PC>::activate_svc_handler (SH *sh) {
  if (sh->open ((void *) this) == -1) sh->close ();
}
```


Connector Class Protected Interface

```
protected:
  // Make a new connection.
  virtual SVC_HANDLER *make_svc_handler (void);
  // Accept a new connection.
  virtual int connect_svc_handler
   (SVC_HANDLER *&sh,
    const PEER_CONNECTOR::PEER_ADDR &addr,
    ACE_Time_Value *timeout);
  // Activate a service handler.
  virtual int activate_svc_handler (SVC_HANDLER *);

  // Demultiplexing hooks.
  virtual int handle_output (HANDLE); // Success.
  virtual int handle_input (HANDLE); // Failure.
  virtual int handle_timeout (Time_Value &,
                              const void *);
  // Table that maps an I/O handle to a Svc_Tuple *.
  Hash_Map_Manager<HANDLE, Svc_Tuple *, Null_Mutex>
    handler_map_;

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


Specializing Connector and Svc_Handler



- Producing an application that meets Gateway requirements involves *specializing* ACE components
  - Connector → Proxy_Handler_Connector
  - Svc_Handler → Proxy_Handler → Supplier_Handler and Consumer_Handler

Connector Class Implementation

```
// 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;
  // Hook methods.
  if (sh == 0)
    sh = make_svc_handler ();
  if (connect_svc_handler (sh, raddr,
                           timeout) != -1)
    activate_svc_handler (sh);
}
```

Proxy_Handler Class Public Interface

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

// Unique connection id that denotes Proxy_Handler.
typedef short CONN_ID;

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

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

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


Design Interlude: Parameterizing Synchronization into the ACE `Hash_Map_Manager`

- Q: *What's a good technique to implementing a Routing Table?*
- A: Use a `Hash_Map_Manager` container
  - ACE provides a `Hash_Map_Manager` container that associates *external ids* with *internal ids*, *e.g.*,
    - ∗ External ids (keys) → URI
    - ∗ Internal ids (values) → pointer to memory-mapped file
- Hashing provides $O(1)$ performance in the average-case

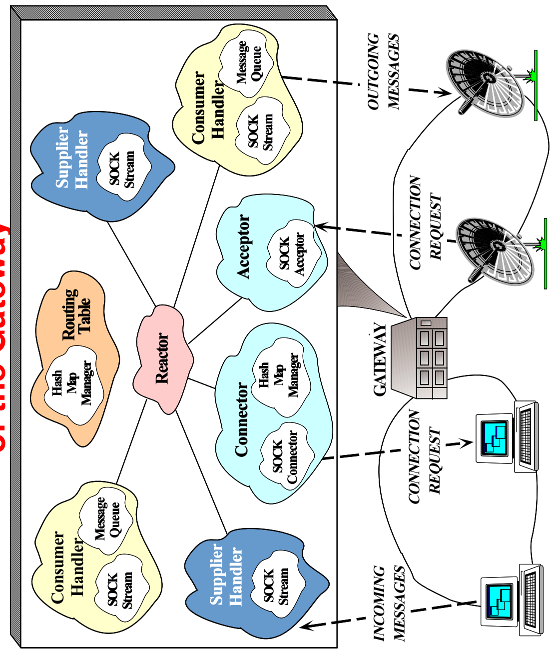
Applying the Strategized Locking pattern to the `Hash_Map_Manager` Class

```
template <class EXT_ID, class INT_ID,
          class LOCK>
class Hash_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> guard (lock_);
    // lock_.read_acquire ();
    if (find_i (ex, in))
      return true;
    else return false;
    // lock_.release ();
  }
private:
  LOCK lock_;
  bool find_i (EXT_ID, INT_ID &);
  // ...
};
```

The `Hash_Map_Manager` uses the template-based Strategized Locking pattern to

- Enhance reuse
- Parameterize different synchronization strategies, *e.g.*:
  - `Null_Mutex`, `Thread_Mutex`, `RW_Mutex`, etc.

Detailed OO Architecture of the Gateway



Note the use of other ACE components, such as the socket wrapper facades and the `Hash_Map_Manager`

Supplier_Handler Interface

```
class Supplier_Handler : public Proxy_Handler
{
public:
  Supplier_Handler (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_;
};
```


Consumer_Handler Interface

```
class Consumer_Handler : public Proxy_Handler
{
public:
  Consumer_Handler (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 *);
};
```


Proxy_Handler_Connector Class Interface

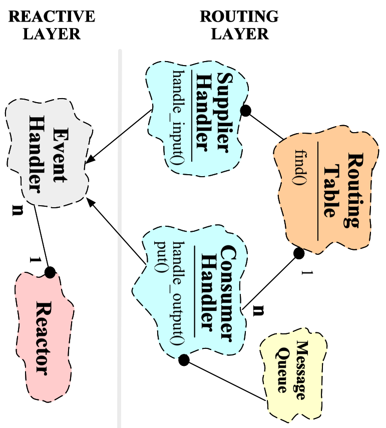
```
class Proxy_Handler_Connector :
  public Connector
    <Proxy_Handler,
    // Type of Svc Handler
    SOCK_Connector>
    // Connection factory
{
public:
    // Initiate (or reinitiate)
    // a connection on
    // the Proxy_Handler.
  int initiate_connection
    (Proxy_Handler *);
}
```

