Alternative Techniques for Designing Concurrent Server Daemons

Douglas C. Schmidt
Washington University, St. Louis

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

Motivation

- Network applications (particularly servers) often handle different types of events simultaneously, e.g.,
  1. I/O events
     - e.g., input, output, exceptions corresponding to interactions with clients
  2. Time-related events
     - e.g., handle timeouts and retransmissions

- Connection-oriented servers often identified clients internally via distinct I/O handles
  - Handles are internal IDs that correspond to external IDs of network resources
  - Handles are typically implemented via integers or pointers

Common Traps and Pitfalls

- Blocking on a single I/O handle in read or accept
  - In general, a "concurrent daemon" should not service one I/O handle at the exclusion of the other handles
    - This will result in starvation for other services

- Polling via "busy waiting"
  - This will result in wasted CPU time

- Excessive process or thread creation
  - It is wasteful to dedicate OS resources while waiting for communication activity to occur

Distributed Logger

- This lecture describes an extended example of a distributed logging facility

- This example illustrates the applicability of various ACE components and covers:
  1. The application-level logging API
  2. The client logging daemon IPC mechanisms
  3. Several alternative concurrent server logging daemon designs and implementations

- The examples illustrate how OO and C++ simplify development and improve several key software quality factors
Distributed Logger (cont'd)

- The distributed logging facility was originally written in C and used `select` and/or `poll` directly.

- The original version was part of a commercial distributed on-line transaction-processing product that was ported from BSD to System V.

- This was later ported to C++ and is now in ACE.

Distributed Logger Architecture (cont'd)

- Note the two levels of I/O multiplexing in the distributed logger architecture:
  1. One or more application processes multiplex their logging records to a single client logging daemon located on each local host.
  2. One or more client logging daemons multiplex their accumulated messages to a single server logging daemon running on a designated host in a network/internetwork.

- Different IPC mechanisms may be used for each component, but the general architectures are the same.
  - Note that ACE reflects these similarities in the design and implementation.

- Server logging daemon collects, formats, and outputs logging records forwarded from multiple client logging daemons residing throughout a network or internetwork.

Distributed Logger Architecture (cont'd)

- The distributed logger provides services that:
  1. Identify processes via their program name, process ID (PID), and host name.
  2. Time-stamp records to facilitate chronological tracing.
  3. Prioritize record delivery at a client logging daemon.

- e.g.,

```c
ACE_ERROR (CLM_ERROR, "unable to fork in function spawn");
ACE_DEBUG (CLM_DEBUG, "sending to server %s", server_host);
```

Feb 30 14:50:13 1997@tango.cs.wustl.edu@22760070client-test
:unable to fork in function spawn
Feb 30 14:50:28 1997@tango.ice.uci.edu@018352020@drwho
:sending to server mambo
Application Logging API

- Provides applications with a thread-safe "variadic" logging interface similar to printf, e.g.,

  ACE_DEBUG ((LM_DEBUG, "server is %s\n", hostname);
  ACE_ERROR ((LM_ERROR, "usage: %n filename\n"));

- In addition to interpreting and expanding the variadic arguments, the API library code also:

  1. Creates a logging record and copies the expanded data into it
  2. Time-stamps the logging record
  3. Adds the PID and program name to the record
  4. Sends the record to the client logging daemon running on the local host
     - e.g., named pipes or STREAM pipes

Application Logging API (cont'd)

- Applications can specify different levels of logging priority (similar to UNIX syslogd), e.g.,

  enum Log_Priority
  {
    LM_SHUTDOWN = 1, /* Shutdown the logger */
    LM_DEBUG = 2, /* Messages with debugging info */
    LM_INFO = 3, /* Informational messages */
    LM_NOTICE = 4, /* Conditions that are not errors but require special handling */
    LM_WARNING = 5, /* Warning messages */
    LM_STARTUP = 6, /* Initialize the logger */
    LM_ERROR = 7, /* Errors */
    LM_CRIT = 8, /* Critical conditions, such as hard device errors */
    LM_ALERT = 9, /* A condition that must be corrected, such as a corrupted database */
    LM_EMERG = 10, /* A panic condition This is normally broadcast to all users */
    LM_MAX = 11 /* Maximum value + 1 */
  };

