Leader/Followers

Douglas C. Schmidt, Carlos O'Ryan, Michael Kircher, Irfan Pyarali, and Frank Buschmann

{schmidt, coryan}@uci.edu,

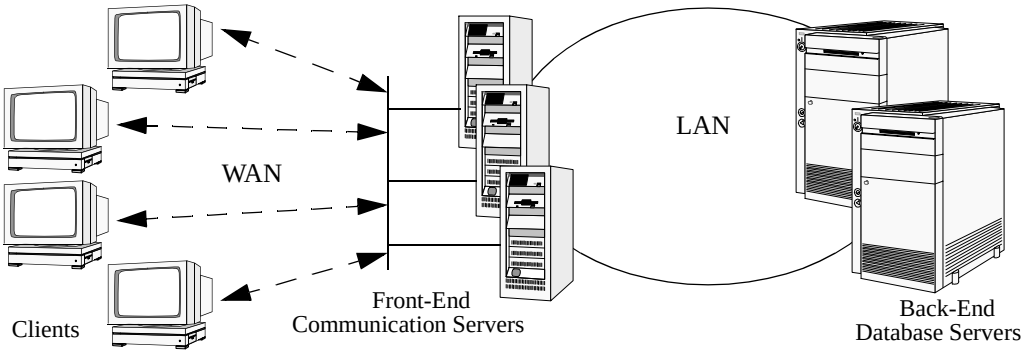
{Michael.Kircher, Frank.Buschmann}@mchp.siemens.de,

irfan@cs.wustl.edu

University of California at Irvine, Siemens AG, and Washington University in Saint Louis

The Leader/Followers architectural pattern provides an efficient concurrency model where multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process service requests that occur on these event sources.

Example Consider the design of a multi-tier, high-volume, on-line transaction processing (OLTP) system. In this design, front-end communication servers route transaction requests from remote clients, such as travel agents, claims processing centers, or point-of-sales terminals, to back-end database servers that process the requests transactionally. After a transaction commits, the database server returns its results to the associated communication server, which then forwards the results back to the originating remote client. This multi-tier architecture is used to improve overall system throughput and reliability via load balancing and redundancy, respectively.It



also relieves back-end servers from the burden of managing different communication protocols with clients.

Front-end communication servers are actually ``hybrid'' client/server applications that perform two primary tasks:

1 They receive requests arriving simultaneously from hundreds or thousands of remote clients over wide area communication links, such as X.25 or TCP/IP.

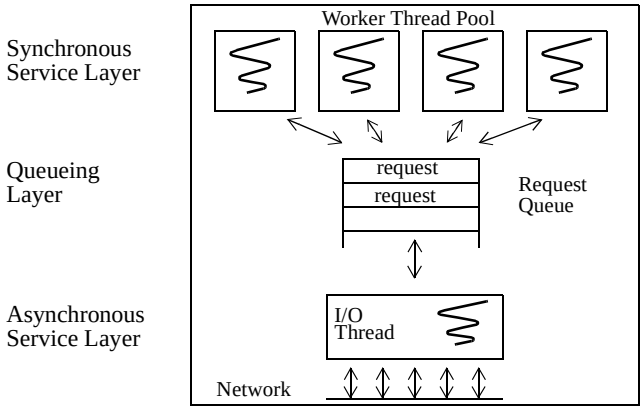
2 They validate the remote client requests and forward valid requests over TCP/IP connections to back-end database servers.

In contrast, the back-end database servers are ``pure'' servers that perform their designated transactions and return results back to clients via front-end communication servers. Both types of OLTP servers spend most of their time processing various types of I/O operations in response to requests.

A common strategy for improving OLTP server performance is to use a multi-threaded concurrency model that processes requests and results simultaneously [HPS99]. In theory, threads can run independently, increasing overall system throughput by overlapping network and disk I/O processing with OLTP computations, such as validations, indexed searches, table merges, triggers, and stored procedure executions. In practice, however, it is challenging to design a multi-threading model that allows front-end and back-end servers to perform I/O operations and OLTP processing efficiently.

One way to multi-thread an OLTP back-end server is to create a thread pool based on the *Half-Sync/Half-Reactive* variant of the Half-Sync/Half-Async pattern [POSA2]. In large-scale OLTP systems, the number of socket handles may be much larger than the number of threads. In this case, an event demultiplexer, such as `select` [Ste97], `poll` [Rago93], or `WaitForMultipleObjects` [Sol98], can be used to wait for events to occur on a socket handle set. However, certain types of event demultiplexers, most notably `select` and `poll`, do not work correctly if invoked with the same handle set by multiple threads. To overcome this limitation, therefore, the server contains a dedicated *network I/O* thread that uses the `select()` [Ste97] event demultiplexer to wait for events to occur on a set of socket handles connected to front-end communication servers.

When activity occurs on handles in the set, `select()` returns control to the network I/O thread and indicates which socket handles in the set have events pending. The I/O thread then reads the transaction requests from the socket handles, stores them into dynamically allocated requests, and inserts these requests into a synchronized message queue implemented using the Monitor Object pattern [POSA2]. This message queue is serviced by a pool of *worker threads*. When a worker thread in the pool is available, it removes a request from the queue, performs the designated transaction, and then returns a response to the front-end communication server.



Although the threading model described above is used in many concurrent applications, it can incur excessive overhead when used for high-volume servers, such as those in our OLTP example. For instance, even with a light workload, the Half-Sync/Half-Reactive thread pool will incur a dynamic memory allocation, multiple synchronization operations, and a context switch to pass a request message between the network I/O thread and a worker thread, which makes even the best-case latency unnecessarily high [PRS+99]. Moreover, if the OLTP back-end server is run on a multi-processor, significant overhead can occur from processor cache coherency protocols used to transfer requests between threads [SKT96].

If the OLTP back-end servers run on an operating system platform that supports asynchronous I/O efficiently, the Half-Sync/Half-Reactive thread pool can be replaced with a purely asynchronous thread pool based on the Proactor pattern [POSA2]. This alternative will reduce much of the synchronization, context switch, and cache coherency overhead outlined above by eliminating the network I/O thread. Many operating systems do not support asynchronous I/O, however, and those that do often support it inefficiently.[1] Yet, it is essential that high-volume OLTP servers demultiplex requests efficiently to threads that can process the results concurrently.

Context An event-driven application where multiple service requests contained in events arrrive from a set of event sources and must be processed efficiently by multiple threads that share the event sources.

Problem Multi-threading is a common technique to implement applications that process multiple events concurrently. Implementing *high-performance* multi-threaded server

1. For instance, UNIX operating systems often support asynchronous I/O by spawning a thread for each asynchronous operation, thereby defeating the potential performance benefits of asynchrony.

applications is hard, however. These applications often process a high volume of multiple types of events, such as the `CONNECT`, `READ`, and `WRITE` events in our OLTP example, that arrive simultaneously. To address this problem effectively, the following three *forces* must be resolved:

- Service requests can arrive from multiple event sources, such as multiple TCP/IP socket handles [Ste97], that are allocated for each connected client. A key design force, therefore, is determining efficient *demultiplexing associations* between threads and event sources. In particular, associating a thread for each event source may be infeasible due to the scalability limitations of applications or the underlying operating system and network platforms.

  Â For our OLTP server applications, it may not be practical to associate a separate thread with each socket handle. In particular, as the number of connections increase significantly, this design may not scale efficiently on many operating system platforms. o

- To maximize performance, key sources of concurrency-related overhead, such as context switching, synchronization, and cache coherency management, must be minimized. In particular, concurrency models that allocate memory dynamically for each request passed between multiple threads will incur significant overhead on conventional multi-processor operating systems.

