Patterns & Frameworks for Service Access & Communication : Part 1

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

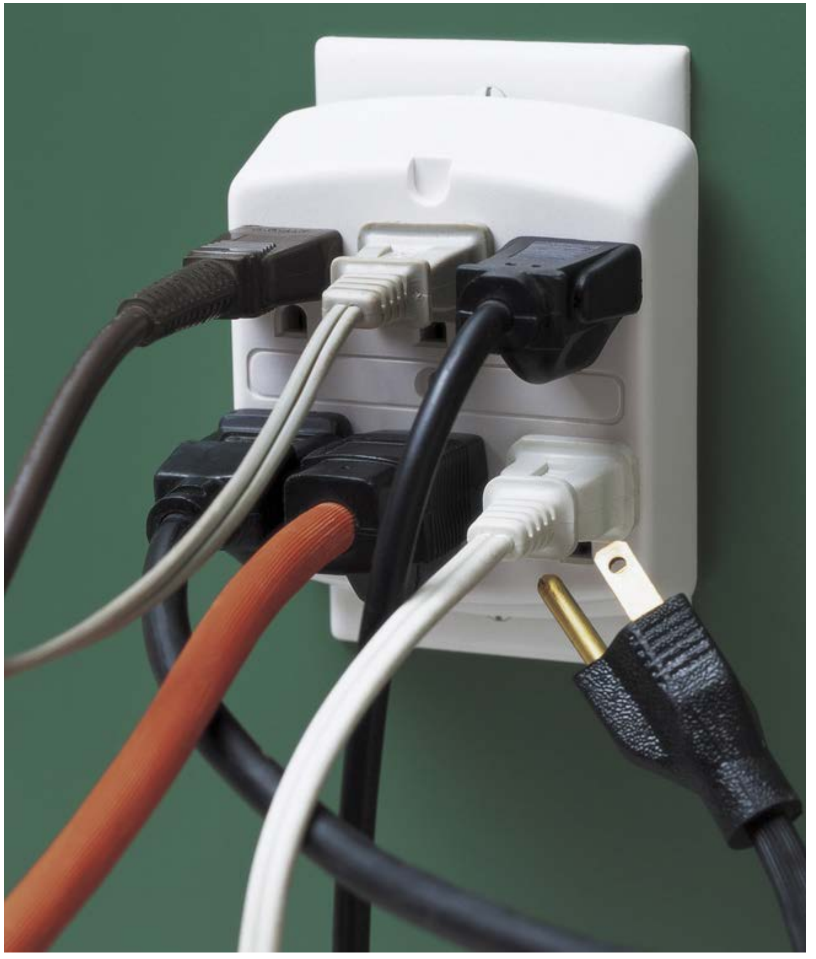
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

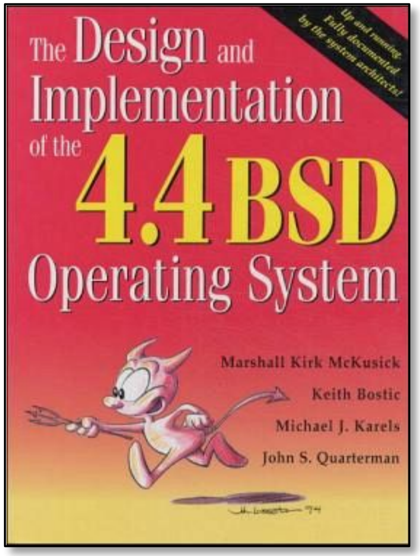
Topics Covered in this Part of the Module

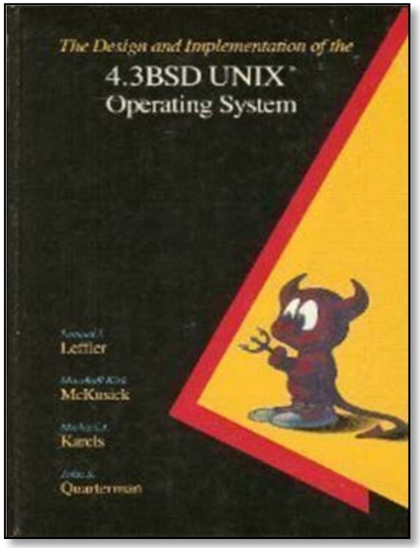
- Summarize the accidental complexities with Socket API



Recap of the Sockets API

- Originally developed in BSD UNIX as a C language interface to the TCP/IP protocol suite
  - Highly influential in hastening the “Internet Revolution”





4.2BSD and 4.3BSD as Examples of the UNIX System

JOHN S. QUARTERMAN, ABRAHAM SILBERSCHATZ, and JAMES L. PETERSON

Department of Computer Sciences, University of Texas, Austin, Texas 78712

This paper presents an in-depth examination of the 4.2 Berkeley Software Distribution, Virtual VAX-11 Version (4.2BSD), which is a version of the UNIX™ Time-Sharing System. There are notes throughout on 4.3BSD, the forthcoming system from the University of California at Berkeley. We trace the historical development of the UNIX system from its conception in 1969 until today, and describe the design principles that have guided this development. We then present the internal data structures and algorithms used by the kernel to support the user interface. In particular, we describe process management, memory management, the file system, the I/O system, and communications. These are treated in as much detail as the UNIX licenses will allow. We conclude with a brief description of the user interface and a set of bibliographic notes.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.4.0 [**Operating Systems**]: General—*UNIX*; D.4.7 [**Operating Systems**]: Organization and Design—*interactive systems*; K.2 [**History of Computing**]: Software—*UNIX*

General Terms: Algorithms, Design, Human Factors, Performance, Reliability, Security

Additional Key Words and Phrases: Flexibility, portability, simplicity

INTRODUCTION

This paper presents an in-depth examination of the *4.2BSD* operating system, the research *UNIX*[1] system developed for the Defense Advanced Research Projects Agency (DARPA) by the University of California at Berkeley. We have chosen 4.2BSD over UNIX System V (the UNIX system currently being licensed by AT&T) because concepts such as internetworking and demand paging are implemented in 4.2BSD but not in System V. Where 4.3BSD, the forthcoming system from Berkeley, differs functionally from 4.2BSD in the areas of interest, such differences are noted.

This paper is not a critique of the design and implementation of 4.2BSD or UNIX; it is an explanation. For comparisons of System V and 4.2BSD, see the literature, particularly the references given in Section 1.1, p. 380. Such comparisons are mostly beyond the scope of this paper.

The *VAX*[2] implementation is used because 4.2BSD was developed on the VAX,

[1] UNIX is a trademark of AT&T Bell Laboratories.

[2] VAX, PDP, TOPS-20, and VMS are trademarks of Digital Equipment Corporation.

Chapter 14 of *Operating Systems Concepts, Second Edition*, by J. L. Peterson and A. Silberschatz (© 1985 by Addison-Wesley, Reading, Massachusetts) and this article were both derived from an earlier common manuscript by J. S. Quarterman. Consequently they share some text. Common portions are reprinted with the permission of Addison-Wesley.

Author's present address: James L. Peterson, MCC, 9430 Research Blvd., Austin, Texas 78759.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1986 ACM 0360-0300/85/1200-0379 $00.75

Computing Surveys, Vol. 17, No. 4, December 1985

The Sockets network programming API is available on most operating systems

Recap of the Sockets API

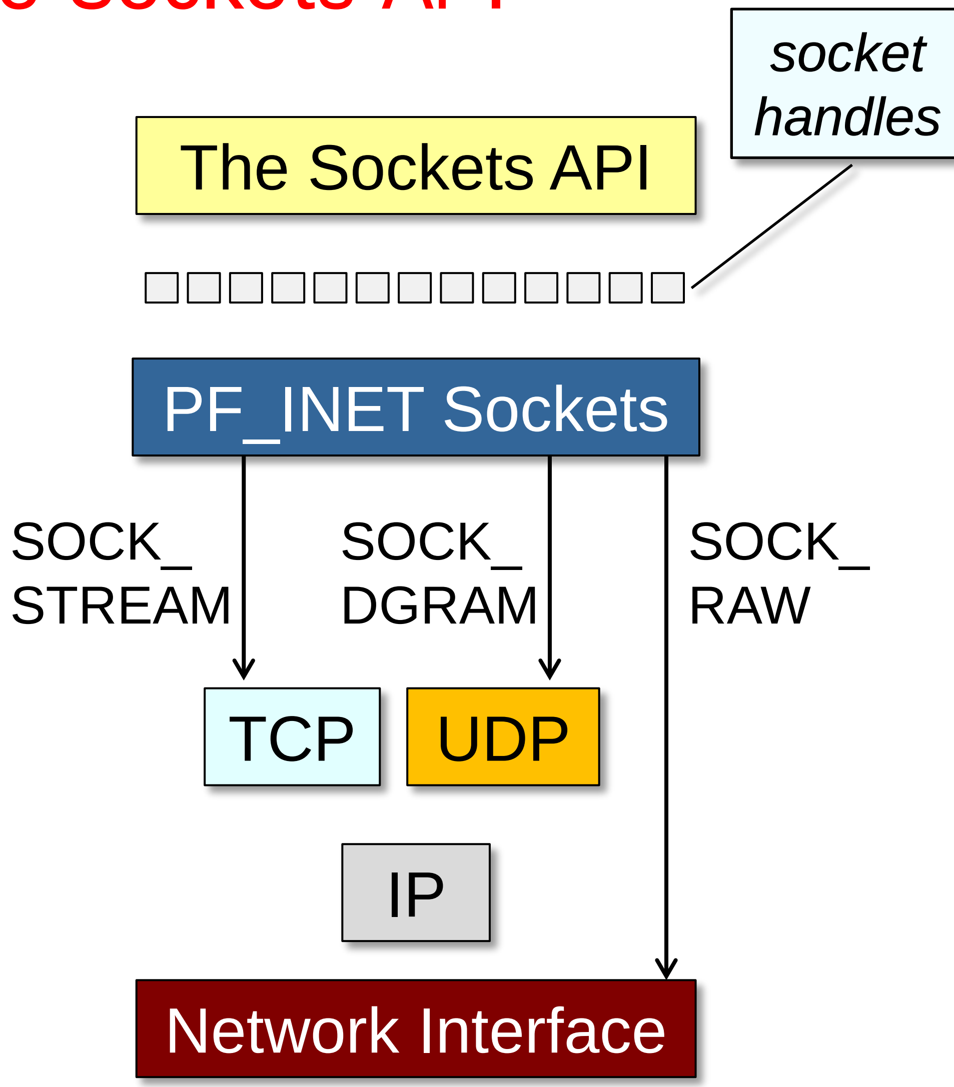
- Originally developed in BSD UNIX as a C language interface to the TCP/IP protocol suite
- The Socket API has approximately two dozen functions

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

See en.wikipedia.org/wiki/Berkeley_sockets for more info on the Sockets API

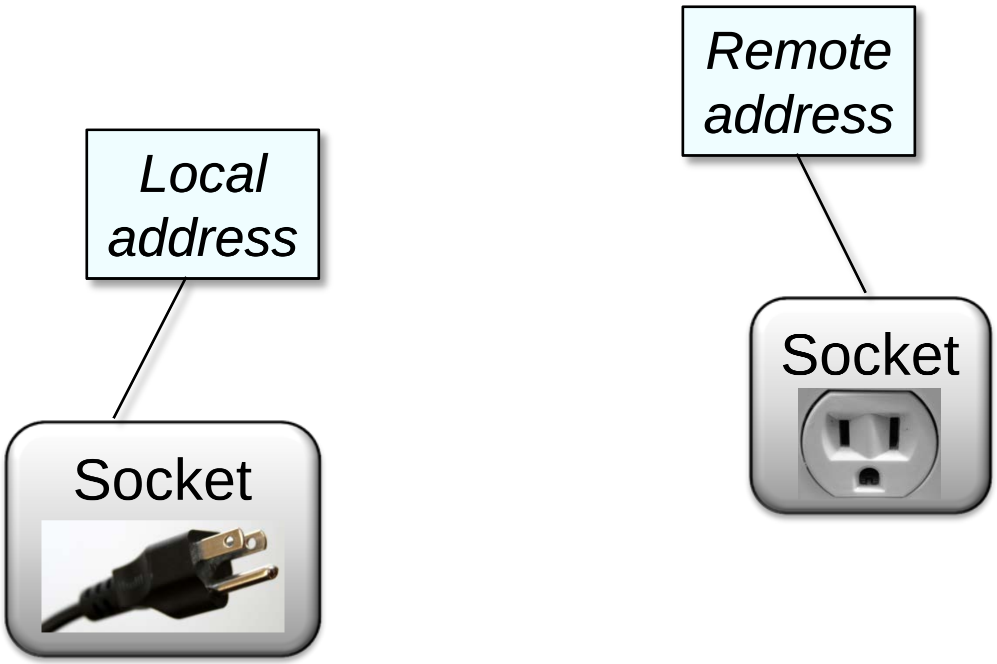
Recap of the Sockets API

- Originally developed in BSD UNIX as a C language interface to the TCP/IP protocol suite
- The Socket API has approximately two dozen functions
- A *socket* is a handle created by the OS that is associated with an end point of a communication channel



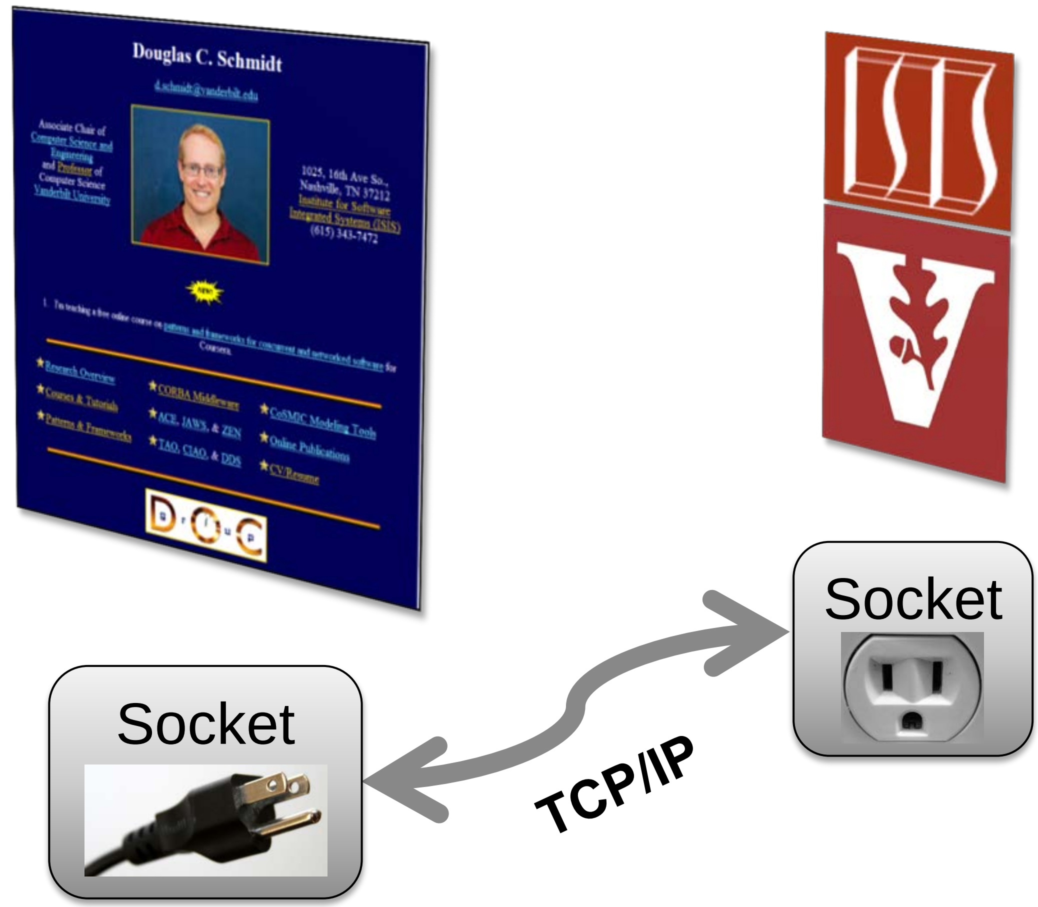
Recap of the Sockets API

- Originally developed in BSD UNIX as a C language interface to the TCP/IP protocol suite
- The Socket API has approximately two dozen functions
- A *socket* is a handle created by the OS that is associated with an end point of a communication channel
- A socket can be bound to a local or remote address
  - e.g., a TCP/UDP port number & IP address



Recap of the Sockets API

- Originally developed in BSD UNIX as a C language interface to the TCP/IP protocol suite
- The Socket API has approximately two dozen functions
- A *socket* is a handle created by the OS that is associated with an end point of a communication channel
- A socket can be bound to a local or remote address
- Sockets are often used to create bi-directional "reliable" communication links between software processes
  - e.g., via TCP/IP



Accidental Complexities with Socket API

Poorly structured, non-uniform, & non-portable

- *API is linear rather than hierarchical*
  - Not structured according to different phases of connection lifecycle & the roles played by participants
  - No consistency among names

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

Accidental Complexities with Socket API

Poorly structured, non-uniform, & non-portable

- *API is linear rather than hierarchical*
  - Not structured according to different phases of connection lifecycle & the roles played by participants
  - No consistency among names
- ***Non-portable & error-prone***
  - **Function names & semantics** – Different on different systems, e.g.:
    - **`send()`** & **`recv()`** used for file & socket I/O on POSIX, but only for socket I/O on Windows
    - **`accept()`** can take 0 client addr parameter on UNIX/Windows, but crashes old RTOS platforms

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

Accidental Complexities with Socket API

Poorly structured, non-uniform, & non-portable

- *API is linear rather than hierarchical*
  - Not structured according to different phases of connection lifecycle & the roles played by participants
  - No consistency among names
- *Non-portable & error-prone*
  - **Function names & semantics** – Different on different systems, e.g.:
    - **`send()`** & **`recv()`** used for file & socket I/O on POSIX, but only for socket I/O on Windows
    - **`accept()`** can take 0 client addr parameter on UNIX/Windows, but crashes old RTOS platforms
  - **Socket handle representations** – Different platforms represent sockets differently
    - e.g., **`int`** on UNIX vs. pointer on Windows