- `Proxy_Handler_Connector` is a concrete factory class that:
  - Establishes connections with Peers to produce `Proxy_Handler`s
  - Activates `Proxy_Handler`s, which then route messages
- `Proxy_Handler_Connector` also ensures reliability by restarting failed connections

Proxy_Handler_Connector Implementation

```
// (re)initiate a connection to a Proxy_Handler
int
Proxy_Handler_Connector::initiate_connection
  (Proxy_Handler *ph)
{
  // Use asynchronous connections...
  if (connect (ph,
               ph->addr (),
               Synch_Options::asynch) == -1) {
    if (errno == EWOULDBLOCK)
      // No error, we're connecting asynchronously.
      return -1;
    else
      // This is a real error, so reschedule
      // ourselves to reconnect.
      reactor ()->schedule_timer
        (ph, 0, ph->timeout ());
  }
  else // We're connected synchronously!
    return 0;
}
```

The Router Pattern



www.cs.wustl.edu/~schmidt/TAPOS-95.ps.gz

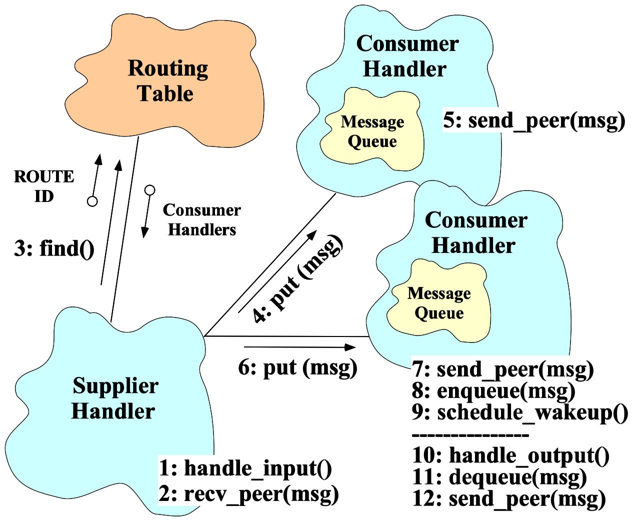
Intent

- *Decouple multiple sources of input from multiple sources of output to prevent blocking*

Forces Resolved:

- Keep misbehaving connections from disrupting the QoS for well-behaved connections
- Allow different concurrency strategies for `Supplier` and `Consumer Handlers`.

Collaboration in Single-threaded Gateway Routing



Note the complex cooperative scheduling logic required to handle output flow control correctly

`Supplier_Handler` and `Consumer_Handler` Implementations

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

int Supplier_Handler::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 Consumer_Handler::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)
}
```


`Supplier_Handler` Message Routing

```
// Route message from a Peer.
int Supplier_Handler::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<Proxy_Handler *>
    si (re->destinations ());
  // Multicast message.
  for (Proxy_Handler *out_ph;
       si.next (out_ph) != -1;
       si.advance ()) {
    Message_Block *newmsg = data->duplicate ();
    if (out_ph->put (newmsg) == -1) // Drop message.
      newmsg->release (); // Decrement ref count.
  }
  delete route_addr;
}
```

Peer_Message Schema

```
// 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.
  char sdu_[MAX_PAYLOAD_SIZE]; // Message payload.
};
```


Design Interlude: Tips on Handling Flow Control

- 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...
  - Strategy pattern: *give reasonable default, but allow substitution*
- A related design issue deals with avoiding output blocking if a Peer connection becomes flow controlled

Supplier Handler Message Reception

```
// Pseudo-code for recv'ing msg via non-blocking I/O

int Supplier_Handler::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

  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
}
```


Design Interlude: Using the ACE `Reactor` to Handle Flow Control

- Q: *How can a flow controlled* `Consumer_Handler` *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 `Proxy_Handler` is able to transmit again

Performing a Non-blocking `put` of a Message

```
int Consumer_Handler::nonblk_put
  (Message_Block *mb) {
  // Try sending message
  // via 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 it's ok to send again.
    reactor ()->schedule_wakeup
      (this, Event_Handler::WRITE_MASK);
  }
}
```

This method is called in two situations:

1. When first trying to send over a connection
2. When flow control abates

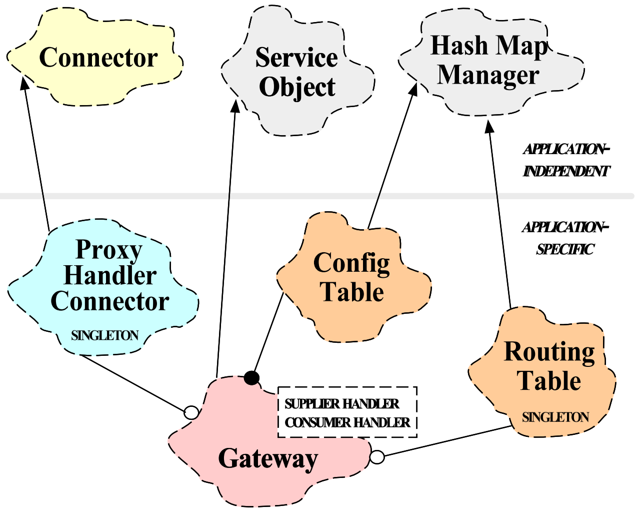
Sending a Message to a Consumer