  Â For instance, implementing our OLTP servers using the *Half-Sync/Half-Reactive* [POSA2] outlined in the *Example* section requires memory to be allocated dynamically in the network I/O thread to store incoming transaction requests into the message queue. This design incurs numerous synchronizations and context switches to insert the request into, or remove the request from, the message queue. o

- Multiple threads that demultiplex events on a shared set of event sources must coordinate to prevent *race conditions*. Race conditions can occur if multiple threads try to access or modify certain types of event sources simultaneously.

  Â For instance, a pool of threads cannot use `select()` [Ste97] to demultiplex a set of socket handles because the operating system will erroneously notify more than one thread calling `select()` when I/O events are pending on the same set of socket handles [Ste97]. Moreover, for bytestream-oriented protocols, such as TCP, having multiple threads invoking `read()` or `write()` on the same socket handle will corrupt or lose data. o

Solution Structure a pool of threads to share a set of event sources efficiently by *taking turns* demultiplexing events that arrive on these event sources and synchronously dispatching the events to application services that process them.

In detail: design a *thread pool* mechanism that allows multiple threads to coordinate themselves and protect critical sections while detecting, demultiplexing, dispatching, and processing events. In this mechanism, allow one thread at a time—the *leader*—to wait for an event to occur from a *set of event sources*. Meanwhile, other threads—the *followers*—can queue up waiting their turn to become the leader. After the current leader thread detects an event from the event source set, it first promotes a follower thread to become the new leader. It then plays the role of a *processing* thread, which demultiplexes and dispatches the event to a designated *event handler* that performs application-specific event handling in the processing thread. Multiple processing threads can handle events concurrently while the current leader thread waits for new events on the set of event sources shared by the threads. After handling its event, a processing thread reverts to a follower role and waits to become the leader thread again.

Structure The participants in the Leader/Followers pattern include the following:

Handles are provided by operating systems to identify event sources, such as network connections or open files, that can generate and queue events. Events can originate from external sources, such as `CONNECT` events or `READ` events sent to a service from clients, or internal sources, such as time-outs. A *handle set* is a collection of handles that can be used to wait for one or more events to occur on handles in the set. A handle set returns to its caller when it is possible to initiate an operation on a handle in the set without the operation blocking.

Â For example, OLTP servers are interested in two types of events—`CONNECT` events and `READ` events—which represent incoming connections and transaction requests, respectively. Both front-end and back-end servers maintain a separate connection for each client, where clients of front-end servers are the so-called ‘remote’ clients and front-end servers themselves are clients of back-end servers. Each connection is a source of events that is represented in a server by a separate socket handle. Our OLTP servers use the `select()` event demultiplexer, which

identifies handles whose event sources have pending events, so that applications can invoke I/O operations on these handles without blocking the calling threads. o

Class Handle and Handle Set	***Collaborator***
Responsibility • A handle identifies a source of events in an operating system • A handle can queue up events • A handle set is a collection of handles	

An *event handler* specifies an interface consisting of one or more hook methods [Pree95] [GoF95]. These methods represent the set of operations available to process application-specific events that occur on handle(s) serviced by an event handler.

Concrete event handlers specialize from the event handler and implement a specific service that the application offers. In particular, concrete event handlers implement the hook method(s) responsible for processing events received from a handle.

Class Event Handler	***Collaborator*** • Handle
Responsibility • Defines an interface for processing events that occur on a handle	

Class Concrete Event Handler	***Collaborator*** • Handle
Responsibility • Defines an application service • Processes events received on a handle in an application-specific manner • Runs in a processing thread	

Â For example, concrete event handlers in OLTP front-end communication servers receive and validate remote client requests, and then forward requests to back-end database servers. Likewise, concrete event handlers in back-end database servers receive transaction requests from front-end servers, read/write the appropriate database records to perform the transactions, and return the results to the

front-end servers. All network I/O operations are performed via socket handles, which identify various sources of events. o

At the heart of the Leader/Followers pattern is a *thread pool*, which is a group of threads that share a synchronizer, such as a semaphore or condition variable, and implement a protocol for coordinating their transition between various roles. One or more threads play the *follower* role and queue up on the thread pool synchronizer waiting to play the *leader* role. One of these threads is selected to be the leader, which waits for an event to occur on any handle in its handle set. When an event occurs the following activities occur:

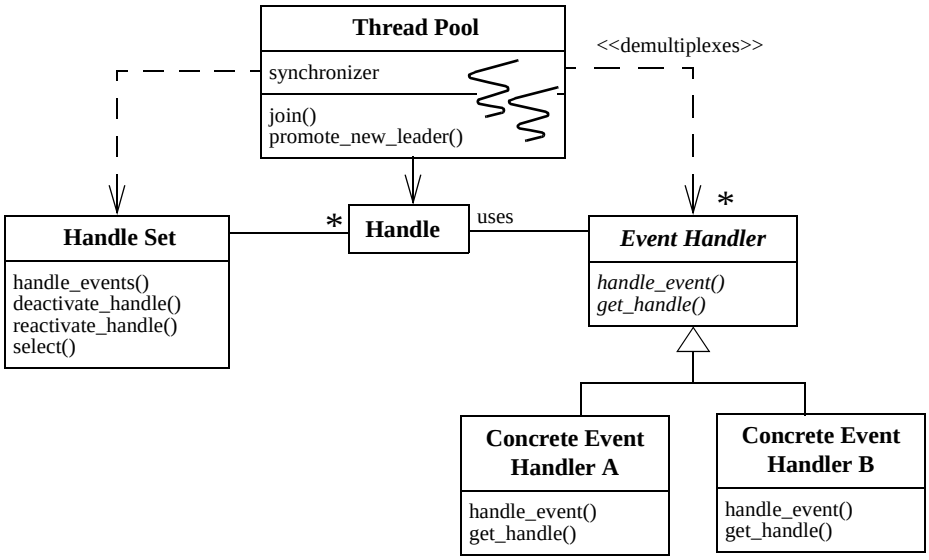
- The current leader thread promotes a follower thread to become the new leader
- The original leader then concurrently plays the role of a *processing thread*, which demultiplexes that event from the handle set to an appropriate event handler and dispatches the handler's hook method to handle the event
- After a processing thread is finished handling an event, it returns to playing the role of a follower thread and waits on the thread pool synchronizer for its turn to become the leader thread again

Class	***Collaborator***
Thread Pool	• Handle Set • Handle • Event Handlers
Responsibility • Threads that take turns playing three roles: in the leader role they await events, in the processing role they process events, in the follower role they queue up to become the leader • Contains a synchronizer	

Â For example, each OLTP server designed using the Leader/Followers pattern can have a pool of threads waiting to process transaction requests that arrive on event sources identified by a handle set. At any point in time, multiple threads in the pool can be processing transaction requests and sending results back to their clients. One thread in the pool is the current leader, which waits for a new `CONNECT` and `READ` event to arrive on the handle set shared by the threads. When this occurs, the leader

thread becomes a processing thread and handles the event, while one of the follower threads in the pool is promoted to become the new leader. o