Client Logging Daemon

- Runs on the local host, reads from the named pipe being written to by different instances of the application logging API (which is linked into different user processes and/or threads)

- When logging records arrive, the client logging daemon behaves as follows:

  1. Reads the records in priority order
  2. Performs network-byte order conversions on multi-byte header fields
  3. Transmits the records to the server logging daemon across the network using TCP
     - However, TCP does not maintain logging record priorities...
     - Note, the client logging daemon may also run as a stand-alone process on a local host

Client Logging Daemon (cont'd)

- The following logging record PDU format is exchanged between the client and server logging daemons:

  class Log_Record {
  public:
    enum {
      MAXLOGMSGLEN = BUFSIZ, /* Maximum logging message. */
      ALIGN_WORDS = 8, /* Most restrictive alignment. */
    };
    Log_Record (void);
    Log_Record (Log_Priority lp, long time_stamp, pid_t pid);
    int print (const char host_name[], FILE *fp = stderr);
    void encode (void);
    void decode (void);
    int length (void);
    void length (int len);
  private:
    long type; /* Type of logging record */
    long length; /* Length of the recording */
    long time_stamp; /* Time logging record generated */
    long pid; /* Process Id generating the record */
    char msg_data[MAXLOGMSGLEN]; /* Logging record data */
  };

Concurrent Daemon Designs

- To motivate the utility of OO network programming techniques, the following slides examine several alternative designs for handling multiple sources of input and output in the distributed logger, e.g.,

  - **Non-blocking I/O concurrent daemon**
    - i.e., "polling"
    - Daemon process continuously sweeps across all open handles, performing non-blocking I/O on each

  - **Multiple-process or multi-threaded concurrent daemon**
    - i.e., fork or thread facilities (e.g., POSIX/Solaris)
    - Allows each separate slave daemon process or thread to block while reading from a single I/O handle

Concurrent Daemon Designs (cont’d)

- Alternative designs (cont’d)
  - **Single-threaded concurrent daemon**
    - i.e., based upon I/O demultiplexing with select and poll
    - select and poll allow blocking, non-blocking, and/or timed-wait on multiple I/O handles simultaneously
    - In certain cases, this approach may be easier to design, more portable, and potentially more efficient than alternative designs

The handle_logging_record Function

- The following function is used in each alternative daemon design to handle the reception of logging records sent from the client logging daemon to the server logging daemon

```
// Perform two recv’s to simulate a record service
// via the underlying bytestream-oriented TCP connection.
// Note that the sender must follow this protocol also...

template <class MUTEX = Null_Mutex>
int handle_logging_record (int handle)
{
    MUTEX lock;
    long m_len;
    Log_Record log_record;

    // The first recv reads the length (stored as a
    // fixed-size integer) of the adjacent logging record.
    size_t n = ACE_OS::recv (handle, &m_len, sizeof m_len);
    if (n != sizeof m_len)
        return n;
    else {
        // Convert byte-ordering
        m_len = ntohl (m_len);

        // The second recv then reads "length" bytes to
        // obtain the actual record.
        n = ACE_OS::recv (handle, (char *) &log_record, m_len);
        if (n != m_len) return -1;

        log_record.decode ();
        if (log_record.length () == n) {
            // Automatically obtain lock for MT designs.
            ACE_Guard<MUTEX> monitor (lock);

            log_record.print (output_device);
            // Automatically release lock here for
            // MT designs.
        }
        return n;
    }
}
```

- Note, fault tolerant applications may require more sophisticated message-oriented data transfer techniques
Polling via Non-blocking I/O

