The Reactor
An Object-Oriented Wrapper for Event-Driven Port Monitoring and Service Demultiplexing (Part 1 of 2)

Douglas C. Schmidt
schmidt@ics.uci.edu
Department of Information and Computer Science
University of California, Irvine, CA 92717, (714) 856-4105

An earlier version of this paper appeared in the February 1993 issue of the C++ Report.

1 Introduction

This is part one of the third article in a series that describes techniques for encapsulating existing operating system (OS) interprocess communication (IPC) services within object-oriented (OO) C++ wrappers. The first article explains the main principles and motivations for OO wrappers [1], which simplify the development of correct, concise, portable, and efficient applications. The second article describes an OO wrapper called `IPC_SAP` [2] that encapsulates the BSD socket and System V TLI system call *Application Programmatic Interfaces* (APIs). `IPC_SAP` enables application programs to access local and remote IPC protocol families such as TCP/IP via a type-secure, object-oriented interface.

This third article presents an OO wrapper for the I/O port monitoring and timer-based event notification facilities provided by the `select` and `poll` system calls.[1] Both `select` and `poll` enable applications to specify a time-out interval to wait for the occurrence of different types of input and output events on one or more I/O descriptors. `select` and `poll` detect when certain I/O or timer events occur and demultiplex these events to the appropriate application(s). As with many other OS APIs, the event demultiplexing interfaces are complicated, error-prone, non-portable, and not easily extensible. An extensible OO framework called the `Reactor` was developed to overcome these limitations. The `Reactor` provides a set of higher-level programming abstractions that simplify the design and implementation of event-driven distributed applications. The `Reactor` also shields developers from many error-prone details in the existing event demultiplexing APIs and improves application portability between different OS variants.

The `Reactor` is somewhat different than the `IPC_SAP` class wrapper described in [2]. `IPC_SAP` added a relatively "thin" OO veneer to the BSD socket and System V TLI APIs. On the other hand, the `Reactor` provides a significantly richer set of abstractions than those offered directly by `select` or `poll`. In particular, the `Reactor` integrates I/O-based port monitoring together with timer-based event notification to provide a general framework for demultiplexing application communication services. Port monitoring is used by event-driven network servers that perform I/O on many connections simultaneously. Since these servers must handle multiple connections it is not feasible to perform blocking I/O on a single connection indefinitely. Likewise, the timer-based APIs enable applications to register certain operations that are periodically or aperiodically activated via a centralized timer facility controlled by the `Reactor`.

This topic is divided into two parts. Part one (presented in this article) describes a distributed logging facility that motivates the need for efficient event demultiplexing, examines several alternative solution approaches, evaluates the advantages and disadvantages of these alternatives, and compares them with the `Reactor`. Part two (appearing in a subsequent issue of the C++ Report) focuses on the OO design aspects of the `Reactor`. In addition, it discusses the design and implementation of the distributed logging facility. This example illustrates precisely how the `Reactor` simplifies the development of event-driven distributed applications.

2 Example: A Distributed Logging Facility

To motivate the utility of event demultiplexing mechanisms, this section describes the requirements and behavior of a distributed logging facility that handles event-driven I/O from multiple sources "simultaneously." As shown in Figure 1, the distributed logging facility offers several services to applications that operate concurrently throughout a network environment. First, it provides a centralized location for recording certain status information used to simplify the management and tracking of distributed application behavior. To facilitate this, the client daemon *time-stamps* outgoing logging records to allow chronological tracing and reconstruction of the execution order of multiple concurrent processes executing on separate host machines. Second, the facility also enables the *prioritized* delivery of logging records. These records are received and forwarded by the client daemon in the order of

[1] The `select` call is available on BSD and SVR4 UNIX platforms, as well as with the WINSOCK API; `poll` is available with System V variants of UNIX.

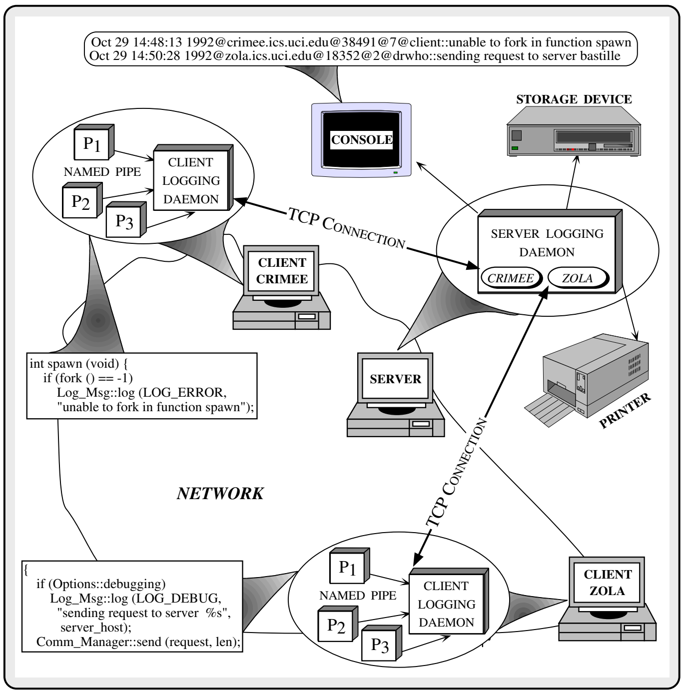

Figure 1: Network Environment for the Distributed Logging Facility

their importance, rather than in the order they were originally generated.

Centralizing the logging activities of many distributed applications within a single server is also useful since it serializes access to shared output devices such as consoles, printers, files, or network management databases. In contrast, without such a centralized facility, it becomes difficult to monitor and debug applications consisting of multiple concurrent processes. For example, the output from ordinary C `stdio` library subroutines (such as `fputs` and `printf`) that are called simultaneously by multiple processes or threads is often scrambled together when it is displayed in a single window or console.