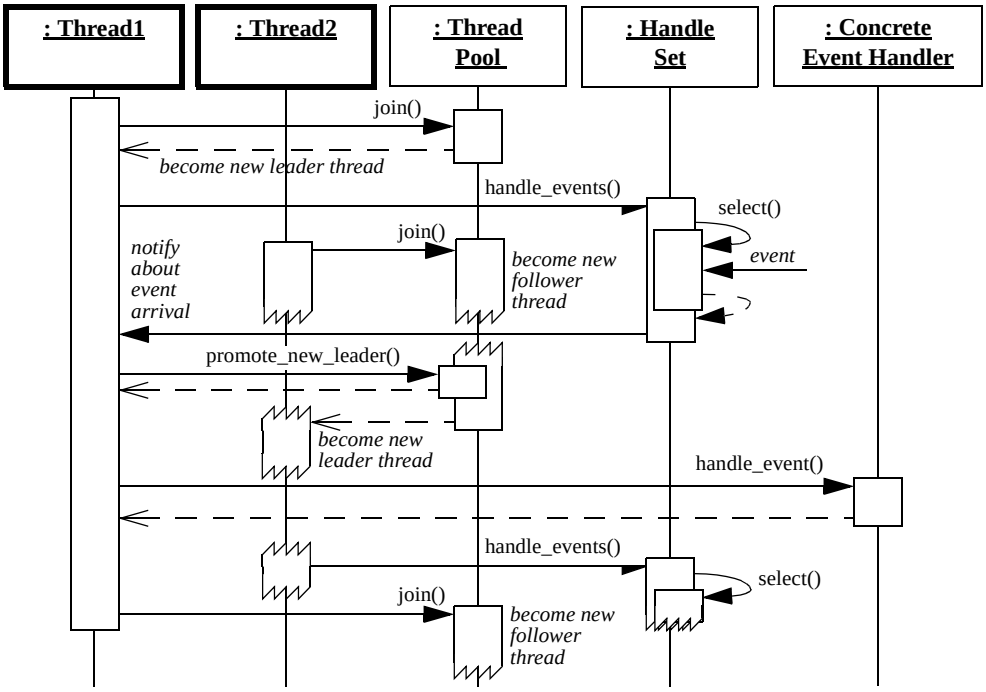
The following UML class diagram illustrates the structure of participants in the Leader/Followers pattern. In this structure, multiple threads share the same instances of the thread pool, event handler, and handle set participants. The thread pool participant ensures the correct and efficient coordination of the threads.



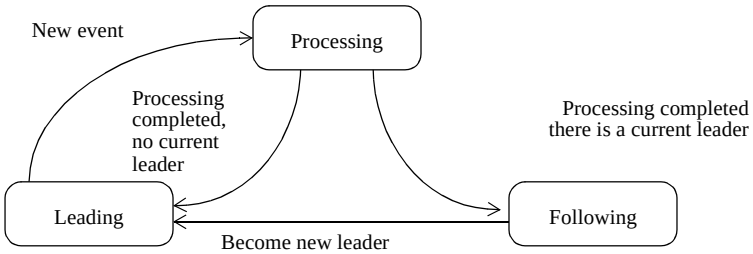
Dynamics The following collaborations occur in the Leader/Followers pattern.

- *Leader thread demultiplexing*. The leader thread waits for an event to occur on any handle in the handle set. If there is no current leader thread, for example, due to events arriving faster than the available threads can service them, the underlying operating system can queue events internally until a leader thread is available.
- *Follower thread promotion*. After the leader thread has detected an new event, it uses the thread pool to choose a follower thread to become the new leader by using one of the promotion protocols described in implementation activity 5.
- *Event handler demultiplexing and event processing*. After helping to promote a follower thread to become the new leader, the former leader thread then plays the role of a processing thread. This thread concurrently demultiplexes the event it detected to the event's associated handler and then dispatches the handler's hook method to process the event. A processing thread can execute concurrently with the leader thread and any other threads that are in the processing state.

- *Rejoining the thread pool*. After the processing thread has run its event handling to completion, it can rejoin the thread pool and wait to process another event. A processing thread can become the leader immediately if there is no current leader thread. Otherwise, the processing thread returns to playing the role of a follower thread and waits on the thread pool synchronizer until it is promoted by a leader thread.



A thread's transitions between states can be visualized in the following diagram:



Implementation The following activities can be used to implement the Leader/Followers pattern.

1 *Choose the handle and handle set mechanisms*. A handle set is a collection of handles that a leader thread can use to wait for an event to occur from a set of event sources, such as TCP/IP sockets. Developers often choose the handles and handle set mechanisms provided by the underlying operating system, rather than implementing them from scratch. The following sub-activities can be performed to choose the handle and handle set mechanisms.

1.1 *Determine the type of handles*. There are two general types of handles:

- *Concurrent handles*. This type allows multiple threads to access a handle to an event source *concurrently* without incurring race conditions that can corrupt, lose, or scramble the data [Ste97]. For instance, the Socket API for record-oriented protocols, such as UDP, allows multiple threads to invoke `read()` or `write()` operations on the same handle concurrently.
- *Iterative handles*. This type requires multiple threads to access a handle to an event source *iteratively* because concurrent access will incur race conditions. For instance, the Socket API for bytestream-oriented protocols, such as TCP, does not guarantee that `read()` or `write()` operations respect application-level message boundaries. Thus, corrupted or lost data can result if I/O operations on the socket are not serialized properly.

1.2 *Determine the type of handle set*. There are two general types of handle sets:

- *Concurrent handle set*. This type can be acted upon concurrently, for example, by a pool of threads. When it is possible to initiate an operation on *one* handle without blocking the operation, a concurrent handle set returns that handle to one of its calling threads. For example, the Win32 `WaitForMultipleObjects()` function [Sol98] supports concurrent handle sets by allowing a pool of threads to wait on the same set of handles simultaneously.
- *Iterative handle set*. This type returns to its caller when it is possible to initiate an operation on *one or more* handles in the set without the operation(s) blocking. Although an iterative handle set can return multiple handles in a single call, it cannot be called simultaneously by multiple threads of control. For example, the `select()` [Ste97] and `poll()` [Rago93] functions support iterative handle sets. Thus, a pool of threads cannot use `select()` or `poll()` to demultiplex events on the same handle set concurrently because multiple threads can be notified that the same I/O events are pending, which elicits erroneous behavior.

The following table summarizes representative examples for each combination of concurrent and iterative handles and handle sets:

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

1.3 *Determine the consequences of selecting certain handle and handle set mechanisms.* In general, the Leader/Followers pattern is used to prevent multiple threads from corrupting or losing data erroneously, such as invoking read operations on a shared TCP bytestream socket handle concurrently or invoking `select()` on a shared handle set concurrently. However, some application use cases need not guard against these problems. In particular, if the handle and handle set mechanisms are both concurrent, many of the following implementation activities can be skipped.

As discussed in *implementation activities* 1.1 and 1.2, the semantics of certain combinations of protocols and network programming APIs support concurrent multiple I/O operations on a shared handle. For example, UDP support in the Socket API ensures a complete message is always read or written by one thread or another, without the risk of a partial `read()` or of data corruption from an interleaved `write()`. Likewise, certain handle set mechanisms, such as the Win32 `WaitForMultipleObjects()` function [Sol98], return a single handle per call, which allows them to be called concurrently by a pool of threads.[2]

In these situations, it may be possible to implement the Leader/Followers pattern by simply using the operating systems thread scheduler to (de)multiplex threads, handle sets, and handles robustly, in which case, implementation activities 2 through 6 can be skipped.

2. However, `WaitForMultipleObjects()` does not by itself address the problem of notifying a particular thread when an event is available, which is necessary to support the bound thread/handle association discussed in the Variants section.