**C++ code for sample non-blocking server logging daemon**

```cpp
int main (void) {
    // Create a server end-point
    ACE_SOCK_Acceptor acceptor ((ACE_INET_Addr) PORT_NUM);
    ACE_SOCK_Stream new_stream;

    // Extract handle
    int s_handle = acceptor.get_handle ();
    int maxhandlep1 = s_handle + 1;

    // Set acceptor in non-blocking mode
    acceptor.enable (ACE_NONBLOCK);

    // Loop forever performing logger server processing
    for (;;) {
        // Poll each handle to see if logging
        // records are immediately available on
        // active network connections
    }
}
```

**Polling via Non-blocking I/O**

* (cont'd) *

- **C++ code for sample non-blocking server logging daemon**

```cpp
for (int handle = s_handle + 1;
    handle < maxhandlep1;
    handle++) {
    ssize_t n = handle_logging_record (handle);
    if (n == -1) { // No input pending
        if (errno == EWOULDBLOCK)
            continue;
    }
    else if (n == 0) { // Keep handles contiguous...
        ACE_OS::dup2 (handle, --maxhandlep1);
        ACE_OS::close (maxhandlep1);
    }
}

// Handle all pending connections
while (acceptor.accept (new_stream) != -1) { // Make new connection non-blocking
    new_stream.enable (ACE_NONBLOCK);
    handle = new_stream.get_handle ();
    assert (handle + 1 == maxhandlep1);
    maxhandlep1++;
    if (errno != EWOULDBLOCK)
        ACE_OS::error ("accept failed");
} /* NOTREACHED */
```
Polling via Non-blocking I/O (cont’d)

- **Advantages**
  - Relatively portable across UNIX and many PC platforms

- **Disadvantages**
  1. Inefficient
     - Wasteful of CPU resources due to “busy waiting”
  2. Non-extensible
     - Difficult to extend server to handle other types of I/O events and services without writing additional special code and modifying existing code
     - Note, this is a general drawback with all the functionally-designed approaches illustrated here

---

Multiple Process Creation (cont’d)

- Pseudo-code for sample multi-process master server logging daemon

```c
initialize acceptor endpoint
loop
   foreach connection request pending loop
      accept request
      fork a child process to handle request
   end loop
end loop
```

- Pseudo-code for sample multi-process slave server logging daemon

```c
loop
   foreach incoming data message from client loop
      call handle_logging_record
   end loop
exit process
end loop
```

- Note, handling the SIGCHLD signal complicates this basic logic somewhat...
Multiple Process Creation  
(cont’d)

• Advantages

1. `fork` is portable (on UNIX)
   - Win32 is more problematic...

2. In general, this design is efficient for certain types of daemons, e.g.,
   - I/O bound
   - Longer-duration/variable-length services
     - e.g., file transfer and rlogin
   - Services that set ownership and permissions based upon userid

3. Also, transparently take advantage of multiple CPUs

Multiple Process Creation  
(cont’d)

• Disadvantages

1. Often wasteful of OS resources
   - e.g., process table slots, virtual memory

2. Incurs additional overhead to schedule and context switch between the multiple processes

3. May require additional synchronization and/or mutual exclusion primitives to serialize access to shared output devices
   - e.g., in `Logging_Handler`

4. `SIGCHLD` signal handling is subtle and non-portable
Multiple Thread Creation (cont'd)

- Sample C++ multi-threaded server logging daemon

```c
static void
logging_acceptor (void)
{
    // Create a server end-point
    ACE_SOCK_Acceptor acceptor ((ACE_INET_Addr) PORT_NUM);
    ACE_SOCK_Stream new_stream;

    // Loop forever performing logging server processing
    for (;;)
    {
        // Wait for client connection request and create
        // a new ACE_SOCK_Stream endpoint (automatically
        // restarted upon interrupts)
        acceptor.accept (new_stream);

        // Create a new thread to handle client request
        ACE_Thread::spawn
            (ACE_THR_FUNC (logging_handler),
             (void *) new_stream.get_handle (),
             THR_DETACHED | THR_NEW_LWP);
    }

    ACE_Thread::exit ();
}
/* NOTREACHED */
```

```c
static void
logging_handler (int handle)
{
    // Handle all logging records from a particular
    // client (run in each slave thread)
    void
    
    // Perform a "blocking" receive and process
    // client logging records until client shuts
    // down the connection
    for (ssize_t n;;) {
        n = handle_logging_record <ACE_Thread_Mutex> (handle);
        if (n <= 0)
            break;
    }

    ACE_OS::close (handle);
    ACE_Thread::exit ();
    /* NOTREACHED */
}
```

