C++ Wrappers for Network Programming Interfaces

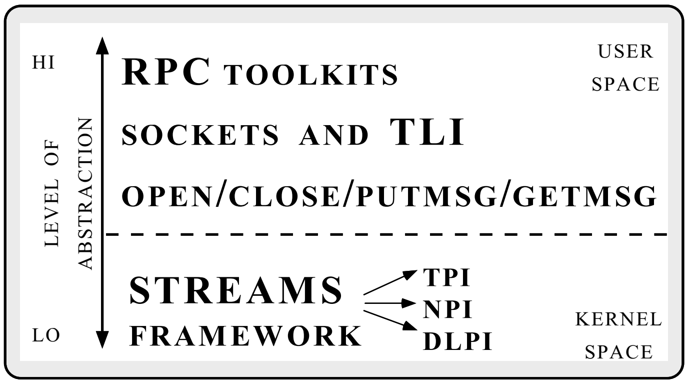
Douglas C. Schmidt

Washington University, St. Louis

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

schmidt@cs.wustl.edu

Introduction



- Network programming may be performed at various levels of abstraction

Introduction (cont'd)

- Choosing the appropriate level involves many factors:

1. *Performance*
   - Higher levels can be less efficient
2. *Functionality*
   - Certain features (*e.g.*, multicasting) are only available at certain levels of abstraction
3. *Ease of programming*
   - RPC-based toolkits are typically easier to use for conventional applications
4. *Portability*
   - The socket API is generally the most portable...

RPC-based Toolkits

- RPC-based toolkits help simplify certain types of distributed applications
  - *e.g.*, "request-response" client/server interactions
- This allows developers to work at higher levels of abstraction by shielding them from details of low-level network IPC mechanisms
  - *e.g.*, sockets, TLI, and STREAMS
- Examples include Sun RPC, DCE, CORBA, DCOM

RPC-based Toolkits (cont'd)

- RPC *stub compilers* automatically generate code to perform presentation layer conversions
  - *e.g.*, network byte-ordering and parameter marshaling

- In addition, RPC runtime library routines handle

1. *Network addressing and remote service identification*
2. *Service registration, port monitoring, and service dispatching*
3. *Authentication and security*
4. *Transport protocol selection and request delivery*
5. *Reliable call semantics*

RPC Limitations

- However, applications may need to use lower-level IPC mechanisms directly to meet certain requirements:

1. *Performance*
2. *Functionality*
3. *Portability*

- For example, application requirements involving *high-bandwidth*, *long-duration*, *bi-directional*, *uninterpreted byte-stream* transfer may not be suitable for RPC
  - *e.g.*, file transfer, remote login, voice, video

RPC Limitations (cont'd)

- Compared with direct use of sockets and TLI, RPC may be much less efficient due to:

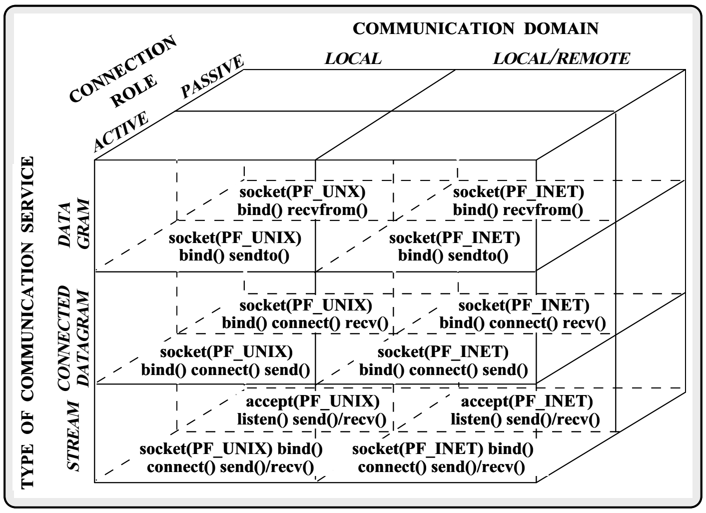
1. Presentation conversion processing and excessive data copying
2. Synchronous client-side and server-side stub behavior
3. Stop-and-wait flow control
4. Non-adaptive retransmission timer schemes
5. Non-optimized demultiplexing

