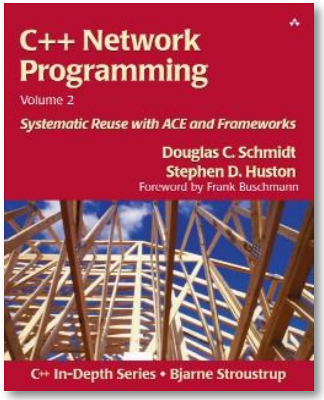


C++ Network Programming

Systematic Reuse with ACE & Frameworks

Dr. Douglas C. Schmidt

d.schmidt@vanderbilt.edu

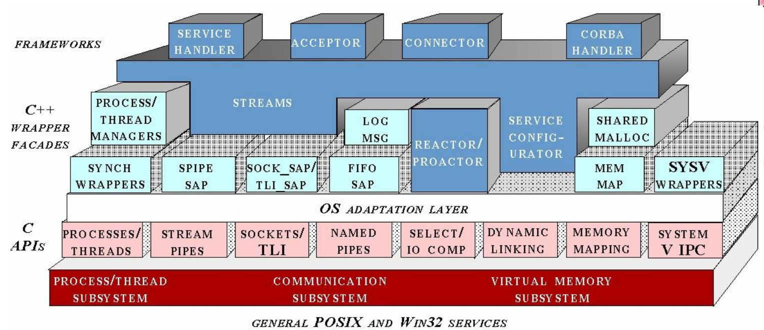
www.dre.vanderbilt.edu/~schmidt/

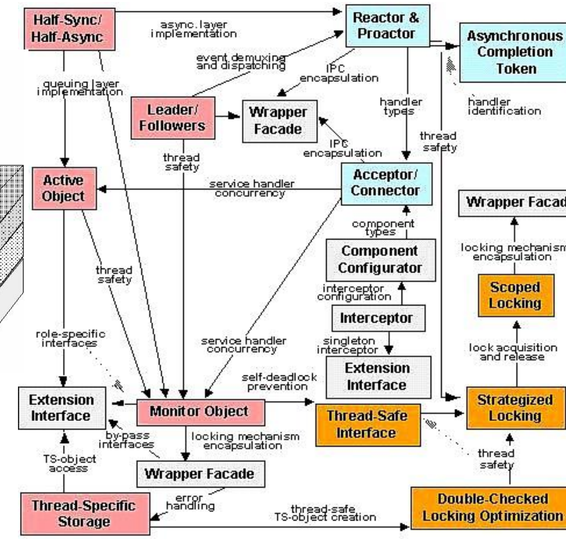
Professor of EECS
Vanderbilt University
Nashville, Tennessee

Presentation Outline

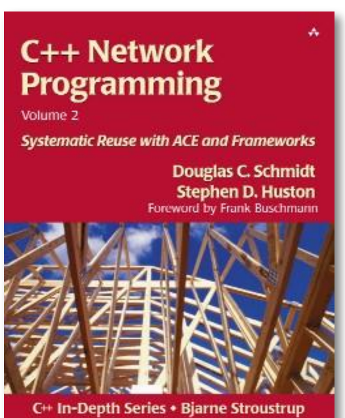
Cover OO techniques & language features that enhance software quality

- ***Patterns,*** which embody reusable software *architectures & designs*
- ***Frameworks***, which can be customized to support concurrent & networked applications



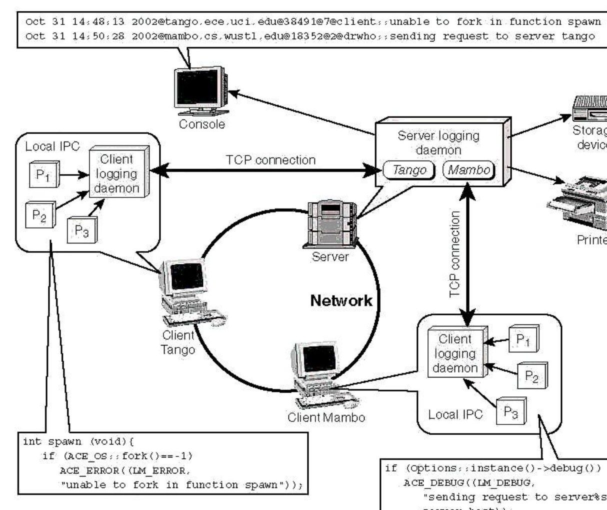


- ***OO language features**, e.g., classes, dynamic binding & inheritance, parameterized types*

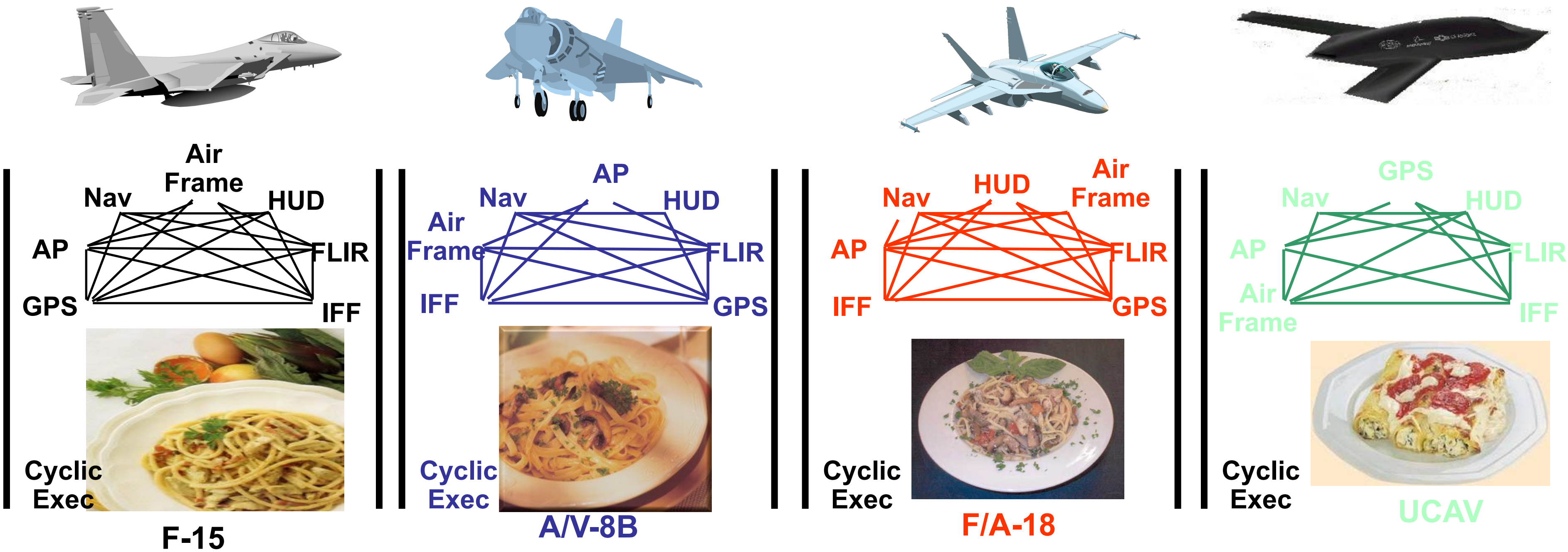


Presentation Organization

1. *Overview of product-line architectures*
2. *Overview of frameworks*
3. *Server/service & configuration design dimensions*
4. *Patterns & frameworks in ACE + applications*



Motivation

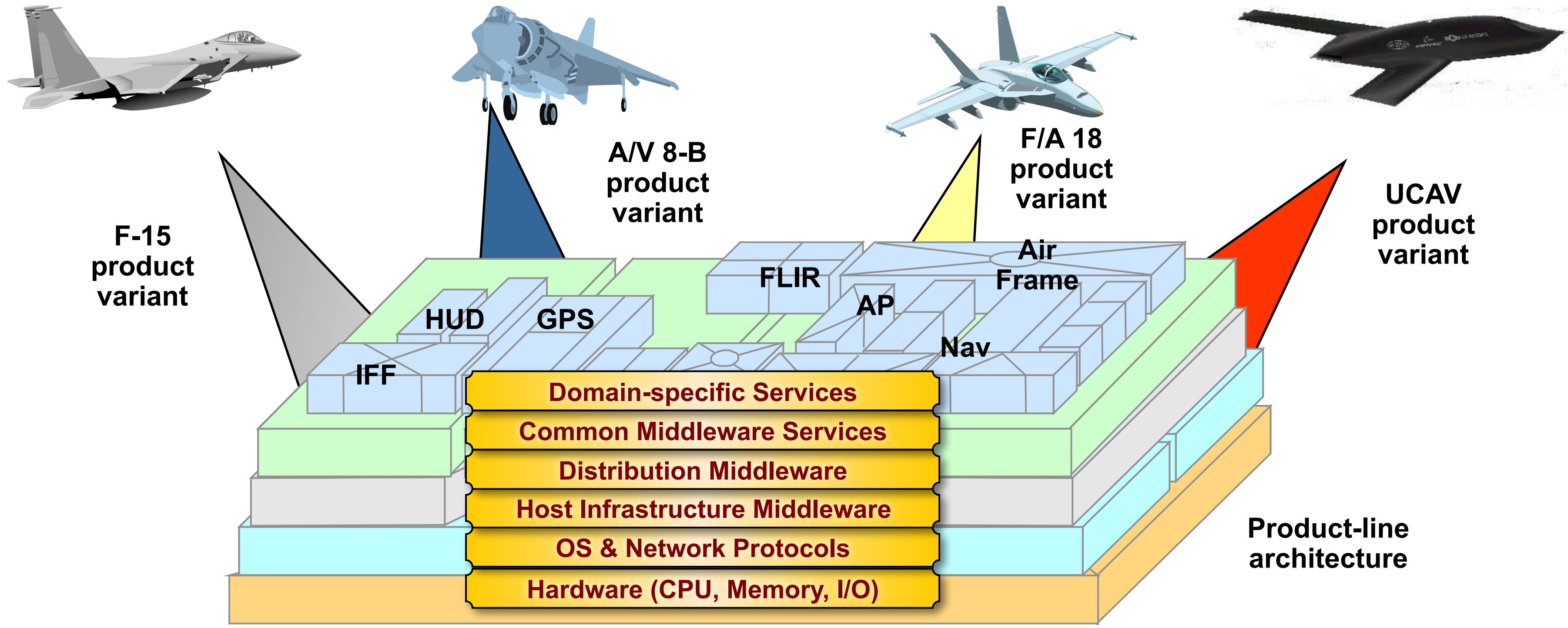


Legacy distributed real-time & embedded (DRE) systems have historically been:

- Stovepiped
- Proprietary
- Brittle & non-adaptive
- Expensive
- Vulnerable

Consequence: Small HW/SW changes have big (negative) impact on DRE system QoS & maintenance

Motivation

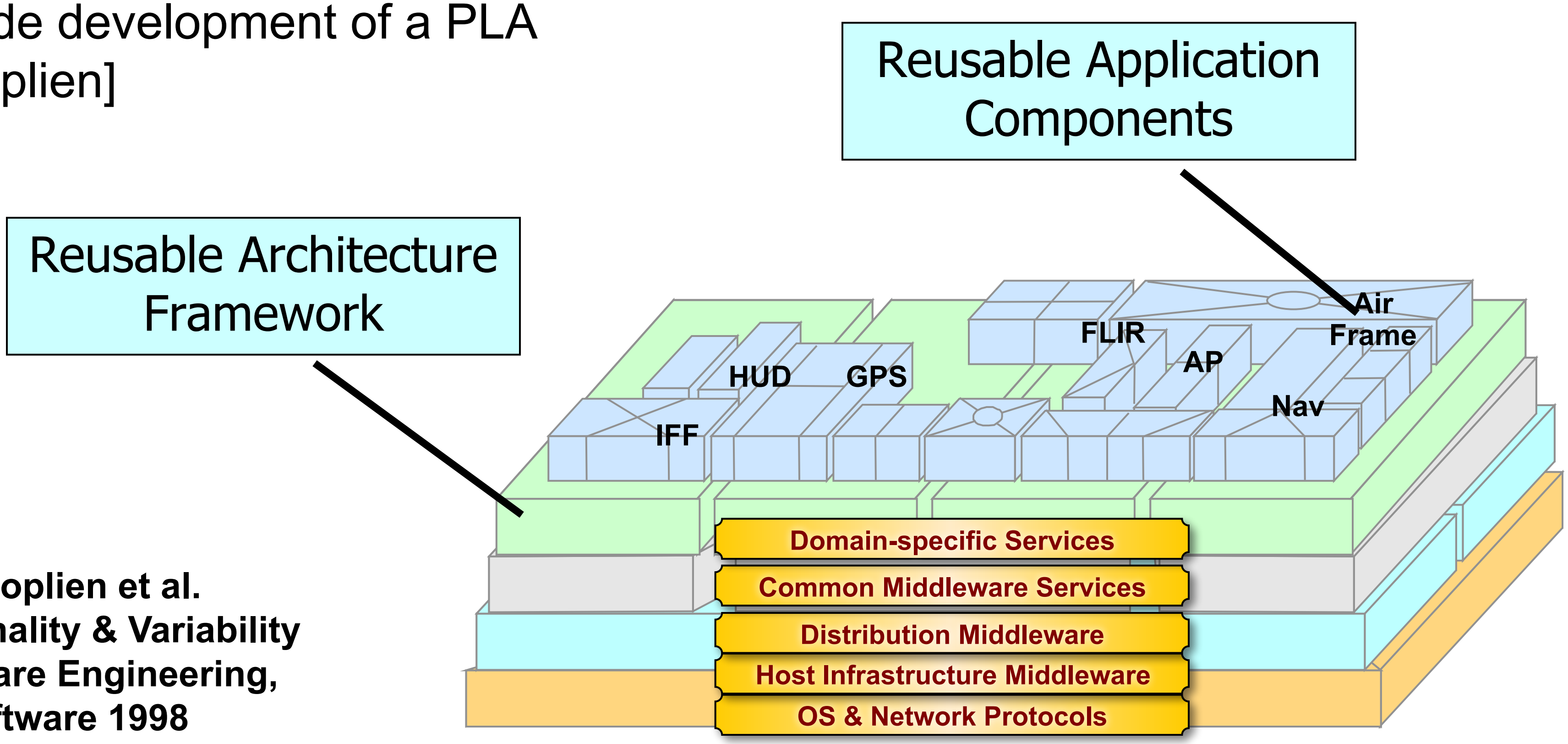


- ***Frameworks*** factors out many reusable general-purpose & domain-specific services from traditional DRE application responsibility
- Essential for ***product-line architectures (PLAs)***
- Product-lines & frameworks offer many configuration opportunities
  - e.g., component distribution & deployment, user interfaces & operating systems, algorithms & data structures, etc.

Overview of Product-line Architectures (PLAs)

- PLA characteristics are captured via *Scope, Commonalities, & Variabilities (SCV) analysis*
  - This process can be applied to identify commonalities & variabilities in a domain to guide development of a PLA [Coplien]

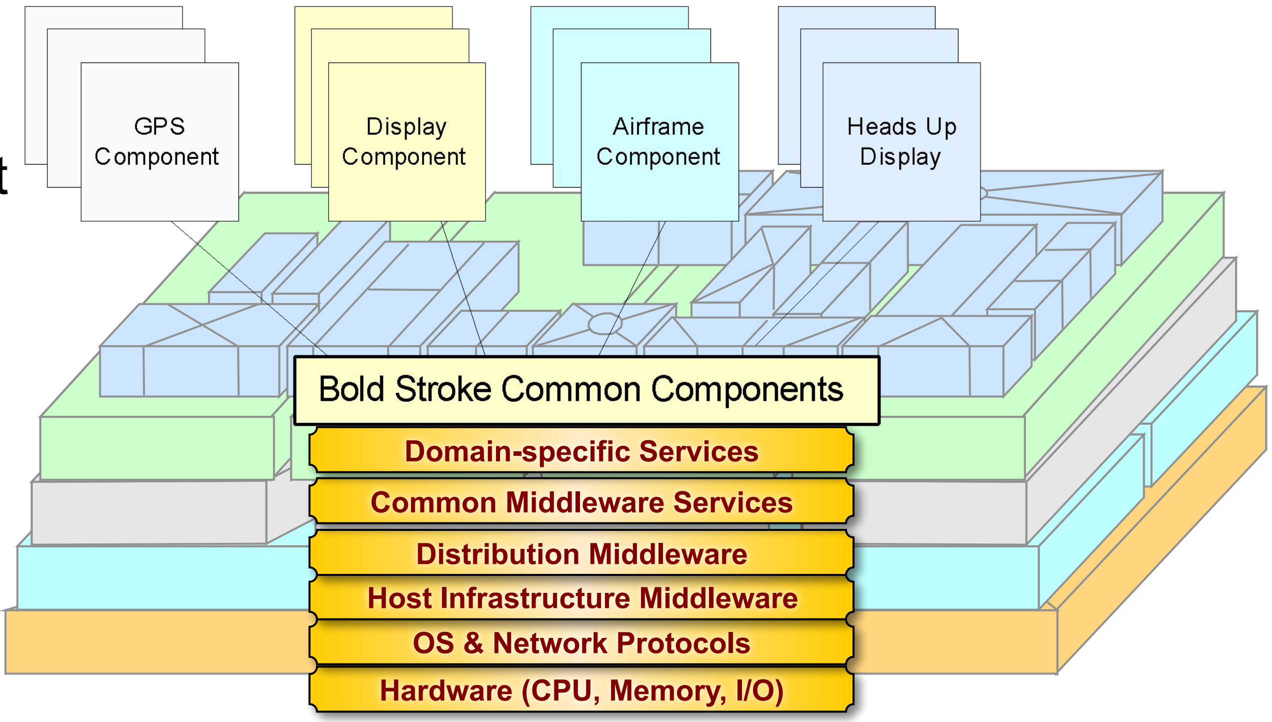
- e.g., applying SCV to Bold Stroke
  - **Scope:** Bold Stroke component architecture, object-oriented application frameworks, & associated components, e.g., GPS, Airframe, & Display



James Coplien et al. Commonality & Variability in Software Engineering, IEEE Software 1998

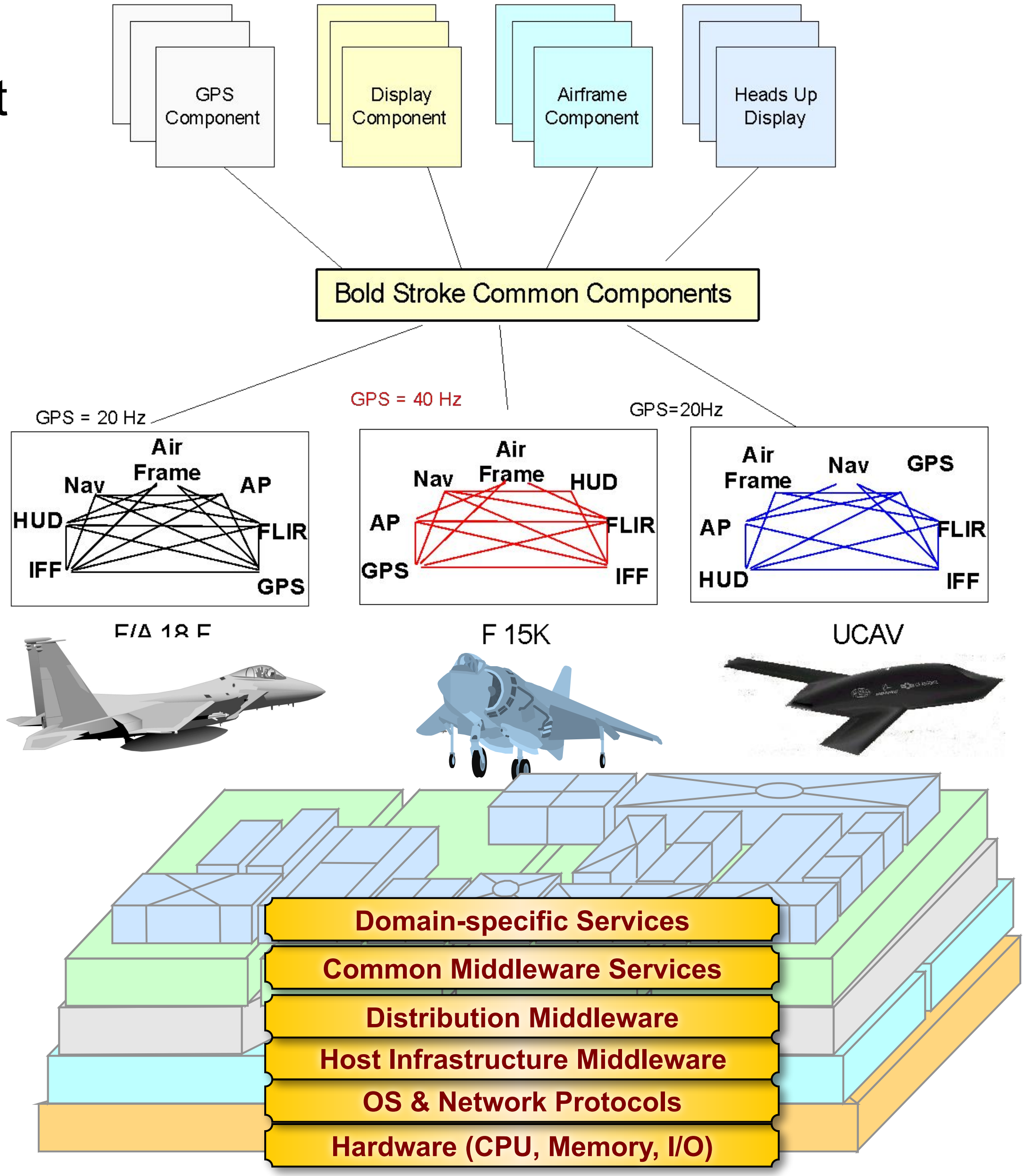
Applying SCV to Bold Stroke PLA

- **Commonalities** describe the attributes that are common across all members of the family
  - Common object-oriented frameworks & set of component types
    - e.g., GPS, Airframe, Navigation, & Display components
  - Common middleware infrastructure
    - e.g., Real-time CORBA & a variant of Lightweight CORBA Component Model (CCM) called Prism



Applying SCV to Bold Stroke PLA

- **Variabilities** describe the attributes unique to the different members of the family
  - Product-dependent component implementations (GPS/INS)
  - Product-dependent component connections
  - Product-dependent component assemblies (e.g., different weapons systems for security concerns)
  - Different hardware, OS, & network/bus configurations

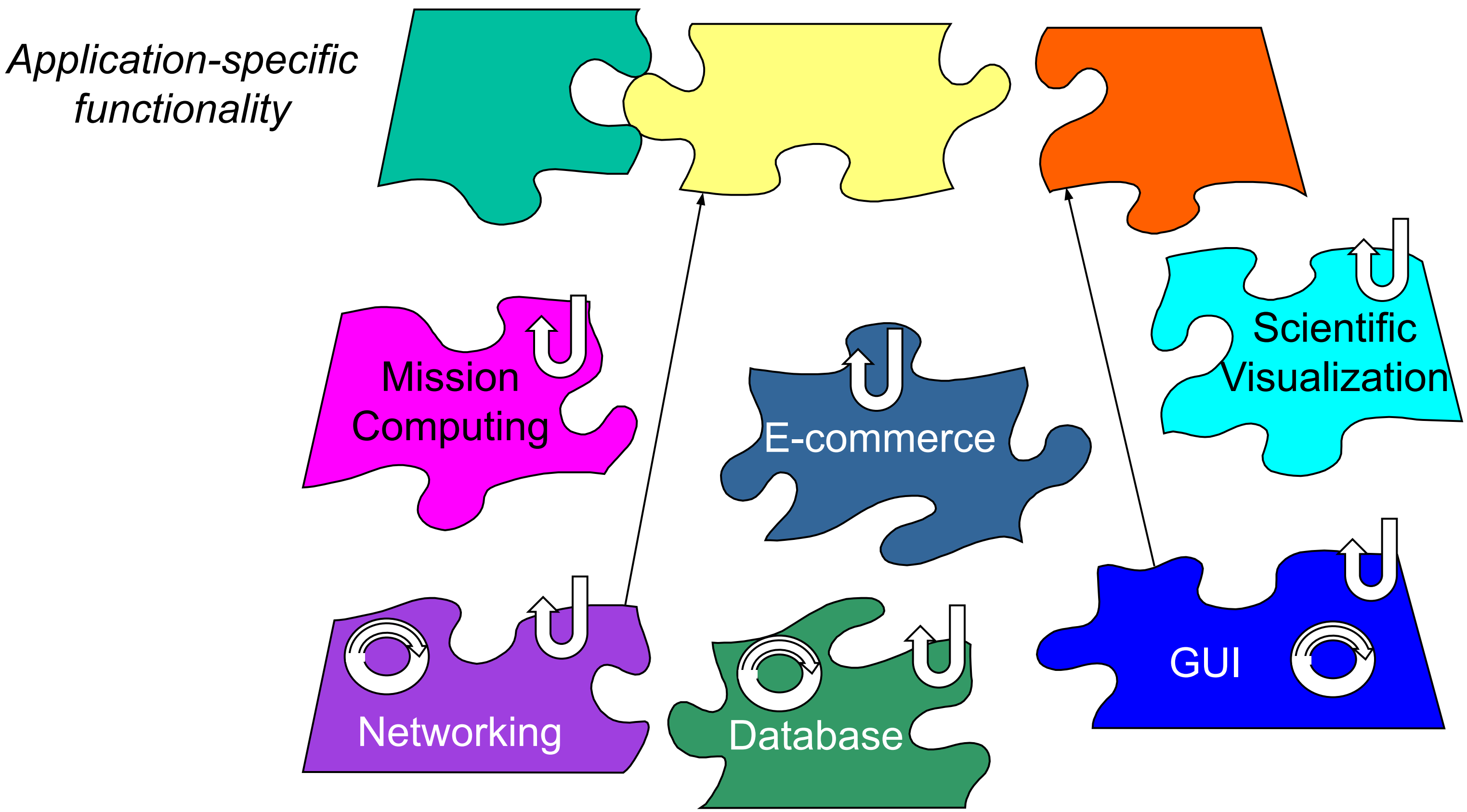


Frameworks are essential for developing PLAs

Overview of Frameworks

Framework Characteristics

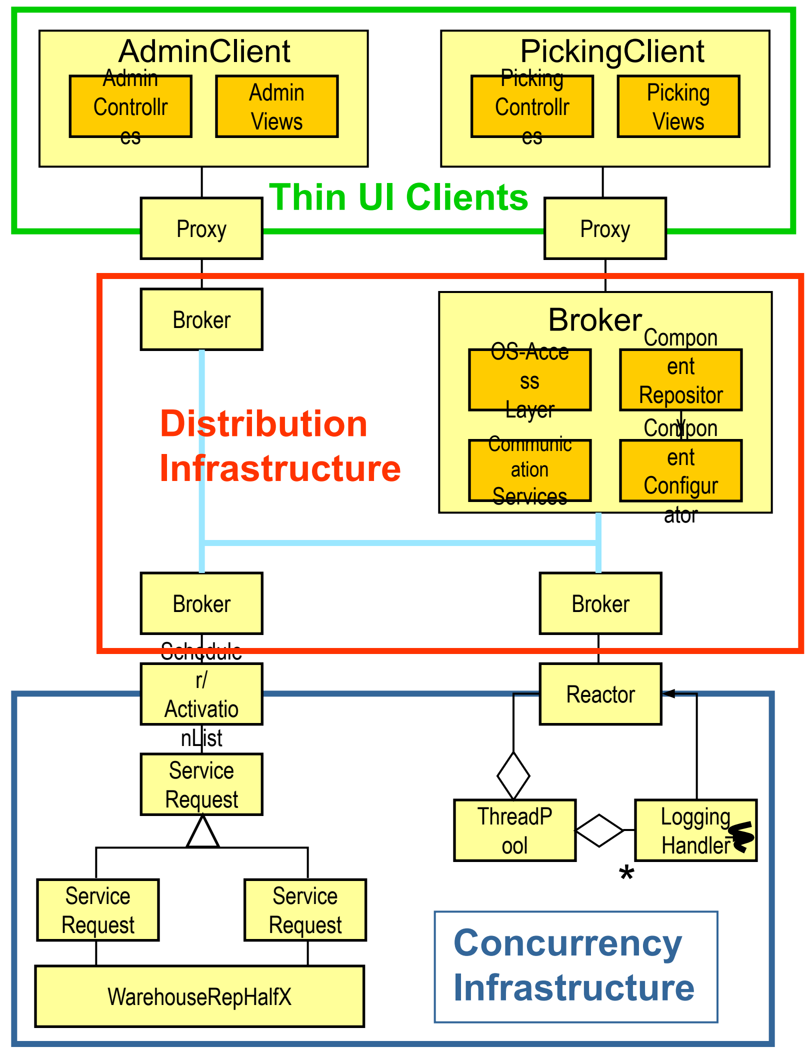
- Frameworks exhibit "inversion of control" at runtime via callbacks
- Frameworks provide integrated domain-specific structures & functionality
- Frameworks are "semi-complete" applications



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

Benefits of Frameworks

- **Design reuse**
  - e.g., by guiding application developers through the steps necessary to ensure successful creation & deployment of software



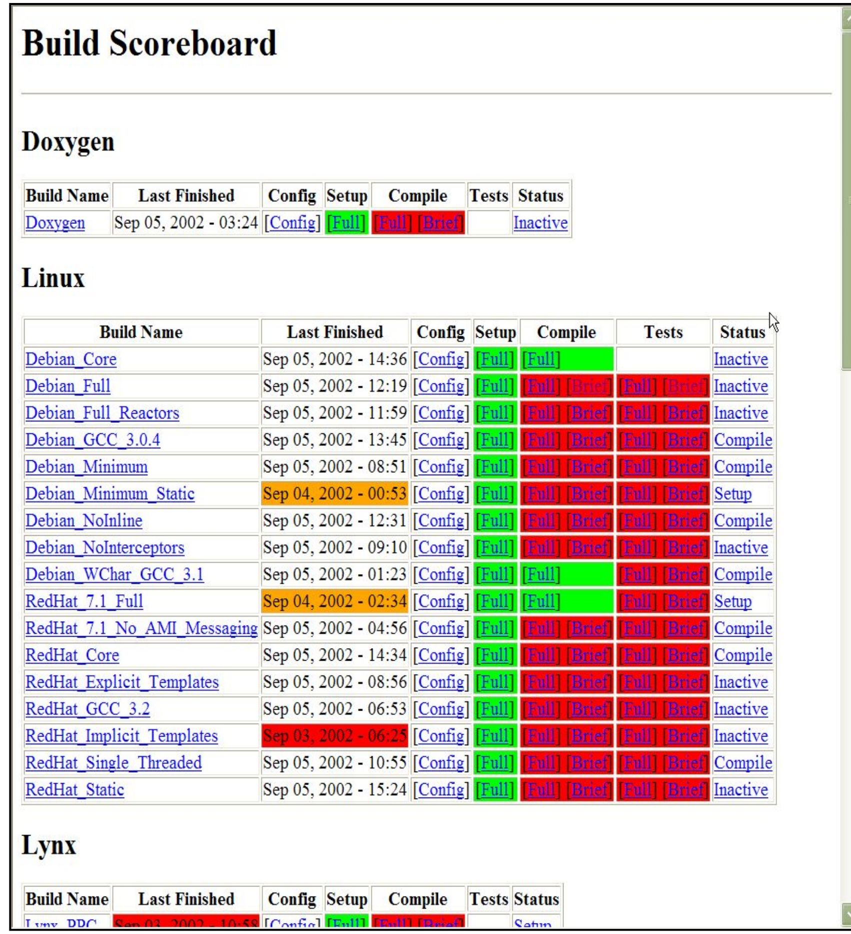
Benefits of Frameworks

- **Design reuse**
  - e.g., by guiding application developers through the steps necessary to ensure successful creation & deployment of software
- **Implementation reuse**
  - e.g., by amortizing software lifecycle costs & leveraging previous development & optimization efforts

```
ACE_Reactor
-----------------------------------------------------------------------
# reactor_ : ACE_Reactor *
# implementation_ : ACE_Reactor_Impl *
-----------------------------------------------------------------------
+ ACE_Reactor (implementation : ACE_Reactor_Impl * = 0,
               delete_implementation : int = 0)
+ open (max_handles : int, restart : int = 0,
        sig_handler : ACE_Sig_Handler * = 0,
        timer_queue : ACE_Timer_Queue * = 0) : int
+ close () : int
+ register_handler (handler : ACE_Event_Handler *,
                    mask : ACE_Reactor_Mask) : int
+ register_handler (io : ACE_HANDLE, handler : ACE_Event_Handler *,
                    mask : ACE_Reactor_Mask) : int
+ remove_handler (handler : ACE_Event_Handler *,
                  mask : ACE_Reactor_Mask) : int
+ remove_handler (io : ACE_HANDLE, mask : ACE_Reactor_Mask) : int
+ remove_handler (hs : const ACE_Handle_Set&, m : ACE_Reactor_Mask) : int
+ suspend_handler (handler : ACE_Event_Handler *) : int
+ resume_handler (handler : ACE_Event_Handler *) : int
+ mask_ops (handler : ACE_Event_handler *,
            mask : ACE_Reactor_Mask, ops : int) : int
+ schedule_wakeup (handler : ACE_Event_Handler *,
                   masks_to_be_added : ACE_Reactor_Mask ) : int
+ cancel_wakeup (handler : ACE_Event_Handler *,
                 masks_to_be_cleared : ACE_Reactor_Mask) : int
+ handle_events (max_wait_time : ACE_Time_Value * = 0) : int
+ run_reactor_event_loop (event_hook : int (*)(void *) = 0) : int
+ end_reactor_event_loop () : int
+ reactor_event_loop_done () : int
+ schedule_timer (handler : ACE_Event_Handler *, arg : void *,
                  delay : ACE_Time_Value &,
                  repeat : ACE_Time_Value & = ACE_Time_Value::zero) : int
+ cancel_timer (handler : ACE_Event_Handler *,
                dont_call_handle_close : int = 1) : int
+ cancel_timer (timer_id : long, arg : void ** = 0,
                dont_call_handle_close : int = 1) : int
+ notify (handler : ACE_Event_Handler * = 0,
          mask : ACE_Reactor_Mask = ACE_Event_Handler::EXCEPT_MASK,
          timeout : ACE_Time_Value * = 0) : int
+ max_notify_iterations (iterations : int) : int
+ purge_pending_notifications (handler : ACE_Event_Handler *,
                    mask : ACE_Reactor_Mask = ALL_EVENTS_MASK) : int
+ instance () : ACE_Reactor *
+ owner (new_owner : ACE_thread_t, old_owner : ACE_thread_t * = 0) : int
```

Benefits of Frameworks

- Design reuse
  - e.g., by guiding application developers through the steps necessary to ensure successful creation & deployment of software
- Implementation reuse
  - e.g., by amortizing software lifecycle costs & leveraging previous development & optimization efforts
- **Validation reuse**
  - e.g., by amortizing the efforts of validating application- & platform-independent portions of software, thereby enhancing software reliability & scalability



Build Scoreboard

Doxygen

Build Name	Last Finished	Config	Setup	Compile	Tests	Status
Doxygen	Sep 05, 2002 - 03:24	[Config]	[Full]	[Full] [Brief]		Inactive

Linux

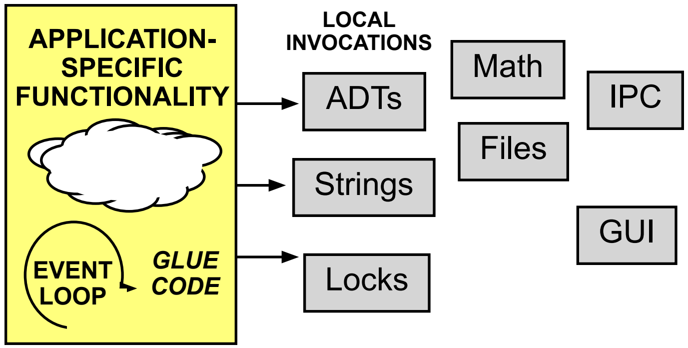
Build Name	Last Finished	Config	Setup	Compile	Tests	Status
Debian_Core	Sep 05, 2002 - 14:36	[Config]	[Full]	[Full]		Inactive
Debian_Full	Sep 05, 2002 - 12:19	[Config]	[Full]	[Full] [Brief]	[Full] [Brief]	Inactive
Debian_Full_Reactors	Sep 05, 2002 - 11:59	[Config]	[Full]	[Full] [Brief]	[Full] [Brief]	Inactive
Debian_GCC_3.0.4	Sep 05, 2002 - 13:45	[Config]	[Full]	[Full] [Brief]	[Full] [Brief]	Compile
Debian_Minimum	Sep 05, 2002 - 08:51	[Config]	[Full]	[Full] [Brief]	[Full] [Brief]	Compile
Debian_Minimum_Static	Sep 04, 2002 - 00:53	[Config]	[Full]	[Full] [Brief]	[Full] [Brief]	Setup
Debian_NoInline	Sep 05, 2002 - 12:31	[Config]	[Full]	[Full] [Brief]	[Full] [Brief]	Compile
Debian_NoInterceptors	Sep 05, 2002 - 09:10	[Config]	[Full]	[Full] [Brief]	[Full] [Brief]	Inactive
Debian_WChar_GCC_3.1	Sep 05, 2002 - 01:23	[Config]	[Full]	[Full]	[Full] [Brief]	Compile
RedHat_7.1_Full	Sep 04, 2002 - 02:34	[Config]	[Full]	[Full]	[Full] [Brief]	Setup
RedHat_7.1_No_AMI_Messaging	Sep 05, 2002 - 04:56	[Config]	[Full]	[Full] [Brief]	[Full] [Brief]	Compile
RedHat_Core	Sep 05, 2002 - 14:34	[Config]	[Full]	[Full] [Brief]	[Full] [Brief]	Compile
RedHat_Explicit_Templates	Sep 05, 2002 - 08:56	[Config]	[Full]	[Full] [Brief]	[Full] [Brief]	Inactive
RedHat_GCC_3.2	Sep 05, 2002 - 06:53	[Config]	[Full]	[Full] [Brief]	[Full] [Brief]	Inactive
RedHat_Implicit_Templates	Sep 03, 2002 - 06:25	[Config]	[Full]	[Full] [Brief]	[Full] [Brief]	Inactive
RedHat_Single_Threaded	Sep 05, 2002 - 10:55	[Config]	[Full]	[Full] [Brief]	[Full] [Brief]	Compile
RedHat_Static	Sep 05, 2002 - 15:24	[Config]	[Full]	[Full] [Brief]	[Full] [Brief]	Inactive

Lynx

Build Name	Last Finished	Config	Setup	Compile	Tests	Status

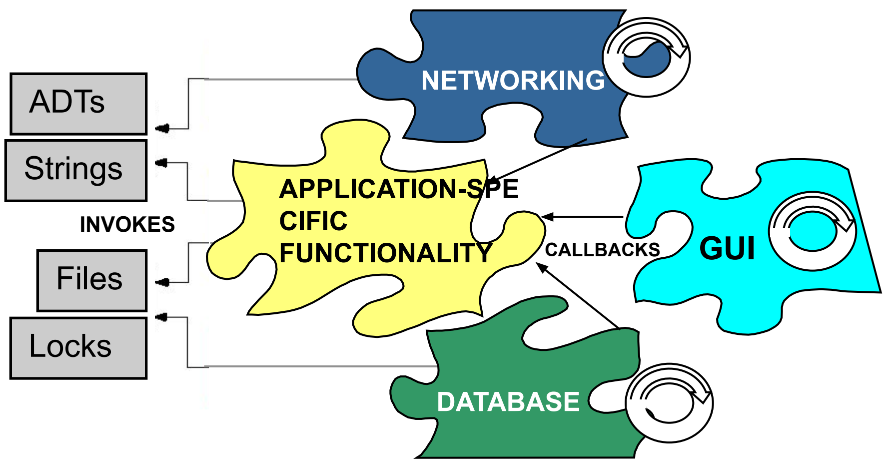
www.dre.vanderbilt.edu/scoreboard

Comparing Reuse Techniques



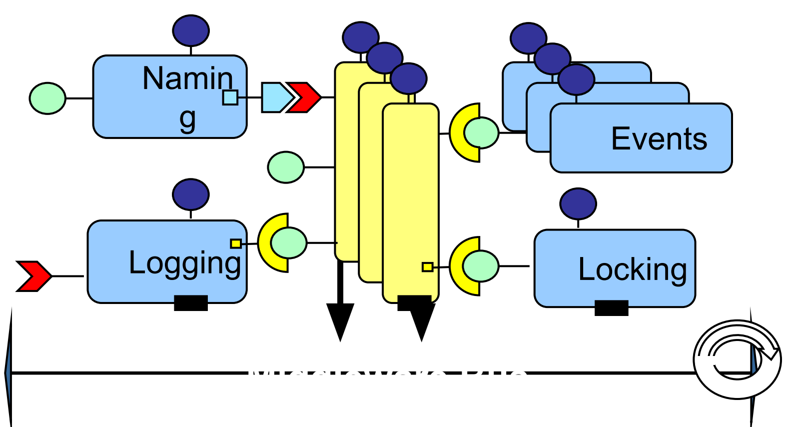
Class Library Architecture

- A ***class*** is a unit of abstraction & implementation in an OO programming language, i.e., a reusable ***type*** that often implements ***patterns***
- Classes in class libraries are typically *passive*



Framework Architecture

- A ***framework*** is an integrated set of classes that collaborate to produce a reusable architecture for a family of applications
- Frameworks implement ***pattern languages***



Component Architecture

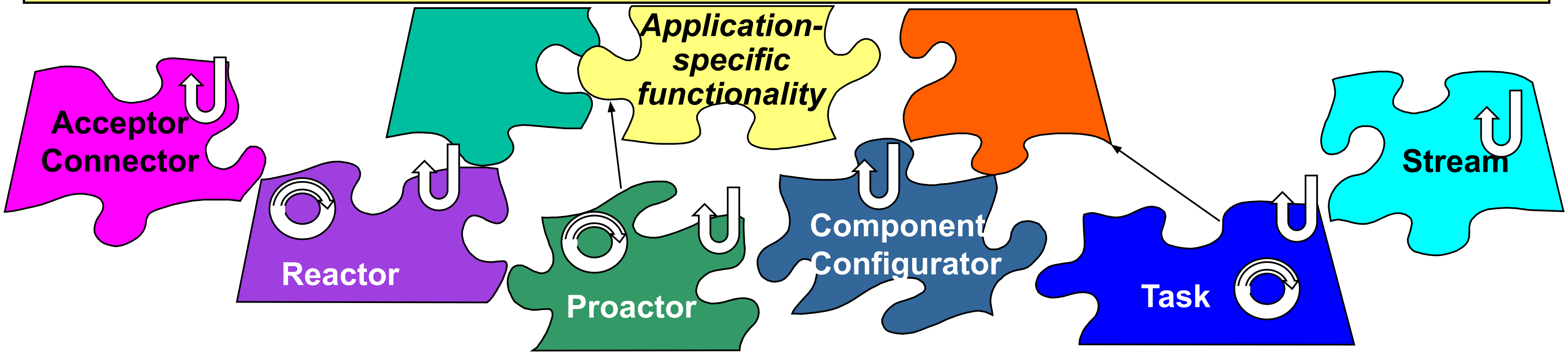
- A ***component*** is an encapsulation unit with one or more interfaces that provide clients with access to its services
- Components can be deployed & configured via ***assemblies***

Taxonomy of Reuse Techniques

Class Libraries	Frameworks	Components
Micro-level	Meso-level	Macro-level
Stand-alone language entities	"Semi-complete" applications	Stand-alone composition entities
Domain-independent	Domain-specific	Domain-specific or Domain-independent
Borrow caller's thread	Inversion of control	Borrow caller's thread

The Frameworks in ACE

ACE frameworks are a product-line architecture for domain of network applications

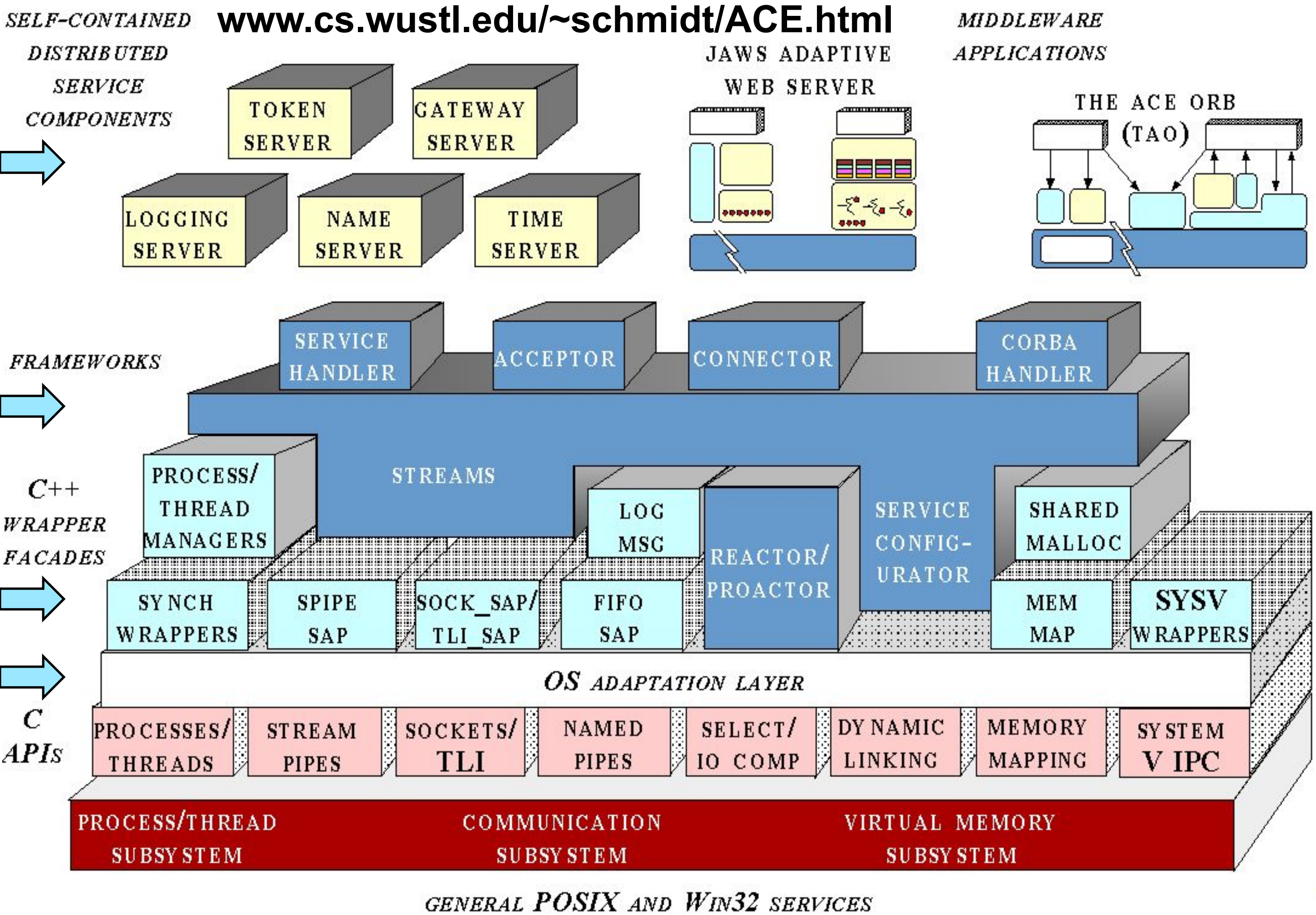


ACE Framework	Inversion of Control & Hook Methods
Reactor & Proactor	Calls back to application-supplied event handlers to perform processing when events occur synchronously & asynchronously
Service Configurator	Calls back to application-supplied service objects to initialize, suspend, resume, & finalize them
Task	Calls back to an application-supplied hook method to perform processing in one or more threads of control
Acceptor/Connector	Calls back to service handlers to initialize them after they are connected
Streams	Calls back to initialize & finalize tasks when they are pushed & popped from a stream

Commonality & Variability in ACE Frameworks

Framework	Commonality	Variability
Reactor	• Time & timer interface • Synchronous initiation event handling interface	• Time & timer implementation • Synchronous event detection, demuxing, & dispatching implementation
Proactor	• Asynchronous completion event handling interface	• Asynchronous operation & completion event handler demuxing & dispatching implementation
Service Configurator	• Methods for controlling service lifecycle • Scripting language for interpreting service directives	• Number, type/implementation, & order of service configuration • Dynamical linking/unlinking implementation
Task	• Intra-process message queueing & processing • Concurrency models	• Strategized message memory management & synchronization • Thread implementations
Acceptor/ Connector	• Synchronous/asynchronous & active/passive connection establishment & service handler initialization	• Communication protocols • Type of service handler • Service handler creation, accept/connect, & activation logic
Streams	• Layered service composition • Message-passing • Leverages Task commonality	• Number, type, & order of services composed • Concurrency model

The Layered Architecture of ACE



Features

- Open-source
- 200,000+ lines of C++
- 40+ person-years of effort
- Ported to many OS platforms

•Large open-source user community

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

•Commercial support by Riverace

- www.riverace.com/

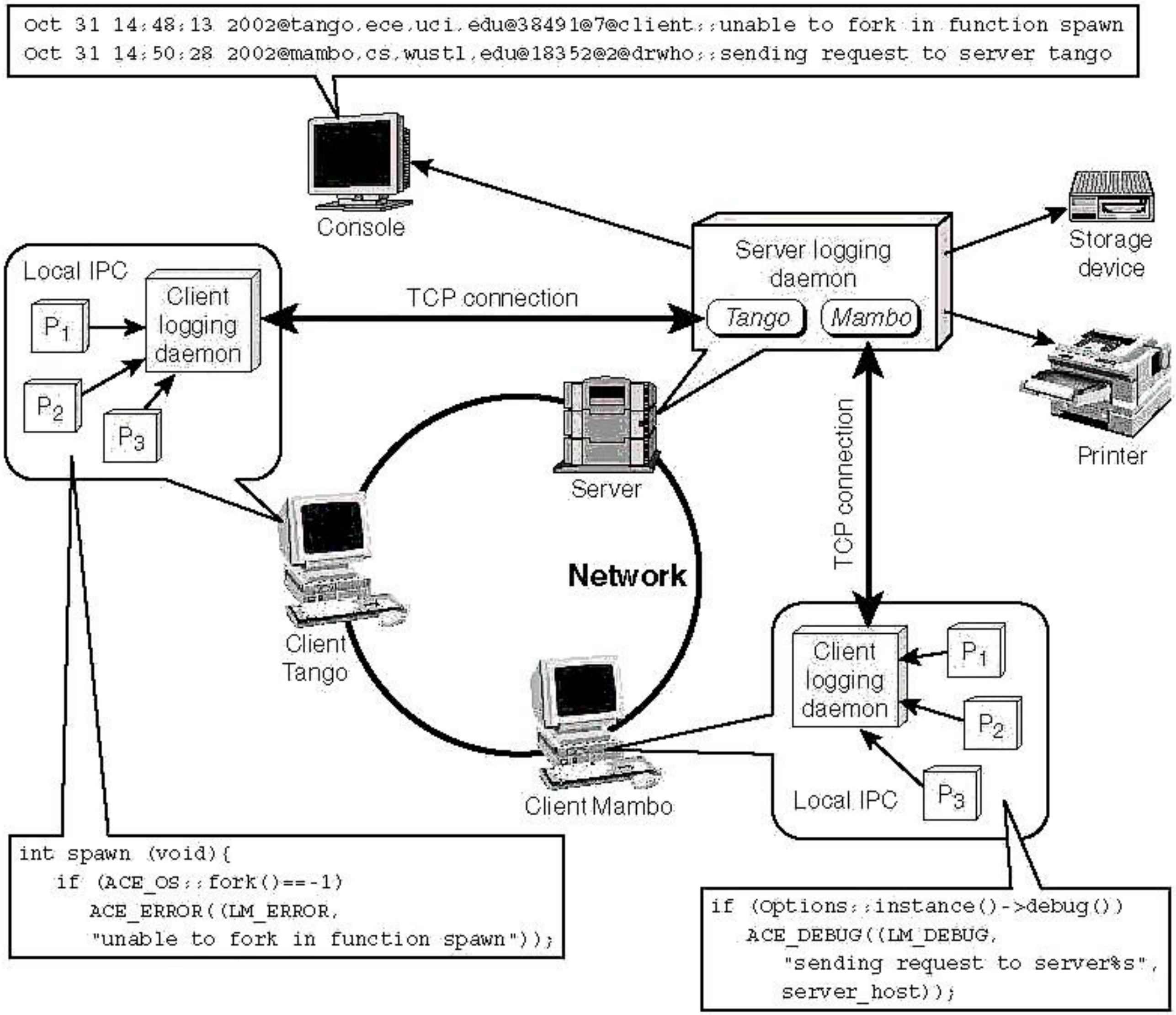
Networked Logging Service Example

Key Participants

- ***Client application processes***
  - Generate log records
- ***Client logging daemons***
  - Buffer log records & transmit them to the server logging daemon
- ***Server logging daemon***
  - Receive, process, & store log records

C++ code for all logging service examples are in

- ACE_ROOT/examples/C++NPv1/
- ACE_ROOT/examples/C++NPv2/



Patterns in the Networked Logging Service

Half-Sync/ Half-Async

Leader/ Followers

Monitor Object

Active Object

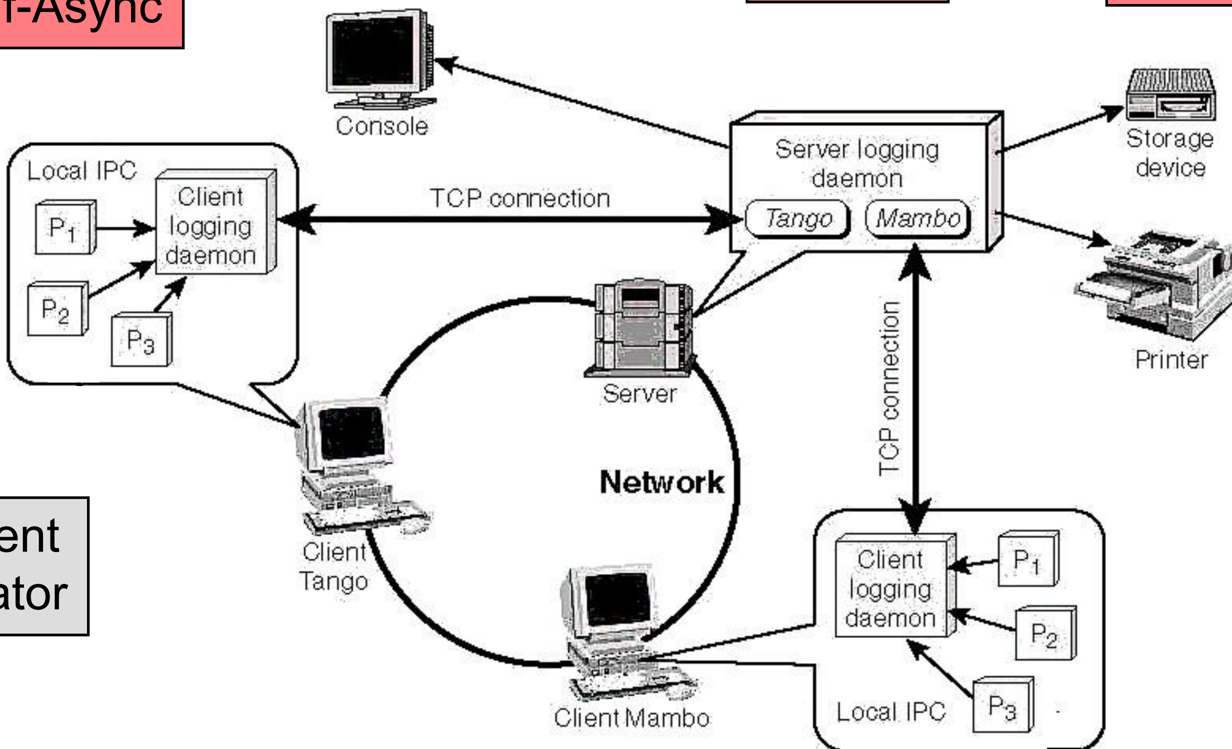
Reactor

Pipes & Filters

Acceptor-Connector

Component Configurator

Proactor



Wrapper Facade

Strategized Locking

Scoped Locking

Thread-safe Interface

Service/Server Design Dimensions

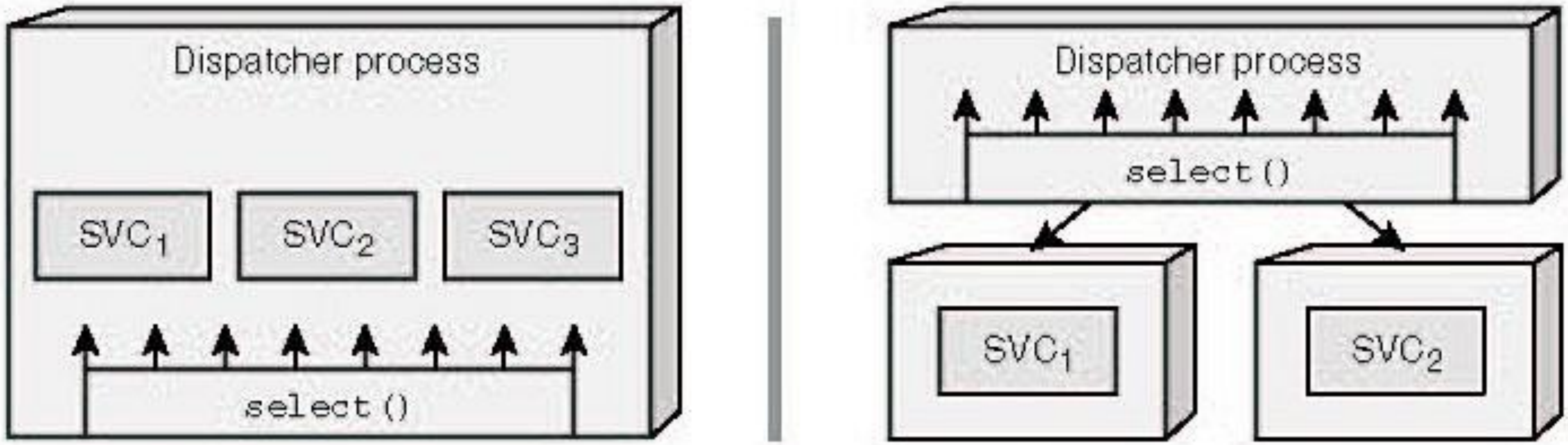
- When designing networked applications, it's important to recognize the difference between a service, which is a capability offered to clients, & a server, which is the mechanism by which the service is offered
- The design decisions regarding services & servers are easily confused, but should be considered separately
- This section covers the following service & server design dimensions:
  - Short- versus long-duration services
  - Internal versus external services
  - Stateful versus stateless services
  - Layered/modular versus monolithic services
  - Single- versus multiservice servers
  - One-shot versus standing servers

Short- versus Long-duration Services

- **Short-duration services** execute in brief, often fixed, amounts of time & usually handle a single request at a time
- Examples include
  - Computing the current time of day
  - Resolving the Ethernet number of an IP address
  - Retrieving a disk block from the cache of a network file server
- To minimize the amount of time spent setting up a connection, short-duration services are often implemented using connectionless protocols
  - e.g., UDP/IP

- **Long-duration services** run for extended, often variable, lengths of time & may handle numerous requests during their lifetime
- Examples include
  - Transferring large software releases via FTP
  - Downloading MP3 files from a Web server using HTTP
  - Streaming audio & video from a server using RTSP
  - Accessing host resources remotely via TELNET
  - Performing remote file system backups over a network
- Services that run for longer durations allow more flexibility in protocol selection. For example, to improve efficiency & reliability, these services are often implemented with connection-oriented protocols
  - e.g., TCP/IP or session-oriented protocols, such as RTSP or SCTP

Internal vs. External Services



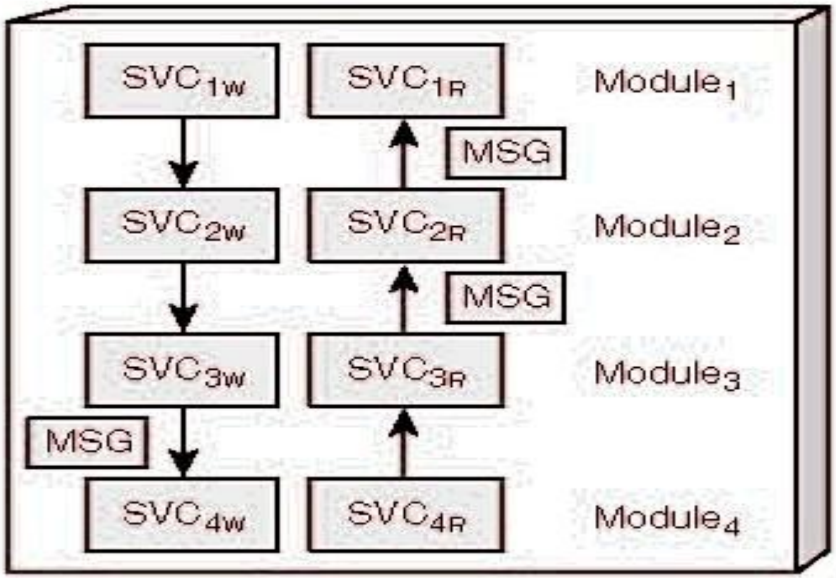
(1) Internal services

(2) External services

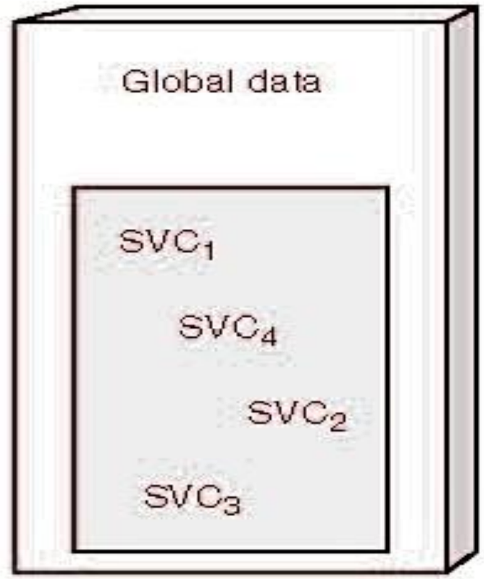
- Internal services execute in the same address space as the server that receives the request
- Communication & synchronization between internal services can be very efficient
- Rogue services can cause problems for other services, however

- External services execute in different process address spaces
- They are generally more robust than internal services since they are isolated from each other
- IPC & synchronization overhead is higher, however

Monolithic vs. Layered/Modular Services



(1) Layered/modular services

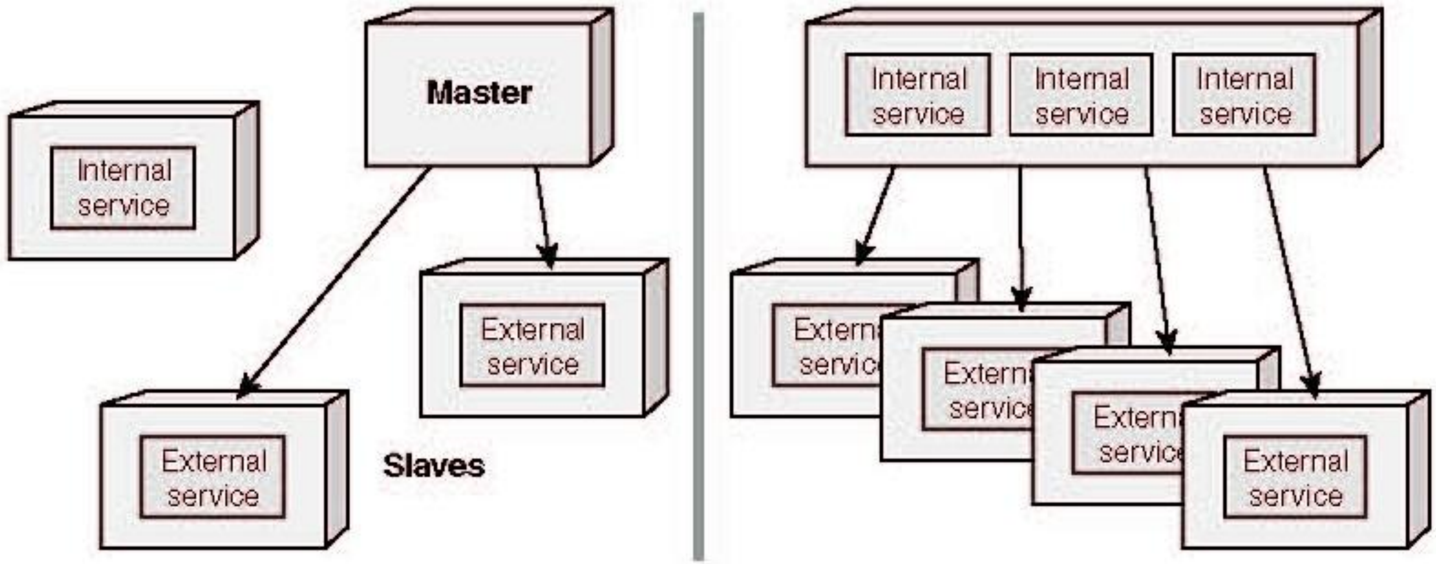


(2) Monolithic services

- Layered/modular services can be decomposed into a series of partitioned & hierarchically related tasks
- They are generally easier to understand, evolve, & maintain
- Performance can be a problem, however

- Monolithic services are tightly coupled clumps of functionality that aren't organized hierarchically
- They are harder to understand, evolve, & maintain
- They may be more efficient, however

Single Service vs. Multiservice Servers



(1) Single-service servers

(2) Multiservice server

- Single-service servers offer only one service
- Deficiencies include:
  - Consuming excessive OS resources
  - Redundant infrastructure code
  - Manual shutdown & restart
  - Inconsistent administration

- Multiservice servers address the limitations with single-service servers by integrating a collection of single-service servers into a single administrative unit
- Master server spawns external services on-demand
- Benefits are the inverse of single-service server deficiencies

Sidebar: Comparing Multiservice Server Frameworks

UNIX **`INETD`**

- Internal services, such as **`ECHO`** & **`DAYTIME`**, are fixed at static link time
- External services, such as **`FTP`** & **`TELNET`**, can be dynamically reconfigured via sending a **`SIGHUP`** signal to the daemon & performing **`socket/bind/listen`** calls on all services listed in the **`inetd.conf`** file
- Since internal services cannot be reconfigured, any new listing of such services must occur via **`fork()`** & **`exec*()`** family of system calls

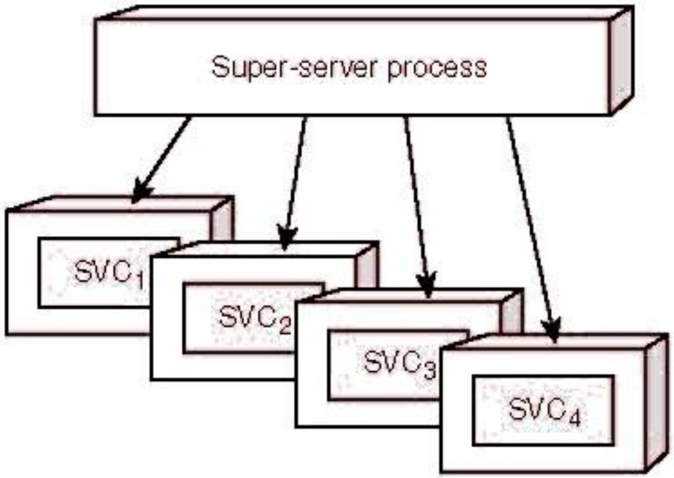
System V UNIX **`LISTEN`** port monitoring

- Like **`INETD`**
- Supports only external services via **`TLI`** & System V **`STREAMS`**
- Supports standing servers by passing initialized file descriptors via **`STREAMS`** pipes from the **`LISTEN`**

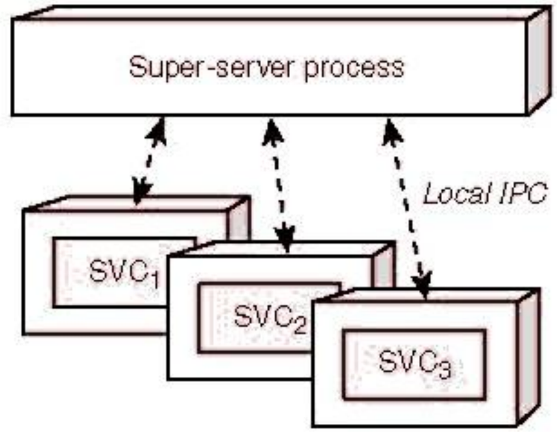
Windows Service Control Manager (**`SCM`**)

- More than just a port monitoring facility
- Uses RPC-based interface to initiate & control administrator-installed services that typically run as separate threads within either a single service or a multiservice daemon process

One-shot vs. Standing Servers



(1) One-shot server



(2) Standing server

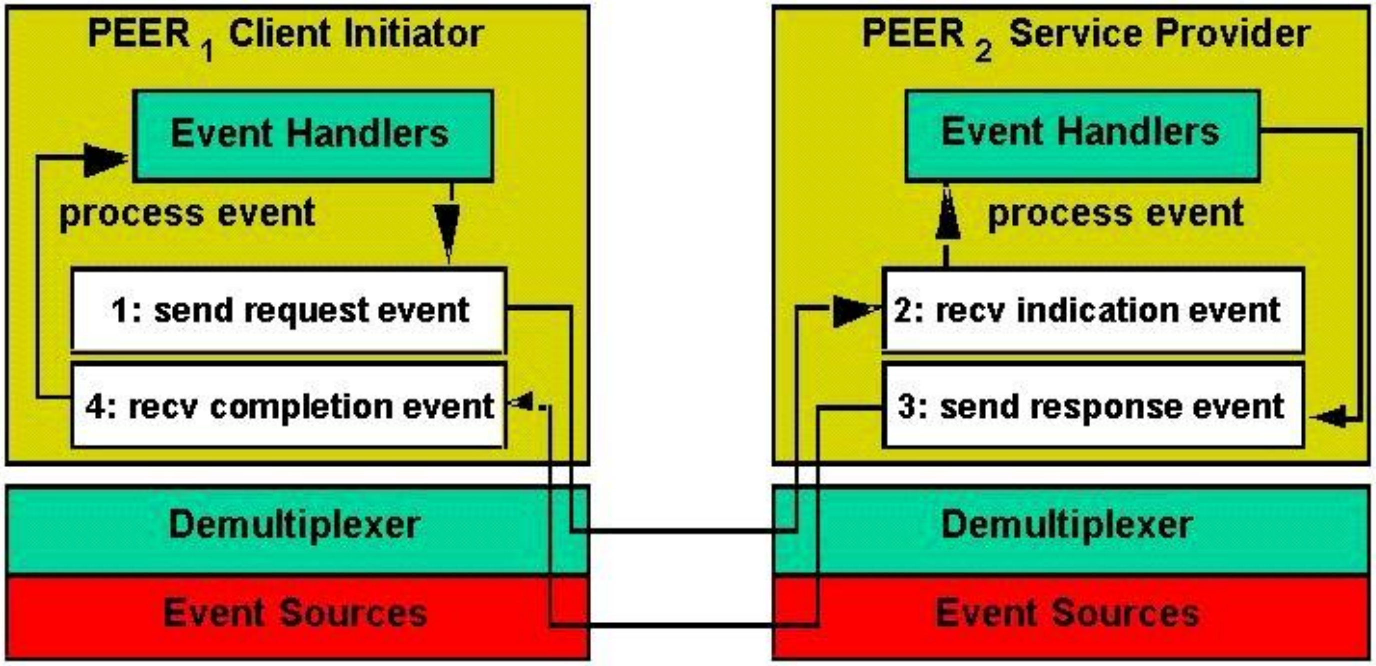
- One-shot servers are spawned on demand, e.g., by an **`inetd`** superserver
- They perform service requests in a separate thread or process
- A one-shot server terminates after the completion of the request or session that triggered its creation
- Primary benefit is lower resource utilization
- Primary drawback is startup latency

- Standing servers continue to run beyond the lifetime of any particular service request or session they process
- Standing servers are often initiated at boot time or by a superserver after the first client request
- Primary benefit is amortized startup latency
- Primary drawback is higher resource utilization

The ACE Reactor Framework

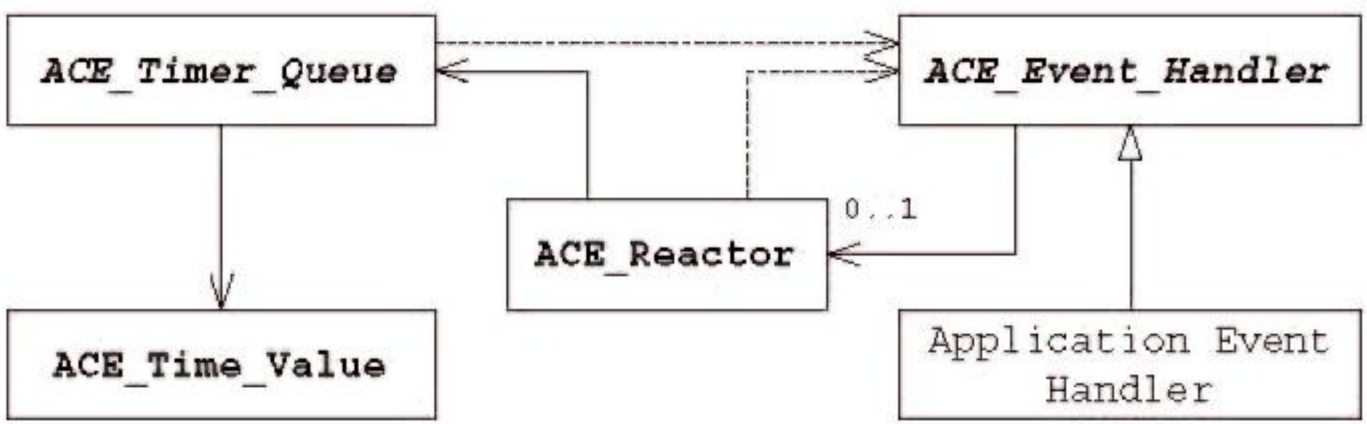
Motivation

- Many networked applications are developed as event-driven programs
- Common sources of events in these applications include activity on an IPC stream for I/O operations, POSIX signals, Windows handle signaling, & timer expirations
- To improve extensibility & flexibility, it's important to decouple the detection, demultiplexing, & dispatching of events from the handling of events



The ACE Reactor Framework

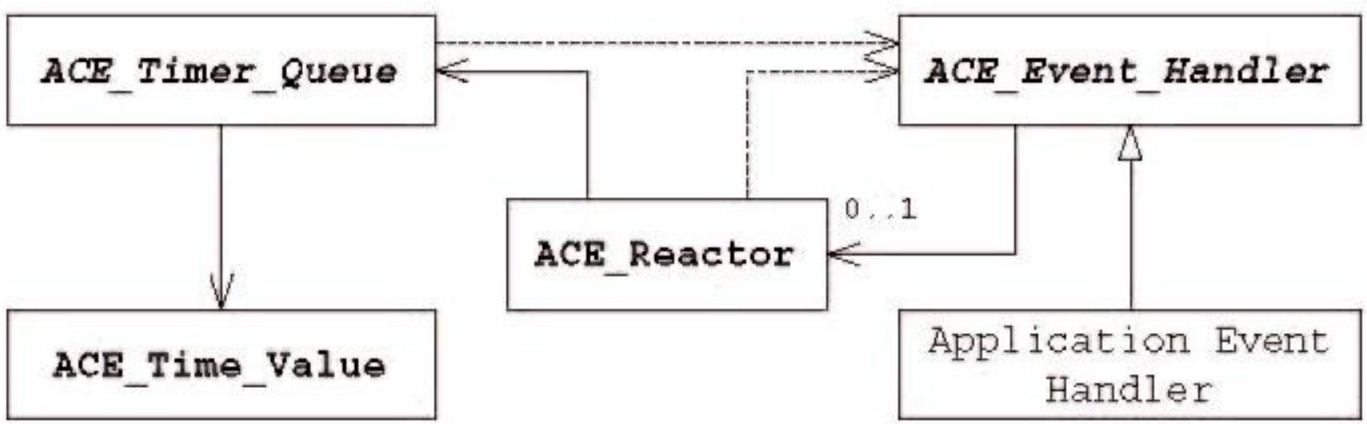
- The ACE Reactor framework implements the Reactor pattern (POSA2)
- This pattern & framework automates the
  - Detection of events from various sources of events
  - Demultiplexing the events to pre-registered handlers of these events
  - Dispatching to hook methods defined by the handlers to process the events in an application-defined manner



The ACE Reactor Framework

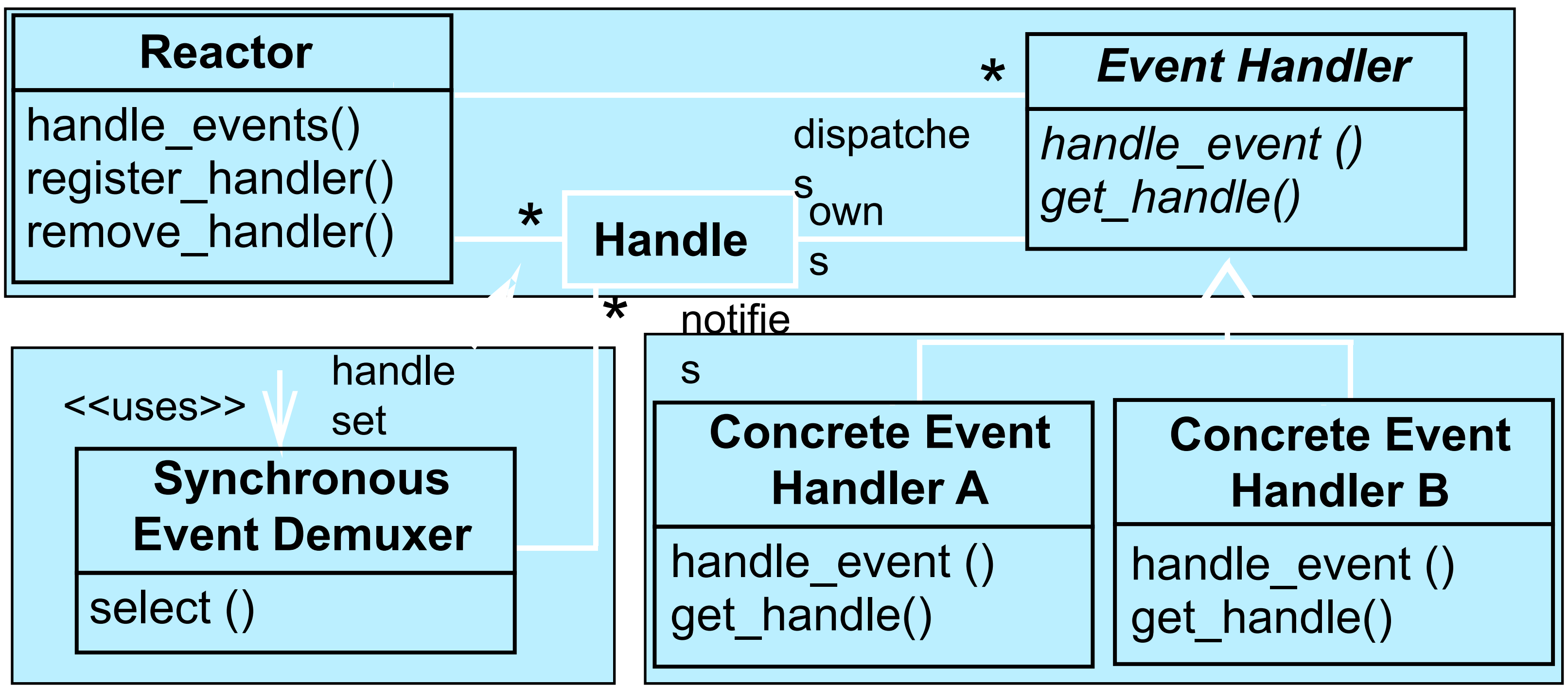
ACE Class	Description
ACE_Time_Value	Provides a portable, normalized representation of time and duration that uses C++ operator overloading to simplify time-related arithmetic and relational operations.
ACE_Event_Handler	An abstract class whose interface defines the hook methods that are the target of ACE_Reactor callbacks. Most application event handlers developed with ACE are descendants of ACE_Event_Handler.
ACE_Timer_Queue	An abstract class defining the capabilities and interface for a timer queue. ACE contains a variety of classes derived from ACE_Timer_Queue that provide flexible support for different timing requirements.
ACE_Reactor	Provides the interface for managing event handler registrations and executing the event loop that drives event detection, demultiplexing, and dispatching in the Reactor framework.