The distributed logging facility is designed using a client/server architecture. The server logging daemon[2] collects, formats, and outputs logging records forwarded from client logging daemons running on multiple hosts throughout a local and/or wide-area network. Output from the logging server may be redirected to various devices such as printers, persistent storage repositories, or logging management consoles.

As shown in Figure 1, the *InterProcess Communication* (IPC) structure of the logging facility involves several levels of demultiplexing. For instance, each client host in the network contains multiple application processes (such as $P_1, P_2, and P_3$) that may participate with the distributed logging facility. Each participating process uses the *application logging API* depicted in the rectangular boxes in Figure 1 to format debugging traces or error diagnostics into *logging records*. A logging record is an object containing several header fields and a payload with a maximum size of approximately 1K bytes. When invoked by an application process, the `Log_Msg::log` API prepends the current process identifier and program name to the record. It then uses the "record-oriented" named pipe IPC mechanism to demultiplex these composite logging records onto a single client logging daemon running on each host machine. The client daemon prepends a time-stamp to the record and then employs a remote IPC service (such as TCP or RPC) to demultiplex the record into a server logging daemon running on a designated host in the network. The server operates in an event-driven manner, processing logging records as they arrive from multiple client daemons. Depending on the logging behavior of the participating applications, the logging records may be sent by arbitrary clients and arrive at the server daemon at arbitrary time intervals.

A separate TCP stream connection is established between each client logging daemon and the designated server logging daemon. Each client connection is represented by a unique I/O descriptor in the server. In addition, the server also maintains a dedicated I/O descriptor to accept new connection requests from client daemons that want to participate with the distributed logging facility. During connection establishment the server caches the client's host name (illustrated by the ovals in the logging server daemon), and uses this information to identify the client in the formatted records it prints to the output device(s).

The complete design and implementation of the distributed logging facility is described in [3]. The remainder of the current article presents the necessary background material by exploring several alternative UNIX mechanisms for handling I/O from multiple sources.

3 Operating System Event Demultiplexing

Modern operating systems such as UNIX, Windows NT, and OS/2 offer several techniques that allow applications to perform I/O on multiple descriptors "simultaneously." This section describes four alternatives and compares and contrasts their advantages and disadvantages. To focus the discussion, each alternative is characterized in terms of the distributed logging facility described in Section 2 above. In particular, each section presents a skeletal server logging daemon implemented with the alternative being discussed. To save space and increase clarity, the examples utilize the OO `IPC_SAP` socket-wrapper library described in a previous C++ Report article [2].

The `handle_logging_record` function shown in Figure 2 is also invoked by all the example server daemons. This function is responsible for receiving and processing the

[2] A daemon is an OS process that runs continuously "in the background," performing system-related services such as updating routing table entries or handling network file system requests.

```
/* Perform two recvs to simulate a message-oriented service
   via the underlying bytestream-oriented TCP connection.
   The first recv reads the length (stored as a fixed-size
   integer) of the adjacent logging record.  The second recv
   then reads "length" bytes to obtain the actual record.
   Note that the sender must also follow this protocol... */

int
handle_logging_record (int fd)
{
  int n;
  long msg_len;
  Log_PDU log_pdu;

  if ((n = ::recv (fd, (char *) &msg_len, sizeof msg_len, 0))
      != sizeof msg_len)
    return n;
  else {
    msg_len = ntohl (msg_len); /* Convert byte-ordering */

    if ((n = ::recv (fd, (char *) &log_pdu, msg_len, 0))
        != msg_len)
      return -1;
    log_pdu.decode ();
    if (log_pdu.get_len () == n)
      /* Obtain lock here for concurrent designs */
      log_pdu.print (output_device);
      /* Release lock here for concurrent designs */
    return n;
  }
}
```

Figure 2: Function for Handling Logging Records

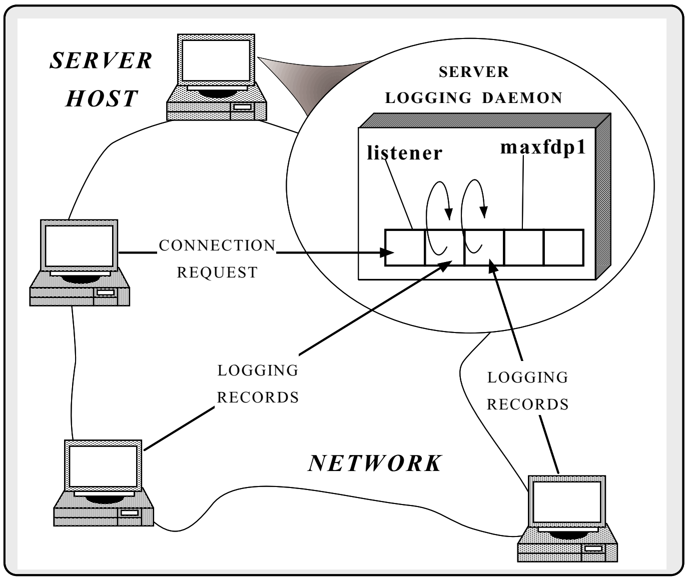

Figure 3: Nonblocking I/O Server

logging records and writing them to the appropriate output device. Any synchronization mechanisms required to serialize access to the output device(s) are also performed in this function. In general, the concurrent multi-process and multi-thread approaches are somewhat more complicated to develop since output must be serialized to avoid scrambling the logging records generated from all the separate processes. To accomplish this, the concurrent server daemons cooperate by using some form of synchronization mechanisms (such as semaphores, locks, or other IPC mechanisms like FIFOs or message queues) in the `handle_logging_record` subroutine.