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

Accidental Complexities with Socket API

Poorly structured, non-uniform, & non-portable

- *API is linear rather than hierarchical*
  - Not structured according to different phases of connection lifecycle & the roles played by participants
  - No consistency among names
- *Non-portable & error-prone*
  - **Function names & semantics** – Different on different systems, e.g.:
    - **`send()`** & **`recv()`** used for file & socket I/O on POSIX, but only for socket I/O on Windows
    - **`accept()`** can take 0 client addr parameter on UNIX/Windows, but crashes old RTOS platforms
  - **Socket handle representations** – Different platforms represent sockets differently
    - e.g., **`int`** on UNIX vs. pointer on Windows
  - **Header files** – Platforms use different paths/names

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

Accidental Complexities with Socket API

Lack of type safety

- I/O handles not amenable to strong type checking at compile time
  - e.g., no type distinction between a socket used for passive listening & a socket used for data transfer

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

Accidental Complexities with Socket API

Lack of type safety

- I/O handles not amenable to strong type checking at compile time
  - e.g., no type distinction between a socket used for passive listening & a socket used for data transfer

Steep learning curve due to complex semantics

- Multiple protocol families & address families
- Options for infrequently features, e.g., multicast & broadcast, async & nonblocking I/O, urgent data delivery

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

Accidental Complexities with Socket API

Lack of type safety

- I/O handles not amenable to strong type checking at compile time
  - e.g., no type distinction between a socket used for passive listening & a socket used for data transfer

Steep learning curve due to complex semantics

- Multiple protocol families & address families
- Options for infrequently features, e.g., multicast & broadcast, async & nonblocking I/O, urgent data delivery

Many low-level details

- Forgetting to use network byte order for port numbers
- Forgetting to initialize C structures, e.g., **`sockaddr`**
- Possibility of omitting a function, such as **`listen()`**
- Possibility of mismatch between protocol & address families due to decoupled usage

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

Example of Socket API Accidental Complexities

Most of these problems won't be detected by the compiler!

```
#include <sys/types.h>
#include <sys/socket.h>

const int PORT_NUM = 10000;

int echo_server ()
{
  struct sockaddr_in addr;
  int addr_len;
  char buf[BUFSIZ];
  int d_handle;
  // Create the local endpoint.
```

Possible differences in header file names

Forgot to initialize to `sizeof (sockaddr_in)` to 0

Use of non-portable handle type

Example of Socket API Accidental Complexities

Meant to say SOCK_STREAM

```
int a_handle = socket (PF_UNIX, SOCK_DGRAM, 0);
if (a_handle == -1) return -1;

```

Use of non-portable return value

```
// Set up address information where server listens.
addr.sin_family = AF_INET;
```

Protocol & address family mismatch

```
addr.sin_port = PORT_NUM;
```

May be wrong byte order

```
addr.sin_addr.addr = INADDR_ANY;

if (bind (a_handle, (struct sockaddr *) &addr,
          sizeof addr) == -1)
  return -1;

```

Structure fields not completely zeroed out

Example of Socket API Accidental Complexities

Forgot call to `listen()` before `accept()`

```
  // Create a new communication endpoint.
```

Can't call `accept()` on `SOCK_DGRAM`

```
  if (d_handle = accept (a_handle, (struct sockaddr *) &addr,
                         &addr_len) != -1) {
    int n;
```

Reading from wrong handle

```
    while ((n = read (a_handle, buf, sizeof buf)) > 0)
      write (d_handle, buf, n);
```

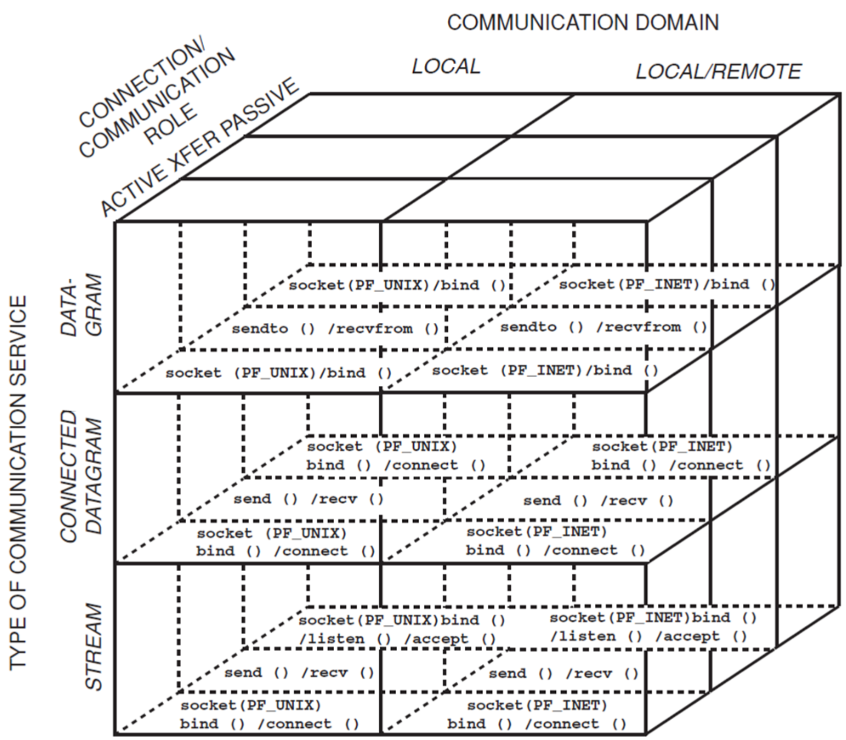
No guarantee that "n" bytes will be written

```

    close (d_handle);
  }
  return 0;
}
```

These types of problems are all too common with the Sockets API

Summary

- The native Socket API suffers from many accidental complexities
  - e.g., it's tedious, error-prone, overly complex, & non-portable/ non-uniform



Summary

- The native Socket API suffers from many accidental complexities
  - e.g., it's tedious, error-prone, overly complex, & non-portable/ non-uniform
- Although our analysis focused on the Socket API, this critique also applies to other native OS systems programming APIs defined using C

C APIs

Thread Library	Stream Pipes	Sockets /TLI	Named Pipes	Select /Poll	Dynamic Linking	Memory Mapping	System V IPC

Process/Thread Subsystem | **Communication Subsystem** | **Virtual Memory Subsystem**

General POSIX, Windows, & RTOS Services

See www.dre.vanderbilt.edu/~schmidt/PDF/C++-wrappers.pdf for more info

Patterns & Frameworks for Service Access & Communication: Part 2

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

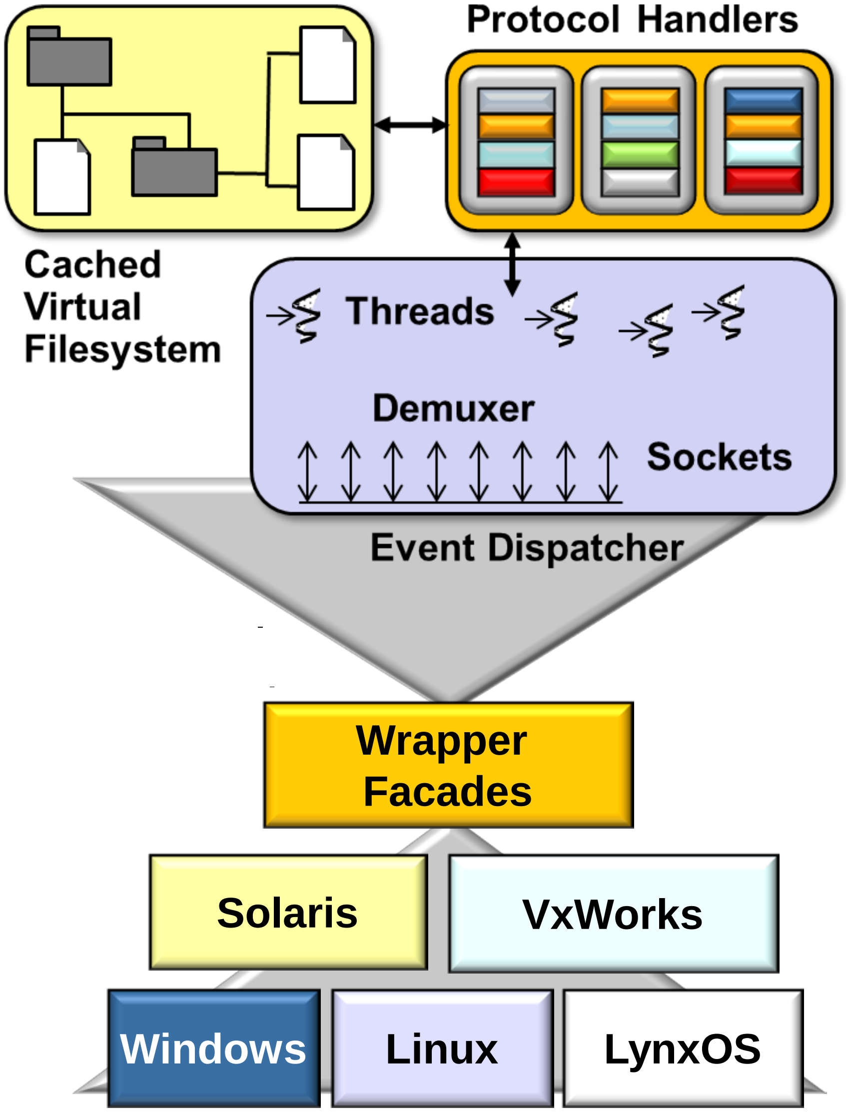
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

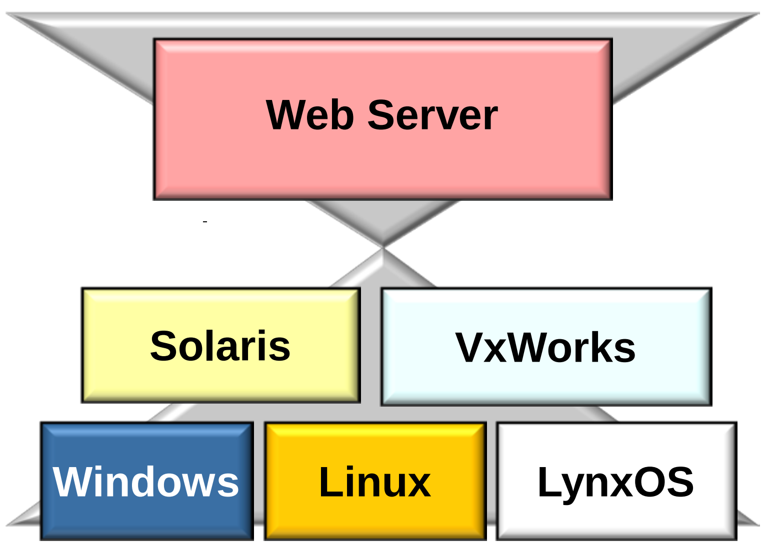
Topics Covered in this Part of the Module

- Summarize the accidental complexities with Socket API
- Describe how the *Wrapper Façade* pattern can alleviate accidental complexities with C APIs for JAWS



Encapsulating Low-level OS APIs

Context	Problem
• Web servers must manage various OS services • e.g., processes, threads, connections, memory, files, etc.	• Programming directly to low-level OS APIs is tedious, error-prone, & non-portable due accidental complexities



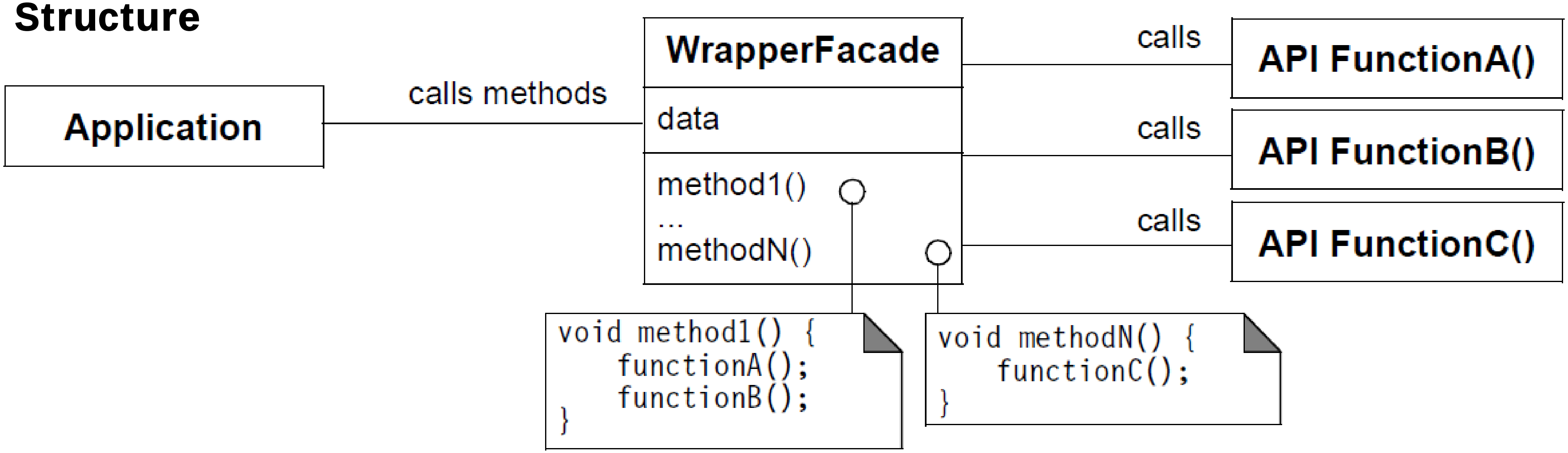

```
#if defined (_WIN32)
    #include <windows.h>
    typedef int ssize_t;
#else
    typedef unsigned int UINT32;
    #include <thread.h>
    #include <unistd.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <memory.h>
#endif /* _WIN32 */

// Keep track of number of logging requests.
static int request_count;

// Lock that serializes concurrent access to request_count.
#if defined (_WIN32)
    static CRITICAL_SECTION lock;
#else
    static mutex_t lock;
#endif /* _WIN32 */
```

Encapsulating Low-level OS APIs

Context	Problem	Solution
• Web servers must manage various OS services • e.g., processes, threads, connections, memory, files, etc.	• Programming directly to low-level OS APIs is tedious, error-prone, & non-portable due accidental complexities	• Apply the *Wrapper Facade* pattern to avoid accessing low-level operating system APIs directly

Wrapper Façade encapsulates data & functions provided by existing C APIs within more concise, robust, portable, maintainable, & cohesive object-oriented classes

Structure

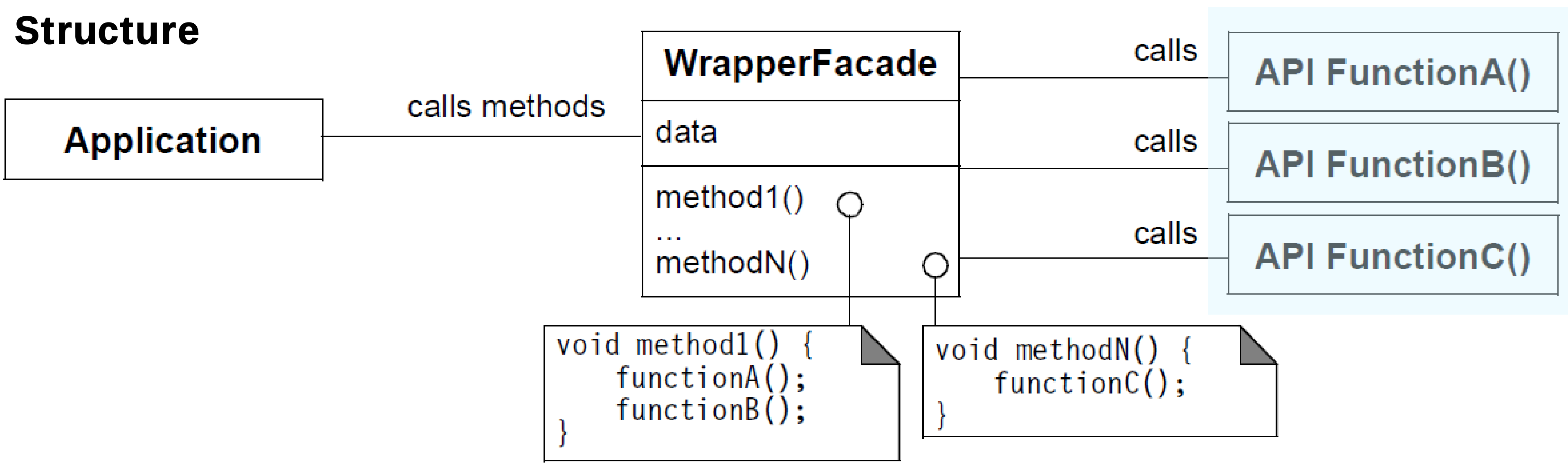


Encapsulating Low-level OS APIs

Context	Problem	Solution
• Web servers must manage various OS services • e.g., processes, threads, connections, memory, files, etc.	• Programming directly to low-level OS APIs is tedious, error-prone, & non-portable due accidental complexities	• Apply the *Wrapper Facade* pattern to avoid accessing low-level operating system APIs directly

Wrapper Façade encapsulates data & functions provided by existing C APIs within more concise, robust, portable, maintainable, & cohesive object-oriented classes