```
int
Consumer_Handler::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) */ {
    // Decrement reference count.
    mb->release ();
    errno = 0;
  }
  return n;
}
```


Finish Sending when Flow Control Abates

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

int
Consumer_Handler::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 ()
      reactor ()->cancel_wakeup
        (this, Event_Handler::WRITE_MASK);
  }
}
```


The `Gateway` Class



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

Dynamically Configuring Gateway into an Application

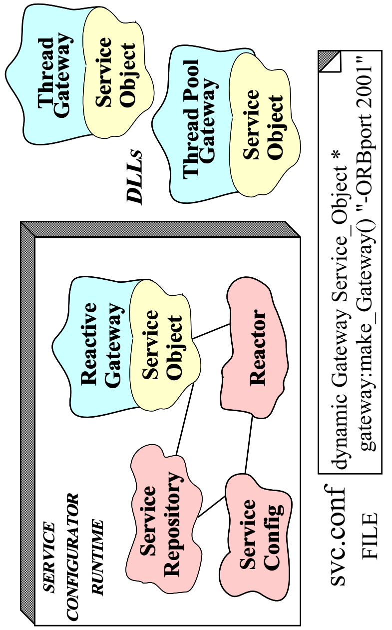
Parameterized by Example of the Service Configurator pattern

Proxy_Handler

```
template
  <class SUPPLIER_HANDLER,
   class CONSUMER_HANDLER>
class Gateway
  : public Service_Object
{
public:
    // Perform initialization.
  virtual int init
    (int argc, char *argv[]);

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

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

  // Run forever, performing the
  // configured services.

  Reactor::run_event_loop ();

  /* NOTREACHED */
}
```


Using the Service Configurator Pattern for the Gateway



We can replace the single-threaded Gateway with a multi-threaded Gateway

Dynamic Linking a Gateway Service

The Gateway service is configured via scripting in a `svc.conf` file:

```
% cat ./svc.conf
static Svc_Manager
  "-p 5150"
dynamic Gateway
Service_Object *
gateway:make_Gateway()
        "-d -p $PORT"
# .dll or .so suffix
# added to "gateway"
# automatically
```

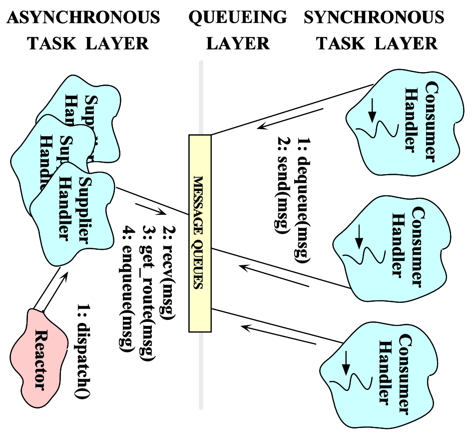
Dynamically linked factory function that allocates a new single-threaded Gateway

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

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


Concurrency Strategies for Patterns

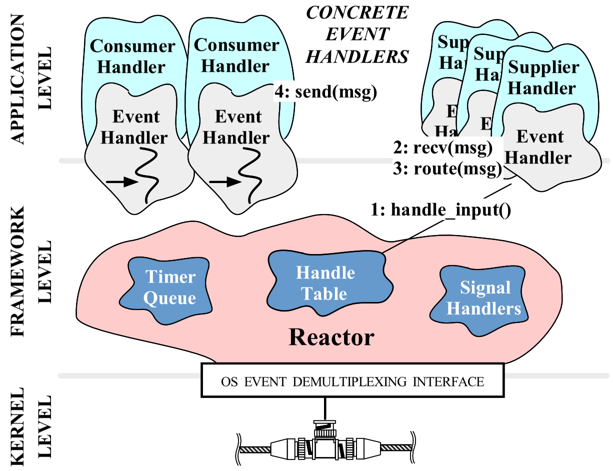
- The Acceptor-Connector pattern does not constrain the concurrency strategies of a `Svc_Handler`
- There are three common choices:
  1. *Run service in same thread of control*
  2. *Run service in a separate thread*
  3. *Run service in a separate process*
- Observe how our patterns and ACE framework push this decision to the "edges" of the design
  - This greatly increases reuse, flexibility, and performance tuning

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



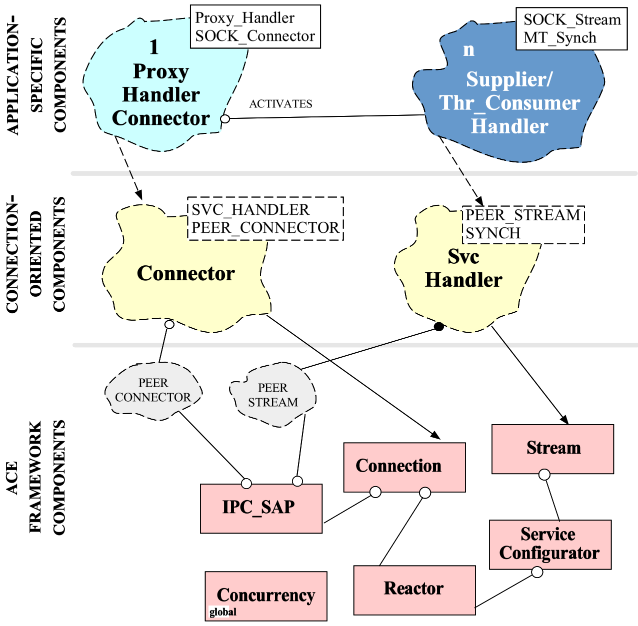
- Reactor plays the role of "async" layer
- Active Object plays the role of "sync" layer