3.1 A Non-blocking I/O Solution

One method for handling I/O on multiple descriptors involves the use of "polling." Polling operates by cycling through a set of open descriptors, checking each one for pending I/O activity. Figure 4 presents a code fragment that illustrates the general structure of this approach. Initially, an `IPC_SAP` listener object is created and set into "non-blocking mode" via the `SOCK_Listener::enable` member function. Next, the main loop of the server iterates across the open descriptors, attempting to receive logging record input from each descriptor. If input is available immediately, it is read and processed. Otherwise, the `handle_logging_record` function returns -1, `errno` is set to EWOULDBLOCK, and the loop continues polling at the next descriptor. After all the open I/O connections have been polled once, the server accepts any new connection requests that have arrived and starts polling the descriptors from the beginning again. When the `handle_logging_record` function returns 0 (signifying the client has closed the connection), the corresponding I/O descriptor is closed. At this point, the server makes a dupli-

```
int
main (void)
{
  /* Create a server end-point */
  SOCK_Listener listener (INET_Addr (PORT_NUM));
  SOCK_Stream new_stream;
  int s_fd = listener.get_fd (); /* Extract descriptor */
  int maxfdp1 = s_fd + 1;

  /* Check for constructor failure */
  if (s_fd == -1)
    ::perror ("server"), ::exit (1);

  /* Set listener in non-blocking mode */
  listener.enable (O_NONBLOCK);

  /* Loop forever performing logger server processing */
  for (;;) {
    /* Poll each descriptor to see if logging
       records are immediately available on
       active network connections */
    for (int fd = s_fd + 1; fd < maxfdp1; fd++) {
      int n;
      if ((n = handle_logging_record (fd)) == -1) {
        if (errno == EWOULDBLOCK) /* No input pending */
          continue;
        else ::perror ("recv failed");
      }
      else if (n == 0) {
        /* Keep descriptors contiguous... */
        ::dup2 (fd, --maxfdp1);
        ::close (maxfdp1);
      }
    }
    /* Check if new connection requests have arrived */
    while (listener.accept (new_stream) != -1) {
      /* Make new connection non-blocking */
      new_stream.enable (O_NONBLOCK);
      fd = new_stream.get_fd ();
      assert (fd + 1 == maxfdp1);
      maxfdp1++;
    }
    if (errno != EWOULDBLOCK) ::perror ("accept failed");
  }
  /* NOTREACHED */
}
```

Figure 4: A Nonblocking I/O Server (Version 1)

```
int
main (void)
{
  /* Create a server end-point */
  SOCK_Listener listener (INET_Addr (PORTNUM));
  SOCK_Stream new_stream;
  fd_set in_use; /* Bitmask for active descriptors */
  int s_fd = listener.get_fd (); /* Extract descriptor */
  int maxfdp1 = s_fd + 1;

  FD_ZERO (&in_use);
  FD_SET (s_fd, &in_use);

  /* Set listener SAP into non-blocking mode */
  listener.enable (O_NONBLOCK);

  /* Loop forever performing logger server processing */
  for (;;) {
    /* Poll each descriptor to see if logging
       records are immediately available on
       active network connections */
    for (int fd = s_fd + 1; fd < maxfdp1; fd++) {
      int n;
      if (FD_ISSET (fd, &in_use) &&
          (n = handle_logging_record (fd)) == -1) {
        if (errno == EWOULDBLOCK) /* No input pending */
          continue;
        else
          ::perror ("recv failed");
      }
      else if (n == 0) {
        ::close (fd);
        FD_CLR (fd, &in_use);
        if (fd + 1 == maxfdp1) {
          while (!FD_ISSET (--fd, &read_fds))
            continue;
          maxfdp1 = fd + 1;
        }
      }
    }
    /* Check if new connection requests have arrived */
    while (listener.accept (new_stream) != -1) {
      /* Make new connection non-blocking */
      new_stream.enable (O_NONBLOCK);
      fd = new_stream.get_fd ();
      FD_SET (fd, &in_use);
      if (fd >= maxfdp1)
        maxfdp1 = fd + 1;
    }
    if (errno != EWOULDBLOCK)
      ::perror ("accept failed");
  }
  /* NOTREACHED */
}
```

Figure 5: A Polling, Nonblocking I/O Server (Version 2)

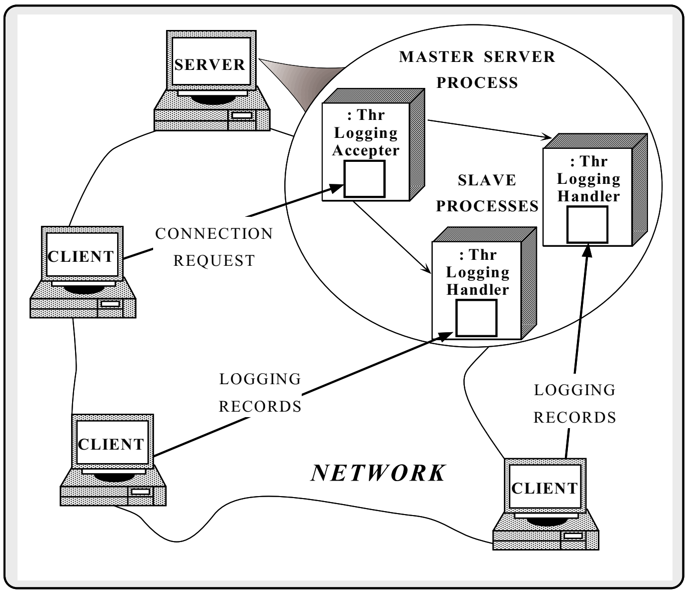

Figure 6: Multi-process Server

cate of the highest descriptor and stores it into the slot number of the terminating descriptor (in order to maintain a contiguous range of descriptors). In contrast, Figure 5 illustrates a similar approach that uses an `fd_set` bitmask to keep track of the currently active descriptors.