Structure

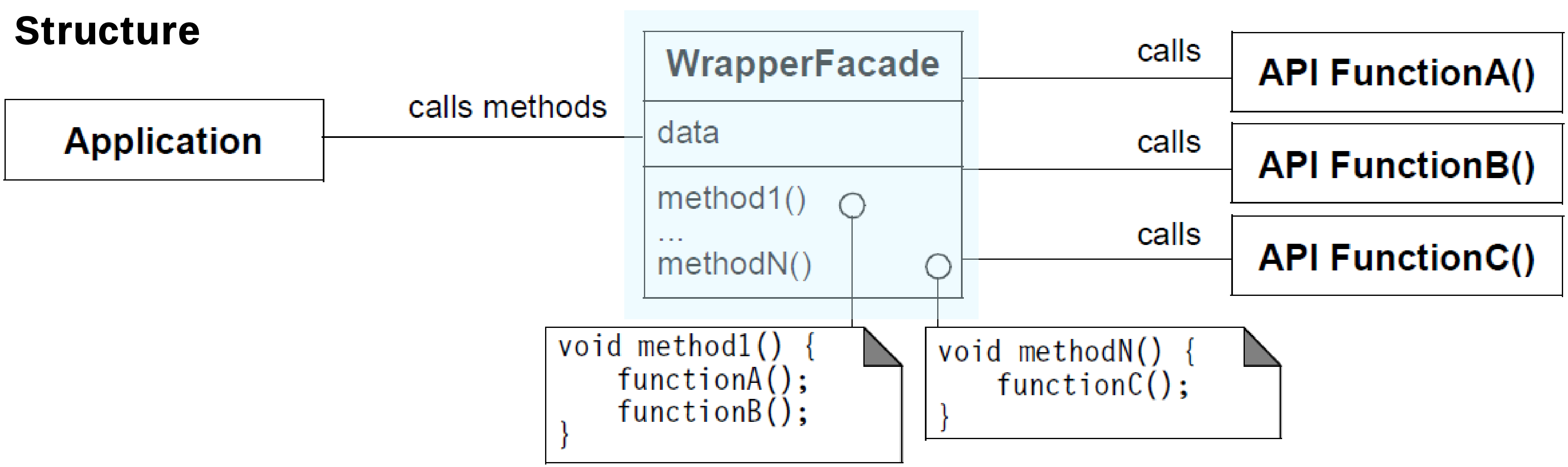


Encapsulating Low-level OS APIs

Context	Problem	Solution
• Web servers must manage various OS services • e.g., processes, threads, connections, memory, files, etc.	• Programming directly to low-level OS APIs is tedious, error-prone, & non-portable due accidental complexities	• Apply the *Wrapper Facade* pattern to avoid accessing low-level operating system APIs directly

Wrapper Façade encapsulates data & functions provided by existing C APIs within more concise, robust, portable, maintainable, & cohesive object-oriented classes

Structure

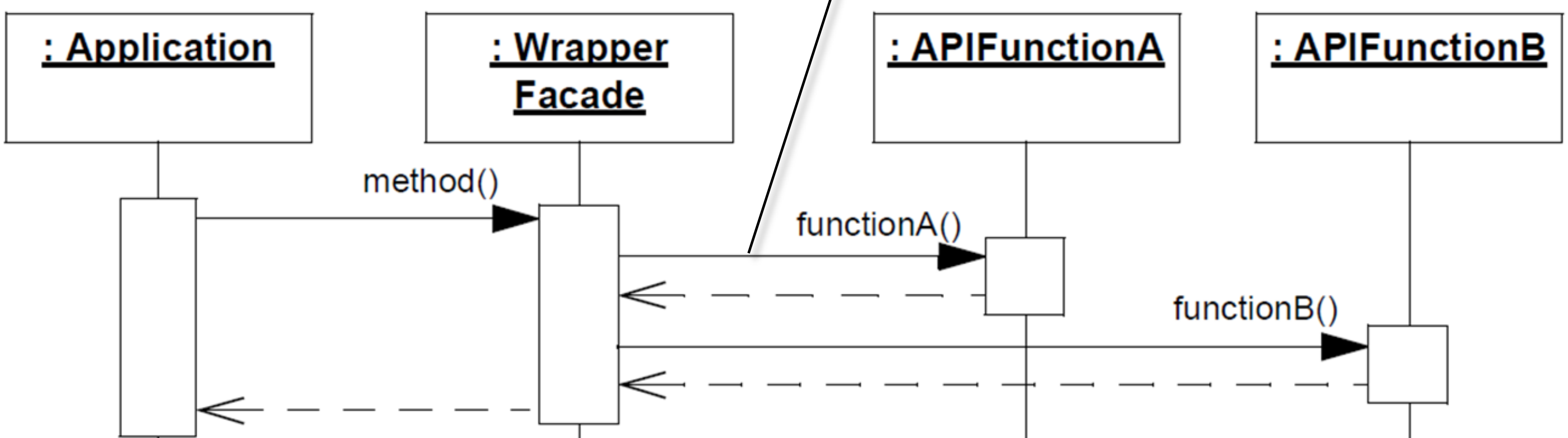


Encapsulating Low-level OS APIs

Context	Problem	Solution
• Web servers must manage various OS services • e.g., processes, threads, connections, memory, files, etc.	• Programming directly to low-level OS APIs is tedious, error-prone, & non-portable due accidental complexities	• Apply the *Wrapper Facade* pattern to avoid accessing low-level operating system APIs directly

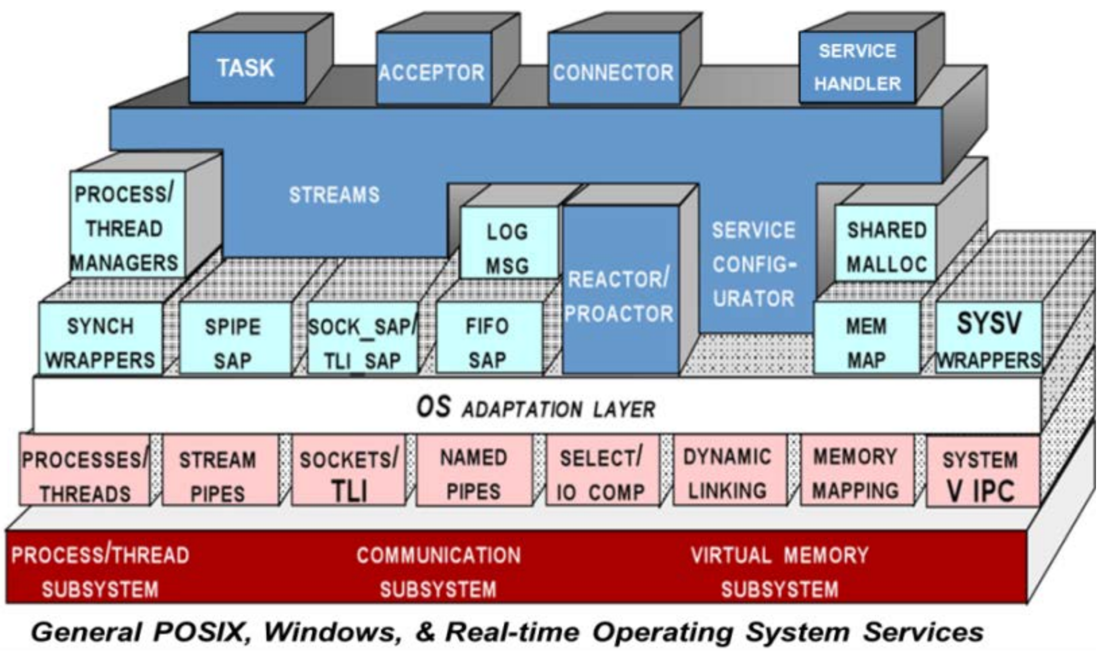
Dynamics

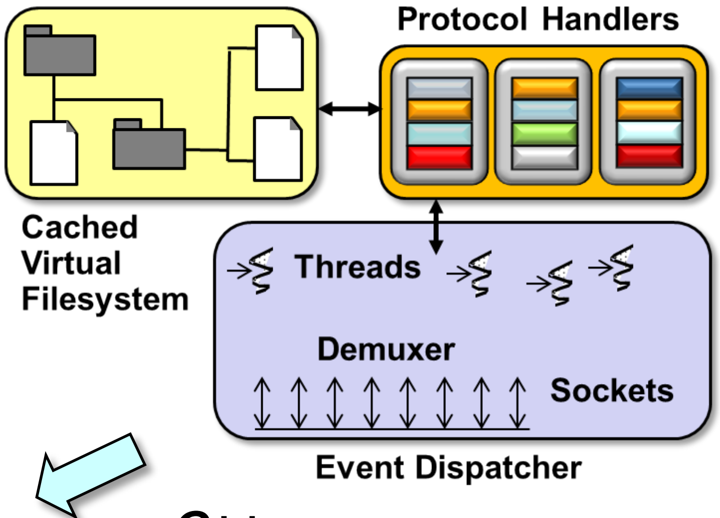
The structure & dynamics look similar to Bridge & Proxy, but the implementation is typically lighter weight



Applying the Wrapper Façade Pattern in JAWS

- JAWS uses the wrapper facades defined by ACE to ensure it can run on many OS platforms
  - e.g., Windows, UNIX, & many real-time operating systems

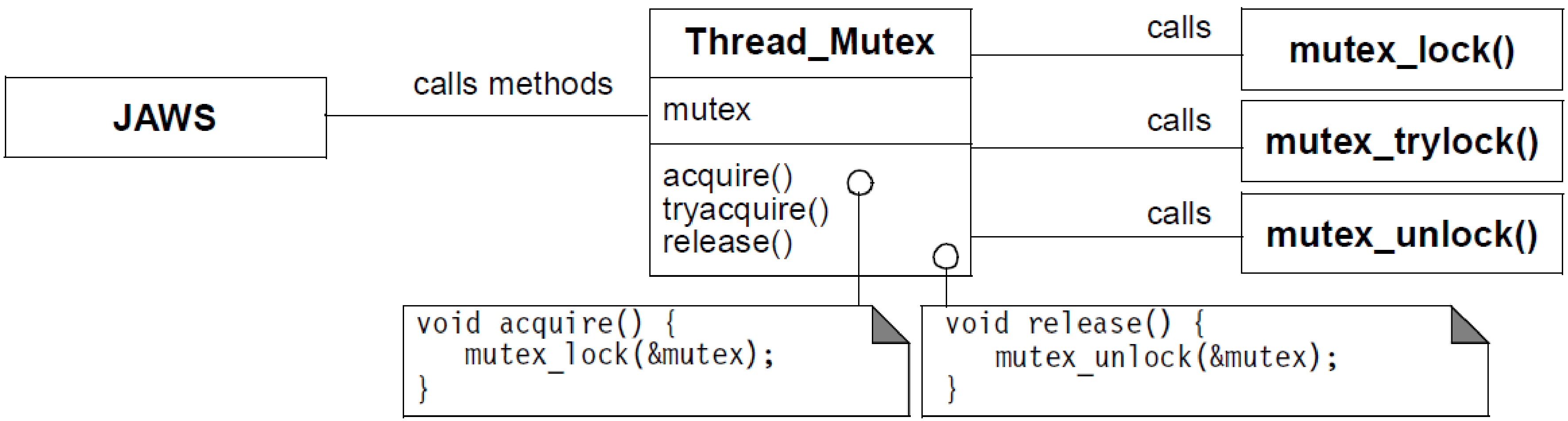




C++ Wrapper Facades

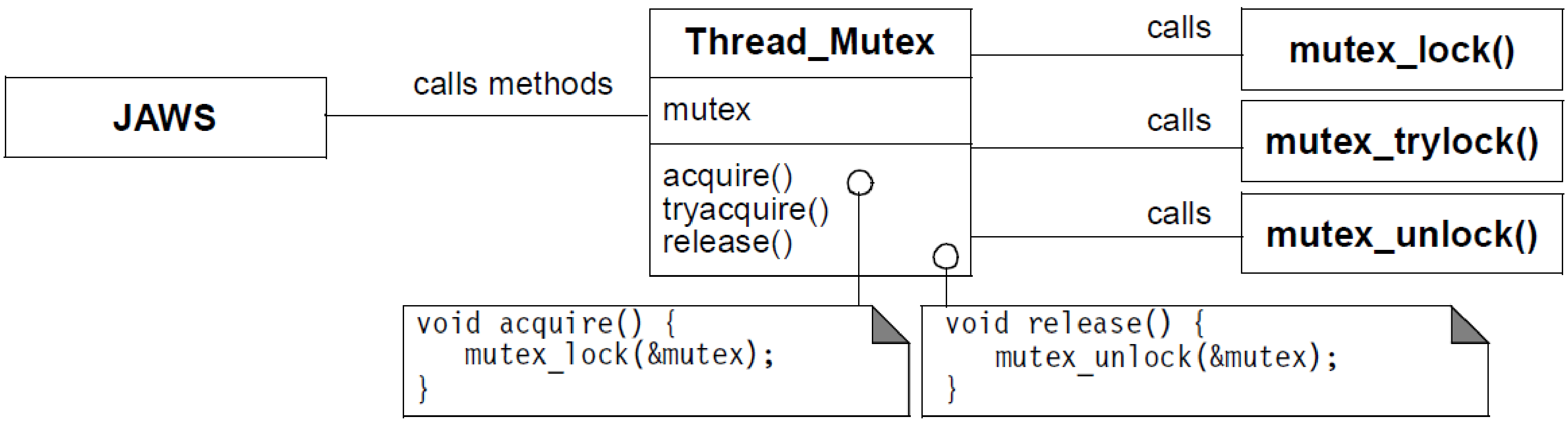
Applying the Wrapper Façade Pattern in JAWS

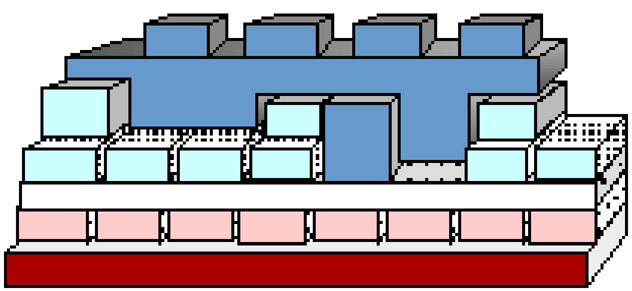
- JAWS uses the wrapper facades defined by ACE to ensure it can run on many OS platforms
  - e.g., Windows, UNIX, & many real-time operating systems
- JAWS uses the **`ACE_Thread_Mutex`** wrapper facade in ACE to portably access mutual exclusion mechanisms provided by various operating systems



Applying the Wrapper Façade Pattern in JAWS

- JAWS uses the wrapper facades defined by ACE to ensure it can run on many OS platforms
  - e.g., Windows, UNIX, & many real-time operating systems
- JAWS uses the **`ACE_Thread_Mutex`** wrapper facade in ACE to portably access mutual exclusion mechanisms provided by various operating systems



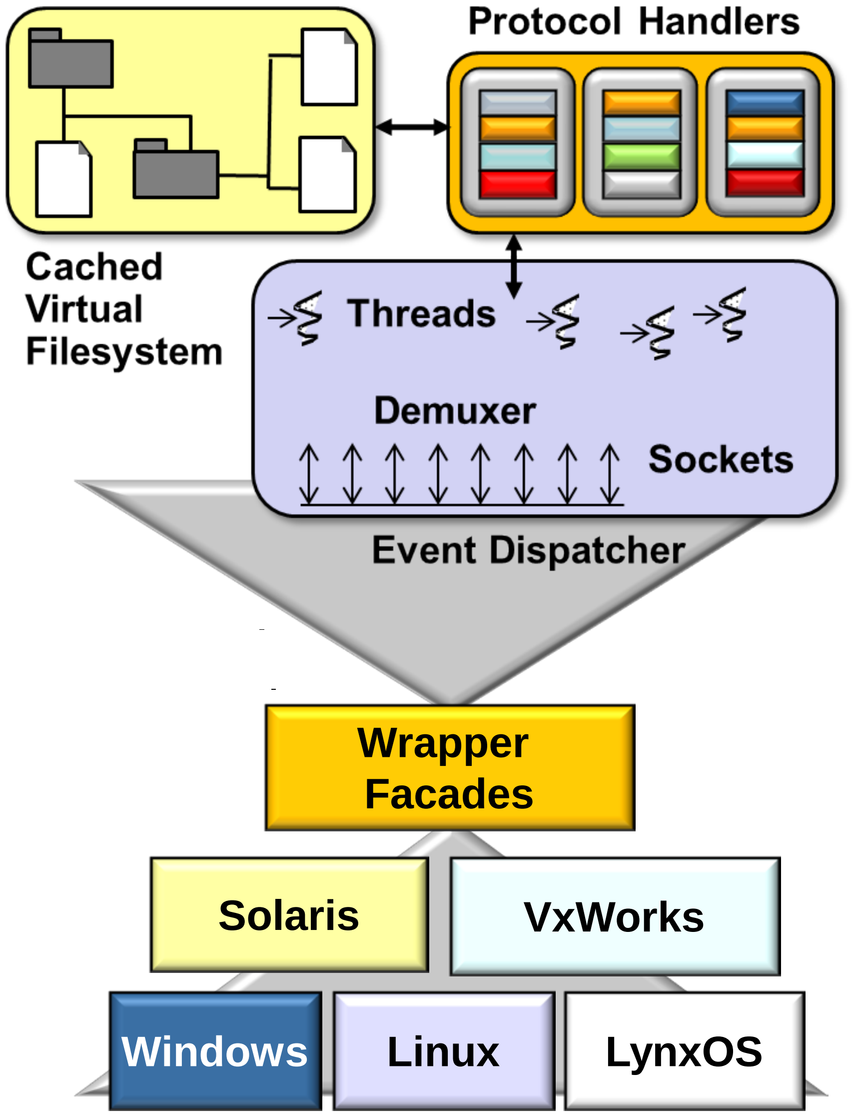


Other ACE wrapper facades used in JAWS encapsulate sockets, process & thread management, memory-mapped & regular files, explicit dynamic linking, etc.