Using the Active Object Pattern for the Gateway

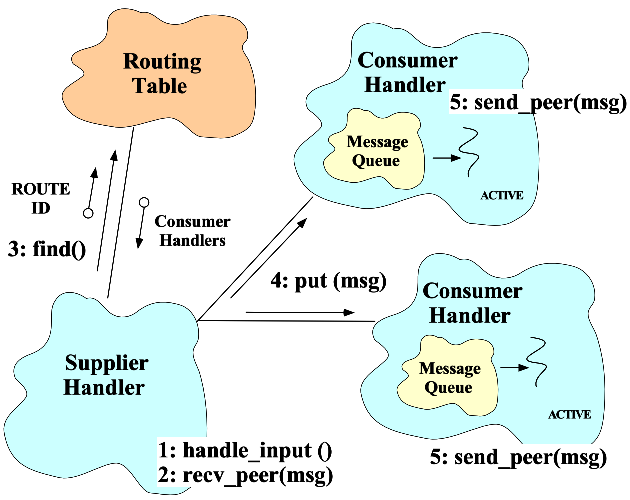


Each `Consumer_Handler` is implemented as an Active Object

Class Diagram for Multi-Threaded Gateway



Collaboration in Multi-threaded Gateway Routing



Note that this design is much simpler since the OS thread scheduler handles blocking

Thr_Consumer_Handler Class Interface

```
#define ACE_USE_MT
#include Proxy_Handler.h

class Thr_Consumer_Handler
  : public Consumer_Handler
{
public:
    // Initialize the object and
    // spawn new thread.
  virtual int open (void *);
    // Send a message to a peer.
  virtual int put (Message_Block *,
                   Time_Value *);
    // Transmit peer messages
    // in separate thread.
  virtual int svc (void);
};
```

New subclass of `Proxy_Handler` uses the Active Object pattern for the `Consumer_Handler`

- 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

Thr_Consumer_Handler Class Implementation

Override definition in the `Consumer_Handler` class

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

  activate (THR_DETACHED);
}
```

- The multi-threaded version of `open` is slightly different since it spawns a new thread to become an active object!
- `activate` is a pre-defined method on ACE `Task`

Thr_Consumer_Handler Class Implementation

```
// Queue up a message for transmission.

int
Thr_Consumer_Handler::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_Consumer_Handler::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);
}
```

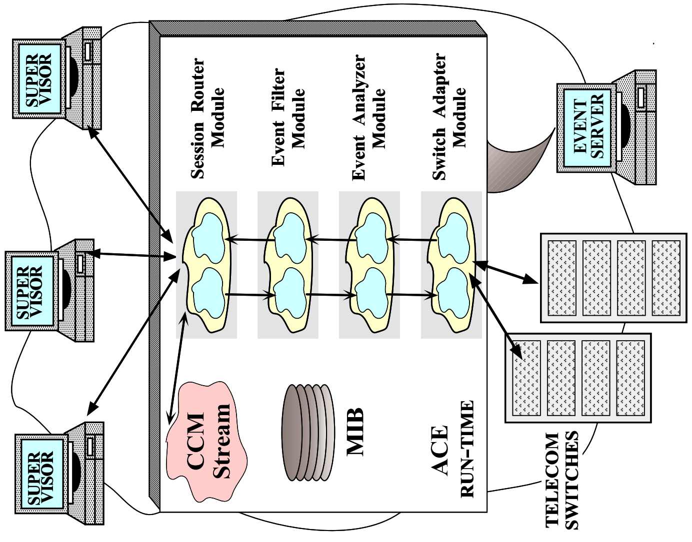

Dynamic Linking a Threaded Gateway Service