- Pseudo-code for sample multi-threaded master server logging daemon

```
initialize acceptor endpoint
loop
    foreach connection request pending loop
        accept request
        spawn a thread to handle request
    end loop
end loop
```

- Pseudo-code for sample multi-thread slave server logging daemon

```
loop
    foreach incoming data message from client loop
        call handle_logging_record
    end loop
    exit thread
end loop
```
Multiple Thread Creation (cont’d)

- **Advantages**
  - Somewhat easier to program than fork
    - e.g., no subtle signal handling semantics
  - Potentially more efficient
    - Modulo the thread library and OS implementation...

- **Disadvantages**
  - Not portable
  - Many threads libraries are incapable of providing adequate performance and functionality
    - e.g., lack of support for sockets in Solaris <= 2.2!
  - Only allow one system call at a time...

Synopsis of select and poll

- select and poll are both I/O multiplexing mechanisms that perform “timed-waits” for input, output, or exception events to occur
  - The select API
    ```c
    int select
    ( int maxhandlep1, // Maximum handle plus 1
      fd_set *readhandles, // bit-mask of "read" handles
      fd_set *writehandles, // bit-mask of "write" handles
      fd_set *excepthandles, // bit-mask of "exception" handles
      struct timeval *tv // Amount of time to wait for events 
    );
    ```
  - The poll API
    ```c
    int poll
    ( struct pollfd *fds, // Handles of interest
      unsigned long nfds, // Number of handles to check
      int timeout // Length of time to wait (in milliseconds) 
    );
    ```

Single-Threaded Concurrent Daemon (select-based)

- Pseudo-code for sample single-threaded, concurrent server logging daemon
  
  ```c
  initialize acceptor endpoint
  initialize select handle sets
  loop
    select on active handles
    foreach active client handle loop
      call handle_logging_record
    end loop
  while connection requests pending loop
    accept the client connection and
    update handle set
  end loop
  end loop
  ```
Single-Threaded Concurrent Daemon (select-based) (cont’d)

- Sample C++ single-threaded, concurrent server logging daemon using I/O multiplexing
  - Note the serialization at the transport layer interface...

```c
int main (void)
{
  ACE_SOCK_Acceptor acceptor (ACE_INET_Addr PORT_NUM);
  ACE_SOCK_Stream new_stream;
  int s_handle = acceptor.get_handle ();
  int maxhandlep1 = s_handle + 1;
  fd_set temp_handles;
  fd_set read_handles;
  FD_ZERO (&temp_handles);
  FD_ZERO (&read_handles);
  FD_SET (s_handle, &read_handles);
  // Loop forever performing logging server processing
  for (;;)
  {
    temp_handles = read_handles; // structure assignment
    // Wait for client I/O events.
    ACE_OS::select (maxhandlep1, &temp_handles, 0, 0, 0);
    // Handle pending logging records first (s_handle + 1)
    // is guaranteed to be lowest client handle)
    for (int handle = s_handle + 1;
        handle < maxhandlep1;
        handle++)
      if (FD_ISSET (handle, &temp_handles)) {
        // Guaranteed not to block in this case!
        ssize_t n = handle_logging_record (handle);
        if (n == -1)
          ACE_OS::error ("logging failed");
        else if (n == 0) {
          // Handle client connection shutdown
          FD_CLR (handle, &read_handles);
          ACE_OS::close (handle);
          if (handle + 1 == maxhandlep1) {
            // Decrement past unused handles
            while (!FD_ISSET (--handle, &read_handles))
              continue;
            maxhandlep1 = handle + 1;
          }
        }
      }
      // Check whether any connection requests arrived
      if (FD_ISSET (s_handle, &temp_handles)) {
        // Handle all pending connection request
        // (note use of "polling" feature)
        while (ACE_OS::select (s_handle + 1, &temp_handles,
                             0, 0, ACE_Time_Value::zero) > 0)
          if (acceptor.accept (new_stream) == -1)
            ACE_OS::error ("accept");
          else {
            handle = new_stream.get_handle ();
            FD_SET (handle, &read_handles);
            if (handle >= maxhandlep1)
              maxhandlep1 = handle + 1;
          }
      }
      /* NOTREACHED */
```