- The classes in the ACE Reactor framework implement the Reactor pattern:



The Reactor Pattern Participants

- The *Reactor* architectural pattern allows event-driven applications to demultiplex & dispatch service requests that are delivered to an application from one or more clients



The Reactor Pattern Dynamics

: Main Program

: Concrete Event Handler

: Reactor

: Synchronous Even Demultiplexer

Con. Event Handler

Events

register_handler()

get_handle()

Handle

handle_events()

Handles

select()

event

handle_event()

Handles

service()

Observations

- Note inversion of control
- Also note how long-running event handlers can degrade the QoS since callbacks steal the reactor's thread!

Pros & Cons of the Reactor Pattern

This pattern offers four **benefits**:

- ***Separation of concerns***
  - This pattern decouples application-independent demuxing & dispatching mechanisms from application-specific hook method functionality
- ***Modularity, reusability, & configurability***
  - This pattern separates event-driven application functionality into several components, which enables the configuration of event handler components that are loosely integrated via a reactor
- ***Portability***
  - By decoupling the reactor's interface from the lower-level OS synchronous event demuxing functions used in its implementation, the Reactor pattern improves portability
- ***Coarse-grained concurrency control***
  - This pattern serializes the invocation of event handlers at the level of event demuxing & dispatching within an application process or thread

This pattern can incur **liabilities*:***

- ***Restricted applicability***
  - This pattern can be applied efficiently only if the OS supports synchronous event demuxing on handle sets
- ***Non-pre-emptive***
  - In a single-threaded application, concrete event handlers that borrow the thread of their reactor can run to completion & prevent the reactor from dispatching other event handlers
- ***Complexity of debugging & testing***
  - It is hard to debug applications structured using this pattern due to its inverted flow of control, which oscillates between the framework infrastructure & the method call-backs on application-specific event handlers

The ACE_Time_Value Class (1/2)

Motivation

- Many types of applications need to represent & manipulate time values

- Different date & time representations are used on OS platforms, such as POSIX, Windows, & proprietary real-time systems
- The **`ACE_Time_Value`** class encapsulates these differences within a portable wrapper facade

The ACE_Time_Value Class (2/2)

Class Capabilities

- This class applies the Wrapper Façade pattern & C++ operator overloading to simplify portable time & duration related operations with the following capabilities:
  - It provides a standardized representation of time that's portable across OS platforms
  - It can convert between different platform time representations
  - It uses operator overloading to simplify time-based comparisons by permitting standard C++ syntax for time-based arithmetic & relational expressions
  - Its constructors & methods normalize time quantities
  - It can represent either a duration or an absolute date & time

The ACE_Time_Value Class API

```
ACE_Time_Value
--------------------------------------------------------------
+ zero : ACE_Time_Value
+ max_time : ACE_Time_Value
- tv_ : timeval
--------------------------------------------------------------
+ ACE_Time_Value (sec : long, usec : long = 0)
+ ACE_Time_Value (t : const struct timeval &)
+ ACE_Time_Value (t : const timespec_t &)
+ ACE_Time_Value (t : const FILETIME &)
+ set (sec : long, usec : long)
+ set (t : const struct timeval &)
+ set (t : const timespec_t &)
+ set (t : const FILETIME &)
+ sec () : long
+ usec () : long
+ msec () : long
+ operator+= (tv : const ACE_Time_Value &) : ACE_Time_Value &
+ operator-= (tv : const ACE_Time_Value &) : ACE_Time_Value &
+ operator*= (d : double) : ACE_Time_Value &
```

This class handles *variability* of time representation & manipulation across OS platforms via a *common* API

Sidebar: Relative vs. Absolute Timeouts

- ***Relative time semantics*** are often used in ACE when an operation used it just once, e.g.:
  - ACE IPC wrapper façade I/O methods as well as higher level frameworks, such as the ACE Acceptor & Connector
  - **`ACE_Reactor`** & **`ACE_Proactor`** event loop & timer scheduling
  - **`ACE_Process`**, **`ACE_Process_Manager`** & **`ACE_Thread_Manager`** **`wait()`** methods
  - **`ACE_Sched_Params`** for time slice quantum

- ***Absolute time semantics*** are often used in ACE when an operation may be run multiple times in a loop, e.g.:
  - ACE synchronizer wrapper facades, such as **`ACE_Thread_Semaphore`** & **`ACE_Condition_Thread_Mutex`**
  - **`ACE_Timer_Queue`** scheduling mechanisms
  - **`ACE_Task`** methods
  - **`ACE_Message_Queue`** methods & classes using them

Using the ACE_Time_Value Class (1/2)

- The following example creates two `ACE_Time_Value` objects whose values can be set via command-line arguments
- It then performs range checking to ensure the values are reasonable

```
#include "ace/OS.h"

const ACE_Time_Value max_interval (60 * 60); // 1 hour.

int main (int argc, char *argv[]) {
  ACE_Time_Value expiration = ACE_OS::gettimeofday ();
  ACE_Time_Value interval;

  ACE_Get_Opt opt (argc, argv, "e:i:"));
  for (int c; (c = opt ()) != -1;)
    switch (c) {
    'e': expiration += ACE_Time_Value (atoi (opt.opt_arg
()));
         break;
    'i': interval = ACE_Time_Value (atoi (opt.opt_arg ()));
         break;
    }
```

Using the ACE_Time_Value Class (2/2)

Note the use of relational operators

```
  if (interval > max_interval)
    cout << "interval must be less than "
         << max_interval.sec () << endl;
  else if (expiration > (ACE_Time_Value::max_time - interval))
    cout << "expiration + interval must be less than "
         << ACE_Time_Value::max_time.sec () << endl;
  return 0;
}
```

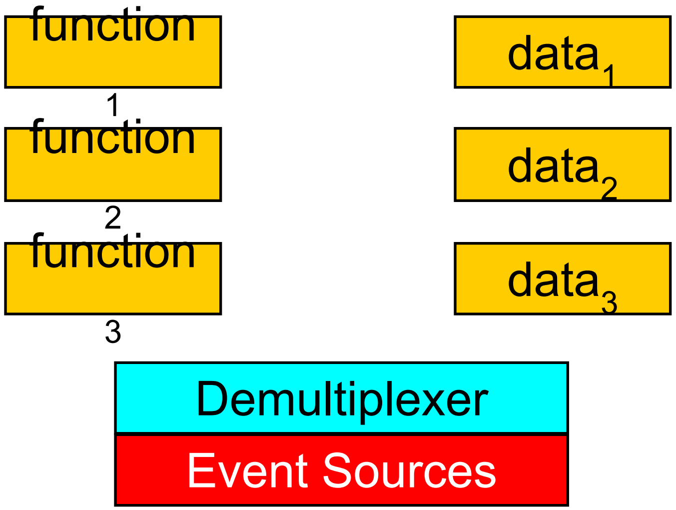
Sidebar: ACE_Get_Opt

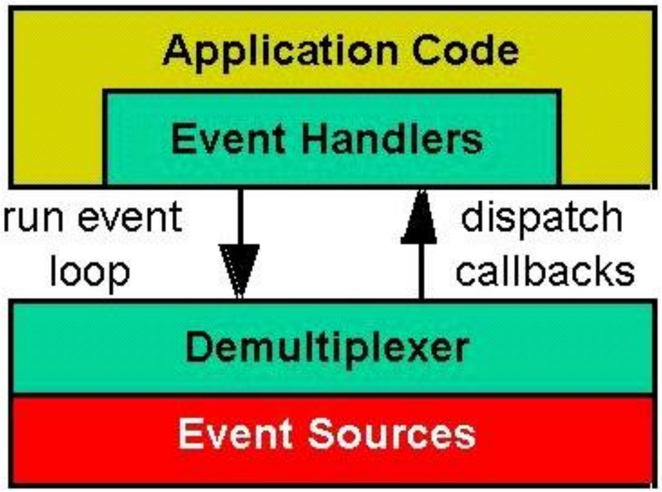
- **`ACE_Get_Opt`** is an iterator for parsing command line options that provides a wrapper façade for the POSIX **`getopt()`** function
  - Each instance of **`ACE_Get_Opt`** maintains its own state, so it can be used reentrantly
  - **`ACE_Get_Opt`** is easier to use than **`getopt()`** since the **`optstring`** & **`argc/argv`** arguments are only passed once to its constructor
  - It also supports "long options," which are more expressive than **`getopt()`**
- **`ACE_Get_Opt`** can be used to parse the **`argc/argv`** pair passed to **`main()`** or to the **`init()`** hook method used by the ACE Service Configurator framework

The ACE_Event_Handler Class (1/2)

Motivation

- Networked applications are often “event driven”
  - i.e., their processing is driven by callbacks
- There are problems with implementing callbacks by defining a separate function for each type of event





- It is therefore more effective to devise an “object-oriented” event demultiplexing mechanism
- This mechanism should implement callbacks via object-oriented event handlers

The ACE_Event_Handler Class (2/2)

Class Capabilities

- This base class of all reactive event handlers provides the following capabilities:
  - It defines hook methods for input, output, exception, timer, & signal events
  - Its hook methods allow applications to extend event handler subclasses in many ways without changing the framework
  - Its use of object-oriented callbacks simplifies the association of data with hook methods that manipulate the data
  - Its use of objects also automates the binding of an event source (or set of sources) with data the event source is associated with, such as a network session
  - It centralizes how event handlers can be destroyed when they're not needed
  - It holds a pointer to the **ACE_Reactor** that manages it, making it simple for an event handler to manage its event (de)registration correctly

The ACE_Event_Handler Class API

```
                         ACE_Event_Handler
--------------------------------------------------------------------
- priority_ : int
- reactor_ : ACE_Reactor *
--------------------------------------------------------------------
# ACE_Event_Handler (r : ACE_Reactor * = 0,
                     prio : int = LO_PRIORITY)
+ ~ACE_Event_Handler ()
+ handle_input (h : ACE_HANDLE = ACE_INVALID_HANDLE) : int
+ handle_output (h : ACE_HANDLE = ACE_INVALID_HANDLE) : int
+ handle_exception (h : ACE_HANDLE = ACE_INVALID_HANDLE) : int
+ handle_timeout (now : ACE_Time_Value &, act : void * = 0) : int
+ handle_signal (signum : int, info : siginfo_t * = 0,
                 ctx : ucontext_t * = 0) : int
+ handle_close (h : ACE_HANDLE, mask : ACE_Reactor_Mask) : int
+ get_handle () : ACE_HANDLE
+ reactor () : ACE_Reactor *
+ reactor (r : ACE_Reactor *)
+ priority () : int
+ priority (prio : int)
```

This class handles *variability* of event processing behavior via a *common* event handler API

Types of Events & Event Handler Hooks

- When an application registers an event handler with a reactor, it must indicate what type(s) of event(s) the event handler should process
- ACE designates these event types via enumerators defined in **`ACE_Event_Handler`** that are associated with **`handle_*()`** hook methods

Event Type	Description
`READ_MASK`	Indicates input events, such as data on a socket or file handle. A reactor dispatches the `handle_input()` hook method to process input events.
`WRITE_MASK`	Indicates output events, such as when flow control abates. A reactor dispatches the `handle_output()` hook method to process output events.
`EXCEPT_MASK`	Indicates exceptional events, such as urgent data on a socket. A reactor dispatches the `handle_exception()` hook method to process exceptional events.
`ACCEPT_MASK`	Indicates passive-mode connection events. A reactor dispatches the `handle_input()` hook method to process connection events.
`CONNECT_MASK`	Indicates a nonblocking connection completion. A reactor dispatches the `handle_output()` hook method to process nonblocking connection completion events.

- These values can be combined (``or'd'' together) to efficiently designate a set of events
- This set of events can populate the **`ACE_Reactor_Mask`** parameter that's passed to the **`ACE_Reactor::register_handler()`** methods

Event Handler Hook Method Return Values

- When registered events occur, the reactor dispatches the appropriate event handler's **`handle_*()`** hook methods to process them
- When a **`handle_*()`** method finishes its processing, it must return a value that's interpreted by the reactor as follows:

Return value	Behavior
Zero (0)	•Indicates that the reactor should continue to detect & dispatch the registered event for this event handler (& handle if it's an I/O event) •This behavior is common for event handlers that process multiple instances of an event, for example, reading data from a socket as it becomes available
Minus one (–1)	•Instructs the reactor to stop detecting the registered event for this event handler (& handle if it's an I/O event)
Greater than zero (> 0)	•Indicates that the reactor should continue to detect & dispatch the registered event for this event handler •If a value >0 is returned after processing an I/O event, the reactor will dispatch this event handler on the handle again *before* the reactor blocks on its event demultiplexer

- Before the reactor removes an event handler, it invokes the handler's hook method **`handle_close()`**, passing **`ACE_Reactor_Mask`** of the event that's now unregistered

Sidebar: Idioms for Designing Event Handlers

- To prevent starvation of activated event handlers, keep the execution time of an event handler's **`handle_*()`** hook methods short
  - Ideally shorter than the average interval between event occurrences
- If an event handler has to run for a long time, consider queueing the request in an **`ACE_Message_Queue`** & processing it later, e.g., using a Half-Sync/Half-Async pattern
- Consolidate an event handler's cleanup activities in its **`handle_close()`** hook method, rather than dispersing them throughout its other methods
  - This idiom is particularly important when dealing with dynamically allocated event handlers that are deallocated via **`delete this`**, because it's easier to check whether there are potential problems with deleting non-dynamically allocated memory
- Only call **`delete this`** in an event handler's **`handle_close()`** method & only after the handler's *final* registered event has been removed from the reactor
  - This idiom avoids dangling pointers that can otherwise occur if an event handler that is registered with a reactor for multiple events is deleted prematurely

Sidebar: Tracking Event Handler Registrations (1/2)

- Applications are responsible for determining when a dynamically allocated event handler can be deleted
- In the following example, the **`mask_`** data member is initialized to accept both read & write events
- The **`this`** object (**`My_Event_Handler`** instance) is then registered with the reactor

```
class My_Event_Handler : public ACE_Event_Handler {
private:
  // Keep track of the events the handler's registered
for.
  ACE_Reactor_Mask mask_;
public:
  // ... class methods shown below ...
};

My_Event_Handler (ACE_Reactor *r): ACE_Event_Handler (r) {
  ACE_SET_BITS (mask_,
                ACE_Event_Handler::READ_MASK
                | ACE_Event_Handler::WRITE_MASK);
  reactor ()->register_handler (this, mask_);
}
```

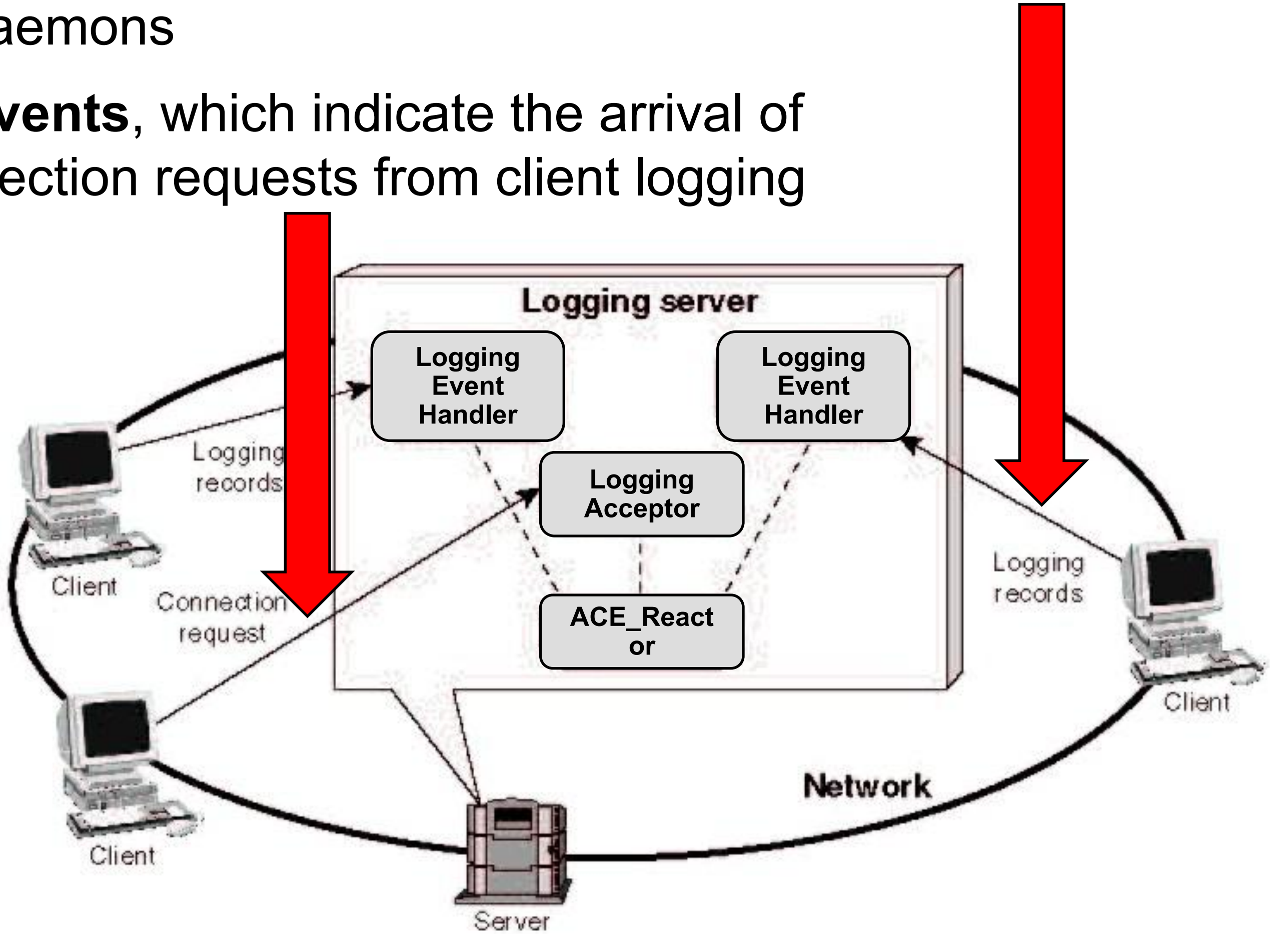
Sidebar: Tracking Event Handler Registrations (2/2)

- Whenever a **`handle_*()`** method returns an error (-1), the reactor passes the corresponding event's mask to the event handler's **`handle_close()`** method to unregister that event
- The **`handle_close()`** method clears the corresponding bit
- Whenever the **`mask_`** data member becomes zero, the dynamically allocated event handler must be deleted

```
virtual int handle_close (ACE_HANDLE, ACE_Reactor_Mask mask)
{
  if (mask == ACE_Event_Handler::READ_MASK) {
    ACE_CLR_BITS (mask_, ACE_Event_Handler::READ_MASK);
    // Perform READ_MASK cleanup logic...
  }
  if (mask == ACE_Event_Handler::WRITE_MASK) {
    ACE_CLR_BITS (mask_, ACE_Event_Handler::WRITE_MASK);
    // Perform WRITE_MASK cleanup logic.
  }
  if (mask_ == 0) delete this;
  return 0;
}
```

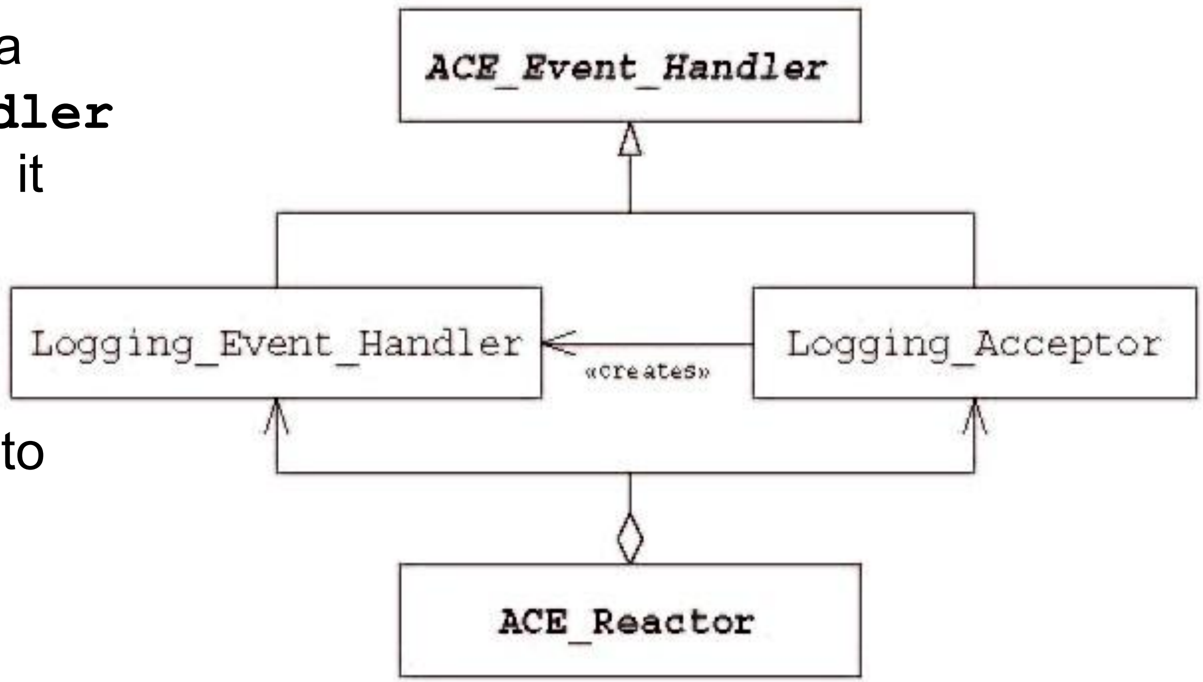
Using the ACE_Event_Handler Class (1/8)

- We implement our logging server by inheriting from `ACE_Event_Handler` & driving its processing via the reactor's event loop to handle two types of events:
  - **Data events**, which indicate the arrival of log records from connected client logging daemons
  - **Accept events**, which indicate the arrival of new connection requests from client logging daemons



Using the ACE_Event_Handler Class (2/8)

- We define two types of event handlers in our logging server:
  - **`Logging_Event_Handler`**
    - Processes log records received from a connected client logging daemon
    - Uses the **`ACE_SOCK_Stream`** to read log records from a connection
  - **`Logging_Acceptor`**
    - A factory that allocates a **`Logging_Event_Handler`** dynamically & initializes it when a client logging daemon connects
    - Uses **`ACE_SOCK_Acceptor`** to initialize **`ACE_SOCK_Stream`** contained in **`Logging_Event_Handler`**



Using the ACE_Event_Handler Class (3/8)

- `Logging_Acceptor` is a factory that allocates a `Logging_Event_Handler` dynamically & initializes it when a client logging daemon connects

```
class Logging_Acceptor : public ACE_Event_Handler {
private:
  // Factory that connects <ACE_SOCK_Stream>s passively.
  ACE_SOCK_Acceptor acceptor_;

public:
  // Simple constructor.
  Logging_Acceptor (ACE_Reactor *r = ACE_Reactor::instance ())
    : ACE_Event_Handler (r) {}

  // Initialization method.
  virtual int open (const ACE_INET_Addr &local_addr);

  // Called by a reactor when there's a new connection to
accept.
  virtual int handle_input (ACE_HANDLE = ACE_INVALID_HANDLE);
```

Note default use of reactor singleton

Key hook method dispatched by reactor

Sidebar: Singleton Pattern

- The Singleton pattern ensures a class has only instance & provides a global point of access to that instance
- e.g.,

```
class Singleton {
public:
  static Singleton *instance(){
    if (instance_ == 0) {
      instance_ =
         new Singleton;
    }
    return instance_;
  }
  void method_1 ();
  // Other methods omitted.
private:
  static Singleton *instance_;
     // Initialized to 0.
};
```

- ACE offers singletons of a number of important classes, accessed via their `instance()` method, e.g., **ACE_Reactor** & **ACE_Thread_Manager**
- You can also turn your class into a singleton via **ACE_Singleton**
- e.g.,

```
class MyClass {…};
typedef
  ACE_Singleton<MyClass,
     ACE_Thread_Mutex>
  TheSystemClass;
…
MyClass *c =
  TheSystemClass::
    instance ();
```

Be careful using Singleton – it can cause tightly coupled designs!

Using the ACE_Event_Handler Class (4/8)

```
  virtual int handle_close (ACE_HANDLE = ACE_INVALID_HANDLE,
                            ACE_Reactor_Mask = 0);
```

Hook method called when object removed from Reactor

```
  // Return the passive-mode socket's I/O handle.
  virtual ACE_HANDLE get_handle () const
    { return acceptor_.get_handle (); }
};

int Logging_Acceptor::open (const ACE_INET_Addr &local_addr)
{
  if (acceptor_.open (local_addr) == -1) return -1;
  return reactor ()->register_handler
           (this, ACE_Event_Handler::ACCEPT_MASK);
}
```

Register ourselves with the reactor for accept events

```
int Logging_Acceptor::handle_close (ACE_HANDLE,
                                    ACE_Reactor_Mask) {
  acceptor_.close ();
  delete this;
  return 0;
}
```

It's ok to "delete this" in this context!

Using the ACE_Event_Handler Class (5/8)

- `Logging_Event_Handler` processes log records received from a connected client logging daemon

```
class Logging_Event_Handler : public ACE_Event_Handler {
protected:
  // File where log records are written.
  ACE_FILE_IO log_file_;

  Logging_Handler logging_handler_; // Connection to remote
peer.
public:
  // Initialize the base class & logging handler.
  Logging_Event_Handler (ACE_Reactor *r)
    : ACE_Event_Handler (r), logging_handler_ (log_file_) {}

  virtual int open (); // Activate the object.

  // Called by a reactor when logging events arrive.
  virtual int handle_input (ACE_HANDLE = ACE_INVALID_HANDLE);
```

Key hook method dispatched by reactor

```
  // Called by a reactor when handler is closing.
  virtual int handle_close (ACE_HANDLE, ACE_Reactor_Mask);
```

Using the ACE_Event_Handler Class (6/8)

Factory method called back by reactor when a connection event occurs

```
int Logging_Acceptor::handle_input (ACE_HANDLE) {
  Logging_Event_Handler *peer_handler = 0;
  ACE_NEW_RETURN (peer_handler,
                  Logging_Event_Handler (reactor ()),
-1);
  if (acceptor_.accept (peer_handler->peer ()) == -1) {
    delete peer_handler;
    return -1;
  } else if (peer_handler->open () == -1) {
    peer_handler->handle_close ();
    return -1;
  }
  return 0;
}
```

Sidebar: ACE Memory Management Macros

- Early C++ compilers returned a **NULL** for failed memory allocations; the newer compilers throw an exception
- ACE macros unify the behavior & return **NULL** irrespective of whether an exception is thrown or not
- They also set **errno** to **ENOMEM**
- **ACE_NEW_RETURN** returns a valid pointer or **NULL** on failure
- **ACE_NEW** simply returns
- **ACE_NEW_NORETURN** continues to execute even on failure

- Following version is for compilers that throw **std::bad_alloc** on allocation failure

```
#define ACE_NEW_RETURN(POINTER,CTOR,RET_VAL) \
 do { try { POINTER = new CTOR; } catch (std::bad_alloc) \
   { errno = ENOMEM; POINTER = 0; return RET_VAL; } \
 } while (0)
```

- Following is for compilers that offer a **nothrow** variant of **operator new**

```
#define ACE_NEW_RETURN(POINTER,CTOR,RET_VAL) \
 do { POINTER = new (ACE_nothrow) CTOR; \
   if (POINTER == 0) { errno = ENOMEM; return RET_VAL; } \
 } while (0)
```

Using the ACE_Event_Handler Class (7/8)

```
int Logging_Event_Handler::open () {
  static std::string logfile_suffix = ".log";
  std::string filename (MAXHOSTNAMELEN, '\0');
  ACE_INET_Addr logging_peer_addr;

  logging_handler_.peer ().get_remote_addr
 (logging_peer_addr);
  logging_peer_addr.get_host_name (filename.c_str (),
                                   filename.size ());
  filename += logfile_suffix;
  ACE_FILE_Connector connector;
  connector.connect (log_file_,
                     ACE_FILE_Addr (filename.c_str ()),
                     0, // No timeout.
                     ACE_Addr::sap_any, // Ignored.
                     0, // Don't try to reuse the addr.
                     O_RDWR|O_CREAT|O_APPEND,
                     ACE_DEFAULT_FILE_PERMS);

  return reactor ()->register_handler
           (this, ACE_Event_Handler::READ_MASK);
}
```

Create the log file

Register with the reactor for input events

Using the ACE_Event_Handler Class (8/8)

Called back by the reactor when a data event occurs

```
int Logging_Event_Handler::handle_input (ACE_HANDLE)
{
  return logging_handler_.log_record ();
}
```

Returns `-1` when client closes connection

```
int Logging_Event_Handler::handle_close (ACE_HANDLE,
                                         ACE_Reactor_Mask)
{
  logging_handler_.close ();
  log_file_.close ();
  delete this;
  return 0;
}
```

Called back by the reactor when `handle_input()` returns `-1`

Sidebar: Event Handler Memory Management (1/2)

Event handlers should generally be allocated dynamically for the following reasons:

- ***Simplify memory management:*** For example, deallocation can be localized in an event handler's **`handle_close()`** method, using the event handler event registration tracking idiom
- ***Avoid "dangling handler" problems:***
  - For example an event handler may be instantiated on the stack or as a member of another class
  - Its lifecycle is therefore controlled externally, however, its reactor registrations are controlled internally to the reactor
  - If the handler gets destroyed while it is still registered with a reactor, there will be unpredictable problems later if the reactor tries to dispatch the nonexistent handler
- ***Avoid portability problems:*** For example, dynamic allocation alleviates subtle problems stemming from the delayed event handler cleanup semantics of the **`ACE_WFMO_Reactor`**

Sidebar: Event Handler Memory Management (2/2)

- **Real-time systems**
  - They avoid or minimize the use of dynamic memory to improve their predictability
  - Event handlers could be allocated statically for such applications
- **Event Handler Memory Management in Real-time Systems**

1. Do not call **`delete this`** in **`handle_close()`**
2. Unregister all events from reactors in the class destructor, at the latest
3. Ensure that the lifetime of a registered event handler is longer than the reactor it's registered with if it can't be unregistered for some reason.
4. Avoid the use of the **`ACE_WFMO_Reactor`** since it defers the removal of event handlers, thereby making it hard to enforce convention 3
5. If using **`ACE_WFMO_Reactor`**, pass the **`DONT_CALL`** flag to **`ACE_Event_Handler::remove_handler()`** & carefully manage shutdown activities without the benefit of the reactor's **`handle_close()`** callback

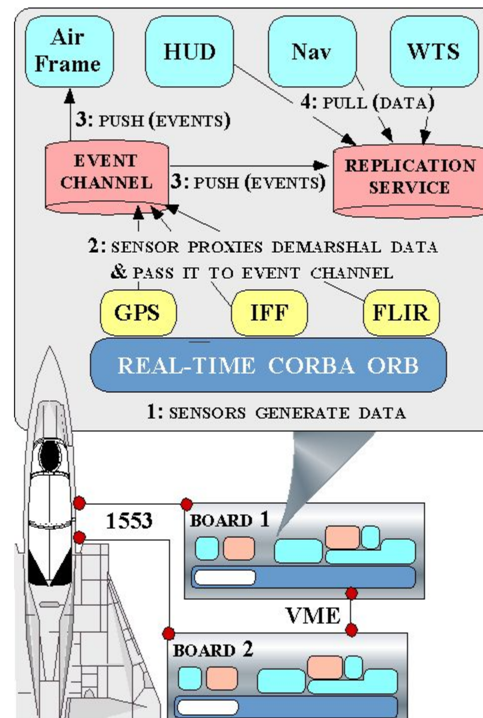
Sidebar: Handling Silent Peers

- A client disconnection, both graceful & abrupt, are handled by the reactor by detecting that the socket has become readable & will dispatch the **`handle_input()`** method, which then detects the closing of the connection
- A client may, however, stop communicating for which no event gets generated in the reactor, which may be due to:
  - A network cable being pulled out & put back shortly
  - A host crashes without closing any connections
- These situations can be dealt with in a number of ways:
  - Wait until the TCP keepalive mechanism abandons the peer & closes the connection, which can be a very slow procedure
  - Implement an application-level policy or mechanism, like a heartbeat that periodically tests for connection liveness
  - Implement an application-level policy where if no data has been received for a while, the connection is considered to be closed

The ACE Timer Queue Classes (1/2)

Motivation

- Many networked applications perform activities periodically or must be notified when specified time periods have elapsed
- Conventional OS timer mechanisms are limited since they
  - Support a limited number of timers &
  - Use signals to expire the timers

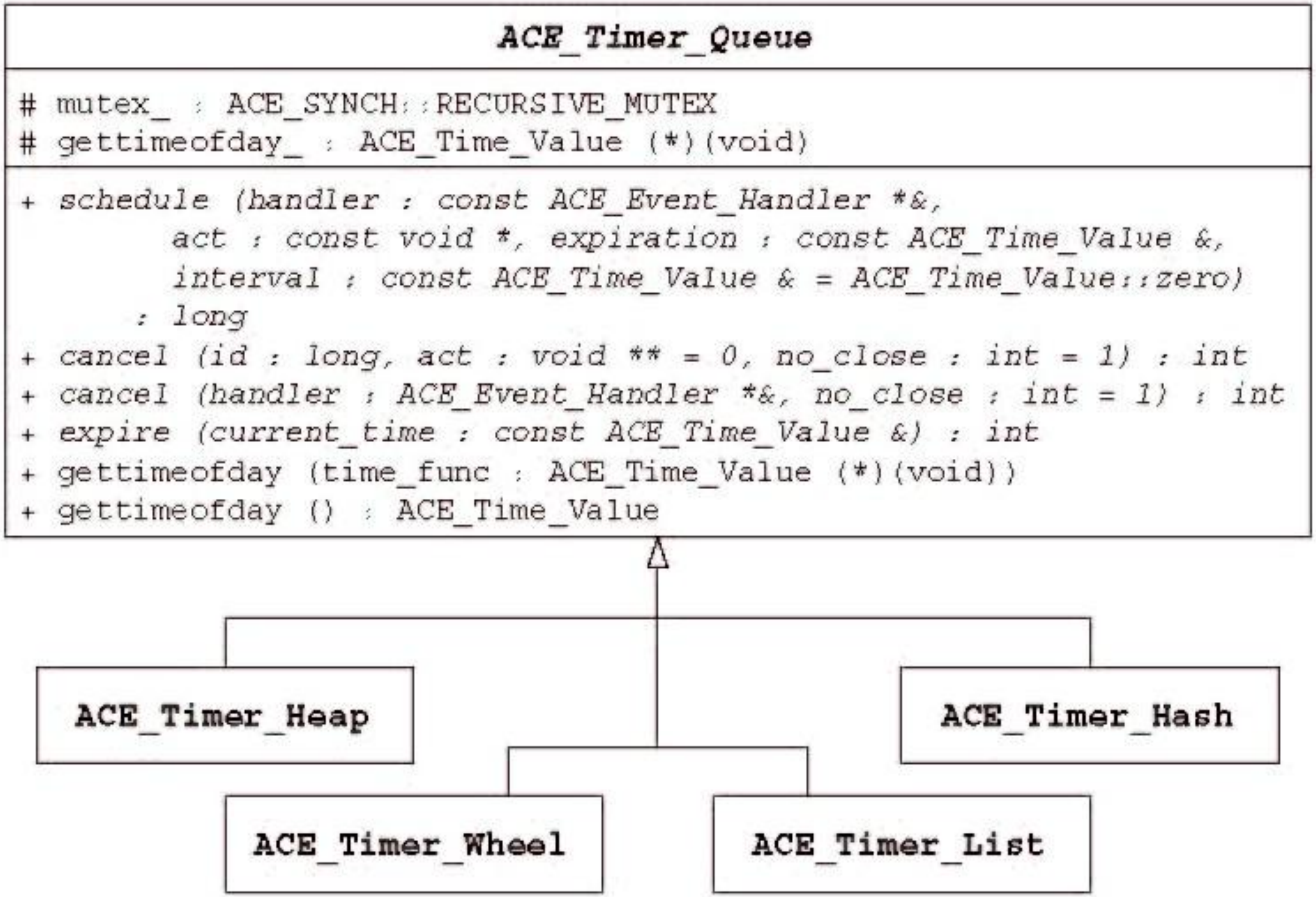


The ACE Timer Queue Classes (2/2)

Class Capabilities

- The ACE timer queue classes allow applications to register time-driven **`ACE_Event_Handler`** subclasses that provides the following capabilities:
  - They allow applications to schedule event handlers whose **`handle_timeout()`** hook methods will be dispatched efficiently & scalably at caller-specified times in the future, either once or at periodic intervals
  - They allow applications to cancel a timer associated with a particular event handler or all timers associated with an event handler
  - They allow applications to configure a timer queue's time source

The ACE Timer Queue Classes API



This class handles *variability* of timer queue mechanisms via a *common* timer queue API

Scheduling ACE_Event_Handler for Timeouts

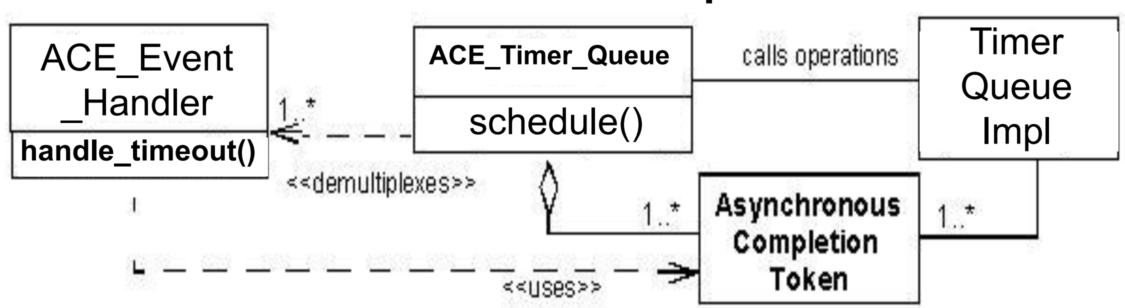
- The **`ACE_Timer_Queue`**'s **`schedule()`** method is passed two parameters:
  1. A pointer to an event handler that will be the target of the subsequent **`handle_timeout()`** dispatching and
  2. A reference to an **`ACE_Time_Value`** indicating the absolute timers future time when the **`handle_timeout()`** hook method should be invoked on the event handler

- **`schedule()`** also takes two more optional parameters:
  3. A **`void`** pointer that's stored internally by the timer queue & passed back unchanged when **`handle_timeout()`** is dispatched
     - This pointer can be used as an *asynchronous completion token* (ACT) in accordance with the *Asynchronous Completion Token* pattern
     - By using an ACT, the same event handler can be registered with a timer queue at multiple future dispatching times
  4. A reference to a second **`ACE_Time_Value`** that designates the interval at which the event handler should be dispatched periodically

The Asynchronous Completion Token Pattern

- This pattern allows an application to efficiently demultiplex & process the responses of an asynchronous operation it invokes on services
- Together with each async operation that a client *initiator* invokes on a *service*, transmit information (i.e., the *ACT*) that identifies how the initiator should process the service's response

Structure & Participants

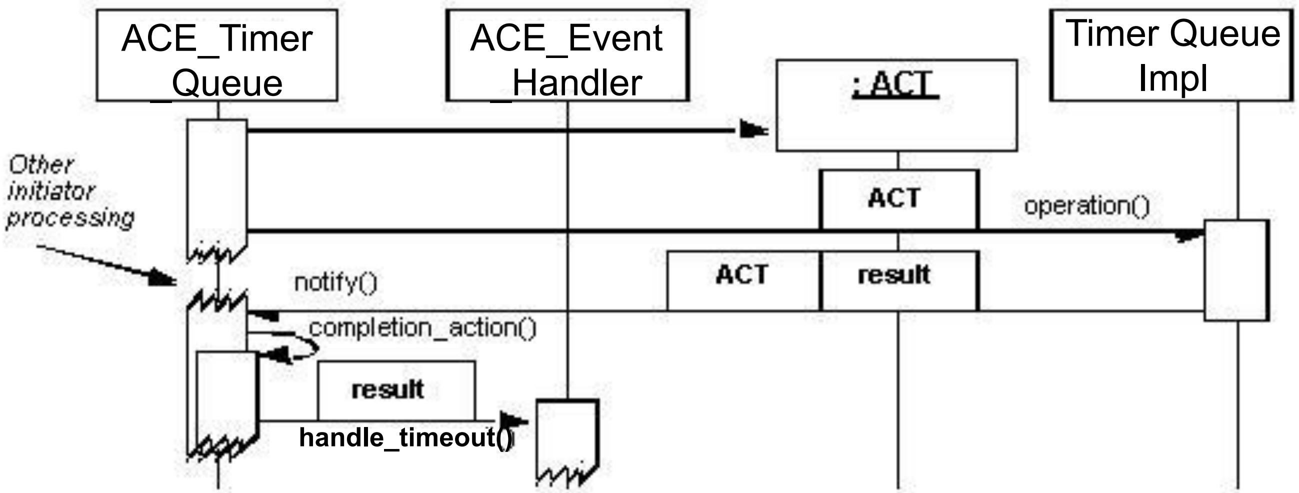


- In the **`ACE_Timer_Queue`**, **`schedule()`** is the async operation & the ACT is a **`void *`** passed to **schedule()**

The Asynchronous Completion Token Pattern

- When the timer queue dispatches the handle_timeout() method on the event handler, the ACE is passed so that it can be used to demux the response efficiently

Dynamic Interactions



- The use of this pattern minimizes the number of event handlers that need to be created to handle timeouts.

Sidebar: ACE Time Sources

- The static time returning methods of **`ACE_Timer_Queue`** are required to provide an accurate basis for timer scheduling & expiration decisions
- In ACE this is done in two ways:
  - **`ACE_OS::gettimeofday()`** is a static method that returns a **`ACE_Time_Value`** containing the current absolute date & time as reported by the OS
  - **`ACE_High_Res_Timer::gettimeofday_hr()`** is a static method that returns the value of an OS-specific high resolution timer, converted to **`ACE_Time_Value`** units based on number of clock ticks since boot time
- The granularities of these two timers varies by three to four orders of magnitude
- For timeout events, however, the granularities are similar due to complexities of clocks, OS scheduling & timer interrupt servicing
- If the application's timer behavior must remain constant, irrespective of whether the system time was changed or not, its timer source must use the **`ACE_High_Res_Timer::gettimeofday_hr()`**

Using the ACE Timer Classes (1/4)

- We now show how to apply ACE timer queue "interval timers" to reclaim resources from those event handlers whose clients log records infrequently
- We use the *Evictor pattern*, which describes how & when to release resources, such as memory & I/O handles, to optimize system resource management

```
class Logging_Acceptor_Ex : public Logging_Acceptor {
public:
  typedef ACE_INET_Addr PEER_ADDR;

  // Simple constructor to pass <ACE_Reactor> to base class.
  Logging_Acceptor_Ex (ACE_Reactor *r = ACE_Reactor::instance
())
    : Logging_Acceptor (r) {}

  int handle_input (ACE_HANDLE) {
    Logging_Event_Handler_Ex *peer_handler = 0;
```

Only difference (variability) is the event handler type…

```
    ACE_NEW_RETURN (peer_handler,
                    Logging_Event_Handler_Ex (reactor ()), -1);
    // ... same as Logging_Acceptor::handle_input()
  }
```

Using the ACE Timer Classes (2/4)

```
class Logging_Event_Handler_Ex : public Logging_Event_Handler
{
private:
  // Time when a client last sent a log record.
  ACE_Time_Value time_of_last_log_record_;

  // Maximum time to wait for a client log record.
  const ACE_Time_Value max_client_timeout_;
public:
  typedef Logging_Event_Handler PARENT;

  // 3600 seconds == one hour.
  enum { MAX_CLIENT_TIMEOUT = 3600 };

  Logging_Event_Handler_Ex
      (ACE_Reactor *reactor,
       const ACE_Time_Value &max_client_timeout
         = ACE_Time_Value (MAX_CLIENT_TIMEOUT))
    : Logging_Event_Handler (reactor),
      time_of_last_log_record (0),
      max_client_timeout_ (max_client_timeout) {}
```

Using the ACE Timer Classes (3/4)

```
  virtual int open (); // Activate the event handler.

  // Called by a reactor when logging events arrive.
  virtual int handle_input (ACE_HANDLE);

  // Called when a timeout expires to check if the client has
  // been idle for an excessive amount of time.
  virtual int handle_timeout (const ACE_Time_Value &tv,
                              const void *act);
};
```

```
int Logging_Event_Handler_Ex::open () {
  int result = PARENT::open ();
  if (result != -1) {
    ACE_Time_Value reschedule (max_client_timeout_.sec () / 4);
    result = reactor ()->schedule_timer
                (this, 0,
                 max_client_timeout_,  // Initial timeout.
                 reschedule);          // Subsequent timeouts.
  }
  return result;
}
```

Creates an interval timer that fires every 15 minutes

Using the ACE Timer Classes (4/4)

```
int Logging_Event_Handler_Ex::handle_input (ACE_HANDLE h)
{
  time_of_last_log_record_ =
    reactor ()->timer_queue ()->gettimeofday ();
  return PARENT::handle_input (h);
}
```

Log the last time this client was active

```
int Logging_Event_Handler_Ex::handle_timeout
     (const ACE_Time_Value &now, const void *)
{
  if (now - time_of_last_log_record_ >= max_client_timeout_)
    reactor ()->remove_handler (this,

ACE_Event_Handler::READ_MASK);
  return 0;
}
```

Evict the handler if client has been inactive too long

Sidebar: Using Timers in Real-time Apps

- Real-time applications must demonstrate predictable behavior
- If a reactor is used to dispatch both I/O & timer queue handlers, the timing variations in I/O handling can cause unpredictable behavior
- The event demultiplexing & synchronization framework integrating I/O handlers & timer mechanisms in the reactor can cause unnecessary overhead for real-time applications
- Real-time applications, must, therefore choose to handle timers in a separate thread using the **`ACE_Timer_Queue`**
- Different thread priorities can be assigned based on the priorities of the timer & I/O events
    - This facility is provided by the **`ACE_Thread_Timer_Queue_Adapter`**
        - See **`$ACE_ROOT/examples/Timer_Queue/`** for examples

Sidebar: Minimizing ACE Timer Queue Memory Allocation

- ACE_Timer_Queue doesn't support a **`size()`** method since there's no generic way to represent size of different implementations of timer queue
- The timer queue subclasses therefore offer size related parameters in their constructors
- The timer queue can resize automatically, however, this strategy involves dynamic memory allocation that can be a source of overhead for real-time applications
- **`ACE_Timer_Heap`** & **`ACE_Timer_Wheel`** classes offer the ability to preallocate timer queue entries
- ACE reactor can use a custom-tuned timer queue using the following:
  1. Instantiate the desired ACE timer queue class with the size & preallocation argument, if any
  2. Instantiate the ACE reactor implementation object with the timer queue from step 1
  3. Instantiate a new **`ACE_Reactor`** object supplying the reactor implementation

The ACE_Reactor Class (1/2)

Motivation

- Event-driven networked applications have historically been programmed using native OS mechanisms, such as the Socket API & the **`select()`** synchronous event demultiplexer
- Applications developed this way, however, are not only nonportable, they are inflexible because they tightly couple low-level event detection, demultiplexing, & dispatching code together with application event processing code
- Developers must therefore rewrite all this code for each new networked application, which is tedious, expensive, & error prone
- It's also unnecessary because much of event detection, demultiplexing, & dispatching can be generalized & reused across many networked applications.

The ACE_Reactor Class (2/2)

Class Capabilities

- This class implements the Facade pattern to define an interface for ACE Reactor framework capabilities:
  - It centralizes event loop processing in a reactive application
  - It detects events via an event demultiplexer provided by the OS & used by the reactor implementation
  - It demultiplexes events to event handlers when the event demultiplexer indicates the occurrence of the designated events
  - It dispatches the hook methods on event handlers to perform application-defined processing in response to the events
  - It ensures that any thread can change a Reactor's event set or queue a callback to an event handler & expect the Reactor to act on the request promptly

The ACE_Reactor Class API

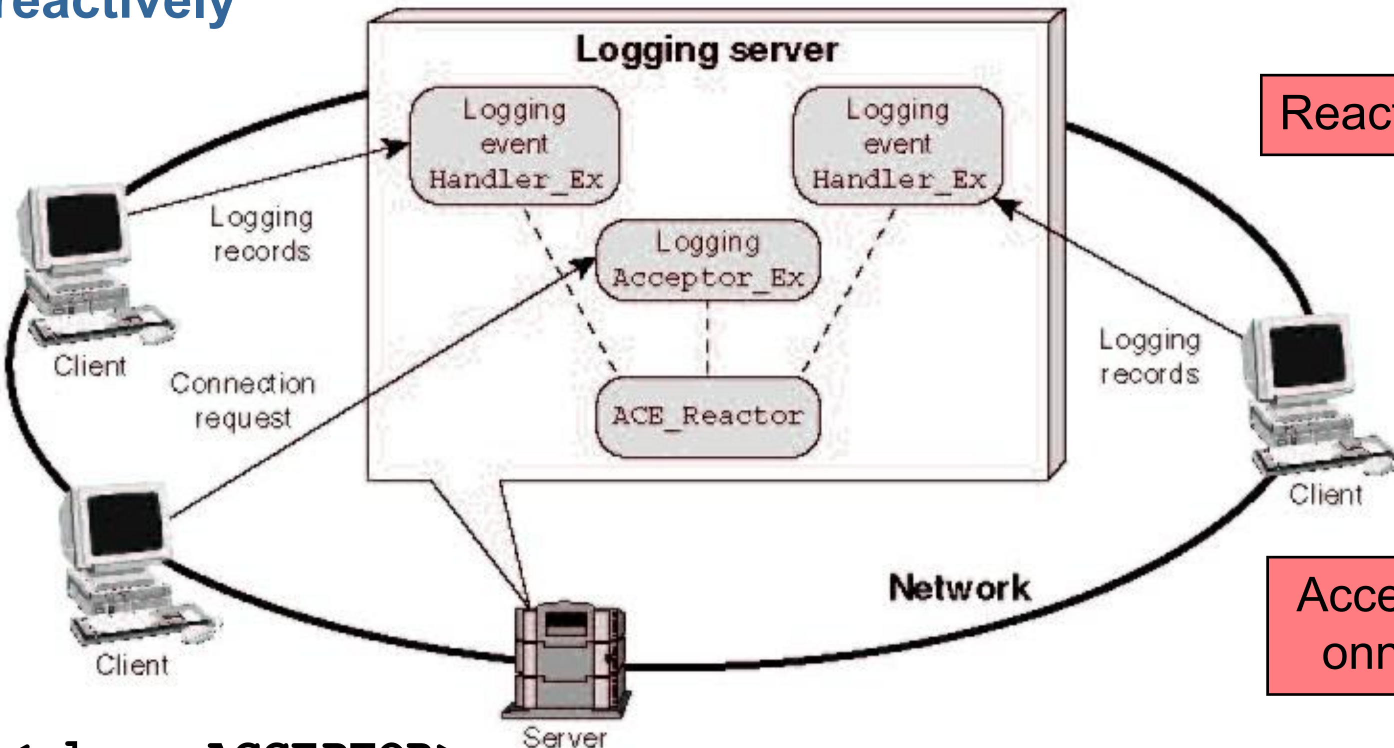
This class handles *variability* of synchronous event demuxing mechanisms via a *common* API

```
ACE_Reactor
---------------------------------------------------------------
# reactor_ : ACE_Reactor *
# implementation_ : ACE_Reactor_Impl *
---------------------------------------------------------------
+ ACE_Reactor (implementation : ACE_Reactor_Impl * = 0,
               delete_implementation : int = 0)
+ open (max_handles : int, restart : int = 0,
        sig_handler : ACE_Sig_Handler * = 0,
        timer_queue : ACE_Timer_Queue * = 0) : int
+ close () : int
+ register_handler (handler : ACE_Event_Handler *,
                    mask : ACE_Reactor_Mask) : int
+ register_handler (io : ACE_HANDLE, handler : ACE_Event_Handler *,
                    mask : ACE_Reactor_Mask) : int
+ remove_handler (handler : ACE_Event_Handler *,
                  mask : ACE_Reactor_Mask) : int
+ remove_handler (io : ACE_HANDLE, mask : ACE_Reactor_Mask) : int
+ remove_handler (hs : const ACE_Handle_Set&, m : ACE_Reactor_Mask) : int
+ suspend_handler (handler : ACE_Event_Handler *) : int
+ resume_handler (handler : ACE_Event_Handler *) : int
+ mask_ops (handler : ACE_Event_handler *,
            mask : ACE_Reactor_Mask, ops : int) : int
+ schedule_wakeup (handler : ACE_Event_Handler *,
                   masks_to_be_added : ACE_Reactor_Mask ) : int
+ cancel_wakeup (handler : ACE_Event_Handler *,
                 masks_to_be_cleared : ACE_Reactor_Mask) : int
+ handle_events (max_wait_time : ACE_Time_Value * = 0) : int
+ run_reactor_event_loop (event_hook : int (*)(void *) = 0) : int
+ end_reactor_event_loop () : int
+ reactor_event_loop_done () : int
+ schedule_timer (handler : ACE_Event_Handler *, arg : void *,
                  delay : ACE_Time_Value &,
                  repeat : ACE_Time_Value & = ACE_Time_Value::zero) : int
+ cancel_timer (handler : ACE_Event_Handler *,
                dont_call_handle_close : int = 1) : int
+ cancel_timer (timer_id : long, arg : void ** = 0,
                dont_call_handle_close : int = 1) : int
+ notify (handler : ACE_Event_Handler * = 0,
          mask : ACE_Reactor_Mask = ACE_Event_Handler::EXCEPT_MASK,
          timeout : ACE_Time_Value * = 0) : int
+ max_notify_iterations (iterations : int) : int
+ purge_pending_notifications (handler : ACE_Event_Handler *,
                    mask : ACE_Reactor_Mask = ALL_EVENTS_MASK) : int
+ instance () : ACE_Reactor *
+ owner (new_owner : ACE_thread_t, old_owner : ACE_thread_t * = 0) : int
```

Using the ACE_Reactor Class (1/4)

- This example illustrates a server that runs in a single thread of control in a single process, handling log records from multiple clients reactively

Wrapper Facade

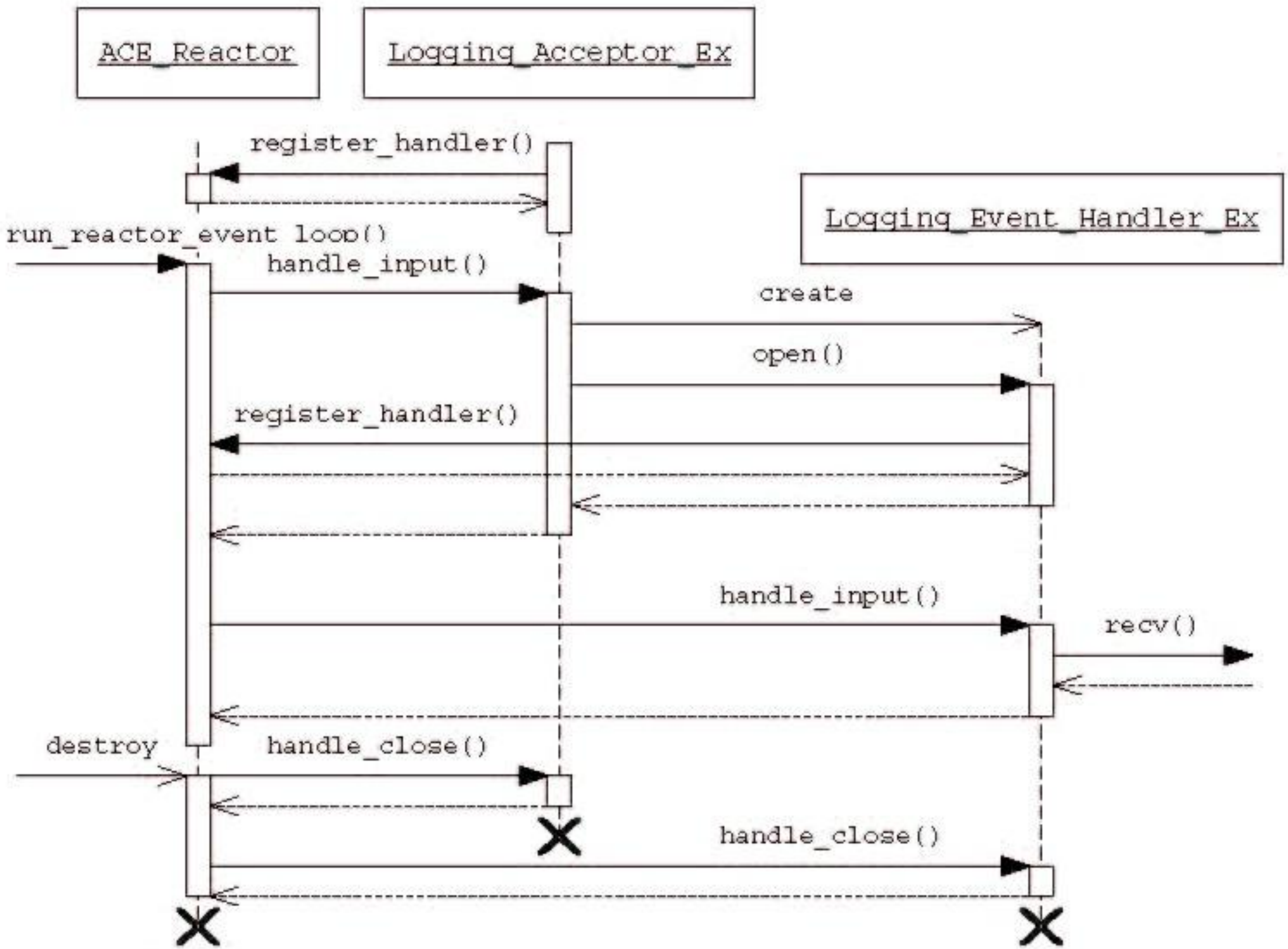


Reactor

Acceptor/Connector

```
template <class ACCEPTOR>
class Reactor_Logging_Server : public ACCEPTOR {
public:
  Reactor_Logging_Server (int argc, char *argv[], ACE_Reactor
*);
```

Using the ACE_Reactor Class (2/4)



Sequence Diagram for Reactive Logging Server

Using the ACE_Reactor Class (3/4)

```
template <class ACCEPTOR>
Reactor_Logging_Server<ACCEPTOR>::Reactor_Logging_Server
  (int argc, char *argv[], ACE_Reactor *reactor)
  : ACCEPTOR (reactor) {
  u_short logger_port = argc > 1 ? atoi (argv[1]) : 0;
  ACE_TYPENAME ACCEPTOR::PEER_ADDR server_addr;
  int result;

  if (logger_port != 0)
    result = server_addr.set (logger_port, INADDR_ANY);
  else
    result = server_addr.set ("ace_logger", INADDR_ANY);
  if (result != -1)
    result = ACCEPTOR::open (server_addr);
  if (result == -1) reactor->end_reactor_event_loop ();
}
```

Shutdown the reactor's event loop if an error occurs

Using the ACE_Reactor Class (4/4)

```
typedef Reactor_Logging_Server<Logging_Acceptor_Ex>
        Server_Logging_Daemon;

int main (int argc, char *argv[]) {
  ACE_Reactor reactor;
  Server_Logging_Daemon *server = 0;
  ACE_NEW_RETURN (server,
                  Server_Logging_Daemon (argc, argv, &reactor),
                  1);

  if (reactor.run_reactor_event_loop () == -1)
    ACE_ERROR_RETURN ((LM_ERROR, "%p\n",
                       "run_reactor_event_loop()"), 1);
  return 0;
}
```

Dynamic allocation ensures proper deletion semantics

Sidebar: Avoiding Reactor Deadlock in Multithreaded Applications (1/2)

- Reactors, though often used in single-threaded applications, can also be used in multithreaded applications
- In multi-threaded applications it is important to avoid deadlock between multiple threads that are sharing an **`ACE_Reactor`**
- **`ACE_Reactor`** attempts to solve this problem to some extent by holding a recursive mutex when it dispatches a callback to an event handler
- If the dispatched callback method directly or indirectly calls back into the reactor within the same thread of control, the recursive mutex's **`acquire()`** method detects this automatically & simply increases its count of the lock recursion nesting depth, rather than deadlocking the thread

Sidebar: Avoiding Reactor Deadlock in Multithreaded Applications (2/2)

- Deadlock can still occur under the following circumstances:
  - The original callback method calls a second method that blocks trying to acquire a mutex that's held by a second thread executing the same method
  - The second thread directly or indirectly calls into the same reactor
  - Deadlock can occur since the reactor's recursive mutex doesn't realize that the second thread is calling on behalf of the first thread where the callback method was dispatched originally
- One way to avoid **`ACE_Reactor`** deadlock in a multithreaded application is to not make blocking calls to other methods from callbacks if those methods are executed concurrently by competing threads that directly or indirectly call back into the same reactor
- It may be necessary to use an **`ACE_Message_Queue`** to exchange information asynchronously if a **`handle_*()`** callback method must communicate with another thread that accesses the same reactor

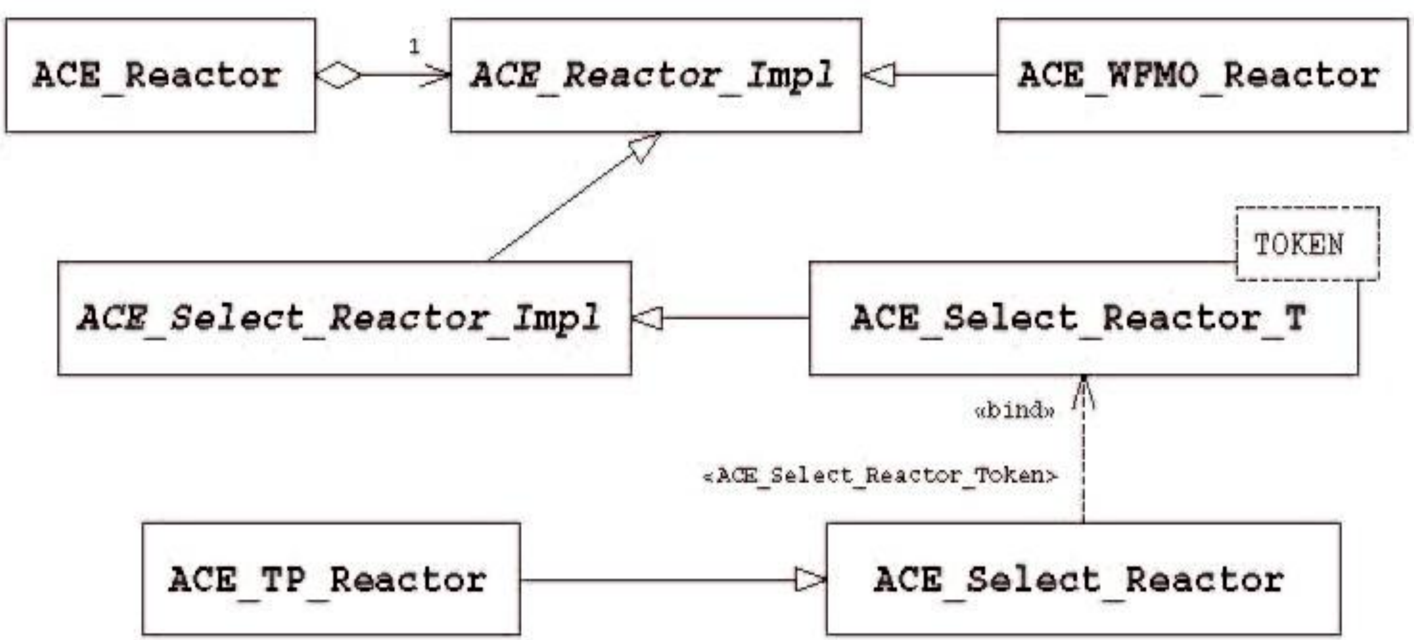
ACE Reactor Implementations (1/2)

- The ACE Reactor framework was designed for extensibility
  - There are nearly a dozen different Reactor implementations in ACE
- The most common ACE Reactor implementations are shown in the following table:

ACE Class	Description
ACE_Select_Reactor	Uses the select() synchronous event demultiplexer function to detect I/O and timer events; incorporates orderly handling of POSIX signals.
ACE_TP_Reactor	Uses the Leader/Followers pattern [POSA2] to extend ACE_Select_Reactor event handling to a pool of threads.
ACE_WFMO_Reactor	Uses the Windows WaitForMultipleObjects() event demultiplexer function to detect socket I/O, timeouts, and Windows synchronization events.