```
% cat ./svc.conf
remove Gateway
dynamic Gateway
Service_Object *
thr_gateway:make_Gateway()
            "-d"
# .dll or .so suffix added
#  to  "thr_Gateway"
# automatically
```

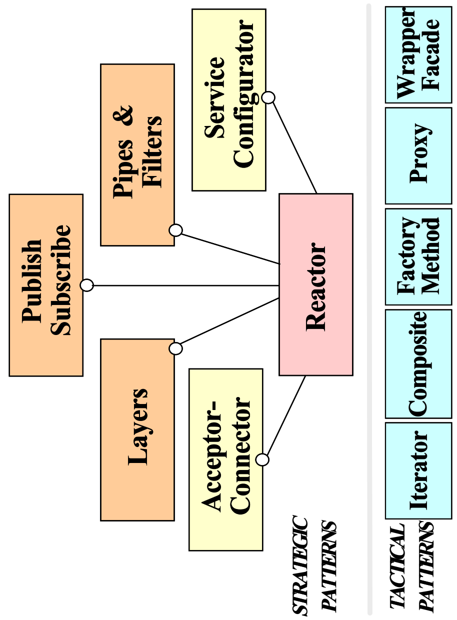
Dynamically linked factory function that allocates a multi-threaded Gateway object

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

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

Call Center Manager (CCM) Event Server Example



Patterns in the CCM Event Server

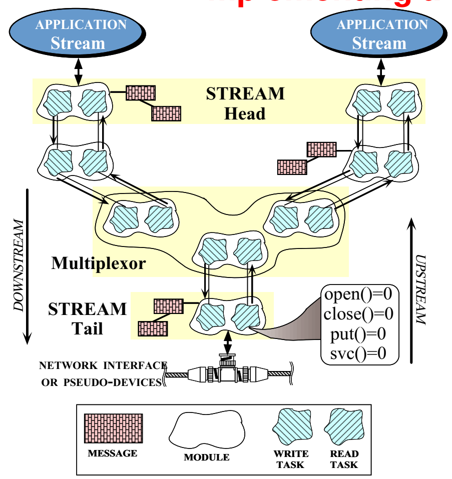


- The Event Server components are based upon a common *pattern language*
- www.cs.wustl.edu/~schmidt/DSEJ-94.ps.gz

ACE Streams

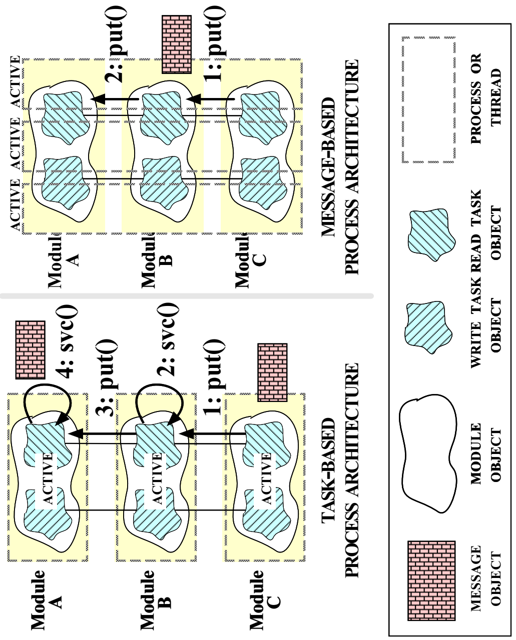
- An ACE `Stream` allows flexible configuration of layered processing modules
- It is an implementation of the *Pipes and Filters* architecture pattern
  - This pattern provides a structure for systems that process a stream of data
  - Each processing step is encapsulated in a filter `Module` component
  - Data is passed through pipes between adjacent filters, which can be re-combined
- The CCM Event Server was design and implemented using ACE Streams

Implementing a Stream in ACE



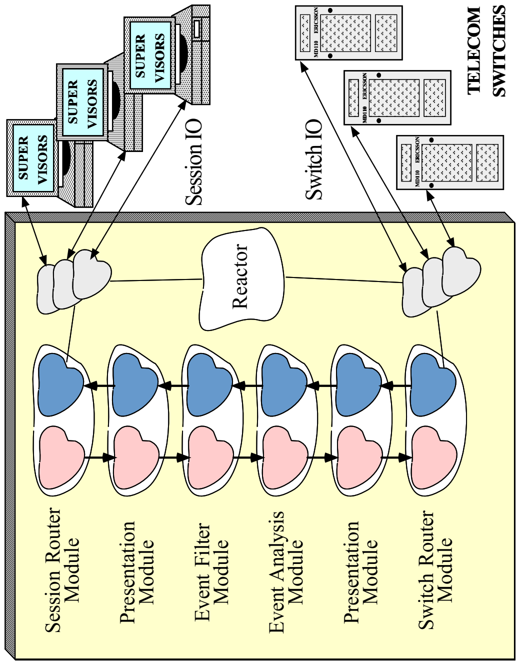
- A `Stream` contains a stack of `Modules`
- Each `Module` contains two `Tasks`
  - *i.e.*, a *read* `Task` and a *write* `Task`
- Each `Task` contains a `Message_Queue` and a pointer to a `Thread_Manager`

Alternative Concurrency Models for Message Processing



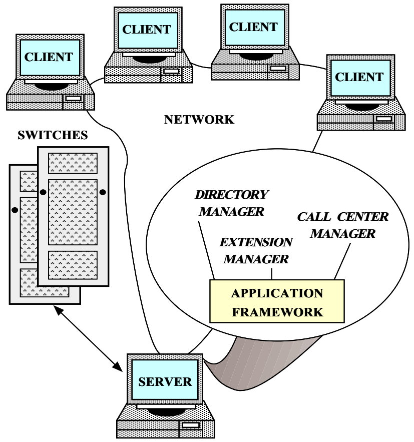
Task-based models are more intuitive but less efficient than Message-based models

Using the Pipes and Filters Pattern for the CCM Event Server



www.cs.wustl.edu/~schmidt/DSEJ-94.ps.gz

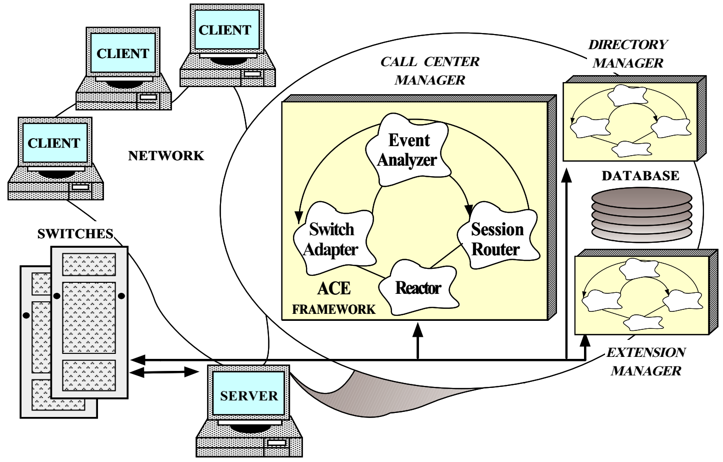
Broader Context: External OS for Telecom Switches



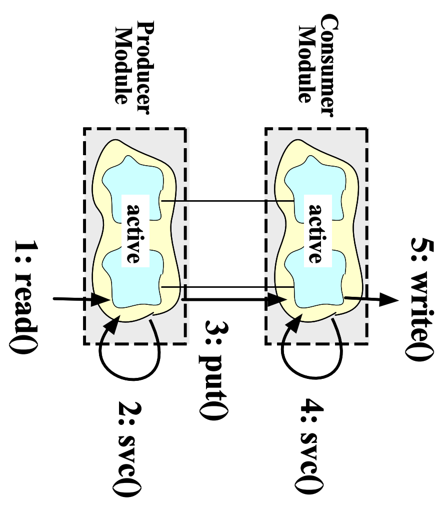
Features

- Allow clients to manage various aspects of telecom switches without modifying the switch software
- Support reuse of existing components based on a common architectural framework

Applying ACE Streams to External OS



ACE Stream Example: Parallel I/O Copy



- Program copies stdin to stdout via the use of a multi-threaded `Stream`
- Stream implements a "bounded buffer"
- Since the data flow is uni-directional the "read" `Task` is always ignored

`Producer` Class Interface

typedef short-hands for templates