Standard APIs for Network IPC

- Sockets and System V TLI are two widely available APIs that allow applications to access lower-level local and remote IPC mechanisms

- Each API mediates access to connection-oriented and connectionless communication services for multiple "protocol families," *e.g.*,
  - TCP/IP
  - XNS and Novell IPX NetWare protocols
  - UNIX domain sockets
  - OSI protocols

Socket Taxonomy



Common Problems with Existing Network Programming Interfaces

1. *Lack of type-safety*

2. *Steep learning curve*

3. *Portability problems*

Lack of Type-safety

- Integer I/O descriptors are not amenable to strong type checking at compile-time
  - *e.g.*, the following code contains many subtle (and all too common) bugs:

```
int buggy_echo_server (u_short port_num)
{
  // Error checking omitted.
  sockaddr_in s_addr;

  int s_fd = socket (PF_UNIX, SOCK_DGRAM, 0);
  s_addr.sin_family = AF_INET;
  s_addr.sin_port = port_num;
  s_addr.sin_addr.s_addr = INADDR_ANY;
  bind (s_fd, (sockaddr *) &s_addr, sizeof s_addr);

  int n_fd = accept (s_fd, 0, 0);
  for (;;) {
    char buf[BUFSIZ];
    ssize_t n = read (s_fd, buf, sizeof buf);
    if (n <= 0) break;
    write (n_fd, buf, n);
  }
}
```

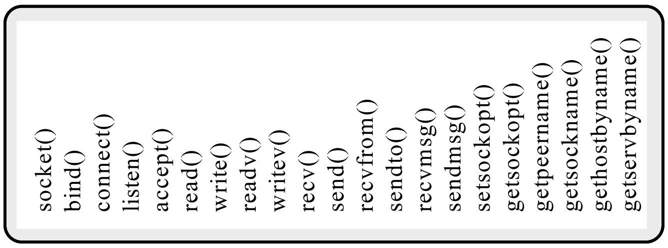

Steep Learning Curve

- Many socket/TLI API routines have complex semantics that must support:

1. Multiple protocol families and address families
   - *e.g.*, TCP, UNIX domain, OSI, XNS, etc.

2. Infrequently used features, *e.g.*,
   - Broadcasting/multicasting
   - Passing open file descriptors
   - Urgent data delivery and reception
   - Asynch I/O, non-blocking I/O, I/O-based and timer-based event multiplexing

Steep Learning Curve (cont'd)

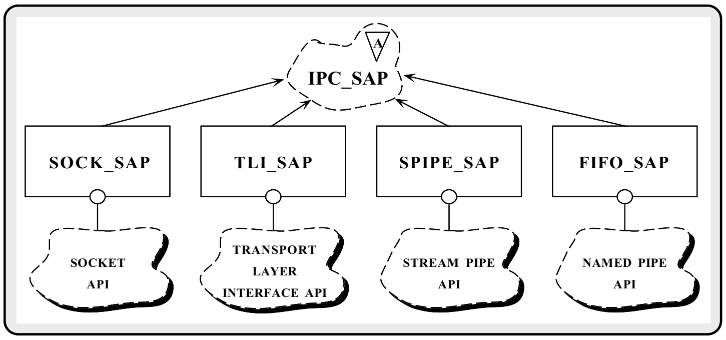


- Note that this API is *linear* rather than *hierarchical*
  - Thus, it gives no hints on how to use it correctly
- In addition, there is no consistency among names...