The primary disadvantage with polling is that it consumes excessive CPU cycles by making unnecessary system calls while "busy-waiting." For instance, if input occurs only intermittently on the I/O descriptors, the server process will repeatedly and superfluously poll descriptors that do not have any pending logging records. On the other hand, if I/O is continuously received up all descriptors, this approach may be reasonable. In addition, an advantage with polling is that it is portable across OS platforms.

3.2 A Multiple Process Solution

Another approach involves designing the application as a "concurrent server,"[3] which creates a separate OS process to manage the communication channel connected to each client logging daemon. Figure 7 presents code that illustrates this technique. The main loop in the *master* server blocks while listening for the arrival of new client connection requests. When a request arrives, a separate *slave* process is created via `fork`. The newly created slave process performs blocking I/O on a single descriptor in the `handle_client` subroutine, which receives all logging records sent from its associated client. When the corresponding client daemon terminates, a 0 is returned from the `recv` system call, which terminates the slave process. At this point, the OS sends a SIGCHLD signal to the master process. The `child_reaper` signal handler catches this signal and "reaps" the zombie'd child's exit status information. Note

[3]Concurrent servers are described in detail in [4].

```
/* Handle all logging records from a particular
   client (run in each slave process) */
void
handle_client (int fd)
{
  /* Perform a "blocking" receive and process
     client logging records until client shuts down
     the connection */
  for (int n; (n = handle_logging_record (fd)) > 0; )
    continue;
  if (n == -1)
    ::perror ("recv failed");
  ::exit (0);
}

/* Reap zombie'd children (run in the master process) */
void
child_reaper (int)
{
  for (int res; (res = ::waitpid (-1, 0, WNOHANG)) > 0
              || (res == -1 && errno == EINTR); )
    continue;
}

/* Master process */
int
main (void)
{
{
  /* Create a server end-point */
  SOCK_Listener listener (INET_Addr (PORT_NUM));
  SOCK_Stream new_stream;
  struct sigaction sa;

  /* Check if server constructor failed */
  if (listener.get_fd () == -1)
    ::perror ("listener"), ::exit (1);

  /* Restart interrupted system calls */
  sa.sa_flags = SA_RESTART;
  ::sigemptyset (&sa.sa_mask);
  sa.sa_handler = child_reaper;
  /* Arrange to reap deceased children */
  if (::sigaction (SIGCHLD, &sa, 0) == -1)
    ::perror ("sigaction"), ::exit (1);

  /* Loop forever performing logging server processing */
  for (;;) {
    /* Wait for client connection request and create a
       new SOCK_Stream endpoint (note, accept is
       automatically restarted after interrupts) */
    if (listener.accept (new_stream) == -1) {
      ::perror ("accept");
      continue;
    }
    /* Create a new process to handle client request */
    switch (::fork ()) {
    case -1:
      ::perror ("fork failed");
      break;
    case 0: /* In child */
      listener.close ();
      handle_client (new_stream.get_fd ());
      /* NOTREACHED */
    default: /* In parent */
      new_stream.close ();
      break;
    }
  }
  /* NOTREACHED */
}
```

Figure 7: A Multi-Process Server

that the occurrence of signals in the server requires the main loop in the master process to handle interrupts correctly. On most UNIX platforms, certain system calls (*e.g.,* `accept`) are *not* restarted automatically when signals occur. An application may detect this by checking if `errno` contains the EINTR value when the `accept` system call returns -1.

The multiple process design has several disadvantages. First, it may consume excessive OS resources (such as process-table slots, one of which is allocated for each client), which may increase the OS scheduling overhead. Second, a context switch is typically required to restart a waiting process when input arrives. Third, handling signals and interrupted system calls properly involves writing subtle and potentially error-prone code. For example, the `sigaction` interface must be used with SVR4 to ensure that the signal disposition remains set to the previously registered call-back function after the first SIGCHLD signal is caught. Finally, increased software complexity results from implementing the mutual exclusion mechanisms that serialize access to output devices. Given the "event-driven, discrete message" communication pattern of the distributed logging facility, this additional overhead and complexity is unnecessarily expensive.

However, certain other types of network servers do benefit significantly from creating separate processes that handle client requests. In particular, this approach improves the response times of servers that are either (1) I/O bound (*e.g.,* complicated relational database queries) or (2) involve simultaneous, longer-duration client services that require a variable amount of time to execute (*e.g.,* file transfer or remote login) [5]. Another advantage is that overall server performance may be improved in an application-transparent manner, if the underlying operating system supports multiple processing elements effectively.

3.3 A Multiple Thread Solution

The third approach utilizes a multi-threaded approach. The example illustrated in Figure 9 uses the SunOS 5.x threads library [6] to implement a multi-threaded concurrent server. Other thread libraries (such as POSIX and Windows NT threads) offer an equivalent solution. In the example code, a new thread a spawned by the `thr_create` routine to handle each client connection. In addition to creating the necessary stack and other data structures necessary to execute a separate thread of control, the `thr_create` routine also calls the `handle_client` function. This function receives all the logging records that arrive from a particular client. Note that when a client shuts down, the `thr_exit` routine is used to exit the particular thread, *not* the entire process.