- Advantages
  - May be more efficient than multi-threading and multi-processing for certain applications
  - e.g., no need to serialize logging record handling since output is single-threaded within a daemon process
  - Does not consume excessive OS resources by creating multiple processes or threads
  - Less context switching and scheduling overhead
  - Does not consume excessive CPU time by performing “busy-waiting”
Single-Threaded Concurrent Daemon (select-based) (cont'd)

- **Disadvantages**
  - Complicated and error-prone low-level interfaces
    - Requires developers to handle many details manually, e.g.,
      - Value/result parameter passing of handle sets requires copying
      - Handle set parsing
      - Multiple bitmasks, interrupts, etc.
  - Updating `maxhandlep` is tricky on close
  - There is a per-process limit on the number of handles available
  - Does not scale up to take advantage of multi-processor platforms
    - *i.e.*, serialization is at transport interface within a single process...

---

```
// Single-Threaded Concurrent Daemon (poll-based)

- **Sample single-threaded, concurrent server logging daemon**

```c
  
  // Maximum per-process open I/O handles
  const int MAX_HANDLES = 200;

  int main (void)
  {
    // Create a server end-point
    ACE.SOCK_Acceptor acceptor ((ACE_INET_Addr) PORT_NUM);
    ACE.SOCK_Stream new_stream;
    int s_handle = acceptor.get_handle ();
    struct pollfd poll_array[MAX_HANDLES];
    for (int i = 0; i < MAX_HANDLES; i++) {
      poll_array[i].fd = -1;
      poll_array[i].events = POLLLN;
    }
    poll_array[0].fd = s_handle;
    for (int nhandles = 1;;)
    {  // Wait for client I/O events.
      ACE_OS::poll (poll_array, nhandles);
    }
  }
```

---

Single-Threaded Concurrent Daemon (poll-based) (cont’d)

- **Advantages**
  - The same basic advantages as the `select-based` approach
  - However, compared to `select`, `poll` facilitates easier “packing” of handles in the `poll_fd` array
    - **poll** also detects a wider range of events than `select`
      - *e.g.*, priority-band events

- **Disadvantages (cont’d)**
  - Same as `select-based`
Overview of the Reactor

- The Reactor encapsulates the select and poll I/O multiplexing facilities
  - It is a portable interface to an OO library of extensible, reusable, and type-secure C++ classes
  - The Reactor addresses many limitations with the existing UNIX I/O demultiplexing facilities, while preserving the benefits they offer

- The Reactor helps simplify network programming by integrating mechanisms that support multiplexing of:
  1. Synchronous I/O-based events
  2. Timer-based events

- When these events occur, the Reactor automatically dispatches previously-registered “call-back” member functions that perform application-specific services

Overview of the Reactor (cont’d)

- The Reactor’s object-oriented design is based upon domain analysis of typical client/server I/O multiplexing structures and functionality
- A primary design goal is to decouple
  1. mechanisms for sensing, demultiplexing, and dispatching the I/O-based and timer-based events from
  2. policies of the application-specific services

- The Reactor forms the basis for more comprehensive OO daemon configuration, port multiplexing, and service dispatching frameworks
  - e.g., the Service Configurator framework in ACE
Single-threaded Concurrent Daemon (Reactor-based)

- Class relationships via Booch notation

---

Single-threaded Concurrent Daemon (Reactor-based) (cont’d)

- Pseudo-code for sample Reactor-based single-threaded, concurrent server logging daemon