ACE Reactor Implementations (2/2)

- The relationships amongst these classes are shown in the adjacent diagram
  - Note the use of the Bridge pattern
- The **`ACE_Select_Reactor`** & **`ACE_TP_Reactor`** are more similar than the **`ACE_WFMO_Reactor`**
- It’s fairly straightforward to create your own Reactor



The ACE_Select_Reactor Class (1/2)

Motivation

- The **`select()`** function is the most common synchronous event demultiplexer

```
int select (int width,           // Maximum handle plus 1
       fd_set *read_fds,         // Set of "read" handles
       fd_set *write_fds,        // Set of "write" handles
       fd_set *except_fds,       // Set of "exception"
handles
       struct timeval *timeout);// Time to wait for events
```

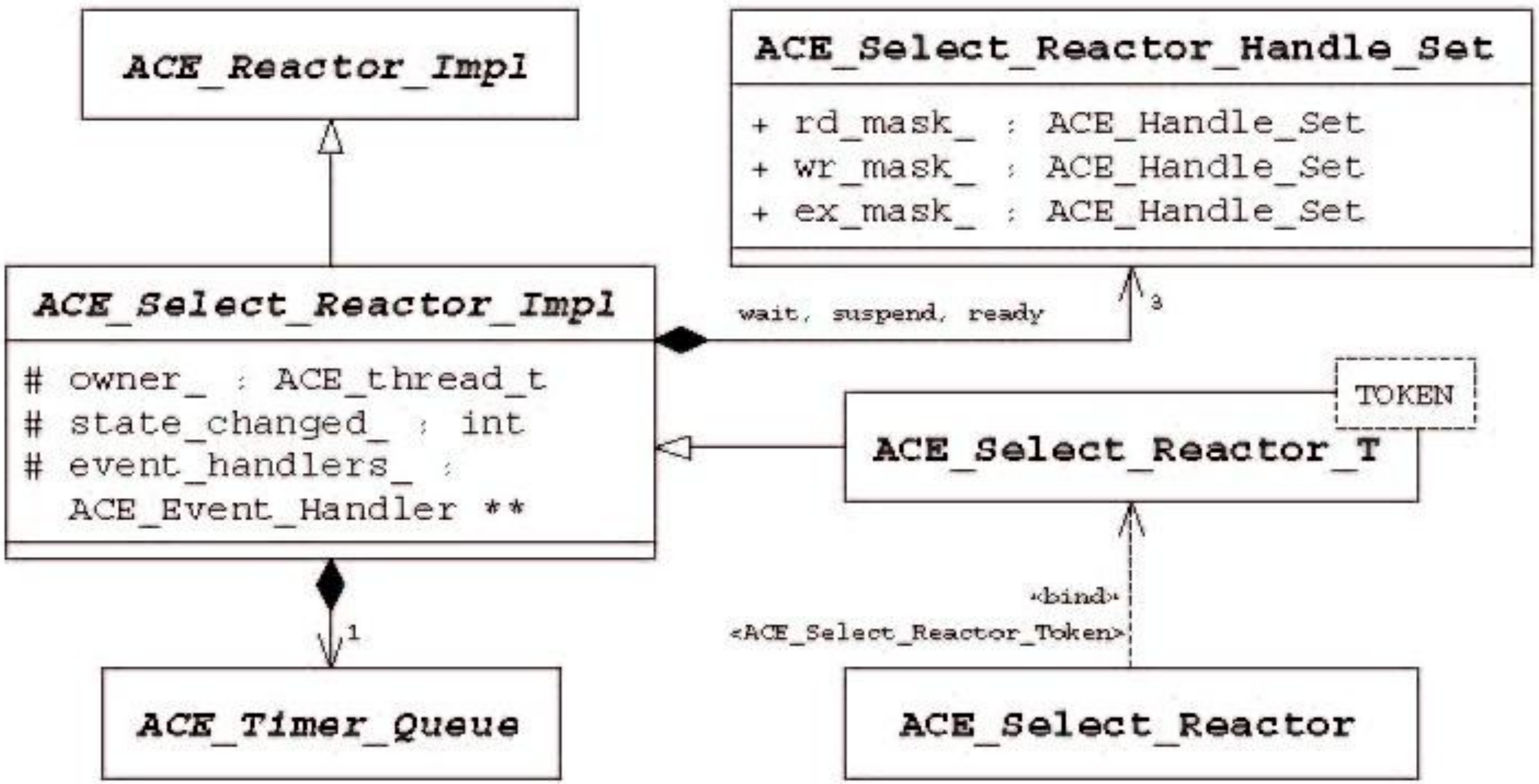
- The **`select()`** function is tedious, error-prone, & non-portable
- ACE therefore defines the **`ACE_Select_Reactor`** class, which is the default on all platforms except Windows

The ACE_Select_Reactor Class (2/2)

Class Capabilities

- This class is an implementation of the **`ACE_Reactor`** interface that provides the following capabilities:
  - It supports reentrant reactor invocations, where applications can call the **handle_events()** method from event handlers that are being dispatched by the same reactor
  - It can be configured to be either synchronized or nonsynchronized, which trades off thread safety for reduced overhead
  - It preserves fairness by dispatching all active handles in its handle sets before calling **`select()`** again

The ACE_Select_Reactor Class API



Sidebar: Controlling the Size of ACE_Select_Reactor (1/2)

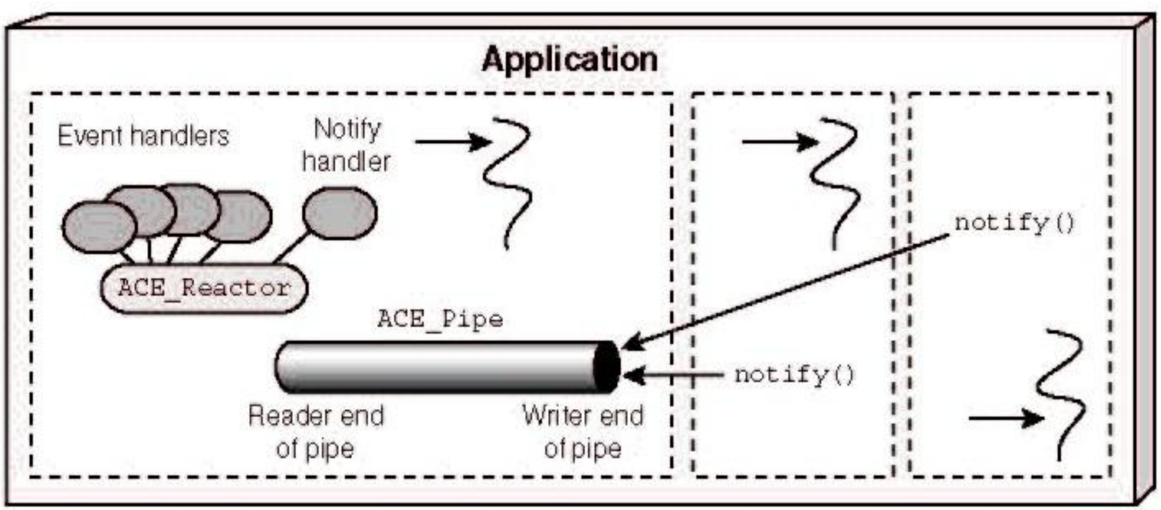
- The number of event handlers that can be managed by an **`ACE_Select_Reactor`** defaults to the value of the **`FD_SETSIZE`** macro, which is used to manipulate the size of **`fd_set`**
- **`FD_SETSIZE`** can play an important role in increasing the number of possible event handlers in **`ACE_Select_Reactor`**
- This value can be controlled as follows:
  - To create an **`ACE_Select_Reactor`** that's *smaller* than the default size of **`FD_SETSIZE`**, simply pass in the value to the **`ACE_Select_Reactor::open()`** method
    - No recompilation of the ACE library is necessary
  - To create an **`ACE_Select_Reactor`** that's larger than the default size of **`FD_SETSIZE`**, change the value of **`FD_SETSIZE`** in the **`$ACE_ROOT/ace/config.h`** file
    - Recompilation of the ACE library (& possibly the OS kernel & C library on some platforms) is required
    - After recompiling & reinstalling the necessary libraries, pass in the desired number of event handlers to the **`ACE_Select_Reactor::open()`** method
    - The number of event handlers must be less than or equal to the new **`FD_SETSIZE`** & the maximum number of handles supported by the OS

Sidebar: Controlling the Size of ACE_Select_Reactor (2/2)

- Although the steps described above make it possible to handle a large number of I/O handles per **`ACE_Select_Reactor`**, it's not necessarily a good idea since performance may suffer due to deficiencies with **`select()`**
- To handle a large numbers of handles, consider using the **`ACE_Dev_Poll_Reactor`** that's available on certain UNIX platforms
- An alternative choice could be a design using asynchronous I/O based on the ACE Proactor framework
  - The ACE Proactor is available on Windows & certain UNIX platforms that support asynchronous I/O
- Avoid the temptation to divide a large number of handles between multiple instances of **`ACE_Select_Reactor`** since one of the deficiencies stems from the need for **`select()`** to scan large **`fd_set`** structures, not ACE's use of **`select()`**

The ACE_Select_Reactor Notification Mechanism

- **`ACE_Select_Reactor`** implements its default notification mechanism via an **`ACE_Pipe`**
  - This class is a bidirectional IPC mechanism that's implemented via various OS features on different platforms
- The two ends of the pipe play the following roles:



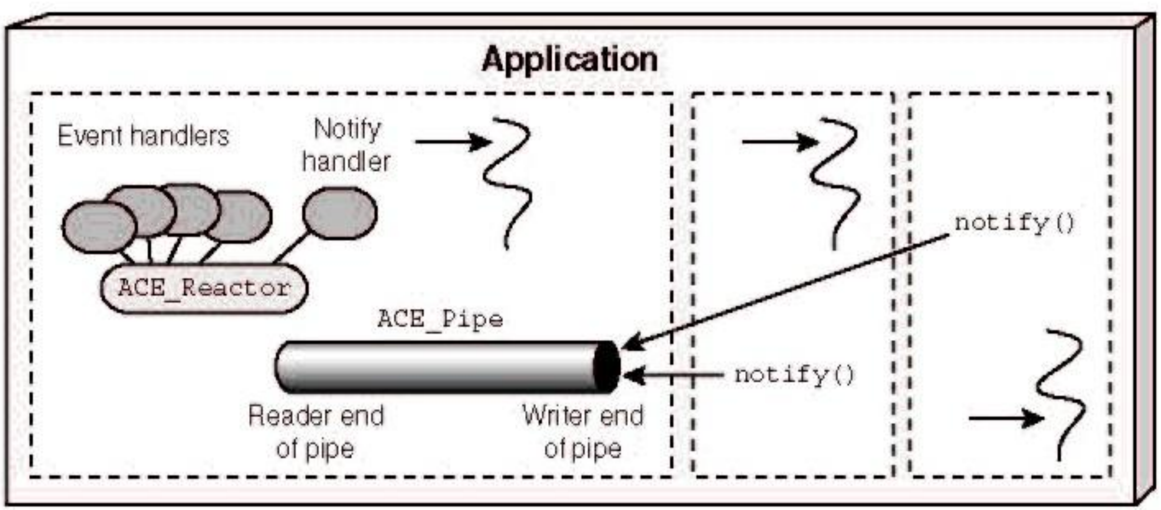
The ACE_Select_Reactor Notification Mechanism

The writer role

- The **`ACE_Select_Reactor`**'s **`notify()`** method exposes the writer end of the pipe to application threads, which use the **`notify()`** method to pass event handler pointers to an **`ACE_Select_Reactor`** via its notification pipe

The reader role

- The **`ACE_Select_Reactor`** registers the reader end of the pipe internally with a **`READ_MASK`**
- When the reactor detects an event in the reader end of its notification pipe it wakes up & dispatches its notify handler to process a user-configurable number of event handlers from the pipe
- The number of handlers dispatched is controlled by **`max_notify_iterations()`**



Sidebar: The ACE_Token Class (1/2)

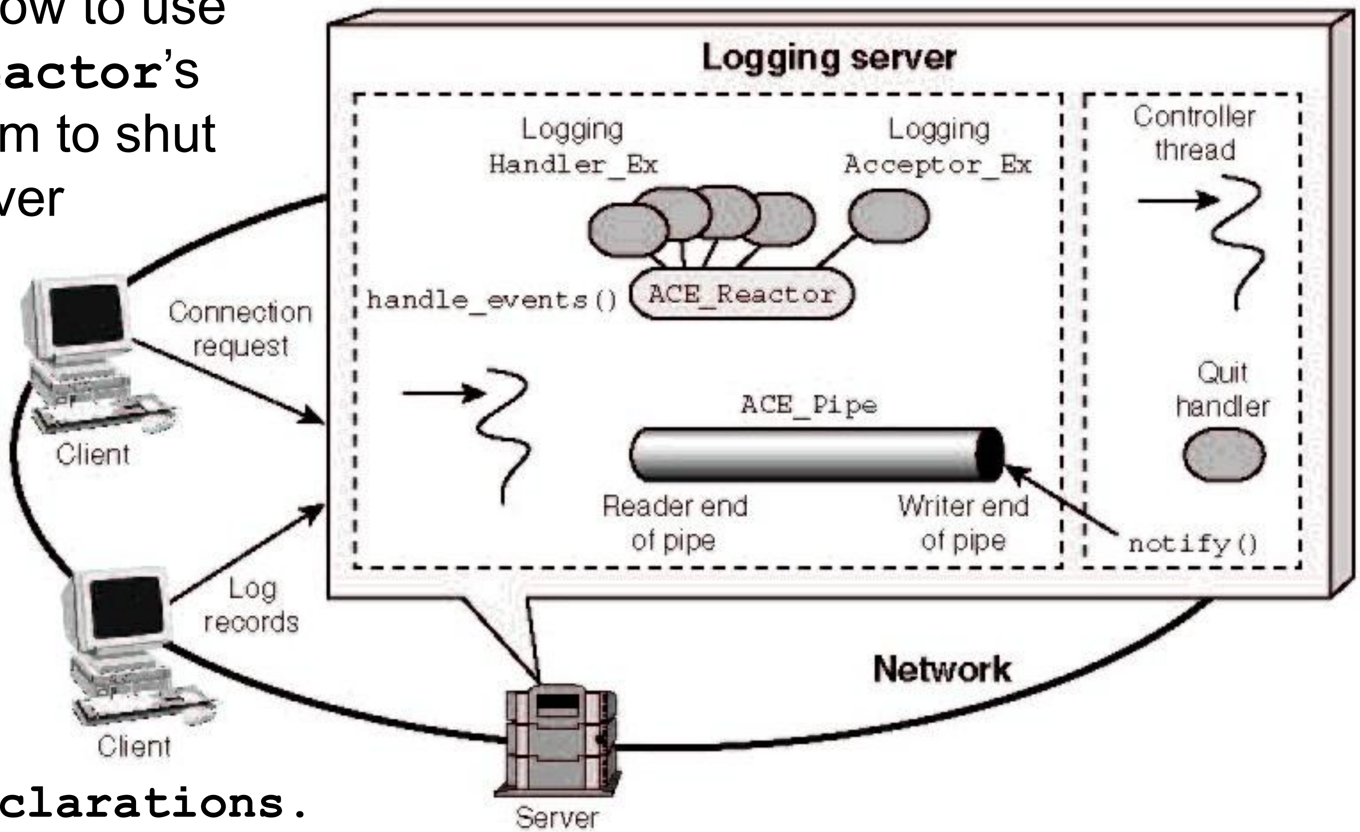
- **`ACE_Token`** is a lock whose interface is compatible with other ACE synchronization wrapper facades, such as **`ACE_Thread_Mutex`** or **`ACE_RW_Mutex`**
- It has the following capabilities:
  - It implements recursive mutex semantics
  - Each **`ACE_Token`** maintains two ordered lists that are used to queue high- & low-priority threads waiting to acquire the token
  - Threads requesting the token using **`ACE_Token::acquire_write()`** are kept in the high-priority list & take precedence over threads that call **`ACE_Token::acquire_read()`**, which are kept in the low-priority list
  - Within a priority list, threads that are blocked awaiting to acquire a token are serviced in either **`FIFO`** or **`LIFO`** order according to the current queueing strategy as threads release the token
  - The **`ACE_Token`** queueing strategy can be obtained or set via calls to **`ACE_Token::queueing_strategy()`** & defaults to **`FIFO`**, which ensures the fairness among waiting threads
  - In contrast, UNIX International & Pthreads mutexes don't strictly enforce any particular thread acquisition ordering

Sidebar: The ACE_Token Class (2/2)

- For applications that don't require strict **`FIFO`** ordering, the **`ACE_Token LIFO`** strategy can improve performance by maximizing CPU cache affinity.
- The **`ACE_Token::sleep_hook()`** hook method is invoked if a thread can't acquire a token immediately
  - This method allows a thread to release any resources it's holding before it waits to acquire the token, thereby avoiding deadlock, starvation, & unbounded priority inversion
- **`ACE_Select_Reactor`** uses an **`ACE_Token`**-derived class named **`ACE_Select_Reactor_Token`** to synchronize access to a reactor
  - Requests to change the internal states of a reactor use **`ACE_Token::acquire_write()`** to ensure other waiting threads see the changes as soon as possible
- **`ACE_Select_Reactor_Token`** overrides its **`sleep_hook()`** method to notify the reactor of pending threads via its notification mechanism

Using the ACE_Select_Reactor Class (1/4)

- This example show how to use the **ACE_Select_Reactor**'s **notify()** mechanism to shut down the logging server cleanly




```
// Forward declarations.
ACE_THR_FUNC_RETURN controller (void *);
ACE_THR_FUNC_RETURN event_loop (void *);

typedef Reactor_Logging_Server<Logging_Acceptor_Ex>
          Server_Logging_Daemon;

```

Using the ACE_Select_Reactor Class (2/4)

```
int main (int argc, char *argv[]) {
  ACE_Select_Reactor select_reactor;
  ACE_Reactor reactor (&select_reactor);

  Server_Logging_Daemon *server = 0;
  ACE_NEW_RETURN (server,
                  Server_Logging_Daemon (argc, argv, &reactor),
                  1);
  ACE_Thread_Manager::instance()->spawn (event_loop, &reactor);
  ACE_Thread_Manager::instance()->spawn (controller, &reactor);
  return ACE_Thread_Manager::instance ()->wait ();
}

static ACE_THR_FUNC_RETURN event_loop (void *arg) {
  ACE_Reactor *reactor = ACE_static_cast (ACE_Reactor *, arg);

  reactor->owner (ACE_OS::thr_self ());
  reactor->run_reactor_event_loop ();
```

Ensure we get the `ACE_Select_Reactor`

Barrier synchronization

Become “owner” (only needed for `ACE_Select_Reactor`)

Using the ACE_Select_Reactor Class (3/4)

Runs in a separate thread of control

```
static ACE_THR_FUNC_RETURN controller (void *arg) {
  ACE_Reactor *reactor = ACE_static_cast (ACE_Reactor *, arg);
  Quit_Handler *quit_handler = 0;
  ACE_NEW_RETURN (quit_handler, Quit_Handler (reactor), 0);

  for (;;) {
    std::string user_input;
    std::getline (cin, user_input, '\n');
    if (user_input == "quit") {
      reactor->notify (quit_handler);
      break;
    }
  }
  return 0;
}
```

Use the notify pipe to wakeup the reactor & inform it to shut down by calling `handle_exception()`

Using the ACE_Select_Reactor Class (4/4)

```
class Quit_Handler : public ACE_Event_Handler {
public:
  Quit_Handler (ACE_Reactor *r): ACE_Event_Handler (r) {}

  virtual int handle_exception (ACE_HANDLE) {
    reactor ()->end_reactor_event_loop ();
    return -1;
  }

  virtual int handle_close (ACE_HANDLE, ACE_Reactor_Mask)
  {
    delete this;
    return 0;
  }

private:

  // Private destructor ensures dynamic allocation.
  virtual ~Quit_Handler () {}
};
```

Trigger call to handle_close() method

It's ok to "delete this" in this context

Sidebar: Avoiding Reactor Notification Deadlock

- The ACE Reactor framework's notification mechanism enables a reactor to
  - Process an open-ended number of event handlers
  - Unblock from its event loop
- By default, the reactor notification mechanism is implemented with a bounded buffer & **`notify()`** uses a blocking send call to insert notifications into the queue
- A deadlock can therefore occur if the buffer is full & **`notify()`** is called by a **`handle_*()`** method of an event handler
- There are several ways to avoid such deadlocks:
  - ***Pass a timeout to the `notify()` method***
    - This solution pushes the responsibility for handling buffer overflow to the thread that calls **`notify()`**
  - ***Design the application so that it doesn't generate calls to `notify()` faster than a reactor can process them***
    - This is ultimately the best solution, though it requires careful analysis of program behavior

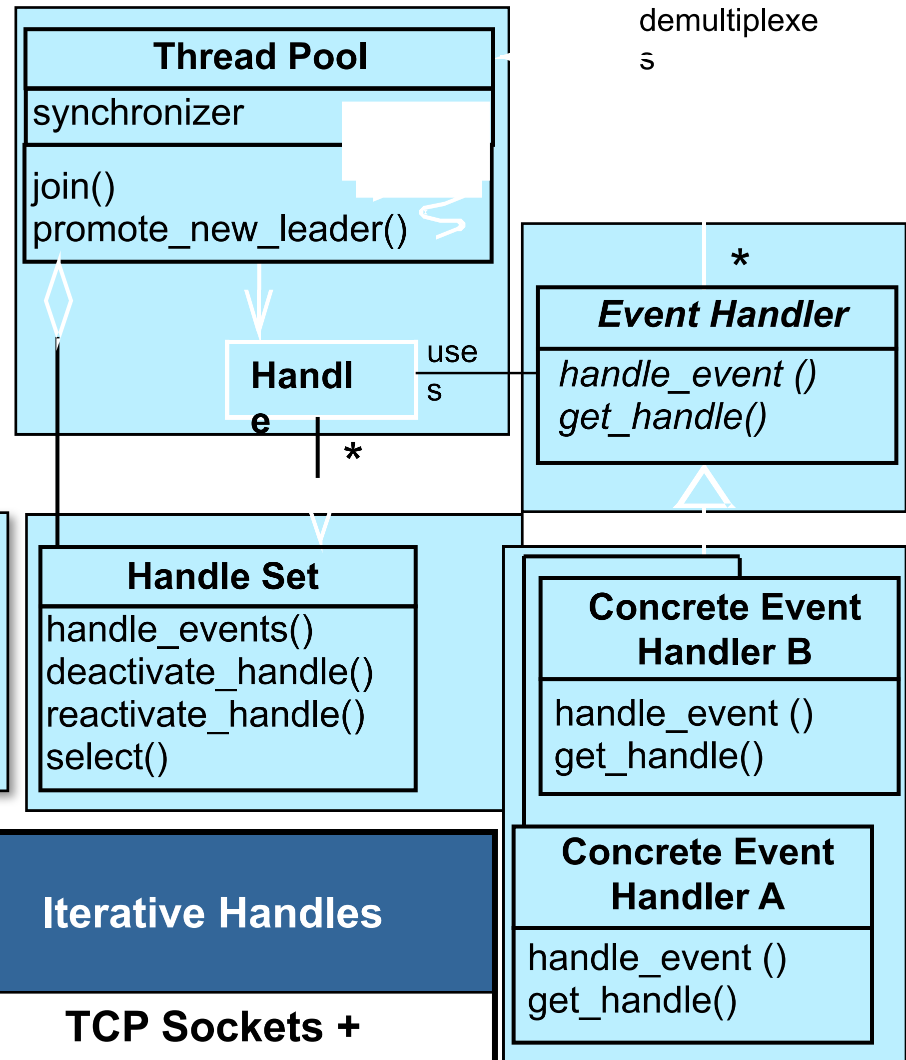
Sidebar: Enlarging ACE_Select_Reactor's Notifications

- In some situations, it's possible that a notification queued to an **`ACE_Select_Reactor`** won't be delivered until after the desired event handler is destroyed
- This delay stems from the time window between when the **`notify()`** method is called & the time when the reactor reacts to the notification pipe, reads the notification information from the pipe, & dispatches the associated callback
- Although application developers can often work around this scenario & avoid deleting an event handler while notifications are pending, it's not always possible to do so
- ACE offers a way to change the **`ACE_Select_Reactor`** notification queueing mechanism from an **`ACE_Pipe`** to a user-space queue that can grow arbitrarily large
- This alternate mechanism offers the following benefits:
  - Greatly expands the queueing capacity of the notification mechanism, also helping to avoid deadlock
  - Allows the **`ACE_Reactor::purge_pending_notifications()`** method to scan the queue & remove desired event handlers
- To enable this feature, add **`#define ACE_HAS_REACTOR_NOTIFICATION_QUEUE`** to your **`$ACE_ROOT/ace/config.h`** file & rebuild ACE
- This option is not enabled by default because the additional dynamic memory allocation required may be prohibitive for high-performance or embedded systems

The Leader/Followers Pattern

The Leader/Followers architectural pattern (P2) provides an efficient concurrency model where multiple threads take turns sharing event sources to detect, demux, dispatch, & process service requests that occur on the event sources

This pattern eliminates the need for—& the overhead of—a separate Reactor thread & synchronized request queue used in the Half-Sync/Half-Async pattern



Handles / Handle Sets	Concurrent Handles	Iterative Handles
Concurrent Handle Sets	UDP Sockets + `WaitForMultipleObjects()`	TCP Sockets + `WaitForMultpleObjects()`
Iterative Handle Sets	UDP Sockets + `select()/poll()`	TCP Sockets + `select()/poll()`

Leader/Followers Pattern Dynamics

Thread 1 **Thread 2** **: Thread Pool** **: Handle Set** **: Concrete Event Handler**

1. *Leader thread demuxing*
2. *Follower thread promotion*
3. *Event handler demuxing & event processing*
4. *Rejoining the thread pool*

join()

handle_events()

join()

event

handle_event()

deactivate_handle()

thread 2 sleeps until it becomes the leader

promote_new_leader()

thread 2 waits for a new event, thread 1 processes current event

handle_events()

reactivate_handle()

join()

event

handle_event()

thread 1 sleeps until it becomes the leader

deactivate_handle()

Pros & Cons of Leader/Followers Pattern

This pattern provides two **benefits**:

- ***Performance enhancements***
  - This can improve performance as follows:
    - It enhances CPU cache affinity & eliminates the need for dynamic memory allocation & data buffer sharing between threads
    - It minimizes locking overhead by not exchanging data between threads, thereby reducing thread synchronization
    - It can minimize priority inversion because no extra queueing is introduced in the server
    - It doesn’t require a context switch to handle each event, reducing dispatching latency
- ***Programming simplicity***
  - The Leader/Follower pattern simplifies the programming of concurrency models where multiple threads can receive requests, process responses, & demultiplex connections using a shared handle set

This pattern also incur **liabilities**:

- ***Implementation complexity***
  - The advanced variants of the Leader/ Followers pattern are hard to implement
- ***Lack of flexibility***
  - In the Leader/ Followers model it is hard to discard or reorder events because there is no explicit queue
- ***Network I/O bottlenecks***
  - The Leader/Followers pattern serializes processing by allowing only a single thread at a time to wait on the handle set, which could become a bottleneck because only one thread at a time can demultiplex I/O events

The ACE_TP_Reactor Class (1/2)

Motivation

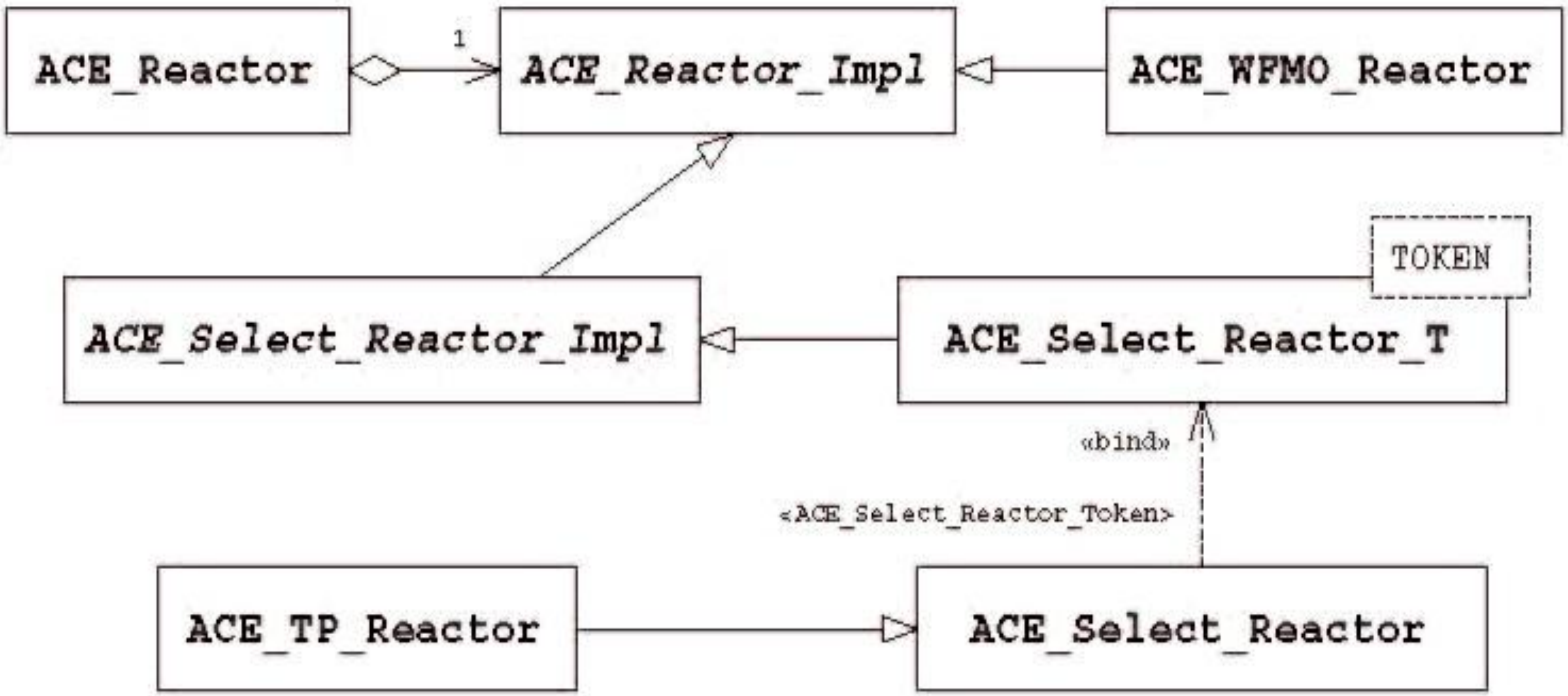
- Although **`ACE_Select_Reactor`** is flexible, it's somewhat limited in multithreaded applications because only the owner thread can **`ACE_Select_Reactor`** call its **`handle_events()`** method
- One way to solve this problem is to spawn multiple threads & run the event loop of a separate instance of **`ACE_Select_Reactor`** in each of them
  - This design can be hard to program, however, since it requires developers to implement a proxy that partitions event handlers evenly between the reactors to divide the load evenly across threads
- The **`ACE_TP_Reactor`** is intended to simplify the use of the ACE Reactor in multithreaded applications

The ACE_TP_Reactor Class (2/2)

Class Capabilities

- This class inherits from **`ACE_Select_Reactor`** & implements the **`ACE_Reactor`** interface & uses the Leader/Followers pattern to provide the following capabilities:
  - A pool of threads can call its **`handle_events()`** method, which can improve scalability by handling events on multiple handles concurrently
    - It prevents multiple I/O events from being dispatched to the same event handler simultaneously in different thread
    - This constraint preserves the **`ACE_Select_Reactor`**'s I/O dispatching behavior, alleviating the need to add synchronization locks to a handler's I/O processing
    - After a thread obtains a set of active handles from **`select()`**, the other reactor threads dispatch from that handle set instead of calling **`select()`** again

The ACE_TP_Reactor Class API

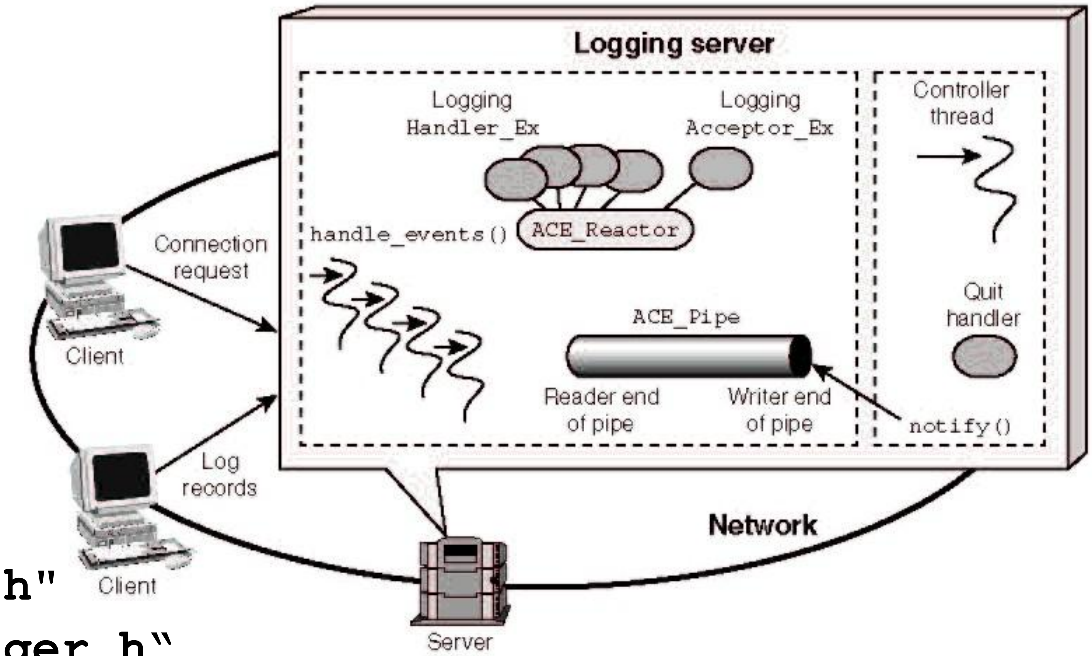


Pros & Cons of ACE_TP_Reactor

- Compared to other thread pool models, such as the half-sync/half-async model, **`ACE_TP_Reactor`** keeps all event processing local to the thread that dispatches the handler, which yields the following benefits:
  - It enhances CPU cache affinity & eliminates the need to allocate memory dynamically & share data buffers between threads
  - It minimizes locking overhead by not exchanging data between threads
  - It minimizes priority inversion since no extra queueing is used
  - It doesn't require a context switch to handle each event, which reduces latency
- Given the added capabilities of the **`ACE_TP_Reactor`**, here are two reasons why you would still use the **`ACE_Select_Reactor`**:
  - ***Less overhead*** – While **`ACE_Select_Reactor`** is less powerful than the **`ACE_TP_Reactor`** it also incurs less time & space overhead
    - Moreover, single-threaded applications can instantiate the **`ACE_Select_Reactor_T`** template with an **`ACE_Noop_Token`**-based token to eliminate the internal overhead of acquiring & releasing tokens completely
  - ***Implicit serialization*** – **`ACE_Select_Reactor`** is particularly useful when explicitly writing serialization code at the application-level is undesirable
    - e.g., application programmers who are unfamiliar with synchronization techniques may prefer to let the **`ACE_Select_Reactor`** serialize their event handling, rather than using threads & adding locks in their application code

Using the ACE_TP_Reactor Class (1/2)

- This example revises the **`ACE_Select_Reactor`** example to spawn a pool of threads that share the **`Reactor_Logging_Server`**'s I/O handles




```
#include "ace/streams.h"
#include "ace/Reactor.h"
#include "ace/TP_Reactor.h"
#include "ace/Thread_Manager.h"
#include "Reactor_Logging_Server.h"
#include <string>
// Forward declarations
ACE_THR_FUNC_RETURN controller (void *);
ACE_THR_FUNC_RETURN event_loop (void *);

typedef Reactor_Logging_Server<Logging_Acceptor_Ex>
        Server_Logging_Daemon;

```

Note reuse

Using the ACE_TP_Reactor Class (2/2)

```
int main (int argc, char *argv[]) {
  const size_t N_THREADS = 4;
  ACE_TP_Reactor tp_reactor;
  ACE_Reactor reactor (&tp_reactor);
  auto_ptr<ACE_Reactor> delete_instance
    (ACE_Reactor::instance (&reactor));

  Server_Logging_Daemon *server = 0;
  ACE_NEW_RETURN (server,
                  Server_Logging_Daemon (argc, argv,
                    ACE_Reactor::instance ()), 1);
  ACE_Thread_Manager::instance ()->spawn_n
    (N_THREADS, event_loop, ACE_Reactor::instance ());
  ACE_Thread_Manager::instance ()->spawn
    (controller, ACE_Reactor::instance ());
  return ACE_Thread_Manager::instance ()->wait ();
}
```

Ensure we get the ACE_TP_Reactor

Spawn multiple threads

The ACE_WFMO_Reactor Class (1/2)

Motivation

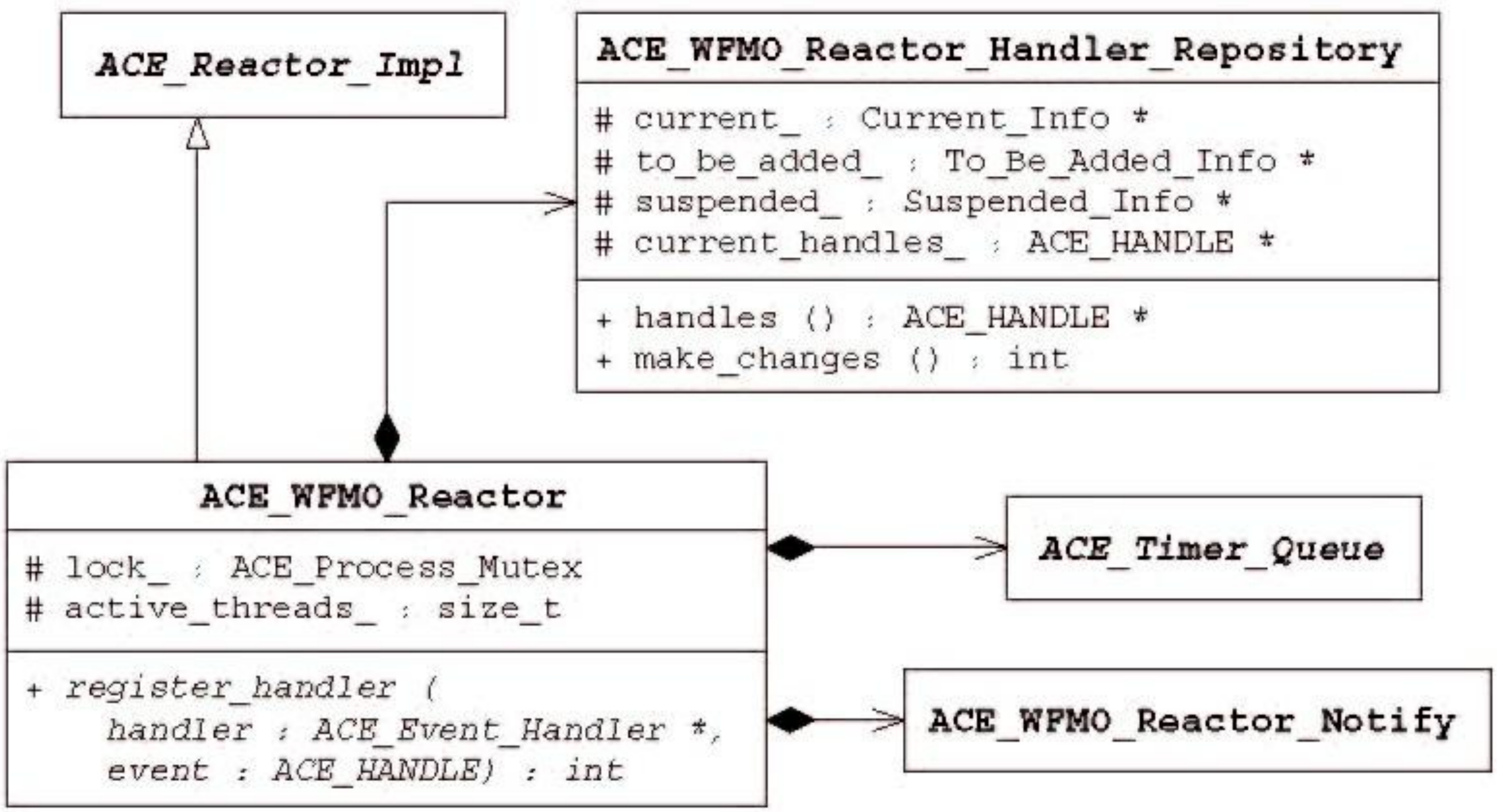
- Although **`select()`** is widely available, it's not always the best demuxer:
  - On UNIX platforms, it only supports demuxing of I/O handles
  - On Windows, **`select()`** only supports demultiplexing of socket handles
  - It can only be called by one thread at a time for a particular set of I/O handles, which can degrade potential parallelism
- **`ACE_WFMO_Reactor`** uses **`WaitForMultipleObjects()`** to alleviate these problems & is the default **`ACE_Reactor`** implementation on Windows

The ACE_WFMO_Reactor Class (2/2)

Class Capabilities

- This class is an implementation of the **`ACE_Reactor`** interface that also provides the following capabilities:
  - It enables a pool of threads to call its **`handle_events()`** method concurrently
  - It allows applications to wait for socket I/O events & scheduled timers, similar to the **`select()`**-based reactors, & also integrates event demultiplexing & dispatching for all event types that **`WaitForMultipleObjects()`** supports

The ACE_WFMO_Reactor Class API



Sidebar: The WaitForMultipleObjects() Function

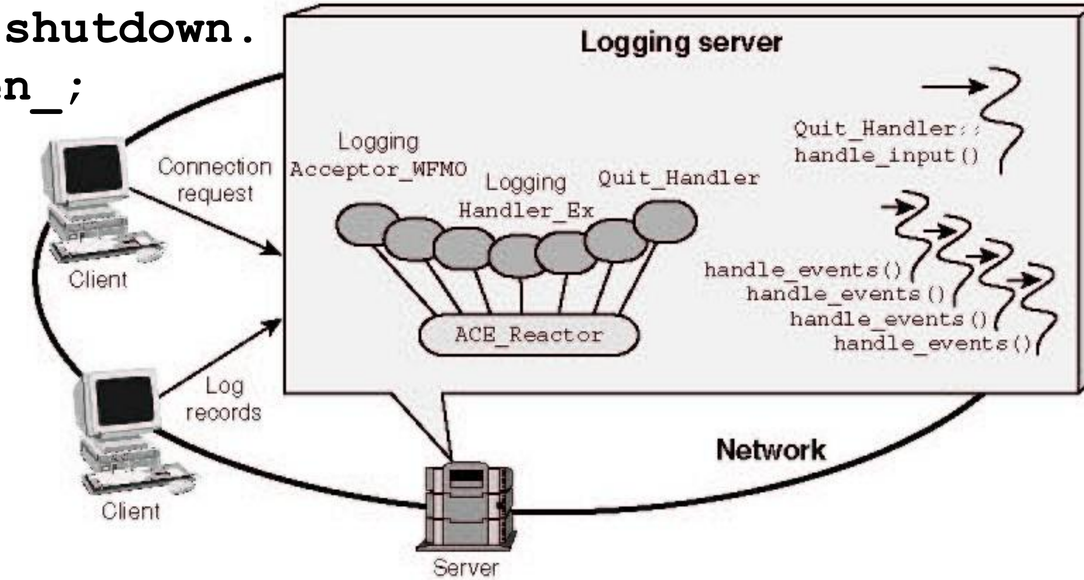
- The Windows **`WaitForMultipleObjects()`** event demultiplexer function is similar to **`select()`**
- It blocks on an array of up to 64 handles until one or more of them become active (which is known as being “signaled” in Windows terminology) or until the interval in its timeout parameter elapses
- It can be programmed to return to its caller when either any one or more of the handles becomes active or all the handles become active
- In either case, it returns the index of the lowest active handle in the caller-specified array of handles
- Unlike the **`select()`** function, which only demultiplexes I/O handles, **`WaitForMultipleObjects()`** can wait for many types of Windows objects, including a thread, process, synchronizer (e.g., event, semaphore, or mutex), change notification, console input, & timer

Sidebar: Why ACE_WFMO_Reactor is Windows Default

- The **`ACE_WFMO_Reactor`** is the default implementation of the **`ACE_Reactor`** on Windows platforms for the following reasons:
  - It lends itself more naturally to multithreaded processing, which is common on Windows
  - **`ACE_WFMO_Reactor`** was developed before **`ACE_TP_Reactor`** & was the first reactor to support multithreaded event handling
  - Applications often use signalable handles in situations where a signal may have been used on POSIX (e.g., child process exit) & these events can be dispatched by **`ACE_WFMO_Reactor`**
  - It can handle a wider range of events than the **`ACE_Select_Reactor`**, which can only handle socket & timer events on Windows.
  - It's easily integrated with **`ACE_Proactor`** event handling

Using the ACE_WFMO_Reactor Class (1/5)

```
class Quit_Handler : public ACE_Event_Handler {
private:
  // Keep track of when to shutdown.
  ACE_Manual_Event quit_seen_;
public:
```




```
Quit_Handler (ACE_Reactor *r): ACE_Event_Handler (r) {
  SetConsoleMode (ACE_STDIN,
                  ENABLE_LINE_INPUT | ENABLE_ECHO_INPUT
                  | ENABLE_PROCESSED_INPUT);
  if (reactor ()->register_handler
      (this, quit_seen_.handle ()) == -1
      || ACE_Event_Handler::register_stdin_handler
         (this, r, ACE_Thread_Manager::instance ()) ==
-1)
    r->end_reactor_event_loop ();
```

This method only works on Windows

Sidebar: ACE_Manual_Event & ACE_Auto_Event

- ACE provides two synchronization wrapper facade classes : **`ACE_Manual_Event`** & **`ACE_Auto_Event`**
- These classes allow threads in a process to wait on an event or inform other threads about the occurrence of a specific event in a thread-safe manner
- On Windows these classes are wrapper facades around native event objects, whereas on other platforms ACE emulates the Windows event object facility
- Events are similar to condition variables in the sense that a thread can use them to either signal the occurrence of an application-defined event or wait for that event to occur
- Unlike stateless condition variables, a signaled event remains set until a class-specific action occurs
  - e.g., an **`ACE_Manual_Event`** remains set until it is explicitly reset & an **`ACE_Auto_Event`** remains set until a single thread waits on it
- These two classes allow users to control the number of threads awakened by signaling operations, & allows an event to indicate a state transition, even if no threads are waiting at the time the event is signaled
- Events are more expensive than mutexes, but provide better control over thread scheduling
- Events provide a simpler synchronization mechanism than condition variables
- Condition variables are more useful for complex synchronization activities, however, since they enable threads to wait for arbitrary condition expressions

Using the ACE_WFMO_Reactor Class (2/5)

```
virtual int handle_input (ACE_HANDLE h) {
   CHAR user_input[BUFSIZ];
   DWORD count;
   if (!ReadFile (h, user_input, BUFSIZ, &count, 0)) return
-1;
   user_input[count] = '\0';
   if (ACE_OS_String::strncmp (user_input, "quit", 4) == 0)
     return -1;
   return 0;
 }
virtual int handle_close (ACE_HANDLE, ACE_Reactor_Mask)
 { quit_seen_.signal (); return 0; }
```

This is a Windows-specific function

This hook method is called when a handle is signaled

```
virtual int handle_signal (int, siginfo_t *, ucontext_t *)
 { reactor ()->end_reactor_event_loop (); return 0; }

~Quit_Handler () {
  ACE_Event_Handler::remove_stdin_handler
    (reactor (), ACE_Thread_Manager::instance ());
  reactor ()->remove_handler (quit_seen_.handle (),
                              ACE_Event_Handler::DONT_CALL);
```

Using the ACE_WFMO_Reactor Class (3/5)

```
class Logging_Event_Handler_WFMO
    : public Logging_Event_Handler_Ex {
public:
  Logging_Event_Handler_WFMO (ACE_Reactor *r)
    : Logging_Event_Handler_Ex (r) {}
```

We need a lock since the `ACE_WFMO_Reactor` doesn't suspend handles…

```
protected:
  int handle_input (ACE_HANDLE h) {
    ACE_GUARD_RETURN (ACE_SYNCH_MUTEX, monitor, lock_, -1);
    return logging_handler_.log_record ();
  }

  ACE_Thread_Mutex lock_; // Serialize threads in thread
pool.
};
```

Sidebar: Why ACE_WFMO_Reactor Doesn't Suspend Handlers (1/2)

- The **`ACE_WFMO_Reactor`** doesn't implement a handler suspension protocol internally to minimize the amount of policy imposed on application classes
- In particular, multithreaded applications can process events more efficiently when doing so doesn't require inter-event serialization, e.g., when receiving UDP datagrams
- This behavior isn't possible in the **`ACE_TP_Reactor`** because of the semantic differences in the functionality of the following OS event demultiplexing mechanisms:
  - **`WaitForMultipleObjects()`**
    - When demultiplexing a socket handle's I/O event, one **`ACE_WFMO_Reactor`** thread will obtain the I/O event mask from **`WSAEnumNetworkEvents()`**, & the OS atomically clears that socket's internal event mask
    - Even if multiple threads demultiplex the socket handle simultaneously, only one obtains the I/O event mask & will dispatch the handler
    - The dispatched handler must take some action that re-enables demultiplexing for that handle before another thread will dispatch it
  - **`select()`**
    - There's no automatic OS serialization for **`select()`**
    - If multiple threads were allowed to see a ready-state socket handle, they would all dispatch it, yielding unpredictable behavior at the **`ACE_Event_Handler`** layer & reduced performance due to multiple threads all working on the same handle

Sidebar: Why ACE_WFMO_Reactor Doesn’t Suspend Handlers (2/2)

- It's important to note that the handler suspension protocol can't be implemented in the application event handler class when it's used in conjunction with the **`ACE_WFMO_Reactor`**
- This is because suspension requests are queued & aren't acted on immediately
- A handler could therefore receive upcalls from multiple threads until the handler was actually suspended by the **`ACE_WFMO_Reactor`**
- The **`Logging_Event_Handler_WFMO`** class illustrates how to use mutual exclusion to avoid race conditions in upcalls

Using the ACE_WFMO_Reactor Class (4/5)

```
class Logging_Acceptor_WFMO : public Logging_Acceptor_Ex {
public:
  Logging_Acceptor_WFMO
    (ACE_Reactor *r = ACE_Reactor::instance ())
    : Logging_Acceptor_Ex (r) {}

protected:
  virtual int handle_input (ACE_HANDLE) {
    Logging_Event_Handler_WFMO *peer_handler = 0;
    ACE_NEW_RETURN (peer_handler,
                    Logging_Event_Handler_WFMO (reactor ()),
-1);
    if (acceptor_.accept (peer_handler->peer ()) == -1)
    { delete peer_handler; return -1; }
    else if (peer_handler->open () == -1)
    { peer_handler->handle_close (); return -1; }
    return 0;
  }
};
```

Note the canonical (common) form of this hook method

Using the ACE_WFMO_Reactor Class (5/5)

Main program

```
ACE_THR_FUNC_RETURN event_loop (void *); // Forward
declaration.

typedef Reactor_Logging_Server<Logging_Acceptor_WFMO>
        Server_Logging_Daemon;

int main (int argc, char *argv[]) {
  const size_t N_THREADS = 4;
  ACE_WFMO_Reactor wfmo_reactor;
  ACE_Reactor reactor (&wfmo_reactor);

  Server_Logging_Daemon *server = 0;
  ACE_NEW_RETURN
    (server, Server_Logging_Daemon (argc, argv, &reactor), -1);
  Quit_Handler quit_handler (&reactor);

  ACE_Thread_Manager::instance ()->spawn_n
    (N_THREADS, event_loop, &reactor);
  return ACE_Thread_Manager::instance ()->wait ();
```

Ensure we get the `ACE_WFMO_Reactor`

Constructor registers with reactor

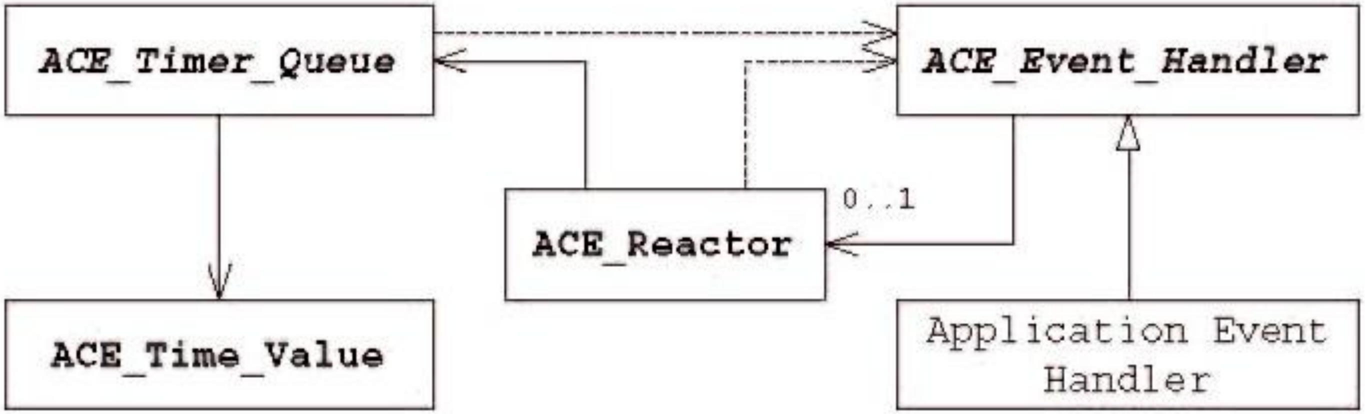
Barrier synchronization

Other Reactors Supported By ACE

- Over the previous decade, ACE's use in new environments has yielded new requirements for event-driven application support
  - e.g., GUI integration is an important area due to new GUI toolkits & event loop requirements
- The following new Reactor implementations were made easier due to the ACE Reactor framework's modular design:

ACE Class	Description
ACE_Dev_Poll_Reactor	Uses the /dev/poll or /dev/epoll demultiplexer. It's designed to be more scalable than select()-based reactors.
ACE_Priority_Reactor	Dispatches events in developer-assigned priority order.
ACE_XtReactor	Integrates ACE with the X11 Toolkit.
ACE_FlReactor	Integrates ACE with the Fast Light (FL) GUI framework.
ACE_QtReactor	Integrates ACE with the Qt GUI toolkit.
ACE_TkReactor	Integrates ACE with the TCL/Tk GUI toolkit.
ACE_Msg_WFMO_Reactor	Adds Windows message handling to ACE_WFMO_Reactor.

Challenges of Using Frameworks Effectively



Now that we've examined the ACE Reactor frameworks, let's examine the challenges of using frameworks in more depth

- Determine if a framework applies to the problem domain & whether it has sufficient quality
- Evaluating the time spent learning a framework outweighs the time saved by reuse
- Learn how to debug applications written using a framework
- Identify the performance implications of integration application logic into a framework
- Evaluate the effort required to develop a new framework

www.cs.wustl.edu/~schmidt/PDF/Queue-04.pdf

Determining Framework Applicability & Quality

Applicability

- Have domain experts & product architects identify common functionality with other domains & conduct trade study of COTS frameworks to address domain-specific & -independent functionality during the design phase
- Conduct pilot studies that apply COTS frameworks to develop representative prototype applications as part of an iterative development approach,
  - e.g., the Spiral model or eXtreme Programming (XP)

Quality

- Will the framework allow applications to cleanly decouple the callback logic from the rest of the software?
- Can applications interact with the framework via a narrow & well defined set of interfaces & facades?
- Does the framework document all the API's that are used by applications to interact with the framework, e.g., does it define pre-conditions & post-conditions of callback methods via contracts?
- Does the framework explicitly specify the startup, shutdown, synchronization, & memory management contracts available for the clients?

Evaluating Economics of Frameworks

- Determining effective framework cost metrics, which measure the savings of reusing framework components vs. building applications from scratch
- Conducting cost/effort estimations, which is the activity of accurately forecasting the cost of buying, building, or adapting a particular framework
- Perform investment analysis & justification, which determines the benefits of applying frameworks in terms of return on investment
- COCOMO 2.0 is a widely used software cost model estimator that can help to predict the effort for new software activities
- The estimates from these types of models can be used as a basis of determining the savings that could be incurred by using frameworks
- A challenge confronting software development organizations, however, is that many existing software cost/effort estimation methodologies are not well calibrated to handle reusable frameworks or standards-based frameworks that provide subtle advantages, such as code portability or refactoring

Effective Framework Debugging Techniques

- Track lifetimes of objects by monitoring their reference counts
- Monitor the internal request queue lengths & buffer sizes maintained by the framework
- Monitor the status of the network connections in distributed systems
- Track the activities of designated threads in a thread pool
- Trace the SQL statements issued by servers to backend databases
- Identify priority inversions in real-time systems
- Track authentication & authorization activities
- Perform design reviews early in application development process to convey interactions between the framework & the application logic
- Conduct code inspections that focus on common mistakes, such as incorrectly applying memory ownership rules for pre-registered components with the frameworks
- Select good automated debugging tools, such as Purify & Valgrind
- Develop automated regression tests

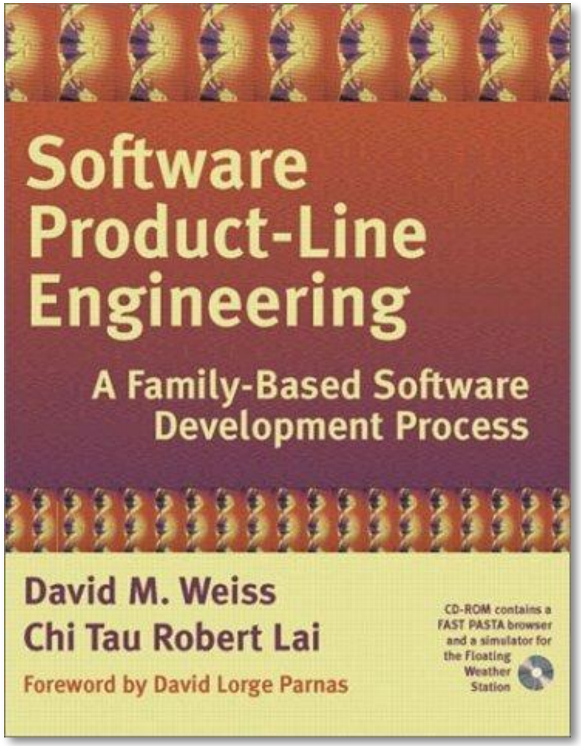
Identify Framework Time & Space Overheads

- **Event dispatching latency**
  - Time required to callback event handlers
- **Synchronization latency**
  - Time spent acquiring/releasing locks in the framework
- **Resource management latency**
  - Time spent allocation/releasing memory & other reusable resources
- **Framework functionality latency**
  - Time spent inside the framework for each operation
- **Dynamic & static memory overhead**
  - Run-time & disk space usage

- Conduct systematic engineering analysis to determine features & properties required from a framework
  - Determine the “sweet spot” of framework
- Develop test cases to empirically evaluate overhead associated with every feature & combination of features
  - Different domains have different requirements
- Locate third-party performance benchmarks & analysis to compare with data collected
  - Use google!

Evaluating Effort of Developing New Framework

- Perform commonality & variability analysis to determine
  - which classes should be fixed, thus defining the stable shape & usage characteristics of the framework
  - which classes should be extensible to support adaptation necessary to use the framework for new applications
- Determine the right protocols for startup & shutdown sequences of operations
- Develop right memory management & re-entrancy rules for the framework
- Develop the right set of (narrow) interfaces that can be used by the clients



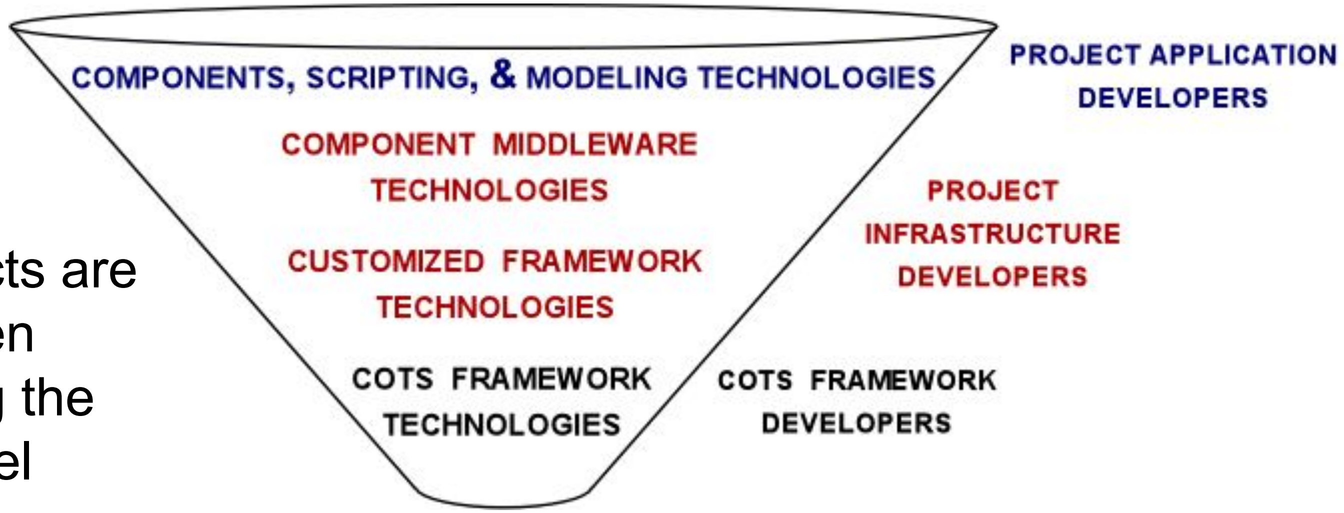
Knowledge of patterns is essential!

Challenges of Using Frameworks Effectively

Observations

- Frameworks are powerful, but hard to develop & use effectively by application developers
  - It's often better to use & customize COTS frameworks than to develop in-house frameworks
- Components are easier for application developers to use, but aren't as powerful or flexible as frameworks

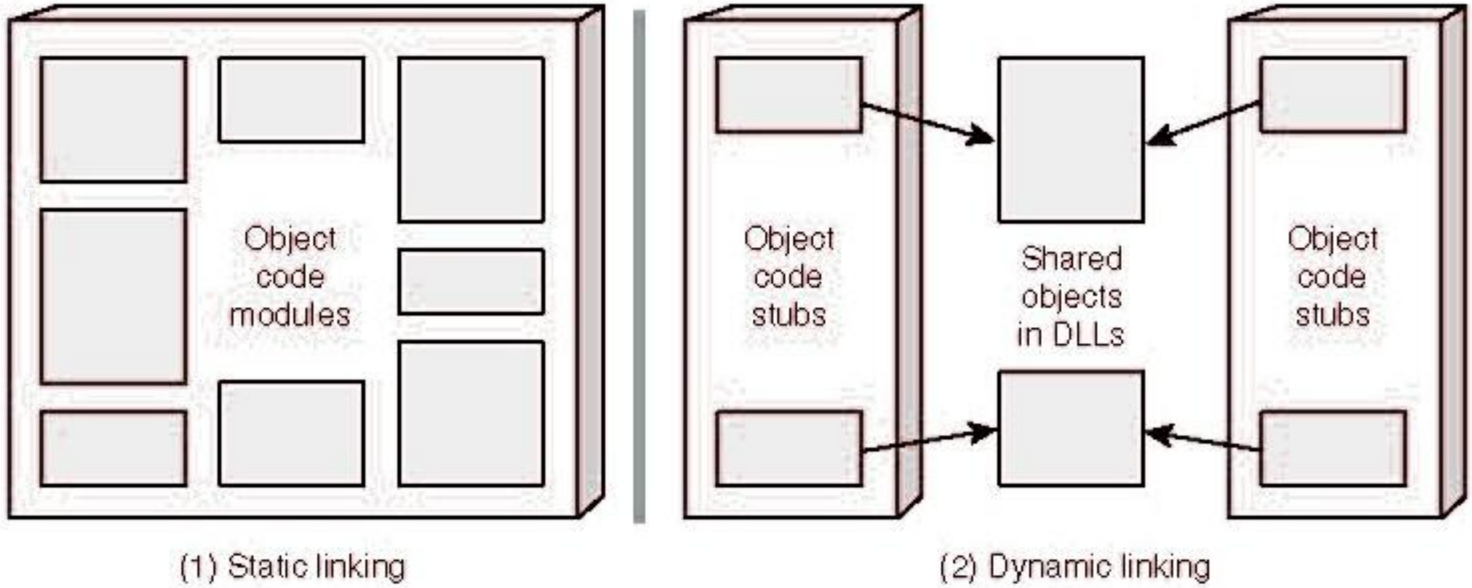
Successful projects are therefore often organized using the "funnel" model



Configuration Design Dimensions

- Networked applications can be created by configuring their constituent services together at various points of time, such as compile time, static link time, installation time, or run time
- This set of slides covers the following configuration design dimensions:
  - Static versus dynamic naming
  - Static versus dynamic linking
  - Static versus dynamic configuration

Static vs. Dynamic Linking & Configuration



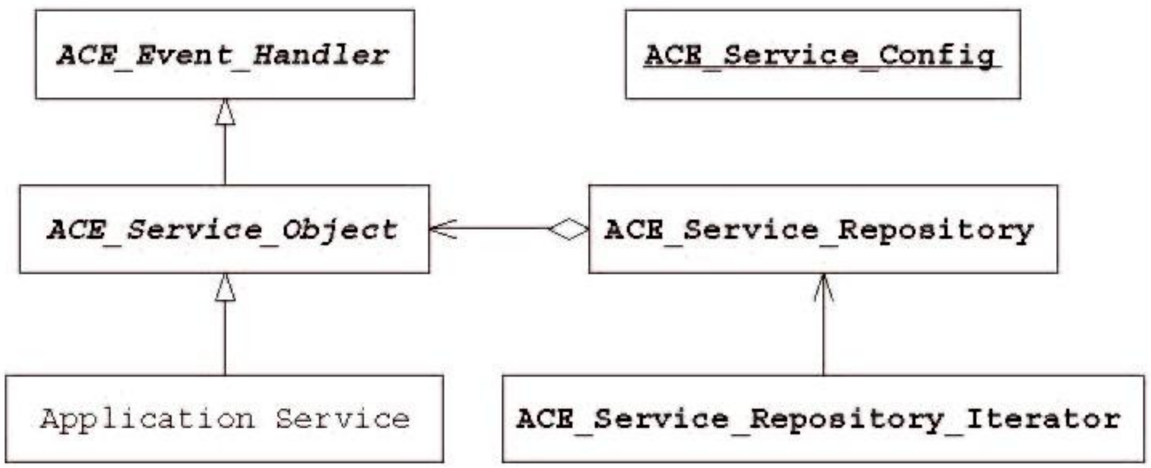
- Static linking creates a complete executable program by binding together all its object files at compile time and/or static link time
- It typically tradesoff increased runtime performance for larger executable sizes

- Dynamic linking loads object files into & unloads object files from the address space of a process when a program is invoked initially or updated at run time
- There are two general types of dynamic linking:
  - Implicit dynamic linking &
  - Explicit dynamic linking
- Dynamic linking can greatly reduce memory usage, though there are runtime overheads

The ACE Service Configuration Framework

- The ACE Service Configurator framework implements the Component Configurator pattern
- It allows applications to defer configuration & implementation decisions about their services until late in the design cycle
  - i.e., at installation time or runtime
- The Service Configurator supports the ability to activate services selectively at runtime regardless of whether they are linked statically or dynamically

- Due to ACE's integrated framework design, services using the ACE Service Configurator framework can also be dispatched by the ACE Reactor framework

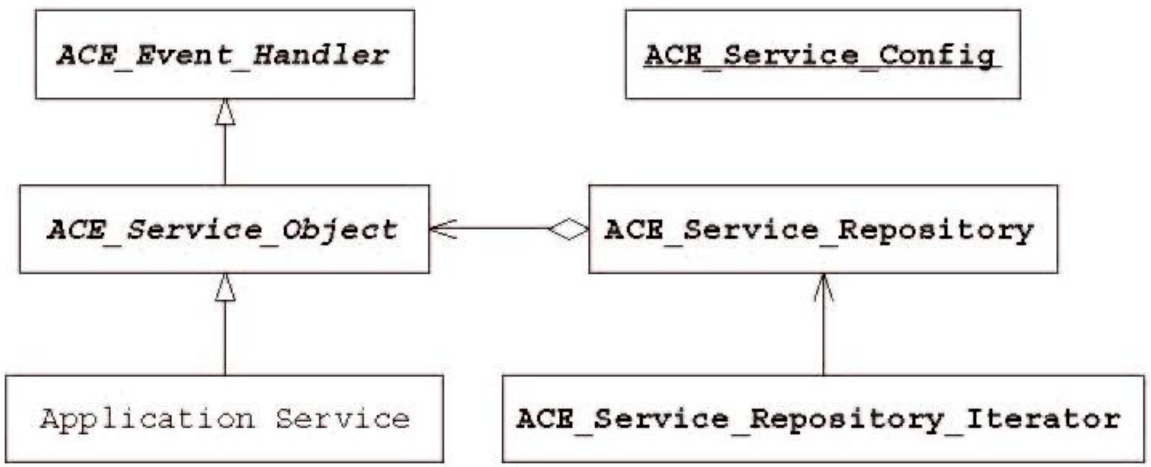


The ACE Service Configuration Framework

- The following classes are associated with the ACE Service Configurator framework

ACE Class	Description
ACE_Service_Object	Defines a uniform interface that the ACE Service Configurator framework uses to configure and control a service implementation. Control operations include initializing, suspending, resuming, and terminating a service.
ACE_Service_Repository	A central repository for all services managed using the ACE Service Configurator framework. It provides methods for locating, reporting on, and controlling all of an application's configured services.
ACE_Service_Repository_Iterator	A portable mechanism for iterating through all the services in a repository.
ACE_Service_Config	Provides an interpreter that parses and executes scripts specifying which services to (re)configure into an application (e.g., by linking and unlinking DLLs) and which services to suspend and resume.