The multi-threaded approach is relatively simple to implement, assuming a reasonable thread library is available, and provides several advantages over a multi-process approach. For example, the complicated signal handling semantics are no longer an issue since the server spawns new threads as "detached." A detached thread in SunOS 5.x never re-synchronizes nor re-joins with the main thread of control

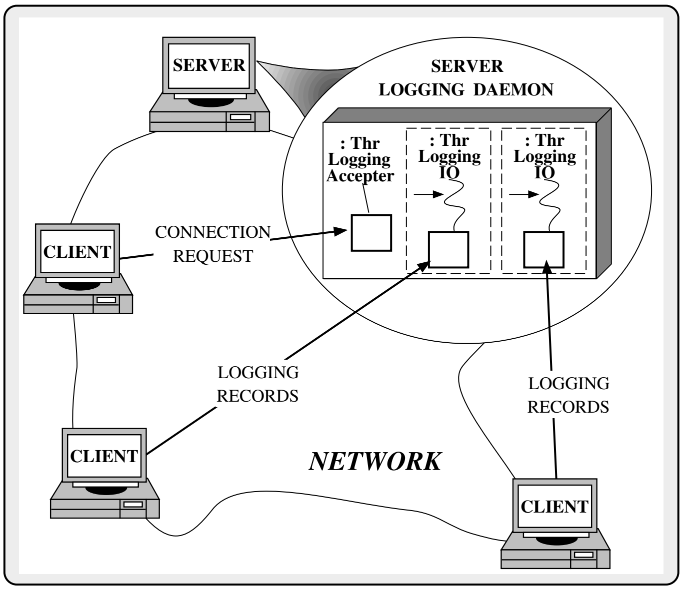

Figure 8: Multi-thread Server

```
/* Handle all logging records from a particular
   client (run in each slave thread) */
void *
handle_client (int fd)
{
  int n;

  /* Perform a "blocking" receive and process
     client logging records until client shuts
     down the connection */
  while ((n = handle_logging_record (fd)) > 0)
    continue;
  if (n == -1)
    ::perror ("recv failed");
  if (::close (fd) == -1)
    ::perror ("close failed");
  /* Exits thread, *not* entire process! */
  ::thr_exit (0);
  /* NOTREACHED */
}
/* Master server */
int
main (void)
{
  /* Create a server end-point */
  SOCK_Listener listener (INET_Addr (PORT_NUM));
  SOCK_Stream new_stream;

  if (listener.get_fd () == -1)
    ::perror ("listener"), ::exit (1);

  /* Loop forever performing logging server processing */
  for (;;) {
    /* Wait for client connection request and create a
       new SOCK_Stream endpoint (automatically
       restarted upon interrupts) */
    if (listener.accept (new_stream) == -1) {
      ::perror ("accept");
      continue;
    }

    /* Create a new thread to handle client request */

    if (::thr_create (0, 0, (void *(*)(void *)) handle_client,
                      (void *) new_stream.get_fd (),
                      THR_DETACHED, 0) != 0)
      ::perror ("thr_create failed");
  }
  /* NOTREACHED */
}
```

Figure 9: A Multi-Thread Server

```
int select
(
  /* Maximum descriptor plus 1 */
  int width,
  /* bit-mask of "read" descriptors to check */
  fd_set *readfds,
  /* bit-mask of "write" descriptors to check */
  fd_set *writefds,
  /* bit-mask of "exception" descriptors to check */
  fd_set *exceptfds,
  /* Amount of time to wait for events to occur */
  struct timeval *timeout
);
```

Figure 10: The select() Interface

```
int poll
(
  /* Array of descriptors of interest */
  struct pollfd fds[],
  /* Number of descriptors to check */
  unsigned long nfds,
  /* Length of time to wait, in milliseconds */
  int timeout
);
```

Figure 11: The poll() Interface

when it exits. Moreover, compared with a process, it may be more efficient to create, execute, and terminate a thread, due to a reduction in context switching overhead [7]. In addition, sharing of global data objects is also often more convenient since no special operations must be performed to obtain shared memory.

Unfortunately, most traditional versions of UNIX do not provide adequate support for threads. For example, some thread variants only allow one outstanding system call per-process, and others do not permit multiple threads of control to utilize certain OS APIs (such as sockets or RPC). In particular, many traditional UNIX and standard C library routines were not designed to be re-entrant, which complicates their use in a multi-threaded application.

3.4 The Event Demultiplexing Solution

The fourth approach utilizes the event demultiplexing facilities available via the `select` and `poll` system calls. These mechanisms overcome many limitations with the other solutions described above. Both `select` and `poll` allow network applications to wait various lengths of time for different types of I/O events to occur on multiple I/O descriptors *without* requiring either polling or multiple process or thread invocations. This section outlines the `select` and `poll` system calls, sketches example implementations of the logging server daemon using these two calls, and contrasts the limitations of the existing event demultiplexing services with the advantages of the `Reactor` OO class library.

3.4.1 The select and poll System Calls

The following paragraphs describe the similarities and differences of the `select` system call (shown in Figure 10) and the `poll` system call (shown in Figure 11). Both these calls support I/O-based and timer-based event demultiplexing. The syntax and semantics of both `select` and `poll` are described in greater detail in [8].

Despite their different APIs, select and poll share many common features. For example, they both wait for various input, output, and exception[4] events to occur on a set of I/O descriptors, and return an integer value indicating how many events occurred. In addition, both system calls enable applications to specify a time-out interval that indicates the maximum amount of time to wait for I/O events to transpire. The three basic time-out intervals include (1) waiting "forever," (*i.e.,* until an I/O event occurs or a signal interrupts the system call), (2) waiting a certain number of time units (measured in either seconds/micro-seconds (select) or milli-seconds (poll)), and (3) performing a "poll" (*i.e.,* checking all the descriptors and returning immediately with the results).

There are also several differences between select and poll. For example, select uses three *descriptor sets* (one for reading, one for writing, and one for exceptions), which are implemented as bit-masks to reduce the amount of space used. Each bit in a bit-mask corresponds to a descriptor that may be enabled to check for particular I/O events. The poll function, on the other hand, is somewhat more general and has a less convoluted interface. The poll API includes an array of pollfd structures, a count of the number of structures in the array, and a timeout value. Each pollfd structure in the array contains (1) the descriptor to check for I/O events (a value of -1 indicates that this entry should be ignored), (2) the event(s) of interest (*e.g.,* various priorities of input and output conditions) on that descriptor, and (3) the event(s) that actually occurred on the descriptor (such as input, output, hangups, and errors), which are enabled upon return from the poll system call. Note that in versions of System V prior to release 4, poll only worked for STREAM devices such as terminals and network interfaces. In particular, it did not work on arbitrary I/O descriptors such as ordinary UNIX files and directories. The select and SVR4 poll system calls operate upon all types of I/O descriptors.