```
typedef Stream<MT_SYNCH> MT_Stream;
typedef Module<MT_SYNCH> MT_Module;
typedef Task<MT_SYNCH> MT_Task;
```

Define the `Producer` interface

```
class Producer : public MT_Task
{
public:
  // Initialize Producer.
  virtual int open (void *)
  {
    // activate() is inherited from class Task.
    activate (THR_BOUND);
  }

  // Read data from stdin and pass to consumer.
  virtual int svc (void);
  // ...
};
```


`Producer` Class Implementation

Runs as a Active Object in a separate thread

```
int Producer::svc (void) {
  for (;;) {
    Message_Block *mb;
    // Allocate a new message.
    ACE_NEW_RETURN (mb,
                    Message_Block (BUFSIZ),
                    -1);
    // Keep reading stdin, until we reach EOF.
    ssize_t n = ACE_OS::read (STDIN,
                              mb->rd_ptr (),
                              mb->size ());
    if (n <= 0) {
      // Send shutdown message to other
      // thread and exit.
      mb->length (0);
      this->put_next (mb);
      break;
    } else {
      mb->wr_ptr (n); // Adjust write pointer.

      // Send the message to the other thread.
      this->put_next (mb);
    }
  }
}
```


`Consumer` Class Interface

Define the `Consumer` interface

```
class Consumer : public MT_Task
{
public:
  // Initialize Consumer.
  virtual int open (void *)
  {
    // <activate> is inherited from class Task.
    activate (THR_BOUND);
  }

  // Enqueue the message on the Message_Queue
  // for subsequent processing in <svc>.
  virtual int put (Message_Block *,
                   Time_Value * = 0)
  {
    // <putq> is inherited from class Task.
    return putq (mb, tv);
  }

  // Receive message from producer
  // and print to stdout.
  virtual int svc (void);
};
```

`Consumer` Class Implementation

`Consumer` dequeues a message from the `Message_Queue`, writes the message to the stderr stream, and deletes the message

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

  // Keep looping, reading a message from the queue,
  // until we get a 0 length message, then quit.
  for (;;) {
      int result = getq (mb);

      if (result == -1) break;
      int length = mb->length ();

      if (length > 0)
        ACE_OS::write (STDOUT, mb->rd_ptr (),
                       length);
      mb->release ();

      if (length == 0) break;
    }
}
```

The `Producer` sends a 0-sized message to inform the `Consumer` to stop reading and exit

Main Driver Function for the Stream

Create `Producer` and `Consumer` Modules and push them onto the Stream

```
int main (int argc, char *argv[])
{
  // Control hierarchically-related
  // Active Objects.
  MT_Stream stream;

  // All processing is performed in the
  // Stream after <push>'s complete.
  stream.push (new MT_Module
               ("Consumer", new Consumer);
  stream.push (new MT_Module
               ("Producer", new Producer));

  // Barrier synchronization: wait for
  // the threads, to exit, then exit
  // the main thread.
  Thread_Manager::instance ()->wait ();
}
```


Evaluation of the ACE Stream Framework

- Structuring active objects via a `Stream` allows "interpositioning"
  - *i.e.*, similar to adding a filter in a UNIX pipeline
- New functionality may be added by "pushing" a new processing `Module` onto a `Stream`, *e.g.*:

```
stream.push (new MT_Module ("Consumer", new Consumer))
stream.push (new MT_Module ("Filter", new Filter));
stream.push (new MT_Module ("Producer", new Producer));
```

- Communication between `Module`s is typically *anonymous*

Concurrency Strategies

- Developing correct, efficient, and robust concurrent applications is challenging
- Below, we examine a number of strategies that addresses challenges related to the following:
  - *Concurrency control*
  - *Library design*
  - *Thread creation*
  - *Deadlock and starvation avoidance*