- These classes are related as follows:



The Component Configurator Pattern

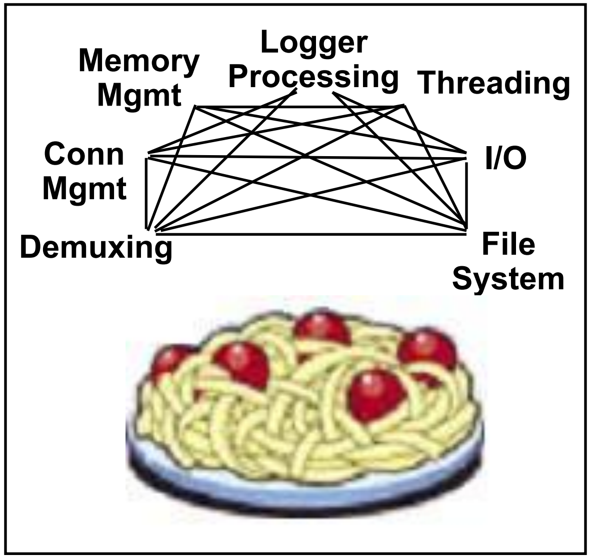
Context

- The implementation of certain application components depends on a variety of factors:
  - Certain factors are *static*, such as the number of available CPUs & operating system support for asynchronous I/O
  - Other factors are *dynamic*, such as system workload

Problem

Prematurely committing to a particular application component configuration is inflexible & inefficient:

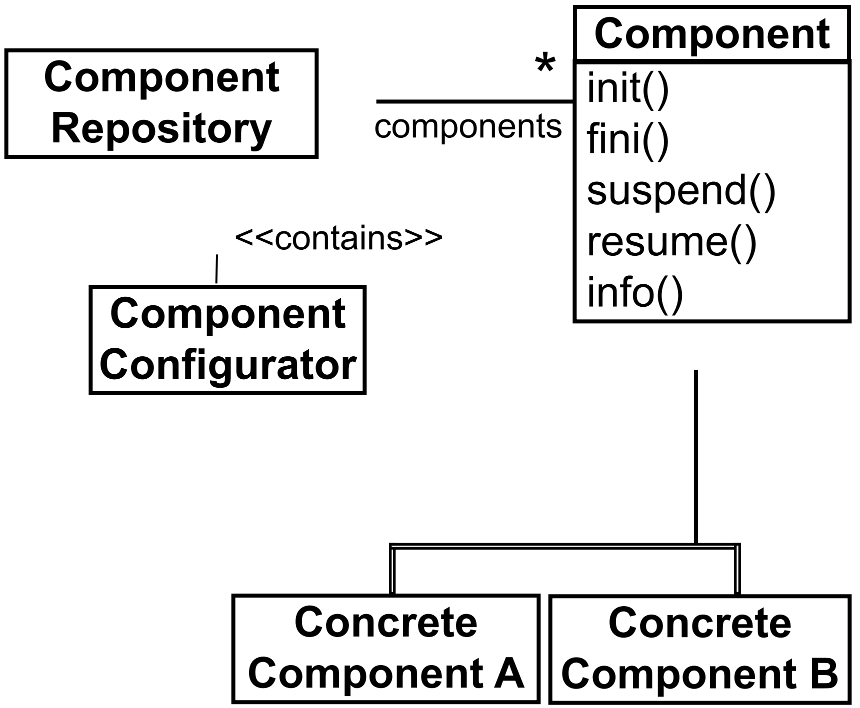
- No single application configuration is optimal for all use cases
- Certain design decisions cannot be made efficiently until run-time



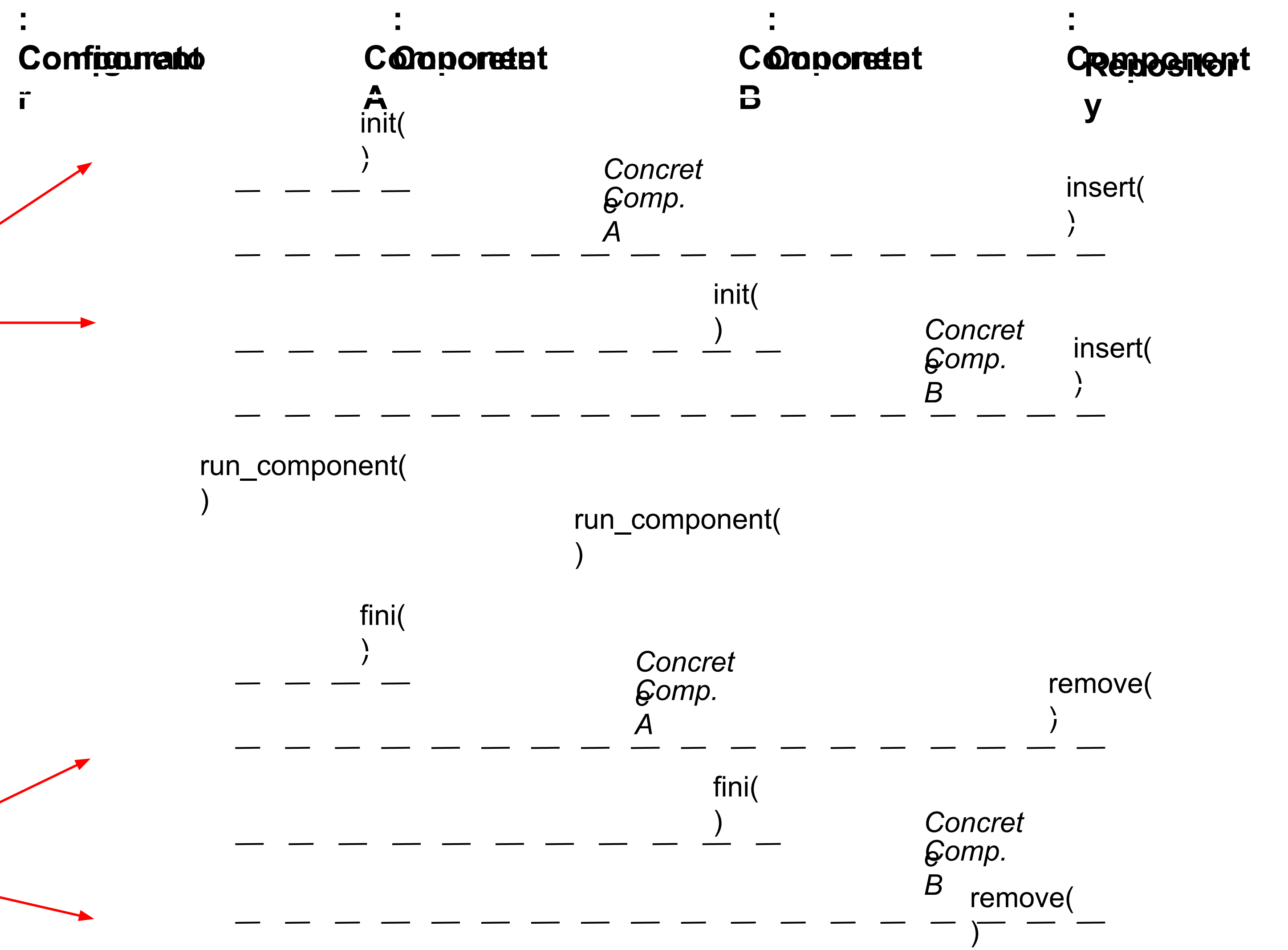
The Component Configurator Pattern

Solution

- Apply the *Component Configurator* design pattern (P2) to enhance server configurability
- This pattern allows an application to link & unlink its component implementations at run-time
- Thus, new & enhanced services can be added without having to modify, recompile, statically relink, or shut down & restart a running application



Component Configurator Pattern Dynamics



1. *Component initialization & dynamic linking*
2. *Component processing*
3. *Component termination & dynamic unlinking*

Pros & Cons of the Component Configurator Pattern

This pattern offers four **benefits**:

- ***Uniformity***
  - By imposing a uniform configuration & control interface to manage components
- ***Centralized administration***
  - By grouping one or more components into a single administrative unit that simplifies development by centralizing common component initialization & termination activities
- ***Modularity, testability, & reusability***
  - Application modularity & reusability is improved by decoupling component implementations from the manner in which the components are configured into processes
- ***Configuration dynamism & control***
  - By enabling a component to be dynamically reconfigured without modifying, recompiling, statically relinking existing code & without restarting the component or other active components with which it is collocated

This pattern also incurs **liabilities**:

- ***Lack of determinism & ordering dependencies***
  - This pattern makes it hard to determine or analyze the behavior of an application until its components are configured at run-time
- ***Reduced security or reliability***
  - An application that uses the Component Configurator pattern may be less secure or reliable than an equivalent statically-configured application
- ***Increased run-time overhead & infrastructure complexity***
  - By adding levels of abstraction & indirection when executing components
- ***Overly narrow common interfaces***
  - The initialization or termination of a component may be too complicated or too tightly coupled with its context to be performed in a uniform manner

The ACE_Service_Object Class (1/2)

Motivation

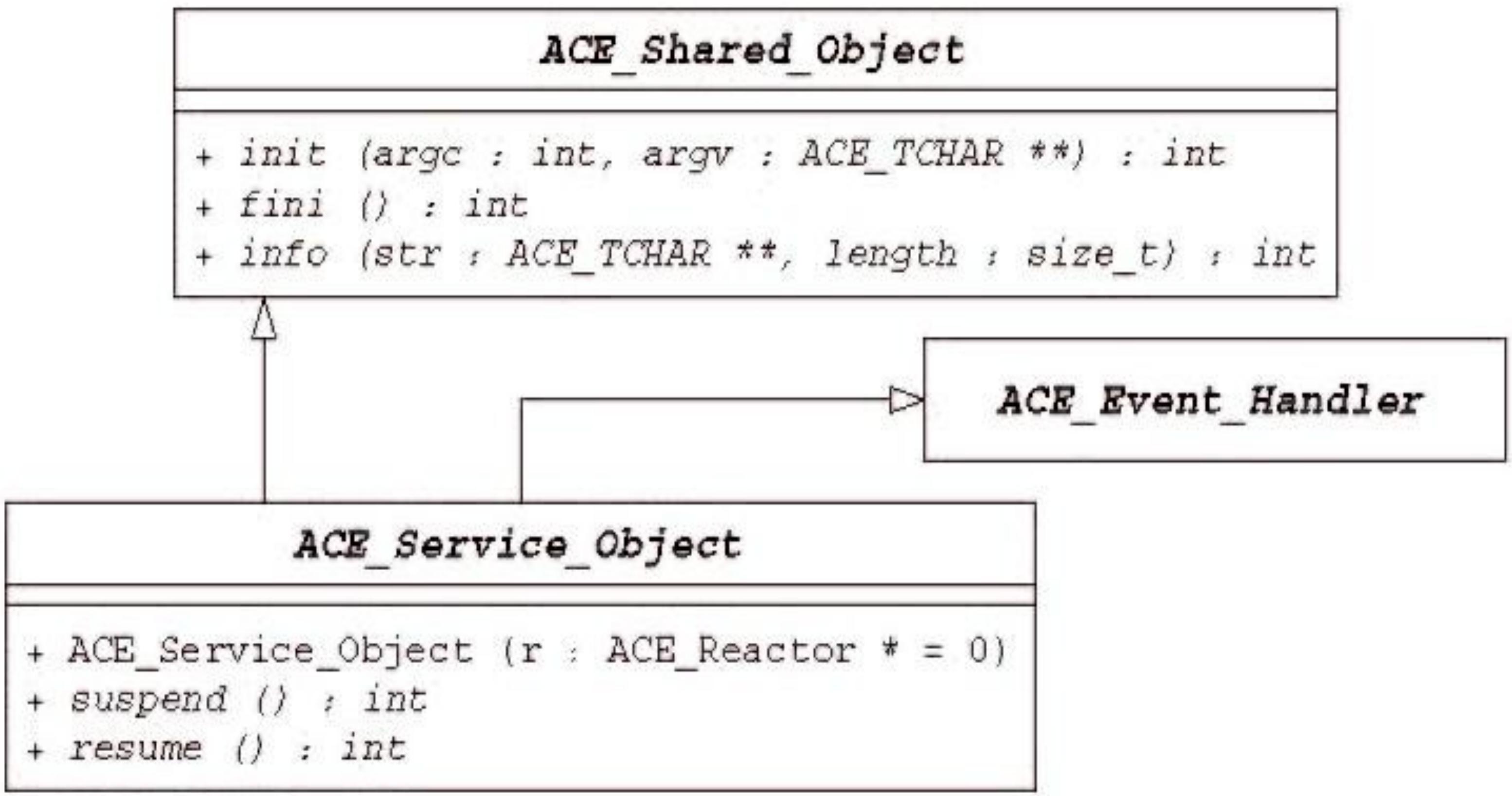
- Configuring & managing service life cycles involves the following aspects:
  - Initialization
  - Execution control
  - Reporting
  - Termination
- Developing these capabilities in an *ad hoc* manner can produce tightly coupled data structures & classes

The ACE_Service_Object Class (2/2)

Class Capabilities

- **`ACE_Service_Object`** provides a uniform interface that allows service implementations to be configured & managed by the ACE Service Configurator framework to provide the following capabilities:
  - It provides hook methods that initialize a service & shut a service down
  - It provides hook methods to suspend service execution temporarily & to resume execution of a suspended service
  - It provides a hook method that reports key service information, such as its purpose, current status, & the port number where it listens for client connections

The ACE_Service_Object Class API



Sidebar: Dealing with Wide Characters in ACE

- Developers outside the United States are acutely aware that many character sets in use today require more than one byte, or octet, to represent each character
- Characters that require more than one octet are referred to as "wide characters"
- The most popular multiple octet standard is ISO/IEC 10646, the Universal Multiple-Octet Coded Character Set (UCS)
- Unicode is a separate standard, but is essentially a restricted subset of UCS that uses two octets for each character (UCS-2)
- To improve portability & ease of use, ACE uses C++ method overloading & the macros described below to use different character types without changing APIs:

Macro	Usage
ACE_HAS_WCHAR	Configuration setting to build ACE with its wide-character methods
ACE_USES_WCHAR	Configuration setting that directs ACE to use wide characters internally
ACE_TCHAR	Defined as either char or wchar_t, to match ACE's internal character width
ACE_TEXT(str)	Defines the string literal str correctly based on ACE_USES_WCHAR
ACE_TEXT_CHAR_TO_TCHAR(str)	Converts a char * string to ACE_TCHAR format, if needed
ACE_TEXT_ALWAYS_CHAR(str)	Converts an ACE_TCHAR string to char * format, if needed

Using the ACE_Service_Object Class (1/4)

- To illustrate the `ACE_Service_Object` class, we reimplement our reactive logging server from the Reactor slides
- This revision can be configured dynamically by the ACE Service Configurator framework, rather than configured statically

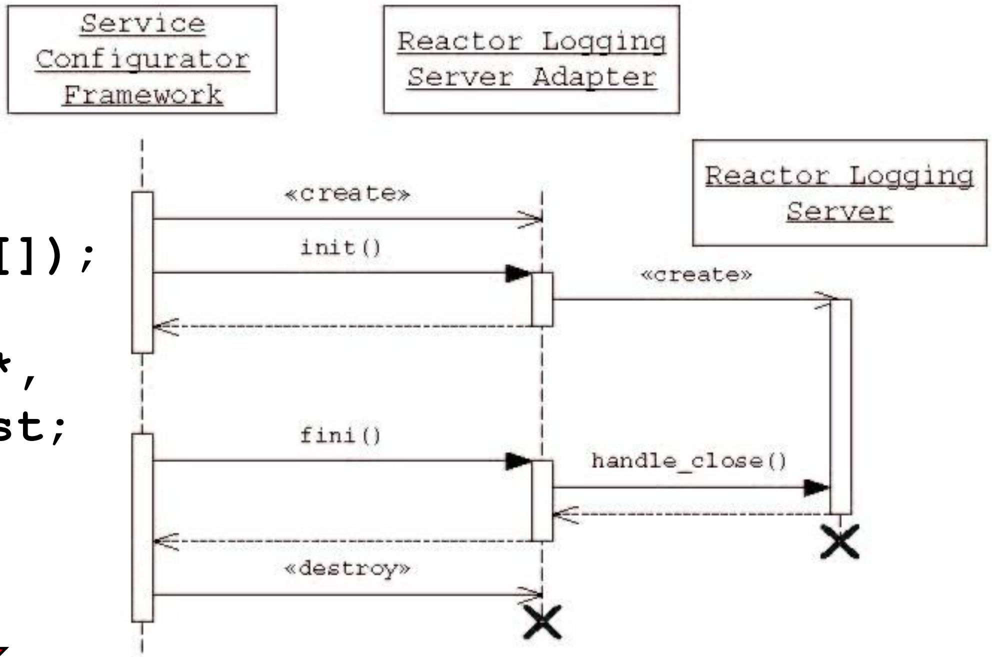
```
template <class ACCEPTOR>
class Reactor_Logging_Server_Adapter : public ACE_Service_Object
{
public:
```

Hook methods inherited from ACE_Service_Object

```
  virtual int init
     (int argc, ACE_TCHAR *argv[]);
  virtual int fini ();
  virtual int info (ACE_TCHAR **,
                    size_t) const;
  virtual int suspend ();
  virtual int resume ();
```

Note reuse of this class

```
private:
  Reactor_Logging_Server<ACCEPTOR> *server_;
}
```



Using the ACE_Service_Object Class (2/4)

```
template <class ACCEPTOR> int
Reactor_Logging_Server_Adapter<ACCEPTOR>::init
  (int argc, ACE_TCHAR *argv[])
{
  int i;
  char **array = 0;
  ACE_NEW_RETURN (array, char*[argc], -1);
  ACE_Auto_Array_Ptr<char *> char_argv (array);

  for (i = 0; i < argc; ++i)
    char_argv[i] = ACE::strnew
(ACE_TEXT_ALWAYS_CHAR(argv[i]));
  ACE_NEW_NORETURN (server_, Reactor_Logging_Server<ACCEPTOR>
                               (i, char_argv.get (),
                                ACE_Reactor::instance ()));
  for (i = 0; i < argc; ++i) ACE::strdelete (char_argv[i]);
  return server_ == 0 ? -1 : 0;
}
```

This hook method is called back by the ACE Service Configurator framework to initialize the service

Sidebar: Portable Heap Operations with ACE

- A surprisingly common misconception is that simply ensuring the proper matching of calls to **`operator new()`** & **`operator delete()`** (or calls to **`malloc()`** & **`free()`**) is sufficient for correct heap management
- While this strategy works if there's one heap per process, there may be multiple heaps
  - e.g., Windows supplies multiple variants of the C/C++ run-time library (such as Debug versus Release & Multithreaded versus Single-threaded), each of which maintains its own heap
  - Memory allocated from one heap must be released back to the same heap
  - It's easy to violate these requirements when code from one subsystem or provider frees memory allocated by another
- To help manage dynamic memory, ACE offers matching allocate & free methods:

Method	Usage
`ACE::strnew()`	Allocates memory for a copy of a character string and copies the string into it.
`ACE::strdelete()`	Releases memory allocated by `strnew()`.
`ACE_OS_Memory::malloc()`	Allocates a memory block of specified size.
`ACE_OS_Memory::calloc()`	Allocates a memory block to hold a specified number of objects, each of a given size. The memory contents are explicitly initialized to 0.
`ACE_OS_Memory::realloc()`	Changes the size of a memory block allocated via `ACE_OS_Memory::malloc()`.
`ACE_OS_Memory::free()`	Releases memory allocated via any of the above three `ACE_OS_Memory` methods.

Using the ACE_Service_Object Class (3/4)

```
template <class ACCEPTOR> int
Reactor_Logging_Server_Adapter<ACCEPTOR>::fini () {
  server_->handle_close (); server_ = 0; return 0;
}
```

This hook method is called by framework to terminate the service

```
template <class ACCEPTOR> int
Reactor_Logging_Server_Adapter<ACCEPTOR>::info
    (ACE_TCHAR **bufferp, size_t length) const {
  ACE_TYPENAME ACCEPTOR::PEER_ADDR local_addr;
  server_->acceptor ().get_local_addr (local_addr);

  ACE_TCHAR buf[BUFSIZ];
  ACE_OS::sprintf (buf,
                   ACE_TEXT ("%hu"),
                   local_addr.get_port_number ());
  ACE_OS_String::strcat
    (buf, ACE_TEXT ("/tcp # Reactive logging server\n"));
  if (*bufferp == 0) *bufferp = ACE::strnew (buf);
  else ACE_OS_String::strncpy (*bufferp, buf, length);
  return ACE_OS_String::strlen (*bufferp);
}
```

This hook method is called by framework to query the service

Using the ACE_Service_Object Class (4/4)

```
template <class ACCEPTOR> int
Reactor_Logging_Server_Adapter<ACCEPTOR>::suspend ()
{
   return server_->reactor ()->suspend_handler (server_);
}
```

These hook methods are called by framework to suspend/resume a service

```
template <class ACCEPTOR> int
Reactor_Logging_Server_Adapter<ACCEPTOR>::resume ()
{
  return server_->reactor ()->resume_handler (server_);
}
```

The ACE_Service_Repository Class (1/2)

Motivation

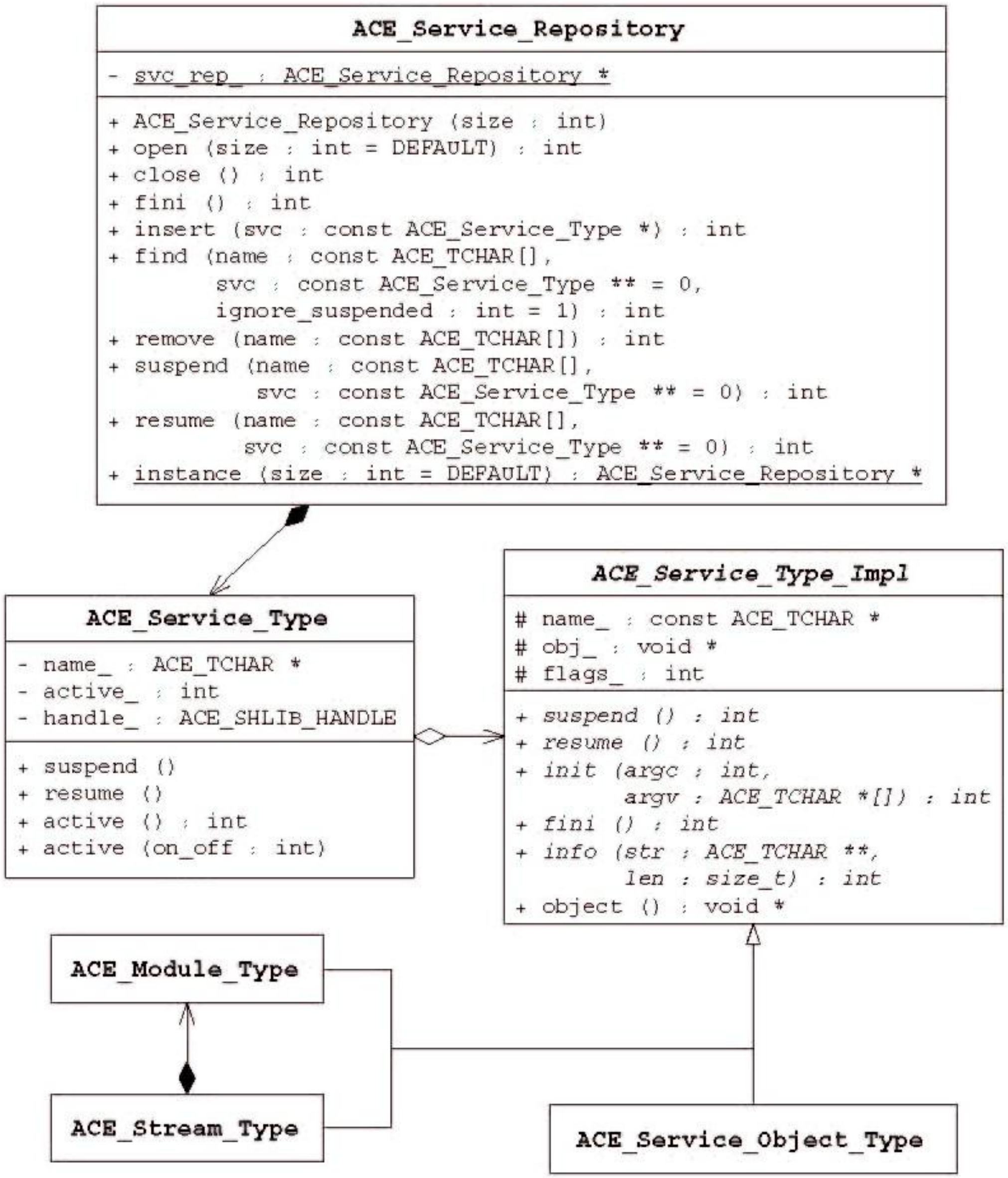
- Applications may need to know what services they are configured with
- Application services in multiservice servers may require access to each other
- To provide info on configured services & to avoid tightly coupling these services, **`ACE_Service_Repository`** enables applications & services to locate each other at run time

The ACE_Service_Repository Class (2/2)

Class Capabilities

- This class implements the Manager pattern (PLoPD3) to control service objects configured by the Service Configurator & to provide the following capabilities:
  - It keeps track of all service implementations configured into an application & maintains service status
  - It provides the mechanism by which the ACE Service Configurator framework inserts, manages, & removes services
  - It provides a convenient mechanism to terminate all services, in reverse order
  - It allows an individual service to be located by its name

The ACE_Service_Repository Class API



Sidebar: The ACE_Dynamic_Service Template (1/2)

- The **ACE_Dynamic_Service** singleton template provides a type-safe way to access the **ACE_Service_Repository** programmatically
- An application process can use this template to retrieve services registered with its local **ACE_Service_Repository**
- If an instance of the **Server_Logging_Daemon** service has been linked dynamically & initialized by the ACE Service Configurator framework, an application can use the **ACE_Dynamic_Service** template to access the service programmatically as shown below:

```
typedef Reactor_Logging_Server_Adapter<Logging_Acceptor>
        Server_Logging_Daemon;

Server_Logging_Daemon *logging_server =
  ACE_Dynamic_Service<Server_Logging_Daemon>::instance
    (ACE_TEXT ("Server_Logging_Daemon"));

ACE_TCHAR *service_info = 0;
logging_server->info (&service_info);
ACE_DEBUG ((LM_DEBUG, "%s\n", service_info));
ACE::strdelete (service_info);
```

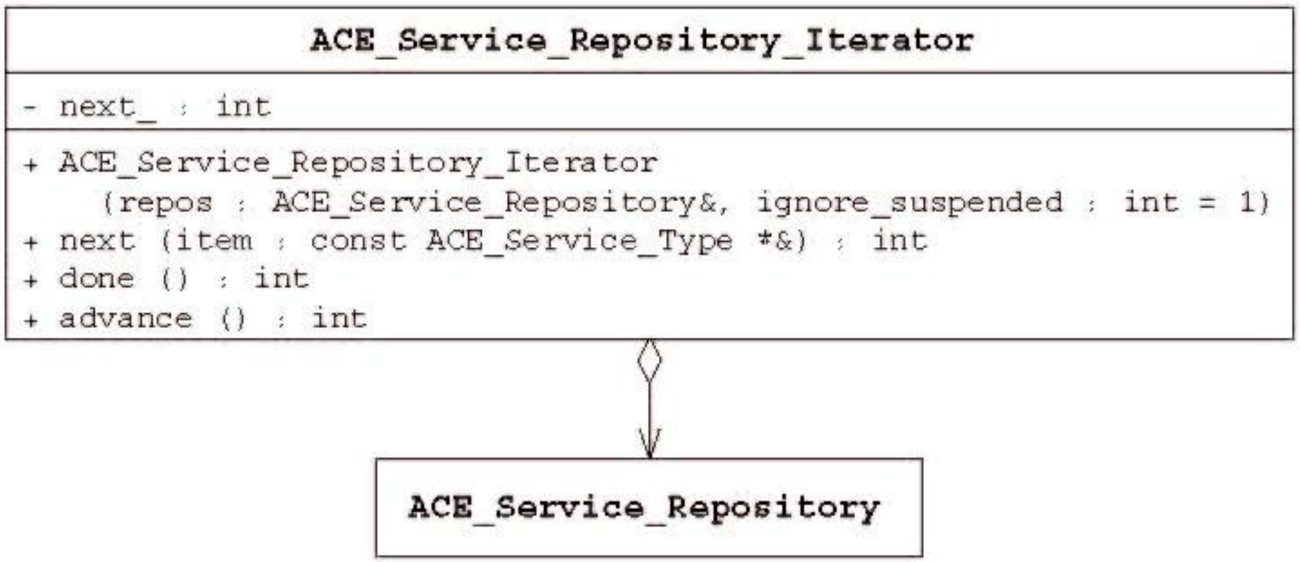
Sidebar: The ACE_Dynamic_Service Template (2/2)

- As shown below, the **TYPE** template parameter ensures that a pointer to the appropriate type of service is returned from the static **instance()** method:

```
template <class TYPE> class ACE_Dynamic_Service {
public:
  // Use <name> to search the <ACE_Service_Repository>.
  static TYPE *instance (const ACE_TCHAR *name) {
    const ACE_Service_Type *svc_rec;
    if (ACE_Service_Repository::instance ()->find
           (name, &svc_rec) == -1) return 0;
    const ACE_Service_Type_Impl *type = svc_rec->type
();
    if (type == 0) return 0;
    ACE_Service_Object *obj =
      ACE_static_cast (ACE_Service_Object *,
                       type->object ());
    return ACE_dynamic_cast (TYPE *, obj);
  }
};
```

The ACE_Service_Repository_Iterator Class

- **`ACE_Service_Repository_Iterator`** implements the Iterator pattern (GoF) to provide applications with a way to sequentially access the **`ACE_Service_Type`** items in an **`ACE_Service_Repository`** without exposing its internal representation



Never delete entries from an **`ACE_Service_Repository`** that's being iterated over since the **`ACE_Service_Repository_Iterator`** is not a ***robust iterator***

Using the ACE_Service_Repository Class (1/8)

- This example illustrates how the **`ACE_Service_Repository`** & **`ACE_Service_Repository_Iterator`** classes can be used to implement a **`Service_Reporter`** class
- This class provides a "meta-service" that clients can use to obtain information on all services that the ACE Service Configurator framework has configured into an application statically or dynamically
- A client interacts with a **`Service_Reporter`** as follows:
  - The client establishes a TCP connection to the **`Service_Reporter`** object
  - The **`Service_Reporter`** returns a list of all the server's services to the client
  - The **`Service_Reporter`** closes the TCP/IP connection

Using the ACE_Service_Repository Class (2/8)

```
class Service_Reporter : public ACE_Service_Object {
public:
  Service_Reporter (ACE_Reactor *r = ACE_Reactor::instance
())
    : ACE_Service_Object (r) {}

  virtual int init (int argc, ACE_TCHAR *argv[]);
  virtual int fini ();
  virtual int info (ACE_TCHAR **, size_t) const;
  virtual int suspend ();
  virtual int resume ();

protected:
  virtual int handle_input (ACE_HANDLE);
  virtual ACE_HANDLE get_handle () const
  { return acceptor_.get_handle (); }
private:
  ACE_SOCK_Acceptor acceptor_; // Acceptor instance.
  enum { DEFAULT_PORT = 9411 };
};
```

These hook methods are inherited from ACE_Service_Object

These hook methods are inherited from ACE_Event_Handler

Using the ACE_Service_Repository Class (3/8)

This hook method is called back by the ACE Service Configurator framework to initialize the service

```
int Service_Reporter::init (int argc, ACE_TCHAR *argv[]) {
  ACE_INET_Addr local_addr
(Service_Reporter::DEFAULT_PORT);
  ACE_Get_Opt get_opt (argc, argv, ACE_TEXT ("p:"), 0);
  get_opt.long_option (ACE_TEXT ("port"),
                       'p', ACE_Get_Opt::ARG_REQUIRED);
  for (int c; (c = get_opt ()) != -1;)
    if (c == 'p') local_addr.set_port_number
                   (ACE_OS::atoi (get_opt.opt_arg ()));
  acceptor_.open (local_addr);
  return reactor ()->register_handler
                (this,
                 ACE_Event_Handler::ACCEPT_MASK);
}
```

Listen for connections

Register to handle connection events

Using the ACE_Service_Repository Class (4/8)

This method is called back by ACE_Reactor

```
int Service_Reporter::handle_input (ACE_HANDLE) {
  ACE_SOCK_Stream peer_stream;
  acceptor_.accept (peer_stream);

  ACE_Service_Repository_Iterator iterator
     (*ACE_Service_Repository::instance (), 0);

  for (const ACE_Service_Type *st;
          iterator.next (st) != 0;
          iterator.advance ()) {
     iovec iov[3];
     iov[0].iov_base = ACE_const_cast (char *, st->name ());
     iov[0].iov_len =
       ACE_OS_String::strlen (st->name ()) * sizeof
(ACE_TCHAR);
     const ACE_TCHAR *state = st->active () ?
            ACE_TEXT (" (active) ") : ACE_TEXT (" (paused) ");
     iov[1].iov_base = ACE_const_cast (char *, state);
     iov[1].iov_len =
       ACE_OS_String::strlen (state) * sizeof (ACE_TCHAR);
```

Note that this is an iterative server

Note that this is the use of the Iterator pattern

Using the ACE_Service_Repository Class (5/8)

```
    ACE_TCHAR *report = 0;   // Ask info() to allocate buffer.
    int len = st->type ()->info (&report, 0);
    iov[2].iov_base = ACE_static_cast (char *, report);
    iov[2].iov_len = ACE_static_cast (size_t, len);
    iov[2].iov_len *= sizeof (ACE_TCHAR);
    peer_stream.sendv_n (iov, 3);
    ACE::strdelete (report);
  }

  peer_stream.close ();
  return 0;
}
```

Gather-write call

Using the ACE_Service_Repository Class (6/8)

```
int Service_Reporter::info (ACE_TCHAR **bufferp,
                            size_t length) const {
  ACE_INET_Addr local_addr;
  acceptor_.get_local_addr (local_addr);

  ACE_TCHAR buf[BUFSIZ];
  ACE_OS::sprintf
    (buf, ACE_TEXT ("%hu"), local_addr.get_port_number
());
  ACE_OS_String::strcat
    (buf, ACE_TEXT ("/tcp # lists services in daemon\n"));
  if (*bufferp == 0) *bufferp = ACE::strnew (buf);
  else ACE_OS_String::strncpy (*bufferp, buf, length);
  return ACE_OS_String::strlen (*bufferp);
}

int Service_Reporter::suspend ()
{ return reactor ()->suspend_handler (this); }

int Service_Reporter::resume ()
{ return reactor ()->resume_handler (this); }
```

Using the ACE_Service_Repository Class (7/8)

```
int Service_Reporter::fini () {
  reactor ()->remove_handler
    (this,
     ACE_Event_Handler::ACCEPT_MASK
     | ACE_Event_Handler::DONT_CALL);
  return acceptor_.close ();
}
```

Note the use of the `DONT_CALL` mask to avoid recursion

```
ACE_FACTORY_DEFINE (ACE_Local_Service,
Service_Reporter)

ACE_STATIC_SVC_DEFINE (
  Reporter_Descriptor,
  ACE_TEXT ("Service_Reporter"),
  ACE_SVC_OBJ_T,
  &ACE_SVC_NAME (Service_Reporter),
  ACE_Service_Type::DELETE_THIS
  | ACE_Service_Type::DELETE_OBJ,
  0 // This object is not initially active.
)

ACE_STATIC_SVC_REQUIRE (Reporter_Descriptor)
```

These macros integrate the service with the ACE Service Configurator framework

Using the ACE_Service_Repository Class (8/8)

- The `ACE_FACTORY_DEFINE` macro generates these functions automatically

```
void _gobble_Service_Reporter (void *arg) {
  ACE_Service_Object *svcobj =
    ACE_static_cast (ACE_Service_Object *, arg);
  delete svcobj;
}
```

We use `extern "C"` to avoid "name mangling"

```
extern "C" ACE_Service_Object *
_make_Service_Reporter (void (**gobbler) (void *)) {
  if (gobbler != 0) *gobbler =
_gobble_Service_Reporter;
  return new Service_Reporter;
}
```

This function is typically designated in a `svc.conf` file

Sidebar: The ACE Service Factory Macros (1/2)

- **Factory & gobbler function macros**
  - Static & dynamic services must supply a factory function to create the service object & a "gobbler" function to delete it
  - ACE provides the following three macros to help generate & use these functions:
    - **`ACE_FACTORY_DEFINE(LIB, CLASS)`**, which is used in an implementation file to define the factory & gobbler functions for a service
      - **`LIB`** is the ACE export macro prefix used with the library containing the factory function
      - **`CLASS`** is the type of service object the factory must create
    - **`ACE_FACTORY_DECLARE(LIB, CLASS)`**, which declares the factory function defined by the **`ACE_FACTORY_DEFINE`** macro
      - Use this macro to generate a reference to the factory function from a compilation unit other than the one containing the **`ACE_FACTORY_DEFINE`** macro
    - **`ACE_SVC_NAME(CLASS)`**, which generates the name of the factory function defined via the **`ACE_FACTORY_DEFINE`** macro
      - The generated name can be used to get the function address at compile time, such as for the **`ACE_STATIC_SVC_DEFINE`** macro, below

Sidebar: The ACE Service Factory Macros (2/2)

- **Static service information macro**
  - ACE provides the following macro to generate static service registration information, which defines the service name, type, & a pointer to the factory function the framework calls to create a service instance:
    - **`ACE_STATIC_SVC_DEFINE(REG, NAME, TYPE, FUNC_ADDR, FLAGS, ACTIVE)`**, which is used in an implementation file to define static service info
      - **`REG`** forms the name of the information object, which must match the parameter passed to **`ACE_STATIC_SVC_REQURE`** & **`ACE_STATIC_SVC_REGISTER`**
      - Other parameters set **`ACE_Static_Svc_Descriptor`** attribute
- **Static service registration macros**
  - The static service registration information must be passed to the ACE Service Configurator framework at program startup
  - The following two macros cooperate to perform this registration:
    - **`ACE_STATIC_SVC_REQUIRE(REG)`**, which is used in the service implementation file to define a static object whose constructor will add the static service registration information to the framework's list of known static services.
    - **`ACE_STATIC_SVC_REGISTER(REG)`**, which is used at the start of the main program to ensure the object defined in **`ACE_STATIC_SVC_REQUIRES`** registers the static service no later than the point this macro appears

Sidebar: The ACE_Service_Manager Class

- **`ACE_Service_Manager`** provides clients with access to administrative commands to access & manage the services currently offered by a network server
- These commands "externalize" certain internal attributes of the services configured into a server
- During server configuration, an **`ACE_Service_Manager`** is typically registered at a well-known communication port, e.g., port 9411
- Clients can connect to an **`ACE_Service_Manager`** at that port & issue one of the following commands:
  - **`help`**, which lists of all services configured into an application via the ACE Service Configurator framework
  - **`reconfigure`**, which is triggered to reread the local service configuration file
- If a client sends anything other than these two commands, its input is passed to **`ACE_Service_Config::process_directive()`**, which enables remote configuration of servers via command-line instructions such as

```
% echo "suspend My_Service" | telnet hostname 9411
```

- It's therefore important to use the **`ACE_Service_Manager`** only if your application runs in a trusted environment since a malicious attacker can use it to deny access to legitimate services or configure rogue services in a *Trojan Horse* manner
- **`ACE_Service_Manager`** is therefore a static service that ACE disables by default

The ACE_Service_Config Class (1/2)

Motivation

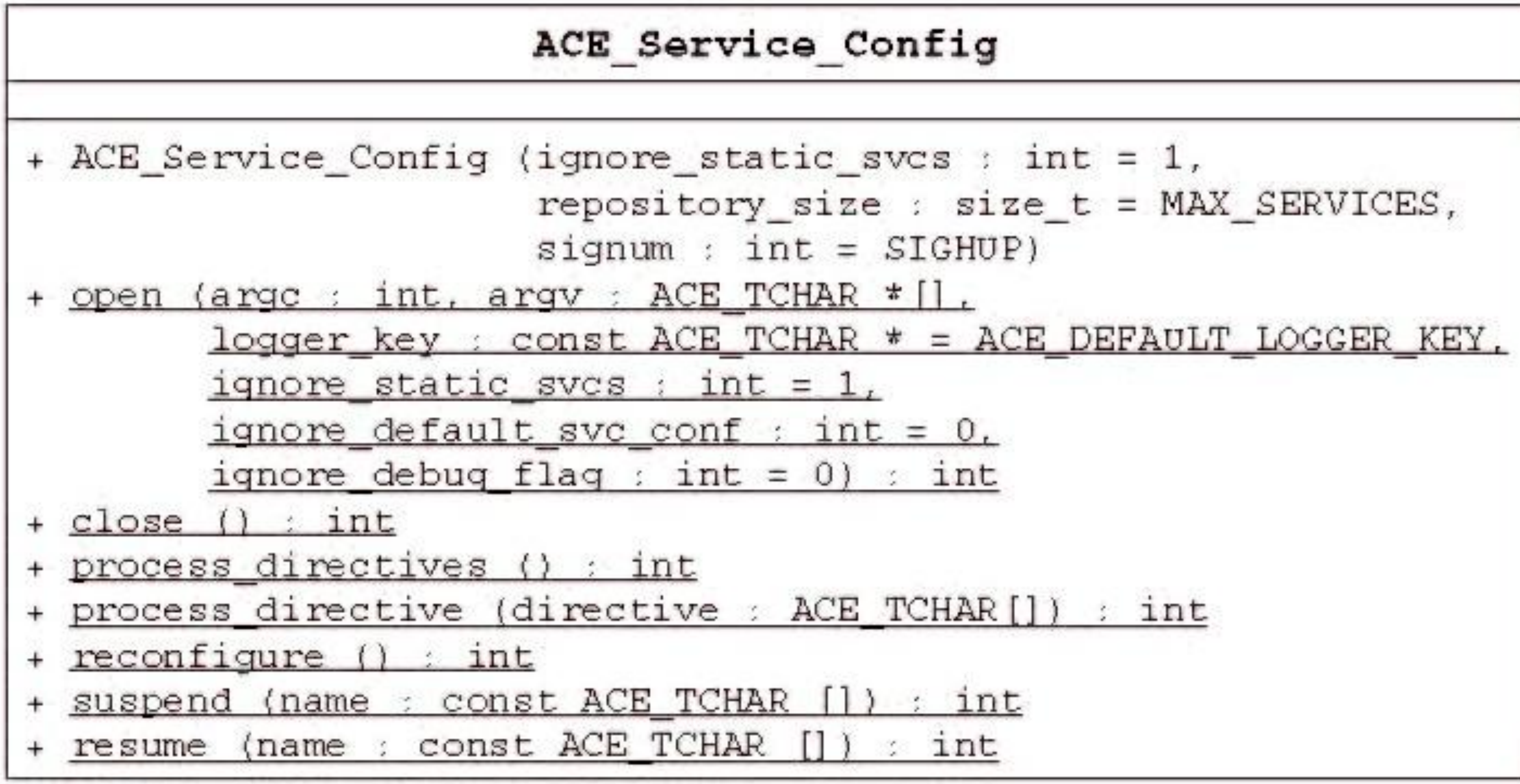
- Statically configured applications have the following drawbacks:
  - Service configuration decisions are made prematurely in the development cycle
  - Modifying a service may affect other services adversely
  - System performance may scale poorly

The ACE_Service_Config Class (2/2)

Class Capabilities

- This class implements the Façade pattern to integrate other Service Configurator classes & coordinate the activities necessary to manage the services in an application via the following capabilities:
  - It interprets a scripting language can provide the Service Configurator with directives to locate & initialize a service's implementation at run time, as well as to suspend, resume, reinitialize, & shut down a component after it's been initialized
  - It supports the management of services located in the application (static services) as well as those that must be linked dynamically (dynamic services) from separate shared libraries (DLLs)
  - It allows service reconfiguration at run time

The ACE_Service_Config Class API



ACE_Service_Config Options

- There's only one instance of **`ACE_Service_Config`**'s state in a process
- This class is a variant of the Monostate pattern, which ensures a unique state for its instances by declaring all data members to be static
- The **`open()`** method is the common way of initializing the **`ACE_Service_Config`**
- It parses arguments passed in the **`argc`** & **`argv`** parameters, skipping the first parameter (**`argv[0]`**) since that's the name of the program
- The options recognized by **`ACE_Service_Config`** are outlined in the following table:

Option	Description
'-b'	Turn the application process into a *daemon* (see Sidebar 5 on page 32).
'-d'	Display diagnostic information as directives are processed.
'-f'	Supply a file containing directives other than the default `svc.conf` file. This argument can be repeated to supply multiple configuration files.
'-n'	Don't process `static` directives, which eliminates the need to initialize the `ACE_Service_Repository` statically.
'-s'	Designate the signal to be used to cause the `ACE_Service_Config` to reprocess its configuration file. By default, `SIGHUP` is used.
'-S'	Supply a directive to the `ACE_Service_Config` directly. This argument can be repeated to process multiple directives.
'-y'	Process `static` directives, which requires the static initialization of the `ACE_Service_Repository`.

Service Configuration Directives

- Directives are commands that can be passed to the ACE Service Configurator framework to designate its behavior
- The following directives are supported:

Directive	Description
`dynamic`	Dynamically link a service and initialize it by calling its `init()` hook method.
`static`	Call the `init()` hook method to initialize a service that was linked statically.
`remove`	Remove a service completely, that is, call its `fini()` hook method and unlink it from the application process when it's no longer used.
`suspend`	Call a service's `suspend()` hook method to pause it without removing it.
`resume`	Call a service's `resume()` hook method to continue processing a service that was suspended earlier.
`stream`	Initialize an ordered list of hierarchically related modules.

- Directives can be specified to **`ACE_Service_Config`** in either of two ways:
  - Using configuration files (named **`svc.conf`** by default) that contain one or more directives
  - Programmatically, by passing individual directives as strings to the **`ACE_Service_Config::process_directive()`** method

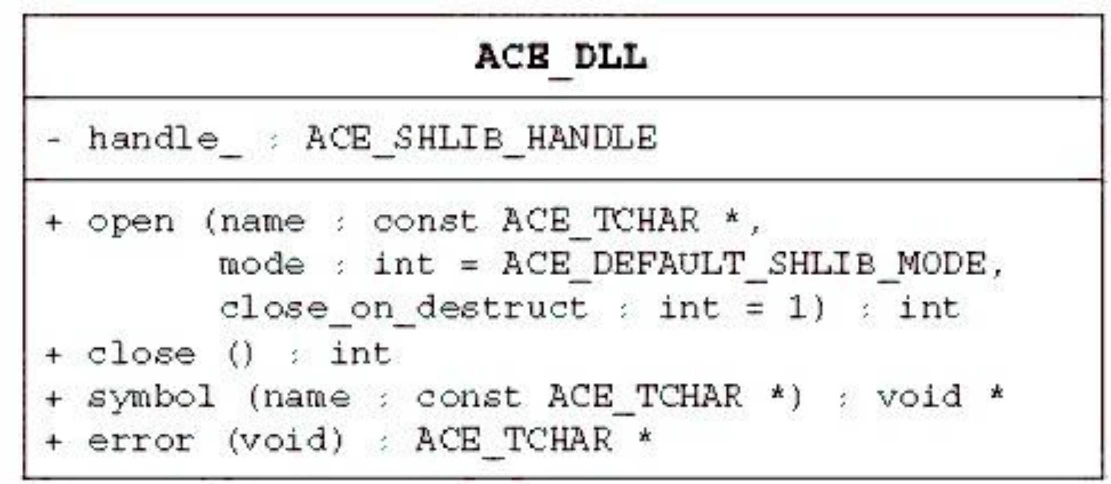
BNF for the svc.conf File

- The complete Backus/Naur Format (BNF) syntax for svc.conf files parsed by the **`ACE_Service_Config`** is shown below:

```
<svc-conf-entries> ::= <svc-conf-entries> <svc-conf-entry> | NULL
<svc-conf-entry>   ::= <dynamic> | <static> | <suspend> |
                        <resume> | <remove> | <stream>
<dynamic> ::= dynamic <svc-location> <parameters-opt>
<static> ::= static <svc-name> <parameters-opt>
<suspend> ::= suspend <svc-name>
<resume> ::= resume <svc-name>
<remove> ::= remove <svc-name>
<stream> ::= stream <streamdef> '{' <module-list> '}'
<streamdef> ::= <svc-name> | dynamic | static
<module-list> ::= <module-list> <module> | NULL
<module> ::= <dynamic> | <static> | <suspend> |
             <resume> | <remove>
<svc-location> ::= <svc-name> <svc-type> <svc-factory> <status>
<svc-type> ::= Service_Object '*' | Module '*' | Stream '*' | NULL
<svc-factory> ::= PATHNAME ':' FUNCTION '(' ')'
<svc-name> ::= STRING
<status> ::= active | inactive | NULL
<parameters-opt> ::= '"' STRING '"' | NULL
```

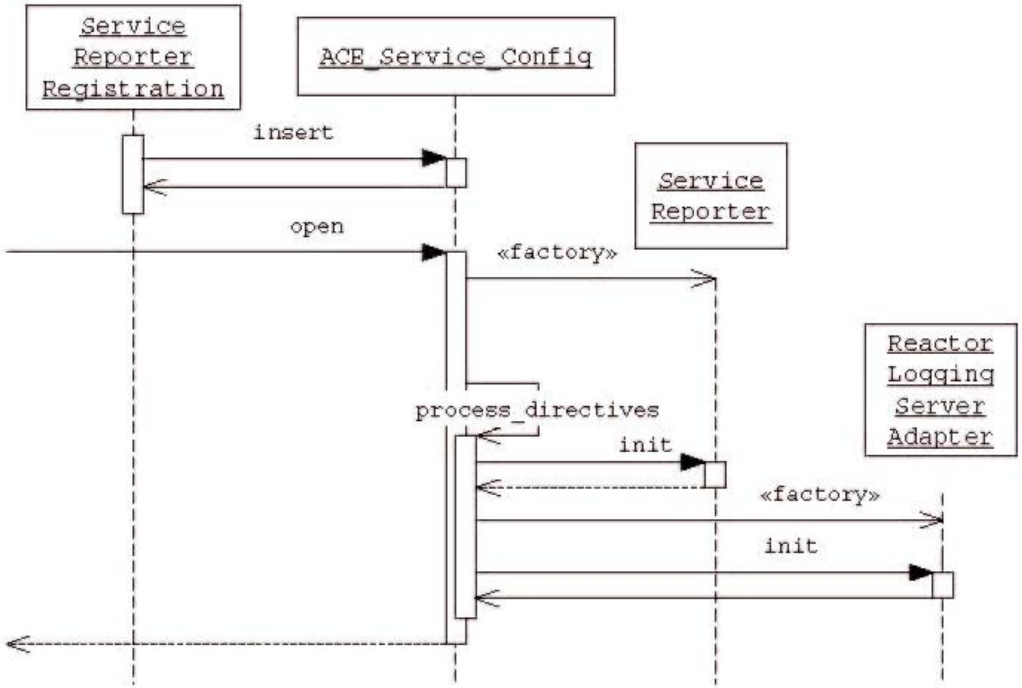
Sidebar: The ACE_DLL Class

- ACE defines the **ACE_DLL** wrapper facade class to encapsulate explicit linking/unlinking functionality
- This class eliminates the need for applications to use error-prone, weakly typed handles & also ensures that resources are released properly by its destructor
- It also uses the **ACE::ldfind()** method to locate DLLs via the following algorithms:
  - **DLL filename expansion**, where **ACE::ldfind()** determines the name of the DLL by adding the appropriate prefix & suffix
    - e.g., it adds the lib prefix & **.so** suffix for Solaris & the **.dll** suffix for Windows
  - **DLL search path**, where **ACE::ldfind()** will also search for the designated DLL using the platform's DLL search path environment variable
    - e.g., it searches for DLLs using **LD_LIBRARY_PATH** on many UNIX systems & PATH on Windows
- The key methods in the ACE_DLL class are outlined in the adjacent UML diagram



Using the ACE_Service_Config Class (1/3)

- This example shows how to apply the ACE Service Configurator framework to create a server whose initial configuration behaves as follows:
    - It statically configures an instance of **`Service_Reporter`**
    - It dynamically links & configures the **`Reactor_Logging_Server_Adapter`** template into the server's address space



- We later show how to dynamically reconfigure the server to support a different implementation of a reactive logging service

Using the ACE_Service_Config Class (2/3)

- We start by writing the following generic `main()` program
- This program uses a `svc.conf` file to configure the `Service_Reporter` & `Reactor_Logging_Server_Adapter` services into an application process & then runs the reactor's event loop

```
#include "ace/OS.h"
#include "ace/Service_Config.h"
#include "ace/Reactor.h"

int ACE_TMAIN (int argc, ACE_TCHAR *argv[]) {
  ACE_STATIC_SVC_REGISTER (Reporter);

  ACE_Service_Config::open
    (argc, argv, ACE_DEFAULT_LOGGER_KEY, 0);

  ACE_Reactor::instance ()->run_reactor_event_loop ();
  return 0;
}
```

Most of the rest of the examples use a similar main() function!

Using the ACE_Service_Config Class (3/3)

This is the `SLD.cpp` file used to define the `Server_Logging_Daemon` type

```
#include "Reactor_Logging_Server_Adapter.h"
#include "Logging_Acceptor.h"
#include "SLD_export.h"

typedef
Reactor_Logging_Server_Adapter<Logging_Acceptor>
        Server_Logging_Daemon;

ACE_FACTORY_DEFINE (SLD, Server_Logging_Daemon)
```

This svc.conf file is used to configure the main program

```
static Service_Reporter "-p $SERVICE_REPORTER_PORT"

dynamic Server_Logging_Daemon Service_Object *
SLD:_make_Server_Logging_Daemon()
  "$SERVER_LOGGING_DAEMON_PORT"
```

The `ACE_Service_Config` interpreter uses `ACE_ARGV` to expand environment variables

Sidebar: The ACE_ARGV Class

- The **`ACE_ARGV`** class is a useful utility class that can
  - Transform a string into an **`argc`**/**`argv`**-style vector of strings
  - Incrementally assemble a set of strings into an **`argc`**/**`argv`** vector
  - Transform an **`argc`**/**`argv`**-style vector into a string
- During the transformation, the class can substitute environment variable values for each $-delimited environment variable name encountered.
- **`ACE_ARGV`** provides an easy & efficient mechanism to create arbitrary command-line arguments
  - Consider its use whenever command-line processing is required, especially when environment variable substitution is desirable
- ACE uses **`ACE_ARGV`** extensively, particularly in its Service Configurator framework

Sidebar: Using XML to Configure Services (1/2)

- **`ACE_Service_Config`** can be configured to interpret an XML scripting language
- The Document Type Definition (DTD) for this language is shown below:

```
<!ELEMENT ACE_Svc_Conf (dynamic|static|suspend|resume
                        |remove|stream|streamdef)*>
<!ELEMENT streamdef ((dynamic|static),module)>
<!ATTLIST streamdef id IDREF #REQUIRED>
<!ELEMENT module (dynamic|static|suspend|resume|remove)+>
<!ELEMENT stream (module)>
<!ATTLIST stream id IDREF #REQUIRED>
<!ELEMENT dynamic (initializer)>
<!ATTLIST dynamic id ID #REQUIRED
                  status (active|inactive) "active"
                  type (module|service_object|stream)
                  #REQUIRED>
<!ELEMENT initializer EMPTY>
<!ATTLIST initializer init CDATA #REQUIRED
                      path CDATA #IMPLIED
                      params CDATA #IMPLIED>
<!ELEMENT static EMPTY>
<!ATTLIST static id ID #REQUIRED
                 params CDATA #IMPLIED>
<!ELEMENT suspend EMPTY>
<!ATTLIST suspend id IDREF #REQUIRED>
<!ELEMENT resume EMPTY>
<!ATTLIST resume id IDREF #REQUIRED>
<!ELEMENT remove EMPTY>
<!ATTLIST remove id IDREF #REQUIRED>
```

- The syntax of this XML configuration language is different, though its semantics are the same
- Although it's more verbose to compose, the ACE XML configuration file format is more flexible

Sidebar: Using XML to Configure Services (2/2)

- The XML representation of the **`svc.conf`** file shown earlier is shown below:

```
<ACE_Svc_Conf>
  <static id='Service_Reporter'
          params='-p $SERVICE_REPORTER_PORT'/>

  <dynamic id='Server_Logging_Daemon'
           type='service_object'>
    <initializer path='SLD'
                 init='_make_Server_Logging_Daemon'
                 params='$SERVER_LOGGING_DAEMON_PORT'/>
  </dynamic>
</ACE_Svc_Conf>
```

- The XML **`svc.conf`** file is more verbose than the original format since it specifies field names explicitly
- However, the XML format allows svc.conf files to express expanded capabilities, since new sections & fields can be added without affecting existing syntax
- There's also no threat to backwards compatibility, as might occur if fields were added to the original format or the field order changed

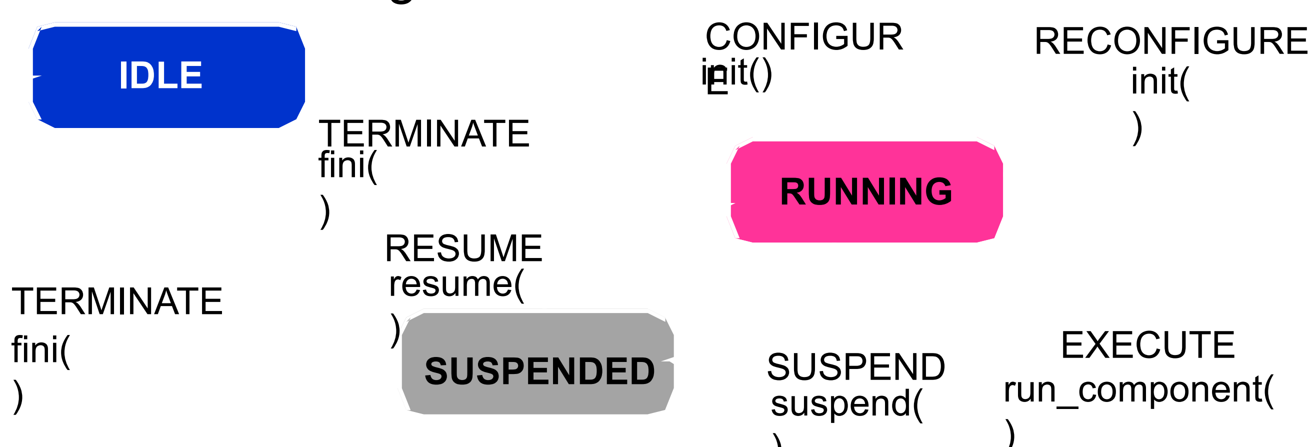
Sidebar: The ACE DLL Import/Export Macros

- Windows has specific rules for explicitly importing & exporting symbols in DLLs
- Developers with a UNIX background may not have encountered these rules in the past, but they are important for managing symbol usage in DLLs on Windows
- ACE makes it easy to conform to these rules by supplying a script that generates the necessary import/export declarations & a set of guidelines for using them successfully
- To ease porting, the following procedure can be used on all platforms that ACE runs on:
  - Select a concise mnemonic for each DLL to be built
  - Run the **`$ACE_ROOT/bin/generate_export_file.pl`** Perl script, specifying the DLL's mnemonic on the command line
    - The script will generate a platform-independent header file & write it to the standard output
    - Redirect the output to a file named **`<mnemonic>_export.h`**
  - **`#include`** the generated file in each DLL source file that declares a globally visible class or symbol
  - To use in a class declaration, insert the keyword **`<mnemonic>_Export`** between class & the class name
  - When compiling the source code for the DLL, define the macro **`<mnemonic>_BUILD_DLL`**

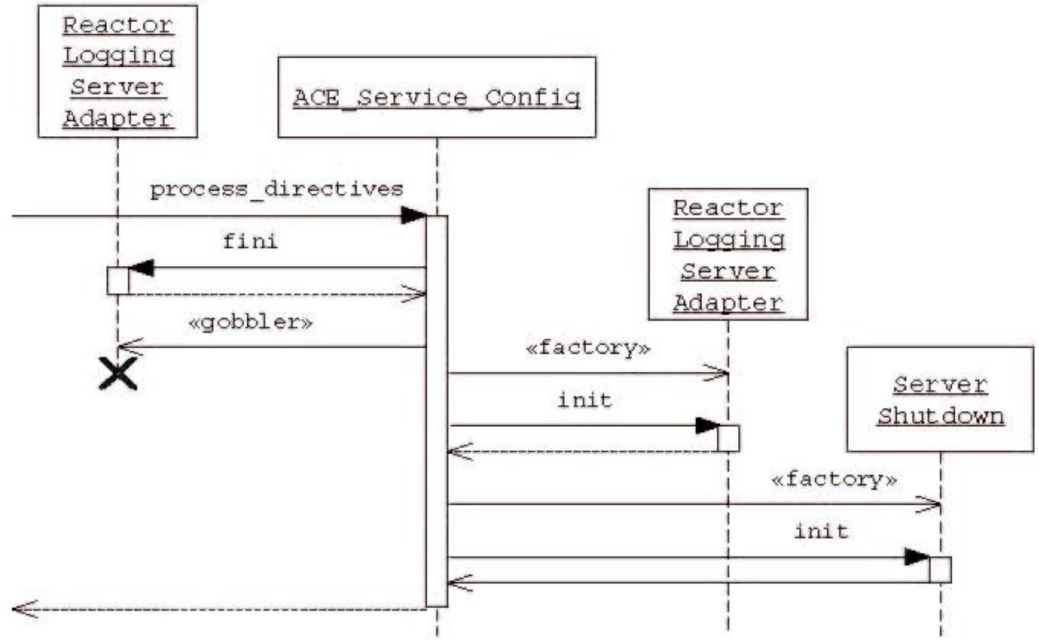
Service Reconfiguration

- An application using the ACE Service Configurator can be reconfigured at runtime using the following mechanisms:
  - On POSIX, **`ACE_Service_Config`** can be integrated with the ACE Reactor framework to reprocess its **`svc.conf`** files(s) upon receipt of a **`SIGHUP`** signal
  - By passing the "**`reconfigure`**" command via **`ACE_Service_Manager`**
  - An application can request its **`ACE_Service_Config`** to reprocess its configuration files at any time
    - e.g., a Windows directory change notification event can be used to help a program learn when its configuration file changes & trigger reprocessing of the configuration
  - An application can also specify individual directives for its **`ACE_Service_Config`** to process at any time via the **`process_directive()`** method

Reconfiguration State Chart



Reconfiguring a Logging Server



- By using the ACE Service Configurator, a logging server can be reconfigured dynamically to support new services & new service implementations

Logging Server Process

Configure a logging server.
dynamic Server_Logging_Daemon Service_Object
* SLD:make_Server_Logging_Daemon()
"$SERVER_LOGGING_DAEMON_PORT"

Logging Server Process

Reconfigure a logging server.
remove Server_Logging_Daemon
dynamic Server_Logging_Daemon Service_Object
* SLDex:make_Server_Logging_Daemon_Ex()
"$SERVER_LOGGING_DAEMON_PORT"
dynamic Server_Shutdown Service_Object *
SLDex:_make_Server_Shutdown()

INITIAL CONFIGURATION

AFTER RECONFIGURATION

Using Reconfiguration Features (1/2)

- The original logging server configuration has the following limitations:
  - It uses **`Logging_Acceptor`**, which doesn't time out idle logging handlers
  - **`ACE_Reactor::run_reactor_event_loop()`** can't be shut down on the reactor singleton
- We can add these capabilities without affecting existing code or the **`Service_Reporter`** service by defining a new **`svc.conf`** file & instructing the server to reconfigure itself

```
remove Server_Logging_Daemon

dynamic Server_Logging_Daemon Service_Object *
SLDex:_make_Server_Logging_Daemon_Ex()
  "$SERVER_LOGGING_DAEMON_PORT"

dynamic Server_Shutdown Service_Object *
SLDex:_make_Server_Shutdown()
```

This is the updated `svc.conf` file

This `SLDex.cpp` file defines the new `Server_Logging_Daemon_Ex` type

```
typedef
Reactor_Logging_Server_Adapter<Logging_Acceptor_Ex>
        Server_Logging_Daemon_Ex;
```

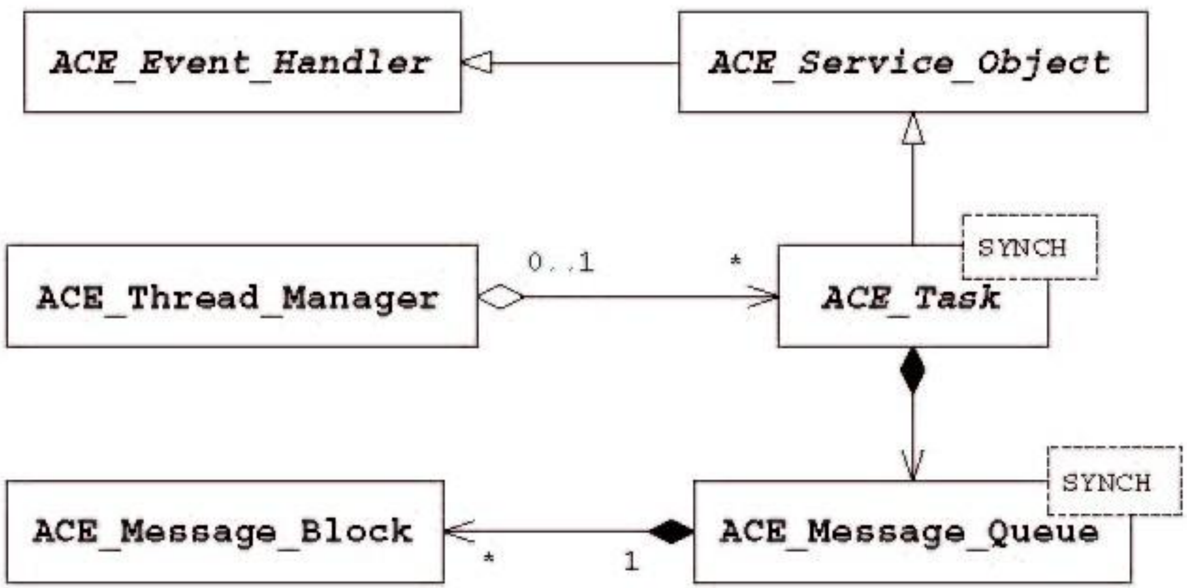
Using Reconfiguration Features (2/2)

```
class Server_Shutdown : public ACE_Service_Object
{
public:
 virtual int init (int, ACE_TCHAR *[]) {
    reactor_ = ACE_Reactor::instance ();
    return ACE_Thread_Manager::instance ()->spawn
             (controller, reactor_, THR_DETACHED);
  }
 virtual int fini () {
    Quit_Handler *quit_handler = 0;
    ACE_NEW_RETURN (quit_handler,
                    Quit_Handler (reactor_), -1);
    return reactor_->notify (quit_handler);
  }

  // ... Other method omitted ...
private:
  ACE_Reactor *reactor_;
};

ACE_FACTORY_DEFINE (SLDEX, Server_Shutdown)
```

Note how we can cleanly add shutdown features via the ACE Service Configurator framework!