3.4.2 select-based Logging Server Example

Figure 13 illustrates a code fragment that uses the BSD select system call to perform the main processing loop of the server logging daemon. This server implementation employs two descriptor sets: (1) read_fds (which keeps track of the I/O descriptors associated with active client connections) and (2) temp_fds (which is a copy of the read_fds descriptor set that is passed by "value/result" to the select system call). Initially, the only bit enabled in the read_fds descriptor set corresponds to the I/O descriptor that "listens" for new incoming connection requests to arrive from client logging daemons.

After the initialization is complete, the main loop invokes select with temp_fds as its only descriptor set argument (since the server is not interested in either "write" or "exception" events). Since the final argument is a NULL struct timeval * NULL pointer, the select call blocks until one or more clients send logging records or request new connections (note that select must be manually restarted if interrupts occur). When select returns, the temp_fds variable is modified to indicate which descriptors have pending logging record data or new client connection requests. Logging records are handled first by iterating through the temp_fds set checking for descriptors that are now ready for reading (note that the semantics of select guarantee that recv will not block on this read). The recv function returns 0 when a client shuts the connection down. This informs the main server loop to clear the particular bit representing that connection in the read_fds set.

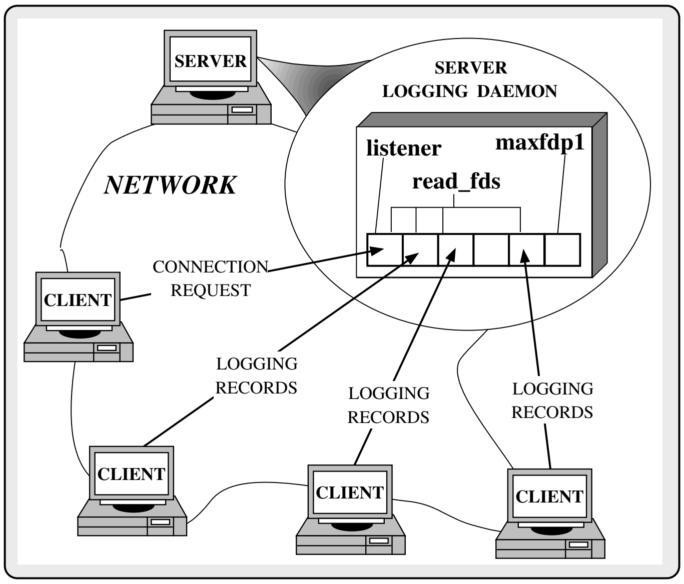

Figure 12: select-based Server

After all the pending logging records have been processed, the server checks whether new connection requests have arrived on the listening I/O descriptor. If one or more requests have arrived, they are all accepted and the corresponding bits are enabled in the read_fds descriptor set. This section of code illustrates the "polling" feature of select. For example, if both fields in the struct timeval argument are set to 0, select will check the enabled descriptor and return immediately to notify the application if there are any pending connection requests. Note how the server uses the width variable to keep track of the largest I/O descriptor value. This value limits the number of descriptors that select must inspect upon each invocation.

3.4.3 poll-based Logging Server Example

Figure 14 reimplements the main processing loop of the server logging daemon using the System V UNIX poll system call in place of select. Note that the overall structure of the two servers is almost identical. However, a number of minor modifications must be made to accommodate the

[4] A common example of exception events are the TCP protocol's "urgent" data, which informs applications that special activities may have occurred on a communication channel.

```
int
main (void)
{
  /* Create a server end-point */
  SOCK_Listener listener (INET_Addr (PORT_NUM));
  SOCK_Stream new_stream;
  fd_set temp_fds;
  fd_set read_fds;
  int s_fd = listener.get_fd ();
  int maxfdp1 = s_fd + 1;

  /* Check for constructor failure */
  if (s_fd == -1)
    ::perror ("server"), ::exit (1);

  FD_ZERO (&temp_fds);
  FD_ZERO (&read_fds);
  FD_SET (s_fd, &read_fds);

  /* Loop forever performing logging server processing */
  for (;;) {
    temp_fds = read_fds; /* structure assignment */

    /* Wait for client I/O events (handle interrupts) */
    while (::select (maxfdp1, &temp_fds, 0, 0, 0) == -1
           && errno == EINTR)
      continue;

    /* Handle pending logging records first (s_fd + 1
       is guaranteed to be lowest client descriptor) */
    for (int fd = s_fd + 1; fd < maxfdp1; fd++)
      if (FD_ISSET (fd, &temp_fds)) {
        int n;
        /* Guaranteed not to block in this case! */
        if ((n = handle_logging_record (fd)) == -1)
          ::perror ("logging failed");
        else if (n == 0) {
          /* Handle client connection shutdown */
          FD_CLR (fd, &read_fds);
          ::close (fd);
          if (fd + 1 == maxfdp1) {
            /* Skip past unused descriptors */
            while (!FD_ISSET (--fd, &read_fds))
              continue;
            maxfdp1 = fd + 1;
          }
        }
      }

    if (FD_ISSET (s_fd, &temp_fds)) {
      static struct timeval poll_tv = {0, 0};

      /* Handle all pending connection requests
         (note use of "polling" feature) */
      while (::select (s_fd + 1, &temp_fds,
                      0, 0, &poll_tv) > 0)
        if (listener.accept (new_stream) == -1)
          ::perror ("accept");
        else {
          fd = new_stream.get_fd ();
          FD_SET (fd, &read_fds);
          if (fd >= maxfdp1)
            maxfdp1 = fd + 1;
        }
    }
  }
  /* NOTREACHED */
}
```