General Threading Guidelines

- A threaded program should not arbitrarily enter non-threaded (*i.e.*, "unsafe") code
- Threaded code may refer to unsafe code only from the main thread
  - *e.g.*, beware of `errno` problems
- Use reentrant OS library routines ('_r') rather than non-reentrant routines
- Beware of thread global process operations, such as file I/O
- Make sure that `main` terminates cleanly
  - *e.g.*, beware of `thr_exit (3T)`, `exit (2)`, and "falling off the end"

Thread Creation Strategies

- Use threads for independent jobs that must maintain state for the life of the job
- Don't spawn new threads for very short jobs
- Use threads to take advantage of CPU concurrency
- Only use "bound" threads when absolutely necessary
- If possible, tell the threads library how many threads are expected to be active simultaneously
  - *e.g.*, use `thr_setconcurrency`

General Locking Guidelines

- Don't hold locks across long duration operations (*e.g.*, I/O) that can impact performance
  - Use ACE "Tokens" instead...
- Beware of holding non-recursive mutexes when calling a method outside a class
  - The method may reenter the module and deadlock
- Don't lock at too small of a level of granularity
- Make sure that threads obey the global lock hierarchy
  - But this is easier said than done...

Locking Alternatives

- *Code locking*
  - Associate locks with body of functions
    - Typically performed using bracketed mutex locks
  - Often called a *Monitor Object*
- *Data locking*
  - Associate locks with data structures and/or objects
  - Permits a more fine-grained style of locking
- Data locking allows more concurrency than code locking, but may incur higher overhead

Single-lock Strategy

- One way to simplify locking is use a single, application-wide mutex lock
- Each thread must acquire the lock before running and release it upon completion
- The advantage is that most legacy code doesn't require changes
- The disadvantage is that parallelism is eliminated
  - Moreover, interactive response time may degrade if the lock isn't released periodically

Monitor Object Strategy

- A more OO locking strategy is to use a Monitor Object
  - www.cs.wustl.edu/∼schmidt/monitor.ps.gz
- Monitor Object synchronization mechanisms allow concurrent method invocations
  - Either eliminate access to shared data or use synchronization objects
  - Hide locking mechanisms behind method interfaces
    - Therefore, modules should not export data directly
- Advantage is transparency
- Disadvantages are increased overhead from excessive locking and lack of control over method invocation order

Active Object Strategy

- Each task is modeled as an active object that maintains its own thread of control
- Messages sent to an object are queued up and processed asynchronously with respect to the caller
  - *i.e.*, the order of execution may differ from the order of invocation
- This approach is more suitable to message passing-based concurrency
- The ACE `Task` class can be used to implement active objects
  - www.cs.wustl.edu/∼schmidt/Act-Obj.ps.gz

Invariants

- In general, an invariant is a condition that is always true
- For concurrent programs, an invariant is a condition that is always true when an associated lock is *not* held
  - However, when the lock is held the invariant may be false
  - When the code releases the lock, the invariant must be re-established
- *e.g.*, enqueueing and dequeueing messages in the `Message_Queue` class

Run-time Stack Problems

- Most threads libraries contain restrictions on stack usage
  - The initial thread gets the "real" process stack, whose size is only limited by the stacksize limit
  - All other threads get a fixed-size stack
    - Each thread stack is allocated off the heap and its size is fixed at startup time
- Therefore, be aware of "stack smashes" when debugging multi-threaded code
  - Overly small stacks lead to bizarre bugs, *e.g.*,
    - Functions that weren't called appear in backtraces
    - Functions have strange arguments

Deadlock

- Permanent blocking by a set of threads that are competing for a set of resources
- Caused by "circular waiting," *e.g.*,
  - A thread trying to reacquire a lock it already holds
  - Two threads trying to acquire resources held by the other
    - *e.g.*, T_1 and T_2 acquire locks L_1 and L_2 in opposite order
- One solution is to establish a global ordering of lock acquisition (*i.e.*, a *lock hierarchy*)
  - May be at odds with encapsulation...

Avoiding Deadlock in OO Frameworks

- Deadlock can occur due to properties of OO frameworks, *e.g.*,
  - *Callbacks*
  - *Inter-class method calls*
- There are several solutions
  - Release locks before performing callbacks
    - Every time locks are reacquired it may be necessary to reevaluate the state of the object
  - Make private "helper" methods that assume locks are held when called by methods at higher levels
  - Use a Token or a Recursive Mutex

Recursive_Thread_Mutex Implementation

Here is portable implementation of recursive thread mutexes available in ACE:

```
class Recursive_Thread_Mutex
{
public:
  // Initialize a recursive mutex.
  Recursive_Thread_Mutex (void);
  // Implicitly release a recursive mutex.
  ~Recursive_Thread_Mutex (void);
  // Acquire a recursive mutex.
  int acquire (void) const;
  // Conditionally acquire a recursive mutex.
  int tryacquire (void) const;
  // Releases a recursive mutex.
  int release (void) const;

private:
  Thread_Mutex nesting_mutex_;
  Condition_Thread_Mutex mutex_available_;
  thread_t owner_id_;
  int nesting_level_;
};
```