1.4 *Implement an event handler demultiplexing mechanism*. In addition to calling an event demultiplexer to wait for one or more events to occur on its handle set, such as `select()` [Ste97], a Leader/Followers pattern implementation must demultiplex events to event handlers and dispatch their hook methods to process the events. The following are two alternative strategies that can be used to implement this mechanism:

- *Program to a low-level operating system event demultiplexing mechanism*. In this strategy, the handle set demultiplexing mechanisms provided by the operating system are used directly. Thus, a Leader/Follower implementation must implement a demultiplexing table that is a manager [PLoPD3] containing a set of <*handle, event handler, event types*> tuples. Each handle serves as a 'key' that associates handles with event handlers in its demultiplexing table, which also stores the type of event(s), such as `CONNECT` and `READ`, that each event handler will process. The contents of this table are converted into handle sets passed to the native event demultiplexing mechanism, such as `select()` [Ste97] or `WaitForMultipleObjects()` [Sol98].

  The demultiplexing table can be implemented using various search strategies, such as direct indexing, linear search, or dynamic hashing. If handles are represented as a continuous range of integer values, such as on UNIX platforms, direct indexing is most efficient since demultiplexing table tuple entries can be located in constant *O(1)* time. On platforms, such as Win32, where handles are non-contiguous pointers, direct indexing is not feasible and some type of linear search or dynamic hashing must be used to implement a demultiplexing table.

  Â For instance, handles in UNIX are contiguous integer values, which allows our demultiplexing table to be implemented as a fixed-size array of structs. In this design, the handle values themselves index directly into the demultiplexing table's array to locate event handlers or event registration types in constant time. The following class illustrates such an implementation:

```
class Demux_Table {
// Maps <Handle>s to <Event_Handler>s and <Event_Type>s.
public:
  // Convert <Tuple> array to <fd_set>s.
  int convert_to_fd_sets (fd_set &read_fds,
                          fd_set &write_fds,
                          fd_set &except_fds);
  struct Tuple {
    // Pointer to <Event_Handler> that processes
    // the indication events arriving on the handle.
    Event_Handler *event_handler_;
```

```
    // Bit-mask that tracks which types of indication
    // events <Event_Handler> is registered for.
    Event_Type event_type_;
  };

  // Table of <Tuple>s indexed by Handle values. The
  // macro FD_SETSIZE is typically defined in the
  // <sys/socket.h> system header file.
  Tuple table_[FD_SETSIZE];
};
```

In this simple implementation, the `Demux_Table`'s `table_` array is indexed by UNIX handle values, which are unsigned integers ranging from 0 to `FD_SETSIZE - 1`. o

- *Program to a higher-level event demultiplexing pattern*. In this strategy, developers leverage higher-level patterns, such as Reactor, Proactor, and Wrapper Facade, which are defined in [POSA2]. These patterns help to simplify the Leader/Followers implementation and reduce the effort needed to address the accidental complexities of programming to native operating system handle set demultiplexing mechanisms directly. Moreover, applying higher-level patterns makes it easier to decouple the I/O and demultiplexing aspects of a system from its concurrency model, thereby reducing code duplication and maintenance effort.

  Â For example, in our OLTP server example, an event must be demultiplexed to the concrete event handler associated with the socket handle that received the event. The Reactor pattern [POSA2] supports this activity, thereby simplifying the implementation of the Leader/Followers pattern. In the context of the Leader/Followers pattern, however, a reactor only demultiplexes *one* handle to its concrete event handler, regardless of how many handles have events pending on them. The following C++ class illustrates the interface of our Reactor pattern implementation:

```
typedef unsigned int Event_Types;

enum {
  // Types of indication events handled by the
  // <Reactor>. These values are powers of two so
  // their bits can be "or'd" together efficiently.
  ACCEPT_EVENT = 01, // ACCEPT_EVENT is an
  READ_EVENT = 01,   // alias for READ_EVENT.
  WRITE_EVENT = 02, TIMEOUT_EVENT = 04,
  SIGNAL_EVENT = 010, CLOSE_EVENT = 020
};
```

```
class Reactor {
public:
  // Register an <Event_Handler> of a
  // particular <Event_Type>.
  int register_handler (Event_Handler *eh,
                        Event_Type et);
  // Remove an <Event_Handler> of a
  // particular <Event_Type>.
  int remove_handler (Event_Handler *eh,
                      Event_Type et);
  // Entry point into the reactive event loop.
  int handle_events (Time_Value *timeout = 0);
};
```

Developers provide concrete implementations of the `Event_Handler` interface below:

```
class Event_Handler {
public:
     //  Hook method dispatched by a <Reactor> to
         // handle events of a particular type.
  virtual int handle_event (HANDLE,
                            Event_Type et) = 0;

  // Hook method returns the <HANDLE>.
  virtual HANDLE get_handle (void) const = 0;
};
```

o

2 *Implement a protocol for temporarily (de)activating handles in a handle set.* When an event arrives, the leader thread deactivates the handle from consideration in the handle set temporarily, promotes a follower thread to become the new leader, and continues to process the event. Deactivating the handle from the handle set temporarily avoids race conditions that could otherwise occur between the time when a new leader is selected and the event is processed. If the new leader waits on the same handle in the handle set during this interval, it could demultiplex the event a second time, which is erroneous because the dispatch is already in progress. After the event is processed, the handle is reactivated in the handle set, which allows the leader thread to wait for an event to occur on it or any other activated handles in the set.

Â In our OLTP example, this handle deactivation and reactivation protocol can be provided by extending the `Reactor` interface defined in implementation activity 2 of the Reactor pattern [POSA2] as follows:

```
class Reactor {
public:
    // Temporarily deactivate the <HANDLE>
    // from the internal handle set.
    int deactivate_handle (HANDLE, Event_Type et);
```

```
        // Reactivate a previously deactivated
        // <Event_Handler> to the internal handle set.
        int reactivate_handle (HANDLE, Event_Type et);

        // ...
    };                                                          o
```

3 *Implement the thread pool*. To promote a follower thread to the leader role, as well as to determine which thread is the current leader, an implementation of the Leader/Followers pattern must manage a pool of threads. A straightforward way to implement this is to have all the follower threads in the set simply wait on a single synchronizer, such as a semaphore or condition variable. In this design, it does not matter which thread processes an event, as long as all threads in the pool that share the handle set are serialized.

Â For example, the `LF_Thread_Pool` class shown below can be used for the back-end database servers in our OLTP example:

```
class LF_Thread_Pool {
public:
    // By default, use a singleton reactor.
    LF_Thread_Pool (Reactor *reactor =
                    Reactor::instance ()):
        reactor_ (reactor) {}

    // Wait on handle set and demultiplex events
    // to their event handlers.
    int join (Time_Value *timeout = 0);

    // Promote a follower thread to become the
    // leader thread.
    int promote_new_leader (void);

    // Support the <HANDLE> (de)activation protocol.
    int deactivate_handle (HANDLE, Event_Type et);
    int reactivate_handle (HANDLE, Event_Type et);
private:
    // Pointer to the event demultiplexer/dispatcher.
    Reactor *reactor_;

    // The thread id of the leader thread, which is
    // set to NO_CURRENT_LEADER if there is no leader.
    Thread_Id leader_thread_;

    // Follower threads wait on this condition
    // variable until they are promoted to leader.
    Thread_Condition followers_condition_;

    // Serialize access to our internal state.
    Thread_Mutex mutex_;
```

```
};
```

The constructor of `LF_Thread_Pool` caches the reactor passed to it. By default, this reactor implementation uses `select()`, which supports iterative handle sets. Therefore, `LF_Thread_Pool` is responsible for serializing multiple threads that take turns calling `select()` on the reactors handle set. o

Application threads invoke `join()` to wait on a handle set and demultiplex new events to their associated event handlers. As shown in implementation activity 4, this method does not return to its caller until the application terminates or `join()` times out. The `promote_new_leader()` method promotes one of the follower threads in the set to become the new leader, as shown in implementation activity 5.2.