Portability Problems

- Having multiple "standards" (*i.e.*, sockets vs. TLI) makes portability difficult, *e.g.*,
  - May require conditional compilation
  - In addition, important related routines are not included in POSIX standards
    - ▷ *e.g.*, `select()` and/or `poll()` event multiplexing...
- Portability between UNIX and Win32 Sockets is problematic, *e.g.*,
  - *Header files*
  - *Error numbers*
  - *Descriptor types*
  - *Semantics*
  - *I/O controls and options*

The C++ Wrapper Solution

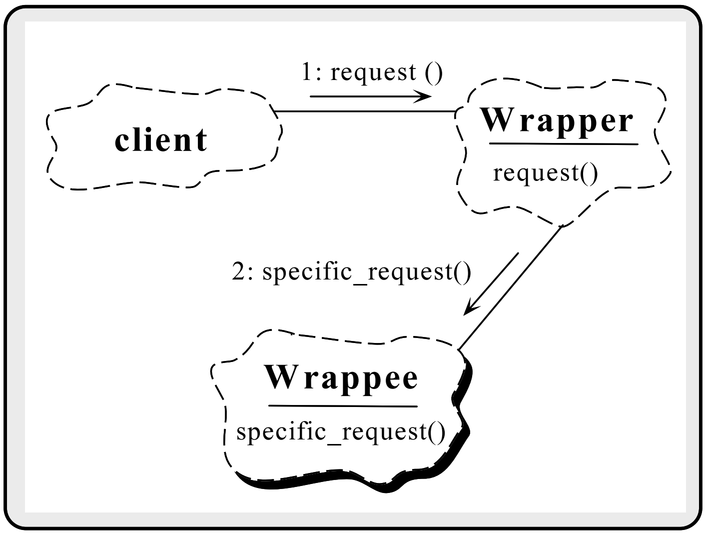


- A: `IPC_SAP` are "wrappers" that encapsulate network programming interfaces like sockets and TLI
  - This is an example of the "Wrapper pattern"

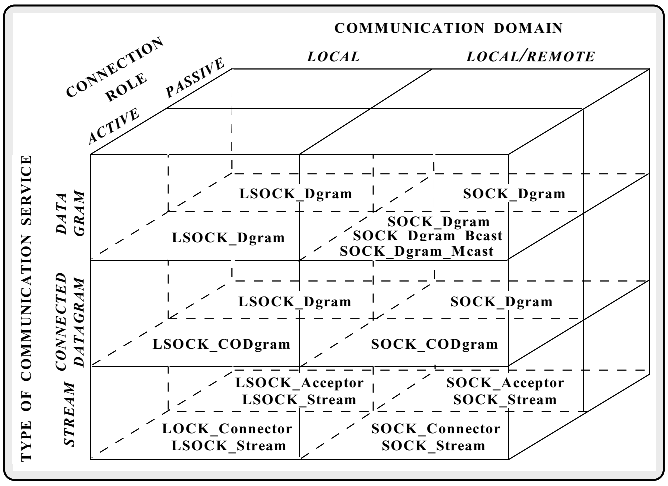
The Wrapper Pattern

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

Structure of the Wrapper Pattern



SOCK_SAP Class Structure



SOCK_SAP Factory Class Interfaces

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

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

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

  SOCK_Acceptor (const INET_Addr &local_addr);

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


SOCK_SAP Stream and Addressing Class Interfaces

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

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

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

OO Design Interlude

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

- A: For the same reasons that Acceptor and Connector are decoupled from Svc_Handler, *e.g.*,
  - A `SOCK_Stream` is only responsible for data transfer
    - ▷ Regardless of whether the connection is established passively or actively
  - This ensures that the `SOCK*` components are never used incorrectly...
    - ▷ *e.g.*, you can't accidentally `read` or `write` on `SOCK_Connectors` or `SOCK_Acceptors`, etc.