The ACE Task Framework

- The ACE Task framework provides powerful & extensible object-oriented concurrency capabilities that can spawn threads in the context of an object
- It can also transfer & queue messages between objects executing in separate threads

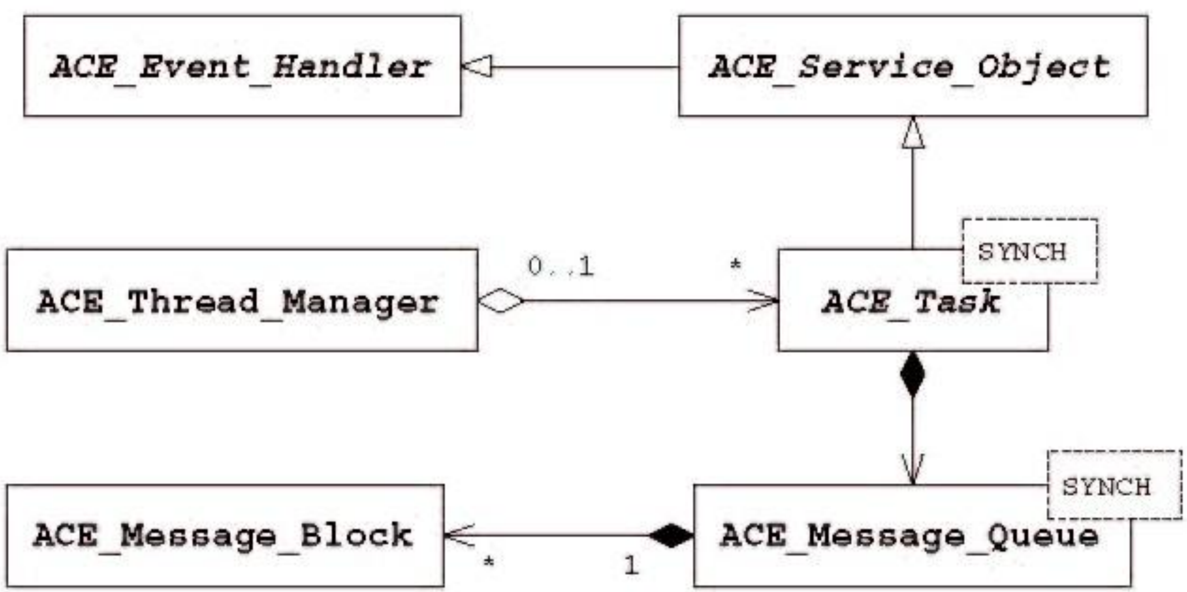


The ACE Task Framework

ACE Class	Description
ACE_Message_Block	Implements the Composite pattern [GoF] to enable efficient manipulation of fixed- and variable-sized messages
ACE_Message_Queue	Provides an intraprocess message queue that enables applications to pass and buffer messages between threads in a process
ACE_Thread_Manager	Allows applications to portably create and manage the lifetime, synchronization, and properties of one or more threads
ACE_Task	Allows applications to create passive or active objects that decouple different units of processing; use messages to communicate requests, responses, data, and control information; and can queue and process messages sequentially or concurrently

- The relationships between classes in ACE Task framework are shown below

- These classes are reused from the ACE Reactor & Service Configurator frameworks



The ACE_Message_Queue Class (1/3)

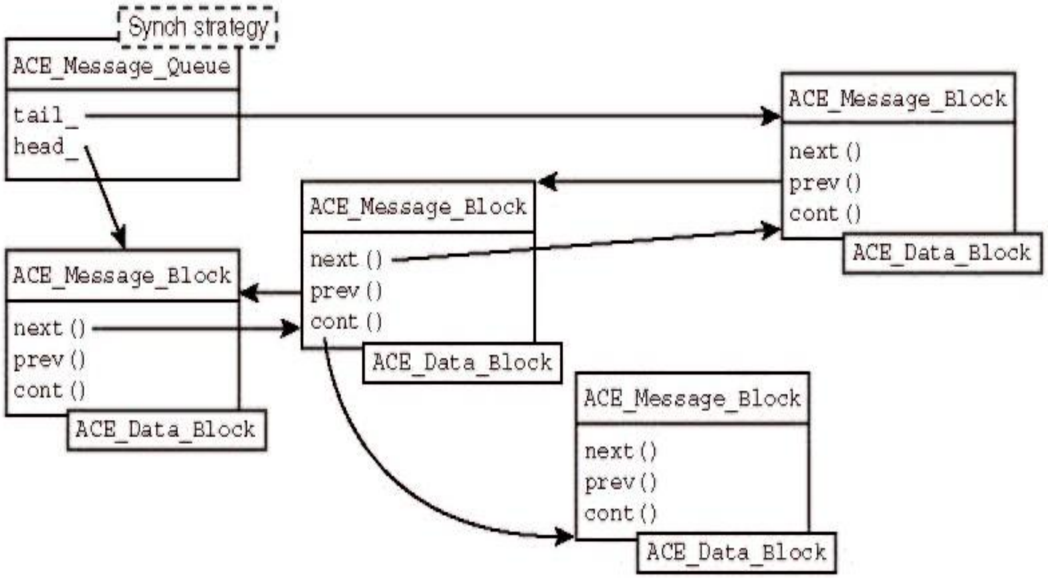
Motivation

- When producer & consumer tasks are collocated in the same process, tasks often exchange messages via an intraprocess message queue
- In this design, producer task(s) insert messages into a synchronized message queue serviced by consumer task(s) that remove & process the messages
- If the queue is full, producers can either block or wait a bounded amount of time to insert their messages
- Likewise, if the queue is empty, consumers can either block or wait a bounded amount of time to remove messages

The ACE_Message_Queue Class (2/3)

Class Capabilities

- This class is a portable intraprocess message queueing mechanism that provides the following capabilities:
  - It allows messages (i.e., **`ACE_Message_Blocks`**) to be enqueued at the front or rear of the queue, or in priority order based on the message's priority
  - Messages can be dequeued from the front or back of the queue



  - **`ACE_Message_Block`** provides an efficient message buffering mechanism that minimizes dynamic memory allocation & data copying

The ACE_Message_Queue Class (3/3)

Class Capabilities

- It can be instantiated for either multi- or single-threaded configurations, allowing trade offs of strict synchronization for lower overhead when concurrent access to a queue isn't required
- In multithreaded configurations, it supports configurable flow control, which prevents fast producers from swamping the processing & memory resources of slower consumers
- It allows timeouts on both enqueue/dequeue operations to avoid indefinite blocking
- It can be integrated with the ACE Reactor
- It provides allocators that can be strategized so the memory used by messages can be obtained from various sources

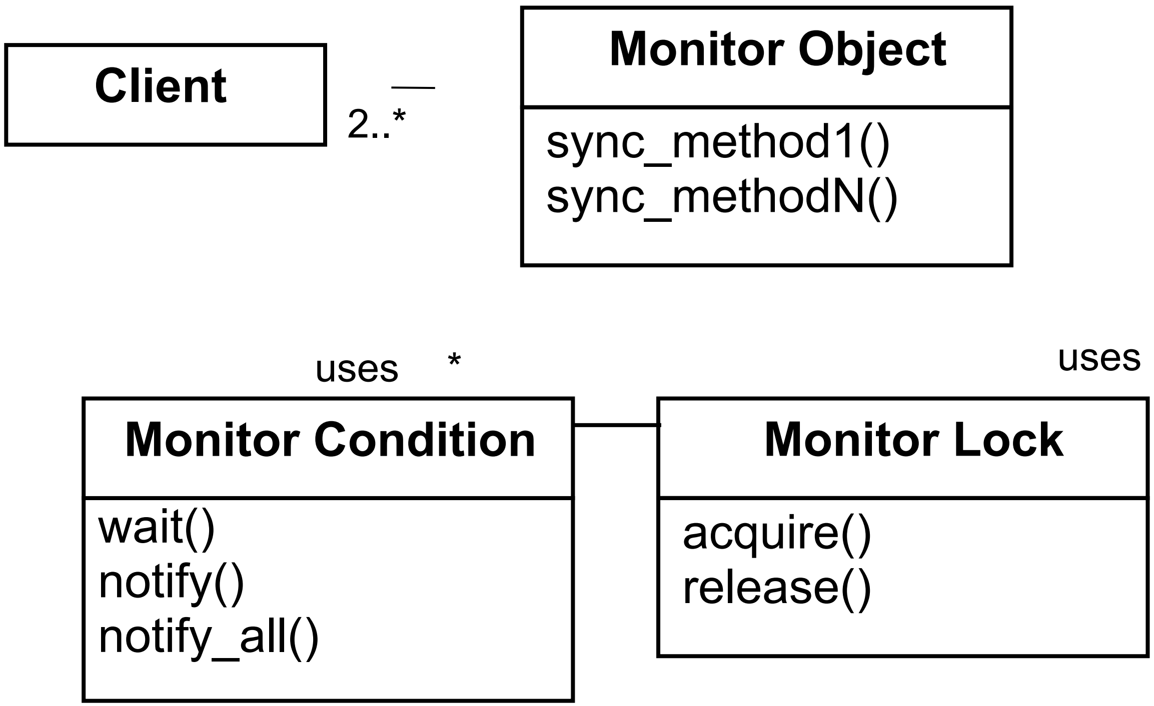
The ACE_Message_Queue Class API

```
                                                          SYNCH_STRATEGY
                         ACE_Message_Queue
# head_ : ACE_Message_Block *
# tail_ : ACE_Message_Block *
# high_water_mark_ : size_t
# low_water_mark_ : size_t

+ ACE_Message_Queue (high_water_mark : size_t = DEFAULT_HWM,
                     low_water_mark : size_t = DEFAULT_LWM,
                     notify : ACE_Notification_Strategy * = 0)
+ open (high_water_mark : size_t = DEFAULT_HWM,
        low_water_mark : size_t = DEFAULT_LWM,
        notify : ACE_Notification_Strategy * = 0) : int
+ flush () : int
+ notification_strategy (s : ACE_Notification_Strategy *) : void
+ is_empty () : int
+ is_full () : int
+ enqueue_tail (item : ACE_Message_Block *,
                timeout : ACE_Time_Value * = 0) : int
+ enqueue_head (item : ACE_Message_Block *,
                timeout : ACE_Time_Value * = 0) : int
+ enqueue_prio (item : ACE_Message_Block *,
                timeout : ACE_Time_Value * = 0) : int
+ dequeue_head (item : ACE_Message_Block *&,
                timeout : ACE_Time_Value * = 0) : int
+ dequeue_tail (item : ACE_Message_Block *&,
                timeout : ACE_Time_Value * = 0) : int
+ high_water_mark (new_hwm : size_t) : void
+ high_water_mark (void) : size_t
+ low_water_mark (new_lwm : size_t) : void
+ low_water_mark (void) : size_t
+ close () : int
+ deactivate () : int
+ activate () : int
+ pulse () : int
+ state () : int
```

The Monitor Object Pattern

- The *Monitor Object* design pattern (POSA2) can be used to synchronize the message queue efficiently & conveniently

- This pattern synchronizes concurrent method execution to ensure that only one method at a time runs within an object
- It also allows an object's methods to cooperatively schedule their execution sequences



- It's instructive to compare Monitor Object pattern solutions with Active Object pattern solutions
  - The key tradeoff is efficiency vs. flexibility

Monitor Object Pattern Dynamics

: Client Thread 1 | **: Client Thread 2** | **: Monitor Object** | **: Monitor Lock** | **: Monitor Condition**

1. *Synchronized method invocation & serialization*
2. *Synchronized method thread suspension*
3. *Monitor condition notification*
4. *Synchronized method thread resumption*

sync_method1()

acquire()

dowork()

wait()

the OS thread scheduler automatically suspends the client thread

sync_method2()

acquire()

the OS thread scheduler atomically releases the monitor lock

the OS thread scheduler automatically resumes the client thread & the synchronized method

dowork()

notify()

release()

dowork()

the OS thread scheduler atomically reacquires the monitor lock

release()

Transparently Parameterizing Synchronization

Problem

- It should be possible to customize component synchronization mechanisms according to the requirements of particular application use cases & configurations
- Hard-coding synchronization strategies into component implementations is *inflexible*
- Maintaining multiple versions of components manually is *not scalable*

Solution

- Apply the *Strategized Locking* design pattern to parameterize component synchronization strategies by making them 'pluggable' types
- Each type objectifies a particular synchronization strategy, such as a mutex, readers/writer lock, semaphore, or 'null' lock
- Instances of these pluggable types can be defined as objects contained within a component, which then uses these objects to synchronize its method implementations efficiently

Applying Strategized Locking to ACE_Message_Queue

```
template <class SYNCH_STRATEGY>
class ACE_Message_Queue {
  // ...
protected:
  // C++ traits that coordinate concurrent access.
  ACE_TYPENAME SYNCH_STRATEGY::MUTEX lock_;
  ACE_TYPENAME SYNCH_STRATEGY::CONDITION notempty_;
  ACE_TYPENAME SYNCH_STRATEGY::CONDITION notfull_;
};
```

Parameterized Strategized Locking

- The traits classes needn't derive from a common base class or use virtual methods!

```
class ACE_NULL_SYNCH {
public:
  typedef ACE_Null_Mutex
          MUTEX;
  typedef ACE_Null_Condition
          CONDITION;
  typedef ACE_Null_Semaphore
          SEMAPHORE;
  // …
};
```

```
class ACE_MT_SYNCH {
public:
  typedef ACE_Thread_Mutex
          MUTEX;
  typedef ACE_Condition_Thread_Mutex
          CONDITION;
  typedef ACE_Thread_Semaphore
          SEMAPHORE;
  // …
};
```

Sidebar: C++ Traits & Traits Class Idioms

- A trait is a type that conveys information used by another class or algorithm to determine policies at compile time
- A traits class is a useful way to collect a set of traits that should be applied in a given situation to alter another class's behavior appropriately
- Traits & traits classes are C++ policy-based class design idioms that are widely used throughout the C++ standard library

- These C++ idioms are similar in spirit to the Strategy pattern, which allows substitution of class behavioral characteristics without requiring a change to the class itself
- The Strategy pattern involves a defined interface that's commonly bound dynamically at run time using virtual methods
- In contrast, the traits & traits class idioms involve substitution of a set of class members and/or methods that can be bound statically at compile time using C++ parameterized types

```
ACE_Message_Queue<ACE_NULL_SYNCH>
  st_mq;
ACE_Message_Block *mb;

// Does not block.
st_mq.dequeue_head (mb);
```

```
ACE_Message_Queue<ACE_MT_SYNCH>
  mt_mq;
ACE_Message_Block *mb;

// Does block.
mt_mq.dequeue_head (mb);
```

Minimizing Unnecessary Locking

Context

- Components in multi-threaded applications that contain intra-component method calls
- Components that have applied the Strategized Locking pattern

Problem

- Thread-safe components should be designed to avoid unnecessary locking
- Thread-safe components should be designed to avoid "self-deadlock"

```
template <class SYNCH_STRAT> int
ACE_Message_Queue<SYNCH_STRAT>::dequeue_head
(ACE_Message_Block &*mb, ACE_Time_Value &tv){
  ACE_GUARD_RETURN (SYNCH_STRAT::MUTEX, g, lock_, -1);
  ...
  while (is_empty ())...
}
template <class SYNCH_STRAT> int
ACE_Message_Queue<SYNCH_STRAT>::is_empty (void) const {
  ACE_GUARD_RETURN (SYNCH_STRAT::MUTEX, g, lock_, -1);
  return cur_bytes_ == 0 && cur_count_ == 0;
}
```

Minimizing Unnecessary Locking

Solution

- Apply the *Thread-safe Interface* design pattern to minimize locking overhead & ensure that intra-component method calls do not incur 'self-deadlock'
- This pattern structures all components that process intra-component method invocations so that interface methods *check* & implementation methods *trust*

```
template <class SYNCH_STRAT> int
ACE_Message_Queue<SYNCH_STRAT>::dequeue_head
(ACE_Message_Block &*mb, ACE_Time_Value &tv) {
  ACE_GUARD_RETURN (SYNCH_STRAT::MUTEX, g, lock_, -1);
  ...
  while (is_empty_i ())...
}
template <class SYNCH_STRAT> int
ACE_Message_Queue<SYNCH_STRAT>::is_empty_i (void) const {
  return cur_bytes_ == 0 && cur_count_ == 0;
}
```

Sidebar: Integrating ACE_Message_Queue & ACE_Reactor

- Some platforms can integrate native message queue events with synchronous event demultiplexing
  - e.g., AIX's **`select()`** can demux events generated by System V message queues
- Although this use of **`select()`** is nonportable, it's useful to integrate a message queue with a reactor in many applications
  - **`ACE_Message_Queue`** therefore offers a portable way to integrate event queueing with the ACE Reactor framework

- The **`ACE_Message_Queue`** class contains methods that can set a notification strategy
- This notification strategy must be derived from **`ACE_Notification_Strategy`**, which allows the flexibility to insert any strategy necessary for your application
- **`ACE_Reactor_Notification_Strategy`**'s constructor associates it with an **`ACE_Reactor`**, an **`ACE_Event_Handler`**, & an event mask
- After the strategy object is associated with an **`ACE_Message_Queue`**, each queued message triggers the following sequence of actions
  - **`ACE_Message_Queue`** calls the strategy's **`notify()`** method
  - **`ACE_Reactor_Notification_Strategy`**'s **`notify()`** method notifies the associated reactor using the reactor notification mechanism
  - The reactor dispatches the notification to the specified event handler using the designated mask

Sidebar: The ACE_Message_Queue_Ex Class

- The **ACE_Message_Queue** class enqueues & dequeues **ACE_Message_Block** objects, which provide a dynamically extensible way to represent messages
- For programs requiring strongly typed messaging, ACE provides the **ACE_Message_Queue_Ex** class, which enqueues & dequeues messages that are instances of a **MESSAGE_TYPE** template parameter, rather than an **ACE_Message_Block**

- **ACE_Message_Queue_Ex** offers the same capabilities as **ACE_Message_Queue**
- Its primary advantage is that application-defined data types can be queued without the need to type cast on enqueue & dequeue or copy objects into the data portion of an **ACE_Message_Block**
- Since **ACE_Message_Queue_Ex** is not derived from **ACE_Message_Queue**, however, it can't be used with the **ACE_Task** class

```
template <class SYNCH_STRATEGY,
          class MESSAGE_TYPE>
class ACE_Message_Queue_Ex {
  int enqueue_tail (MESSAGE_TYPE *, ACE_Time_Value *);
  // ...
};
```

Sidebar: ACE_Message_Queue Shutdown Protocols

- To avoid losing queued messages unexpectedly when an **`ACE_Message_Queue`** needs to be closed, producer & consumer threads can implement the following protocol:

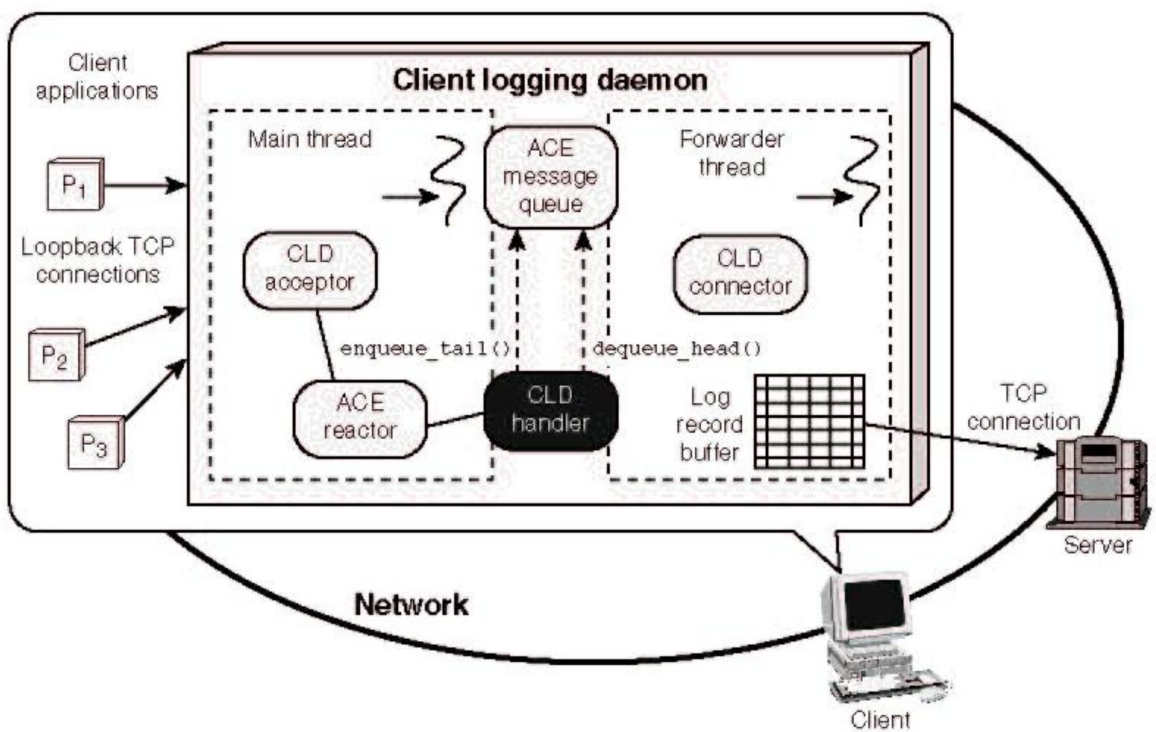
  1. A producer thread can enqueue a special message, such as a message block whose payload is size 0 and/or whose type is **`MB_STOP`**, to indicate that it wants the queue closed
  2. The consumer thread can close the queue when it receives this shutdown message, after processing any other messages ahead of it in the queue

- A variant of this protocol can use **`ACE_Message_Queue::enqueue_prio()`** to boost the priority of the shutdown message so it takes precedence over lower-priority messages that may already reside in the queue

- There are other methods that can be used to close or temporarily deactivate an **`ACE_Message_Queue`**:
  - **`flush()`**, releases the messages in a queue, but doesn't change its state
  - **`deactivate()`**, changes the queue state to **`DEACTIVATED`** & wakes up all threads waiting on enqueue/dequeue operations, but doesn't release any queued messages

Using the ACE_Message_Queue Class (1/20)

- This example shows how **`ACE_Message_Queue`** can be used to implement a client logging daemon
- The implementation uses a producer/consumer concurrency model where separate threads handle input & output processing



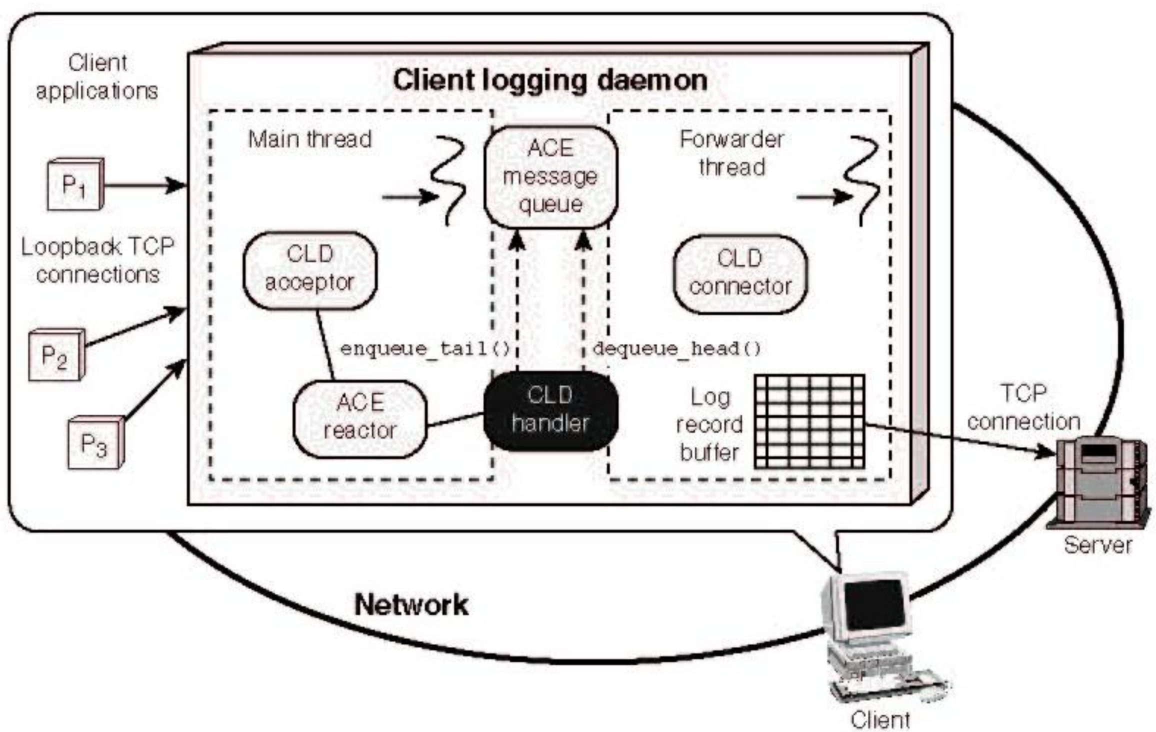
Using the ACE_Message_Queue Class (2/20)

Input Processing

- The main thread uses an event handler & ACE Reactor framework to read log records from sockets connected to client applications via network loopback
- The event handler queues each log record in the synchronized **`ACE_Message_Queue`**

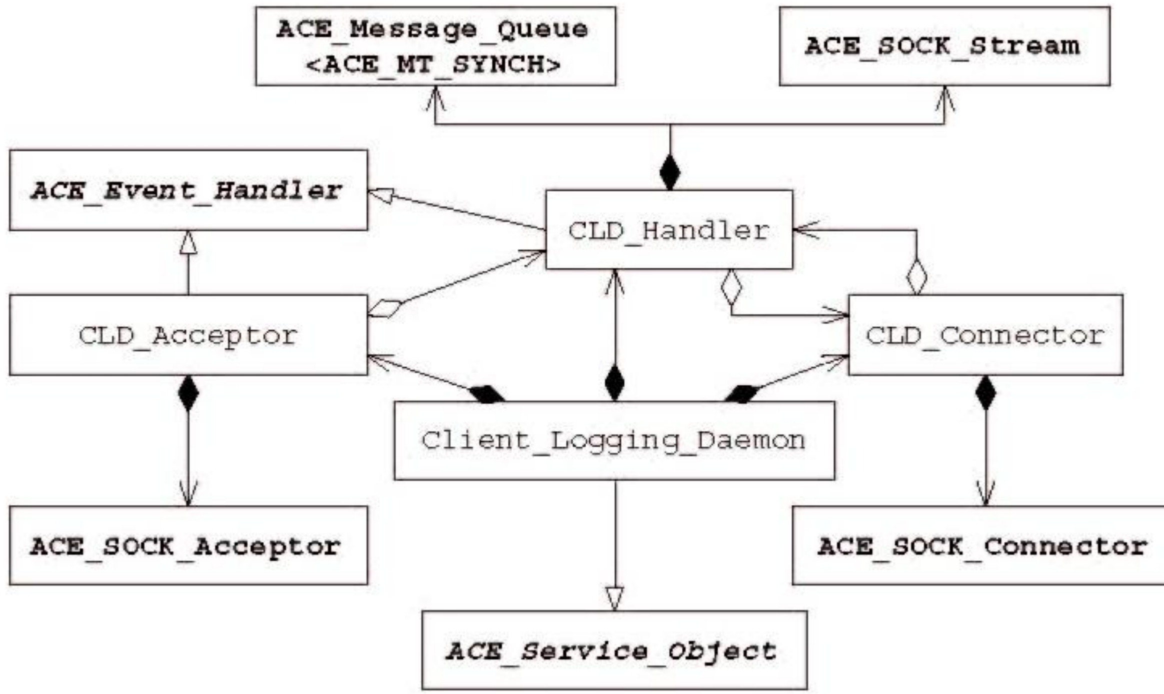
Output Processing

- A separate forwarder thread runs concurrently, performing following steps:
  - Dequeueing messages from the message queue
  - Buffering messages into larger chunks
  - Forwarding the chunks to the server logging daemon over a TCP connection



Using the ACE_Message_Queue Class (3/20)

- **CLD_Handler**: Target of callbacks from the **ACE_Reactor** that receives log records from clients, converts them into **ACE_Message_Blocks**, & inserts them into the synchronized message queue that's processed by a separate thread & forwarded to the logging server



- **CLD_Acceptor**: A factory that passively accepts connections from clients & registers them with the **ACE_Reactor** to be processed by the **CLD_Handler**
- **CLD_Connector**: A factory that actively establishes (& when necessary reestablishes) connections with the logging server
- **Client_Logging_Daemon**: A facade class that integrates the other three classes together

Using the ACE_Message_Queue Class (4/20)

```
#if !defined (FLUSH_TIMEOUT)
#define FLUSH_TIMEOUT 120 /* 120 seconds == 2 minutes. */
#endif /* FLUSH_TIMEOUT */

class CLD_Handler : public ACE_Event_Handler {
public:
  enum { QUEUE_MAX = sizeof (ACE_Log_Record) * ACE_IOV_MAX
};

  // Initialization hook method.
  virtual int open (CLD_Connector *);
  // Shutdown hook method.
  virtual int close ();

  // Accessor to the connection to the logging server.
  virtual ACE_SOCK_Stream &peer () { return peer_; }

  virtual int handle_input (ACE_HANDLE handle);
  virtual int handle_close (ACE_HANDLE = ACE_INVALID_HANDLE,
                            ACE_Reactor_Mask = 0);
```

Maximum size of the queue

Reactor hook methods

Using the ACE_Message_Queue Class (5/20)

```
protected:
  // Forward log records to the server logging daemon.
  virtual ACE_THR_FUNC_RETURN forward ();

  // Send buffered log records using a gather-write operation.
  virtual int send (ACE_Message_Block *chunk[], size_t count);

  // Entry point into forwarder thread of control.
  static ACE_THR_FUNC_RETURN run_svc (void *arg);
```

Adapter function

```
  // A synchronized <ACE_Message_Queue> that queues messages.
  ACE_Message_Queue<ACE_MT_SYNCH> msg_queue_;
```

Note the use of the ACE_MT_SYNCH traits class

```
  ACE_Thread_Manager thr_mgr_; // Manage the forwarder thread.

  CLD_Connector *connector_; // Pointer to our
<CLD_Connector>.

  ACE_SOCK_Stream peer_; // Connection to logging server.
```

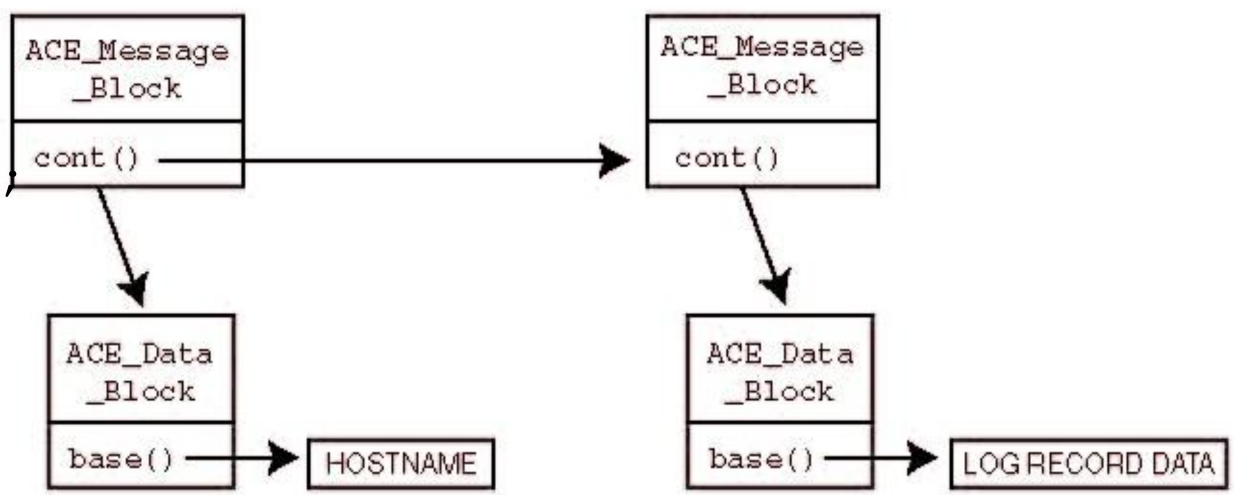
Using the ACE_Message_Queue Class (6/20)

Hook method dispatched by reactor

```
int CLD_Handler::handle_input (ACE_HANDLE handle) {
  ACE_Message_Block *mblk = 0;
  Logging_Handler logging_handler (handle);
```

Note decoupling of read vs. write for log record

```
  if (logging_handler.recv_log_record (mblk) != -1)
    if (msg_queue_.enqueue_tail (mblk->cont ()) != -1)
 {
      mblk->cont (0); mblk->release ();
      return 0; // Success.
  }
  else
    mblk->release ();
  // Error return.
 return -1;
}
```



Using the ACE_Message_Queue Class (7/20)

```
int CLD_Handler::open (CLD_Connector *connector) {
  connector_ = connector;
  int bufsiz = ACE_DEFAULT_MAX_SOCKET_BUFSIZ;
  peer ().set_option (SOL_SOCKET, SO_SNDBUF,
                      &bufsiz, sizeof bufsiz);
  msg_queue_.high_water_mark (CLD_Handler::QUEUE_MAX);
  return thr_mgr_.spawn (&CLD_Handler::run_svc,
                         this, THR_SCOPE_SYSTEM);
}
```

Create new thread of control that invokes run_svc() adapter function

```
ACE_THR_FUNC_RETURN CLD_Handler::run_svc (void *arg) {
  CLD_Handler *handler = ACE_static_cast (CLD_Handler *,
arg);
  return handler->forward ();
}
```

Adapter function forward messages to server logging daemon

Using the ACE_Message_Queue Class (8/20)

```
ACE_THR_FUNC_RETURN CLD_Handler::forward () {
  ACE_Message_Block *chunk[ACE_IOV_MAX];
  size_t message_index = 0;
  ACE_Time_Value time_of_last_send (ACE_OS::gettimeofday
());
  ACE_Time_Value timeout;
  ACE_Sig_Action no_sigpipe ((ACE_SignalHandler) SIG_IGN);
  ACE_Sig_Action original_action;
  no_sigpipe.register_action (SIGPIPE, &original_action);

  for (;;) {
    if (message_index == 0) {
      timeout = ACE_OS::gettimeofday ();
      timeout += FLUSH_TIMEOUT;
    }
    ACE_Message_Block *mblk = 0;
    if (msg_queue_.dequeue_head (mblk, &timeout) == -1) {
      if (errno != EWOULDBLOCK) break;
      else if (message_index == 0) continue;
    } else {
      if (mblk->size () == 0
          && mblk->msg_type () ==
```

Ignore SIGPIPE signal

Wait a bounded period of time for next message

Shutdown protocol

Using the ACE_Message_Queue Class (9/20)

```
         chunk[message_index] = mblk;
         ++message_index;
       }
       if (message_index >= ACE_IOV_MAX
           || (ACE_OS::gettimeofday () -
time_of_last_send
                 >= FLUSH_TIMEOUT)) {
         if (send (chunk, message_index) == -1) break;
         time_of_last_send = ACE_OS::gettimeofday ();
       }
     }

     if (message_index > 0) send (chunk, message_index);
     msg_queue_.close ();
     no_sigpipe.restore_action (SIGPIPE,
original_action);
     return 0;
   }
```

Send buffered messages at appropriate time

Send any remaining buffered messages

Restore signal disposition

Using the ACE_Message_Queue Class (10/20)

```
int CLD_Handler::send (ACE_Message_Block *chunk[],
                       size_t &count) {
  iovec iov[ACE_IOV_MAX];
  size_t iov_size;
  int result = 0;

  for (iov_size = 0; iov_size < count; ++iov_size) {
    iov[iov_size].iov_base = chunk[iov_size]->rd_ptr
();
    iov[iov_size].iov_len = chunk[iov_size]->length ();
  }

  while (peer ().sendv_n (iov, iov_size) == -1)
    if (connector_->reconnect () == -1) {
      result = -1;
      break;
    }

```

Initialize gather-write buffer

Send gather-write buffer

Trigger reconnection upon failed send

Using the ACE_Message_Queue Class (11/20)

```
  while (iov_size > 0) {
    chunk[--iov_size]->release (); chunk[iov_size] = 0;
  }
  count = iov_size;
  return result;
}
```

Release dynamically allocated buffers

```
int CLD_Handler::close () {
  ACE_Message_Block *shutdown_message = 0;
  ACE_NEW_RETURN
    (shutdown_message,
     ACE_Message_Block (0, ACE_Message_Block::MB_STOP),
-1);
  msg_queue_.enqueue_tail (shutdown_message);
  return thr_mgr_.wait ();
}
```

Initiate shutdown protocol

Barrier synchronization

Using the ACE_Message_Queue Class (12/20)

```
class CLD_Acceptor : public ACE_Event_Handler {
public:
  // Initialization hook method.
  virtual int open (CLD_Handler *, const ACE_INET_Addr &,
                    ACE_Reactor * = ACE_Reactor::instance
());
  virtual int handle_input (ACE_HANDLE handle);
  virtual int handle_close (ACE_HANDLE = ACE_INVALID_HANDLE,
                            ACE_Reactor_Mask = 0);
  virtual ACE_HANDLE get_handle () const;
```

Reactor hook methods

```
protected:
  ACE_SOCK_Acceptor acceptor_;
```

Factory that connects ACE_SOCK_Stream's passively

```
  // Pointer to the handler of log records.
  CLD_Handler *handler_;
};
```

Using the ACE_Message_Queue Class (13/20)

```
int CLD_Acceptor::open
  (CLD_Handler *h, const ACE_INET_Addr &addr, ACE_Reactor *r)
{
  reactor (r); // Store reactor pointer.
  handler_ = h;
  if (acceptor_.open (addr) == -1
      || reactor ()->register_handler
            (this, ACE_Event_Handler::ACCEPT_MASK) == -1)
    return -1;
  return 0;
}
int CLD_Acceptor::handle_input (ACE_HANDLE) {
  ACE_SOCK_Stream peer_stream;
  if (acceptor_.accept (peer_stream) == -1) return
-1;
  else if (reactor ()->register_handler
             (peer_stream.get_handle (),
              handler_,
              ACE_Event_Handler::READ_MASK) == -1)
   return -1;
 else return 0;
```

Listen for connections

Register for connection events

Reactor dispatches this method

Register for read events

Using the ACE_Message_Queue Class (14/20)

```
class CLD_Connector {
public:
  // Establish connection to logging server at <remote_addr>.
  int connect (CLD_Handler *handler,
               const ACE_INET_Addr &remote_addr);

  // Re-establish a connection to the logging server.
  int reconnect ();

private:
  // Pointer to the <CLD_Handler> that we're connecting.
  CLD_Handler *handler_;

  // Address at which the logging server is listening
  // for connections.
  ACE_INET_Addr remote_addr_;
}
```

Using the ACE_Message_Queue Class (15/20)

```
int CLD_Connector::connect
     (CLD_Handler *handler,
      const ACE_INET_Addr &remote_addr) {
  ACE_SOCK_Connector connector;

  if (connector.connect (handler->peer (), remote_addr) == -1)
    return -1;
  else if (handler->open (this) == -1)
  { handler->handle_close (); return -1; }
  handler_ = handler;
  remote_addr_ = remote_addr;
  return 0;
}
```

These steps form the core part of the active side of the Acceptor/Connector pattern

Using the ACE_Message_Queue Class (16/20)

```
int CLD_Connector::reconnect () {
  // Maximum # of times to retry connect.
  const size_t MAX_RETRIES = 5;

  ACE_SOCK_Connector connector;
  ACE_Time_Value timeout (1); // Start with 1 second
timeout.
  size_t i;
  for (i = 0; i < MAX_RETRIES; ++i) {
    if (i > 0) ACE_OS::sleep (timeout);
    if (connector.connect (handler_->peer (), remote_addr_,
                           &timeout) == -1)
      timeout *= 2;
    else {
      int bufsiz = ACE_DEFAULT_MAX_SOCKET_BUFSIZ;
      handler_->peer ().set_option (SOL_SOCKET, SO_SNDBUF,
                                    &bufsiz, sizeof bufsiz);
      break;
    }
  }
  return i == MAX_RETRIES ? -1 : 0;
}
```

Called when connection has broken

Exponential backoff algorithm

Using the ACE_Message_Queue Class (17/20)

- This class brings together all parts of the client logging daemon

```
class Client_Logging_Daemon : public ACE_Service_Object {
public:

  virtual int init (int argc, ACE_TCHAR *argv[]);
  virtual int fini ();
  virtual int info (ACE_TCHAR **bufferp, size_t length = 0)
const;
  virtual int suspend ();
  virtual int resume ();

protected:
  // Receives, processes, & forwards log records.
  CLD_Handler handler_;

  // Factory that passively connects the <CLD_Handler>.
  CLD_Acceptor acceptor_;

  // Factory that actively connects the <CLD_Handler>.
  CLD_Connector connector_;
};
```

Enables dynamic linking

Service Configurator hook methods

Using the ACE_Message_Queue Class (18/20)

Initialization hook method called by ACE Service Configurator framework

```
int Client_Logging_Daemon::init (int argc, ACE_TCHAR *argv[])
{
  u_short cld_port = ACE_DEFAULT_SERVICE_PORT;
  u_short sld_port = ACE_DEFAULT_LOGGING_SERVER_PORT;
  ACE_TCHAR sld_host[MAXHOSTNAMELEN];
  ACE_OS_String::strcpy (sld_host, ACE_LOCALHOST);

  ACE_Get_Opt get_opt (argc, argv, ACE_TEXT ("p:r:s:"), 0);
  get_opt.long_option (ACE_TEXT ("client_port"), 'p',
                       ACE_Get_Opt::ARG_REQUIRED);
  get_opt.long_option (ACE_TEXT ("server_port"), 'r',
                       ACE_Get_Opt::ARG_REQUIRED);
  get_opt.long_option (ACE_TEXT ("server_name"), 's',
                       ACE_Get_Opt::ARG_REQUIRED);

  for (int c; (c = get_opt ()) != -1;)
    switch (c) {
    case 'p': // Client logging daemon acceptor port number.
      cld_port = ACE_static_cast
        (u_short, ACE_OS::atoi (get_opt.opt_arg ()));
      break;
```

Using the ACE_Message_Queue Class (19/20)

```
    case 'r': // Server logging daemon acceptor port number.
      sld_port = ACE_static_cast
        (u_short, ACE_OS::atoi (get_opt.opt_arg ()));
      break;
    case 's': // Server logging daemon hostname.
      ACE_OS_String::strsncpy
        (sld_host, get_opt.opt_arg (), MAXHOSTNAMELEN);
      break;
    }

  ACE_INET_Addr cld_addr (cld_port);
  ACE_INET_Addr sld_addr (sld_port, sld_host);

  if (acceptor_.open (&handler_, cld_addr) == -1)
    return -1;
  else if (connector_.connect (&handler_, sld_addr) == -1)
  { acceptor_.handle_close (); return -1; }
  return 0;
}
```

Establish connection passively

Establish connection actively

Using the ACE_Message_Queue Class (20/20)

Create entry point for ACE Service Configurator framework

```
ACE_FACTORY_DEFINE (CLD,
Client_Logging_Daemon)
```

`svc.conf` file for client logging daemon

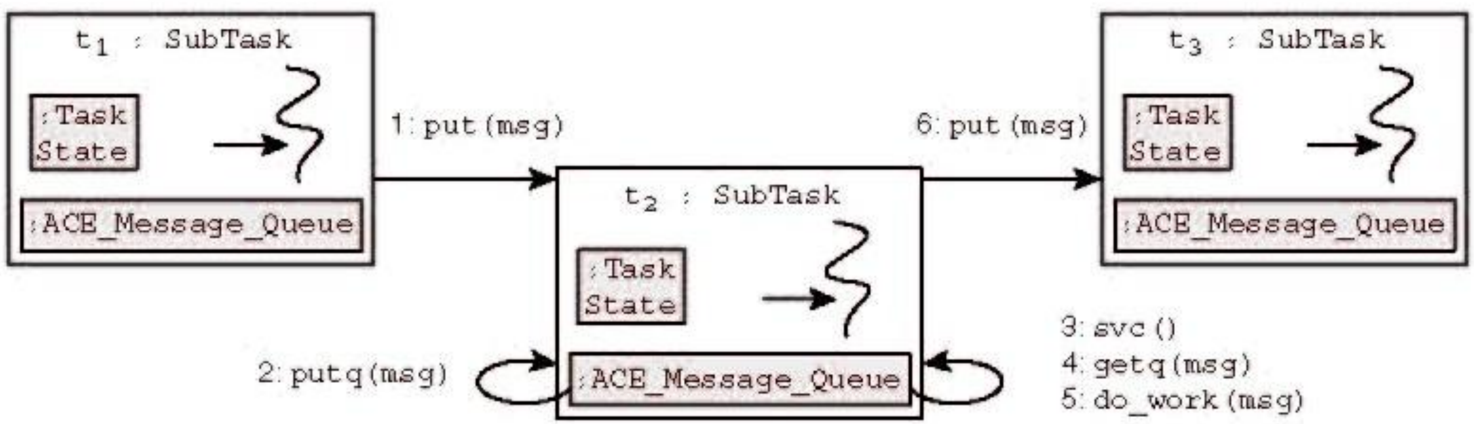
```
dynamic Client_Logging_Daemon Service_Object *
CLD:_make_Client_Logging_Daemon()
      "-p $CLIENT_LOGGING_DAEMON_PORT"
```

> The main() function is the same as the one we showed for the ACE Service Configurator example!!!!

The ACE_Task Class (1/2)

Motivation

- The **`ACE_Message_Queue`** class can be used to
  - Decouple the flow of information from its processing
  - Link threads that execute producer/consumer services concurrently
- To use a producer/consumer concurrency model effectively in an object-oriented program, however, each thread should be associated with the message queue & any other service-related information
- To preserve modularity & cohesion, & to reduce coupling, it's therefore best to encapsulate an **`ACE_Message_Queue`** with its associated data & methods into one class whose service threads can access it directly

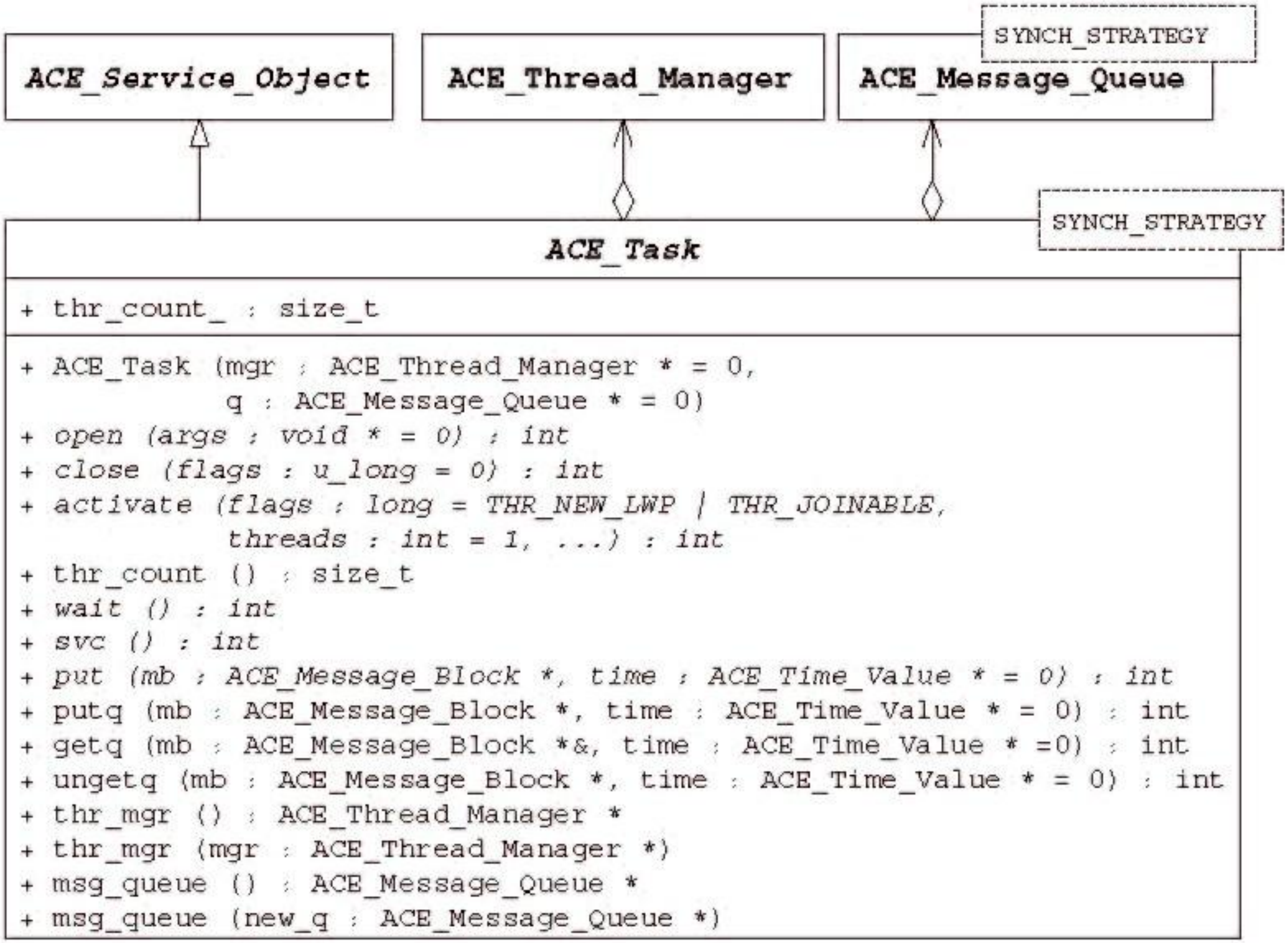


The ACE_Task Class (2/2)

Class Capabilities

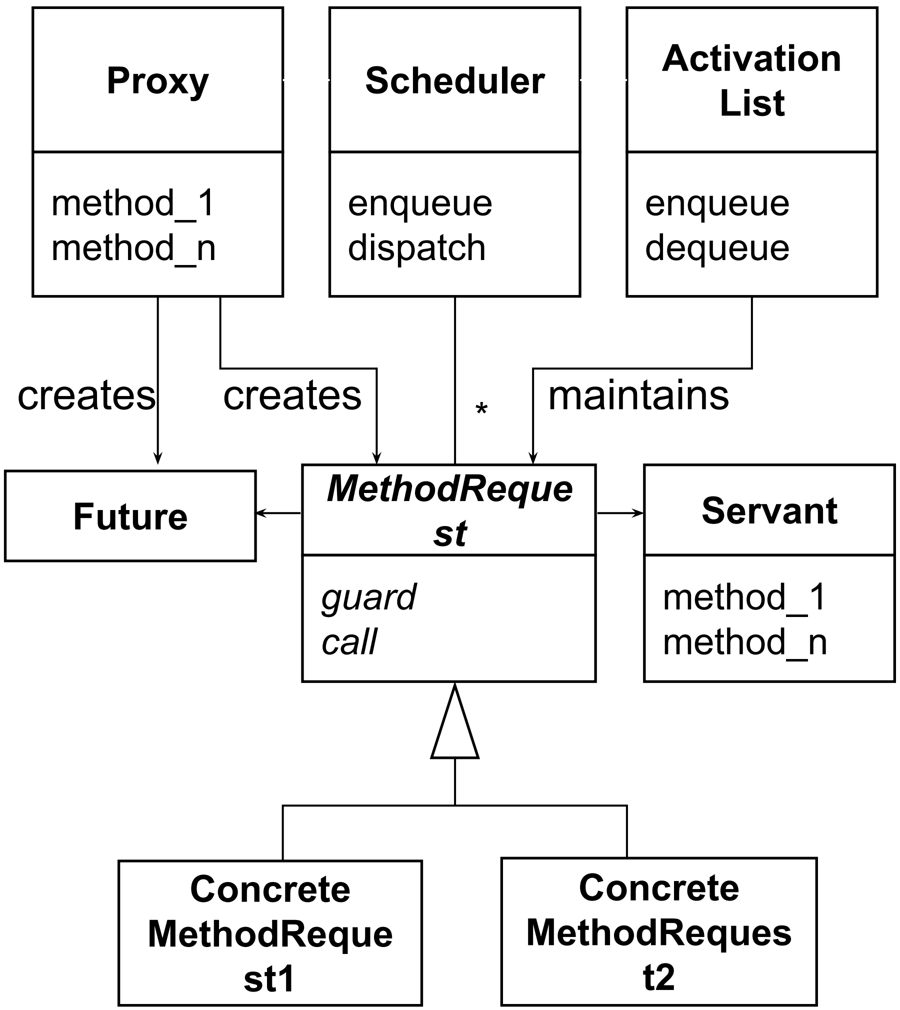
- `ACE_Task` is the basis of ACE's OO concurrency framework that provides the following capabilities:
  - It uses an `ACE_Message_Queue` to separate data & requests from their processing
  - It uses `ACE_Thread_Manager` to activate the task so it runs as an *active object* that processes its queued messages in one or more threads
  - Since each thread runs a designated class method, they can access all of the task's data members directly
  - It inherits from `ACE_Service_Object`, so its instances can be configured dynamically via the ACE Service Configurator framework
  - It's a descendant of `ACE_Event_Handler`, so its instances can also serve as event handlers in the ACE Reactor framework
  - It provides virtual hook methods that application classes can reimplement for task-specific service execution & message handling

The ACE_Task Class API



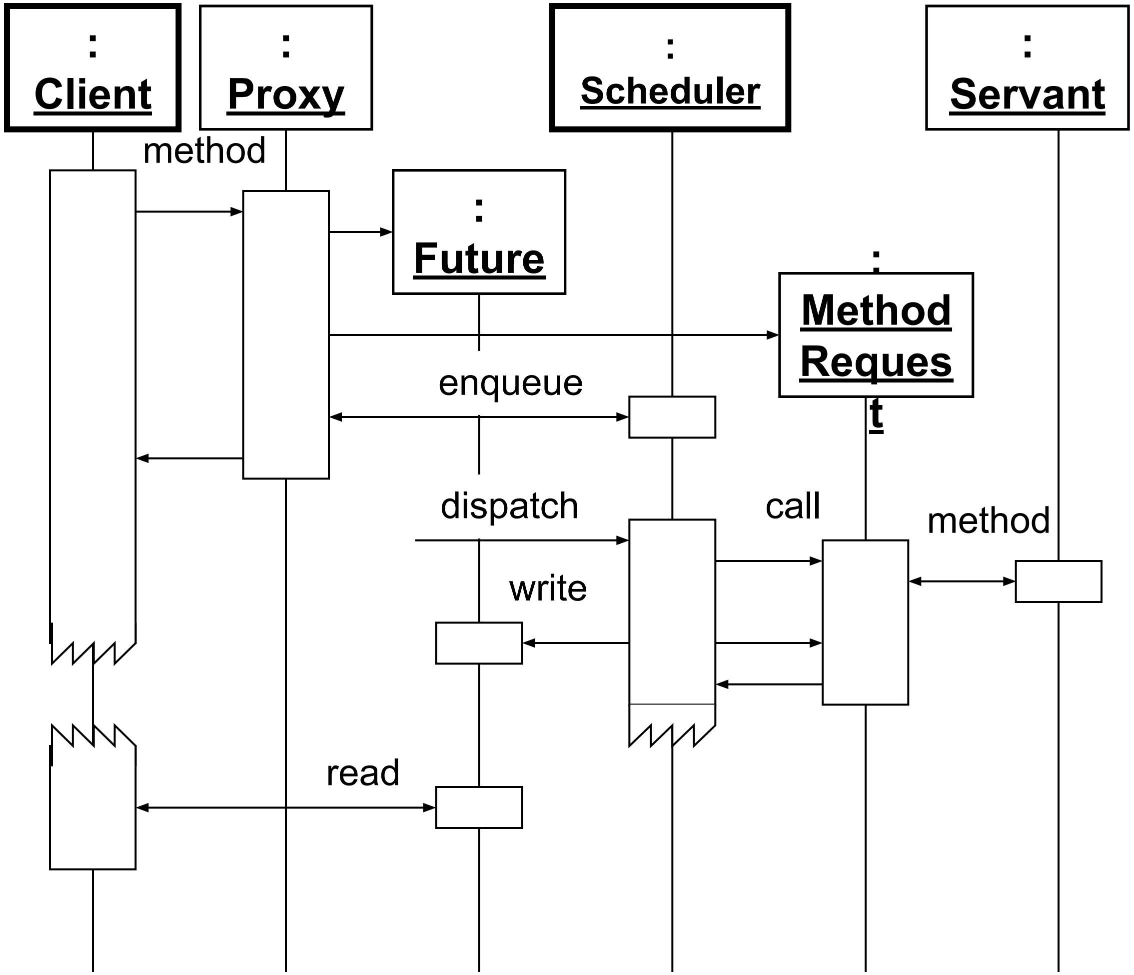
The Active Object Pattern

- The *Active Object* design pattern decouples method invocation from method execution using an object-oriented programming model



- A *proxy* provides an interface that allows clients to access methods of an object
- A *concrete method request* is created for every method invoked on the proxy
- A *scheduler* receives the method requests & dispatches them on the servant when they become runnable
- An *activation list* maintains pending method requests
- A *servant* implements the methods
- A *future* allows clients to access the results of a method call on the proxy

Active Object Pattern Dynamics



- A *client* invokes a method on the *proxy*
- The *proxy* returns a future to the client, & creates a *method request, *which it passes to the *scheduler*
- The *scheduler* enqueues the *method request* into the *activation list* (not shown here)
- When the *method request* becomes runnable, the *scheduler* dequeues it from the *activation list* (not shown here) & executes it in a different thread than the client
- The *method request* executes the method on the *servant* & writes results, if any, to the *future*
- *Clients* obtain the method's results via the *future*

Clients can obtain result from futures via blocking, polling, or callbacks

Pros & Cons of the Active Object Pattern

This pattern provides four **benefits**:

- ***Enhanced type-safety***
  - Cf. async forwarder/receiver message passing
- ***Enhances concurrency & simplifies synchronized complexity***
  - Concurrency is enhanced by allowing client threads & asynchronous method executions to run simultaneously
  - Synchronization complexity is simplified by using a scheduler that evaluates synchronization constraints to serialized access to servants
- ***Transparent leveraging of available parallelism***
  - Multiple active object methods can execute in parallel if supported by the OS/hardware
- ***Method execution order can differ from method invocation order***
  - Methods invoked asynchronous are executed according to the synchronization constraints defined by their guards & by scheduling policies
  - Methods can be "batched" & sent wholesale to enhance throughput

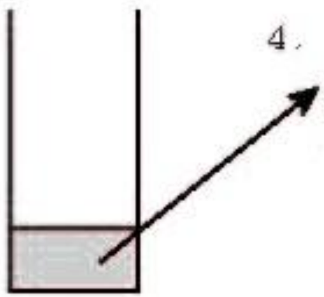
This pattern also has some **liabilities**:

- ***Higher overhead***
  - Depending on how an active object's scheduler is implemented, context switching, synchronization, & data movement overhead may occur when scheduling & executing active object invocations
- ***Complicated debugging***
  - It is hard to debug programs that use the Active Object pattern due to the concurrency & non-determinism of the various active object schedulers & the underlying OS thread scheduler

Activating an ACE_Task

- **`ACE_Task::svc_run()`** is a static method used by **`activate()`** as an adapter function
- It runs in the newly spawned thread(s) of control, which provide an execution context for the **`svc()`** hook method
- The following illustrates the steps associated with activating an **`ACE_Task`** using the Windows **`_beginthreadex()`** function to spawn the thread

```
1. ACE_Task::activate()
2. ACE_Thread_Manager::spawn
      (svc_run, this);
3. _beginthreadex
      (0, 0,
      svc_run, this,
      0, &thread_id);
```



Run-time thread stack

```
4. template <SYNCH_STRATEGY>
   ACE_THR_FUNC_RETURN
   ACE_Task<SYNCH_STRATEGY>::svc_run
     (ACE_Task<SYNCH_STRATEGY> *t) {
     // ...
     int status = t->svc();
     // ...
     return reinterpret_cast
    (ACE_THR_FUNC_RETURN, status);
   }
```

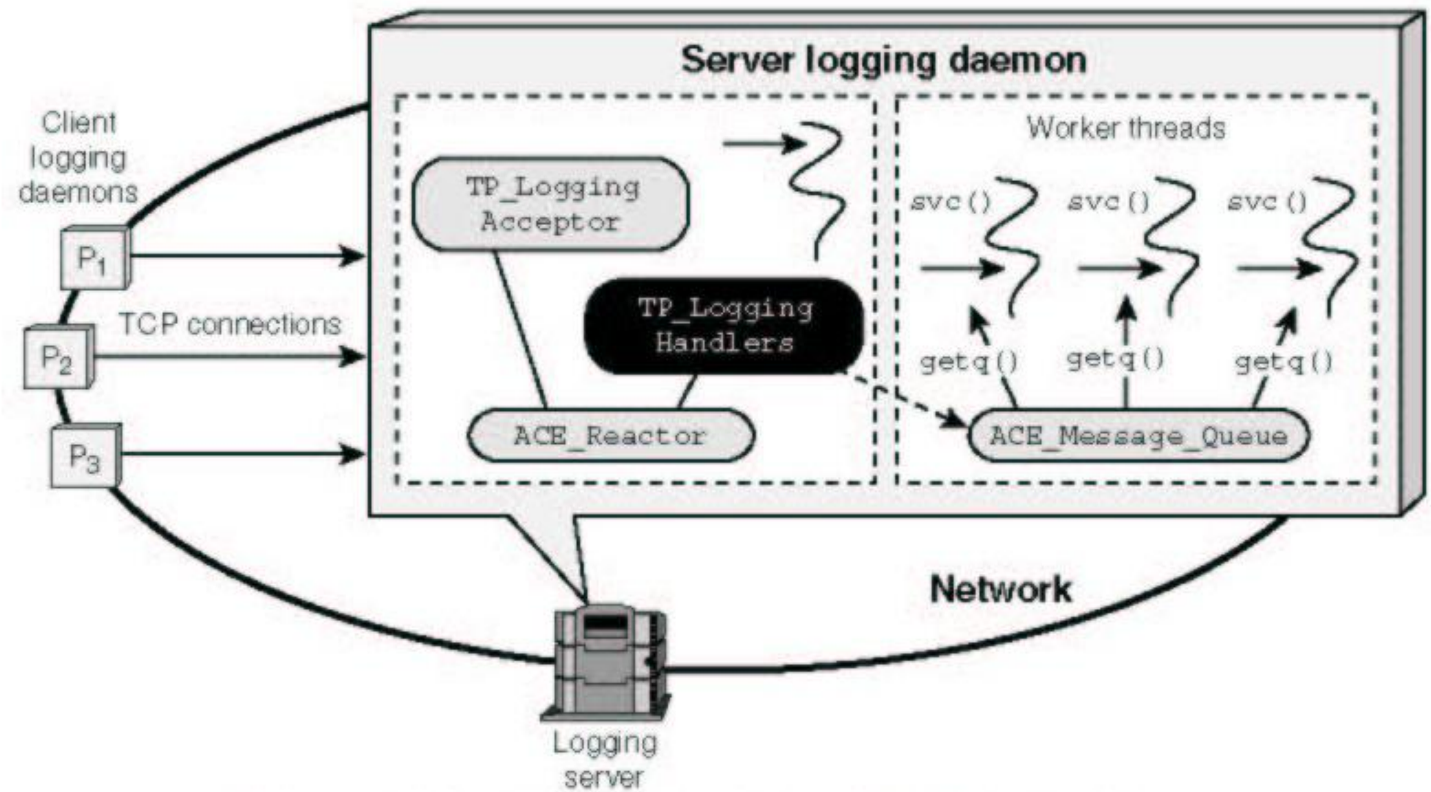
- Naturally, the **`ACE_Task`** class shields applications from OS-specific details

Sidebar: Comparing ACE_Task with Java Threads

- **`ACE_Task::activate()`** is similar to the Java **`Thread.start()`** method since they both spawn internal threads
  - The Java **`Thread.start()`** method spawns only one thread, whereas **`activate()`** can spawn multiple threads within the same **`ACE_Task`**, making it easy to implement thread pools
- **`ACE_Task::svc()`** is similar to the Java **`Runnable.run()`** method since both methods are hooks that run in newly spawned thread(s)
  - The Java **`run()`** hook method executes in only a single thread per object, whereas the **`ACE_Task::svc()`** method can execute in multiple threads per task object
- **`ACE_Task`** contains a message queue that allows applications to exchange & buffer messages
  - In contrast, this type of queueing capability must be added by Java developers explicitly

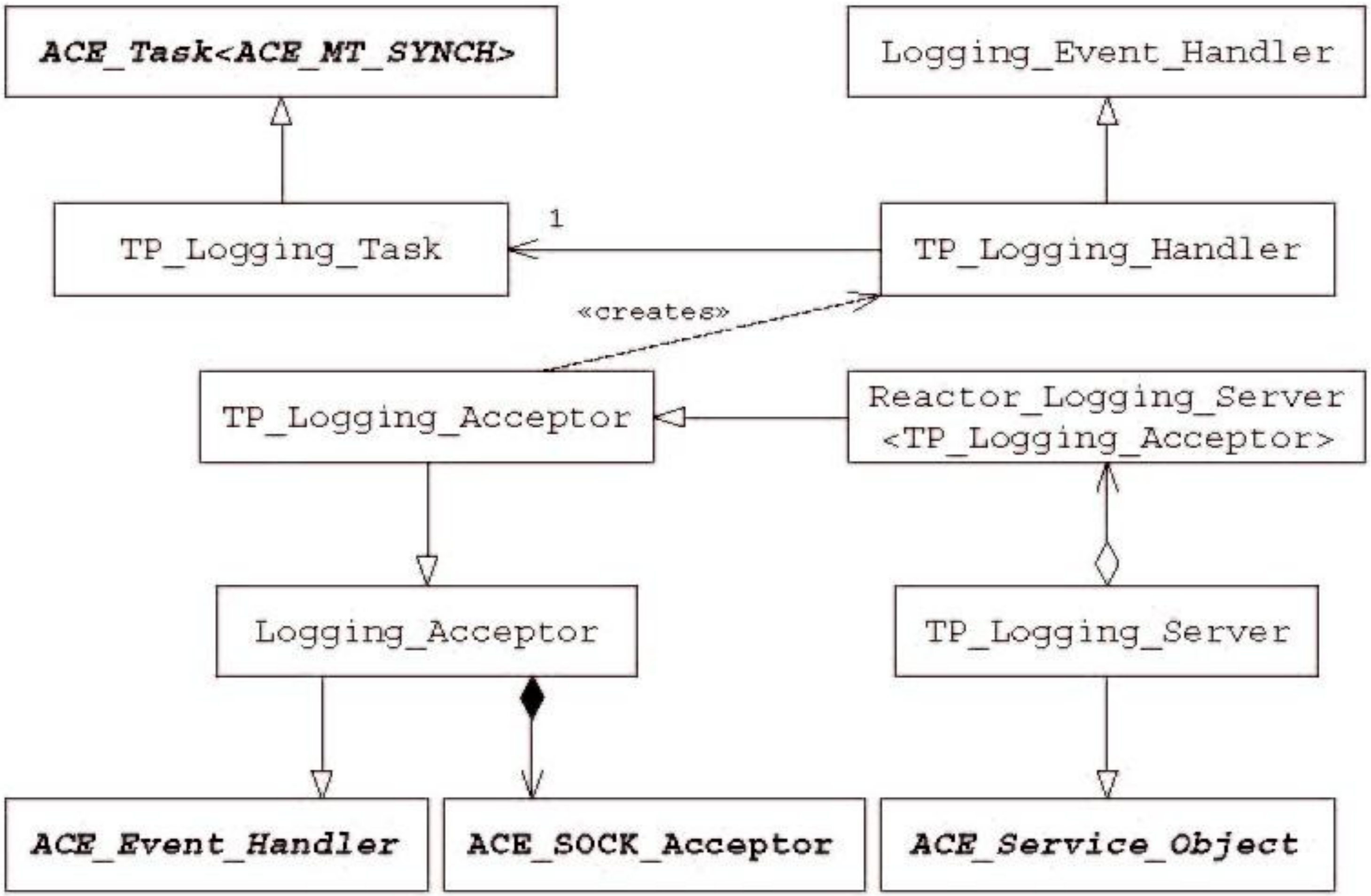
Using the ACE_Task Class (1/13)

- This example combines **`ACE_Task`** & **`ACE_Message_Queue`** with the **`ACE_Reactor`** & **`ACE_Service_Config`** to implement a concurrent server logging daemon using the thread pool concurrency model



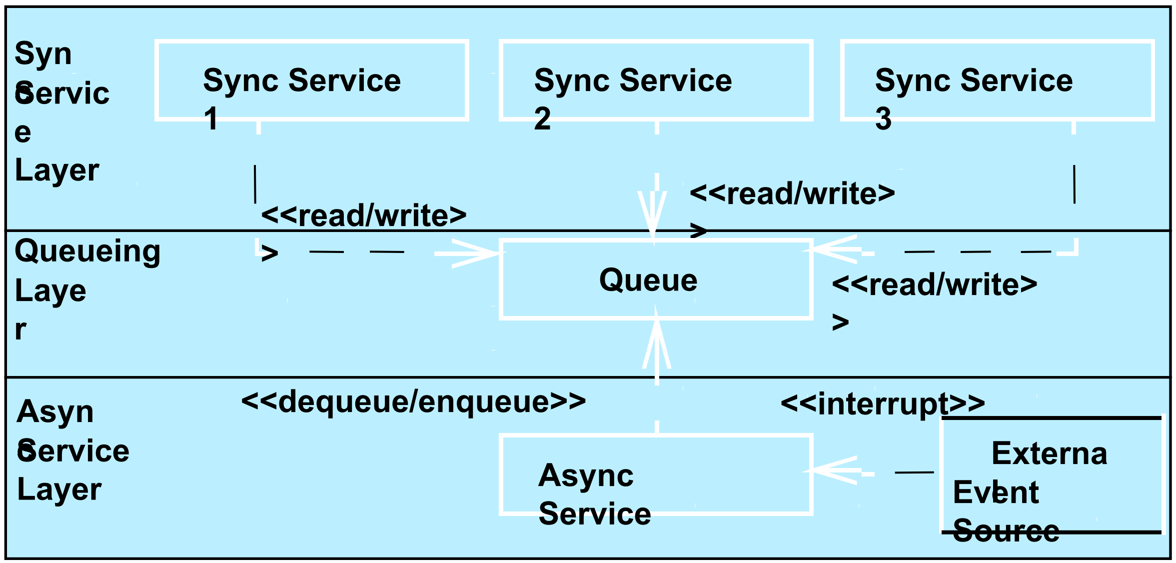
Using the ACE_Task Class (2/13)

- This server design is based on the *Half Sync/Half-Async* pattern & the eager spawning thread pool strategy



The Half-Sync/Half-Async Pattern

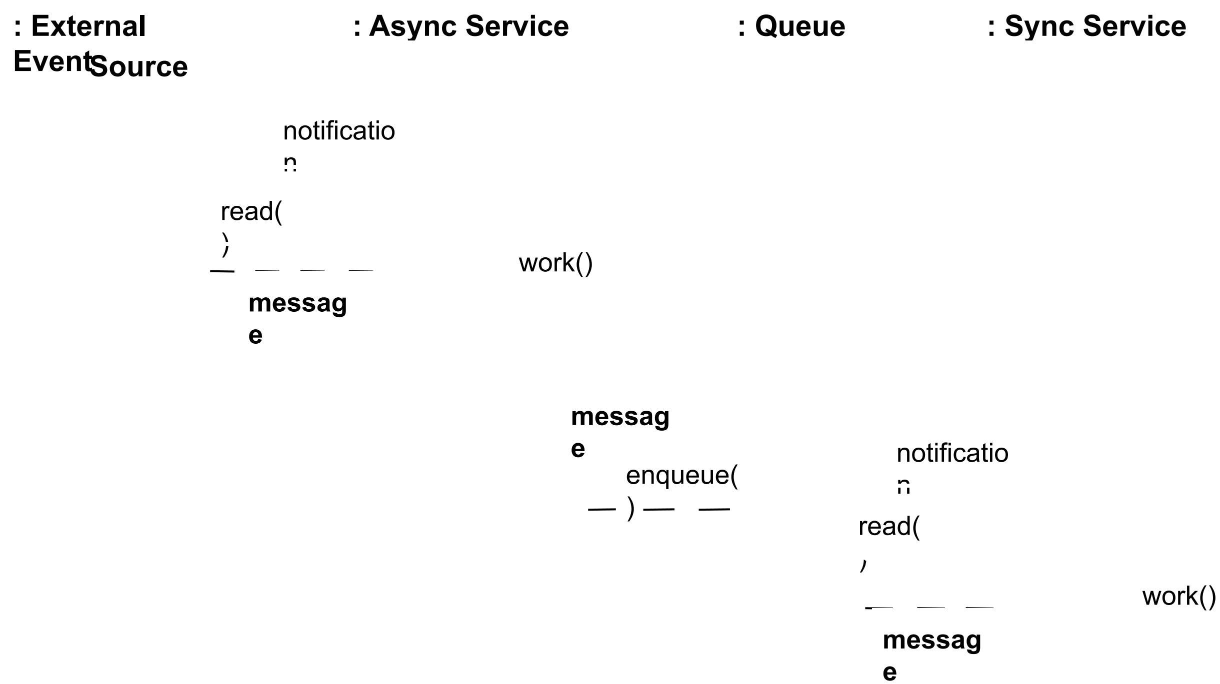
The *Half-Sync/Half-Async* architectural pattern decouples async & sync service processing in concurrent systems, to simplify programming without unduly reducing performance



This solution yields two benefits:

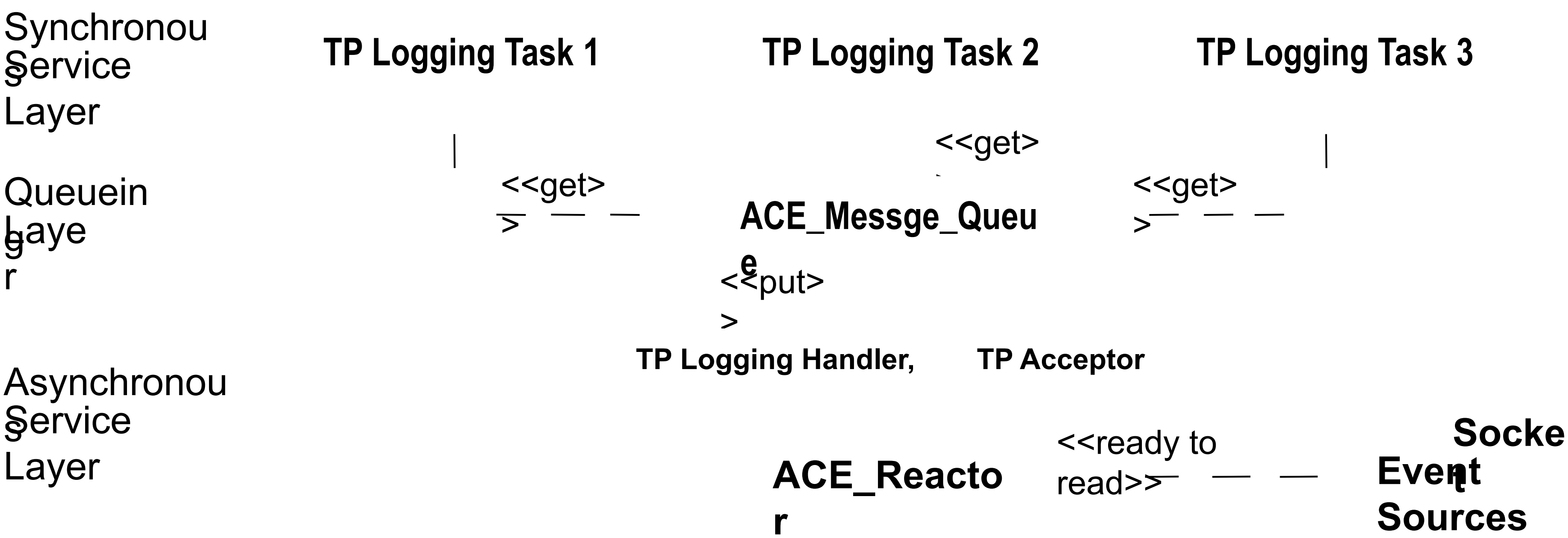
1. Threads can be mapped to separate CPUs to scale up server performance via multi-processing
2. Each thread blocks independently, which prevents a flow-controlled connection from degrading the QoS that other clients receive

Half-Sync/Half-Async Pattern Dynamics



- This pattern defines two service processing layers—one async & one sync—along with a queueing layer that allows services to exchange messages between the two layers
- The pattern allows sync services, such as logging record protocol processing, to run concurrently, relative both to each other & to async services, such as event demultiplexing