  ```plaintext
  initialize acceptor endpoint
  initialize Reactor object with acceptor object
  loop
    call Reactor event loop function
  end loop
  ```

- Pseudo-code for Reactor event dispatcher function

  ```plaintext
  wait for set of client handles to become active
  foreach active client handle loop
    invoke appropriate service call-back routine
  end loop
  ```

---

Single-threaded Concurrent Daemon (Reactor-based) (cont’d)

- The server logging daemon is decoupled into several modular components that perform different tasks

  - Application-specific components
    - Process logging records
  - Connection-related components
    1. Accept
      - Accepts connection requests from clients
        - Dynamically creates a Svc_Handler object per-client and registers it with the Reactor
    2. Svc_Handler
      - Performs I/O with clients
  - ACE framework components
    - Perform IPC, event demultiplexing, dynamic linking, etc.
Single-threaded Concurrent Daemon (Reactor-based) (cont’d)

- **C++** interface for registering and dispatching event objects

```cpp
class ACE_Event_Handler {
public:
    // Returns the I/O handle associated.
    virtual int get_handle (void) const = 0;

    // Called when object is removed from the ACE_Reactor
    virtual int handle_close (int handle);
    // Called when input becomes available on HANDLE
    virtual int handle_input (int handle);
    // Called when output is possible on HANDLE
    virtual int handle_output (int handle);
    // Called when urgent data is available on HANDLE
    virtual int handle_exception (int handle);
    // Called when timer expires (TV stores the current time and ARG is the argument given when the handler was originally scheduled)
    virtual int handle_timeout (const Time_Value &tv,
        const void *arg = 0);
};
```

Single-threaded Concurrent Daemon (Reactor-based) (cont’d)

- Template class interface for accepting connection requests from remote client daemons

```cpp
template <class SVC_HANDLER,
    class PEER_ACCEPTOR>
class Accepter : public ACE_Event_Handler {
public:
    Accepter (void);
    Accepter (ACE_Reactor *, const ADDR &a);
    ~Accepter (void);
    int open (ACE_Reactor *, const ADDR &a);

    // Dynamic linking hooks
    virtual int init (int argc, char **argv[]);
    virtual int info (char **info_string,
        int length) const;

private:
    virtual int get_handle (void) const;
    virtual int handle_input (int);
    virtual int handle_close (int = -1);
    PEER_ACCEPTOR acceptor_; // Accept connections
    ACE_Reactor *reactor_; // Demultiplex events.
};
```

Single-threaded Concurrent Daemon (Reactor-based) (cont’d)

- **Acceptor** implementation

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

template <class SH, class PA> int
Accepter<SH, PA>::open (const PA::PEER_ADDR &addr) {
    acceptor_.open (addr);
}

template <class SH, class PA>
Accepter<SH, PA>::Accepter (const PA::PEER_ADDR &addr) {
    open (addr);
}
```
template <class SH, class PA>  
Acceptor<SH, PA>::init (int argc, char *argv[])  
{  
  PA::PEER_ADDR addr;  
  Get OPT getopt (argc, argv, "p");  
  for (int c; (c = getopt ())) != -1; )  
    switch (c) {  
      case 'p':  
        addr.set (ACE_OS::atoi (getopt.optarg));  
        break;  
      default:  
        break;  
    }  
  return open (addr);  
}  

Single-threaded Concurrent  
Daemon (Reactor-based) (cont'd)