Figure 13: An Event Demultiplexing Server Using the select API

```
/* Maximum per-process open I/O descriptors */
const int MAX_FDS = 200;

int
main (void)
{
  /* Create a server end-point */
  SOCK_Listener listener (INET_Addr (PORT_NUM));
  SOCK_Stream new_stream;
  struct pollfd poll_array[MAX_FDS];
  int s_fd = listener.get_fd ();

  /* Check for constructor failure */
  if (s_fd == -1)
    ::perror ("server"), ::exit (1);

  poll_array[0].fd = s_fd;
  poll_array[0].events = POLLIN;

  for (int nfds = 1;;) {
    /* Wait for client I/O events (handle interrupts) */
    while (::poll (poll_array, nfds, -1) == -1
           && errno == EINTR)
      continue;

    /* Handle pending logging messages first
       (poll_array[i = 1].fd is guaranteed to be
       lowest client descriptor) */

    for (int i = 1; i < nfds; i++) {
      if (poll_array[i].revents & POLLIN) {
        char buf[BUFSIZ];
        int  n;
        /* Guaranteed not to block in this case! */
        if ((n = handle_logging_record (poll_array[i].fd))
             == -1)
          ::perror ("read failed");
        else if (n == 0) {
          /* Handle client connection shutdown */
          if (::close (poll_array[i].fd) == -1)
            ::perror ("close");
          poll_array[i].fd = poll_array[--nfds].fd;
        }
      }
    }
    if (poll_array[0].revents & POLLIN) {
      /* Handle all pending connection requests
         (note use of "polling" feature) */
      while (poll (poll_array, 1, 0) > 0)
        if (listener.accept (new_stream, &client) == -1)
          ::perror ("accept");
        else
          {
            poll_array[nfds].fd = POLLIN;
            poll_array[nfds++].fd = new_stream.get_fd ();
          }
    }
  }
  /* NOTREACHED */
}
```

Figure 14: An Event Demultiplexing Server Using the poll API

`poll` interface. For example, unlike `select` (which uses separate `fd_set` bitmasks for reading, writing, and exception events) `poll` uses a single array of `pollfd` structures. In general, the `poll` API is more versatile than `select`, allowing applications to wait for a wider-range of events (such as "priority-band" I/O events and signals). However, the overall complexity and total number of source lines in the two examples is approximately the same.

3.4.4 Limitations with Existing Event Demultiplexing Services

The event demultiplexing services solve several limitations with the alternative approaches presented above. For example, the event demultiplexing-based server logging daemon requires neither "busy-waiting" nor separate process creation. However, there are still a number of problems associated with using either `select` or `poll` directly. This section describes some of the remaining problems and explains how the `Reactor` is designed to overcome these problems.

• **Complicated and Error-Prone Interfaces:** The interfaces for `select` and `poll` are very general, combining several services such as "timed-waits" and multiple I/O event notification within a single system call entry point. This generality increases the complexity of learning and using the I/O demultiplexing facilities correctly. The `Reactor`, on the other hand, provides a less cryptic API consisting of multiple member functions, each of which performs a single well-defined activity.

In addition, as with many OS APIs, the I/O demultiplexing facilities are weakly-typed. This increases the potential for making common mistakes such as not zeroing-out the `fd_set` structure before enabling the I/O descriptor bits, forgetting that the `width` argument to `select` or `poll` is actually the "maximum enabled I/O descriptor *plus 1*," or neglecting to set the value of the `fd` field of a `struct pollfd` to -1 if that I/O descriptor value should be ignored when calling `poll`.

Since applications built upon the `Reactor` framework do not access `select` or `poll` directly, it is not possible to accidentally misuse these underlying system calls. Moreover, the `Reactor` may be used in conjunction with the strongly-typed local and remote communication services provided by the `IPC_SAP` wrapper library [2]. This further reduces the likelyhood for type errors to arise at run-time.

• **Low-Level Interfaces:** The `select` interface is rather low-level, requiring programmers to manipulate up to three different descriptor set bit-masks. Moreover, these bit-masks are passed to the `select` call using "value/result" parameter semantics. Therefore, as shown in Figure 13, the server code must explicitly store the original descriptor set in a scratch variable, pass this variable to the `select` call (which may modify it), examine the results to determine which descriptors became enabled, and potentially update the original descriptor set. The code to implement this logic tends to be tedious

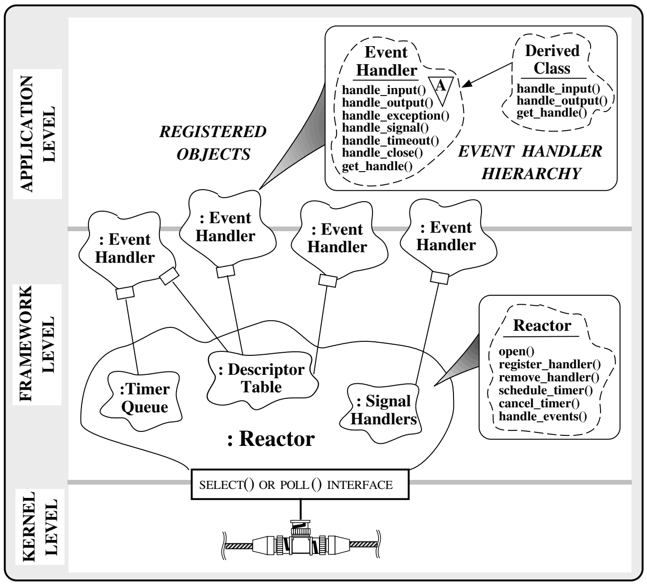

Figure 15: Registering Objects with the Reactor

and prone to subtle errors such as mistakenly updating bits in the wrong descriptor set.