Benefits of the Wrapper Façade Pattern

Concise & robust higher-level OO programming interfaces

- Reduce the tedium & increase the type-safety of developing apps, which decreases certain types of accidental complexities



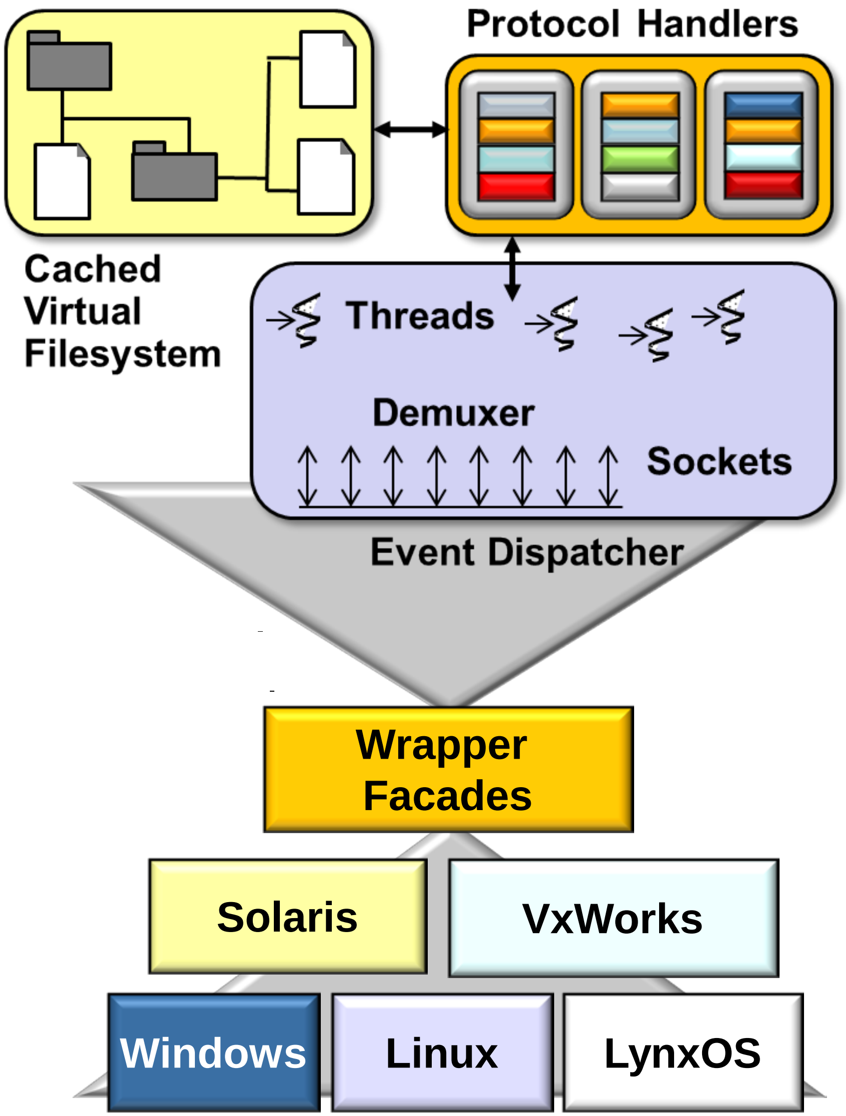
Benefits of the Wrapper Façade Pattern

Concise & robust higher-level OO programming interfaces

- Reduce the tedium & increase the type-safety of developing apps, which decreases certain types of accidental complexities

Portability & maintainability

- Shield app developers from non-portable aspects of lower-level APIs



Benefits of the Wrapper Façade Pattern

Concise & robust higher-level OO programming interfaces

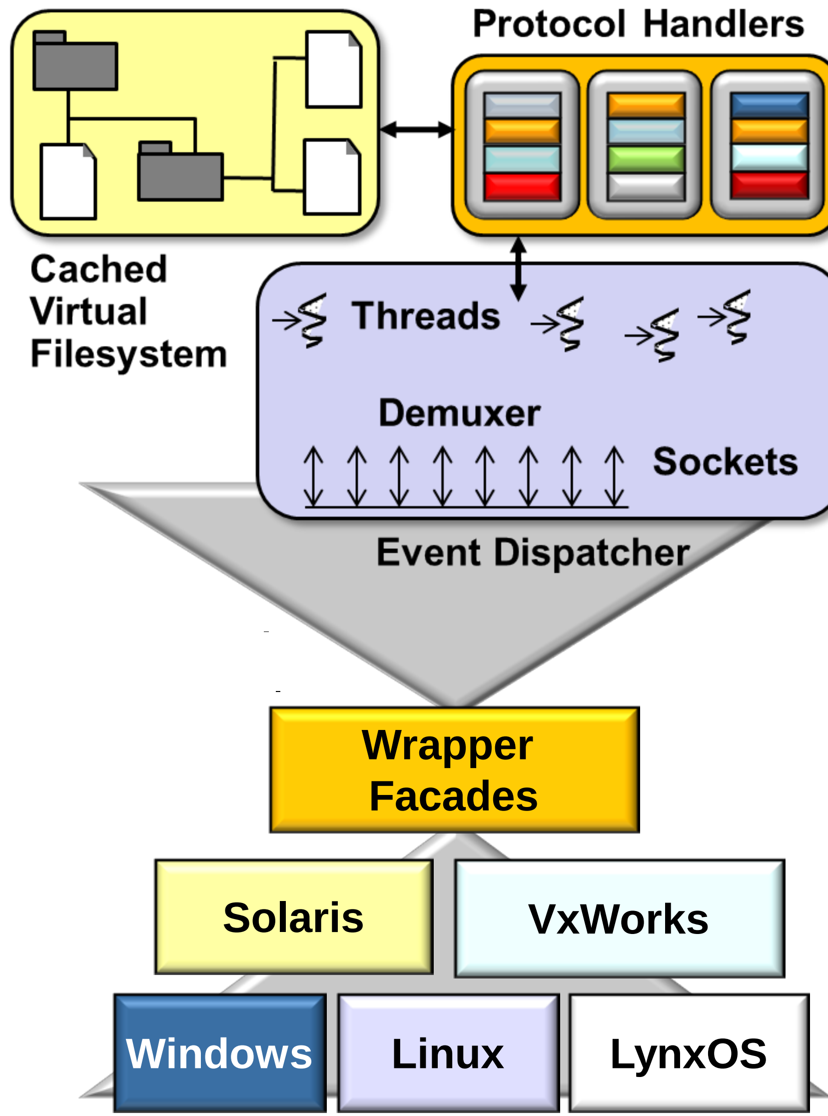
- Reduce the tedium & increase the type-safety of developing apps, which decreases certain types of accidental complexities

Portability & maintainability

- Shield app developers from non-portable aspects of lower-level APIs

Modularity, reusability, & configurability

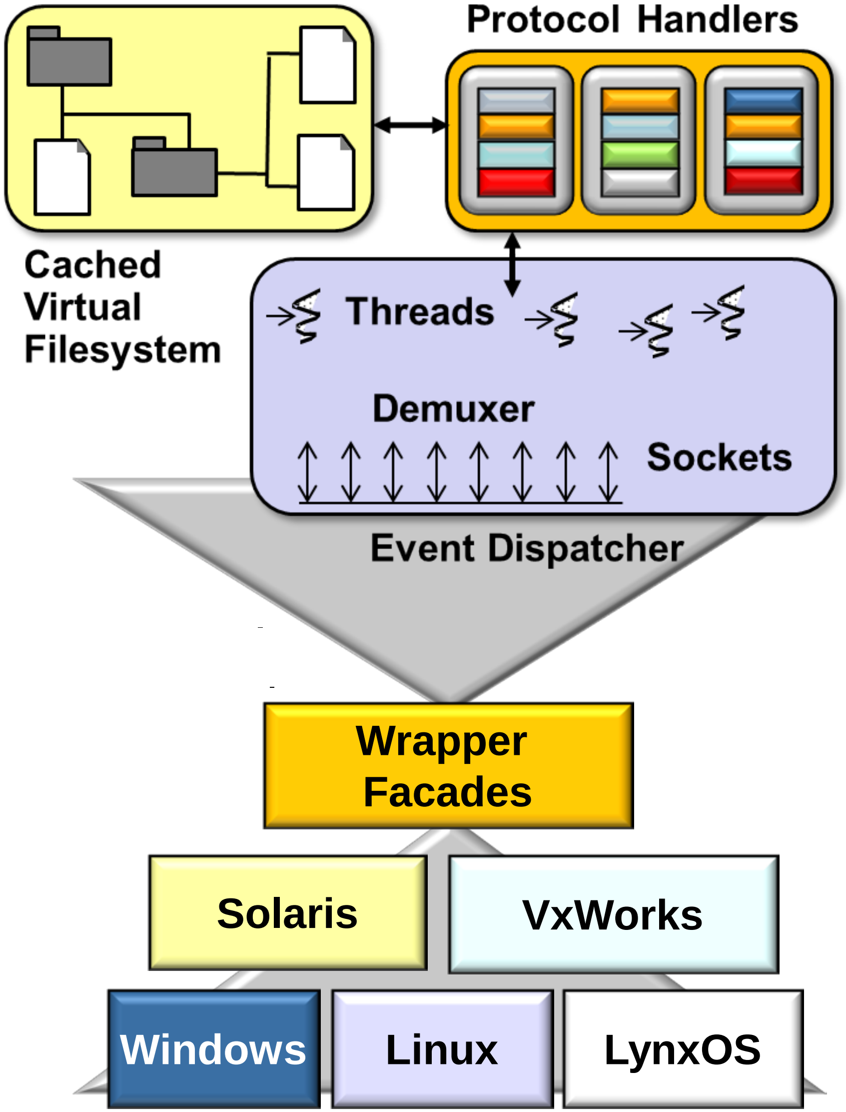
- Creates cohesive & reusable class components that can be 'plugged' into other components in a wholesale fashion
  - e.g., using OO language features like inheritance & parameterized types



Limitations of the Wrapper Façade Pattern

Loss of functionality

- Whenever a portable abstraction is layered on top of an existing API it's possible to lose functionality



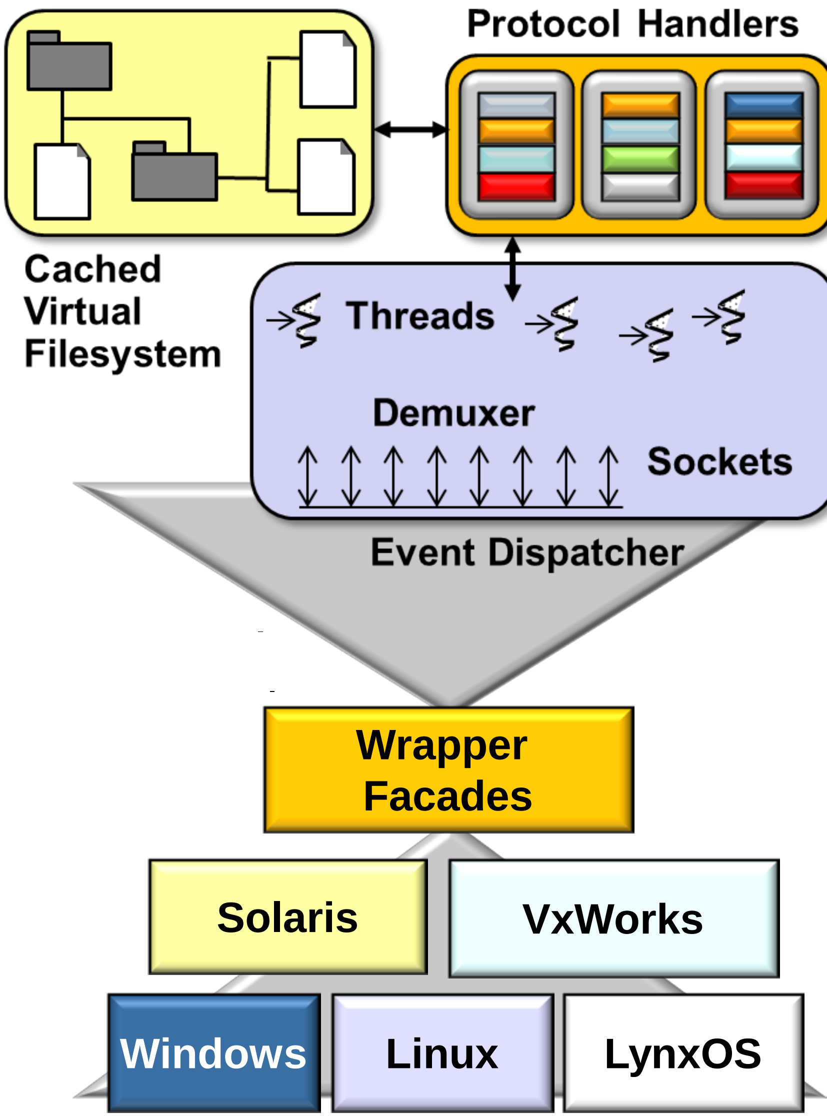
Limitations of the Wrapper Façade Pattern

Loss of functionality

- Whenever a portable abstraction is layered on top of an existing API it's possible to lose functionality

Performance degradation

- Performance can degrade if many forwarding function calls and/or indirections are made per wrapper façade method



Limitations of the Wrapper Façade Pattern

Loss of functionality

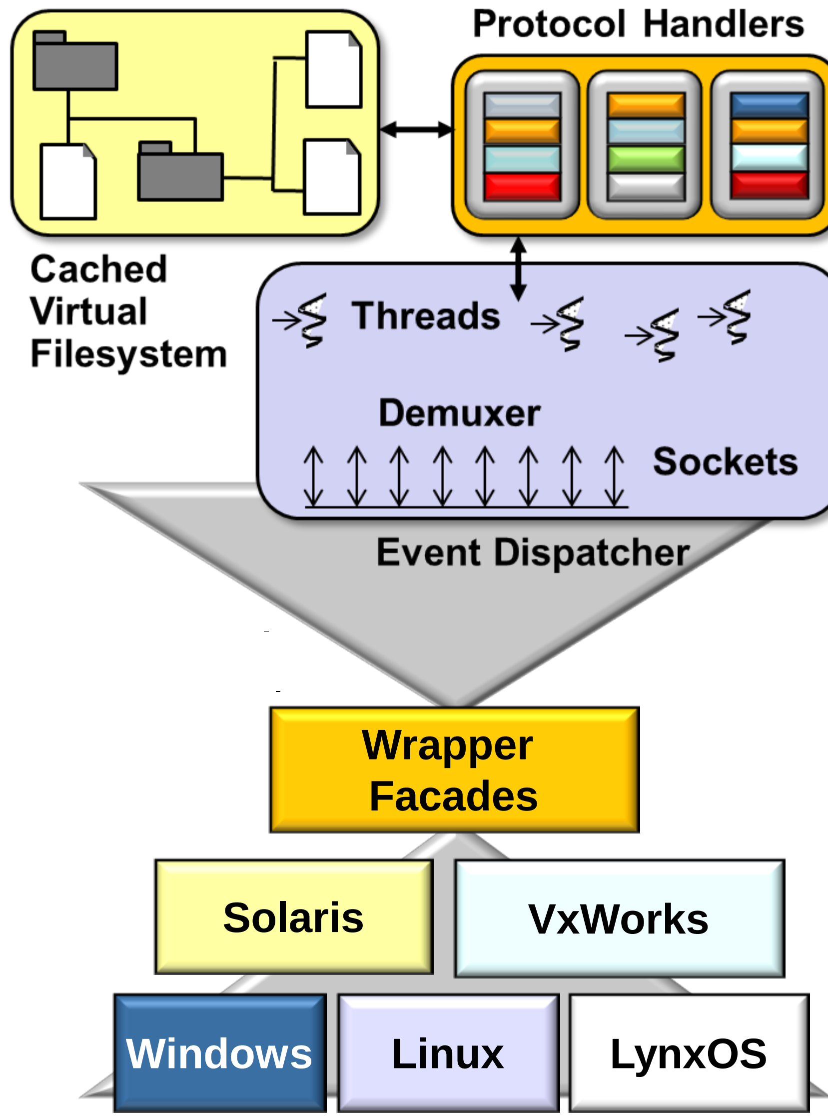
- Whenever a portable abstraction is layered on top of an existing API it's possible to lose functionality

Performance degradation

- Performance can degrade if many forwarding function calls and/or indirections are made per wrapper façade method

Programming language & compiler limitations

- May be hard to define wrapper facades for certain languages due to a lack of language support or limitations with compilers