The `deactivate_handle()` method and the `reactive_handle()` method deactivate and activate handles within a reactor's handle set temporarily. The implementations of these methods simply forward to the same methods defined in the `Reactor` interface shown in implementation activity 2.

Note that a single `followers_condition_` condition variable synchronizer is shared by all threads in this thread pool. As shown in implementation activities 4 and 5, the implementation of `LF_Thread_Pool` is designed using the Monitor Object pattern [POSA2]. o

4 *Implement a protocol to allow threads to initially join (and later rejoin) the thread pool*. This protocol is used in the following two cases:

- After the initial creation of a pool of threads that retrieve and process events; and
- After a processing thread completes and is available to handle another event.

If no leader thread is available, a processing thread can become the leader immediately. If a leader thread is already available, a thread can become a follower by waiting on the thread pool's synchronizer.

Â For example, our back-end database servers can implement the following `join()` method of the `LF_Thread_Pool` to wait on a handle set and demultiplex new events to their associated event handlers:

```
int LF_Thread_Pool::join (Time_Value *timeout) {
     // Use Scoped Locking idiom to acquire mutex
     // automatically in the constructor.
     Guard<Thread_Mutex> guard (mutex_);

     for (;;) {
          while (leader_thread_ != NO_CURRENT_LEADER)
          // Sleep and release <mutex> atomically.
          if (followers_condition_.wait (timeout) == -1
               && errno == ETIME)
```

```
                return -1;

            // Assume the leader role.
            leader_thread_ = Thread::self ();

            // Leave monitor temporarily to allow other
            // follower threads to join the pool.
            guard.release ();

            // After becoming the leader, the thread uses
            // the reactor to wait for an event.
            if (reactor_->handle_events () == -1)
                return;

            // Reenter monitor to serialize the test
            // for <leader_thread_> in the while loop.
            guard.acquire ();
        }
    }
```

Within the `for` loop, the calling thread alternates between its role as a `leader`, `processing`, and `follower` thread. In the first part of this loop, the thread waits until it can be a leader, at which point it uses the reactor to wait for an event on the shared handle set. When the reactor detects an event on a handle, it will demultiplex the event to its associated event handler and dispatch its `handle_event()` method to process the event. After the reactor demultiplexes one event, the thread re-assumes its follower role. These steps continue looping until the application terminates or a timeout occurs. o

5 *Implement the follower promotion protocol*. Immediately after a leader thread detects an event, but before it demultiplexes the event to its event handler and processes the event, it must promote a follower thread to become the new leader. The following two sub-activities can be used to implement this protocol.

5.1 *Implement the handle set synchronization protocol*. If the handle set is iterative and we blindly promote a new leader thread, it is possible that the new leader thread will attempt to handle the same event that was detected by the previous leader thread that is in the midst of processing the event. To avoid this race condition, we must remove the handle from consideration in the handle set before promoting a new follower and dispatching the event to its concrete event handler. The handle must be reactivate in the handle set after the event has been dispatched and processed.

Â An application can implement concrete event handlers that subclass from the `Event_Handler` class defined in the Reactor pattern [POSA2]. Likewise, the Leader/Followers implementation can use the Decorator pattern [GoF95] to create an `LF_Event_Handler` class that decorates `Event_Handler`. This decorator

promotes a new leader thread and activates/deactivates the handler in the reactor's handle set transparently to the concrete event handlers.

```
class LF_Event_Handler : public Event_Handler {
private:
     // This use of <Event_Handler> plays the
     // <ConcreteComponent> role in the Decorator
     // pattern, which is used to implement
     // the application-specific functionality.
     Event_Handler *concrete_event_handler_;
     // Instance of an <LF_Thread_Pool>.
     LF_Thread_Pool *thread_pool_;
public:
     LF_Event_Handler (Event_Handler *eh,
                       LF_Thread_Pool *tp)
          : concrete_event_handler_ (eh),
            thread_pool_ (tp) {}

     virtual int handle_event (HANDLE h, Event_Type et) {
          // Temporarily deactivate the handler in the
          // reactor to prevent race conditions.
          thread_pool_->deactivate_handle (h, et);

          // Promote a follower thread to become leader.
          thread_pool_->promote_new_leader ();

          // Dispatch application-specific event
          // processing code.
          concrete_event_handler_->handle_event (h, et);

          // Reactivate the handle in the reactor.
          thread_pool_->reactivate_handle (h, et);
     }
};
```

As shown above, an application can implement concrete event handlers that subclass from `Event_Handler`. Likewise, the Leader/Followers implementation can use the Decorator pattern to promote a new leader thread and activate/deactivate the handler in the reactor's handle set transparently to developers of concrete event handlers. o

5.2 *Determine the promotion protocol ordering.* The following ordering can be used to determine which follower thread to promote.

- *LIFO order.* In many applications, it does not matter which of the follower threads is promoted next because all threads are equivalent peers. In this case, the leader thread can promote follower threads in *last-in, first-out* (LIFO) order. The LIFO protocol maximizes CPU cache affinity by ensuring that the thread waiting the *shortest* time is promoted first [Sol98], which is an example of the Fresh Work

Before Stale pattern [Mes96]. Implementing a LIFO promotion protocol requires an additional data structure, however, such as a stack of waiting threads, rather than just using a native operating system synchronization object, such as a semaphore.

- *Priority order.* In some applications, particularly real-time applications, threads may run at different priorities. In this case, therefore, it may be necessary to promote follower threads according to their priority. This protocol can be implemented using some type of priority queue, such as a heap [BaLee98]. Although this protocol is more complex than the LIFO protocol, it may be necessary to promote follower threads according to their priorities in order to minimize priority inversion [SMFG00].
- *Implementation-defined order.* This ordering is most common when implementing handle sets using operating system synchronizers, such as semaphores or condition variables, which often dispatch waiting threads in an implementation-defined order. The advantage of this protocol is that it maps onto native operating system synchronizers efficiently.

Â For example, our OLTP back-end database servers could use the following simple protocol to promote follower thread in whatever order they are queued by a native operating system condition variable:

```
int LF_Thread_Pool::promote_new_leader (void) {
     // Use Scoped Locking idiom to acquire mutex
     // automatically in the constructor.
     Guard<Thread_Mutex> guard (mutex_);

     if (leader_thread_ != Thread::self ())
          // Error, only the leader thread can call this.
          return -1;

     // Indicate that we are no longer the leader
     // and notify a <join> method to promote
     // the next follower.
     leader_thread_ = NO_CURRENT_LEADER;
     followers_condition_.notify ();

     // Release mutex automatically in destructor.
}
```

As shown in implementation activity 5.1, the `promote_new_leader()` method is invoked by a `LF_Event_Handler` decorator before it forwards to the concrete event handler that processes an event. o

6 *Implement the event handlers*. Application developers must decide what actions to perform when the hook method of a concrete event handler is invoked by a processing thread in the Leader/Followers pattern implementation. Implementation activity 5 in the Reactor pattern (193) describes various issues associated with implementing concrete event handlers.