Applying Half-Sync/Half-Async Pattern



- Server logging daemon uses Half-Sync/Half-Async pattern to process logging records from multiple clients concurrently in separate threads
- `TP_Logging_Task` removes the request from a synchronized message queue & stores the logging record in a file
- If flow control occurs on its client connection this thread can block without degrading the QoS experienced by clients serviced by other threads in the pool

Pros & Cons of Half-Sync/Half-Async Pattern

This pattern has three **benefits**:

- ***Simplification & performance***
  - The programming of higher-level synchronous processing services are simplified without degrading the performance of lower-level system services
- ***Separation of concerns***
  - Synchronization policies in each layer are decoupled so that each layer need not use the same concurrency control strategies
- ***Centralization of inter-layer communication***
  - Inter-layer communication is centralized at a single access point, because all interaction is mediated by the queueing layer

This pattern also incurs **liabilities**:

- ***A boundary-crossing penalty may be incurred***
  - This overhead arises from context switching, synchronization, & data copying overhead when data is transferred between the sync & async service layers via the queueing layer
- ***Higher-level application services may not benefit from the efficiency of async I/O***
  - Depending on the design of operating system or application framework interfaces, it may not be possible for higher-level services to use low-level async I/O devices effectively
- ***Complexity of debugging & testing***
  - Applications written with this pattern can be hard to debug due its concurrent execution

Using the ACE_Task Class (3/13)

```
class TP_Logging_Task : public ACE_Task<ACE_MT_SYNCH>
{
```

Become an ACE Task with MT synchronization trait

```
public:
  enum { MAX_THREADS = 4 };
```

Hook method called back by Task framework to initialize task

```
  virtual int open (void * = 0)
  {
    return activate (THR_NEW_LWP, MAX_THREADS);
  }
```

Hook method called by client to pass a message to task

```
  virtual int put (ACE_Message_Block *mblk,
                   ACE_Time_Value *timeout = 0)
  {
    return putq (mblk, timeout);
  }
```

Enqueue message for subsequent processing

```
  // … Other methods omitted …
};
```

Sidebar: Avoiding Memory Leaks When Threads Exit

- By default, **ACE_Thread_Manager** (& hence the **ACE_Task** class that uses it) spawns threads with the **THR_JOINABLE** flag
- To avoid leaking resources that the OS holds for joinable threads, an application must call one of the following methods:
  - **ACE_Task::wait()**, which waits for all threads to exit an **ACE_Task** object
  - **ACE_Thread_Manager::wait_task()**, which waits for all threads to exit in a specified **ACE_Task** object
  - **ACE_Thread_Manager::join()**, which waits for a designated thread to exit
- If none of these methods are called, ACE & the OS won't reclaim the thread stack & exit status of a joinable thread, & the program will leak memory
- If it's inconvenient to wait for threads explicitly in your program, you can simply pass **THR_DETACHED** when spawning threads or activating tasks
- Many networked application tasks & long-running daemon threads can be simplified by using detached threads
- However, an application can't wait for a detached thread to finish with **ACE_Task::wait()** or obtain its exit status via **ACE_Thread_Manager::join()**
- Applications can, however, use **ACE_Thread_Manager::wait()** to wait for both joinable & detached threads managed by an **ACE_Thread_Manager** to finish

Using the ACE_Task Class (4/13)

```
typedef ACE_Unmanaged_Singleton<TP_Logging_Task, ACE_Null_Mutex>
        TP_LOGGING_TASK;
```

Unmanaged singletons don't automatically delete themselves on program exit

```
class TP_Logging_Acceptor : public Logging_Acceptor {
public:
  TP_Logging_Acceptor (ACE_Reactor *r = ACE_Reactor::instance
())
    : Logging_Acceptor (r) {}
```

Hook method called by Reactor framework – performs passive portion of Acceptor/Connector pattern

```
  virtual int handle_input (ACE_HANDLE) {
    TP_Logging_Handler *peer_handler = 0;
    ACE_NEW_RETURN (peer_handler,
                    TP_Logging_Handler (reactor ()), -1);
    if (acceptor_.accept (peer_handler->peer ()) == -1) {
      delete peer_handler; return -1;
    } else if (peer_handler->open () == -1)
      peer_handler->handle_close (ACE_INVALID_HANDLE, 0);
    return 0;
  }
```

Sidebar: ACE_Singleton Template Adapter

```
template <class TYPE, class LOCK>
class ACE_Singleton : public ACE_Cleanup {
public:
  static TYPE *instance (void) {
    ACE_Singleton<TYPE, LOCK> *&s = singleton_;
    if (s == 0) {
      LOCK *lock = 0;
      ACE_GUARD_RETURN (LOCK, guard,
            ACE_Object_Manager::get_singleton_lock (lock), 0);
      if (s == 0) {
        ACE_NEW_RETURN (s, (ACE_Singleton<TYPE, LOCK>), 0);
        ACE_Object_Manager::at_exit (s);
      }
    }
    return &s->instance_;
  }
protected:
  ACE_Singleton (void); // Default constructor.
  TYPE instance_; // Contained instance.
  // Single instance of the <ACE_Singleton> adapter.
  static ACE_Singleton<TYPE, LOCK> *singleton_;
};
```

Note Double-Checked Locking Optimization pattern

ACE_Unmanaged_Singleton omits this step

Synchronizing Singletons Correctly

Problem

- Singletons can be problematic in multi-threaded programs

Either too little locking…

```
class Singleton {
public:
  static Singleton *instance ()
  {
    if (instance_ == 0) {
      // Enter critical
      // section.
      instance_ =
        new Singleton;
      // Leave critical
      // section.
    }
    return instance_;
  }
  void method_1 ();
  // Other methods omitted.
private:
  static Singleton *instance_;
  // Initialized to 0
  // by linker.
};
```

… or too much

```
class Singleton {
public:
  static Singleton *instance ()
  {
    Guard<Thread_Mutex>
      g (lock_);
    if (instance_ == 0) {
      // Enter critical
      // section.
      instance_ = new Singleton;
      // Leave critical
      // section.
    }
    return instance_;
  }
private:
  static Singleton *instance_;
  // Initialized to 0
  // by linker.
  static Thread_Mutex lock_;
};
```

Double-checked Locking Optimization Pattern

Solution

- Apply the *Double-Checked Locking Optimization* design pattern (POSA2) to reduce contention & synchronization overhead whenever critical sections of code must acquire locks in a thread-safe manner just once during program execution

```
// Perform first-check to
// evaluate 'hint'.
if (first_time_in is TRUE)
{
  acquire the mutex
  // Perform double-check to
  // avoid race condition.
  if (first_time_in is TRUE)
  {
    execute the critical section
    set first_time_in to FALSE
  }
  release the mutex
}
```

```
class Singleton {
public:
  static Singleton *instance ()
  {
  // First check
  if (instance_ == 0) {
    Guard<Thread_Mutex> g(lock_);
    // Double check.
    if (instance_ == 0)
      instance_ = new Singleton;
    }
    return instance_;
  }
private:
  static Singleton *instance_;
  static Thread_Mutex lock_;
};
```

Pros & Cons of Double-Checked Locking Optimization Pattern

This pattern has two **benefits**:

- ***Minimized locking overhead***
  - By performing two first-time-in flag checks, this pattern minimizes overhead for the common case
  - After the flag is set the first check ensures that subsequent accesses require no further locking
- ***Prevents race conditions***
  - The second check of the first-time-in flag ensures that the critical section is executed just once

This pattern has some **liabilities**:

- ***Non-atomic pointer or integral assignment semantics***
  - If an `instance_` pointer is used as the flag in a singleton implementation, all bits of the singleton `instance_` pointer must be read & written atomically in a single operation
  - If the write to memory after the call to new is not atomic, other threads may try to read an invalid pointer
- ***Multi-processor cache coherency***
  - Certain multi-processor platforms, such as the COMPAQ Alpha & Intel Itanium, perform aggressive memory caching optimizations in which read & write operations can execute 'out of order' across multiple CPU caches, such that the CPU cache lines will not be flushed properly if shared data is accessed without locks held

Using the ACE_Task Class (5/13)

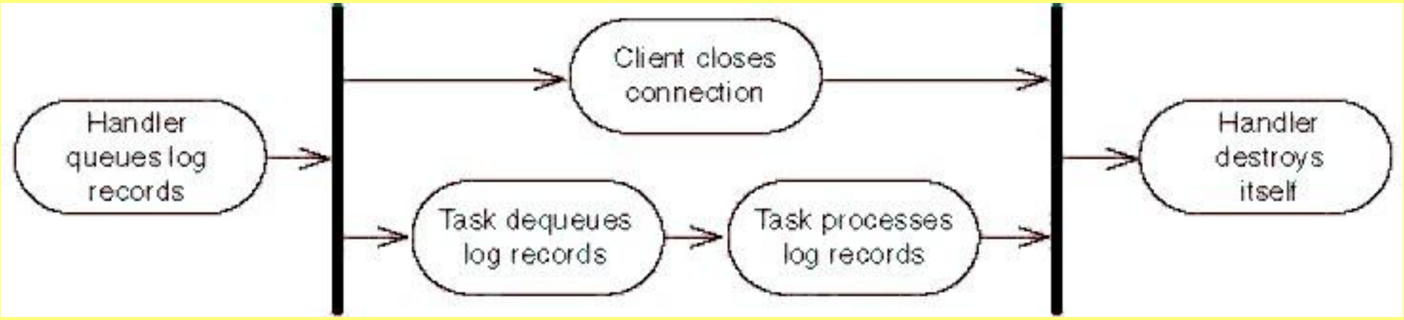
```
class TP_Logging_Handler : public Logging_Event_Handler {
  friend class TP_Logging_Acceptor;
protected:
  virtual ~TP_Logging_Handler () {} // No-op destructor.
```

Implements the protocol for shutting down handlers concurrently

```
  // Number of pointers to this class instance that currently
  // reside in the <TP_LOGGING_TASK> singleton's message
queue.
  int queued_count_;

  // Indicates whether <Logging_Event_Handler::handle_close()>
  // must be called to cleanup & delete this object.
  int deferred_close_;

  // Serialize access to <queued_count_> & <deferred_close_>.
  ACE_Thread_Mutex lock_;
```

Sidebar: Closing TP_Logging_Handlers Concurrently

- A challenge with thread pool servers is closing objects that can be accessed concurrently by multiple threads
  - e.g., we must therefore ensure that a **`TP_Logging_Handler`** object isn't destroyed while there are still pointers to it in use by **`TP_LOGGING_TASK`**
- When a logging client closes a connection, **`TP_Logging_Handler`**'s **`handle_input()`** returns -1 & the reactor then calls the handler's **`handle_close()`** method, which ordinarily cleans up resources & deletes the handler
  - Unfortunately, this would wreak havoc if one or more pointers to that handler were still enqueued or being used by threads in the **`TP_LOGGING_TASK`** pool
- We therefore use a reference counting protocol to ensure the handler isn't destroyed while a pointer to it is still in use
- The protocol counts how often a handler resides in the **`TP_LOGGING_TASK`** message queue
- If the count is greater than 0 when the logging client socket is closed then **`TP_Logging_Handler::handle_close()`** can't yet destroy the handler
- Later, as the **`TP_LOGGING_TASK`** processes each log record, the handler's reference count is decremented
- When the count reaches 0, the handler can finish processing the close request that was deferred earlier



Using the ACE_Task Class (6/13)

```
public:
  TP_Logging_Handler (ACE_Reactor *reactor)
    : Logging_Event_Handler (reactor),
      queued_count_ (0),
      deferred_close_ (0) {}
```

Hook methods dispatched by Reactor framework

```
  // Called when input events occur, e.g., connection or
data.
  virtual int handle_input (ACE_HANDLE);

  // Called when this object is destroyed, e.g., when it's
  // removed from a reactor.
  virtual int handle_close (ACE_HANDLE, ACE_Reactor_Mask);
};
```

Using the ACE_Task Class (7/13)

Hook method dispatched by Reactor when logging record arrives

```
int TP_Logging_Handler::handle_input (ACE_HANDLE) {
  ACE_Message_Block *mblk = 0;
  if (logging_handler_.recv_log_record (mblk) != -1) {
```

Note decoupling of recv vs. write!

```
    ACE_Message_Block *log_blk = 0;
    ACE_NEW_RETURN
      (log_blk, ACE_Message_Block
                    (ACE_reinterpret_cast (char *, this)),
-1);
    log_blk->cont (mblk);
```

Add ourselves to composite message

This lock protects the reference count

```
    ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, lock_, -1);
    if (TP_LOGGING_TASK::instance ()->put (log_blk) == -1)
    { log_blk->release (); return -1; }
    ++queued_count_;
    return 0;
  } else return -1;
```

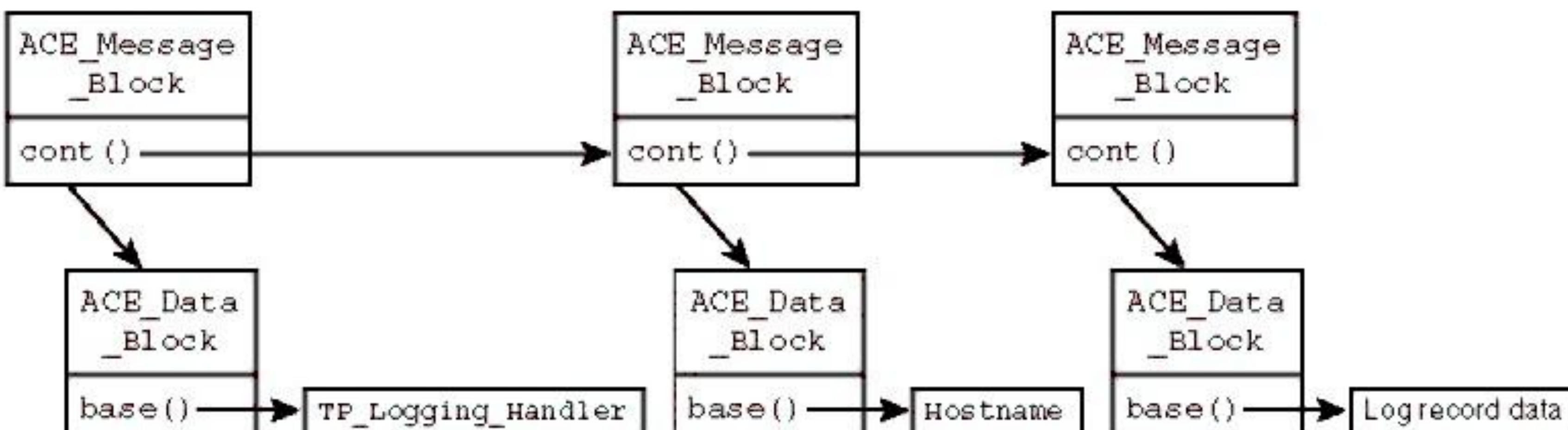
Store composite message into message queue (half-asynch)

Note fact that there's one more instance of ourselves in use!

Using the ACE_Task Class (8/13)

```
int TP_Logging_Handler::handle_input (ACE_HANDLE) {
  ACE_Message_Block *mblk = 0;
  if (logging_handler_.recv_log_record (mblk) != -1) {
    ACE_Message_Block *log_blk = 0;
    ACE_NEW_RETURN
      (log_blk, ACE_Message_Block
                    (ACE_reinterpret_cast (char *, this)),
-1);
    log_blk->cont (mblk);
    ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, lock_, -1);
    if (TP_LOGGING_TASK::instance ()->put (log_blk) == -1)
    { log_blk->release (); return -1; }
    ++queued_count_;
    return 0;
  } else return -1;
}
```

This is the composite message created by this method & placed onto the message queue



Using the ACE_Task Class (9/13)

This hook method is dispatched by the reactor & does the bulk of the work for the deferred shutdown processing

```
int TP_Logging_Handler::handle_close (ACE_HANDLE handle,
                                      ACE_Reactor_Mask) {
  int close_now = 0;
  if (handle != ACE_INVALID_HANDLE) {
    ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, lock_, -1);
    if (queued_count_ == 0) close_now = 1;
    else deferred_close_ = 1;
  } else {
    ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, lock_, -1);
    queued_count_--;
    if (queued_count_ == 0) close_now = deferred_close_;
  }

  if (close_now) return Logging_Event_Handler::handle_close
();
  return 0;
}
```

Called implicitly

Called explicitly

We can only close when there are no more instances of TP_Logging_Handler in use!

Using the ACE_Task Class (10/13)

This hook method runs in its own thread(s) of control & is called back by the ACE Task framework

```
int TP_Logging_Task::svc () {
```

This loop blocks until new composite message is queued (half-sync)

```
  for (ACE_Message_Block *log_blk; getq (log_blk) != -1; ) {
```

Remove TP_Logging_Handler pointer from composite message

```
    TP_Logging_Handler *tp_handler = ACE_reinterpret_cast
      (TP_Logging_Handler *, log_blk->rd_ptr ());
```

Write log record to log file

```
    Logging_Handler logging_handler (tp_handler->log_file
());
    logging_handler.write_log_record (log_blk->cont ());
    log_blk->release ();
    tp_handler->handle_close (ACE_INVALID_HANDLE, 0);
  }
  return 0;
}
```

Indicate that we're no longer using the handler

Using the ACE_Task Class (11/13)

This is the primary "façade" class that brings all the other parts together

```
class TP_Logging_Server
  : public ACE_Service_Object {
```

We can dynamically configure this via the ACE Service Configurator framework

```
protected:
  // Contains the reactor, acceptor, & handlers.
  typedef
Reactor_Logging_Server<TP_Logging_Acceptor>
          LOGGING_DISPATCHER;
```

We can reuse the Reactor_Logging_Server from previous versions of our server logging daemon

```
  LOGGING_DISPATCHER *logging_dispatcher_;
public:
  TP_Logging_Server (): logging_dispatcher_ (0) {}
  // Other methods defined below...
};
```

Sidebar: Destroying an ACE_Task

- Before destroying an **`ACE_Task`** that's running as an active object, ensure that the thread(s) running its **`svc()`** hook method have exited
- If a task's life cycle is managed externally, one way to ensure a proper destruction sequence looks like this:

```
My_Task *task = new Task; // Allocate a new task dynamically.
task->open (); // Initialize the task.
task->activate (); // Run task as an active object.
// ... do work ...
// Deactive the message queue so the svc() method unblocks
// & the thread exits.
task->msg_queue ()->deactivate ();
task->wait (); // Wait for the thread to exit.
delete task; // Reclaim the task memory.
```

- If a task is allocated dynamically, however, it may be better to have the task's **`close()`** hook delete itself when the last thread exits the task, rather than calling delete on a pointer to the task directly
  - You may still want to **`wait()`** on the threads to exit the task, however, particularly if you're preparing to shut down the process
  - On some OS platforms, when the main thread returns from **`main()`**, the entire process will be shut down immediately, whether there were other threads active or not

Using the ACE_Task Class (12/13)

This hook method is dispatched by ACE Service Configurator framework

```
virtual int init (int argc, ACE_TCHAR *argv[]) {
  int i;
  char **array = 0;
  ACE_NEW_RETURN (array, char*[argc], -1);
  ACE_Auto_Array_Ptr<char *> char_argv (array);
  for (i = 0; i < argc; ++i)
    char_argv[i] = ACE::strnew (ACE_TEXT_ALWAYS_CHAR
(argv[i]));
  ACE_NEW_NORETURN (logging_dispatcher_,
      TP_Logging_Server::LOGGING_DISPATCHER
        (i, char_argv.get (), ACE_Reactor::instance ()));
  for (i = 0; i < argc; ++i) ACE::strdelete (char_argv[i]);
  if (logging_dispatcher_ == 0) return -1;
  else return TP_LOGGING_TASK::instance ()->open ();
}
```

Using the ACE_Task Class (13/13)

This hook method is called by ACE Service Configurator framework to shutdown the service

```
virtual int fini () {
  TP_LOGGING_TASK::instance ()->flush ();
  TP_LOGGING_TASK::instance ()->wait ();
  TP_LOGGING_TASK::close ();
  delete logging_dispatcher_;
  return 0;
}

ACE_FACTORY_DEFINE (TPLS, TP_Logging_Server)
```

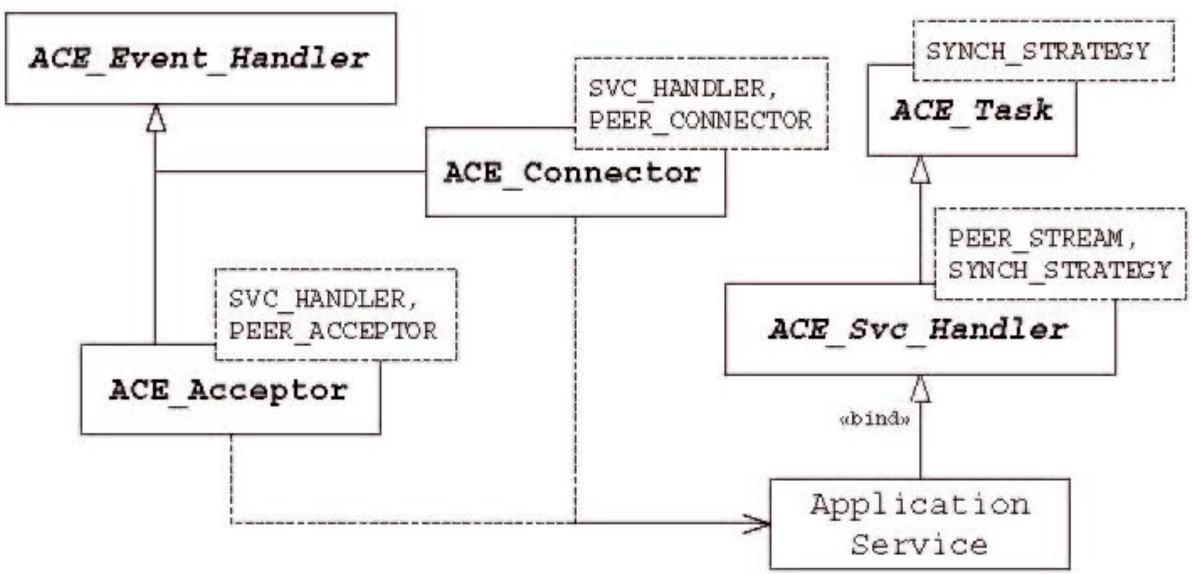
`svc.conf` file for thread pool server logging daemon

```
dynamic TP_Logging_Server Service_Object *
TPLS:_make_TP_Logging_Server()
"$TP_LOGGING_SERVER_PORT"
```

The main() function is the same as the one we showed for the ACE Service Configurator example!!!!

The ACE Acceptor/Connector Framework

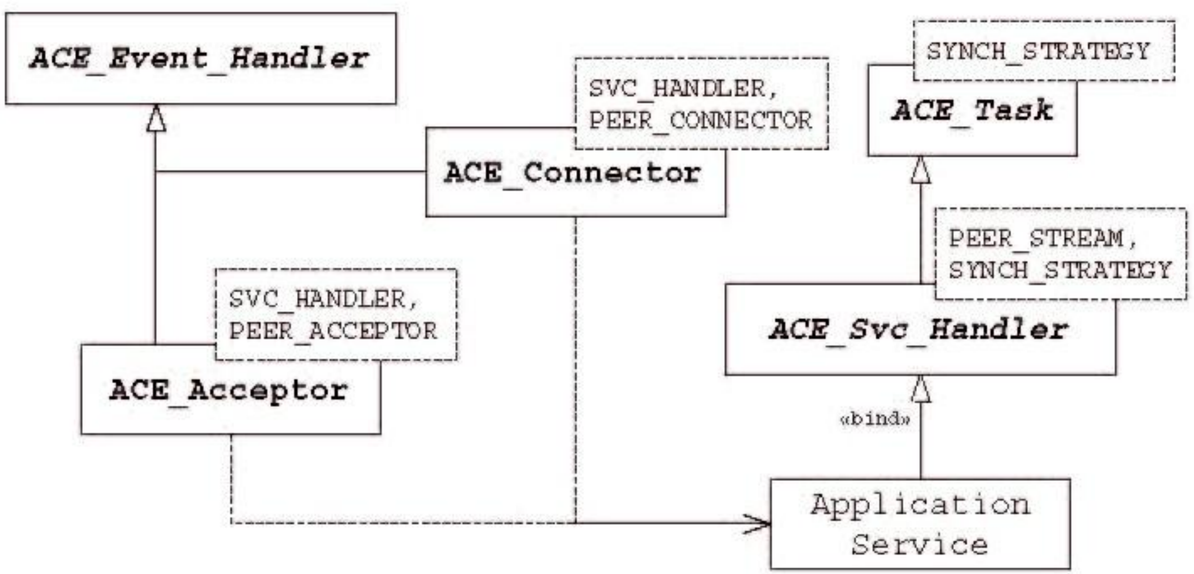
- The ACE Acceptor/Connector framework implements the Acceptor/Connector pattern (POSA2)
- This pattern enhances software reuse & extensibility by decoupling the activities required to connect & initialize cooperating peer services in a networked application from the processing they perform once they're connected & initialized



The ACE Acceptor/Connector Framework

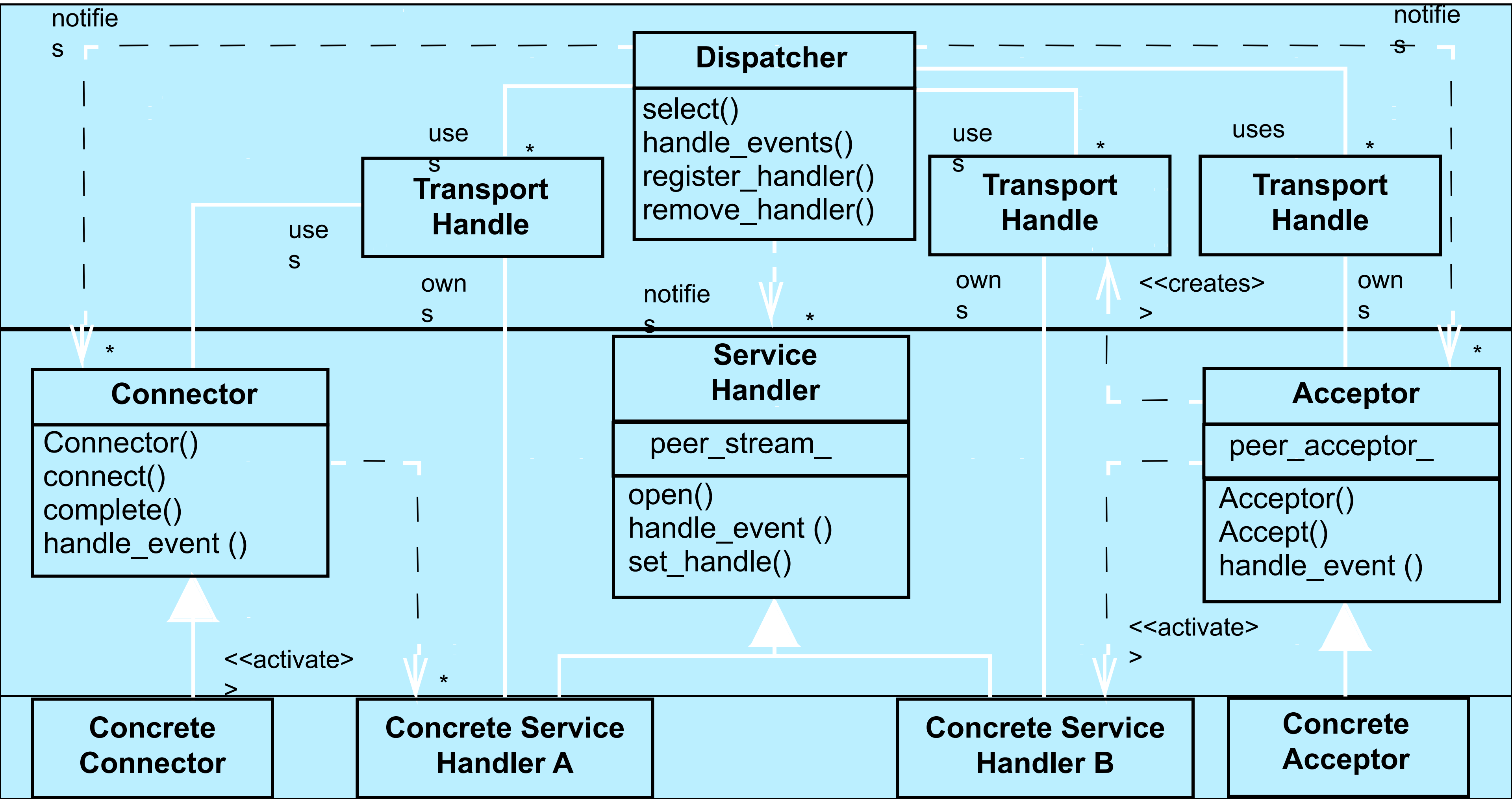
ACE Class	Description
ACE_Svc_Handler	Represents the local end of a connected service and contains an IPC endpoint used to communicate with a connected peer.
ACE_Acceptor	This factory waits passively to accept a connection and then initializes an ACE_Svc_Handler in response to an active connection request from a peer.
ACE_Connector	This factory actively connects to a peer acceptor and then initializes an ACE_Svc_Handler to communicate with its connected peer.

- The relationships between the ACE Acceptor/Connector framework classes that networked applications can use to establish connections & initialize peer services are shown in the adjacent figure

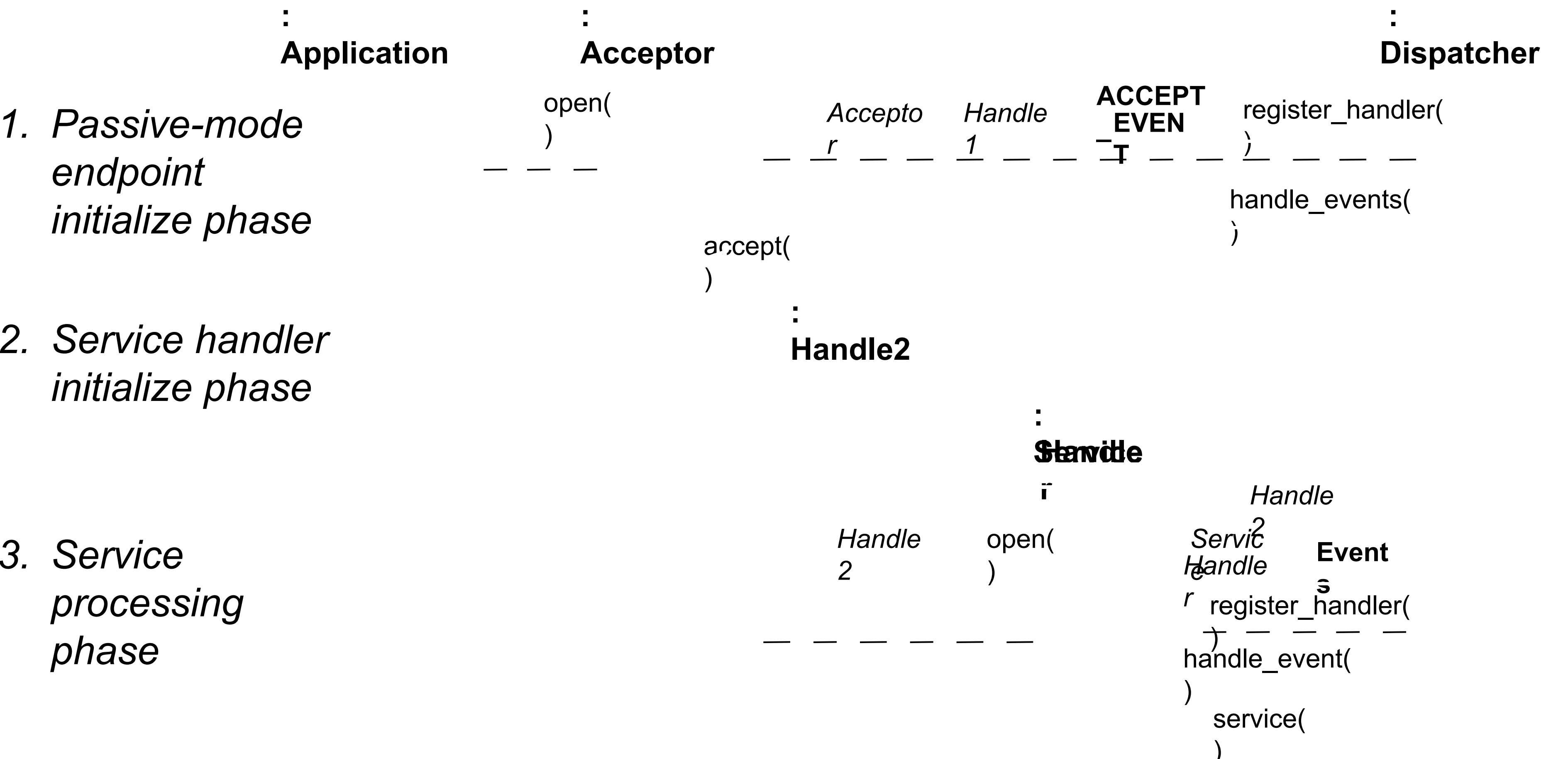


The Acceptor/Connector Pattern

•The *Acceptor/Connector* design pattern (POSA2) decouples the connection & initialization of cooperating peer services in a networked system from the processing performed by the peer services after being connected & initialized



Acceptor Dynamics

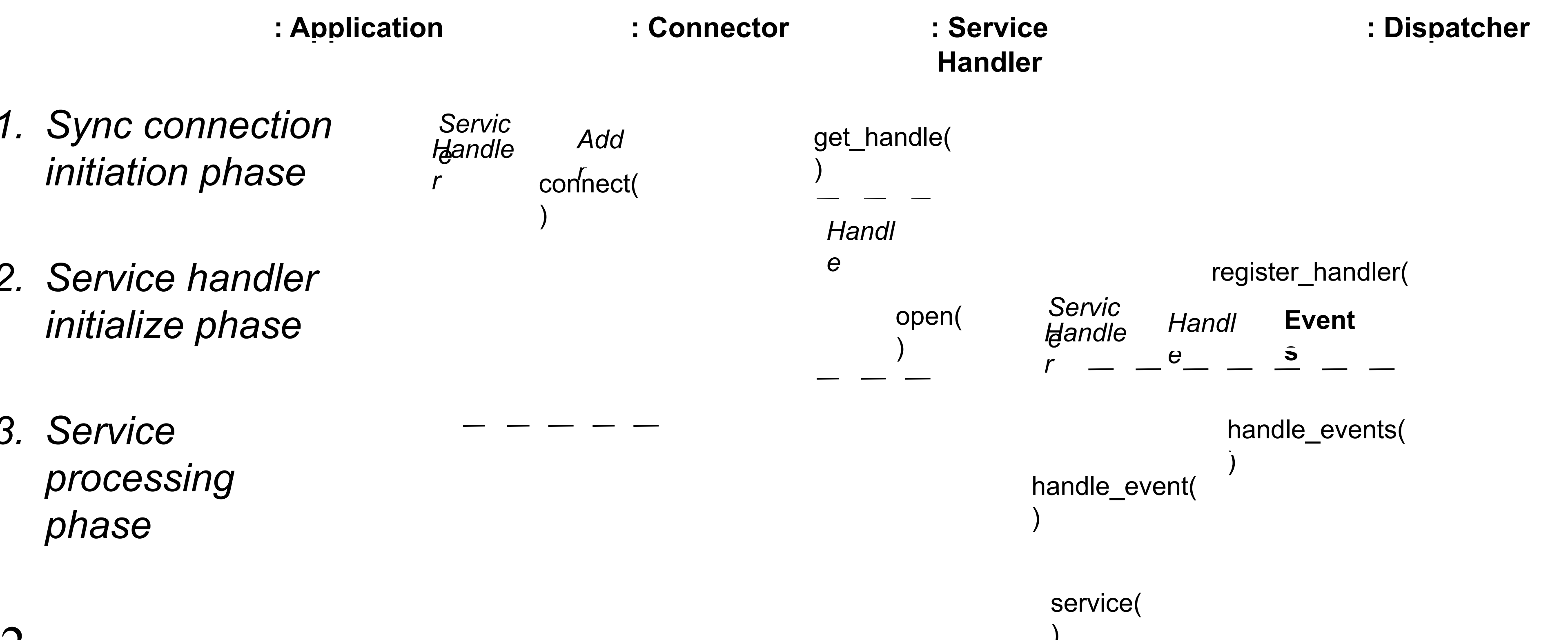


- The **`Acceptor`** ensures that passive-mode transport endpoints aren't used to read/write data accidentally
  - And vice versa for data transport endpoints…
- There is typically one **`Acceptor`** factory per-service/per-port
  - Additional demuxing can be done at higher layers, *a la* CORBA

Synchronous Connector Dynamics

Motivation for Synchrony

- If connection latency is negligible
  - *e.g.*, connecting with a server on the same host via a 'loopback' device
- If multiple threads of control are available & it is efficient to use a thread-per-connection to connect each service handler synchronously
- If the services must be initialized in a fixed order & the client can't perform useful work until all connections are established

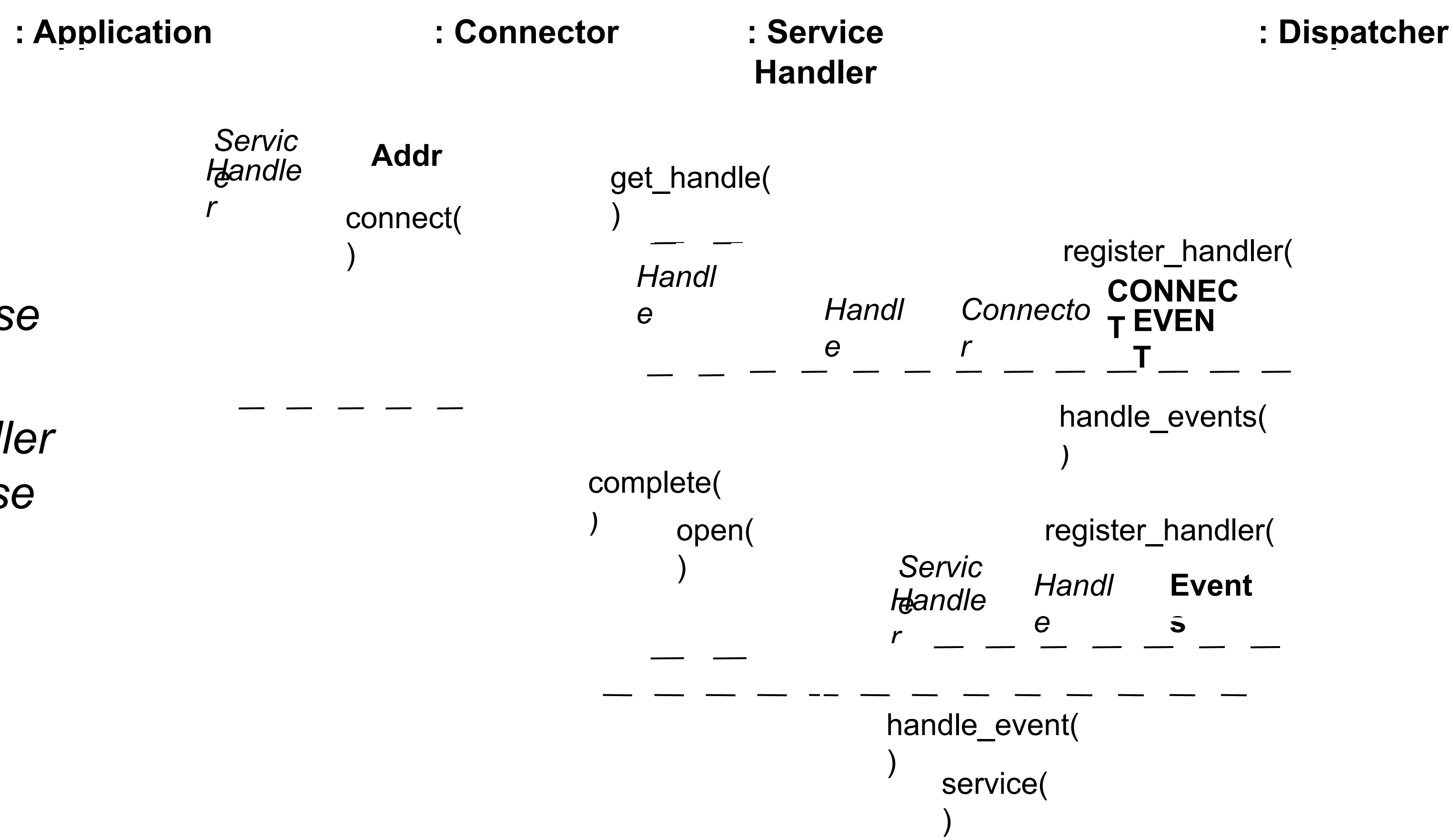


Asynchronous Connector Dynamics

Motivation for Asynchrony

- If client is establishing connections over high latency links
- If client is a single-threaded application
- If client is initializing many peers that can be connected in an arbitrary order

1. *Async connection initiation phase*
2. *Service handler initialize phase*
3. *Service processing phase*



The ACE_Svc_Handler Class (1/2)

Motivation

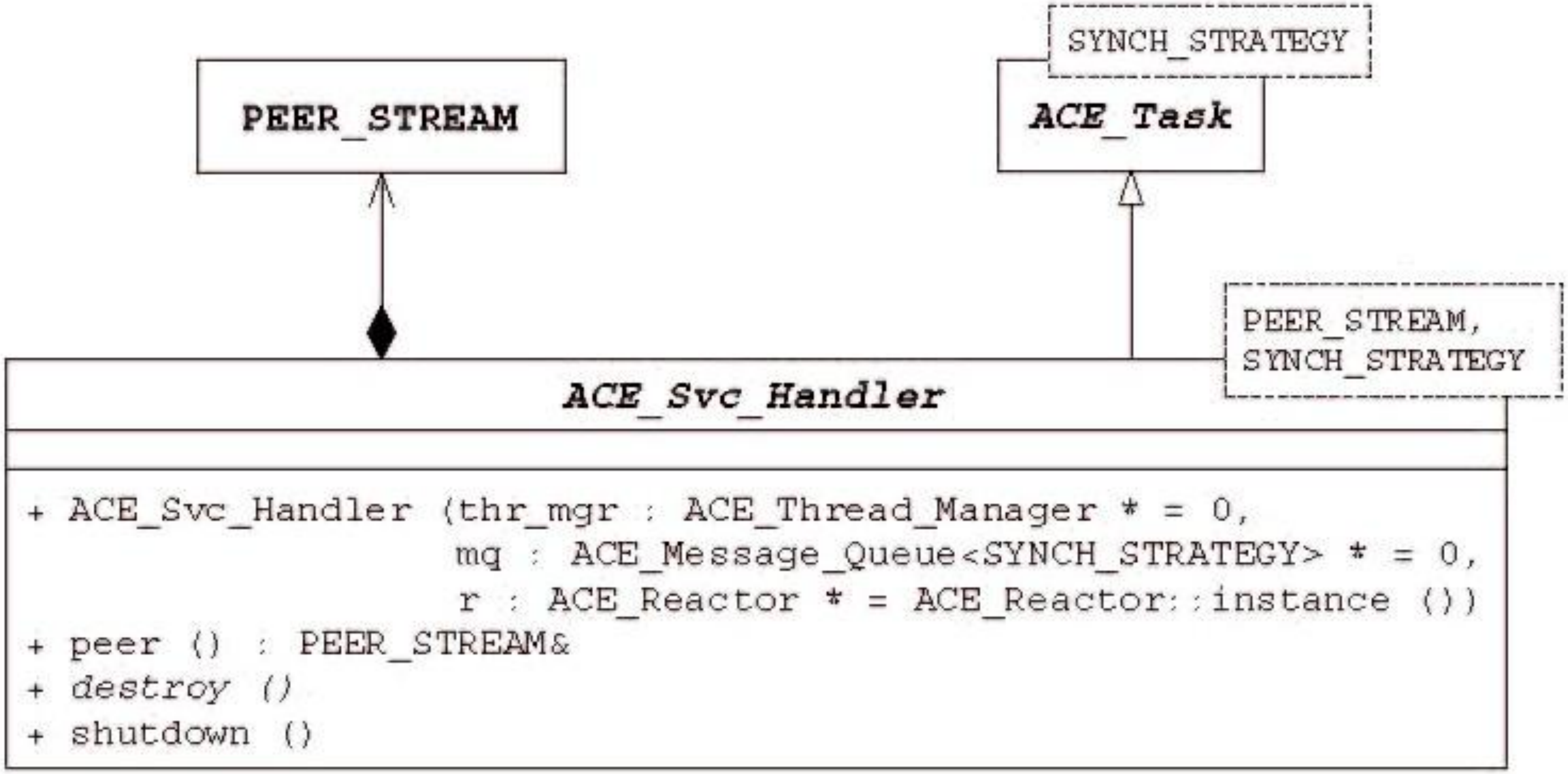
- A service handler is the portion of a networked application that either implements or accesses (or both, in the case of a peer-to-peer arrangement) a service
- Connection-oriented networked applications require at least two communicating service handlers – one for each end of every connection
- To separate concerns & allow developers to focus on the functionality of their service handlers, the ACE Acceptor/Connector framework defines the **`ACE_Svc_Handler`** class

The ACE_Svc_Handler Class (2/2)

Class Capabilities

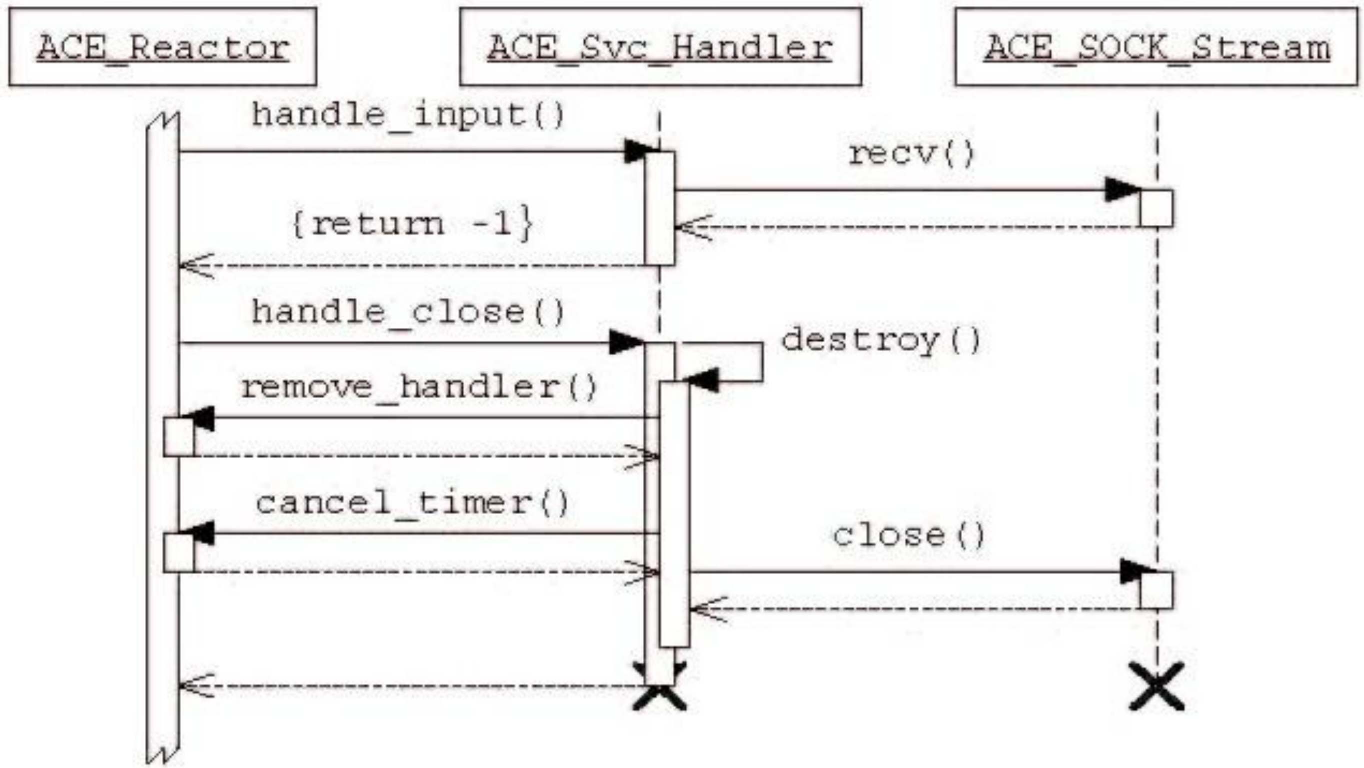
- This class is the basis of ACE's synchronous & reactive data transfer & service processing mechanisms & it provides the following capabilities:
  - It provides the basis for initializing & implementing a service in a synchronous and/or reactive networked application, acting as the target of the **`ACE_Connector`** & **`ACE_Acceptor`** connection factories
  - It provides an IPC endpoint used by a service handler to communicate with its peer service handler
  - Since **`ACE_Svc_Handler`** derives directly from **`ACE_Task`** (& indirectly from **`ACE_Event_Handler`**), it inherits the ACE concurrency, queueing, synchronization, dynamic configuration, & event handling framework capabilities
  - It codifies the most common practices of reactive network services, such as registering with a reactor when a service is opened & closing the IPC endpoint when unregistering a service from a reactor

The ACE_Svc_Handler Class API



This class handles *variability* of IPC mechanism & synchronization strategy via a *common* network I/O API

Combining ACE_Svc_Handler w/Reactor



- An instance of **ACE_Svc_Handler** can be registered with the ACE Reactor framework for READ events
- The Reactor framework will then dispatch the **ACE_Svc_Handler::handle_input()** when input arrives on a connection

Sidebar: Decoupling Service Handler Creation from Activation

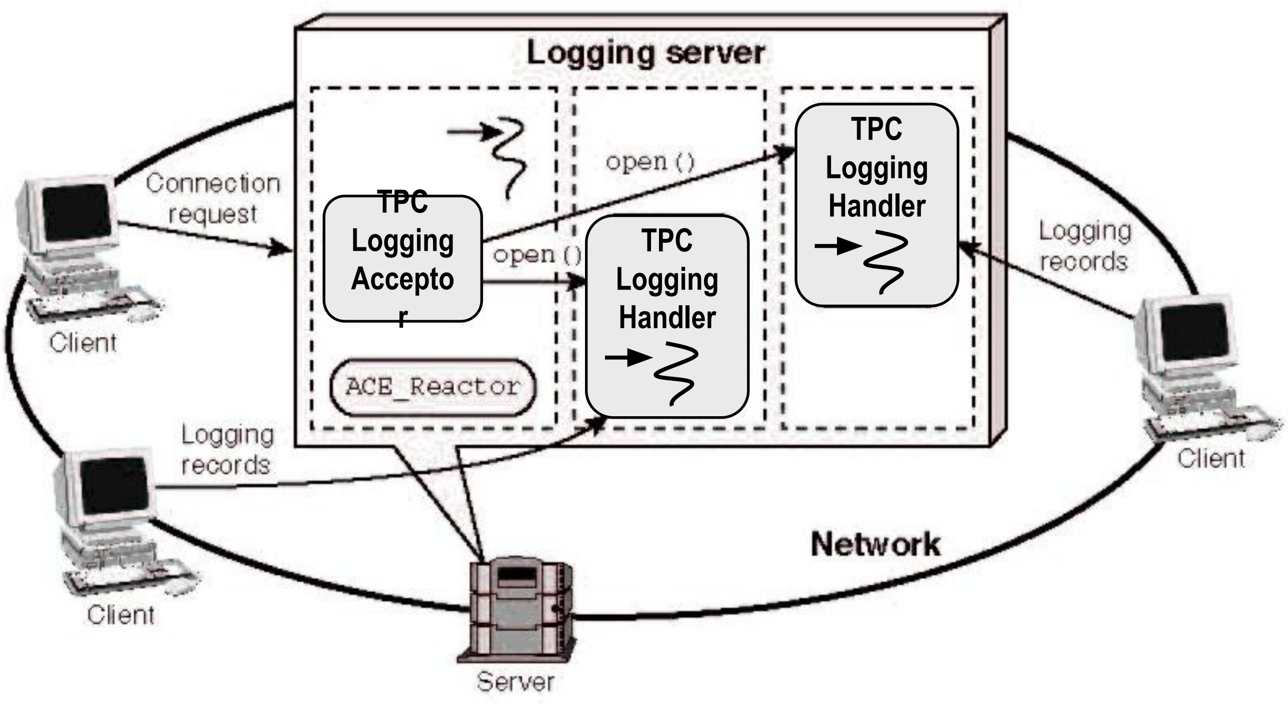
- The motivations for decoupling service activation from service creation in the ACE Acceptor/Connector framework include:
  - ***To make service handler creation flexible***
    - ACE allows for wide flexibility in the way an application creates (or reuses) service handlers.
    - Many applications create new handlers dynamically as needed, but some may recycle handlers or use a single handler for all connections
  - ***To simplify error handling***
    - ACE doesn't rely on native C++ exceptions
    - The constructor used to create a service handler therefore shouldn't perform any operations that can fail
    - Instead, any such operations should be placed in the **`open()`** hook method, which must return −1 if activation fails
  - ***To ensure thread safety***
    - If a thread is spawned in a constructor it's not possible to ensure that the object has been initialized completely before the thread begins to run
    - To avoid this potential race condition, the ACE Acceptor/Connector framework decouples service handler creation from activation

Sidebar: Determining a Service Handler's Storage Class

- **`ACE_Svc_Handler`** objects are often allocated dynamically by the **`ACE_Acceptor`** & **`ACE_Connector`** factories in the ACE Acceptor/Connector framework
- There are situations, however, when service handlers are allocated differently, such as statically or on the stack
- To reclaim a handler's memory correctly, without tightly coupling it with the classes & factories that may instantiate it, the **`ACE_Svc_Handler`** class uses the C++ Storage Class Tracker idiom
- This idiom performs the following steps to determine automatically whether a service handler was allocated statically or dynamically & act accordingly:
  - **`ACE_Svc_Handler`** overloads operator new, which allocates memory dynamically & sets a flag in thread-specific storage that notes this fact
  - The **`ACE_Svc_Handler`** constructor inspects thread-specific storage to see if the object was allocated dynamically, recording the result in a data member
  - When the **`destroy()`** method is eventually called, it checks the "dynamically allocated" flag
    - If the object was allocated dynamically, **`destroy()`** deletes it
    - If not, it will simply let the **`ACE_Svc_Handler`** destructor clean up the object when it goes out of scope

Using the ACE_Svc_Handler Class (1/4)

- This example illustrates how to use the `ACE_Svc_Handler` class to implement a logging server based on the *thread-per-connection* concurrency model
- Note how little "glue" code needs to be written manually since the various ACE frameworks to most of the dirty work…



Using the ACE_Svc_Handler Class (2/4)

```
class TPC_Logging_Handler
  : public ACE_Svc_Handler<ACE_SOCK_Stream, ACE_NULL_SYNCH>
{

protected:
  ACE_FILE_IO log_file_; // File of log records.

  // Connection to peer service handler.
  Logging_Handler logging_handler_;

public:
  TPC_Logging_Handler (): logging_handler_ (log_file_) {}

// ... Other methods shown below ...
```

Become a service handler

Using the ACE_Svc_Handler Class (3/4)

Activation hook method called back by Acceptor for each connection

```
virtual int open (void *) {
  static const ACE_TCHAR LOGFILE_SUFFIX[] = ACE_TEXT (".log");
  ACE_TCHAR filename[MAXHOSTNAMELEN + sizeof (LOGFILE_SUFFIX)];
  ACE_INET_Addr logging_peer_addr;

  peer ().get_remote_addr (logging_peer_addr);
  logging_peer_addr.get_host_name (filename, MAXHOSTNAMELEN);
  ACE_OS_String::strcat (filename, LOGFILE_SUFFIX);

  ACE_FILE_Connector connector;
  connector.connect (log_file_,
                     ACE_FILE_Addr (filename),
                     0, // No timeout.
                     ACE_Addr::sap_any, // Ignored.
                     0, // Don't try to reuse the addr.
                     O_RDWR|O_CREAT|O_APPEND,
                     ACE_DEFAULT_FILE_PERMS);

  logging_handler_.peer ().set_handle (peer ().get_handle ());
  return activate (THR_NEW_LWP | THR_DETACHED);
}
```

Become an active object & calls the svc() hook method

Using the ACE_Svc_Handler Class (4/4)

Runs in our own thread of control

```
 virtual int svc () {
    for (;;)
```

Note how we're back to a single log method

```
      switch (logging_handler_.log_record ()) {
      case -1: return -1; // Error.
      case 0: return 0; // Client closed
connection.
      default: continue; // Default case.
      }
    /* NOTREACHED */
    return 0;
  }
};
```

Sidebar: Working Around Lack of Traits Support

- If you examine the ACE Acceptor/Connector framework source code closely, you'll notice that the IPC class template argument to **`ACE_Acceptor`**, **`ACE_Connector`**, & **`ACE_Svc_Handler`** is a macro rather than a type parameter
- Likewise, the synchronization strategy parameter to the ACE_Svc_Handler is a macro rather than a type parameter
- ACE uses these macros to work around the lack of support for traits classes & templates in some C++ compilers
- To work portably on those platforms, ACE class types, such as **`ACE_INET_Addr`** or **`ACE_Thread_Mutex`**, must be passed as explicit template parameters, rather than accessed as traits of traits classes, such as **`ACE_SOCK_Addr::PEER_ADDR`** or **`ACE_MT_SYNCH::MUTEX`**
- To simplify the efforts of application developers, ACE defines a set of macros that conditionally expand to the appropriate types, some of which are shown in the following table:

ACE Class	Description
`ACE_SOCK_ACCEPTOR`	Expands to either `ACE_SOCK_Acceptor` or `ACE_SOCK_Acceptor` and `ACE_INET_Addr`
`ACE_SOCK_CONNECTOR`	Expands to either `ACE_SOCK_Connector` or to `ACE_SOCK_Connector` and `ACE_INET_Addr`
`ACE_SOCK_STREAM`	Expands to either `ACE_SOCK_Stream` or to `ACE_SOCK_Stream` and `ACE_INET_Addr`

Sidebar: Shutting Down Blocked Service Threads

- Service threads often perform blocking I/O operations (this is often a bad idea)
- If the service thread must be stopped before its normal completion, however, the simplicity of this model can cause problems
- Some techniques to force service threads to shut down include:
  - ***Exit the server process***, letting the OS abruptly terminate the peer connection, as well as any other open resources, such as files (a log file, in the case of this chapter's examples)
    - This approach can result in lost data & leaked resources e.g., System V IPC objects are vulnerable in this approach
  - ***Enable asynchronous thread cancellation & cancel the service thread***
    - This design isn't portable & can also abandon resources if not programmed correctly
  - ***Close the socket***, hoping that the blocked I/O call will abort & end the service thread
    - This solution can be effective, but doesn't work on all platforms
  - ***Rather than blocking I/O, use timed I/O & check a shutdown flag***, or use the **`ACE_Thread_Manager`** cooperative cancellation mechanism, to cleanly shut down between I/O attempts
    - This approach is also effective, but may delay the shutdown by up to the specified timeout

The ACE_Acceptor Class (1/2)

Motivation

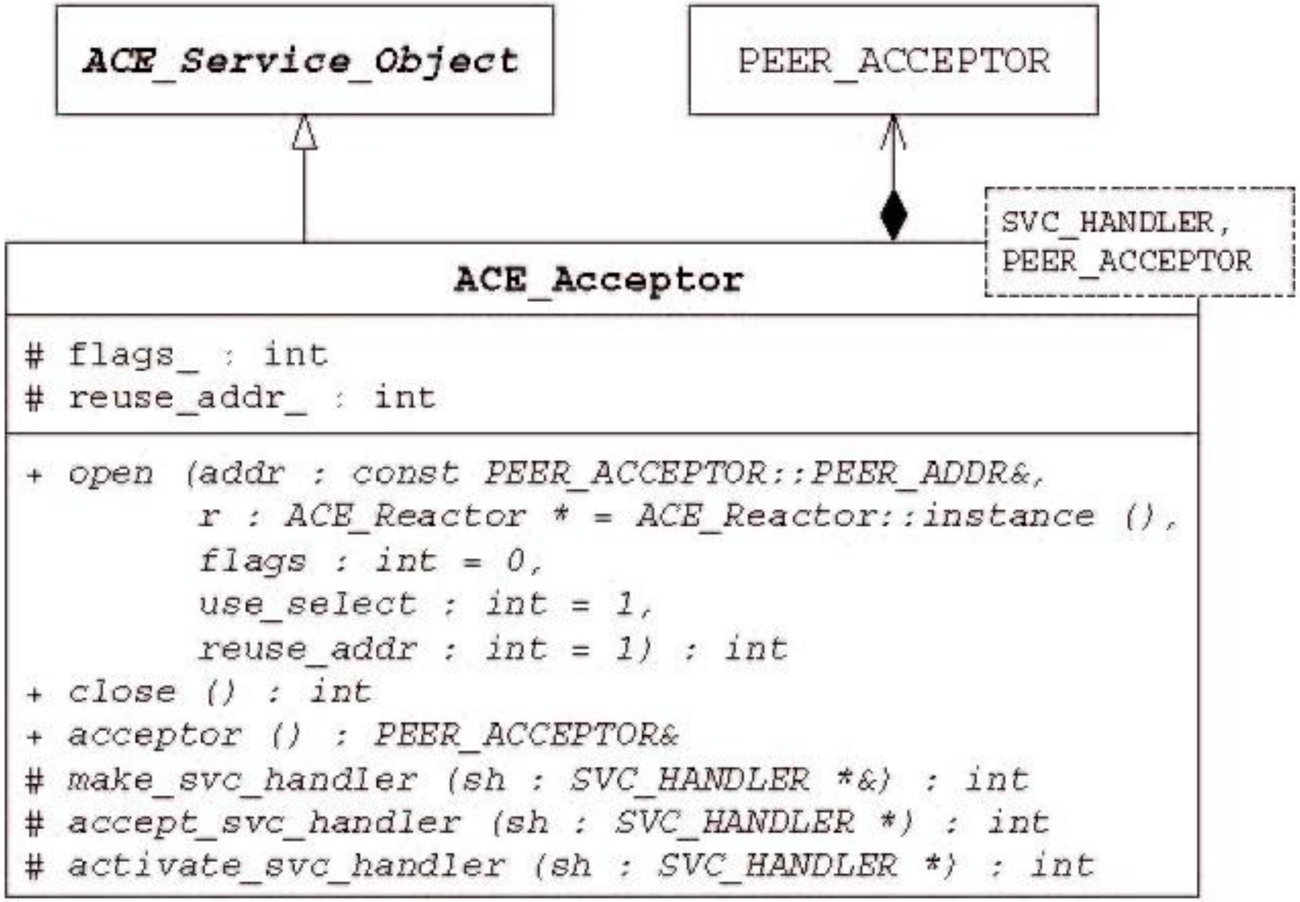
- Many connection-oriented server applications tightly couple their connection establishment & service initialization code in ways that make it hard to reuse existing code
- The ACE Acceptor/Connector framework defines the **`ACE_Acceptor`** class so that application developers needn't rewrite this code repeatedly

The ACE_Acceptor Class (2/2)

Class Capabilities

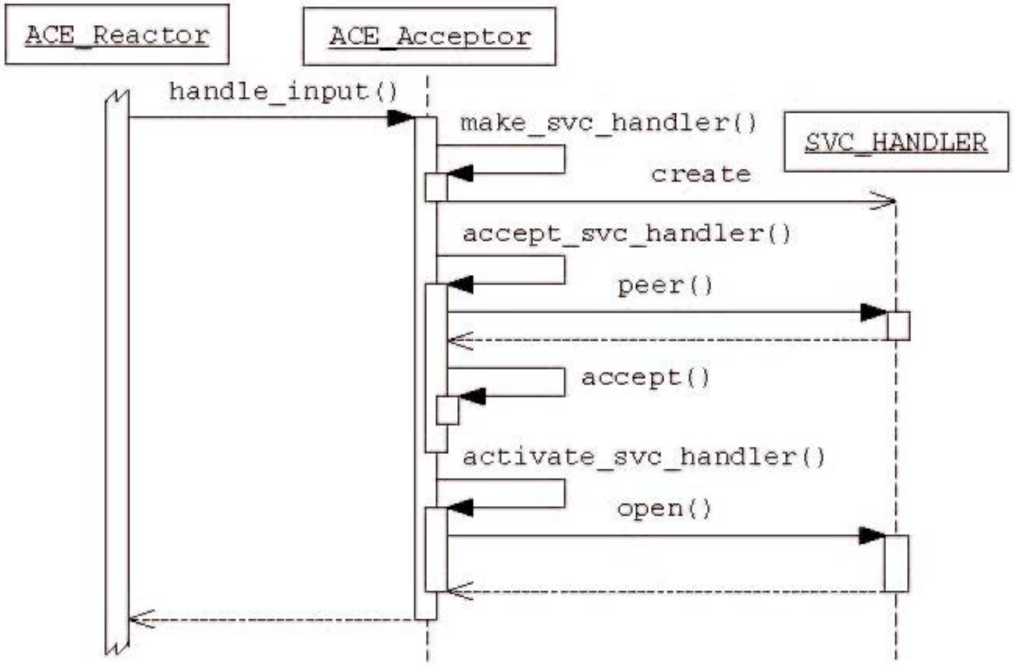
- This class is a factory that implements the Acceptor role in the Acceptor/Connector pattern to provide the following capabilities:
  - It decouples the passive connection establishment & service initialization logic from the processing performed by a service handler after it's connected & initialized
  - It provides a passive-mode IPC endpoint used to listen for & accept connections from peers
  - The type of this IPC endpoint can be parameterized with many of ACE's IPC wrapper façade classes, thereby separating lower-level connection mechanisms from application-level service initialization policies
  - It automates the steps necessary to connect the IPC endpoint passively & create/activate its associated service handlers
  - Since `ACE_Acceptor` derives from `ACE_Service_Object`, it inherits the event-handling & configuration capabilities from the ACE Reactor & Service Configurator frameworks

The ACE_Acceptor Class API



This class handles *variability* of IPC mechanism & service handler via a *common* connection establishment & service handler initialization API

Combining ACE_Acceptor w/Reactor



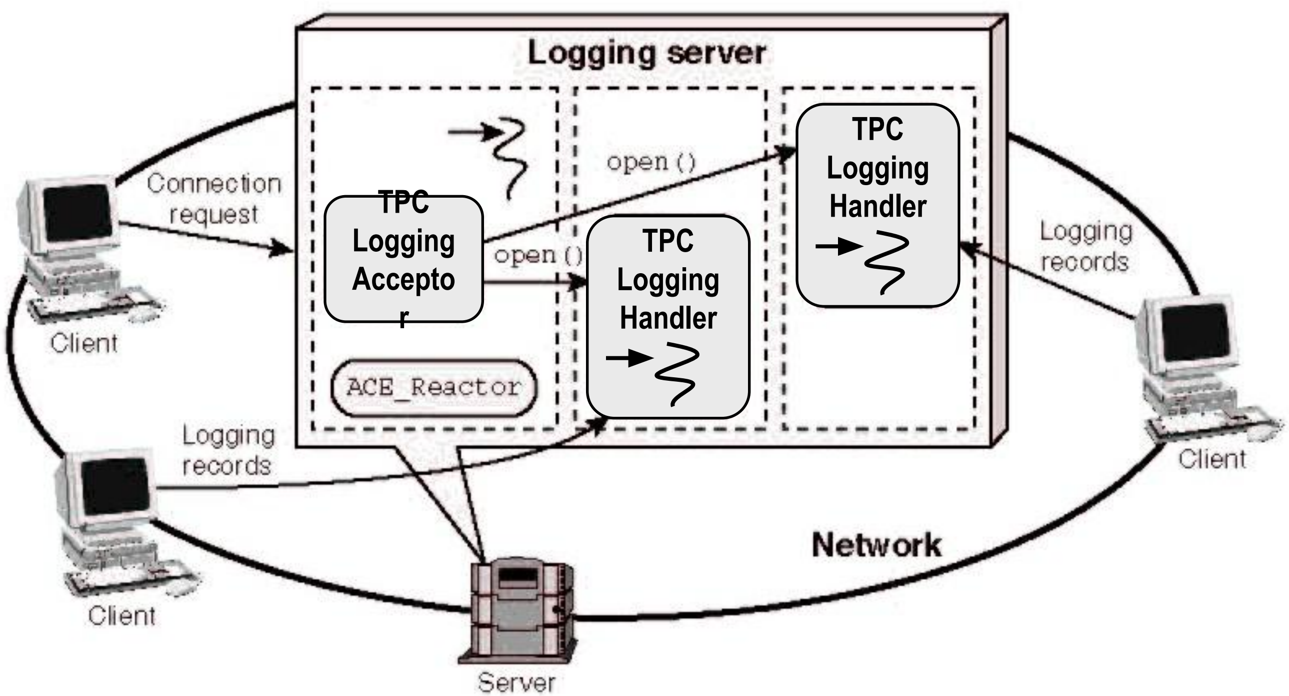
- An instance of **`ACE_Acceptor`** can be registered with the ACE Reactor framework for ACCEPT events
- The Reactor framework will then dispatch the **`ACE_Acceptor::handle_input()`** when input arrives on a connection

Sidebar: Encryption & Authorization Protocols

- To protect against potential attacks or third-party discovery, many networked applications must authenticate the identities of their peers & encrypt sensitive data sent over a network
- To provide these capabilities, various cryptography packages, such as OpenSSL, & security protocols, such as Transport Layer Security (TLS), have been developed
- These packages & protocols provide library calls that ensure authentication, data integrity, & confidentiality between two communicating applications
  - For example, the TLS protocol can encrypt/decrypt data sent/received across a TCP/IP network
  - TLS is based on an earlier protocol named the Secure Sockets Layer (SSL), which was developed by Netscape
- The OpenSSL toolkit used by the examples in this chapter is based on the SSLeay library

Using the ACE_Acceptor (1/7)

- This example is another variant of our server logging daemon
- It uses the `ACE_Acceptor` instantiated with an `ACE_SOCK_Acceptor` to listen on a passive-mode TCP socket handle defined by the "ace_logger" service entry
- This revision of the server uses the thread-per-connection concurrency model to handle multiple clients simultaneously
- It also uses SSL authentication via interceptors



Using the ACE_Acceptor (2/7)

```
#include "ace/SOCK_Acceptor.h"
#include <openssl/ssl.h>

class TPC_Logging_Acceptor
  : public ACE_Acceptor <TPC_Logging_Handler, ACE_SOCK_Acceptor>
{
```

Become an acceptor

```
protected:
  // The SSL ``context'' data structure.
  SSL_CTX *ssl_ctx_;

  // The SSL data structure corresponding to authenticated
  // SSL connections.
  SSL *ssl_;

public:
  typedef ACE_Acceptor<TPC_Logging_Handler, ACE_SOCK_Acceptor>
          PARENT;
  typedef ACE_SOCK_Acceptor::PEER_ADDR PEER_ADDR;
  TPC_Logging_Acceptor (ACE_Reactor *)
    : PARENT (r), ssl_ctx_ (0), ssl_ (0) {}
```

Using the ACE_Acceptor (3/7)

```
  // Destructor frees the SSL resources.
  virtual ~TPC_Logging_Acceptor (void) {
    SSL_free (ssl_);
    SSL_CTX_free (ssl_ctx_);
  }

  // Initialize the acceptor instance.
  virtual int open
    (const ACE_SOCK_Acceptor::PEER_ADDR &local_addr,
     ACE_Reactor *reactor = ACE_Reactor::instance (),
     int flags = 0, int use_select = 1, int reuse_addr =
1);

  // <ACE_Reactor> close hook method.
  virtual int handle_close
    (ACE_HANDLE = ACE_INVALID_HANDLE,
     ACE_Reactor_Mask =
ACE_Event_Handler::ALL_EVENTS_MASK);

  virtual int accept_svc_handler (TPC_Logging_Handler *sh);
};
```

Hook method for connection establishment & authentication

Using the ACE_Acceptor (4/7)

```
#include "ace/OS.h"
#include "Reactor_Logging_Server_Adapter.h"
#include "TPC_Logging_Server.h"
#include "TPCLS_export.h"

#if !defined (TPC_CERTIFICATE_FILENAME)
#  define TPC_CERTIFICATE_FILENAME "tpc-cert.pem"
#endif /* !TPC_CERTIFICATE_FILENAME */
#if !defined (TPC_KEY_FILENAME)
#  define TPC_KEY_FILENAME "tpc-key.pem"
#endif /* !TPC_KEY_FILENAME */

int TPC_Logging_Acceptor::open
    (const ACE_SOCK_Acceptor::PEER_ADDR &local_addr,
           ACE_Reactor *reactor,
           int flags, int use_select, int reuse_addr)
{
  if (PARENT::open (local_addr, reactor, flags,
                    use_select, reuse_addr) != 0)
    return -1;
```

Delegate to parent (ACE_Acceptor::open())

Using the ACE_Acceptor (5/7)

```
  OpenSSL_add_ssl_algorithms ();
  ssl_ctx_ = SSL_CTX_new (SSLv3_server_method ());
  if (ssl_ctx_ == 0) return -1;

  if (SSL_CTX_use_certificate_file (ssl_ctx_,
TPC_CERTIFICATE_FILENAME,
                                    SSL_FILETYPE_PEM) <= 0
      || SSL_CTX_use_PrivateKey_file (ssl_ctx_,
                                      TPC_KEY_FILENAME,
                                      SSL_FILETYPE_PEM) <= 0
      || !SSL_CTX_check_private_key (ssl_ctx_))
    return -1;
  ssl_ = SSL_new (ssl_ctx_);
  return ssl_ == 0 ? -1 : 0;
}
```

Do initialization for server-side of SSL authentication

Sidebar: ACE_SSL* Wrapper Facades

- Although the OpenSSL API provides a useful set of functions, it suffers from the usual problems incurred by native OS APIs written in C
- To address these problems, ACE provides classes that encapsulate OpenSSL using an API similar to the ACE C++ Socket wrapper facades
  - e.g., the **`ACE_SOCK_Acceptor`**, **`ACE_SOCK_Connector`**, & **`ACE_SOCK_Stream`** classes described in Chapter 3 of C++NPv1 have their SSL-enabled counterparts: **`ACE_SSL_SOCK_Acceptor`**, **`ACE_SSL_SOCK_Connector`**, & **`ACE_SSL_SOCK_Stream`**
- The ACE SSL wrapper facades allow networked applications to ensure the integrity & confidentiality of data exchanged across a network.
- They also follow the same structure & APIs as their Socket API counterparts, which makes it easy to replace them wholesale using C++ parameterized types & the **`ACE_Svc_Handler`** template class
  - e.g., to apply the ACE wrapper facades for OpenSSL to our networked logging server we can simply remove all the OpenSSL API code & instantiate the **`ACE_Acceptor`**, **`ACE_Connector`**, & **`ACE_Svc_Handler`** with the **`ACE_SSL_SOCK_Acceptor`**, **`ACE_SSL_SOCK_Connector`**, & **`ACE_SSL_SOCK_Stream`**, respectively

Using the ACE_Acceptor (6/7)

Called back by Acceptor to accept connection into service handler

```
int TPC_Logging_Acceptor::accept_svc_handler
      (TPC_Logging_Handler *sh) {
   if (PARENT::accept_svc_handler (sh) == -1) return -1;
```

Delegate to parent (ACE_Acceptor::accept_svc_handler())

```
   SSL_clear (ssl_);  // Reset for new SSL connection.
   SSL_set_fd
      (ssl_, ACE_reinterpret_cast (int, sh->get_handle ()));

```

Verify authentication via SSL

```
   SSL_set_verify
      (ssl_,
       SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT,
       0);
   if (SSL_accept (ssl_) == -1
        || SSL_shutdown (ssl_) == -1) return -1;
   return 0;
}
```

Using the ACE_Acceptor (7/7)

Hook method dispatched by Reactor framework to shutdown acceptor

```
int TPC_Logging_Acceptor::handle_close (ACE_HANDLE h,
                                        ACE_Reactor_Mask mask)
{
  PARENT::handle_close (h, mask);
  delete this;
  return 0;
}
typedef
Reactor_Logging_Server_Adapter<TPC_Logging_Acceptor>
        TPC_Logging_Server;

ACE_FACTORY_DEFINE (TPCLS, TPC_Logging_Server)
```

`svc.conf` file for thread-per-connection client logging daemon

```
dynamic TPC_Logging_Server Service_Object *
TPCLS:_make_TPC_Logging_Server() "$TPC_LOGGING_SERVER_PORT"
```

The main() function is the same as the one we showed for the ACE Service Configurator example!!!!