Patterns & Frameworks for Service Access & Communication: Part 3

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

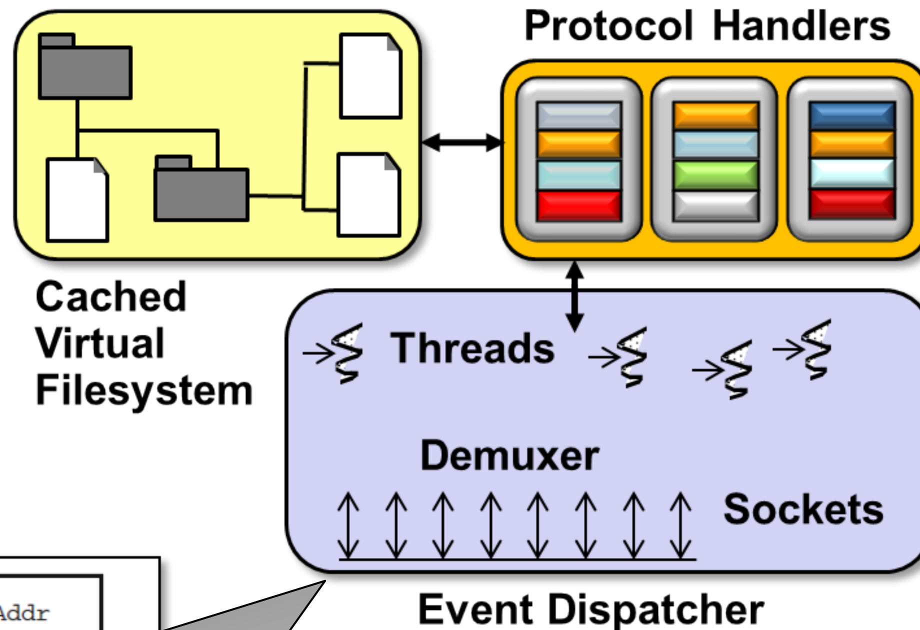
Professor of Computer Science

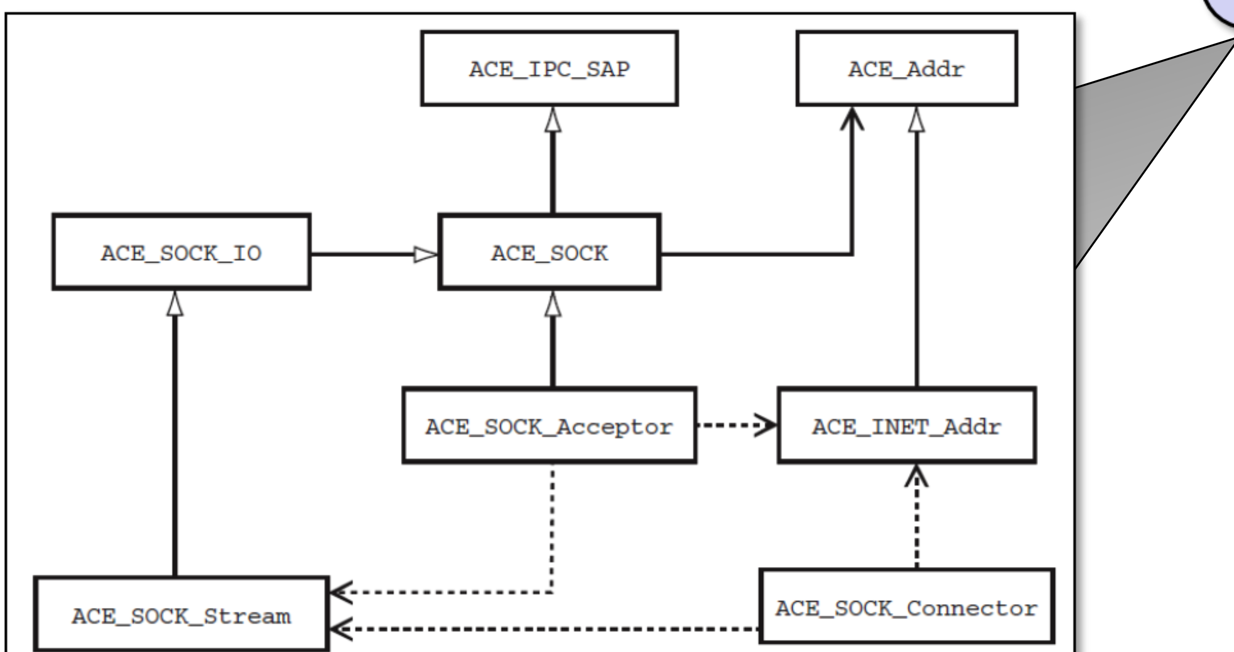
Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

- Summarize the accidental complexities with Socket API
- Describe how the *Wrapper Façade* pattern can alleviate accidental complexities with C APIs for JAWS
- Describe the ACE C++ Socket wrapper façades

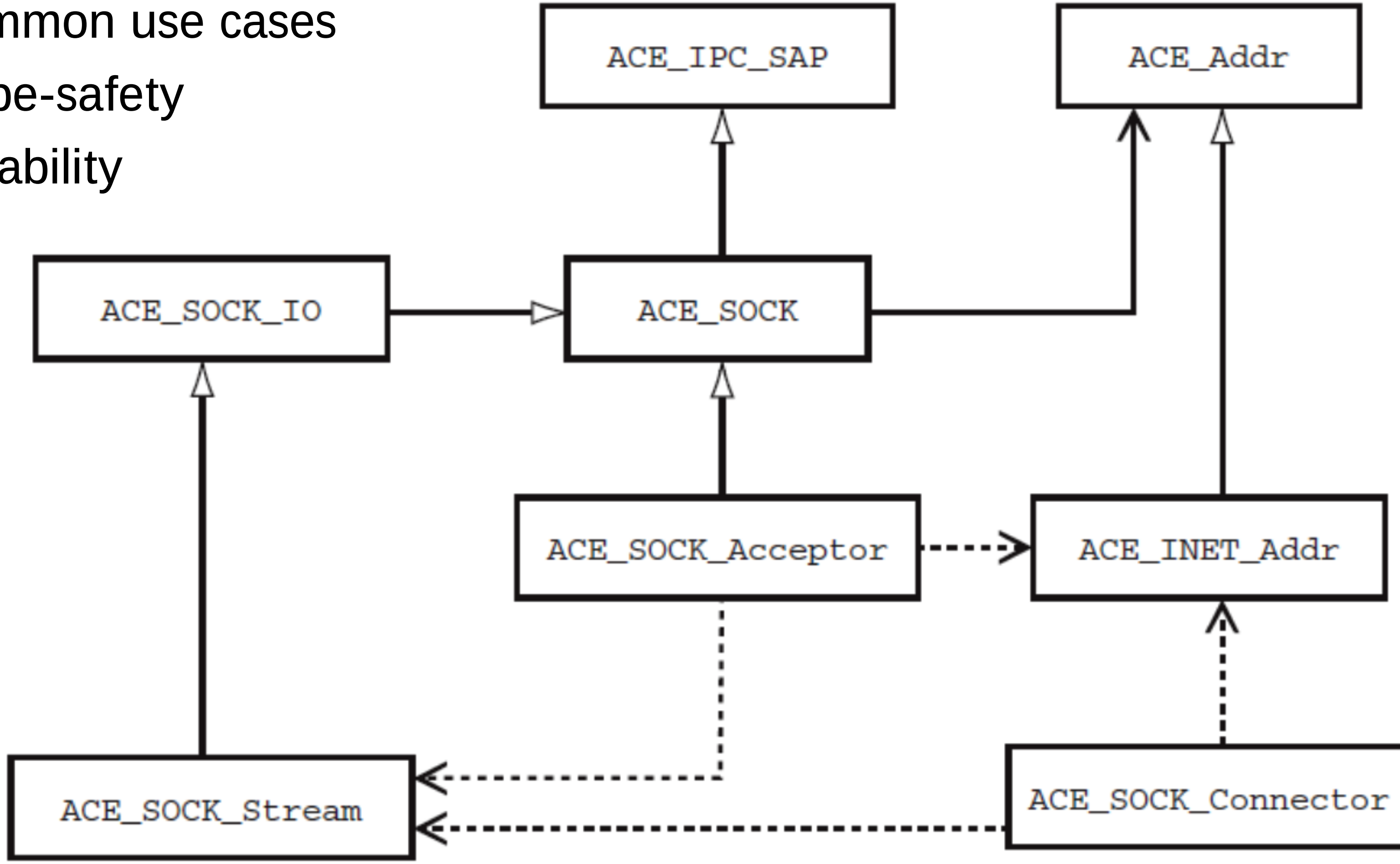




ACE Socket Wrapper Façade Classes

ACE defines a set of C++ classes that address limitations with Socket API

- Building blocks for higher-level abstractions
- Simplify common use cases
- Enhance type-safety
- Ensure portability



These classes are designed in accordance with the *Wrapper Facade* pattern

ACE Socket Wrapper Façade Classes

ACE defines a set of C++ classes that address limitations with Socket API

- Building blocks for higher-level abstractions
- Simplify common use cases
- Enhance type-safety
- Ensure portability

ACE Class	Description
ACE_INET_Addr	Encapsulates the Internet-domain address family
ACE_SOCK_IO **ACE_SOCK_Stream**	Encapsulate the data transfer mechanisms supported by data-mode sockets
ACE_SOCK_Connector	A factory that connects to a peer acceptor & then initializes a new endpoint of communication in an **ACE_SOCK_Stream** object
ACE_SOCK_Acceptor	A factory that initializes a new endpoint of communication in an **ACE_SOCK_Stream** object in response to a connection request from a peer connector

ACE Socket Wrapper Façade Classes

ACE defines a set of C++ classes that address limitations with Socket API

- Building blocks for higher-level abstractions
- Simplify common use cases
- Enhance type-safety
- Ensure portability

ACE Class	Description
ACE_INET_Addr	Encapsulates the Internet-domain address family
ACE_SOCK_IO **ACE_SOCK_Stream**	Encapsulate the data transfer mechanisms supported by data-mode sockets
ACE_SOCK_Connector	A factory that connects to a peer acceptor & then initializes a new endpoint of communication in an **ACE_SOCK_Stream** object
ACE_SOCK_Acceptor	A factory that initializes a new endpoint of communication in an **ACE_SOCK_Stream** object in response to a connection request from a peer connector

ACE Socket Wrapper Façade Classes

ACE defines a set of C++ classes that address limitations with Socket API

- Building blocks for higher-level abstractions
- Simplify common use cases
- Enhance type-safety
- Ensure portability

ACE Class	Description
ACE_INET_Addr	Encapsulates the Internet-domain address family
ACE_SOCK_IO **ACE_SOCK_Stream**	Encapsulate the data transfer mechanisms supported by data-mode sockets
ACE_SOCK_Connector	A factory that connects to a peer acceptor & then initializes a new endpoint of communication in an **ACE_SOCK_Stream** object
ACE_SOCK_Acceptor	A factory that initializes a new endpoint of communication in an **ACE_SOCK_Stream** object in response to a connection request from a peer connector

ACE Socket Wrapper Façade Classes

ACE defines a set of C++ classes that address limitations with Socket API

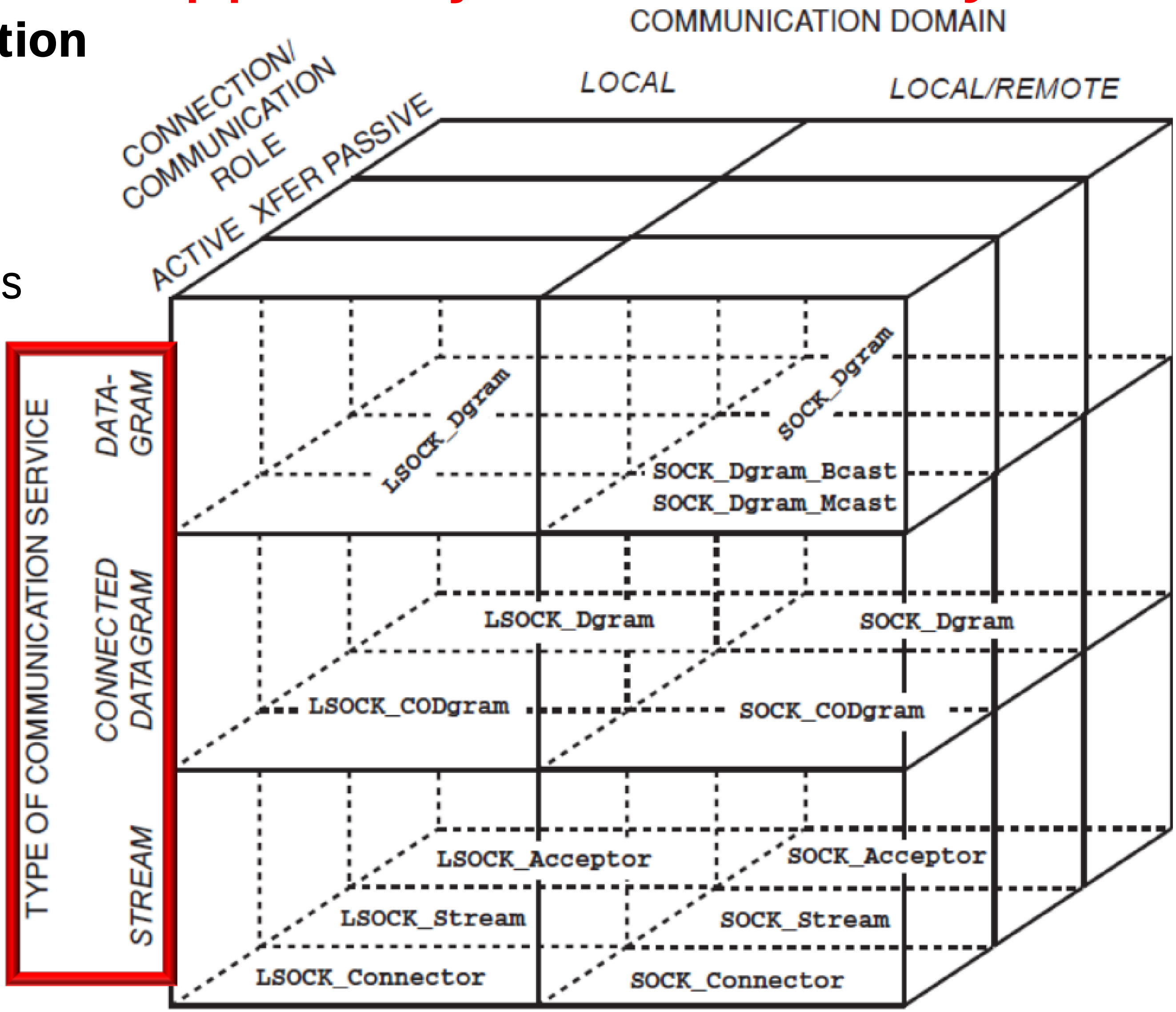
- Building blocks for higher-level abstractions
- Simplify common use cases
- Enhance type-safety
- Ensure portability

ACE Class	Description
ACE_INET_Addr	Encapsulates the Internet-domain address family
ACE_SOCK_IO **ACE_SOCK_Stream**	Encapsulate the data transfer mechanisms supported by data-mode sockets
ACE_SOCK_Connector	A factory that connects to a peer acceptor & then initializes a new endpoint of communication in an **ACE_SOCK_Stream** object
ACE_SOCK_Acceptor	A factory that initializes a new endpoint of communication in an **ACE_SOCK_Stream** object in response to a connection request from a peer connector

See www.dre.vanderbilt.edu/~schmidt/PDF/Socket-Wrappers.pdf for more info

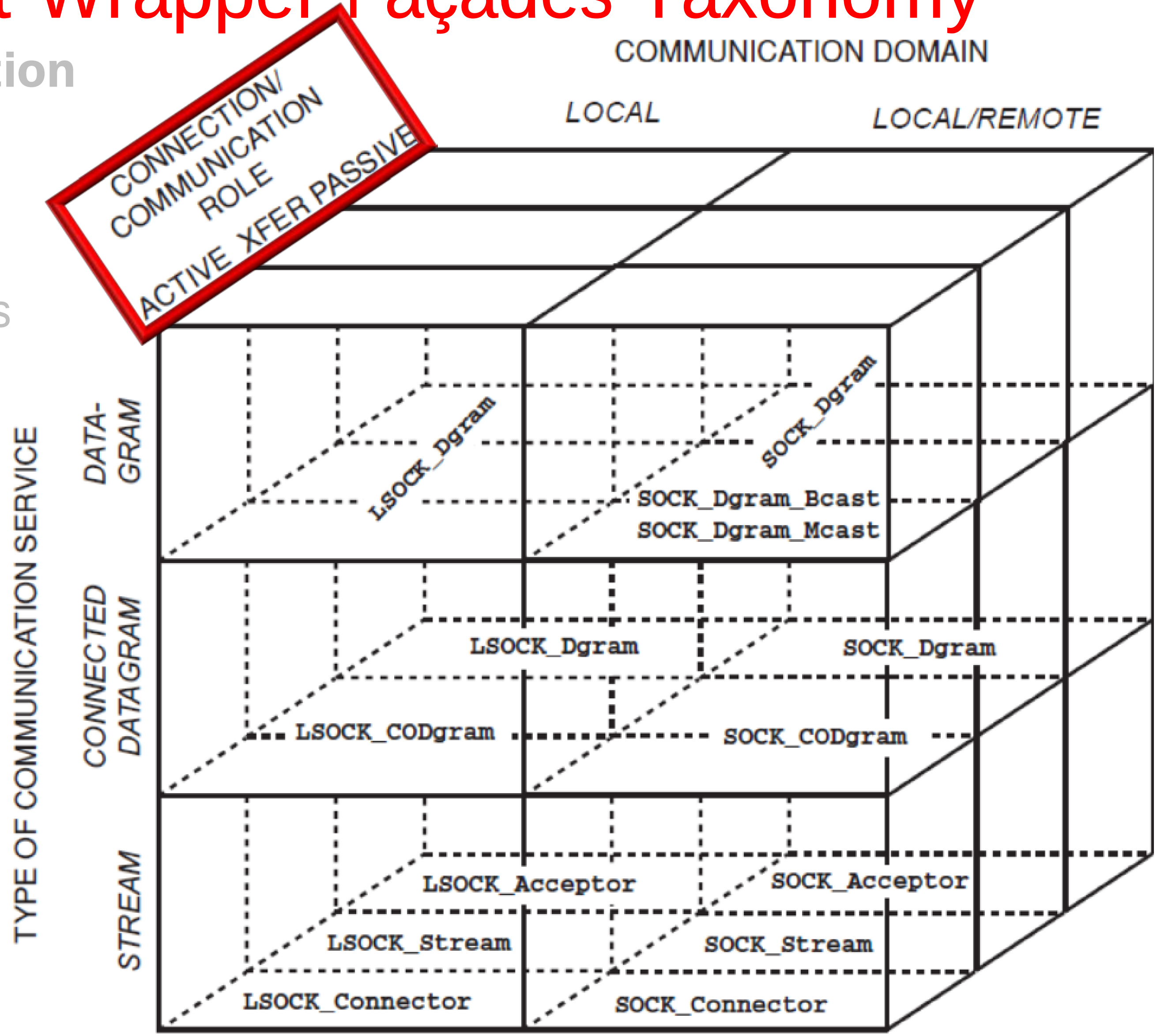
ACE Socket Wrapper Façades Taxonomy

- **Type of communication service**
  - e.g., streams vs. datagrams vs. connected datagrams