Example Resolved The OLTP back-end database servers described in the *Example* section can use the Leader/Followers pattern to implement a thread pool that demultiplexes I/O events from socket handles to their event handlers efficiently. In this design, there is no designated network I/O thread. Instead, a pool of threads is pre-allocated during database server initialization, as follows:

```
const int MAX_THREADS = ...;

// Forward declaration.
void *worker_thread (void *);

int main (void) {
     LF_Thread_Pool thread_pool (Reactor::instance ());
     // Code to set up a passive-mode Acceptor omitted.
     for (int i = 0; i < MAX_THREADS - 1; i++)
          Thread_Manager::instance ()->spawn
               (worker_thread, &thread_pool);

     // The main thread participates in the thread pool.
     thread_pool.join ();
};
```

These threads are not bound to any particular socket handle. Thus, all threads in this pool take turns playing the role of a network I/O thread by invoking the `LF_Thread_Pool::join()` method, as follows:

```
void *worker_thread (void *arg) {
     LF_Thread_Pool *thread_pool =
          reinterpret_cast <LF_Thread_Pool *> (arg);

     // Each worker thread participates in the thread pool.
     thread_pool->join ();
};
```

As shown in implementation activity 4, the `join()` method allows only the leader thread to use the `Reactor` singleton to `select()` on a shared handle set of sockets connected to OLTP front-end communication servers. If requests arrive when all threads are busy, they will be queued in socket handles until threads in the pool are available to execute the requests.

When a request event arrives, the leader thread deactivates the socket handle temporarily from consideration in `select()`'s handle set, promotes a follower

thread to become the new leader, and continues to handle the request event as a processing thread. This processing thread then reads the request into a buffer that resides in the run-time stack or is allocated using the Thread-Specific Storage pattern [POSA2].[3] All OLTP activities occur in the processing thread. Thus, no further context switching, synchronization, or data movement is necessary until the processing completes. When it finishes handling a request, the processing thread returns to playing the role of a follower and waits on the synchronizer in the thread pool. Moreover, the socket handle it was processing is reactivated in the `Reactor` singleton's handle set so that `select()` can wait for I/O events to occur on it, along with other sockets in the handle set.

Variants *Bound Handle/Thread Associations.* The earlier sections in this pattern describe *unbound* handle/thread associations, where there is no fixed association between threads and handles. Thus, any thread can process any event that occurs on any handle in a handle set. Unbound associations are often used when a pool of worker threads take turns demultiplexing a shared handle set.

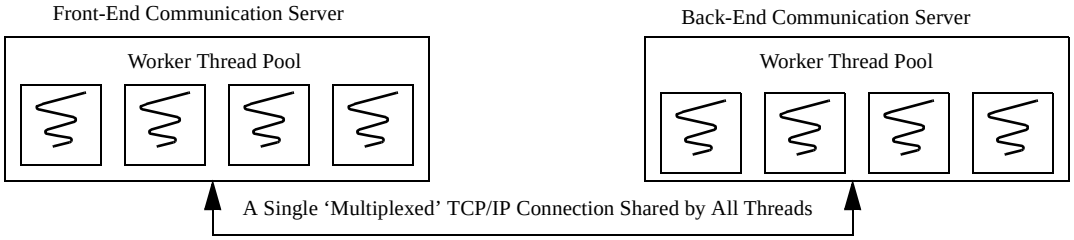
Â For example, our OLTP back-end database server example illustrates an unbound association between threads in the pool and handles in the handle set managed by `select()`. Concrete event handlers that process request events in a database server can run in any thread. Thus, there is no need to maintain bound associations between handles and threads. In this case, maintaining an unbound thread/handle association simplifies back-end server programming. o

An important variant of the Leader/Followers pattern uses *bound* handle/thread associations. In this use case, each thread is bound to its own handle, which it uses to process particular events. Bound associations are often used in the client-side of an application when a thread waits on a socket handle for a response to a two-way request it sent to a server. In this case, the client application thread expects to process the response event on this handle in the same thread that sent the original request.

Â For example, threads in our OLTP front-end communication server forward incoming client requests to a specific back-end server chosen to process the request. To reduce the consumption of operating system resources in large-scale multi-tier OLTP systems, worker threads in front-end server processes can communicate to

3. In contrast, the Half-Sync/Half-Reactive thread pool described in the *Example* section must allocate each request dynamically from a shared heap because the request is passed between threads.

back-end servers using *multiplexed connections* [SMFG00], as shown in the following figure.



After a request is sent, the worker thread waits for a result to return on a multiplexed connection to the back-end server. In this case, maintaining a bound thread/handle association simplifies front-end server programming and minimizes unnecessary context management overhead for transaction processing [OMG97b]. o

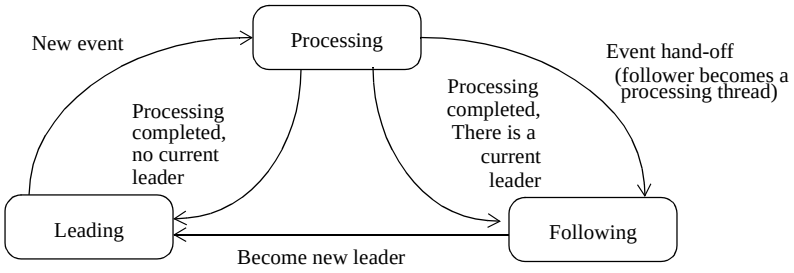
Supporting bound thread/handle associations requires changes to the following sections of the Leader/Followers pattern.

Structure. In the bound handle/thread association variant, the leader thread in the thread pool may need to hand-off an event to a follower thread if the leader does not have the necessary context to process the event. Thus, the follower threads wait to either become the leader thread or to receive event hand-offs from the current leader thread. The leader/follower thread pool can be maintained *implicitly*, for example, using a synchronizer, such as a semaphore or condition variable, or *explicitly*, using a container and the Manager pattern [PLoPD3]. The choice depends largely on whether the leader thread must notify a specific follower thread explicitly to perform event hand-offs.

Dynamics. The bound handle/thread association variant of the Leader/Followers pattern requires changes to the following two collaborations:

- *Follower thread promotion*: After the leader detects a new event, it checks the handle associated with the event to determine which thread is responsible for processing it. If the leader thread discovers that it is responsible for the event, it promotes a follower thread to become the new leader using the same protocols described in implementation activity 5 above. Conversely, if the event is intended for another thread, the leader must hand-off the event to the designated follower thread. This follower thread can then process the event. Meanwhile, the current leader thread continues to wait for another event to occur on the handle set.
- *Event handler demultiplexing and event processing*: Either the processing thread continues to handle the event it detected or a follower thread processes the event

that the leader thread handed off to it. The following diagram illustrates the additional transition between the following state and the processing state:



Implementation The *implementation activities* 1, 2, and 4 in the earlier *Implementation* section require no changes. However, the following changes are required to support bound handle/thread associations in the other *implementation activities*.

3 *Implement the thread pool*. In the bound handle/thread design a leader thread can hand-off new events to specific follower threads. For example, a reply received over a multiplexed connection by the leader thread in a front-end OLTP communication server may belong to one of the follower threads. This scenario is particularly relevant in high-volume, multi-tier distributed systems, where responses often arrive in a different order than requests were initiated.

In addition, a bound handle/thread association may be necessary if an application multiplexes connections among two or more threads, in which case the thread pool can serialize access to the multiplexed connection. This multiplexed design minimizes the number of network connections used by the front-end server. However, front-end server threads must now serialize access to the connection when sending and receiving over a multiplexed connection to avoid corrupting the request and reply data, respectively.