The ACE_Connector Class (1/2)

Motivation

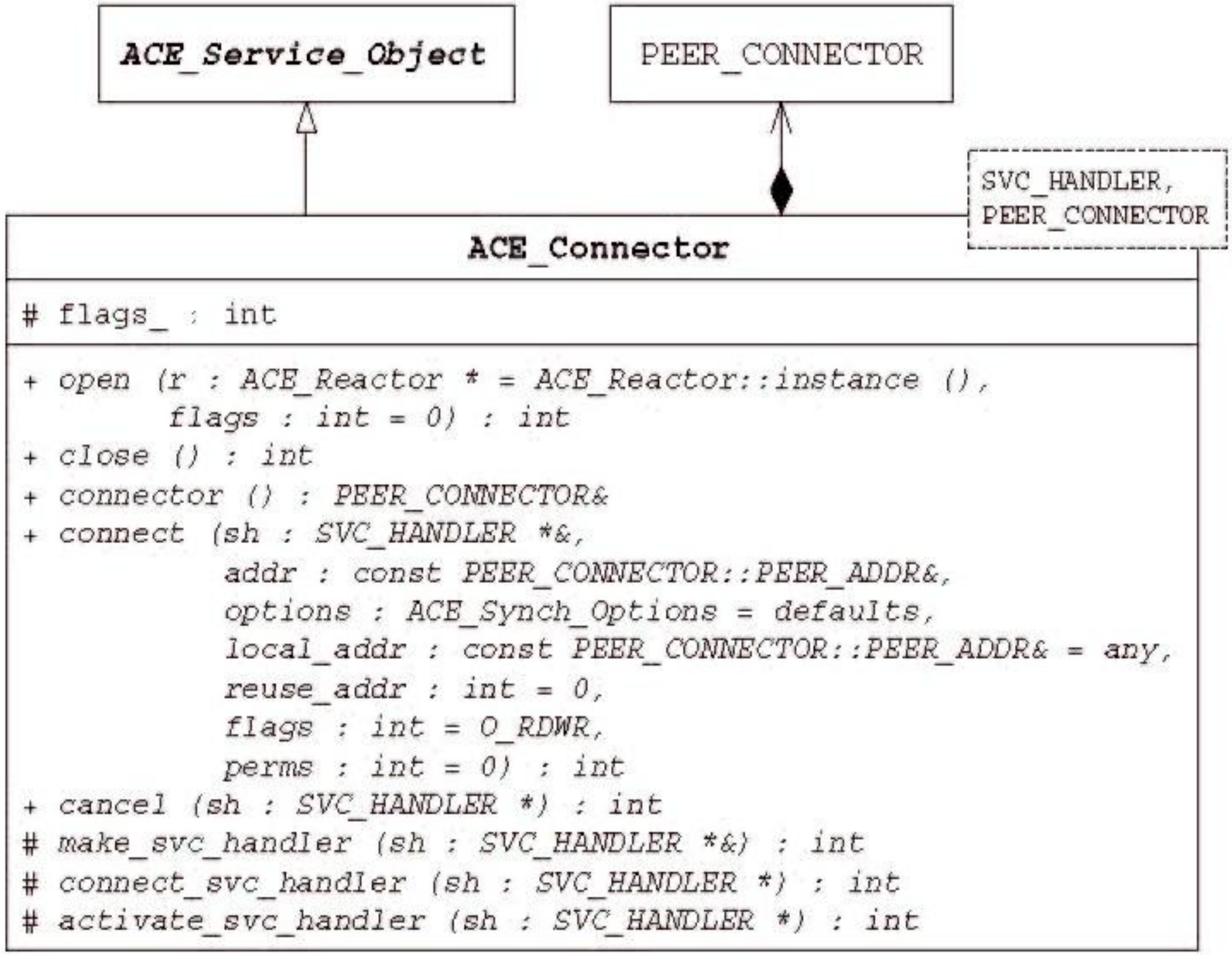
- We earlier focused on how to decouple the functionality of service handlers from the steps required to passively connect & initialize them
- It's equally useful to decouple the functionality of service handlers from the steps required to actively connect & initialize them
- Moreover, networked applications that communicate with a large number of peers may need to actively establish many connections concurrently, handling completions as they occur
- To consolidate these capabilities into a flexible, extensible, & reusable abstraction, the ACE Acceptor/Connector framework defines the **`ACE_Connector`** class

The ACE_Connector Class (2/2)

Class Capabilities

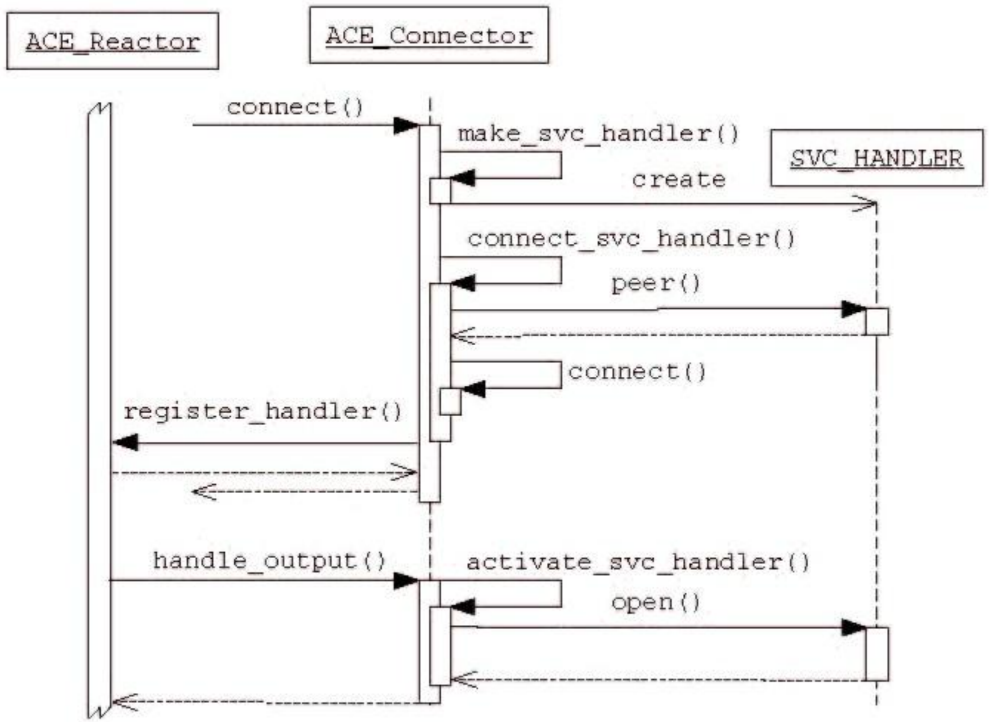
- This class is a factory class that implements the Connector role in the Acceptor/Connector pattern to provide the following capabilities:
  - It decouples the active connection establishment & service initialization logic from the processing performed by a service handler after it's connected & initialized
  - It provides an IPC factory that can actively establish connections with a peer acceptor either synchronously or reactively
  - The type of this IPC endpoint can be parameterized with many of ACE's IPC wrapper facade classes, thereby separating lower-level connection mechanisms from application-level service initialization policies
  - It automates the steps necessary to connect the IPC endpoint actively as well as to create & activate its associated service handler
  - Since **`ACE_Connector`** derives from **`ACE_Service_Object`** it inherits all the event handling & dynamic configuration capabilities provided by the ACE Reactor & ACE Service Configurator frameworks

The ACE_Connector Class API



This class handles *variability* of IPC mechanism & service handler via a *common* connection establishment & service handler initialization API

Combining ACE_Connector w/Reactor



- An instance of **`ACE_Connector`** can be registered with the ACE Reactor framework for CONNECT events
- The Reactor framework will then dispatch the **`ACE_Acceptor::handle_output()`** when non-blocking connections complete

ACE_Synch_Options for ACE_Connector

- Each **`ACE_Connector::connect()`** call tries to establish a connection with its peer
- If **`connect()`** gets an immediate indication of connection success or failure, it ignores the **`ACE_Synch_Options`** parameter
- If it doesn't get an immediate indication of connection success/failure, however, **`connect()`** uses its **`ACE_Synch_Options`** parameter to vary completion processing

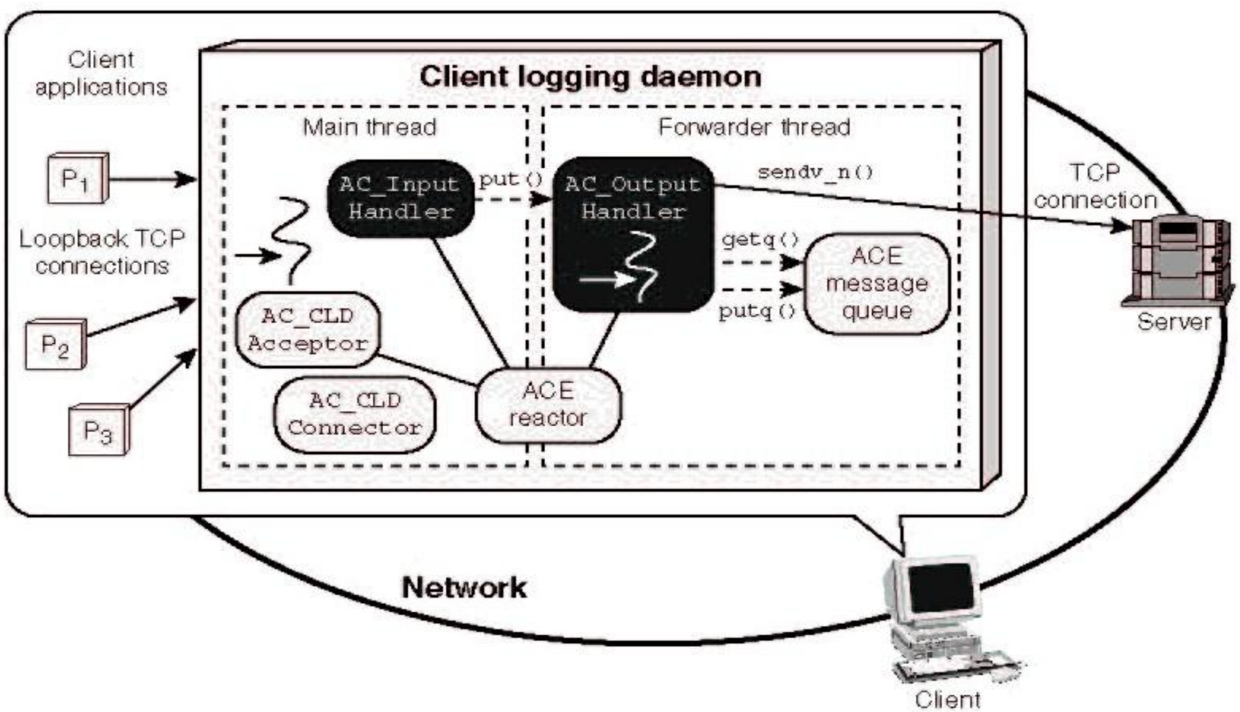
```
class ACE_Synch_Options {
  // Options flags for controlling synchronization.
  enum { USE_REACTOR = 1, USE_TIMEOUT = 2 };
  ACE_Synch_Options
    (u_long options = 0,
     const ACE_Time_Value &timeout = ACE_Time_Value::zero,
     const void *act = 0);
};
```

- The adjacent table illustrates how **`connect()`** behaves depending on its **`ACE_Synch_Options`** parameters

Reactor	Timeout	Behavior
Yes	0,0	Return −1 with `errno` `EWOULDBLOCK`; service handler is closed via reactor event loop.
Yes	Time	Return −1 with `errno` `EWOULDBLOCK`; wait up to specified amount of time for completion using the reactor.
Yes	`NULL`	Return −1 with `errno` `EWOULDBLOCK`; wait for completion indefinitely using the reactor.
No	0,0	Close service handler directly; return −1 with `errno` `EWOULDBLOCK`.
No	Time	Block in `connect_svc_handler()` up to specified amount of time for completion; if still not completed, return −1 with `errno` `ETIME`.
No	`NULL`	Block in `connect_svc_handler()` indefinitely for completion.

Using the ACE_Connector Class (1/24)

- This example applies the ACE Acceptor/Connector framework to enhance our earlier client logging daemon
  - It also integrates with the ACE Reactor & Task frameworks
- This client logging daemon version uses two threads to perform its input & output tasks



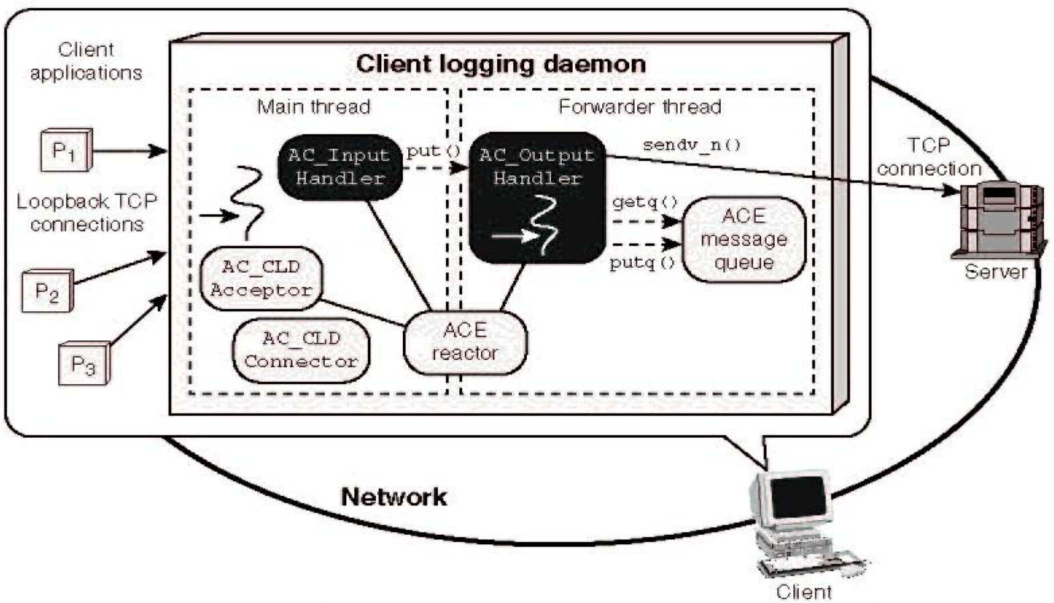
Using the ACE_Connector Class (2/24)

Input processing

- The main thread uses the singleton **`ACE_Reactor`**, an **`ACE_Acceptor`**, & an **`ACE_Svc_Handler`** passive object to read log records from sockets connected to client applications via the network loopback device
- Each log record is queued in a second **`ACE_Svc_Handler`** that runs as an active object

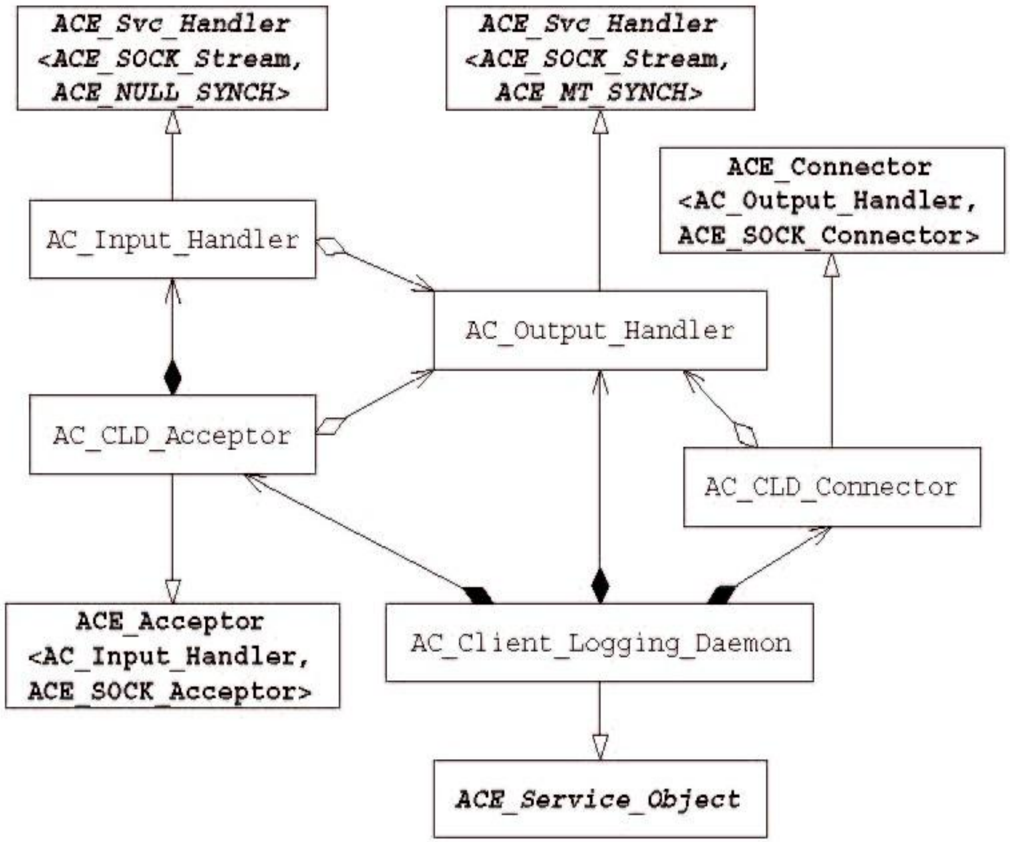
Output processing

- The active object **`ACE_Svc_Handler`** runs in its own thread, dequeueing messages from its message queue, buffering the messages into chunks, & forwarding these chunks to the server logging daemon over a TCP connection
- A subclass of **`ACE_Connector`** is used to (re)establish & authenticate connections with the logging server



Using the ACE_Connector Class (3/24)

- The classes comprising the client logging daemon based on the ACE Acceptor/Connector framework are:
  - **`AC_Input_Handler`**: A target of callbacks from the **`ACE_Reactor`** that receives log records from clients, stores each in an **`ACE_Message_Block`**, & passes them to **`AC_Output_Handler`** for processing
  - **`AC_Output_Handler`**: An active object that runs in its own thread, whose **`put()`** method enqueues message blocks passed to it from the **`AC_Input_Handler`** & whose **`svc()`** method dequeues messages from its synchronized message queue & forwards them to the logging server



  - **`AC_CLD_Acceptor`**: A factory that passively accepts connections from clients & registers them with the singleton **`ACE_Reactor`** to be processed by the **`AC_Input_Handler`**
  - **`AC_CLD_Connector`**: A factory that actively (re)establishes & authenticates connections with the logging server
  - **`AC_Client_Logging_Daemon`**: A facade class that integrates the other classes together

Using the ACE_Connector Class (4/24)

```
class AC_Input_Handler
  : public ACE_Svc_Handler<ACE_SOCK_Stream, ACE_NULL_SYNCH>
{
```

Become a service handler to receive logging records from clients

```
public:
  AC_Input_Handler (AC_Output_Handler *handler = 0)
    : output_handler_ (handler) {}
  virtual int open (void *); // Initialization hook method.
  virtual int close (u_int = 0); // Shutdown hook method.
```

Hook methods dispatched by Acceptor/Connector framework

```
protected:
  virtual int handle_input (ACE_HANDLE handle);
  virtual int handle_close (ACE_HANDLE = ACE_INVALID_HANDLE,
                            ACE_Reactor_Mask = 0);
```

Hook methods dispatched by Reactor framework

```
  // Pointer to the output handler.
  AC_Output_Handler *output_handler_;

  // Keep track of connected client handles.
  ACE_Handle_Set connected_clients_;
```

Sidebar: Single vs. Multiple Service Handlers

- The server logging daemon implementation in **`ACE_Acceptor`** example dynamically allocates a new service handler for each connected client, whereas this client logging daemon implementation uses a single service handler for all connected clients
- The rationale & tradeoffs for these approaches are:
  - If each service handler maintains separate state information for each client (in addition to the connection handle) then allocating a service handler per client is generally the most straightforward design
  - If each service handler does not maintain separate state for each client, then a server that allocates one service handler for all clients can potentially use less space & perform faster than if it allocates a handler dynamically for each client
  - It's generally much easier to manage memory if a separate service handler is allocated dynamically for each client since the ACE Acceptor/Connector framework classes embody the most common behavior for this case---the service handler simply calls **`destroy()`** from its **`handle_close()`** hook method
  - If service handler initialization can be performed from multiple threads, such as when using multiple dispatching threads with **`ACE_WFMO_Reactor`**, the design must take possible race conditions into account & use appropriate synchronization to avoid mishandling connections

Using the ACE_Connector Class (5/24)

Dispatched by Reactor framework when client logging records arrive

```
int AC_Input_Handler::handle_input (ACE_HANDLE handle)
{
  ACE_Message_Block *mblk = 0;
  Logging_Handler logging_handler (handle);
```

Read & enqueue client logging record

```
  if (logging_handler.recv_log_record (mblk) != -1)
    if (output_handler_->put (mblk->cont ()) != -1) {
      mblk->cont (0);
      mblk->release ();
      return 0; // Success return.
    } else mblk->release ();
  return -1; // Error return.
}
```

Using the ACE_Connector Class (6/24)

```
int AC_Input_Handler::open (void *) {
  ACE_HANDLE handle = peer ().get_handle ();
  if (reactor ()->register_handler
        (handle, this, ACE_Event_Handler::READ_MASK) == -1)
```

Register same event handler to READ events for all handles

```
    return -1;
  connected_clients_.set_bit (handle);
```

Track connected clients

```
  return 0;
}
```

```
int AC_Input_Handler::handle_close (ACE_HANDLE handle,
                                    ACE_Reactor_Mask) {
  connected_clients_.clr_bit (handle);
  return ACE_OS::closesocket (handle);
}
```

Track disconnected clients

Using the ACE_Connector Class (7/24)

```
int AC_Input_Handler::close (u_int) {
  ACE_Message_Block *shutdown_message = 0;
  ACE_NEW_RETURN
    (shutdown_message,
     ACE_Message_Block (0, ACE_Message_Block::MB_STOP), -1);
  output_handler_->put (shutdown_message);

  reactor ()->remove_handler
    (connected_clients_, ACE_Event_Handler::READ_MASK);

  return output_handler_->wait ();
}
```

Initiate shutdown protocol

Remove all the connected clients

Barrier synchronization

Using the ACE_Connector Class (8/24)

```
class AC_Output_Handler
  : public ACE_Svc_Handler<ACE_SOCK_Stream, ACE_MT_SYNCH> {
```

Become a service handler for sending logging records to server logging daemon

```
public:
  enum { QUEUE_MAX = sizeof (ACE_Log_Record) * ACE_IOV_MAX };

  virtual int open (void *);
```

Dispatched by Acceptor/Connector framework to initiate connections

```
  virtual int put (ACE_Message_Block *, ACE_Time_Value * =
0);
```

Entry point into AC_Output_Handler

```
protected:
  // Pointer to connection factory for <AC_Output_Handler>.
  AC_CLD_Connector *connector_;

  virtual int handle_input (ACE_HANDLE handle);
```

Dispatched by Reactor when connection to server logging daemon disconnects

Using the ACE_Connector Class (9/24)

```
  virtual int svc ();
```

Hook method that ACE Task framework uses to forward log records to server logging daemon

```
  // Send buffered log records using a gather-write operation.
  virtual int send (ACE_Message_Block *chunk[], size_t &count);
};

#if !defined (FLUSH_TIMEOUT)
#define FLUSH_TIMEOUT 120 /* 120 seconds == 2 minutes. */
#endif /* FLUSH_TIMEOUT */

int AC_Output_Handler::put (ACE_Message_Block *mb,
                            ACE_Time_Value *timeout) {
  int result;
  while ((result = putq (mb, timeout)) == -1)
    if (msg_queue ()->state () !=
ACE_Message_Queue_Base::PULSED)
      break;
  return result;
}
```

Implements reconnection logic

Using the ACE_Connector Class (10/24)

```
int AC_Output_Handler::open (void *connector) {
  connector_ =
    ACE_static_cast (AC_CLD_Connector *, connector);
  int bufsiz = ACE_DEFAULT_MAX_SOCKET_BUFSIZ;
  peer ().set_option (SOL_SOCKET, SO_SNDBUF,
                      &bufsiz, sizeof bufsiz);
  if (reactor ()->register_handler
        (this, ACE_Event_Handler::READ_MASK) == -1)
```

Register to receive a callback when connection to server logging daemon breaks

```
    return -1;
  if (msg_queue ()->activate ()
      == ACE_Message_Queue_Base::ACTIVATED) {
    msg_queue ()->high_water_mark (QUEUE_MAX);
    return activate (THR_SCOPE_SYSTEM);
```

Become an active object the first time we're called

```
  } else return 0;
}
```

Using the ACE_Connector Class (11/24)

```
int AC_Output_Handler::svc () {
  ACE_Message_Block *chunk[ACE_IOV_MAX];
  size_t message_index = 0;
  ACE_Time_Value time_of_last_send (ACE_OS::gettimeofday ());
  ACE_Time_Value timeout;
  ACE_Sig_Action no_sigpipe ((ACE_SignalHandler) SIG_IGN);
  ACE_Sig_Action original_action;
  no_sigpipe.register_action (SIGPIPE, &original_action);

  for (;;) {
    if (message_index == 0) {
      timeout = ACE_OS::gettimeofday ();
      timeout += FLUSH_TIMEOUT;
    }
    ACE_Message_Block *mblk = 0;
    if (getq (mblk, &timeout) == -1) {
      if (errno == ESHUTDOWN) {
        if (connector_->reconnect () == -1) break;
        continue;
      } else if (errno != EWOULDBLOCK) break;
      else if (message_index == 0) continue;
```

Ignore SIGPIPE signal

Wait a bounded period of time for next message

Reconnect protocol

Using the ACE_Connector Class (12/24)

```
    } else {
      if (mblk->size () == 0
          && mblk->msg_type () ==
ACE_Message_Block::MB_STOP)
        { mblk->release (); break; }
      chunk[message_index] = mblk;
      ++message_index;
    }
    if (message_index >= ACE_IOV_MAX
        || (ACE_OS::gettimeofday () - time_of_last_send
            >= FLUSH_TIMEOUT)) {
      if (send (chunk, message_index) == -1) break;
      time_of_last_send = ACE_OS::gettimeofday ();
    }
  }

  if (message_index > 0) send (chunk, message_index);
  no_sigpipe.restore_action (SIGPIPE, original_action);
  return 0;
}
```

Reconnect protocol

Send buffered messages at appropriate time

Send any remaining buffered messages

Restore signal disposition

Using the ACE_Connector Class (13/24)

This method is dispatched by Reactor when connection to server logging daemon is broken

```
int AC_Output_Handler::handle_input (ACE_HANDLE h)
{
  peer ().close ();
  reactor ()->remove_handler
    (h, ACE_Event_Handler::READ_MASK
        | ACE_Event_Handler::DONT_CALL);
  msg_queue ()->pulse ();
  return 0;
}
```

Cleanup resources associated with broken connection

Unblock the output thread from its message queue so it can retrigger the connection

Using the ACE_Connector Class (14/24)

```
class AC_CLD_Acceptor
  : public ACE_Acceptor<AC_Input_Handler, ACE_SOCK_Acceptor>
{
```

Become an acceptor

```
public:
  AC_CLD_Acceptor (AC_Output_Handler *handler = 0)
    : output_handler_ (handler), input_handler_ (handler) {}
protected:
  typedef ACE_Acceptor<AC_Input_Handler, ACE_SOCK_Acceptor>
          PARENT;
```

Factory method dispatched by Acceptor/Connector framework

```
  virtual int make_svc_handler (AC_Input_Handler *&sh);
```

Hook method dispatched by Reactor framework

```
  virtual int handle_close (ACE_HANDLE = ACE_INVALID_HANDLE,
                            ACE_Reactor_Mask = 0);

  // Pointer to the output handler.
  AC_Output_Handler *output_handler_;

  // Single input handler.
  AC_Input_Handler input_handler_;
```

Using the ACE_Connector Class (15/24)

```
class AC_CLD_Connector
  : public ACE_Connector<AC_Output_Handler, ACE_SOCK_Connector>
{
```

Become a connector

```
public:
  typedef ACE_Connector<AC_Output_Handler, ACE_SOCK_Connector>
          PARENT;

  AC_CLD_Connector (AC_Output_Handler *handler = 0)
    : handler_ (handler), ssl_ctx_ (0), ssl_ (0) {}

  virtual ~AC_CLD_Connector (void) { // Frees the SSL resources.
    SSL_free (ssl_);
    SSL_CTX_free (ssl_ctx_);
  }

  // Initialize the Connector.
  virtual int open (ACE_Reactor *r = ACE_Reactor::instance (),
                    int flags = 0);
```

Using the ACE_Connector Class (16/24)

Connection establishment & authentication hook method called by Acceptor/Connector framework

```
protected:
  virtual int connect_svc_handler
    (AC_Output_Handler *svc_handler,
     const ACE_SOCK_Connector::PEER_ADDR &remote_addr,
     ACE_Time_Value *timeout,
     const ACE_SOCK_Connector::PEER_ADDR &local_addr,
     int reuse_addr, int flags, int perms);

  // Pointer to <AC_Output_Handler> we're connecting.
  AC_Output_Handler *handler_;

  // Address at which logging server listens for connections.
  ACE_INET_Addr remote_addr_;

  SSL_CTX *ssl_ctx_; // The SSL "context" data structure.

  // The SSL data structure corresponding to authenticated
SSL
  // connections.
  SSL *ssl_;
};
```

Using the ACE_Connector Class (17/24)

```
#if !defined (CLD_CERTIFICATE_FILENAME)
#  define CLD_CERTIFICATE_FILENAME "cld-cert.pem"
#endif /* !CLD_CERTIFICATE_FILENAME */
#if !defined (CLD_KEY_FILENAME)
#  define CLD_KEY_FILENAME "cld-key.pem"
#endif /* !CLD_KEY_FILENAME */

int AC_CLD_Connector::open (ACE_Reactor *r, int flags) {
  if (PARENT::open (r, flags) != 0) return -1;
  OpenSSL_add_ssl_algorithms ();
  ssl_ctx_ = SSL_CTX_new (SSLv3_client_method ());
  if (ssl_ctx_ == 0) return -1;
  if (SSL_CTX_use_certificate_file (ssl_ctx_,
                                    CLD_CERTIFICATE_FILENAME,
                                    SSL_FILETYPE_PEM) <= 0
     || SSL_CTX_use_PrivateKey_file (ssl_ctx_,
                                     CLD_KEY_FILENAME,
                                     SSL_FILETYPE_PEM) <= 0
     || !SSL_CTX_check_private_key (ssl_ctx_))
    return -1;
  ssl_ = SSL_new (ssl_ctx_);
  if (ssl_ == 0) return -1;
  return 0;
}
```

Perform client-side of SSL authentication

Using the ACE_Connector Class (18/24)

```
int AC_CLD_Connector::connect_svc_handler
     (AC_Output_Handler *svc_handler,
      const ACE_SOCK_Connector::PEER_ADDR &remote_addr,
      ACE_Time_Value *timeout,
      const ACE_SOCK_Connector::PEER_ADDR &local_addr,
      int reuse_addr, int flags, int perms) {
  if (PARENT::connect_svc_handler
        (svc_handler, remote_addr, timeout,
         local_addr, reuse_addr, flags, perms) == -1) return -1;
  SSL_clear (ssl_);
  SSL_set_fd (ssl_, ACE_reinterpret_cast
                            (int, svc_handler->get_handle ()));

  SSL_set_verify (ssl_, SSL_VERIFY_PEER, 0);

  if (SSL_connect (ssl_) == -1
        || SSL_shutdown (ssl_) == -1) return -1;
  remote_addr_ = remote_addr;
  return 0;
}
```

Using the ACE_Connector Class (19/24)

Called when connection has broken

```
int AC_CLD_Connector::reconnect () {
  // Maximum number of times to retry connect.
  const size_t MAX_RETRIES = 5;
  ACE_Time_Value timeout (1);
  size_t i;
  for (i = 0; i < MAX_RETRIES; ++i) {
    ACE_Synch_Options options
(ACE_Synch_Options::USE_TIMEOUT,
                               timeout);
    if (i > 0) ACE_OS::sleep (timeout);
    if (connect (handler_, remote_addr_, options) == 0)
      break;
    timeout *= 2;
  }
  return i == MAX_RETRIES ? -1 : 0;
}
```

Exponential backoff algorithm

Using the ACE_Connector Class (20/24)

```
class AC_Client_Logging_Daemon : public ACE_Service_Object {
```

Integrate with ACE Service Configurator framework

```
protected:
  // Factory that passively connects the <AC_Input_Handler>.
  AC_CLD_Acceptor acceptor_;

  // Factory that actively connects the <AC_Output_Handler>.
  AC_CLD_Connector connector_;

  // The <AC_Output_Handler> connected by <AC_CLD_Connector>.
  AC_Output_Handler output_handler_;
public:
  AC_Client_Logging_Daemon ()
    : acceptor_ (&output_handler_),
      connector_ (&output_handler_) {}

  virtual int init (int argc, ACE_TCHAR *argv[]);
  virtual int fini ();
  virtual int info (ACE_TCHAR **bufferp, size_t length = 0)
const;
  virtual int suspend ();
  virtual int resume ();
```

Hook method dispatched by ACE Service Configurator framework

Using the ACE_Connector Class (21/24)

Hook method dispatched by ACE Service Configurator framework

```
int AC_Client_Logging_Daemon::init
        (int argc, ACE_TCHAR *argv[]) {
  u_short cld_port = ACE_DEFAULT_SERVICE_PORT;
  u_short sld_port = ACE_DEFAULT_LOGGING_SERVER_PORT;
  ACE_TCHAR sld_host[MAXHOSTNAMELEN];
  ACE_OS_String::strcpy (sld_host, ACE_LOCALHOST);
  ACE_Get_Opt get_opt (argc, argv, ACE_TEXT ("p:r:s:"), 0);
  get_opt.long_option (ACE_TEXT ("client_port"), 'p',
                       ACE_Get_Opt::ARG_REQUIRED);
  get_opt.long_option (ACE_TEXT ("server_port"), 'r',
                       ACE_Get_Opt::ARG_REQUIRED);
  get_opt.long_option (ACE_TEXT ("server_name"), 's',
                       ACE_Get_Opt::ARG_REQUIRED);

  for (int c; (c = get_opt ()) != -1;)
    switch (c) {
    case 'p': // Client logging daemon acceptor port number.
      cld_port = ACE_static_cast
        (u_short, ACE_OS::atoi (get_opt.opt_arg ()));
```

Using the ACE_Connector Class (22/24)

```
    case 'r': // Server logging daemon acceptor port number.
      sld_port = ACE_static_cast
        (u_short, ACE_OS::atoi (get_opt.opt_arg ()));
      break;
    case 's': // Server logging daemon hostname.
      ACE_OS_String::strsncpy
        (sld_host, get_opt.opt_arg (), MAXHOSTNAMELEN);
      break;
    }

  ACE_INET_Addr cld_addr (cld_port);
  ACE_INET_Addr sld_addr (sld_port, sld_host);

  if (acceptor_.open (cld_addr) == -1) return -1;
  AC_Output_Handler *oh = &output_handler_;

  if (connector_.connect (oh, sld_addr) == -1)
  { acceptor_.close (); return -1; }
  return 0;
```

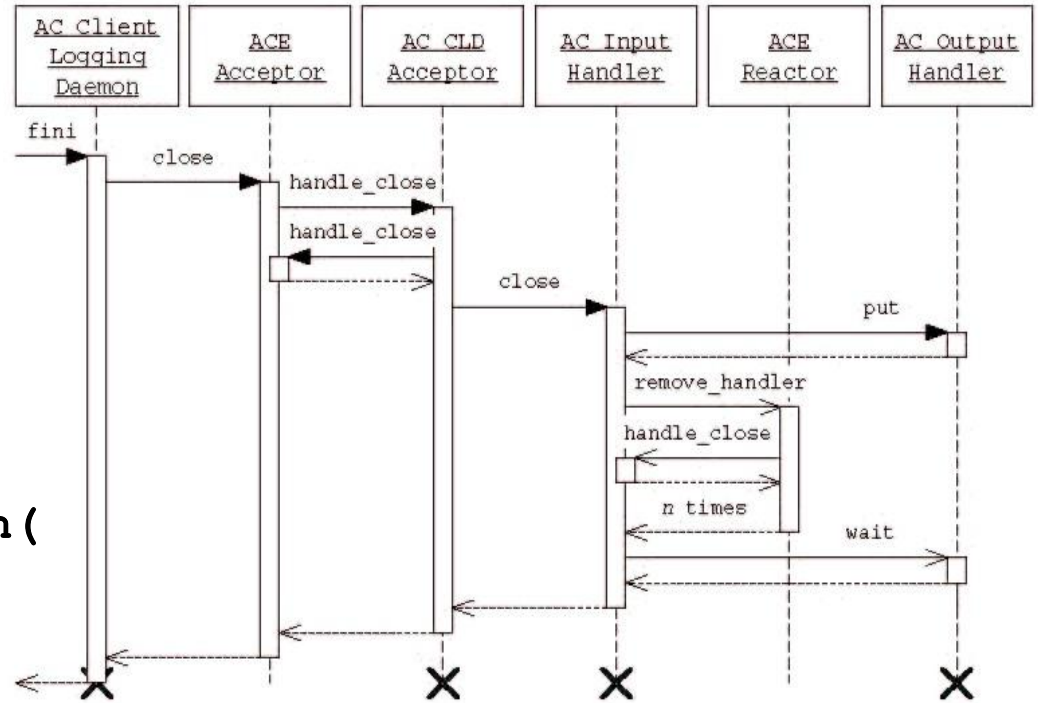
Establish connection passively

Establish connection actively

Using the ACE_Connector Class (23/24)

`svc.conf` file for producer/consumer client logging daemon

```
dynamic
AC_Client_Logging_Daemon
Service_Object *
AC_CLD
:
_make_AC_Client_Logging_Daemon(
)
"-p
$CLIENT_LOGGING_DAEMON_PORT"
```



Shutdown hook method dispatched by ACE Service Configurator framework

```
int AC_Client_Logging_Daemon::fini ()
{ return acceptor_.close (); }

ACE_FACTORY_DEFINE (AC_CLD,
AC_Client_Logging_Daemon)
```

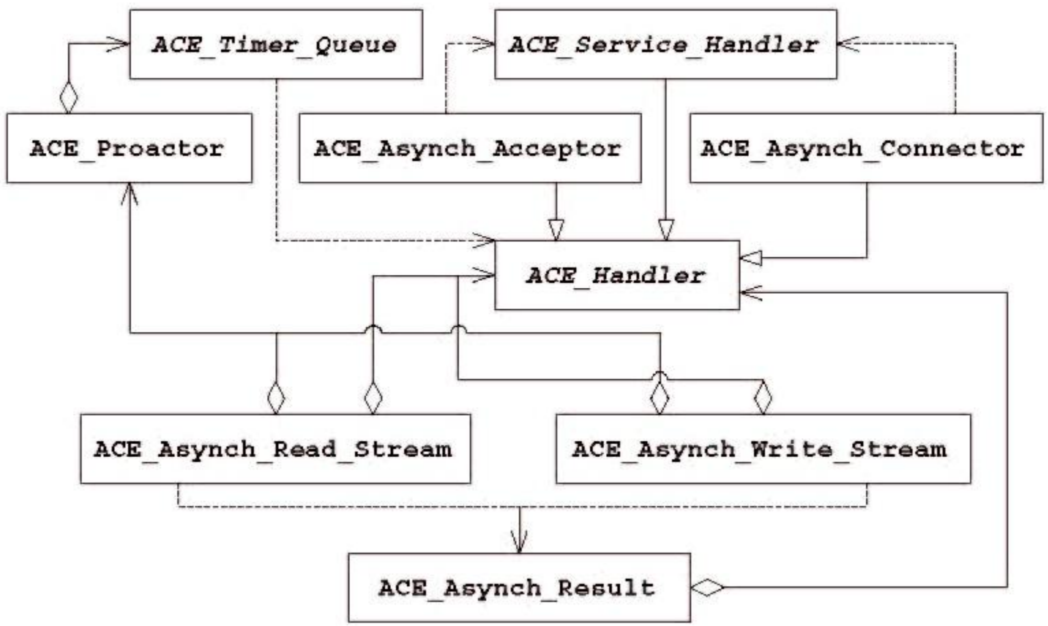
Using the ACE_Connector Class (24/24)

```
#include "ace/OS.h"
#include "ace/Reactor.h"
#include "ace/Select_Reactor.h"
#include "ace/Service_Config.h"

int ACE_TMAIN (int argc, ACE_TCHAR *argv[]) {
  ACE_Select_Reactor *select_reactor;
  ACE_NEW_RETURN (select_reactor, ACE_Select_Reactor, 1);
  ACE_Reactor *reactor;
  ACE_NEW_RETURN (reactor, ACE_Reactor (select_reactor, 1), 1);
  ACE_Reactor::close_singleton ();
  ACE_Reactor::instance (reactor, 1);

  ACE_Service_Config::open (argc, argv);

  ACE_Reactor::instance ()->run_reactor_event_loop ();
  return 0;
}
```

This main() function is slight different from earlier ones, but still uses the ACE Service Configurator framework

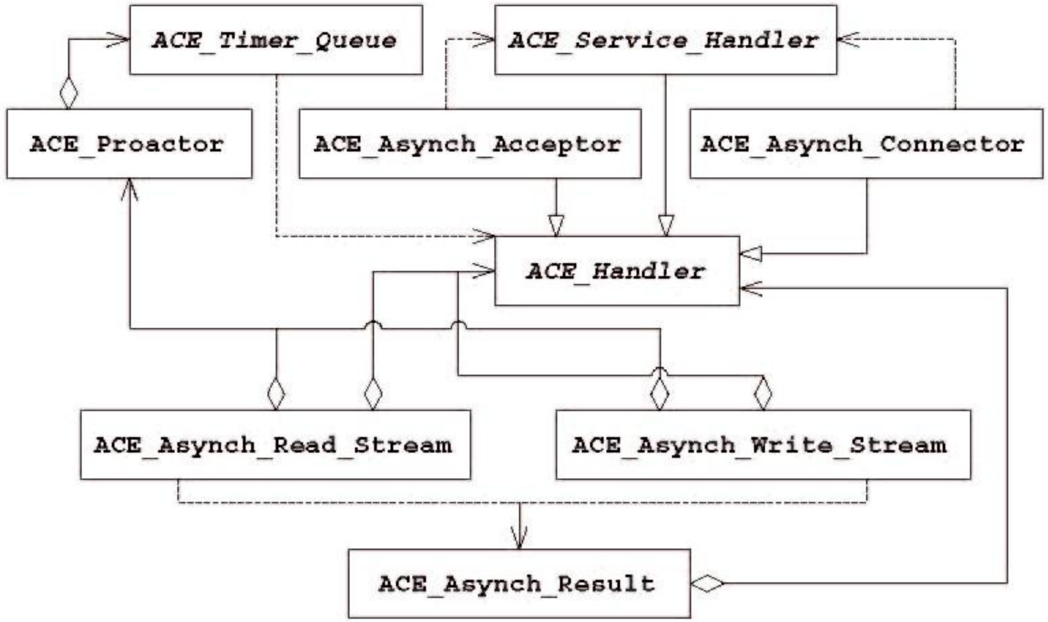
The ACE Proactor Framework

- The ACE Proactor framework alleviates reactive I/O bottlenecks without introducing the complexity & overhead of synchronous I/O & multithreading
- This framework allows an application to execute I/O operations via two phases:
  1. The application can initiate one or more asynchronous I/O operations on multiple I/O handles in parallel without having to wait until they complete
  2. As each operation completes, the OS notifies an application-defined completion handler that then processes the results from the completed I/O operation



The ACE Proactor Framework

ACE Class	Description
ACE_Handler	Defines the interface for receiving the results of asynchronous I/O operations and handling timer expirations.
ACE_Asynch_Read_Stream ACE_Asynch_Write_Stream ACE_Asynch_Result	Initiate asynchronous read and write operations on an I/O stream and associate each with an ACE_Handler object that will receive the results of those operations.
ACE_Asynch_Acceptor ACE_Asynch_Connector	An implementation of the Acceptor-Connector pattern that establishes new TCP/IP connections asynchronously.
ACE_Service_Handler	Defines the target of the ACE_Asynch_Acceptor and ACE_Asynch_Connector connection factories and provides the hook methods to initialize a TCP/IP-connected service.
ACE_Proactor	Manages timers and asynchronous I/O completion event demultiplexing. This class is analogous to the ACE_Reactor class in the ACE Reactor framework.



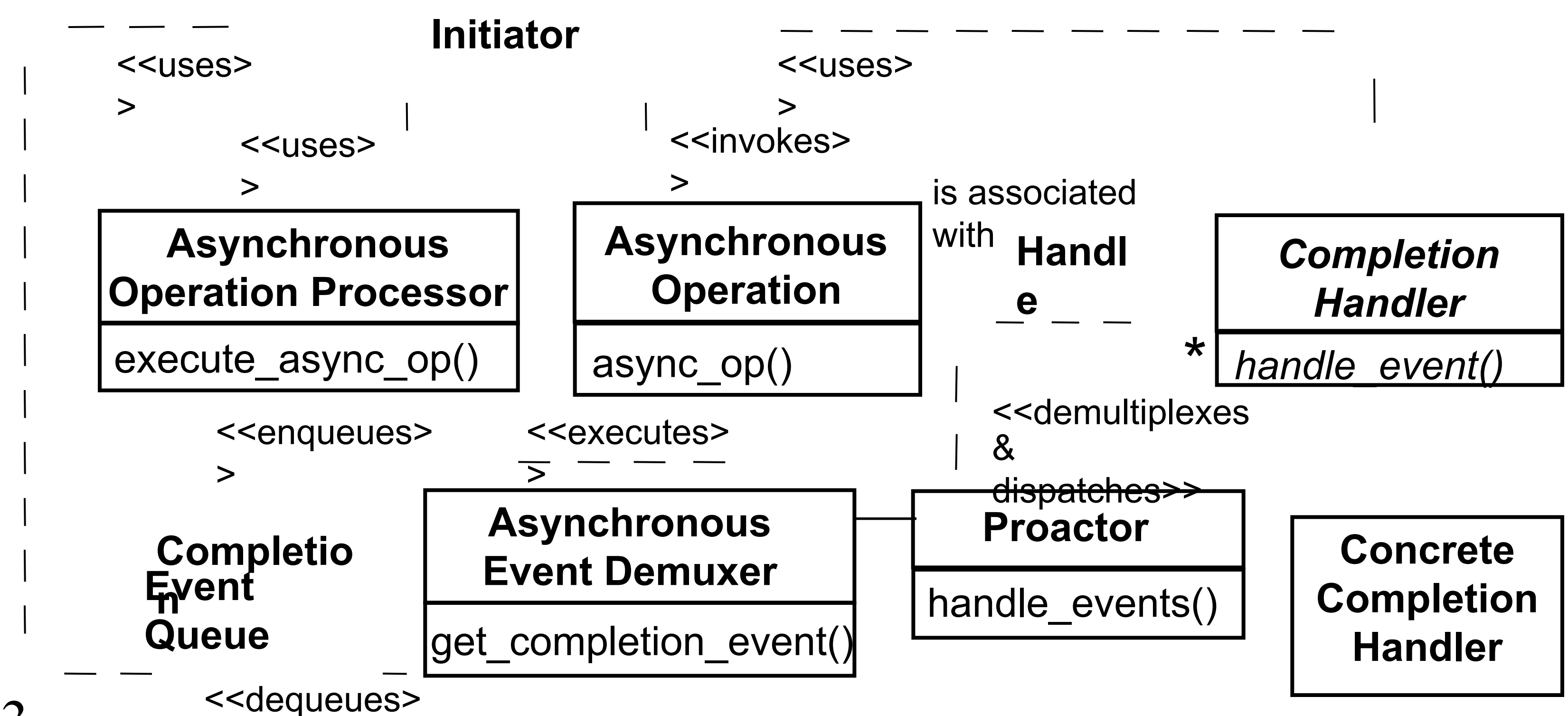
The Proactor Pattern

Problem

- Developing software that achieves the potential efficiency & scalability of async I/O is hard due to the separation in time & space of async operation invocations & their subsequent completion events

Solution

- Apply the *Proactor* architectural pattern (P2) to make efficient use of async I/O
- This pattern allows event-driven applications to efficiently demux & dispatch service requests triggered by the completion of async operations, thereby achieving performance benefits of concurrency without incurring its many liabilities



Dynamics in the Proactor Pattern

: Initiator | **: Asynchronous Operation Processor** | **: Asynchronous Operation** | **: Completion Event Queue** | **: Proactor** | **Completion Handler**

1. *Initiate operation*
2. *Process operation*
3. *Run event loop*
4. *Generate & queue completion event*
5. *Dequeue completion event & perform completion processing*

Completio n Handle
Completio n Ev. Queue
exec_async operation ()
async_operation( )
handle_events( )
even t
Resul t
Resul t
even t
Resul t
Resul t
service( )
handle event( )

Note *similarities* & *differences* with the Reactor pattern, *e.g.:*
- Both process events via callbacks
- However, it's generally easier to multi-thread a proactor

Sidebar: Asynchronous I/O Portability Issues

- The following OS platforms supported by ACE provide asynchronous I/O mechanisms:
  - **Windows platforms** that support both overlapped I/O & I/O completion ports
    - Overlapped I/O is an efficient & scalable I/O mechanism on Windows
    - Windows performs completion event demultiplexing via I/O completion ports & event handles
    - An I/O completion port is a queue managed by the Windows kernel to buffer I/O completion events
  - **POSIX platforms** that implement the POSIX.4 AIO specification
    - This specification was originally designed for disk file I/O, but can also be used for network I/O with varying degrees of success
    - An application thread can wait for completion events via **`aio_suspend()`** or be notified by real-time signals, which are tricky to integrate into an event-driven application
    - In general, POSIX.4 AIO requires extra care to program the proactive model correctly & efficiently
    - Despite UNIX's usual interchangeability of I/O system functions across IPC mechanisms, integration of the POSIX AIO facility with other IPC mechanisms, such as the Socket API, leaves much to be desired…
      - e.g., Socket API functions, such as **`connect()`** & **`accept()`**, are not integrated with the POSIX AIO model, & some AIO implementations can't handle multiple outstanding operations on a handle under all conditions

The ACE Async Read/Write Stream Classes

Motivation

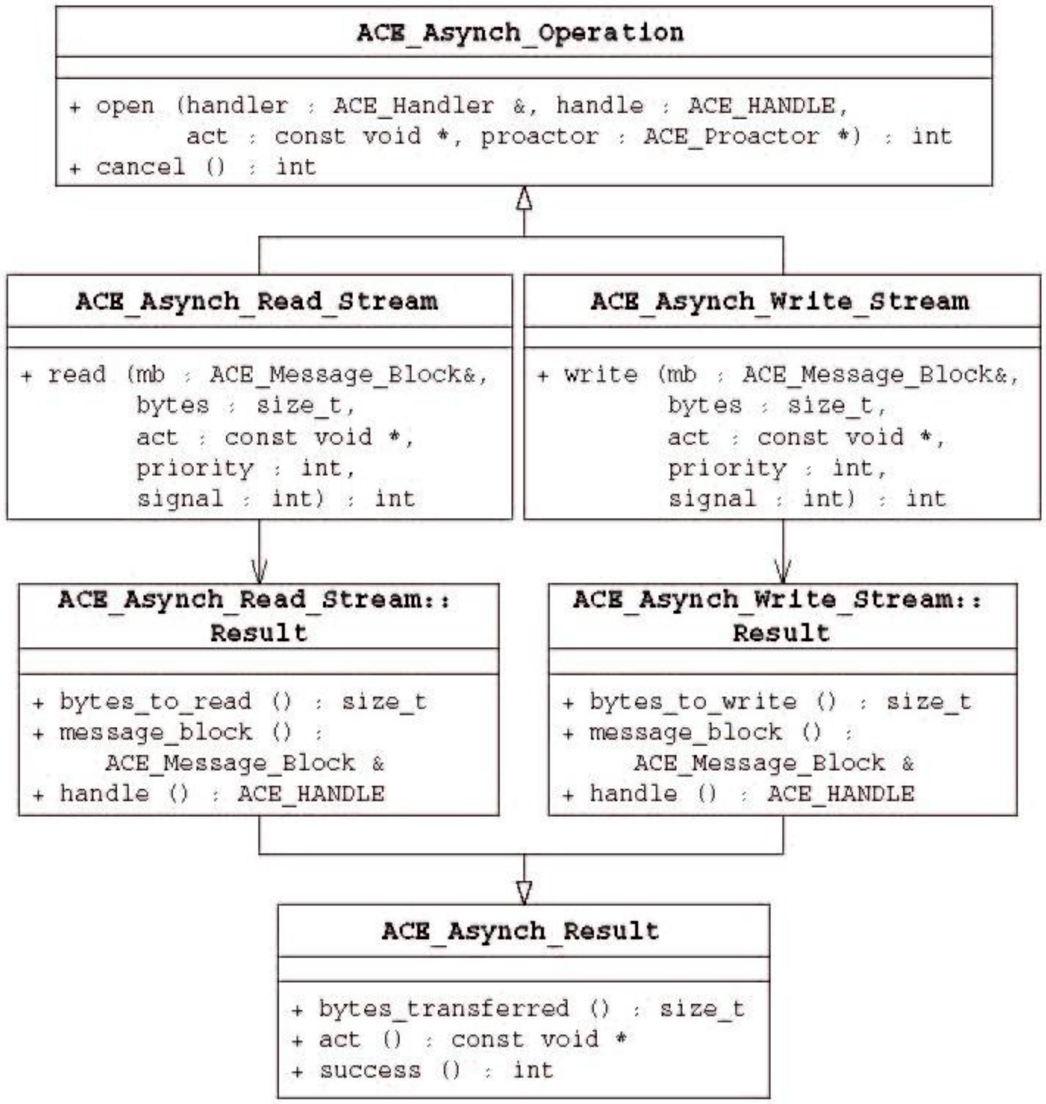
- The proactive I/O model is generally harder to program than reactive & synchronous I/O models
- In particular, there's a time/space separation between asynchronous invocation & completion handling that requires tricky state management
  - e.g., asynchronous processing is hard to program since the bookkeeping details & data fragments must be managed explicitly, rather than implicitly on the run-time stack
- There are also significant accidental complexities associated with asynchronous I/O on many OS platforms

The ACE Async Read/Write Stream Classes

Class Capabilities

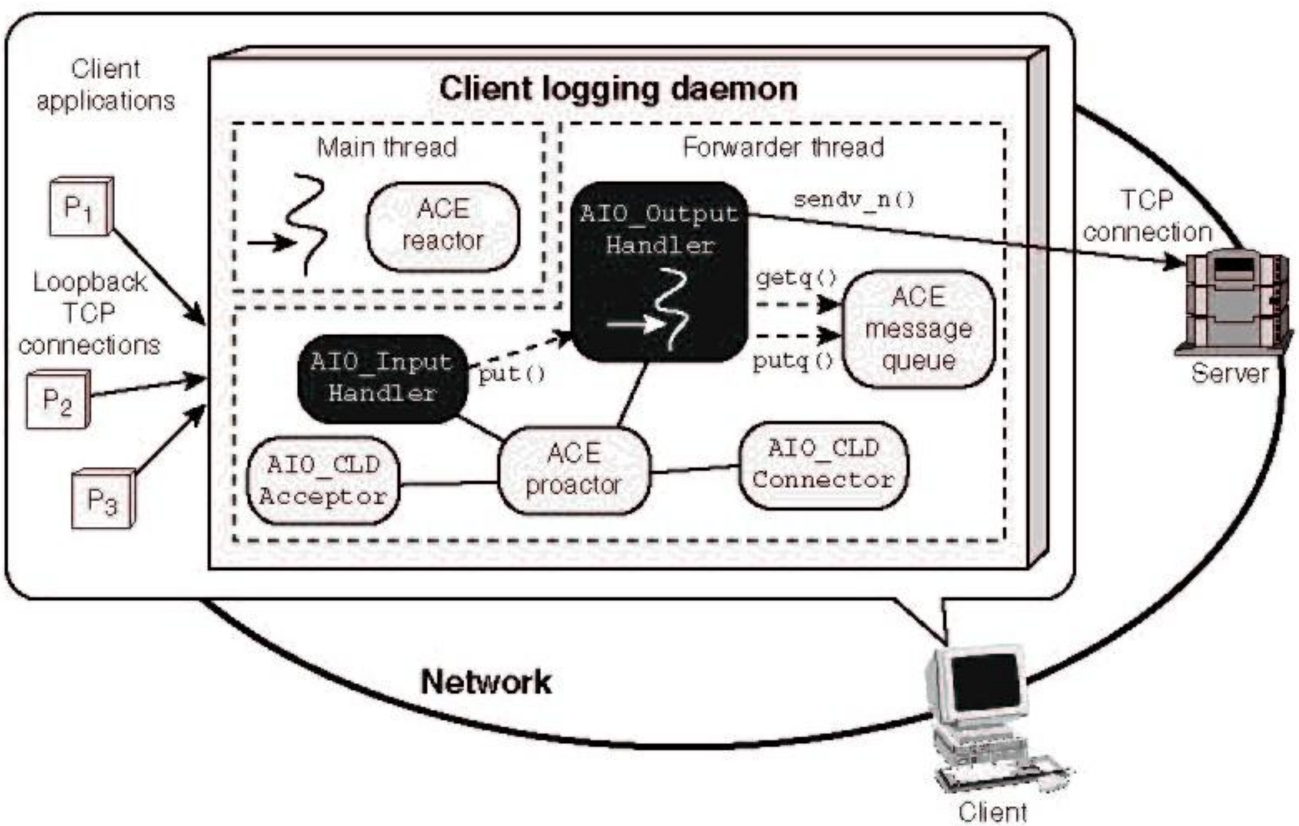
- These are factory classes that enable applications to initiate portable asynchronous **`read()`** & **`write()`** operations to provide the following capabilities:
  - They can initiate asynchronous I/O operations on a stream-oriented IPC mechanism, such as a TCP socket
  - They bind an I/O handle, an **`ACE_Handler`** object, & a **`ACE_Proactor`** to process I/O completion events correctly & efficiently
  - They create an object that carries an operation's parameters through the ACE Proactor framework to its completion handler
  - They derive from **`ACE_Asynch_Operation`**, which provides the interface to initialize the object & to request cancellation of outstanding I/O operations

The ACE Async Read/Write Stream Class APIs



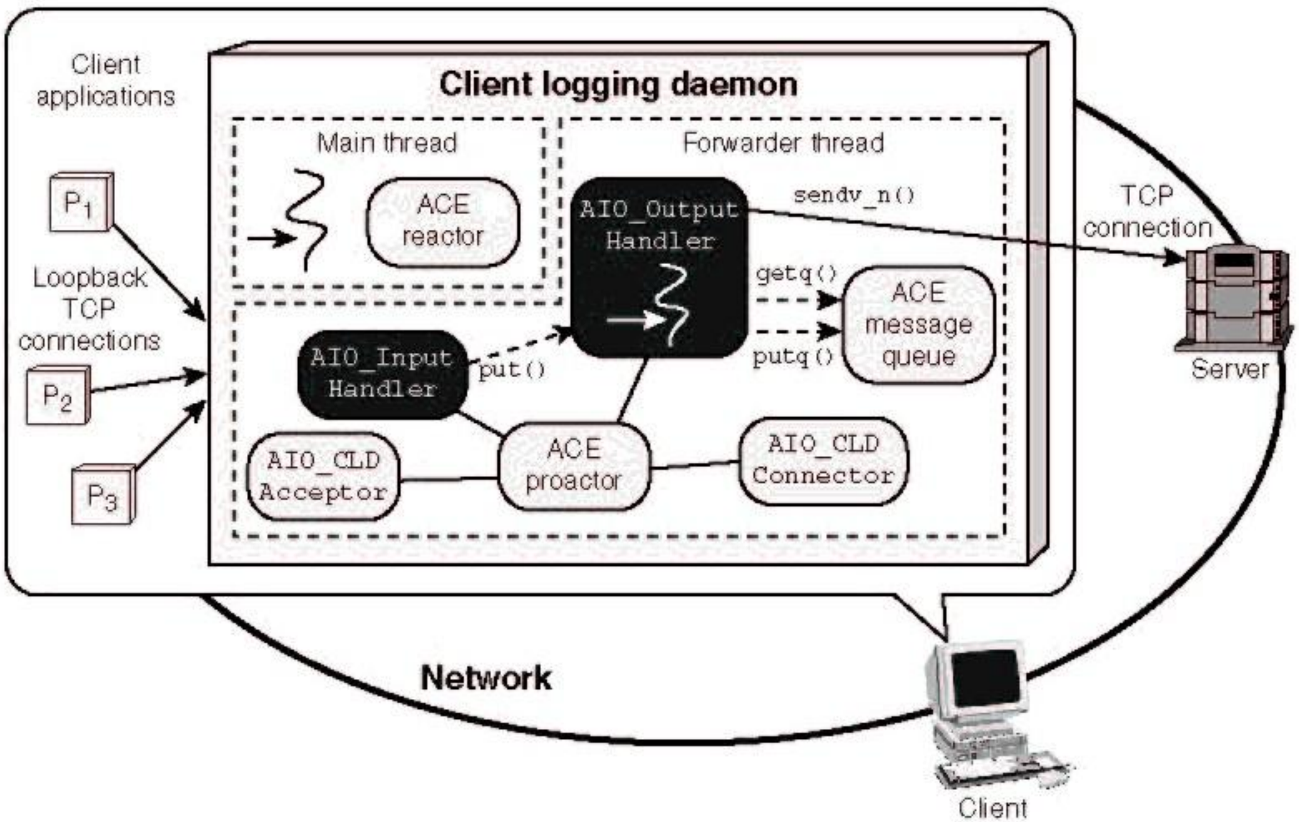
Using the ACE Async Read/Write Stream Classes (1/6)

- This example reimplements the client logging daemon service using the ACE Proactor framework
- This illustrates the use of asynchronous I/O for both input & output



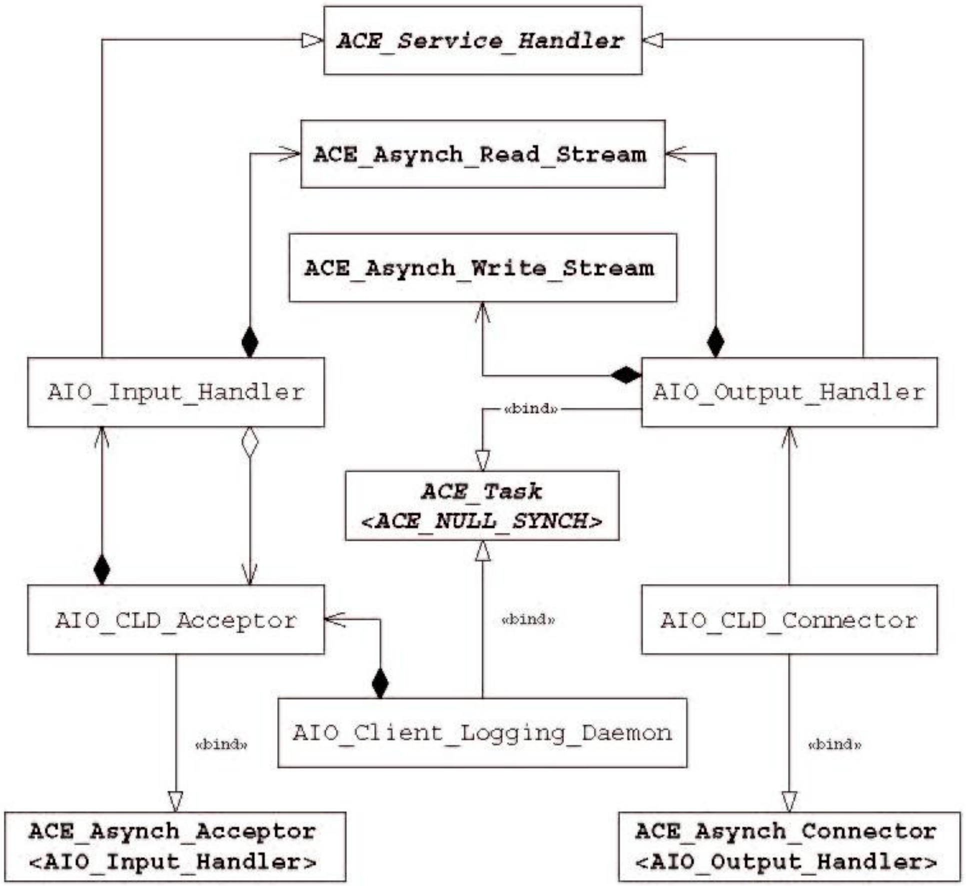
Using the ACE Async Read/Write Stream Classes (2/6)

- Although the classes used in the proactive client logging daemon service are similar to those in the Acceptor/Connector version, the proactive version uses a single application thread to initiate & handle completions for all its I/O operations



Using the ACE Async Read/Write Stream Classes (3/6)

- The classes comprising the client logging daemon based on the ACE Proactor framework are outlined below:
  - **AIO_Output_Handler**: A message forwarder that initiates asynchronous **`write()`** operations to forward messages to the logging server
  - **AIO_CLD_Connector**: A factory that actively (re)establishes & authenticates connections with the logging server & activates an **`AIO_Output_Handler`**
  - **AIO_Input_Handler**: Processes log record data received from logging clients via asynchronous **`read()`** operations & passes completed log records to **`AIO_Output_Handler`** for output processing



  - **AIO_CLD_Acceptor**: A factory that accepts connections from logging clients & creates a new **`AIO_Input_Handler`** for each
  - **AIO_Client_Logging_Daemon**: A facade class that integrate the other classes together

Using the ACE Async Read/Write Stream Classes (4/6)

```
class AIO_Output_Handler
  : public ACE_Task<ACE_NULL_SYNCH>, public ACE_Service_Handler {
```

Inherit message passing from ACE_Task & open() activation hook from ACE_Service_Handler

```
public:
  AIO_Output_Handler (): can_write_ (0) {}
  virtual ~AIO_Output_Handler ();
```

Entry point into the AIO_Output_Handler

```
  virtual int put (ACE_Message_Block *, ACE_Time_Value * = 0);
```

Hook method called by ACE_Asynch_Connector when async server connection completes

```
  virtual void open (ACE_HANDLE new_handle,
                     ACE_Message_Block &message_block);
protected:
  ACE_Asynch_Read_Stream reader_; // Detects connection loss.
  ACE_Asynch_Write_Stream writer_; // Sends records to server.
  int can_write_; // Safe to begin sending a log record?
```

We only send a single async operation at a time

```
  // Initiate the send of a log record.
  void start_write (ACE_Message_Block *mblk = 0);
};
```

Using the ACE Async Read/Write Stream Classes (5/6)

```
typedef ACE_Unmanaged_Singleton<AIO_Output_Handler,
                                ACE_Null_Mutex>
OUTPUT_HANDLER;
```

Hook method called when async server connection completes

```
void AIO_Output_Handler::open
  (ACE_HANDLE new_handle, ACE_Message_Block &) {
  ACE_SOCK_Stream temp_peer (new_handle);
  int bufsiz = ACE_DEFAULT_MAX_SOCKET_BUFSIZ;
  temp_peer.set_option (SOL_SOCKET, SO_SNDBUF,
                        &bufsiz, sizeof bufsiz);
```