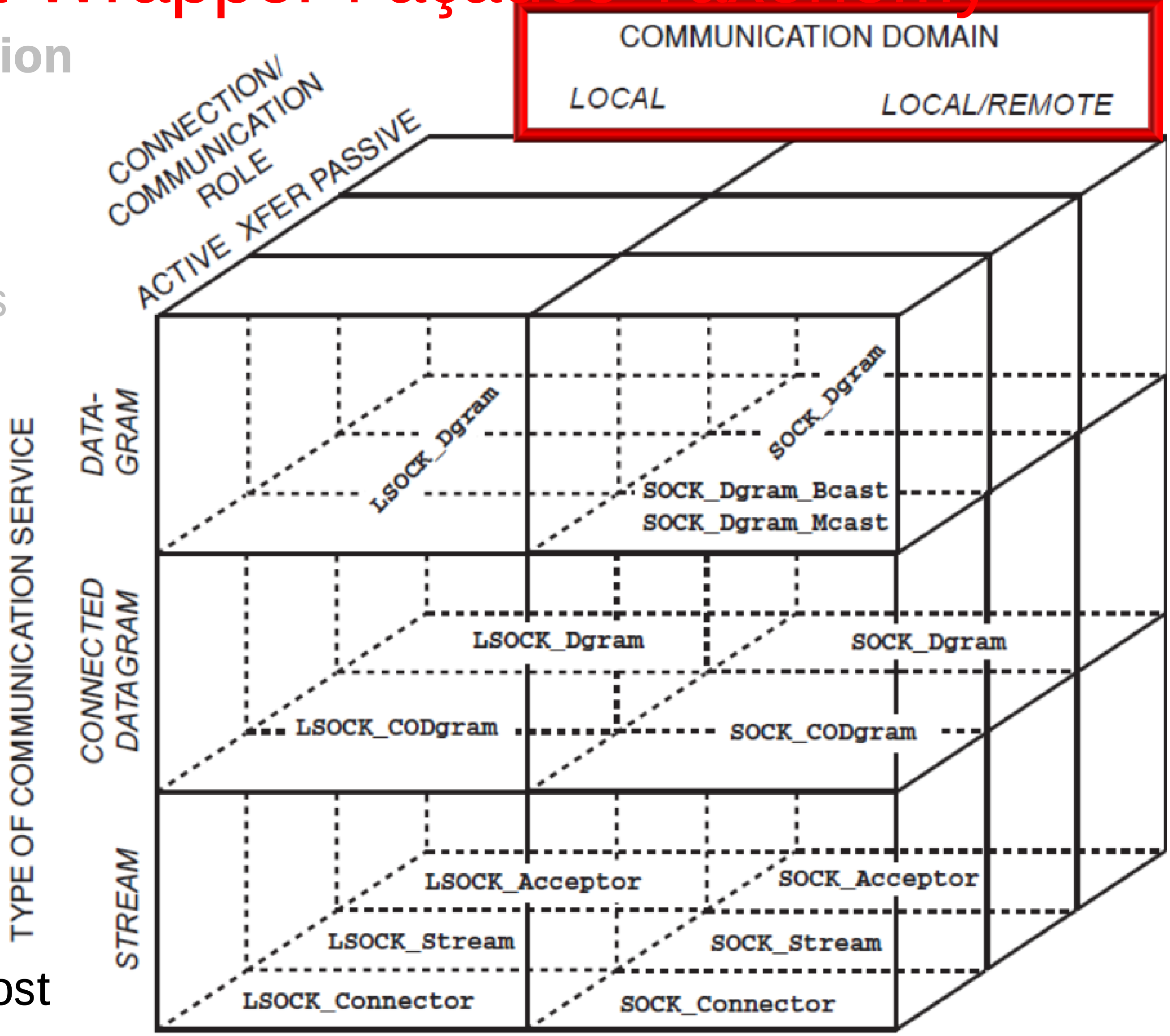
ACE Socket Wrapper Façades Taxonomy

- Type of communication service
  - e.g., streams vs. datagrams vs. connected datagrams
- **Connection & communication role**
  - e.g., clients often initiate connections *actively*, whereas servers often accept them *passively*



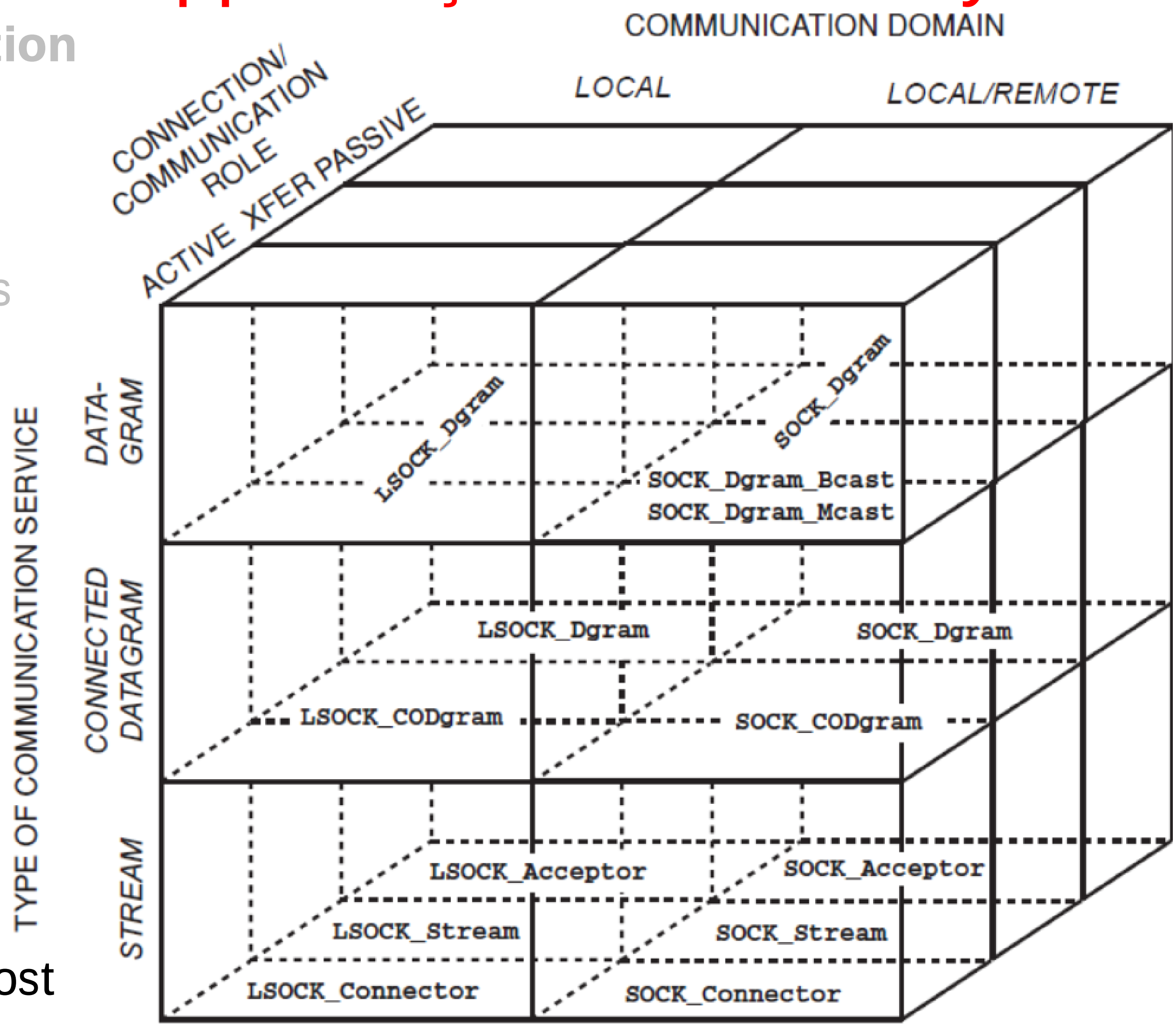
ACE Socket Wrapper Façades Taxonomy

- **Type of communication service**
  - e.g., streams vs. datagrams vs. connected datagrams
- **Connection & communication role**
  - e.g., clients often initiate connections *actively*, whereas servers often accept them *passively*
- **Communication domain**
  - e.g., local host only vs. local or remote host



ACE Socket Wrapper Façades Taxonomy

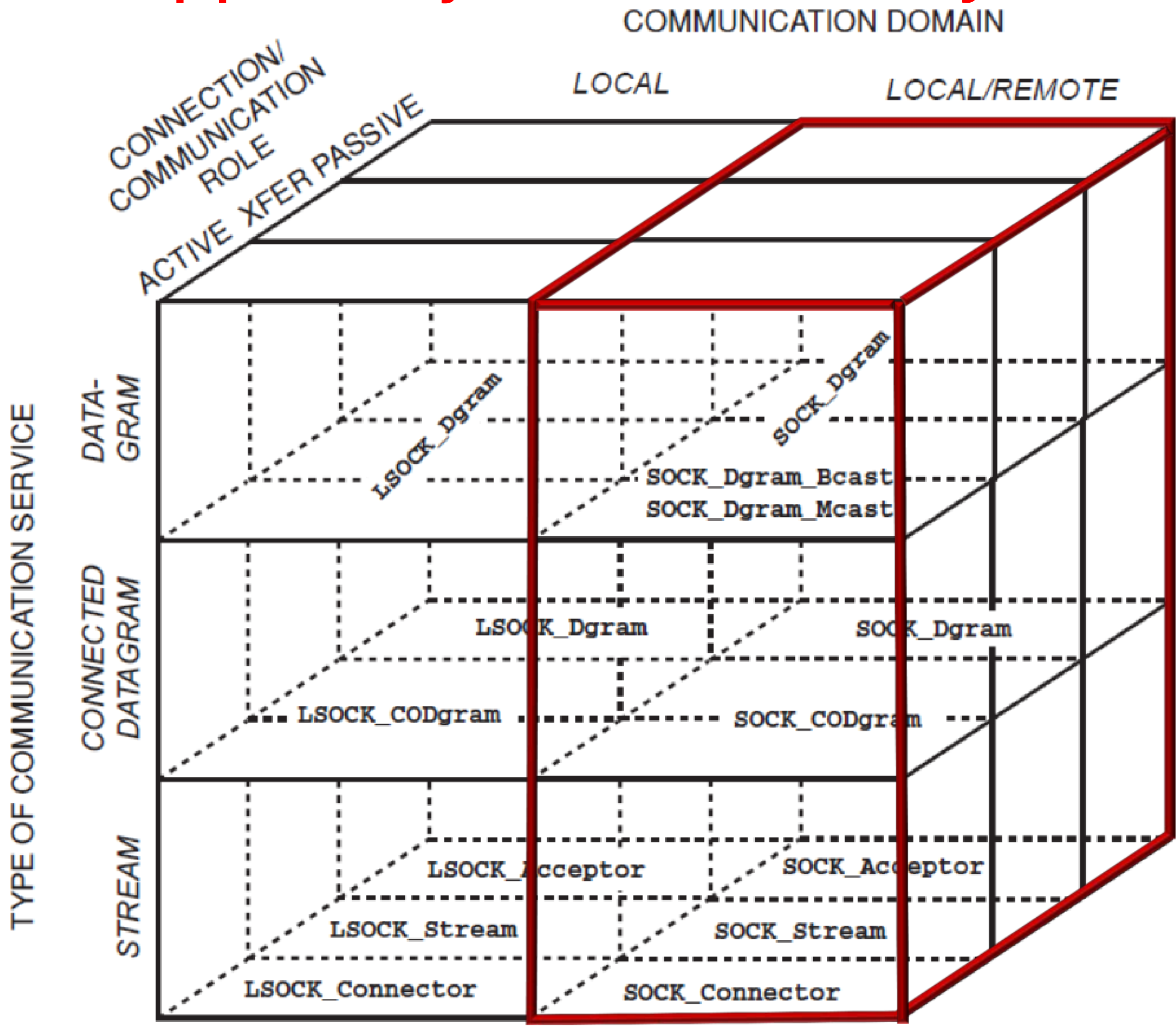
- **Type of communication service**
  - e.g., streams vs. datagrams vs. connected datagrams
- **Connection & communication role**
  - e.g., clients often initiate connections *actively*, whereas servers often accept them *passively*
- **Communication domain**
  - e.g., local host only vs. local or remote host



Each of these *types*, *roles*, & *domains* are codified in the OO class design

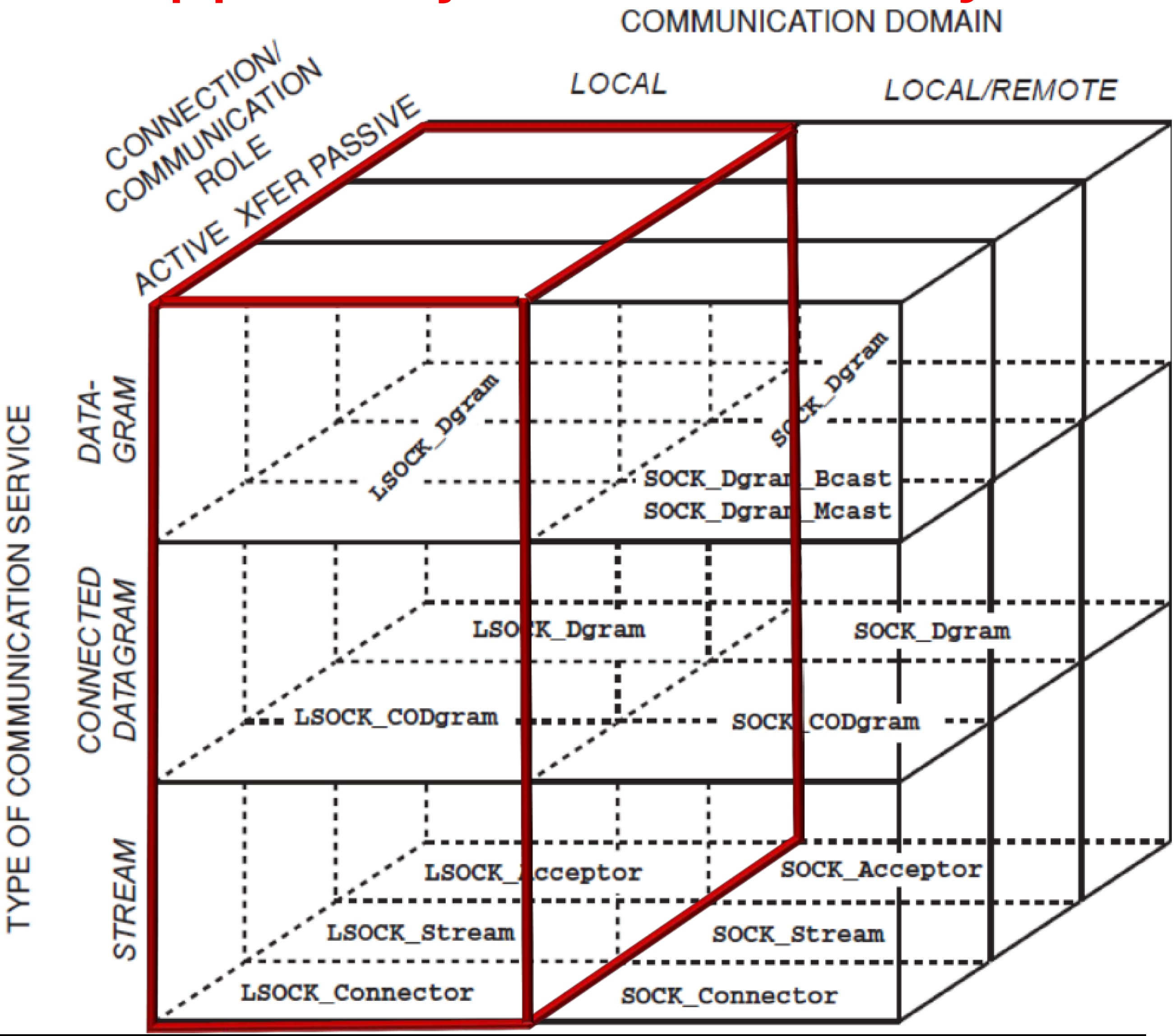
ACE Socket Wrapper Façades Taxonomy

- ACE Socket wrapper façade structure reflects the domain of networked IPC & provide the following capabilities:
  - **ACE_SOCK_*** classes encapsulate Internet-domain Socket API functionality