Socket vs. SOCK_SAP Examples

- SOCK_SAP `echo_server` implementation:

```
int echo_server (u_short port_num)
{
  // Error handling omitted.
  INET_Addr my_addr (port_num);
  SOCK_Acceptor acceptor (my_addr);
  SOCK_Stream new_stream;

  acceptor.accept (new_stream);

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


SOCK_SAP Revision of Echo Server

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

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

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

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

  for (;;)
  {
    char buf[BUFSIZ];
    ssize_t n = stream.recv (buf, sizeof buf);
    stream.send_n (buf, n);
  }
}
// ...
echo_server<SOCK_Acceptor> (port);
```

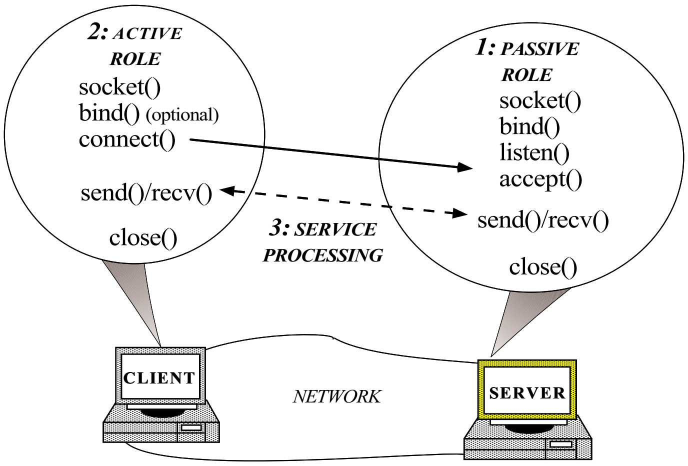

Socket vs. SOCK_SAP Examples (cont'd)

- The following 4 slides illustrate differences between using the Socket interface vs. the SOCK_SAP API

- The example is a simple client/server "network pipe" application that
  1. Starts an "iterative daemon" at a well-known port on a server host
  2. Client connects to the server daemon and then transmits its standard input stream to the server
  3. The server prints the contents to its standard output

- Note, the server portion of the "network pipe" application may actually run either locally or remotely...

Network Pipe with Sockets



Socket vs. SOCK_SAP Examples (cont'd)

- *e.g.*,

```
% ./server &
% echo "hello world" | ./client localhost
client localhost.cs.wustl.edu%: hello world
```

- Note that the SOCK_SAP example:
  1. Requires *much* less code (about 1/2 to 2/3 less)
  2. Provides greater clarity and less potential for errors
  3. Operates at no loss of efficiency

Socket vs. SOCK_SAP Examples (cont'd)

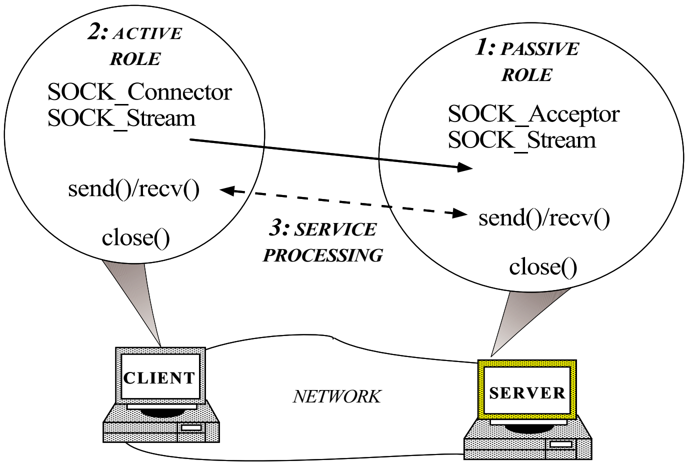
- BSD socket client

```
#define PORT_NUM 10000

int
main (int argc, char *argv[]) {
  struct sockaddr_in saddr;
  struct hostent *hp;
  char *host = argc > 1 ? argv[1] : "ics.uci.edu";
  u_short port_num = argc > 2
    htons (argc > 2 ? atoi (argv[2]) : PORT_NUM);
  char buf[BUFSIZ];
  int s_fd;
  int w_bytes;
  int r_bytes;
  int n;

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

  /* Determine IP address of the server */
  hp = gethostbyname (host);
```



```
  /* Set up the address information to contact the server */
  memset ((void *) &saddr, 0, sizeof saddr);
  saddr.sin_family = AF_INET;
  saddr.sin_port = port_num;
  memcpy (&saddr.sin_addr, hp->h_addr, hp->h_length);

  /* Establish connection with remote server */
  connect (s_fd, (struct sockaddr *) &saddr,
           sizeof saddr);

  /* Send data to server (correctly handles
     "incomplete writes" due to flow control) */

  while ((r_bytes = read (0, buf, sizeof buf)) > 0)
    for (w_bytes = 0; w_bytes < r_bytes; w_bytes += n)
      n = write (s_fd, buf + w_bytes, r_bytes - w_bytes);

  /* Explicitly close the connection */
  close (s_fd);
  return 0;
}
```