Bind proactor & I/O handle to async read & write objects

```
  reader_.open (*this, new_handle, 0, proactor ());
  writer_.open (*this, new_handle, 0, proactor ());

  ACE_Message_Block *mb;
  ACE_NEW (mb, ACE_Message_Block ( ));
  reader_.read (*mb, 1);
  ACE_Sig_Action no_sigpipe ((ACE_SignalHandler) SIG_IGN);
  no_sigpipe.register_action (SIGPIPE, 0);
  can_write_ = 1;
  start_write (0);
}
```

Initiate async read operation to detect connection failure

See if there are any messages queued for delivery

Using the ACE Async Read/Write Stream Classes (6/6)

```
void AIO_Output_Handler::start_write
      (ACE_Message_Block *mblk) {
  if (mblk == 0) {
    ACE_Time_Value nonblock (0);
    getq (mblk, &nonblock);
  }
  if (mblk != 0) {
    can_write_ = 0;
    if (writer_.write (*mblk, mblk->length ()) ==
-1)
      ungetq (mblk);
  }
}
```

Initiate async write

Entry point to AIO_Output_Handler – called by AIO_Input_Handler

```
int AIO_Output_Handler::put (ACE_Message_Block *mb,
                             ACE_Time_Value *timeout) {
  if (can_write_)
  { start_write (mb); return 0; }
  return putq (mb, timeout);
}
```

Initiate async write, if possible, otherwise queue message

The ACE_Handler Class (1/2)

Motivation

- Proactive & reactive I/O models differ since proactive I/O initiation & completion are distinct steps that occur separately (possibly in different threads)
- Using separate classes for the initiation & completion processing avoids unnecessarily coupling the two

The ACE_Handler Class (2/2)

Class Capabilities

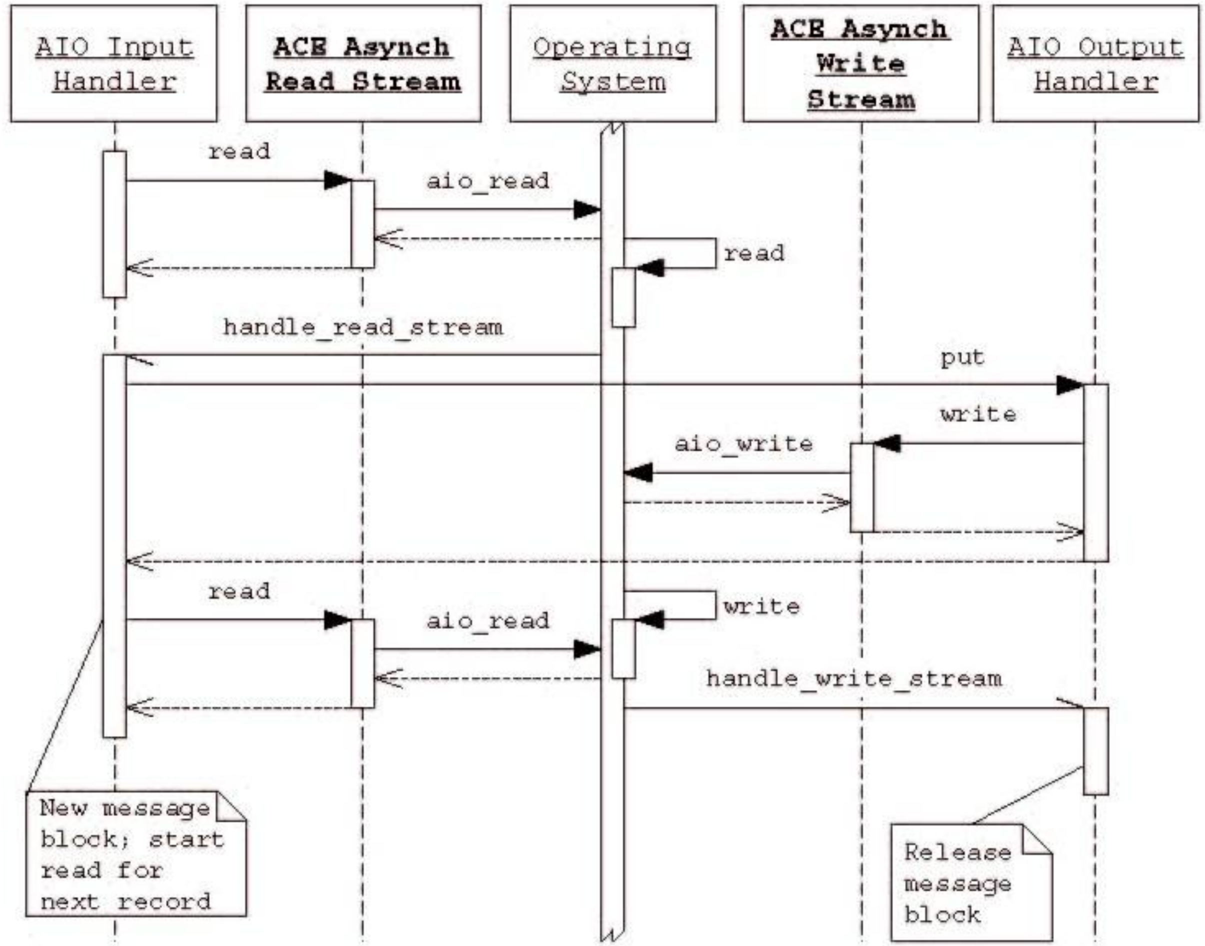
- **`ACE_Handler`** is the base class of all asynchronous completion handlers in the ACE Proactor framework
  - It plays a similar (albeit inverse) role to the **`ACE_Event_Handler`** in the Reactor framework
- This class provides the following capabilities:
  - It provides hook methods to handle completion of all asynchronous I/O operations defined in ACE, including connection establishment & I/O operations on an IPC stream
  - It provides a hook method to handle timer expiration

The ACE_Handler Class API

```
ACE_Handler
----------------------------------------------------------------
# proactor_ : ACE_Proactor *
----------------------------------------------------------------
+ handle () : ACE_HANDLE
+ handle_read_stream (result :
          const ACE_Asynch_Read_Stream::Result &)
+ handle_write_stream (result :
          const ACE_Asynch_Write_Stream::Result &)
+ handle_time_out (tv : const ACE_Time_Value &,
                   act : const void *)
+ handle_accept (result :
          const ACE_Asynch_Accept::Result &)
+ handle_connect (result :
          const ACE_Asynch_Connect::Result &)
```

Using the ACE_Handler Class (1/6)

•The **`AIO_Input_Handler`** class receives log records from logging clients by initiating asynchronous **`read()`** calls & assembling the data fragments into log records that are then forwarded to the server logging daemon via **`AIO_Output_Handler`**



•This class uses the Proactor pattern & asynchronous input operations to concurrently process I/O requests across all logging clients using a single thread

Using the ACE_Handler Class (2/6)

```
class AIO_Input_Handler
  : public   ACE_Service_Handler // Inherits from ACE_Handler
{
```

Inherit open() activation hook from ACE_Service_Handler

```
public:
  AIO_Input_Handler (AIO_CLD_Acceptor *acc = 0)
    : acceptor_ (acc), mblk_ (0) {}

  virtual ~AIO_Input_Handler ();
```

Called by ACE_Asynch_Acceptor when a client connects

```
  virtual  void open (ACE_HANDLE new_handle,
                      ACE_Message_Block &message_block);
protected:
  enum { LOG_HEADER_SIZE = 8 };   // Length of CDR header.
  AIO_CLD_Acceptor *acceptor_;    // Our creator.
  ACE_Message_Block *mblk_;       // Buffer to receive log
record.
  ACE_Asynch_Read_Stream reader_; // Asynchronous read() factory.
```

Handle async received logging records from client applications

```
  virtual   void handle_read_stream
    (const ACE_Asynch_Read_Stream::Result &result);
```

Using the ACE_Handler Class (3/6)

```
void AIO_Input_Handler::open
    (ACE_HANDLE new_handle, ACE_Message_Block &) {
  reader_.open (*this, new_handle, 0, proactor ());
  ACE_NEW_NORETURN
    (mblk_, ACE_Message_Block (ACE_DEFAULT_CDR_BUFSIZE));
  ACE_CDR::mb_align (mblk_);
```

Initiate asynchronous read of log record header to bootstrap the daemon

```
  reader_.read (*mblk_, LOG_HEADER_SIZE);
}
```

Hook method called back when an async read completes

```
void AIO_Input_Handler::handle_read_stream
      (const ACE_Asynch_Read_Stream::Result &result) {
  if (!result.success () || result.bytes_transferred () == 0)
    delete this;
  else if (result.bytes_transferred() <
result.bytes_to_read())
```

Initiate another asynchronous read to get the rest of log record header

```
    reader_.read (*mblk_, result.bytes_to_read () -
                          result.bytes_transferred ());
  else if (mblk_->length () == LOG_HEADER_SIZE) {
    ACE_InputCDR cdr (mblk_);
```

Using the ACE_Handler Class (4/6)

```
    ACE_CDR::Boolean byte_order;
    cdr >> ACE_InputCDR::to_boolean (byte_order);
    cdr.reset_byte_order (byte_order);

    ACE_CDR::ULong length;
    cdr >> length;

    mblk_->size (length + LOG_HEADER_SIZE);
    reader_.read (*mblk_, length);
  }
  else {
    if (OUTPUT_HANDLER::instance ()->put (mblk_) == -1)
      mblk_->release ();

    ACE_NEW_NORETURN
      (mblk_, ACE_Message_Block
(ACE_DEFAULT_CDR_BUFSIZE));
    ACE_CDR::mb_align (mblk_);
    reader_.read (*mblk_, LOG_HEADER_SIZE);
  }
}
```

Initiate asynchronous read to obtain rest of log record

Enqueue log record for output processing

Initiate new async read to rebootstrap the input process

Using the ACE_Handler Class (5/6)

Called when an async write to server logging daemon completes

```
void AIO_Output_Handler::handle_write_stream
          (const ACE_Asynch_Write_Stream::Result &result)
{
  ACE_Message_Block &mblk = result.message_block ();
  if (!result.success ()) {
    mblk.rd_ptr (mblk.base ());
    ungetq (&mblk);
  }
  else {
    can_write_ = handle () == result.handle ();
    if (mblk.length () == 0) {
      mblk.release ();
      if (can_write_) start_write ();
    }
    else if (can_write_) start_write (&mblk);
    else { mblk.rd_ptr (mblk.base ()); ungetq (&mblk);
}
  }
}
```

If we can write another log record to the server logging daemon, go ahead & initiate it asynchronously

Using the ACE_Handler Class (6/6)

This method is called back by the Proactor when the connection to the server logging daemon fails

```
void AIO_Output_Handler::handle_read_stream
       (const ACE_Asynch_Read_Stream::Result &result)
{
  result.message_block ().release ();
  writer_.cancel ();
  ACE_OS::closesocket (result.handle ());
  handle (ACE_INVALID_HANDLE);
  can_write_ = 0;
  CLD_CONNECTOR::instance ()->reconnect ();
}
```

Initiate reconnection

Sidebar: Managing ACE_Message_Block Pointers

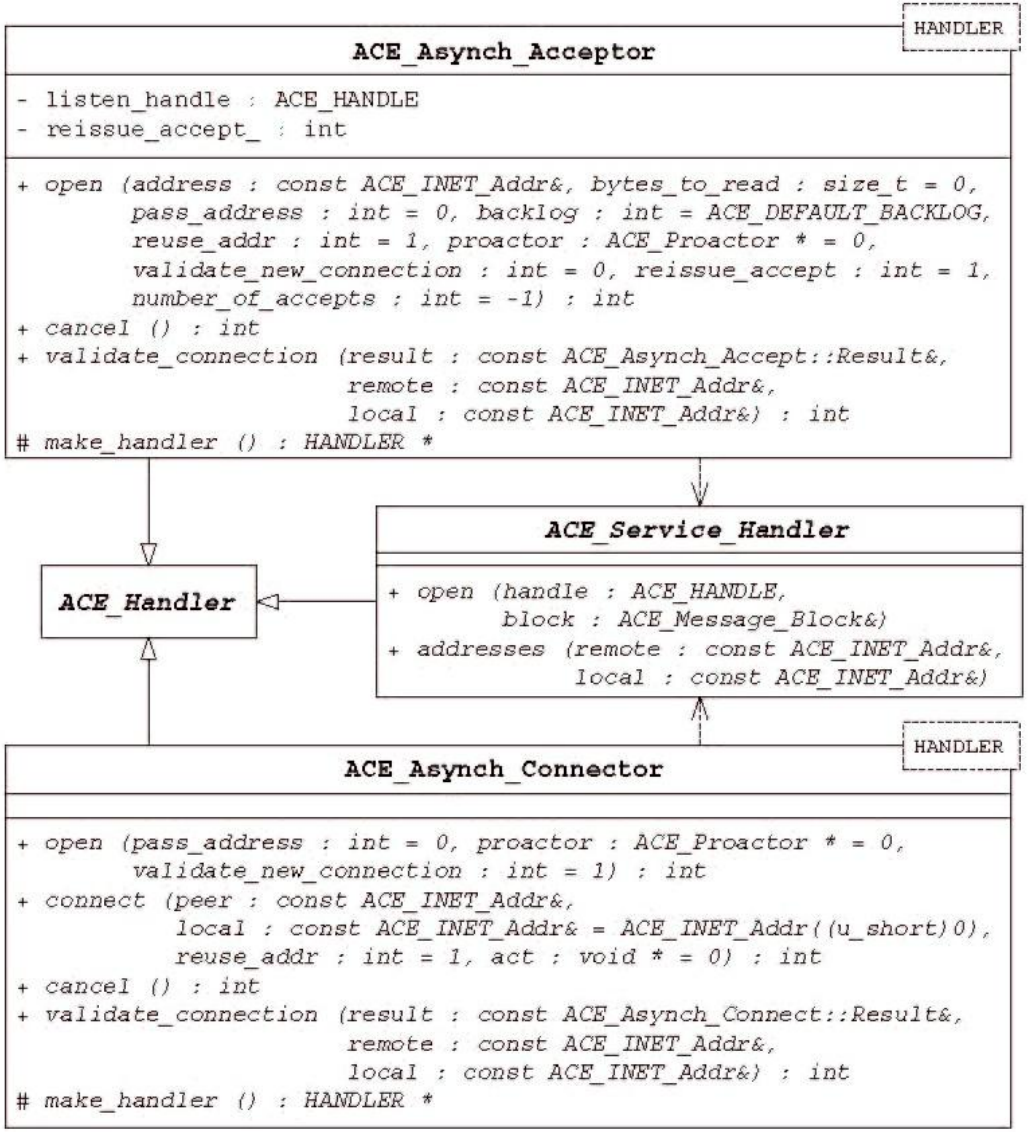
- When initiating an asynchronous **`read()`** or **`write()`**, the request must specify an **`ACE_Message_Block`** to either receive or supply the data
- The ACE Proactor framework's completion handling mechanism updates the **`ACE_Message_Block`** pointers to reflect the amount of data read or written as follows:
  - ***Read***
    - The initial read buffer pointer is the message's **`wr_ptr()`**
    - At completion, the **`wr_ptr`** is advanced by the number of bytes read
  - ***Write***
    - The initial write buffer pointer is the message's **`rd_ptr()`**
    - At completion, the **`rd_ptr`** is advanced by the number of bytes written
- It may seem counterintuitive to use the write pointer for reads & the read pointer for writes
- It may therefore help to consider that when reading data, it's being written into the message block
- Similarly, when writing data, it's being read from the message block
- Upon completion, the updated length of data in the **`ACE_Message_Block`** is larger for reads (because the write pointer has advanced) & smaller for writes (because the read pointer has advanced)

The Proactive Acceptor/Connector Classes

Class Capabilities

- **`ACE_Asynch_Acceptor`** is another implementation of the acceptor role in the Acceptor/Connector pattern
- This class provides the following capabilities:
  - It initiates asynchronous passive connection establishment
  - It acts as a factory, creating a new service handler for each accepted connection
  - It can cancel a previously initiated asynchronous **`accept()`** operation
  - It provides a hook method to obtain the peer's address when the new connection is established
  - It provides a hook method to validate the peer before initializing the new service handler

The Proactive Acceptor/Connector Classes APIs



Sidebar: ACE_Service_Handler vs. ACE_Svc_Handler

- The **ACE_Service_Handler** class plays a role analogous to that of the ACE Acceptor/Connector framework's **ACE_Svc_Handler** class
- Although the ACE Proactor framework could have reused **ACE_Svc_Handler** as the target of **ACE_Asynch_Acceptor** & **ACE_Asynch_Connector**, a separate class was chosen for the following reasons:
  - Networked applications that use proactive connection establishment also often use proactive I/O
    - The target of asynchronous connection completions should therefore be a class that can participate seamlessly with the rest of the ACE Proactor framework
  - **ACE_Svc_Handler** encapsulates an IPC object, but since the ACE Proactor framework uses I/O handles internally
    - Thus, the additional IPC object could be confusing
  - **ACE_Svc_Handler** is designed for use with the ACE Reactor framework since it descends from **ACE_Event_Handler**
    - ACE therefore maintains separation in its frameworks to avoid unnecessary coupling & faciliate ACE toolkit subsets

Using Proactive Acceptor/Connector Classes (1/4)

- This example illustrates how the classes in the proactive implementation are separated into separate input & output roles

```
class AIO_CLD_Acceptor
  : public ACE_Asynch_Acceptor<AIO_Input_Handler> {
```

Become an ACE_Asynch_Acceptor

```
public:
  void close (void); // Cancel accept & close all clients.

  // Remove handler from client set.
  void remove (AIO_Input_Handler *ih)
  { clients_.remove (ih); }

protected:
  virtual   AIO_Input_Handler *make_handler (void);
```

Service handler factory method

```
  // Set of all connected clients.
  ACE_Unbounded_Set<AIO_Input_Handler *> clients_;
};
```

Using Proactive Acceptor/Connector Classes (2/4)

```
AIO_Input_Handler *AIO_CLD_Acceptor::make_handler (void)
{
  AIO_Input_Handler *ih;
  ACE_NEW_RETURN (ih, AIO_Input_Handler (this), 0);
  if (clients_.insert (ih) == -1) { delete ih; return 0;
}
  return ih;
}

AIO_Input_Handler::~AIO_Input_Handler () {
  reader_.cancel ();
  ACE_OS::closesocket (handle ());
  if (mblk_ != 0) mblk_->release ();
  mblk_ = 0;
  acceptor_->remove (this);
}

void AIO_CLD_Acceptor::close (void) {
  ACE_Unbounded_Set_Iterator<AIO_Input_Handler *>
    iter (clients_.begin ());
  AIO_Input_Handler **ih;
  while (iter.next (ih)) delete *ih;
}
```

Keep track of client input handlers

Iterator pattern used to cleanup input handlers

Using Proactive Acceptor/Connector Classes (3/4)

```
class AIO_CLD_Connector
  : public   ACE_Asynch_Connector<AIO_Output_Handler> {
```

Become an ACE_Asynch_Connector

```
public:
  enum { INITIAL_RETRY_DELAY = 3, MAX_RETRY_DELAY = 60 };

  // Constructor.
  AIO_CLD_Connector ()
    : retry_delay_ (INITIAL_RETRY_DELAY), ssl_ctx_ (0), ssl_
(0)
  { open (); }
```

Hook method to detect failure & validate peer before opening handler

```
  virtual int validate_connection
    (const ACE_Asynch_Connect::Result &result,
     const ACE_INET_Addr &remote, const ACE_INET_Addr &local);
```

Using Proactive Acceptor/Connector Classes (4/4)

```
protected:
                  Hook method to create a new output handler
  virtual  AIO_Output_Handler *make_handler (void)
  { return OUTPUT_HANDLER::instance (); }

  // Address at which logging server listens for connections.
  ACE_INET_Addr remote_addr_;

  // Seconds to wait before trying the next connect
  int retry_delay_;

  // The SSL "context" data structure.
  SSL_CTX *ssl_ctx_;

  // The SSL data structure corresponding to authenticated
  // SSL connections.
  SSL *ssl_;
};

typedef ACE_Unmanaged_Singleton<AIO_CLD_Connector, ACE_Null_Mutex>
        CLD_CONNECTOR;
```

Sidebar: Emulating Async Connections on POSIX

- Windows has native capability for asynchronously connecting sockets
- In contrast, the POSIX.4 AIO facility was designed primarily for use with disk I/O, so it doesn't include any capability for asynchronous TCP/IP connection establishment
- To provide uniform capability across all asynchronous I/O-enabled platforms, ACE emulates asynchronous connection establishment where needed
- To emulate asynchronous connection establishment, active & passive connection requests are begun in nonblocking mode by the **`ACE_Asynch_Acceptor`** & **`ACE_Asynch_Connector`**
- If the connection doesn't complete immediately (which is always the case for passive connections), the socket handle is registered with an instance of **`ACE_Select_Reactor`** managed privately by the framework
- An ACE Proactor framework-spawned thread (unseen by the application) runs the private reactor's event loop
- When the connection request completes, the framework regains control via a reactor callback & posts the completion event
- The original application thread receives the completion event back in the **`ACE_Asynch_Acceptor`** or **`ACE_Asynch_Connector`** class, as appropriate

The ACE_Proactor Class (1/2)

Motivation

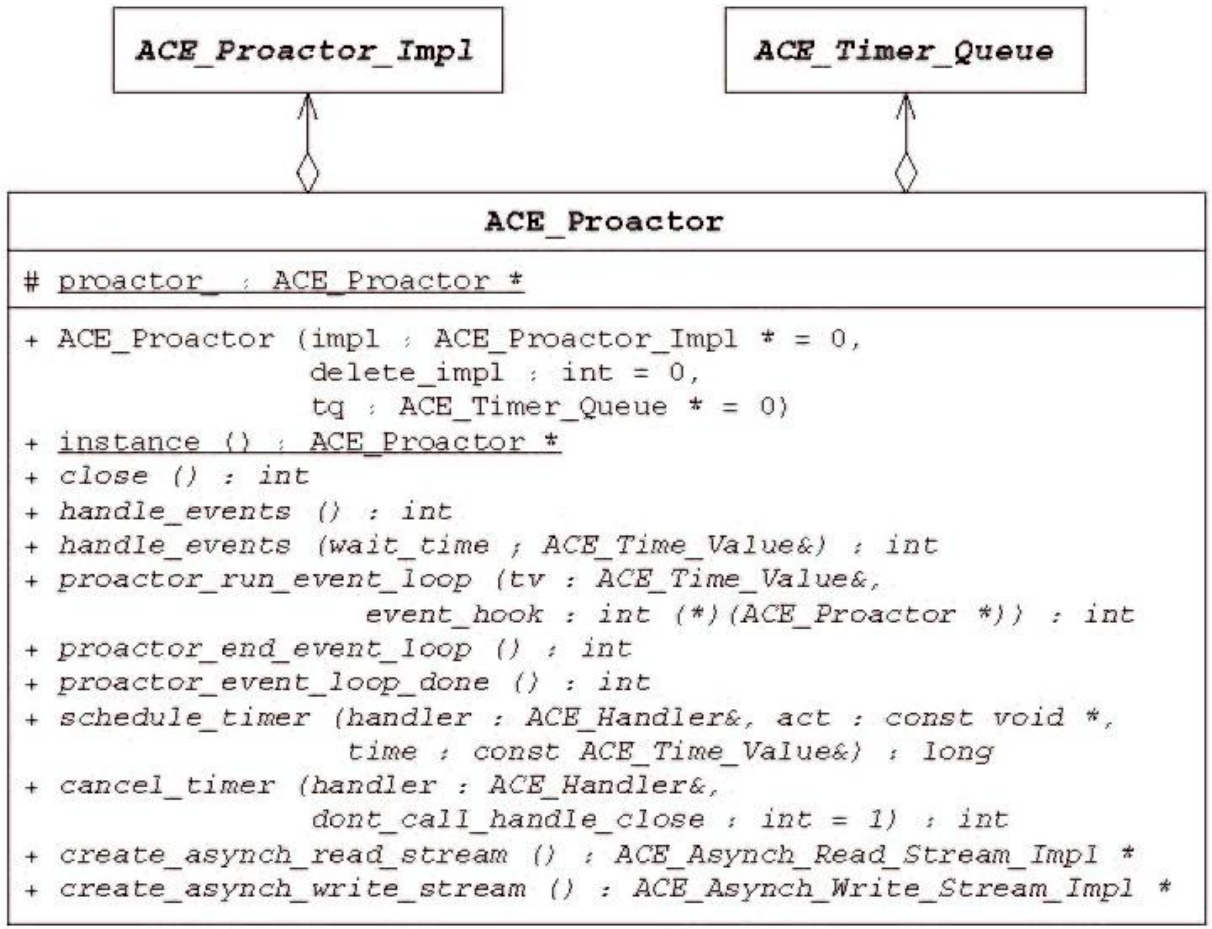
- Asynchronous I/O operations are handled in two steps: *initiation* & *completion*
- Since multiple steps & classes are involved, there must be a way to demultiplex the completion events & efficiently associate each completion event with the operation that completed & the completion handler that will process the result

The ACE_Proactor Class

Class Capabilities

- This class implements the Facade pattern to allow applications to access the various ACE Proactor framework features that provide the following capabilities:
  - Centralize event loop processing in a proactive application
  - Dispatch timer expirations to their associated **`ACE_Handle`** objects
  - Demultiplex completion events to completion handlers & dispatch hook methods on completion handlers

The ACE_Proactor Class API



Using the ACE_Proactor Class (1/7)

- We use the following `validate_connection()` hook method to insert application-defined behavior (e.g., SSL authentication) into `ACE_Asynch_Connector`'s connection completion handling

```
int AIO_CLD_Connector::validate_connection
      (const ACE_Asynch_Connect::Result &result,
       const ACE_INET_Addr &remote, const ACE_INET_Addr &)
 {
  remote_addr_ = remote;
  if (!result.success ()) {
    ACE_Time_Value delay (retry_delay_);
    retry_delay_ *= 2;
    if (retry_delay_ > MAX_RETRY_DELAY)
      retry_delay_ = MAX_RETRY_DELAY;
    proactor ()->schedule_timer (*this, 0, delay);
    return -1;
 }

  retry_delay_ = INITIAL_RETRY_DELAY;

  if (ssl_ctx_ == 0) {
    OpenSSL_add_ssl_algorithms ();
    ssl_ctx_ = SSL_CTX_new (SSLv3_client_method ());
    if (ssl_ctx_ == 0) return -1;
```

If the connection isn't established, use the Proactor's timer queueing mechanism to reinitiate it via expontential backoff

Using the ACE_Proactor Class (2/7)

```
    if (SSL_CTX_use_certificate_file (ssl_ctx_,
CLD_CERTIFICATE_FILENAME,
                                      SSL_FILETYPE_PEM) <= 0
        || SSL_CTX_use_PrivateKey_file (ssl_ctx_,
                                        CLD_KEY_FILENAME,
                                        SSL_FILETYPE_PEM) <= 0
        || !SSL_CTX_check_private_key (ssl_ctx_)) {
      SSL_CTX_free (ssl_ctx_);
      ssl_ctx_ = 0;
      return -1;
    }
    ssl_ = SSL_new (ssl_ctx_);
    if (ssl_ == 0) {
      SSL_CTX_free (ssl_ctx_); ssl_ctx_ = 0;
      return -1;
    }
  }
```

Using the ACE_Proactor Class (3/7)

```
  SSL_clear (ssl_);
  SSL_set_fd
   (ssl_, ACE_reinterpret_cast (int,
result.connect_handle()));

  SSL_set_verify (ssl_, SSL_VERIFY_PEER, 0);

  if (SSL_connect (ssl_) == -1
      || SSL_shutdown (ssl_) == -1) return -1;
  return 0;
}
```

Try to reinitiate a connection after the timer expires

```
void AIO_CLD_Connector::handle_time_out (const ACE_Time_Value &,
                                         const void *)
{ connect (remote_addr_); }
```

Using the ACE_Proactor Class (4/7)

```
class AIO_Client_Logging_Daemon
  : public ACE_Task<ACE_NULL_SYNCH> {
```

Become an ACE_Task to be configured dynamically, run concurrently, & provide a queue

```
protected:
  ACE_INET_Addr cld_addr_; // Our listener address.
  ACE_INET_Addr sld_addr_; // The logging server's address.

  // Factory that passively connects the
<AIO_Input_Handler>.
  AIO_CLD_Acceptor acceptor_;

public:
  virtual int init (int argc, ACE_TCHAR *argv[]);
  virtual int fini ();
  virtual int svc (void);
};
```

ACE Service Configurator framework hook methods

Using the ACE_Proactor Class (5/7)

Called back by Service Configurator framework to initialize the daemon when it's linked dynamically

```
int AIO_Client_Logging_Daemon::init
     (int argc, ACE_TCHAR *argv[]) {
  u_short cld_port = ACE_DEFAULT_SERVICE_PORT;
  u_short sld_port = ACE_DEFAULT_LOGGING_SERVER_PORT;
  ACE_TCHAR sld_host[MAXHOSTNAMELEN];
  ACE_OS::strcpy (sld_host, ACE_LOCALHOST);

  // Process options (omitted)

  if (cld_addr_.set (cld_port) == -1 ||
      sld_addr_.set (sld_port, sld_host) == -1)
    return -1;
  return activate ();
}
```

Become an active object

Using the ACE_Proactor Class (6/7)

Hook method dispatched in separate thread to run client logging daemon's proactor loop concurrently

```
int AIO_Client_Logging_Daemon::svc (void) {
  if (acceptor_.open (cld_addr_) == -1) return -1;
  if (CLD_CONNECTOR::instance ()->connect (sld_addr_) == 0)
    ACE_Proactor::instance ()->proactor_run_event_loop ();
  acceptor_.close ();
  CLD_CONNECTOR::close ();
  OUTPUT_HANDLER::close ();
  return 0;
}
```

Called by ACE Service Configurator framework to shut down the proactor

```
int AIO_Client_Logging_Daemon::fini () {
  ACE_Proactor::instance ()->proactor_end_event_loop ();
  wait ();
  return 0;
}
```

Barrier synchronization

Using the ACE_Proactor Class (7/7)

```
ACE_FACTORY_DEFINE (AIO_CLD,
AIO_Client_Logging_Daemon)
```

`svc.conf` **file for Proactive client logging daemon**

```
dynamic AIO_Client_Logging_Daemon Service_Object *
AIO_CLD:_make_AIO_Client_Logging_Daemon()
  "-p $CLIENT_LOGGING_DAEMON_PORT"
```

The main() function is the same as the one we showed for the ACE Service Configurator example!!!!

Sidebar: Integrating Proactive & Reactive Events on Windows

- The ACE Reactor & ACE Proactor event loops require different event detection & demultiplexing mechanisms that often execute in separate threads
- On Windows, however, ACE provides a way to integrate the two event loop mechanisms so they can both be driven by a single thread
- The **`ACE_Proactor`** Windows implementation uses an I/O completion port to detect completion events
- When one or more asynchronous operations complete, Windows signals the corresponding I/O completion port handle
- This handle can therefore be registered with an **`ACE_WFMO_Reactor`**, as follows:

```
ACE_Proactor::close_singleton ();
ACE_WIN32_Proactor *impl = new ACE_WIN32_Proactor (0, 1);
ACE_Proactor::instance (new ACE_Proactor (impl, 1), 1);
ACE_Reactor::instance ()->register_handler
  (impl, impl->get_handle ());
// ... Other registration & initiation code omitted.
ACE_Reactor::instance ()->run_reactor_event_loop ();
ACE_Reactor::instance ()->remove_handler
  (impl->get_handle (), ACE_Event_Handler::DONT_CALL);
```

Proactor POSIX Implementations

- The ACE Proactor implementations on POSIX systems present multiple mechanisms for initiating I/O operations & detecting their completions
  - Many UNIX AIO implementations are buggy, however…

ACE Proactor Variant	Description
ACE_POSIX_AIOCB_Proactor	This implementation maintains a parallel list of aiocb structures and Result objects. Each outstanding operation is represented by an entry in each list. The aio_suspend() function suspends the event loop until one or more asynchronous I/O operations complete.
ACE_POSIX_SIG_Proactor	This implementation is derived from ACE_POSIX_AIOCB_Proactor, but uses POSIX real-time signals to detect asynchronous I/O completion. The event loop uses the sigtimedwait() and sigwaitinfo() functions to pace the loop and retrieve information about completed operations. Each asynchyronous I/O operation started using this proactor has a unique value associated with its aiocb that's communicated with the signal noting its completion. This design makes it easy to locate the aiocb and its parallel Result object, and dispatch the correct completion handler.
ACE_SUN_Proactor	This implementation is also based on ACE_POSIX_AIOCB_Proactor, but it uses the Sun-specific asynchronous I/O facility instead of the POSIX.4 AIO facility. This implementation works much like ACE_POSIX_AIOCB_Proactor, but uses the Sun-specific aiowait() function to detect I/O completions.

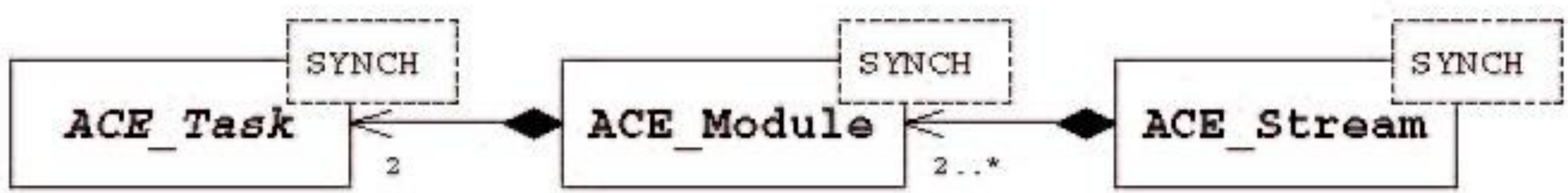
- Sun's Solaris OS offers its own proprietary version of asynchronous I/O
- On Solaris 2.6 & above, the performance of the Sun-specific asynchronous I/O functions is significantly higher than that of Solaris's POSIX.4 AIO

The ACE Streams Framework

- The ACE Streams framework is based on the Pipes & Filters pattern
- This framework simplifies the development of layered/modular applications that can communicate via bidirectional processing modules

ACE Class	Description
ACE_Task	A cohesive unit of application-defined functionality that uses messages to communicate requests, responses, data, and control information and can queue and process messages sequentially or concurrently.
ACE_Module	A distinct bidirectional processing layer in an application that contains two ACE_Task objects—one for "reading" and one for "writing"
ACE_Stream	Contains an ordered list of interconnected ACE_Module objects that can be used to configure and execute layered application-defined services

- The most important relationships between classes in the ACE Streams framework are shown below

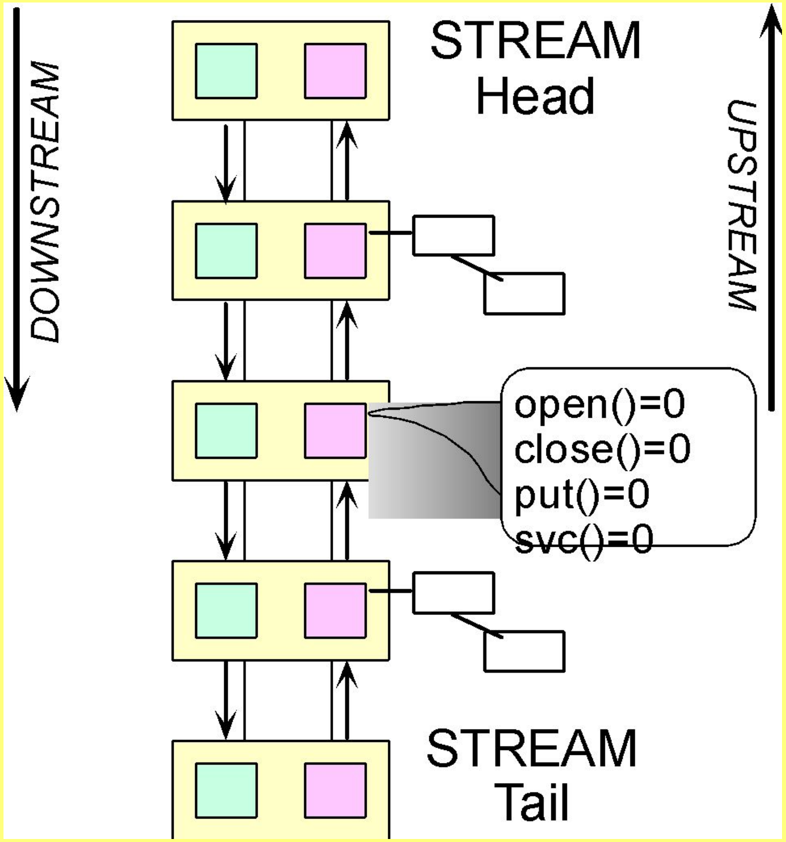


The Pipes & Filters Pattern

- The Pipes & Filters architectural pattern (POSA1) is a common way of organizing layered/modular applications
- This pattern defines an architecture for processing a stream of data in which each processing step is encapsulated in some type of filter component
- Data is passed between adjacent filters via a communication mechanism, which can range from IPC channels connecting local or remote processes to simple pointers that reference objects within the same process
  - Each filter can add, modify, or remove data before passing it along to the next filter
  - Filters are often stateless, in which case data passing through the filter are transformed & passed along to the next filter without being stored
- Common examples of the Pipes & Filters pattern include
  - The UNIX pipe IPC mechanism used by UNIX shells to create unidirectional pipelines
  - System V STREAMs, which provides a framework for integrating bidirectional protocols into the UNIX kernel

Sidebar: ACE Streams Relationship to SVR4 STREAMS

- The class names & design of the ACE Streams framework correspond to similar componentry in System V STREAMS
- The techniques used to support extensibility & concurrency in these two frameworks differ significantly, however
  - e.g., application-defined functionality is added in System V STREAMS via tables of pointers to C functions, whereas in the ACE Streams framework it's added by subclassing from **`ACE_Task`**, which provides greater type safety & extensibility
- The ACE Streams framework also uses the ACE Task framework to enhance the coroutine-based concurrency mechanisms used in System V STREAMS



- These ACE enhancements enable more effective use of multiple CPUs on shared memory multiprocessing platforms by reducing the likelihood of deadlock & simplifying flow control between **`ACE_Task`** active objects in an **`ACE_Stream`**

The ACE_Module Class (1/2)

Motivation

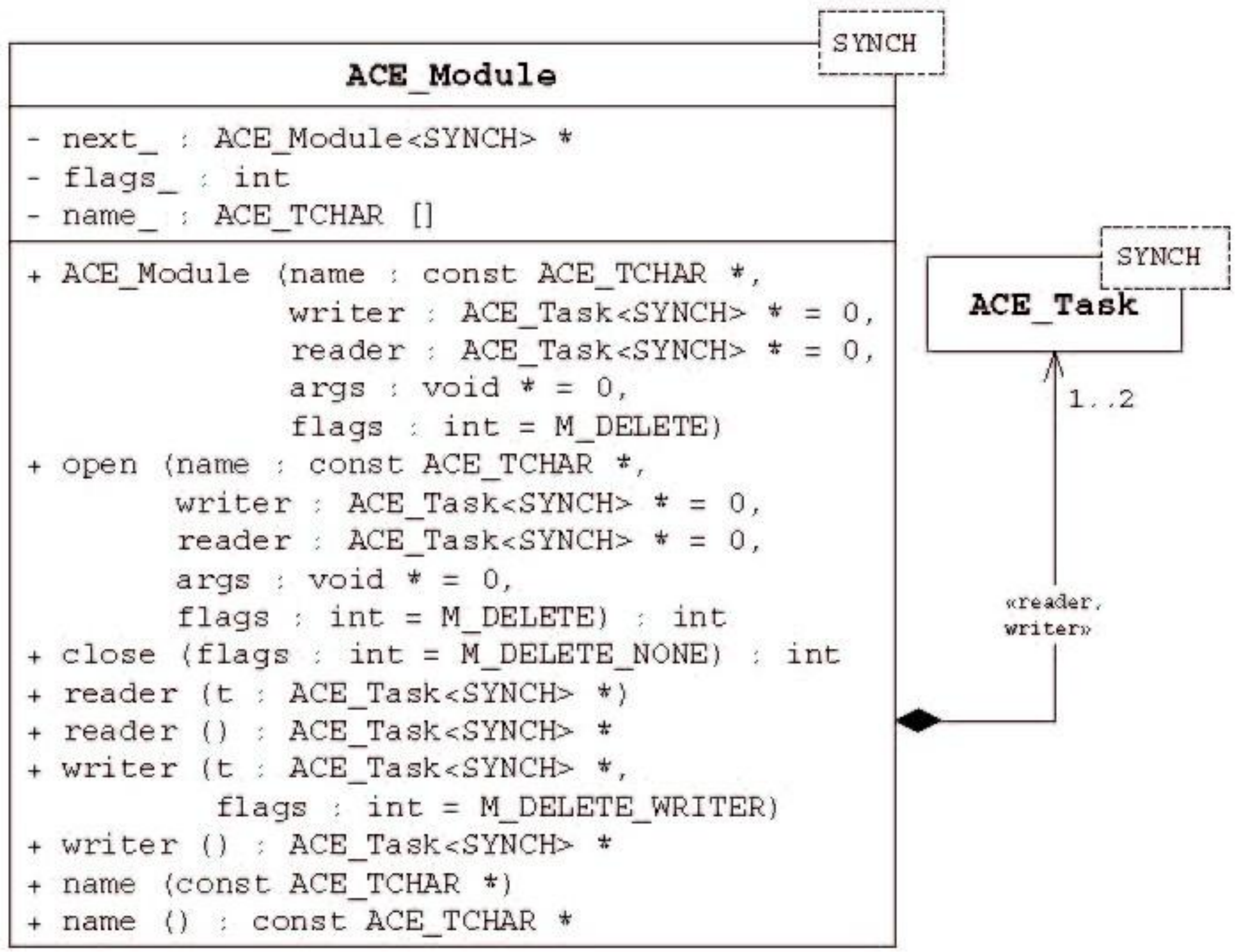
- Many networked applications can be modeled as an ordered series of processing layers that are related hierarchically & that exchange messages between adjacent layers
- Each layer can handle a self-contained portion (such as input or output, event analysis, event filtering, or service processing) of a service or networked application

The ACE_Module Class (2/2)

Class Capabilities

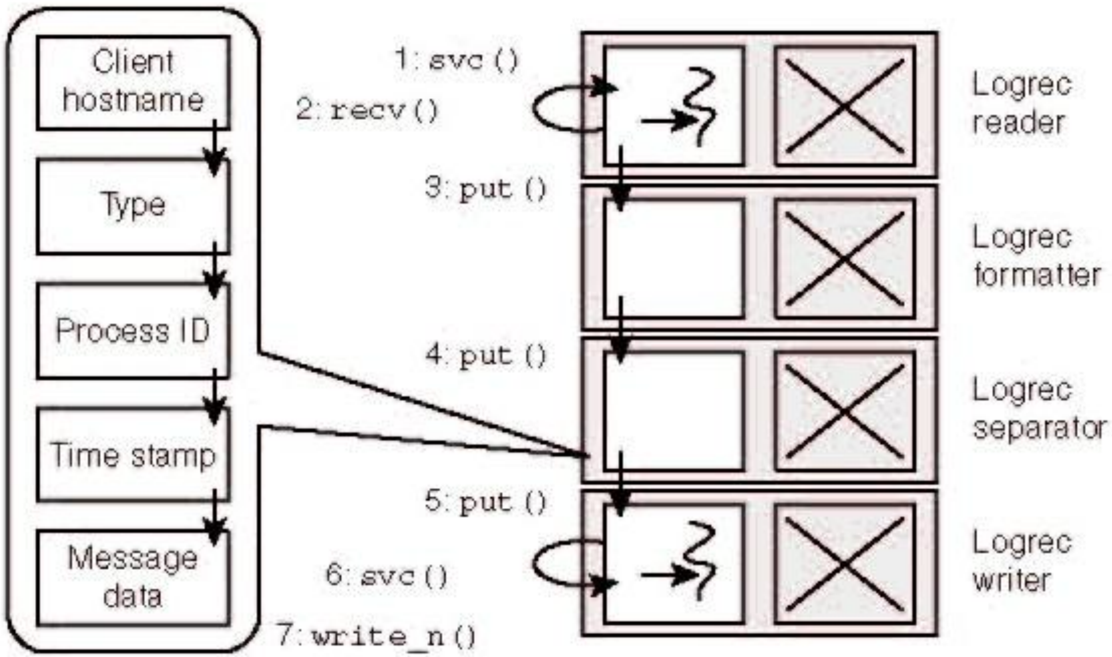
- This class defines a distinct layer of application-defined functionality that provides the following capabilities:
  - Each **`ACE_Module`** is a bidirectional application-defined processing layer containing a pair of reader & writer tasks that derive from **`ACE_Task`**
  - The reader & writer **`ACE_Task`** objects contained in an **`ACE_Module`** collaborate with adjacent **`ACE_Task`** objects by passing messages
  - The objects composed into an **`ACE_Module`** can be varied & replaced

The ACE_Module Class API



Using the ACE_Module Class (1/15)

- Most fields in a log record are stored in a CDR-encoded binary format, which is concise but not easily understood by humans
- This example develops a program called **`display_logfile`** that reads log records stored by our logging servers, formats the information, & prints it in a human-readable format



- **Logrec Reader** converts the log records in a logfile into a canonical composite message block format that's processed by other modules in an **`ACE_Stream`**
- **Logrec Formatter** determines how the fields in the log record will be formatted, for example by converting them from binary to ASCII
- **Logrec Separator** inserts message blocks containing a separator string between the existing message blocks in a composite log record message
- **Logrec Writer** prints formatted log record messages to the standard output, where they can be redirected to a file, printer, or console

Using the ACE_Module Class (2/15)

```
template <class TASK>
class Logrec_Module : public ACE_Module<ACE_MT_SYNCH> {
public:
  Logrec_Module (const ACE_TCHAR *name)
    : ACE_Module<ACE_MT_SYNCH>
                    (name,
                     &task_, // Initialize writer-side task.
                     0,      // Ignore reader-side task.
                     0,
                     ACE_Module<ACE_MT_SYNCH>::M_DELETE_READER)
{}
private:
  TASK task_;
};

#define LOGREC_MODULE(NAME) \
  typedef Logrec_Module<NAME> NAME##_Module
```

Using the ACE_Module Class (3/15)

```
class Logrec_Reader : public ACE_Task<ACE_MT_SYNCH> {
private:
  ACE_TString filename_; // Name of logfile.
  ACE_FILE_IO logfile_;  // File containing log records.

public:
  enum {MB_CLIENT = ACE_Message_Block::MB_USER,
        MB_TYPE, MB_PID, MB_TIME, MB_TEXT};

  Logrec_Reader (const ACE_TString &file): filename_ (file)
{}
  // ... Other methods shown below ...
};

 virtual int open (void *) {
    ACE_FILE_Addr name (filename_.c_str ());
    ACE_FILE_Connector con;
    if (con.connect (logfile_, name) == -1) return -1;
    return activate ();
  }
```

Using the ACE_Module Class (4/15)

```
virtual int svc () {
  const size_t FILE_READ_SIZE = 8 * 1024;
  ACE_Message_Block mblk (FILE_READ_SIZE);

  for (;; mblk.crunch ()) {
    ssize_t bytes_read = logfile_.recv (mblk.wr_ptr (),
                                        mblk.space ());
    if (bytes_read <= 0) break;
    mblk.wr_ptr (ACE_static_cast (size_t, bytes_read));
    for (;;) {
      size_t name_len = ACE_OS_String::strnlen
                          (mblk.rd_ptr (), mblk.length
());
      if (name_len == mblk.length ()) break;

      char *name_p = mblk.rd_ptr ();
      ACE_Message_Block *rec = 0, *head = 0, *temp = 0;
      ACE_NEW_RETURN
        (head, ACE_Message_Block (name_len, MB_CLIENT),
0);
      head->copy (name_p, name_len);
      mblk.rd_ptr (name_len + 1);   // Skip nul also
```

Using the ACE_Module Class (5/15)

```
      size_t need = mblk.length () +
ACE_CDR::MAX_ALIGNMENT;
      ACE_NEW_RETURN (rec, ACE_Message_Block (need), 0);
      ACE_CDR::mb_align (rec);
      rec->copy (mblk.rd_ptr (), mblk.length ());

      ACE_InputCDR cdr (rec); rec->release ();
      ACE_CDR::Boolean byte_order;
      if (!cdr.read_boolean (byte_order)) {
        head->release (); mblk.rd_ptr (name_p); break;
      }
      cdr.reset_byte_order (byte_order);

      ACE_CDR::ULong length;
      if (!cdr.read_ulong (length)) {
        head->release (); mblk.rd_ptr (name_p); break;
      }
      if (length > cdr.length ()) {
        head->release (); mblk.rd_ptr (name_p); break;
      }
      ACE_NEW_RETURN
        (temp, ACE_Message_Block (length, MB_TEXT), 0);
```

Using the ACE_Module Class (6/15)

```
        ACE_NEW_RETURN
          (temp,
           ACE_Message_Block (2 * sizeof (ACE_CDR::Long),
                              MB_TIME, temp), 0);
        ACE_NEW_RETURN
          (temp,
           ACE_Message_Block (sizeof (ACE_CDR::Long),
                              MB_PID, temp), 0);
        ACE_NEW_RETURN
          (temp,
           ACE_Message_Block (sizeof (ACE_CDR::Long),
                              MB_TYPE, temp), 0);
        head->cont (temp);
        // Extract the type...
        ACE_CDR::Long *lp = ACE_reinterpret_cast
                         (ACE_CDR::Long *, temp->wr_ptr
());
        cdr >> *lp;
        temp->wr_ptr (sizeof (ACE_CDR::Long));
        temp = temp->cont ();
```

Using the ACE_Module Class (7/15)

```
        // Extract the PID...
        lp = ACE_reinterpret_cast
            (ACE_CDR::Long *, temp->wr_ptr ());
        cdr >> *lp;
        temp->wr_ptr (sizeof (ACE_CDR::Long));
        temp = temp->cont ();
        // Extract the timestamp...
        lp = ACE_reinterpret_cast
          (ACE_CDR::Long *, temp->wr_ptr ());
        cdr >> *lp; ++lp; cdr >> *lp;
        temp->wr_ptr (2 * sizeof (ACE_CDR::Long));
        temp = temp->cont ();
        // Extract the text length, then the text
message
        ACE_CDR::ULong text_len;
        cdr >> text_len;
        cdr.read_char_array (temp->wr_ptr (), text_len);
        temp->wr_ptr (text_len);
```

Using the ACE_Module Class (8/15)

```
        if (put_next (head) == -1) break;
        mblk.rd_ptr (mblk.length () - cdr.length ());
      }
    }

    ACE_Message_Block *stop = 0;
    ACE_NEW_RETURN
      (stop,
       ACE_Message_Block (0, ACE_Message_Block::MB_STOP),
0);
    put_next (stop);
    return 0;
  }
```

Using the ACE_Module Class (9/15)

```
class Logrec_Reader_Module : public ACE_Module<ACE_MT_SYNCH> {
public:
  Logrec_Reader_Module (const ACE_TString &filename)
    : ACE_Module<ACE_MT_SYNCH>
                    (ACE_TEXT ("Logrec Reader"),
                     &task_, // Initialize writer-side.
                     0,      // Ignore reader-side.
                     0,

ACE_Module<ACE_MT_SYNCH>::M_DELETE_READER),
      task_ (filename) {}

private:
  // Converts the logfile into chains of message blocks.
  Logrec_Reader task_;
};
```

Using the ACE_Module Class (10/15)

```
class Logrec_Formatter : public ACE_Task<ACE_MT_SYNCH> {
private:
  typedef void (*FORMATTER[5])(ACE_Message_Block *);
  static FORMATTER format_; // Array of format static
methods.

public:
 virtual int put (ACE_Message_Block *mblk, ACE_Time_Value *)
{
    if (mblk->msg_type () == Logrec_Reader::MB_CLIENT)
      for (ACE_Message_Block *temp = mblk;
           temp != 0;
           temp = temp->cont ()) {
        int mb_type =
          temp->msg_type () - ACE_Message_Block::MB_USER;
        (*format_[mb_type])(temp);
      }
    return put_next (mblk);
  }

 static void format_client (ACE_Message_Block *) { return; }
```

Using the ACE_Module Class (11/15)

```
static void format_long (ACE_Message_Block *mblk) {
   ACE_CDR::Long type = * (ACE_CDR::Long *) mblk->rd_ptr ();
   mblk->size (11); // Max size in ASCII of 32-bit word.
   mblk->reset ();
   mblk->wr_ptr ((size_t) sprintf (mblk->wr_ptr (), "%d",
type));
  }

static void format_time (ACE_Message_Block *mblk) {
   ACE_CDR::Long secs = * (ACE_CDR::Long *)mblk->rd_ptr ();
   mblk->rd_ptr (sizeof (ACE_CDR::Long));
   ACE_CDR::Long usecs = * (ACE_CDR::Long *)mblk->rd_ptr ();
   char timestamp[26]; // Max size of ctime_r() string.
   time_t time_secs (secs);
   ACE_OS::ctime_r (&time_secs, timestamp, sizeof timestamp);
   mblk->size (26); // Max size of ctime_r() string.
   mblk->reset ();
```

Using the ACE_Module Class (12/15)

```
    timestamp[19] = '\0'; // NUL-terminate after the time.
    timestamp[24] = '\0'; // NUL-terminate after the date.
    size_t fmt_len (sprintf (mblk->wr_ptr (),
                             "%s.%03d %s",
                             timestamp + 4,
                             usecs / 1000,
                             timestamp + 20));
    mblk->wr_ptr (fmt_len);
  }

 static void format_string (ACE_Message_Block *) { return;
}
};

Logrec_Formatter::FORMATTER Logrec_Formatter::format_ = {
  format_client, format_long,
  format_long, format_time, format_string
};

LOGREC_MODULE (Logrec_Formatter);
```

Using the ACE_Module Class (13/15)

```
class Logrec_Separator : public ACE_Task<ACE_MT_SYNCH> {
private:
  ACE_Lock_Adapter<ACE_Thread_Mutex> lock_strategy_;
public:
    virtual int put (ACE_Message_Block *mblk,
                     ACE_Time_Value *) {
      if (mblk->msg_type () == Logrec_Reader::MB_CLIENT) {
        ACE_Message_Block *separator = 0;
        ACE_NEW_RETURN
          (separator,
           ACE_Message_Block (ACE_OS_String::strlen ("|") +
1,
                              ACE_Message_Block::MB_DATA,
                              0, 0, 0, &lock_strategy_), -1);
        separator->copy ("|");

        ACE_Message_Block *dup = 0;
```

Using the ACE_Module Class (14/15)

```
      for (ACE_Message_Block *temp = mblk; temp != 0; )
{
        dup = separator->duplicate ();
        dup->cont (temp->cont ());
        temp->cont (dup);
        temp = dup->cont ();
      }
      ACE_Message_Block *nl = 0;
      ACE_NEW_RETURN (nl, ACE_Message_Block (2), 0);
      nl->copy ("\n");
      dup->cont (nl);
      separator->release ();
    }

    return put_next (mblk);
  }

LOGREC_MODULE (Logrec_Separator);
```

Using the ACE_Module Class (15/15)

```
class Logrec_Writer : public ACE_Task<ACE_MT_SYNCH> {
public:
  // Initialization hook method.
  virtual int open (void *) { return activate (); }

 virtual int put (ACE_Message_Block *mblk, ACE_Time_Value *to)
  { return putq (mblk, to); }

 virtual int svc () {
    int stop = 0;
    for (ACE_Message_Block *mb; !stop && getq (mb) != -1; ) {
      if (mb->msg_type () == ACE_Message_Block::MB_STOP)
        stop = 1;
      else ACE::write_n (ACE_STDOUT, mb);

      put_next (mb);
    }
    return 0;
  }
};
LOGREC_MODULE (Logrec_Writer);
```

Sidebar: ACE_Task Relation to ACE Streams

- **`ACE_Task`** also contains methods that can be used with the ACE Streams framework

Method	Description
`module()`	Returns a pointer to the task's module if there is one, else 0
`next()`	Returns a pointer to the next task in a stream if there is one, else 0
`sibling()`	Returns a pointer to a task's sibling in a module
`put_next()`	Passes a message block to the adjacent task in a stream
`can_put()`	Returns 1 if a message block can be enqueued via `put_next()` without blocking due to intrastream flow control, else 0
`reply()`	Passes a message block to the sibling task's adjacent task of a stream, which enables a task to reverse the direction of a message in a stream

- An **`ACE_Task`** that's part of an **`ACE_Module`** can use **`put_next()`** to forward a message block to an adjacent module
  - This method follows the module's **`next()`** pointer to the right task, then calls its **`put()`** hook method, passing it the message block.
  - The **`put()`** method borrows the thread from the task that invoked **`put_next()`**
- If a task runs as an active object, its **`put()`** method can enqueue the message on the task's message queue & allow its **`svc()`** hook method to handle the message concurrently with respect to other processing in a stream

Sidebar: Serializing ACE_Message_Block Reference Counts

- If shallow copies of a message block are created and/or released in different threads there's a potential race condition on access to the reference count & shared data
  - Access to these data must therefore be serialized
- Since there are multiple message blocks involved, an external locking strategy is applied
  - A message block can therefore be associated with an instance of **`ACE_Lock_Adapter`**
- **`Logrec_Separator::put()`** accesses message blocks from multiple threads, so the **`ACE_Lock_Adapter`** is parameterized with an **`ACE_Thread_Mutex`**
  - This locking strategy serializes calls to the message block's **`duplicate()`** & **`release()`** methods to avoid race conditions when a message block is created & released concurrently by different threads
- Although **`Logrec_Separator::put()`** calls **`separator->release()`** before forwarding the message block to the next module, we take this precaution because a subsequent module inserted in the stream may process the blocks using multiple threads

The ACE_Stream Class (1/2)

Motivation

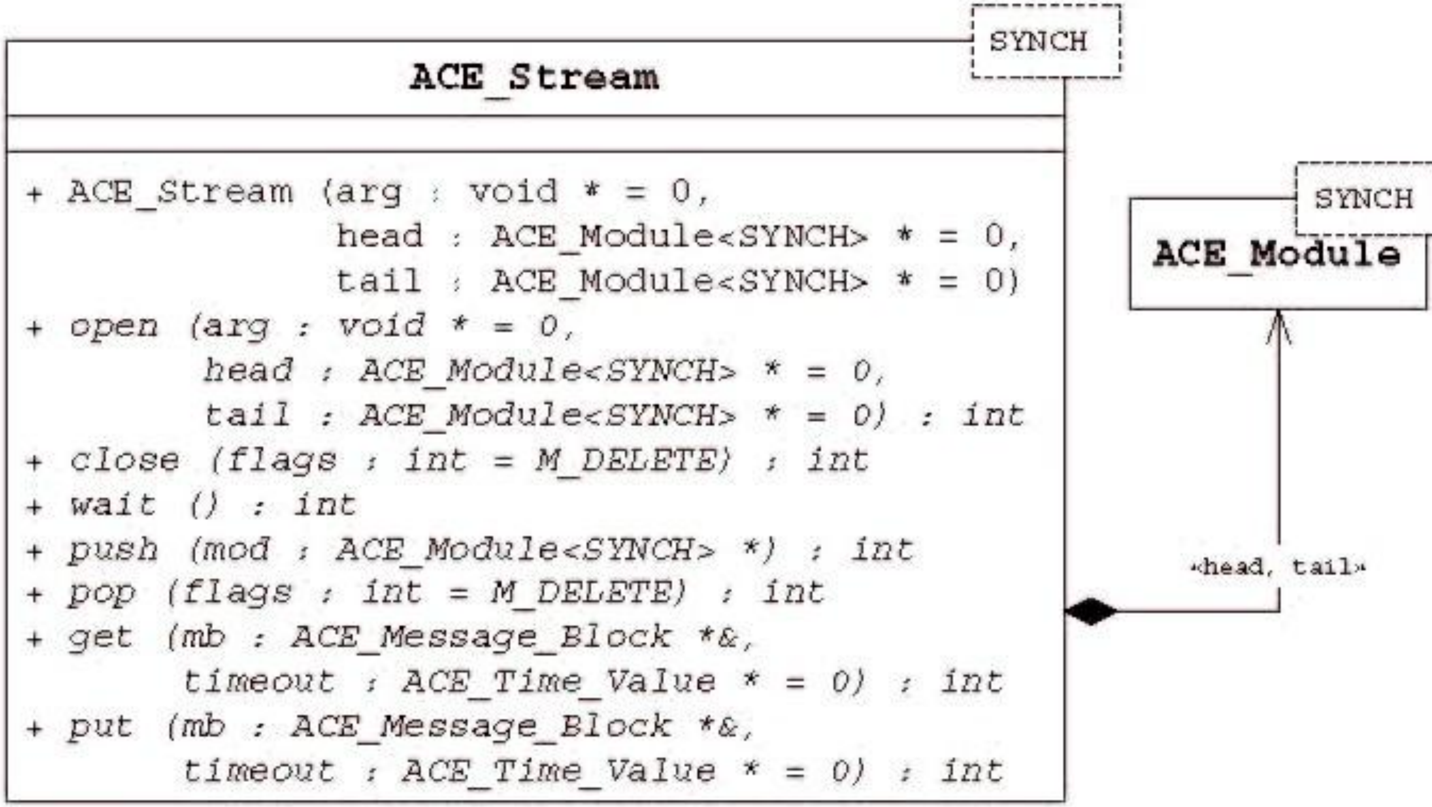
- **`ACE_Module`** does not provide a facility to connect or rearrange modules in a particular order
- **`ACE_Stream`** enables developers to build & manage a series of hierarchically related module layers as a single object

The ACE_Stream Class (2/2)

Class Capabilities

- **`ACE_Stream`** implements the Pipes & Filters pattern to enable developers to configure & execute hierarchically related services by customizing reusable application-independent framework classes to provide the following capabilities:
  - Provides methods to dynamically add, replace, & remove **`ACE_Module`** objects to form various stream configurations
  - Provides methods to send/receive messages to/from an **`ACE_Stream`**
  - Provides a mechanism to connect two **`ACE_Stream`** streams together
  - Provides a way to shut down all modules in a stream & wait for them all to stop

The ACE_Stream Class API



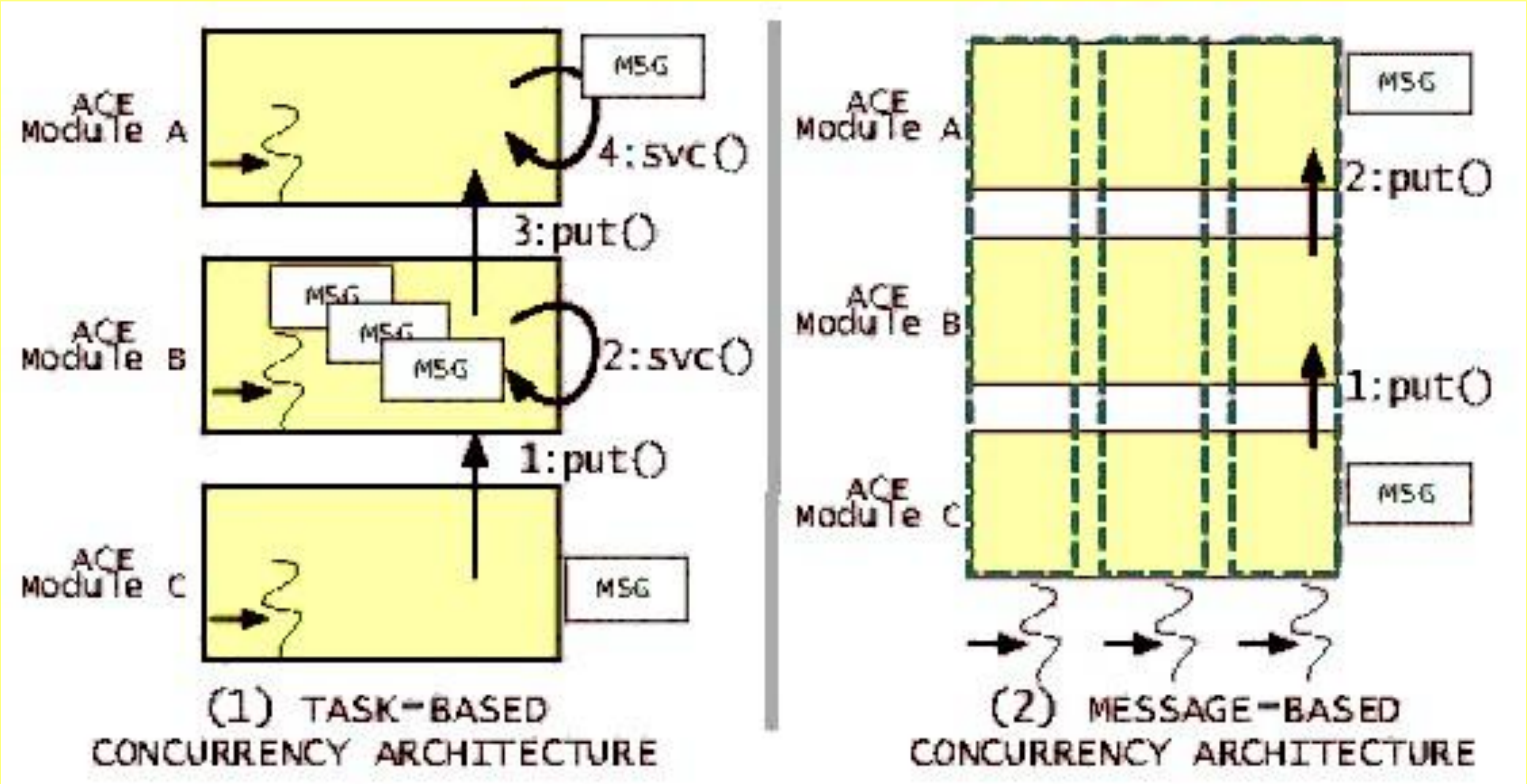
Using the ACE_Stream Class

- This example shows how to configure the **`display_logfile`** program with an **`ACE_Stream`** object that contains the modules

```
int ACE_TMAIN (int argc, ACE_TCHAR *argv[]) {
  if (argc != 2) ACE_ERROR_RETURN
                  ((LM_ERROR, "usage: %s logfile\n", argv[0]),
1);
  ACE_TString logfile (argv[1]);
  ACE_Stream<ACE_MT_SYNCH> stream;
  if (stream.push
      (new Logrec_Writer_Module (ACE_TEXT ("Writer"))) != -1
      && stream.push
      (new Logrec_Separator_Module (ACE_TEXT ("Separator"))) !=
-1
      && stream.push
      (new Logrec_Formatter_Module (ACE_TEXT ("Formatter"))) !=
-1
      && stream.push
      (new Logrec_Reader_Module (logfile)) != -1)
    return ACE_Thread_Manager::instance ()->wait () == 0 ? 0 : 1;
```

Sidebar: ACE Streams Framework Concurrency

- The ACE Streams framework supports two canonical concurrency architectures:
  - **Task-based**, where a **`put()`** method can borrow the thread of control from its caller to handle a message immediately, as shown by the message-based architecture
  - **Message-based**, where a **`put()`** method may enqueue a message & defer handling to its task's svc() method that executes concurrently in a separate thread, as shown by the task-based architecture



Additional Information

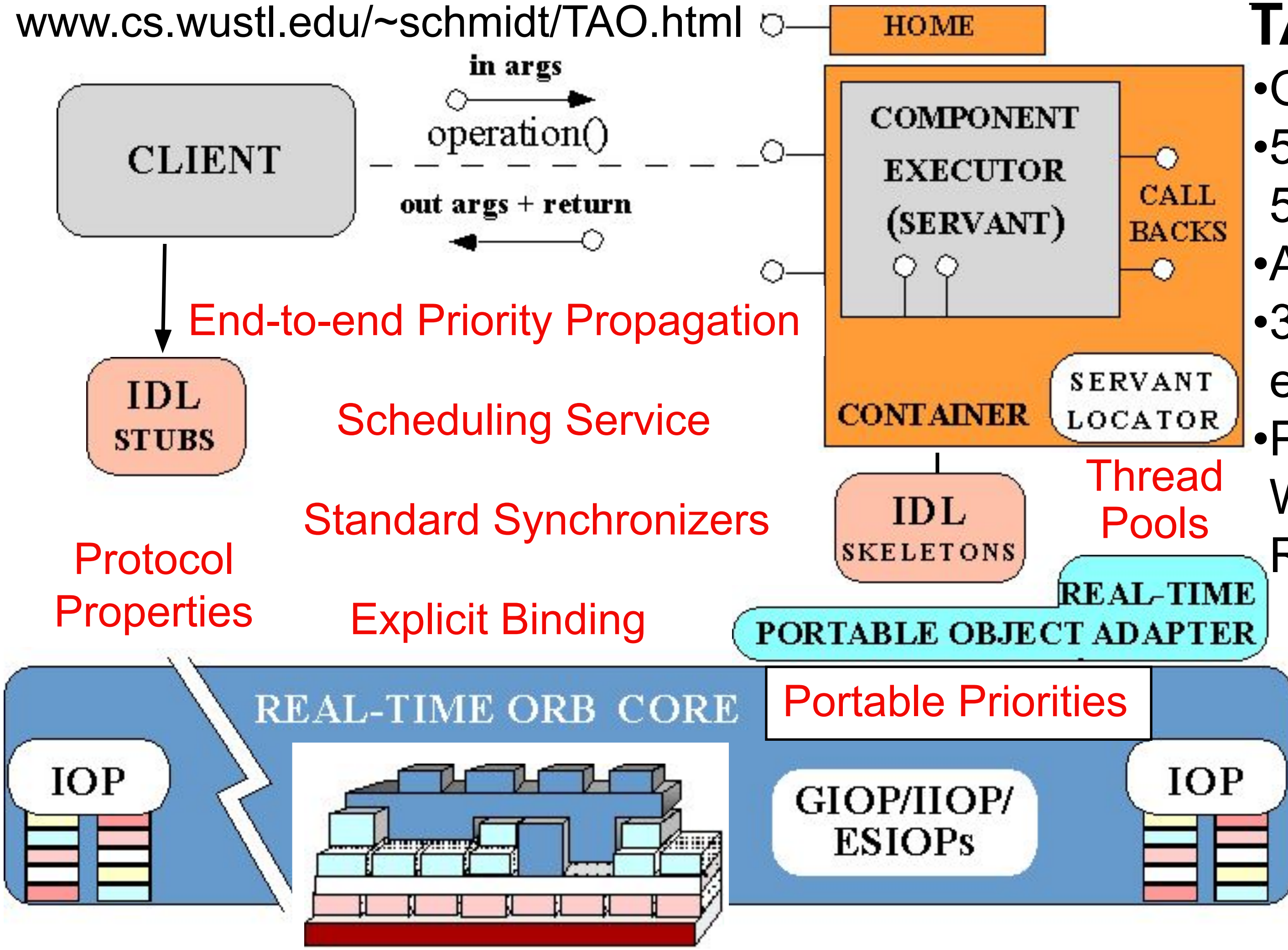
- Patterns & frameworks for concurrent & networked objects
  - `www.posa.uci.edu`
- ACE & TAO open-source middleware
  - `www.cs.wustl.edu/~schmidt/ACE.html`
  - `www.cs.wustl.edu/~schmidt/TAO.html`

- ACE research papers
  - `www.cs.wustl.edu/~schmidt/ACE-papers.html`
- Extended ACE & TAO tutorials
  - UCLA extension, July, 2005
  - `www.cs.wustl.edu/~schmidt/UCLA.html`
- ACE books
  - `www.cs.wustl.edu/~schmidt/ACE/`

Example of Applying ACE Patterns & Frameworks:

Real-time CORBA & The ACE ORB (TAO)

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



TAO Features

- Open-source
- 500+ classes & 500,000+ lines of C++
- ACE/patterns-based
- 30+ person-years of effort
- Ported to UNIX, Win32, MVS, & many RT & embedded OSs
  - *e.g.*, VxWorks, LynxOS, Chorus, QNX

- **Large open-source user community**
  - www.cs.wustl.edu/~schmidt/TAO-users.html

- **Commercially supported**
  - www.theaceorb.com
  - www.prismtechnologies.com