Acquiring a `Recursive_Thread_Mutex`

```
Recursive_Thread_Mutex::acquire (void) const
{
  thread_t t_id = Thread::self ();

  Guard<Thread_Mutex> mon (nesting_mutex_);

  // If there's no contention, grab mutex.
  if (nesting_level_ == 0) {
    owner_id_ = t_id;
    nesting_level_ = 1;
  }
  else if (t_id == owner_id_)
    // If we already own the mutex, then
    // increment nesting level and proceed.
    nesting_level_++;
  else {
    // Wait until nesting level drops
    // to zero, then acquire the mutex.
    while (nesting_level_ > 0)
      mutex_available_.wait ();

    // Note that at this point
    // the nesting_mutex_ is held...

    owner_id_ = t_id;
    nesting_level_ = 1;
  }
  return 0;
```


Releasing and Initializing a `Recursive_Thread_Mutex`

```
Recursive_Thread_Mutex::release (void) const
{
  thread_t t_id = Thread::self ();

  // Automatically acquire mutex.
  Guard<Thread_Mutex> mon (nesting_mutex_);

  nesting_level_--;

  if (nesting_level_ == 0) {
    assert (owner_id_ == t_id);

    // Put the mutex into a known state.
    owner_id_ = OS::NULL_thread;

    // Inform waiters that the mutex is free.
    mutex_available_.signal ();
  }
  return 0;
}

Recursive_Thread_Mutex::Recursive_Thread_Mutex
  (void)
  : nesting_level_ (0), owner_id_ (OS::NULL_thread),
    mutex_available_ (nesting_mutex_)
{}
```


Avoiding Starvation

- Starvation occurs when a thread never acquires a mutex even though another thread periodically releases it
- The order of scheduling is often undefined
- This problem may be solved via:
  - Use of "voluntary pre-emption" mechanisms
    - *e.g.*, `thr_yield` or `Sleep`
  - Using an ACE "Token" that strictly orders acquisition and release

Drawbacks to Multi-threading

- *Performance overhead*
  - Some applications do not benefit directly from threads
  - Synchronization is not free
  - Threads should be created for processing that lasts at least several 1,000 instructions
- *Correctness*
  - Threads are not well protected against interference
  - Concurrency control issues are often tricky
  - Many legacy libraries are not thread-safe
- *Development effort*
  - Developers often lack experience
  - Debugging is complicated (lack of tools)

Lessons Learned using OO 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
  - Verification and validation of generic components is hard

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++***
  - Some language features are not implemented
  - Some development environments are primitive
  - Language has many dark corners and sharp edges
    - Purify helps alleviate many problems...

Lessons Learned using OOD

- Good designs can be boiled down to a few key principles:
  - Separate interface from implementation
  - Determine what is *common* and what is *variable* with an interface and an implementation
  - Allow substitution of *variable* implementations via a *common* interface
    - *i.e.*, the “open/closed” principle & Aspect-Oriented Programming (AOP)
  - Dividing *commonality* from *variability* should be goal-oriented rather than exhaustive
- Design is not simply drawing a picture using a CASE tool, using graphical UML notation, or applying patterns
  - Design is a fundamentally *creative* activity

Software Principles for Distributed Applications

- ***Use patterns/frameworks to decouple policies/mechanisms***
  - Enhance reuse of common concurrent programming components
- ***Decouple service functionality from configuration***
  - Improve flexibility and performance
- ***Use classes, inheritance, dynamic binding, and parameterized types***
  - Improve extensibility and modularity
- ***Enhance performance/functionality with OS features***
  - *e.g.*, implicit and explicit dynamic linking and multi-threading
- ***Perform commonality/variability analysis***
  - Identify uniform interfaces for *variable* components and support pluggability of variation

Conferences and Workshops on Patterns

- Pattern Language of Programs Conferences
  - PLoP, September, 2000, Monticello, Illinois, USA
  - European PLoP, July, 2000, Kloster Irsee, Germany
  - Koala PLoP, May, 2000, Melbourne, Australia
  - hillside.net/patterns/conferences/
- USENIX COOTS
  - January, 2001, San Antonio
  - www.cs.wustl.edu/ schmidt/activities-chair.html

Patterns and Frameworks Literature

Books

- Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* AW, ’94
- *Pattern Languages of Program Design* series by AW, ’95-’99.
- Siemens & Schmidt, *Pattern-Oriented Software Architecture*, Wiley, volumes ’96 & ’00
  - Drafts for POSA2 available at ~schmidt/patterns/patterns.html

Special Issues in Journals

- October ’96 CACM (guest editors: D. Schmidt, R. Johnson, and M. Fayad)
- October ’97 CACM (guest editors: D. Schmidt and M. Fayad)

Magazines

- C++ Report, *e.g.*, columns by Coplien, Vlissides, Schmidt & Vinoski, and Martin

How to Obtain ACE Software and Technical Support

- The ADAPTIVE Communication Environment (ACE) is an OO toolkit designed according to key network programming patterns
- All source code for ACE is freely available
  - www.cs.wustl.edu/~schmidt/ACE.html
- Mailing lists
  - `ace-users@cs.wustl.edu`
  - `ace-users-request@cs.wustl.edu`
  - `ace-announce@cs.wustl.edu`
  - `ace-announce-request@cs.wustl.edu`
- Newsgroup
  - `comp.soft-sys.ace`

Concluding Remarks

- 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 *patterns* to create communication *frameworks* containing *components*
- *Frameworks* and *components* are an effective way to achieve systematic reuse of software