Network Pipe with SOCK_SAP



Socket vs. SOCK_SAP Examples (cont'd)

- SOCK_SAP Client

```
const u_short PORT_NUM = 10000;
int main (int argc, char *argv[])
{
  char buf[BUFSIZ];
  char *host = argc > 1 ? argv[1] : "ics.uci.edu";
  u_short port_num =
    htons (argc > 2 ? atoi (argv[2]) : PORT_NUM);

  INET_Addr server_addr (port_num, host);
  SOCK_Stream cli_stream;
  SOCK_Connector connector.

  // Establish the connection with server.
  connector.connect (cli_stream, server_addr);
```



```
  // Send data to server (correctly handles
  // "incomplete writes").

  for (;;) {
    ssize_t r_bytes = read (0, buf, sizeof buf);
    cli_stream.send_n (buf, r_bytes);
  }

  // Explicitly close the connection.
  cli_stream.close ();
  return 0;
}
```


Socket vs. SOCK_SAP Examples (cont'd)

- BSD socket server

```
#define PORT_NUM 10000
int
main (int argc, char *argv[])
{
  u_short port_num =
    htons (argc > 1 ? atoi (argv[1]) : PORT_NUM);
  struct sockaddr_in saddr;
  int s_fd, n_fd;

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

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

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

  /* Make endpoint listen for service requests */
  listen (s_fd, 5);
```

```
  /* Performs the iterative server activities */
  for (;;) {
    char buf[BUFSIZ];
    struct sockaddr_in cli_addr;
    int r_bytes, cli_addr_len = sizeof cli_addr;
    struct hostent *hp;

    /* Create a new endpoint of communication */
    while ((n_fd = accept (s_fd, (struct sockaddr *)
        &cli_addr, &cli_addr_len)) == -1 && errno == EINTR)
      continue;

    if (n_fd == -1)
      continue;
    hp = gethostbyaddr ((char *) &cli_addr.sin_addr,
                        cli_addr_len, AF_INET);
    printf ("client %s: ", hp->h_name), fflush (stdout);

    /* Read data from client (terminate on error) */
    while ((r_bytes = read (n_fd, buf, sizeof buf)) > 0)
      write (1, buf, r_bytes);

    /* Close the new endpoint
       (listening endpoint remains open) */
    close (n_fd);
  }
  /* NOTREACHED */
}
```


Socket vs. SOCK_SAP Examples (cont'd)

- SOCK_SAP server

```
const u_short PORT_NUM = 10000;

// SOCK_SAP Server.

int
main (int argc, char *argv[])
{
  u_short port_num =
    argc == 1 ? PORT_NUM : ::atoi (argv[1]);

  // Create a server.
  SOCK_Acceptor acceptor ((INET_Addr) port_num);
  SOCK_Stream new_stream;
  INET_Addr cli_addr;
```



```
  // Performs the iterative server activities.

  for (;;) {
    char buf[BUFSIZ];

    // Create a new SOCK_Stream endpoint (note
    //  automatic restart if errno == EINTR).
    acceptor.accept (new_stream, &cli_addr);

    printf ("client %s: ", cli_addr.get_host_name ());
    fflush (stdout);

    // Read data from client (terminate on error).

    for (;;) {
      ssize_t r_bytes;
      r_bytes = new_stream.recv (buf, sizeof buf);
      if (r_bytes <= 0) break;
      write (1, buf, r_bytes);
    }
    // Close new endpoint (listening
    // endpoint stays open).
    new_stream.close ();
  /* NOTREACHED */
}
```