The `Reactor`, on the other hand, completely shields application programmers from such low-level details. As shown in Figure 15, instead of manipulating descriptor set bit-masks, inheritance is used to derive and instantiate composite objects (called "`Event_Handlers`") that perform certain application-defined actions when certain types of events occur. Once instantiated, these `Event_Handler` objects are *registered* with the `Reactor`. The `Reactor` arranges to "call-back" the appropriate member function(s) when (1) I/O events occur on the descriptor associated with the registered object or (2) when timer-based events expire.

Figure 17 depicts the main event-loop of the `Reactor`-based logging server. In this example, a composite class called `Client_Listener` is derived from the `Event_Handler` and `SOCK_Listener` base classes. An instance of this class is then constructed and registered with the `Reactor`. After registration, the server initiates an event-loop that automatically dispatches the `Client_Listener::handle_input` member function when connection requests arrive. Part two of this article describes the design and implementation of this class and the other `Reactor` components in greater detail.

Non-Portable, Non-Standard Interfaces: Although event demultiplexing is not part of the POSIX standard, System V Release 4, BSD UNIX, and WINSOCK all support the `select` API. However, BSD UNIX and WINSOCK do not support `poll`. Likewise, versions of System V prior to Release 4 do not support `select`. Therefore, it is difficult to write portable code that uses event demultiplexing since there are several competing "standards" to choose from, (*i.e.,* BSD and System V UNIX). This often

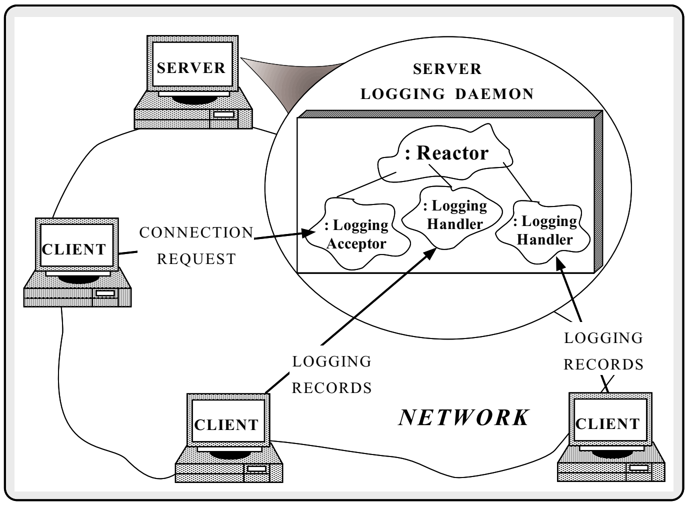

Figure 16: Reactor-based Server

```
/* Pointer to global Reactor object */
static Reactor *reactor;

class Client_Listener :
  public Event_Handler, public SOCK_Listener
{
public:
  /* The following two member functions override
     the virtual functions in the Event_Handler */
  virtual int get_fd (int fd) {
    return SOCK_Listener::get_fd ();
  }
  virtual int handle_input (int fd);
  /* See next article for additional details... */
};

int
main (void)
{
  reactor = Reactor::open ();

  Client_Listener
    listener (INET_Addr (LOGGER_TCP_PORT));

  if (reactor->register_handler (&listener) == -1)
    perror ("reactor");

  /* Loop forever handling logging events */
  for (;;)
    reactor->handle_events ();
}
```

Figure 17: Main Event Loop for Reactor-Based Server Logging Daemon

increases the complexity of developing and maintaining application source code since achieving portability may require the use of conditional compilation that is parameterized by the host OS type.

The `Reactor`, on the other hand, provides a consistent API available across OS platforms. This API not only provides a higher level programming abstraction, but also shields application programs from lexical and syntactic incompatibilities exhibited by the `select` and `poll` demultiplexing mechanisms. Therefore, applications need not maintain multiple source versions or try to merge the event demultiplexing functionality illustrated in Figure 13 and Figure 14 within a single subroutine-based API. Instead, the `Reactor` enables developers to write applications that utilize a single uniform and extensible OO API, which is then mapped onto the appropriate underlying event demultiplexing interface. In this approach, conditional linking may be used in place of conditional compilation to support both `select` and `poll` implementations simultaneously.

Non-Extensible Interfaces: With the event demultiplexing solutions shown in Figure 13 and Figure 14, it is necessary to directly modify the original demultiplexing loop in order to modify or extend application services. With the Reactor, on the other hand, the existing infrastructure code is not modified when applications change their behavior. Instead, inheritance is used to create a new derived class that is instantiated, registered, and invoked automatically by the `Reactor` to perform the appropriate operations.

4 Summary

This article presents the background material necessary to understand the behavior, advantages, and disadvantages of existing UNIX mechanisms for handling multiple sources of I/O in a network application. An OO wrapper called the `Reactor` has been developed to encapsulate and overcome the limitations with the `select` and `poll` event demultiplexing system calls. The object-oriented design and implementation of the `Reactor` is explored in greater detail in part two of this article (appearing in the next C++ Report). In addition to describing the class relationships and inheritance hierarchies, the follow-up article presents an extended example involving the distributed logging facility. This example illustrates how the `Reactor` simplifies the development of event-driven network servers that manage multiple client connections simultaneously.

References

[1] D. C. Schmidt, "Systems Programming with C++ Wrappers: Encapsulating Interprocess Communication Services with Object-Oriented Interfaces," *C++ Report*, vol. 4, September/October 1992.

[2] D. C. Schmidt, "IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*, vol. 4, November/December 1992.

[3] D. C. Schmidt, "The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing (Part 2 of 2)," *C++ Report*, vol. 5, September 1993.

[4] W. R. Stevens, *UNIX Network Programming*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[5] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Vol III: Client – Server Programming and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1992.

[6] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[7] A. D. Birrell, "An Introduction to Programming with Threads," Tech. Rep. SRC-035, Digital Equipment Corporation, January 1989.

[8] W. R. Stevens, *Advanced Programming in the UNIX Environment*. Reading, Massachusetts: Addison Wesley, 1992.