Â For example, below we illustrate how a bound handle/thread association implementation of the Leader/Followers pattern can be used for the front-end communication servers in our OLTP example. We focus on how a server can demultiplex events on a single *iterative handle*, which threads in front-end communication servers use to wait for responses from back-end data servers. This example complements the implementation shown in the thread pool in the earlier *Implementation* section, where we illustrated how to use the Leader/Followers pattern to demultiplex an iterative *handle set*.

We first define a `Thread_Context` class:

```
class Thread_Context {
```

```
public:
     // The response we are waiting for.
     int request_id (void) const;
     // Returns true when response is received.
     bool response_received (void);
     void response_received (bool);
     // The condition the thread waits on.
     Thread_Condition *condition (void);
private:   // ... data members omitted for brevity ...
};
```

A `Thread_Context` provides a separate condition variable synchronizer for each waiting thread, which allows a leader thread to notify the appropriate follower thread when its response is received.

Next, we define the `Bound_LF_Thread_Pool` class:

```
class Bound_LF_Thread_Pool {
public:
     Bound_LF_Thread_Pool (Reactor *reactor)
          : reactor_ (reactor),
          leader_thread_ (NO_CURRENT_LEADER) {}

      // Register <context> into the thread pool.
      // It stays there until its response is
      // received.
     int expecting_response (Thread_Context *context);

     // Wait on handle set and demultiplex events
     // to their event handlers.
     int join (Thread_Context *context);

     // Handle the event by parsing its header
     // to determine the request id.
     virtual int handle_event (HANDLE h, Event_Type et);

     // Read the message in handle <h>, parse the header
     // of that message and return the response id
     virtual int parse_header (HANDLE h);

     // Read the body of the message, using the
     // information obtained in <parse_header>
     virtual int read_response_body (HANDLE h);

private:
     // Wrapper facade for the the multiplexed
     // connection stream.
     Reactor *reactor_;

     // The thread id of the leader thread.
     // Set to NO_CURRENT_LEADER if there
```

```
        // is no current leader.
        Thread_Id leader_thread_;

        // The pool of follower threads indexed by
        // the response id.
        typedef std::map<int, Thread_Context *>
            Follower_Threads;
        Follower_Threads follower_threads;

        // Serialize access to our internal state.
        Thread_Mutex mutex_;
    };
```

The following code fragment illustrates how a client thread that wants to send a request to a server uses the `expecting_response()` method to register its `Thread_Context` with the bound thread set, which informs the set that it expects a response:

```
    {
        // ...
        Bound_LF_Thread_Pool *tp = // ...
        // Create a new context, with a new unique
        // request id.
        Thread_Context *context = new Thread_Context;
        tp->expecting_response (context);
        send_request (context->request_id (),
                      /* request args */);
        tp->join (context);
    }
```

This registration must be performed *before* the thread sends the request. Otherwise, the response could arrive before the bound thread pool is informed which threads are waiting for it.

After the request is sent, the client thread invokes the `join()` method defined in the `Bound_LF_Thread_Pool` class to wait for the response. This method performs the following three steps:

- Step (a) — wait as a follower or become a leader.
- Step (b) — dispatch event to bound thread.
- Step (c) — promote new leader end.

The definition of steps (a), (b) and (c) in the `join()` method of `Bound_LF_Thread_Pool` are illustrated in the updated *Variant implementation activities* 4, 5, and 6, respectively, which are shown below.

It is instructive to compare the data members in the `Bound_LF_Thread_Pool` class shown above with those in the `LF_Thread_Pool` defined in the original

implementation activity 3. The primary differences are that the pool of threads in the `LF_Thread_Pool` is *implicit*, namely, the queue of waiting threads blocked on its condition variable synchronizer. In contrast, the `Bound_LF_Thread_Pool` contains an *explicit* pool of threads, represented by the `Thread_Context` objects, and a multiplexed `Reactor` object. Thus, each follower thread can wait on a separate condition variable until they are promoted to become the leader thread or receive an event hand-off from the current leader. o

4 *Implement a protocol to allow threads to initially join (and rejoin) the thread pool*. For bound thread/handle associations, the follower must first add its condition variable to the map in the thread pool and then call `wait()` on it. This allows the leader to use the Specific Notification pattern [Lea99a] to hand-off an event to a specific follower thread.

Â For example, our front-end communication servers must maintain a bound pool of follower threads, which is managed as follows:

```
int Bound_LF_Thread_Pool::join (Thread_Context *context) {
     // Step (a): wait as a follower or become a leader.
     // Use Scoped Locking idiom to acquire mutex
     // automatically in the constructor.
     Guard<Thread_Mutex> guard (mutex_);
     while (leader_thread_ != NO_CURRENT_LEADER
            && !context->response_received ()) {
          // There is a leader, wait as a follower...

          // Insert the context into the thread pool.
          int id = context->response_id ();
          follower_threads_[id] = context;
          // Go to sleep and release <mutex> atomically.
          context->condition ()->wait ();
          // The response has been received, so return.
          if (context->response_received ())
               return 0;
     }
     // There is no current leader, so become the leader.
     for (leader_thread = Thread::self ();
          !context->response_received (); ) {
          // Leave monitor temporarily to allow other
          // follower threads to join the set.
          guard.release ();

          // Demultiplex and dispatch an event.
          if (reactor_->handle_events () == -1)
               return -1;
          // Reenter monitor.
          guard.acquire ();
          // ... more below ...
```

After the thread is promoted to the leader role, the thread must perform all its I/O operations, waiting until its own event is received. In this case the `Event_Handler` forwards the I/O event to the thread pool:

```
class Bound_LF_Event_Handler : public Event_Handler {
private:
     // Instance of a <Bound_LF_Thread_Pool>.
     Bound_LF_Thread_Pool *thread_pool_;
public:
     Bound_LF_Event_Handler (Bound_LF_Thread_Pool *tp)
          :  thread_pool_ (tp) {}

     int handle_event (HANDLE h, Event_Type et) {
          thread_pool_->handle_event (h, et);   }
};
```

Next, the thread pool can handle the event by parsing its header to determine the request id and processing the event as before:

```
int Bound_LF_Event_Handler::handle_event (HANDLE h,
                                          Event_Type) {
     // Step (b): dispatch event to bound thread.
     // Parse the response header and get the response id.
     int response_id = parse_header (h);
     // Find the correct thread.
     Follower::iterator i =
          follower_threads_.find (response_id);

     // We are only interested in the value of
     // the <key, value> pair of the STL map.
     Thread_Context *dest_context = (*i).second;
     follower_threads_.erase (i);

     // Leave monitor temporarily to allow other
     // follower threads to join the set.
     guard.release ();
     // Read response into an application buffer
     dest_context->read_response_body (h);
     // Reenter monitor.
     guard.acquire ();

     // Notify the condition variable to
     // wake up the waiting thread.
     dest_context->response_received (true);
     dest_context->condition ()->notify (); }
```

Application developers are responsible for implementing the `parse_header()` and `read_response_body()` methods, which apply the Template Method pattern [GoF95]. o

5 *Implement the follower promotion protocol*. The following two protocols may be useful for bound handle/thread associations:

- *FIFO order*. A straightforward protocol is to promote the follower threads in *first-in, first-out* (FIFO) order. This protocol can be implemented using a native operating system synchronization object, such as a semaphore, if it queues waiting threads in FIFO order. The benefits of the FIFO protocol for bound thread/handle associations are most apparent when the order of client requests matches the order of server responses. In this case, no unnecessary event hand-offs need be performed because the response will be handled by the leader, thereby minimizing context switching and synchronization overhead.