ACE Socket Wrapper Façades Taxonomy

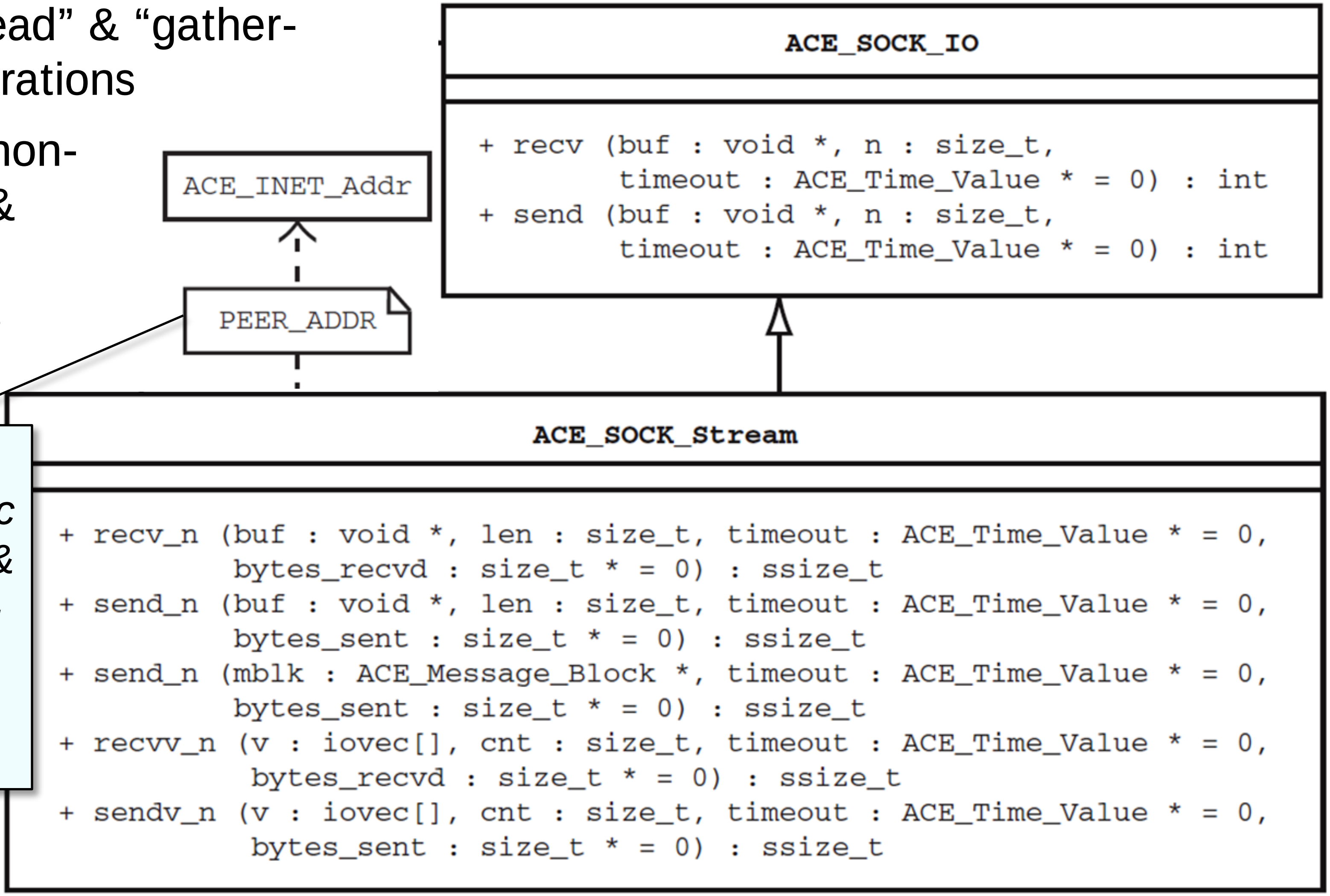
- ACE Socket wrapper façade structure reflects the domain of networked IPC & provide the following capabilities:
  - **ACE_SOCK_*** classes encapsulate Internet-domain Socket API functionality
  - **ACE_LSOCK_*** classes encapsulate Local Socket (UNIX-domain) API functionality



ACE also has wrapper facades for unicast, multicast, broadcast datagrams

The ACE_SOCK_Stream Class

- Encapsulates data transfer mechanisms of data-mode sockets to support:
  - Sending & receiving up to n bytes or exactly n bytes
  - "Scatter-read" & "gather-write" operations
  - Blocking, non-blocking, & timed I/O operations



C++ traits support generic programming & enable whole-sale replace-ment of IPC functionality

Sidebar: Traits for ACE IPC Wrapper Facades

- ACE Socket wrapper facades use traits to define the following dependencies in a generic manner
  - **`PEER_ADDR`** – This trait defines the **`ACE_INET_Addr`** class associated with the ACE Socket Wrapper Façade
  - **`PEER_STREAM`** – This trait defines the **`ACE_SOCK_Stream`** data transfer class associated with the **`ACE_SOCK_Acceptor`** & **`ACE_SOCK_Connector`** factories

Sidebar: Traits for ACE IPC Wrapper Facades

- ACE Socket wrapper facades use traits to define the following dependencies in a generic manner
  - **PEER_ADDR** – This trait defines the **ACE_INET_Addr** class associated with the ACE Socket Wrapper Façade
  - **PEER_STREAM** – This trait defines the **ACE_SOCK_Stream** data transfer class associated with the **ACE_SOCK_Acceptor** & **ACE_SOCK_Connector** factories
- e.g., traits for Socket & TLI wrappers have different dependencies

```
class ACE_SOCK_Stream {
public:
  typedef ACE_INET_Addr
          PEER_ADDR;
  // ...
```

```
class ACE_TLI_Stream {
public:
  typedef ACE_INET_Addr
          PEER_ADDR;
  // ...
```

Sidebar: Traits for ACE IPC Wrapper Facades

- ACE Socket wrapper facades use traits to define the following dependencies in a generic manner
  - **PEER_ADDR** – This trait defines the **ACE_INET_Addr** class associated with the ACE Socket Wrapper Façade
  - **PEER_STREAM** – This trait defines the **ACE_SOCK_Stream** data transfer class associated with the **ACE_SOCK_Acceptor** & **ACE_SOCK_Connector** factories
- e.g., traits for Socket & TLI wrappers have different dependencies

```
class ACE_SOCK_Stream {
public:
  typedef ACE_INET_Addr
          PEER_ADDR;
  // ...
```

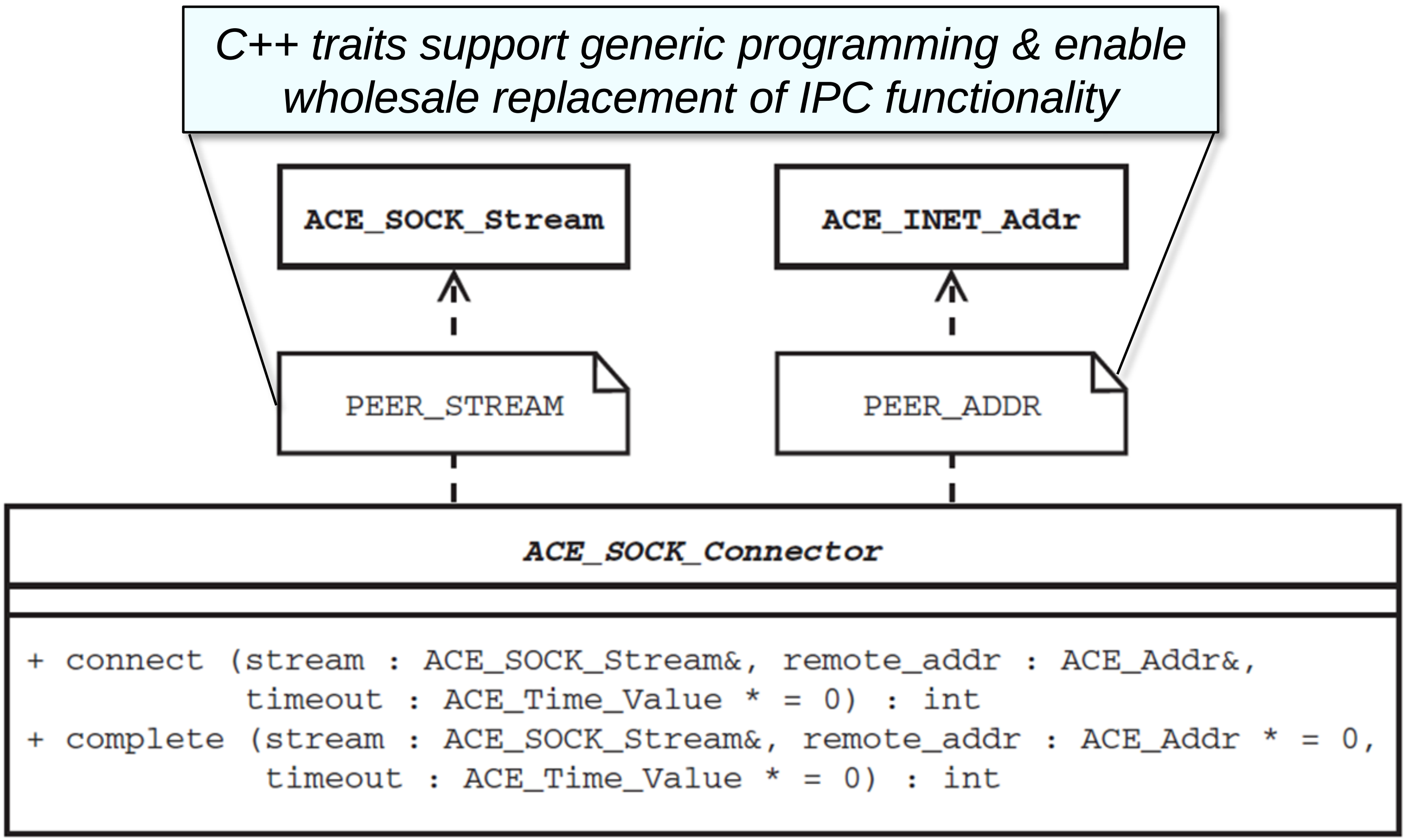
```
class ACE_TLI_Stream {
public:
  typedef ACE_INET_Addr
          PEER_ADDR;
  // ...
```

- Traits make it easier to write generic algorithms & containers

See erdani.com/publications/traits.html for more on C++ traits

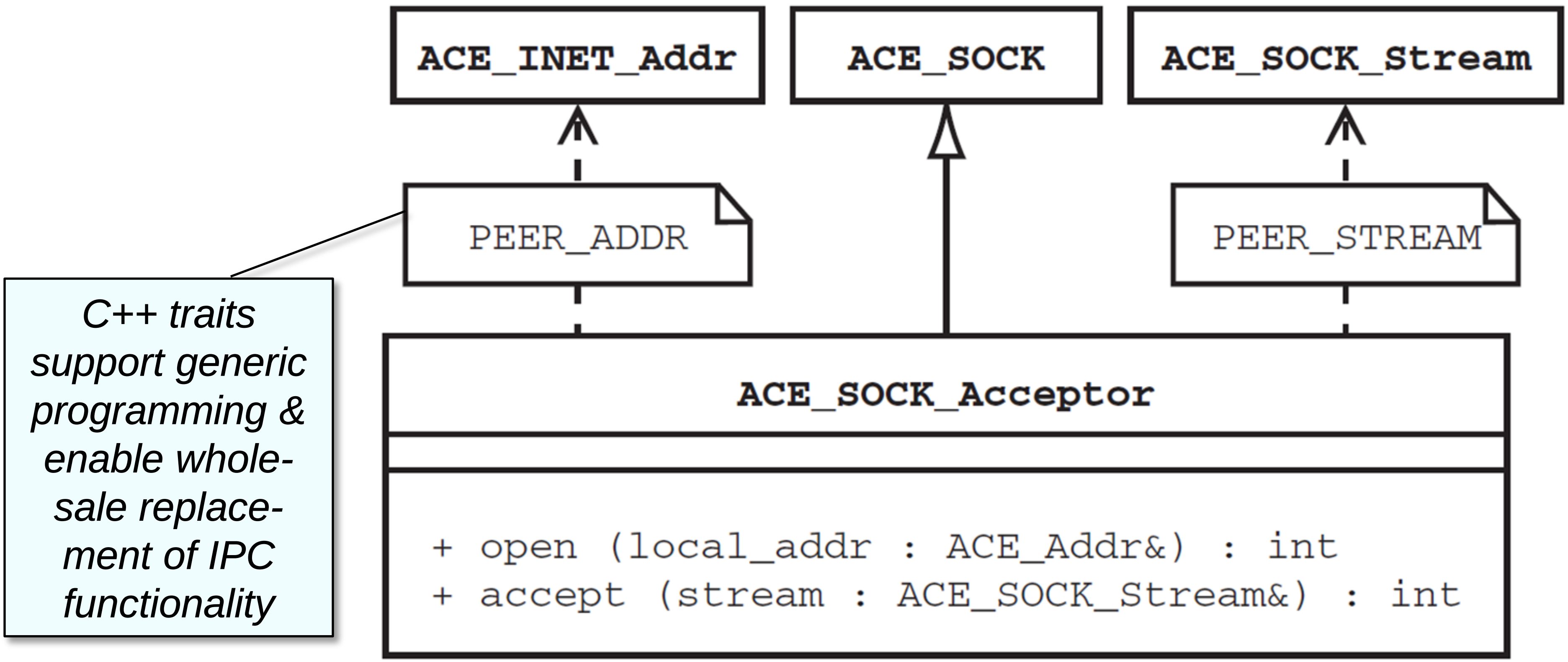
The ACE_SOCK_Connector Class

- A factory that *actively* establishes a new endpoint of communication by
  - Initiating a connection with a peer acceptor & after connection is established then initialize an **`ACE_SOCK_Stream`** object
    - Connection initiation can be *blocking*, *non-blocking*, or *timed*



The ACE_SOCK_Acceptor Class

- A factory that *passively* establishes a new endpoint of communication by
  - Accepting a connection from a peer connector & then initializing an **`ACE_SOCK_Stream`** object after the connection is established
    - Connections accepts can be *blocking*, *non-blocking*, or *timed*



Summary

- There is a confusing asymmetry in the Socket API between (1) connection roles & (2) socket modes
  - e.g., an application may accidentally call **send()** or **recv()** on a data-mode socket handle before it's connected

```
int buggy_echo_client (u_short port_num, const char *s)
{
  int handle = socket (PF_UNIX, SOCK_DGRAM, 0);

  send (handle, s, strlen (s) + 1);

  sockaddr_in s_addr;
  memset (&s_addr, 0, sizeof s_addr);
  s_addr.sin_family = AF_INET;
  s_addr.sin_port = htons (port_num);
  connect (handle, (sockaddr *) &s_addr, sizeof s_addr);
}
```

Operations called in wrong order, but C Socket APIs can't prevent this

Summary

- There is a confusing asymmetry in the Socket API between (1) connection roles & (2) socket modes
  - e.g., an application may also accidentally call **`send()`** or **`recv()`** on a listen-mode socket handle

```
int buggy_echo_server (u_short port_num) {
  int a_socket = socket (PF_UNIX, SOCK_DGRAM, 0);
  listen (a_socket), 5);
  ...
  int handle = accept (a_socket, 0, 0);

  for (char buf[BUFSIZ];;) {
    ssize_t n = read (a_socket, buf, sizeof buf);
    if (n <= 0) break;
    write (handle, buf, n);
  ...
```

Reading from acceptor handle!

Summary

- There is a confusing asymmetry in the Socket API between (1) connection roles & (2) socket modes
  - e.g., an application may also accidentally call **`send()`** or **`recv()`** on a listen-mode socket handle

```
int buggy_echo_server (u_short port_num) {
  int a_socket = socket (PF_UNIX, SOCK_DGRAM, 0);
  listen (a_socket), 5);
  ...
  int handle = accept (a_socket, 0, 0);

  for (char buf[BUFSIZ];;) {
    ssize_t n = read (a_socket, buf, sizeof buf);
    if (n <= 0) break;
    write (handle, buf, n);
  ...
```

Reading from acceptor handle!

- The ACE Socket wrapper facades ensure these errors are detected by the compiler, rather than at runtime