- Template class that performs I/O with remote clients

template <class PEER_STREAM>  
class Svc_Handler : public ACE_Event_Handler  
{  
  public:  
    Svc_Handler (ACE_Reactor *);  
    // = Must be filled in by subclass.  
    virtual int open (void) = 0;  
    virtual int svc (void) = 0;  
    operator PEER_STREAM &();  
    virtual int get_handle (void) const;  
  protected:  
    // = Demultiplexing hook.  
    virtual int handle_input (int);  
    virtual int handle_close (int);  
    // = Ensure dynamic allocation  
    virtual ~Svc_Handler (void);  
  char host_name_[MAXHOSTNAMELEN + 1];  
  // Communicates with connected peer.  
  PEER_STREAM peer_stream_;  
  ACE_Reactor *reactor_;  
};
Single-threaded Concurrent Daemon (Reactor-based) (cont’d)

- Handler implementation

```c++
#define CS PEER_STREAM

template <class CS>
Svc_Handler<CS>::Svc_Handler (ACE_Reactor *r)
    : reactor_ (r) {}
    // Extract the underlying CS (e.g., for
    // purposes of accept()).

template <class CS>
Svc_Handler<CS>::operator CS &() { return peer_stream_; }
    // Initiate the virtual function call-back.

template <class CS> int
Svc_Handler<CS>::handle_input (int)
    { // Hook method.
      return svc ();
    }
```

- Define the classes that perform server logging daemon functionality

```c++
class Logging_Handler :
    public Svc_Handler<ACE.SOCK_Stream>
{ public:
    Logging_Handler (ACE_Reactor *);
    virtual int open (void);
    virtual int svc (void);
};

typedef Acceptor<Logging_Handler,
    ACE.SOCK_Acceptor>
    Logging_Acceptor;
```

- Implementing the application-specific functions

```c++
// Constructor.
Logging_Handler::Logging_Handler (ACE_Reactor *reactor)
    : Svc_Handler<ACE.SOCK_Stream> (reactor) {} 

// Open hook (register with ACE_Reactor).
int
Logging_Handler::open (void)
    { reactor_.register_handler
        (this, ACE_Event_Handler::READ_MASK);
    }
```
// Callback routine for handling the reception of remote logging transmissions.

int Logging_Handler::svc (void)
{
    ssize_t n = peer_stream_recv (&len, sizeof len);
    int len;

    switch (n) {
        default: return -1; /* NOTREACHED */
        case 0: return 0; /* NOTREACHED */
        case sizeof (int): { Log_Record lp;
                                len = ntohl (len);
                                n = peer_stream_recv_n ((void *)&lp, len);
                                lp.decode ();
                                if (lp.len == n)
                                    lp.print (host_name, 0, stderr);
                                break;
        }
        return 0;
    }
}

Single-threaded Concurrent Daemon (Reactor-based) (cont’d)

- **Main event-loop for the server logging daemon**

```c
int main (int argc, char *argv[])
{
    // Event demultiplexor.
    ACE_Reactor reactor;

    // Create the Acceptor.
    Logging_Acceptor acceptor ((ACE_INET_Addr) port);

    // Register handler.
    reactor.register_handler (&acceptor, ACE_Event_Handler::READ_MASK);

    // Performs event loop.
    for (;;) reactor.handle_events ();
}
```

Single-threaded Concurrent Daemon (Reactor-based) (cont’d)

- **Advantages**
  - **OO design** decouples the low-level I/O-based event multiplexing mechanisms from the application-specific service policies
    - This improves extensibility, portability, and reuse significantly
  - **Use of parameterized types** decouples the reliance on a particular network IPC interface
    - *e.g.*, both socket-based and TLI-based C++ wrappers may be used

- **Disadvantages**
  - The flow of control for the Reactor's event-driven service dispatching is somewhat difficult to follow at first
  - Parameterized types tend to be slow to compile!
Summary

- There are a wide variety of alternative designs for structuring concurrent network server daemons

- Object-oriented techniques are useful for devising highly decoupled software architectures that are modular, reusable, extensible, and efficient

- C++ features such as inline functions, parameterized types, inheritance, and dynamic binding facilitate the implementation and design of such architectures