ACE C++ Wrapper Design Principles

- The following principles applied throughout the ACE C++ wrappers:
  - *Enforce typesafety at compile-time*
  - *Allow controlled violations of typesafety*
  - *Simplify for the common case*
  - *Replace one-dimensional interfaces with hierarchical class categories*
  - *Enhance portability with parameterized types*
  - *Inline performance critical methods*
  - *Define auxiliary classes to hide error-prone details*

Enforce Typesafety at Compile-Time

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

```
int s_sd = socket (PF_INET, SOCK_STREAM, 0);
// ...
bind (s_sd, ...); // Bind address.
listen (s_sd); // Make a passive-mode socket.

// Error not detected until run-time.
read (s_sd, buf, sizeof buf);
```

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

```
SOCK_Acceptor acceptor (port);

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


Allow Controlled Violations of Typesafety

- *Make it easy to use SOCK_SAP correctly, hard to use it incorrectly, but not impossible to use it in ways the class designers did not anticipate*
  - *e.g.*, it may be necessary to retrieve the underlying socket descriptor

```
fd_set rd_sds;

FD_ZERO (&rd_sds);

FD_SET (acceptor.get_handle (), &rd_sds);

select (acceptor.get_handle () + 1, &rd_sds, 0, 0, 0);
```


Simplify for the Common Case

- *Supply default parameters for common method arguments*

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

- The result is extremely concise for the common case:

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


Simplify for the Common Case (cont'd)

- *Define parsimonious interfaces*
  - *e.g.*, use LSOCK to pass socket descriptors:

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

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

  - versus

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

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

Simplify for the Common Case (cont'd)

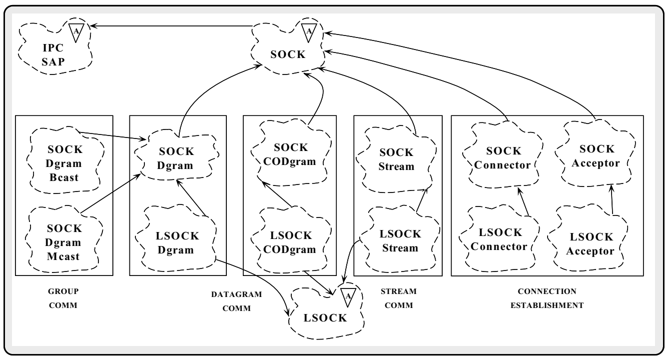
- *Combine multiple operations into a single operation*
  - *e.g.*, creating a conventional passive-mode socket requires multiple calls:

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

  - `SOCK_Acceptor` combines this into a single operation:

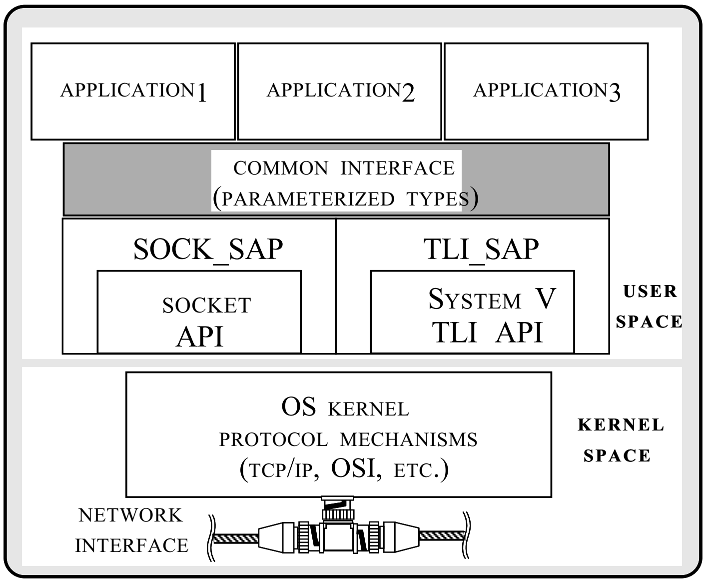
```
SOCK_Acceptor acceptor ((INET_Addr) port);
```


Create Hierarchical Class Categories



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

Enhance Portability with Parameterized Types



Enhance Portability with Parameterized Types (cont'd)

- Switching wholesale between sockets and TLI simply requires instantiating a different C++ wrapper, *e.g.*,

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

int main (void)
{
  // ...

  // Invoke the echo_server with appropriate
  // network programming interfaces.
  echo_server<PEER_ACCEPTOR> (port);
}
```