Patterns & Frameworks for Service Access & Communication: Part 4

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

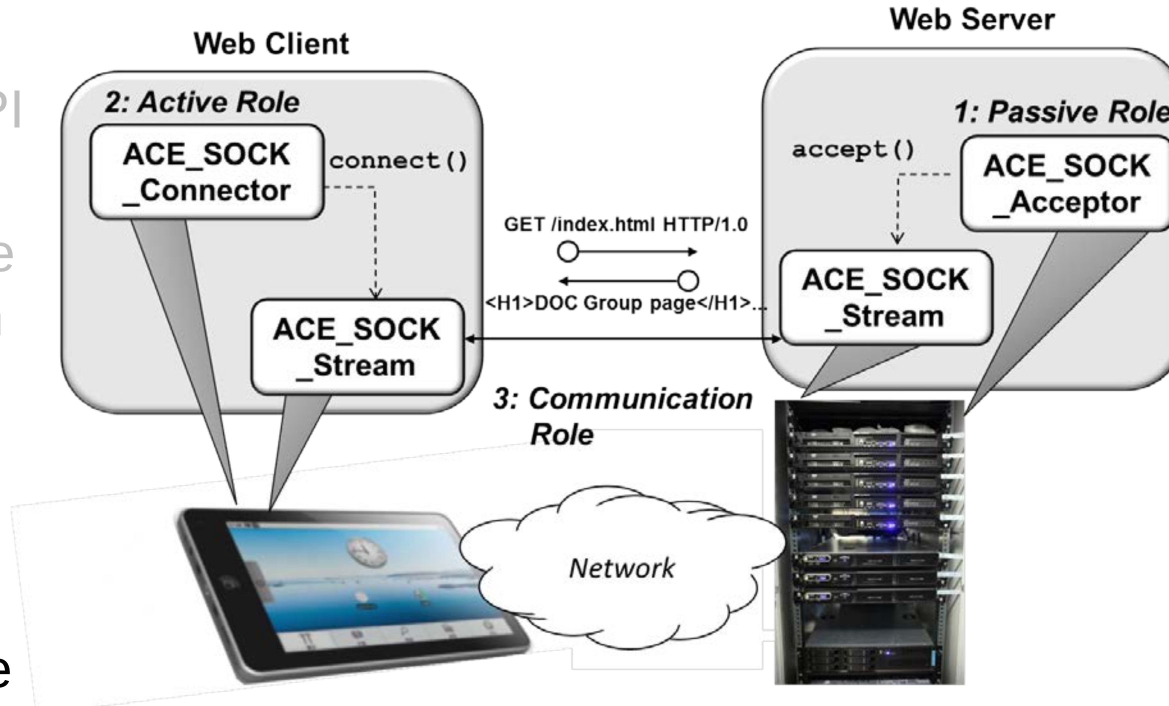
Professor of Computer Science

Institute for Software Integrated Systems

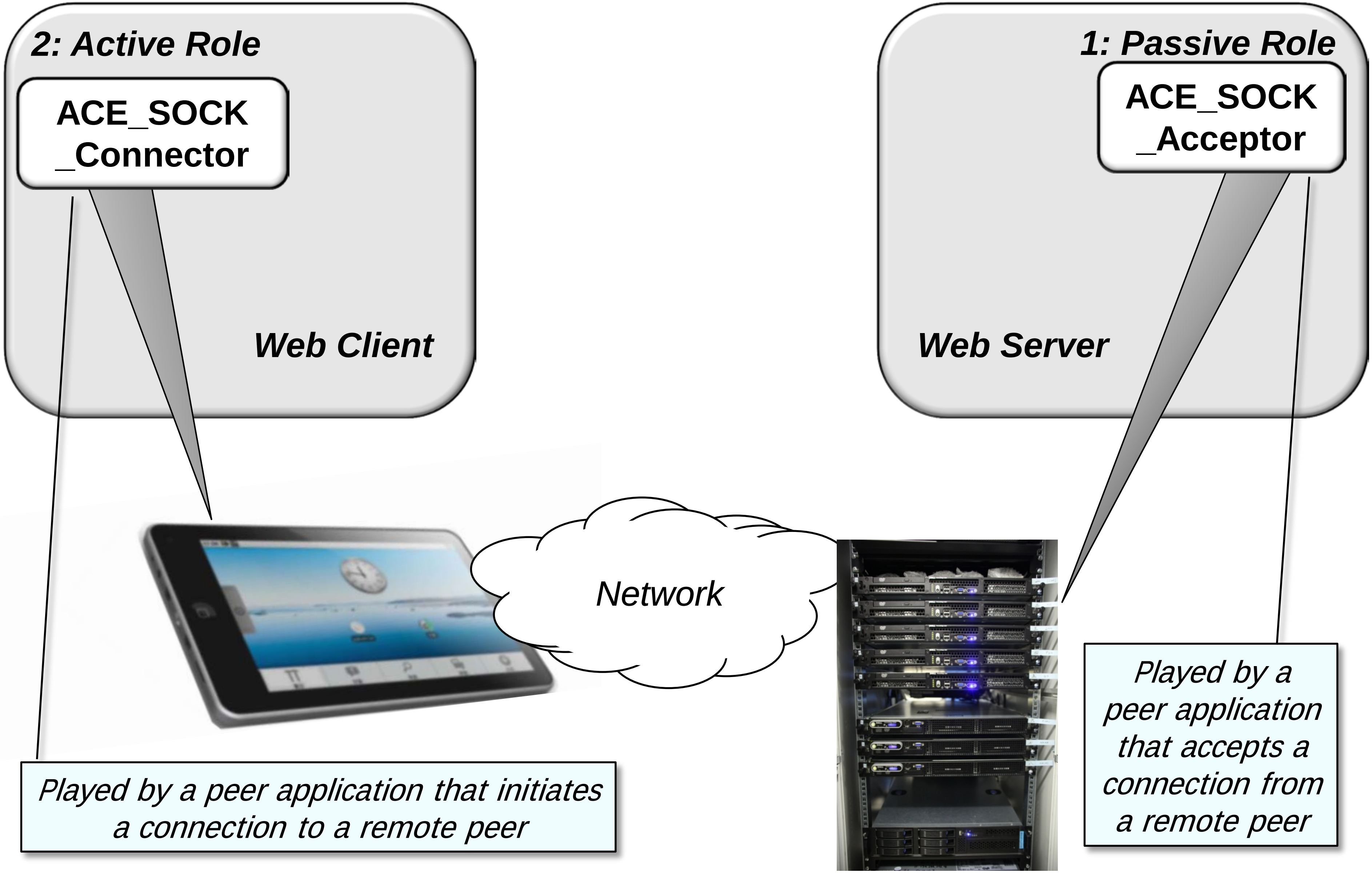
Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

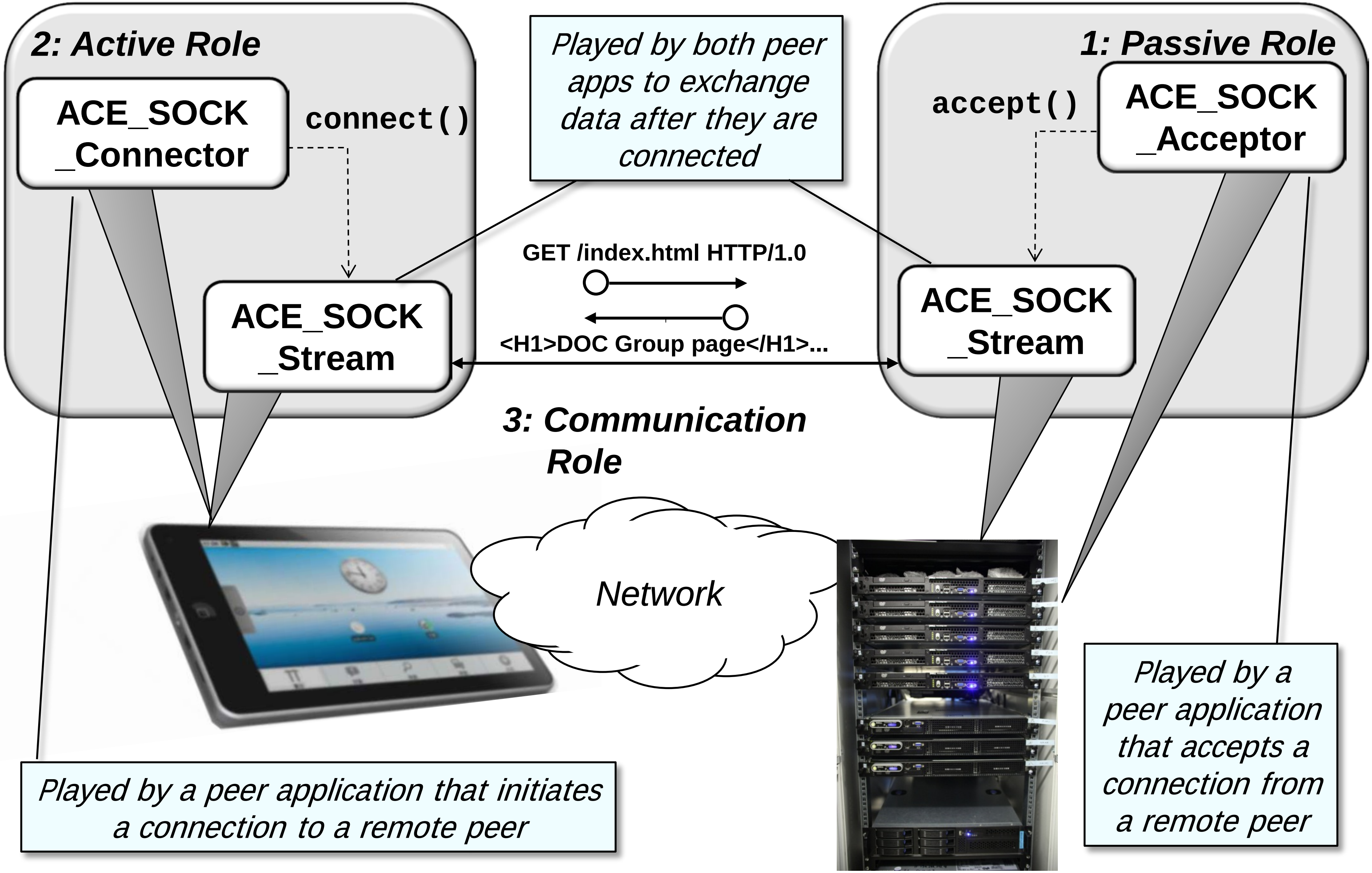
- Summarize the accidental complexities with Socket API
- Describe how the *Wrapper Façade* pattern can alleviate accidental complexities with C APIs for JAWS
- Describe the ACE C++ Socket wrapper façades
- Apply the ACE C++ Socket wrapper facades to a simple iterative web client/server



ACE Socket Wrapper Facades for Web Client/Server



ACE Socket Wrapper Facades for Web Client/Server



Using ACE_SOCK_Connector for a Web Client

- An **ACE_SOCK_Connector** can connect a client app to a web server

Process command-line args

```
int main (int argc, char *argv[]) {
  str::string pathname = argc > 1 ? argv[1] : "/index.html";
  std::string server_hostname =
    argc > 2 ? argv[2] : "www.dre.vanderbilt.edu";

  typedef ACE_SOCK_Connector CONNECTOR;

  CONNECTOR connector;
  CONNECTOR::PEER_STREAM peer;
  CONNECTOR::PEER_ADDR peer_addr (80, server_hostname.c_str());
  ACE_Time_Value timeout (10); // 10 second timeout.


  if (connector.connect (peer, peer_addr, &timeout) == -1)
    return 1;
  ... // Defined later
```

Instantiate the connector, data transfer, & address objects

Block up to 10 seconds to establish connection or detect failure

Using ACE_SOCK_Stream for a Web Server

- An **ACE_SOCK_Stream** can send & receive data to & from a web server

Initialize the `iovec` vector for gather-write I/O

```
// ... Continue from ACE_SOCK_Connector code shown above ...
char buf[BUFSIZ];
iovec iov[3];
iov[0].iov_base = (char *) "GET ";
iov[0].iov_len = 4; // Length of "GET ".
iov[1].iov_base = (char *) pathname.c_str();
iov[1].iov_len = pathname.length();
iov[2].iov_base = (char *) " HTTP/1.0\r\n\r\n";
iov[2].iov_len = 13; // Length of " HTTP/1.0\r\n\r\n";

if (peer.sendv_n (iov, 3) == -1) return 1;
```

Perform blocking gather-write on `ACE_SOCK_Stream`

```
ACE_Time_Value timeout (10); // 10 second timeout
```

Perform timed read on `ACE_SOCK_Stream`

```
for (ssize_t n;
     (n = peer.recv (buf, sizeof buf, &timeout)) > 0; )
  ACE::write_n (ACE_STDOUT, buf, n);
...
```

Using ACE_SOCK_Acceptor for a Web Server

- An **ACE_SOCK_Acceptor** & **ACE_SOCK_Stream** can accept connections & send/receive data to/from a web client

```
int main (){
  typedef ACE_SOCK_Acceptor ACCEPTOR;
  ACCEPTOR::PEER_ADDR server_addr (80);
  ACCEPTOR acceptor;

  if (acceptor.open (server_addr) == -1) return 1;

  for (ACCEPTOR::PEER_STREAM peer;;) {
    if (acceptor.accept (peer) == -1) return 1;

    std::string pathname (get_pathname (peer));
    ACE_Mem_Map mapped_file (pathname.c_str ());

    if (peer.send_n (mapped_file.addr (),
                     mapped_file.size ()) == -1)
      return 1;
    ...
```

Instantiate the acceptor, data transfer, & address objects & listen for connections on port 80

Accept a new connection

Memory map requested file

Return requested data (could use timed send to avoid blocking for extended duration)

Summary

- The ACE Socket wrapper facades resolve the following problems with the Socket API :
  - **Error-Prone** – The ACE Socket wrapper facades operations are type-safe

```
int a_handle = socket(...);
listen(a_handle);
...
read(a_handle, ...);

int d_handle = accept(a_handle, ...);
accept(d_handle, ...);
```

This erroneous code compiles!

Summary

- The ACE Socket wrapper facades resolve the following problems with the Socket API :
  - **Error-Prone** – The ACE Socket wrapper facades operations are type-safe

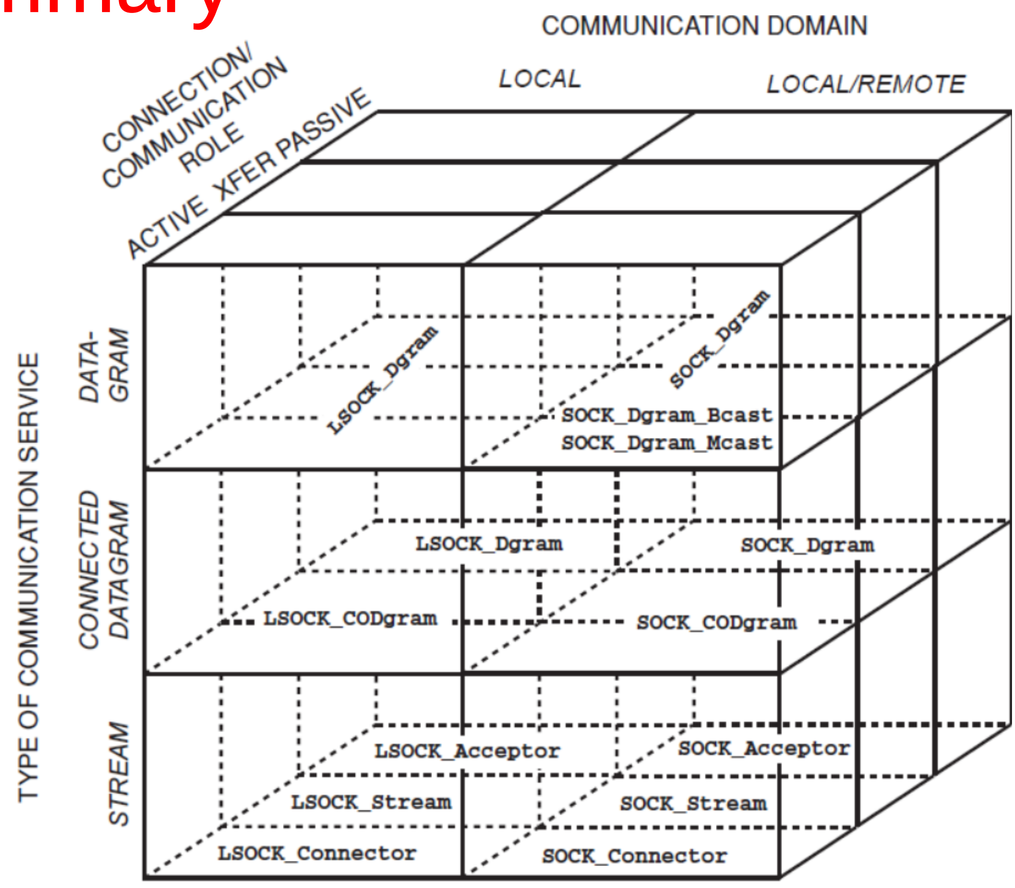
```
int a_handle = socket(...);
listen(a_handle);
...
read(a_handle, ...);

int d_handle = accept(a_handle, ...);
accept(d_handle, ...);
```

```
ACE_SOCK_Acceptor acceptor;
ACE_SOCK_Stream stream;
...
acceptor.read(...);
...
stream.accept(...);
```

This erroneous code *won't* compile!

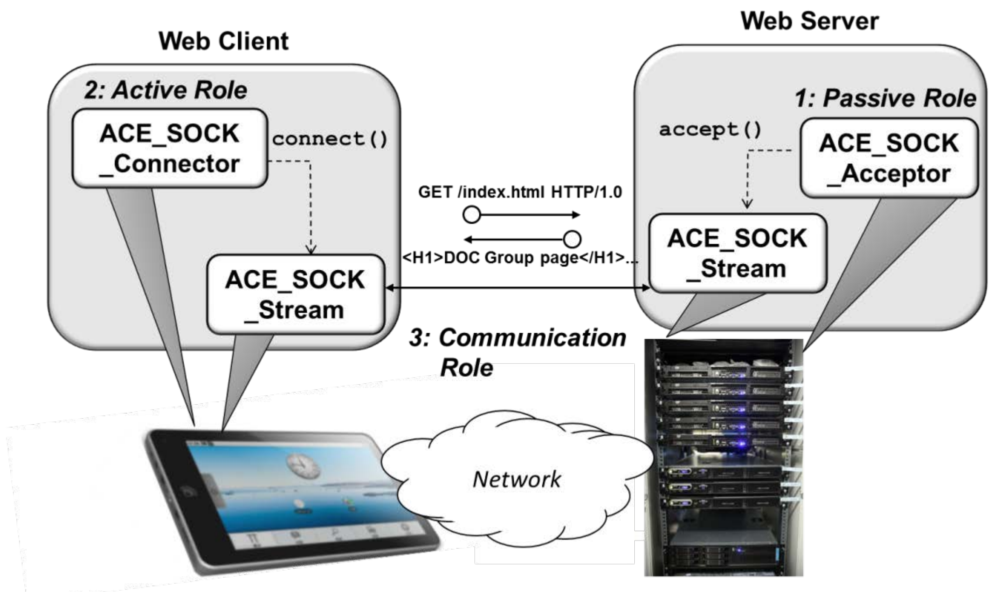
Summary

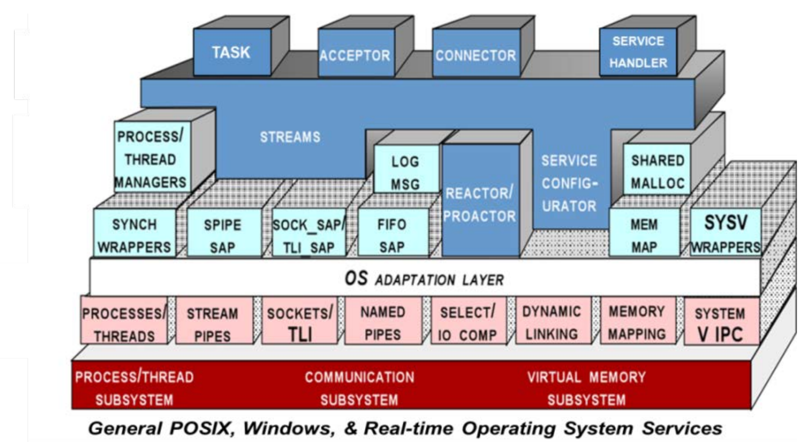
- The ACE Socket wrapper facades resolve the following problems with the Socket API :
  - **Error-Prone** – The ACE Socket wrapper facades operations are type-safe
  - **Overly complex** – "Surface area" is minimized by clustering wrapper facades into classes for actively connecting, passively accepting, & transferring data



Summary

- The ACE Socket wrapper facades resolve the following problems with the Socket API :
  - **Error-Prone** – The ACE Socket wrapper facades operations are type-safe
  - **Overly complex** – "Surface area" is minimized by clustering wrapper facades into classes for actively connecting, passively accepting, & transferring data
  - **Non-portable & non-uniform** – The client app & web server source code compiles & runs correctly & efficiently on all platforms that ACE supports





We need more than just wrapper facades to develop effective web servers!