Inline Performance Critical Methods

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

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

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


Define Auxiliary Classes to Hide Error-Prone Details

- Standard C socket addressing is awkward and error-prone
  - *e.g.*, easy to neglect to zero-out a **sockaddr_in** or convert port numbers to network byte-order, etc.

- IPC_SAP defines addressing classes to handle these details

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


Summary of IPC_SAP OOD/OOP

- "Domain analysis" identifies and groups related classes of existing API behavior
  - Example "subdomains" for IPC_SAP include
    1. *Local context management and options, data transfer, connection/termination handling, etc.*
    2. *Datagrams vs. streams*
    3. *Local vs. remote addressing*
    4. *Client vs. server*
  - These relationships are directly reflected in the IPC_SAP inheritance hierarchy

Summary of IPC_SAP OOD/OOP (cont'd)

- IPC_SAP is designed to maximize reusability and sharing of components
  - Inheritance is used to *factor out* commonality and *decouple* variation *e.g.*,
    - ▷ Push common services "upwards" in the inheritance hierarchy
    - ▷ Factor out variations in client/server portions of socket API
    - ▷ Decouple datagram vs. stream operations, local vs. remote, etc.
  - Inheritance also supports "functional subsetting"
    - ▷ *e.g.*, passing open file descriptors...

Summary of IPC_SAP OOD/OOP (cont'd)

- Performance improvements techniques used in IPC_SAP include:
  - Inline functions are used to avoid additional subroutine call penalties
  - Dynamic binding is used sparingly to reduce time/space overhead
    - *i.e.*, virtually eliminated for "fast path"
      - *e.g.*, `recv/send`
- Note the difference between the *composition* vs. *decomposition/composition* aspects in design complexity
  - *i.e.*, IPC_SAP is primarily an exercise in *composition*, since the basic components already exist
  - Most complex OO designs involve both aspects...
    - *e.g.*, the ACE ASX, Service Configurator, and Reactor frameworks, etc.

Concluding Remarks

- Defining C++ wrappers for existing OS APIs simplifies the development of correct, portable, and extensible applications
  - C++ inline functions ensure that performance isn't sacrificed
- ACE SOCK_SAP is an example of applying C++ wrappers to standard UNIX and Windows NT network programming interfaces
  - *e.g.*, sockets, TLI, named pipes, STREAM pipes, etc.
- ACE wrappers can be integrated conveniently with CORBA to provide a flexible, high-performance network programming mechanism

Obtaining ACE

- The ADAPTIVE Communication Environment (ACE) is an OO toolkit designed according to key network programming patterns
- All source code for ACE is freely available
  - Anonymously ftp to `wuarchive.wustl.edu`
  - Transfer the files `/languages/c++/ACE/*.gz` and `gnu/ACE-documentation/*.gz`
- Mailing list
  - ace-users@cs.wustl.edu
  - ace-users-request@cs.wustl.edu
- WWW URL
  - http://www.cs.wustl.edu/~schmidt/ACE.html