  One drawback with the FIFO promotion protocol, however, is that the thread that is promoted next is the thread that has been waiting the *longest*, thereby minimizing CPU cache affinity [SKT96]. Thus, it is likely that state information, such as translation lookaside buffers, register windows, instructions, and data, residing within the CPU cache for this thread will have been flushed.

- *Specific order*. This ordering is common when implementing a bound thread pool, where it is necessary to hand-off events to a particular thread. In this case, the protocol implementation is more complex because it must maintain a collection of synchronizers.

  ÂFor example, this protocol can be implemented as part of the `Bound_LF_Thread_Pool`'s `join()` method to promote a new leader, as follows:

```
int Bound_LF_Thread_Pool::join (Thread_Context *context)
{
     // ... details omitted ...
     // Step (c): Promote a new leader.
     Follower_Threads::iterator i =
          follower_threads_.begin ();
     if (i == follower_threads_.end ())
          return 0; // No followers, just return.
     Thread_Context *new_leader_context = (*i).second;
     leader_thread_ = NO_CURRENT_LEADER;
     // Remove this follower...
     follower_threads_.erase (i);
     // ... and wake it up as newly promoted leader.
     new_leader_context->condition ()->notify ();
}                                                              o
```

7 *Implement the event hand-off mechanism*. Unbound handle/thread associations do not require event hand-offs between leader and follower threads. For bound handle/thread associations, however, the leader thread must be prepared to hand-off an event to a designated follower thread. The Specific Notification pattern [Lea99a] can be used to implement this hand-off scheme. Each follower thread has its own synchronizer, such as a semaphore or condition variable, and a set of these synchronizers is maintained by the thread pool. When an event occurs, the leader thread can locate and use the appropriate synchronizer to notify a specific follower thread.

Â In our OLTP example, front-end communication servers can use the following protocol to hand-off an event to the thread designated to process the event:

```
int Bound_LF_Thread_Pool::join (Thread_Context *context) {
    // ... Follower code omitted ...
    // Step (b): dispatch event to bound thread.
    for (leader_thread_ = Thread::self ();
         !context->response_received (); ) {
         // ... Leader code omitted ...

         // Parse response header and get the response id.
         int response_id = parse_header (buffer);
         // Find the correct thread.
         Follower::iterator i =
             follower_threads_.find (response_id);
         // We are only interested in the value of
         // the <key, value> pair of the STL map.
         Thread_Context *dest_context = (*i).second;
         follower_threads_.erase (i);

         // Leave monitor temporarily to allow other
         // follower threads to join the set.
         guard.release ();
         // Read response into pre-allocated buffers.
         dest_context->read_response_body (handle);
         // Reenter monitor.
         guard.acquire ();

         // Notify the condition variable to
         // wake up the waiting thread.
         dest_context->response_received (true);
         dest_context->condition ()->notify ();
    }
}
```

Example Resolved

Our OLTP front-end communication servers can use the bound handle/thread association version of the Leader/Follower pattern to wait for both requests from remote clients and responses from back-end servers. The `main()` function implementation can be structured much like the back-end servers described in the original *Example Resolved* section. The main difference is that the front-end server threads use the `Bound_LF_Thread_Pool` class rather than the `LF_Thread_Pool` class to bind threads to particular socket handles once they forward a request to a back-end server. Hence, each thread can wait on a condition variable until its response is received. After the response is received, the front-end server uses the request id to hand-off the response by locating the correct condition variable and notifying the designated waiting thread. This thread then wakes up and processes the response.

Using the Leader/Followers pattern is more scalable than simply blocking in a `read()` on the socket handle because the same socket handle can be shared between multiple front-end threads. This connection multiplexing conserves limited socket handle resources in the server. Moreover, if all threads are waiting for responses, the server will not dead-lock because it can use one of the waiting threads to process new incoming requests from remote clients. Avoiding deadlock is particularly important in multi-tier systems where servers callback to clients to obtain additional information, such as security certificates.

Relaxing Serialization Constraints. There are operating system platforms where multiple leader threads can wait on a handle set simultaneously. For example, the `WaitForMultipleObjects()` function on Win32 [Sol98] supports concurrent handle sets that allow a pool of threads to wait on the same set of handles concurrently. Thus, a thread pool designed using this function can take advantage of multi-processor hardware to handle other event concurrently while other threads wait for events.

On operating systems that support concurrent handle sets, the Leader/Followers pattern described in the *Implementation* section can overly restrict application concurrency because it serializes thread access to handle sets. To relax this serialization constrain, the following variations of the Leader/Followers pattern can be applied to allow multiple leader threads to be active simultaneously:

- *Leader/followers per multiple handle sets*. This variation applies the conventional Leader/Followers implementation to multiple handle sets separately. For instance, each thread is assigned a designated handle set. This variation is particularly useful in applications where multiple handle sets are available. However, this variant limits a thread to use a specific handle set.

- *Multiple leaders and multiple followers*. In this variation, the pattern is extended to support multiple simultaneous leader threads, where any of the leader threads can wait on any handle set. When a thread re-joins the read pool it checks if a leader is associated with every handle set already. If there is a handle set without a leader, the re-joining thread can become the leader of that handle set immediately.

Hybrid Thread Associations. Some applications use hybrid designs that implement both bound and unbound handle/thread associations simultaneously. Likewise, some handles in an application may have dedicated threads to handle certain events, whereas other handles can be processed by any thread. Thus, one variant of the Leader/Follower pattern uses its event hand-off mechanism to notify certain subsets of threads, according to the handle on which event activity occurs.

Â For example, the OLTP front-end communication server may have multiple threads using the Leader/Followers pattern to wait for new request events from clients. Likewise, it may also have threads waiting for responses to requests they invoked on back-end servers. In fact, threads play both roles over their lifetime, starting as threads that dispatch new incoming requests, then issuing requests to the back-end servers to satisfy the client application requirements, and finally waiting for responses to arrive from the back-end server. o

Hybrid Client/Servers. In complex systems, where peer applications play both client and server roles, it is important that the communication infrastructure process incoming requests while waiting for one or more replies. Otherwise, the system can dead-lock because one client dedicates all its threads to block waiting for responses.

In this variant, the binding of threads and handles changes dynamically. For example, a thread may be unbound initially, yet while processing an incoming request the application discovers it requires a service provided by another peer in the distributed system. In this case, the unbound thread dispatches a new request while executing application code, effectively binding itself to the handle used to send the request. Later, when the response arrives and the thread completes the original request, it becomes unbound again.

Alternative Event Sources and Sinks. Consider a system where events are obtained not only through handles but also from other sources, such as shared memory or message queues. For example, in UNIX there are no event demultiplexing functions that can wait for I/O events, semaphore events, and/or message queue events simultaneously. However, a thread can either block waiting for one type of event at the same time. Thus, the Leader/Followers pattern can be extended to wait for more than one type of events simultaneously, as follows:

- A leader thread is assigned to each source of events—as opposed to a single leader thread for the complete system.
- After the event is received, but before processing the event, a leader thread can select any follower thread to wait on this event source.

A drawback with this variant, however, is that the number of participating threads must always be greater than the number of event sources. Therefore, this approach may not scale well as the number of event sources grows.

Known Uses

ACE Thread Pool Reactor framework [Sch97a]. The ACE framework provides an object-oriented framework implementation of the Leader/Followers pattern called the 'thread pool reactor' (`ACE_TP_Reactor`) that demultiplexes events to event handlers within a pool of threads. When using a thread pool reactor, an application pre-spawns a *fixed* number of threads. When these threads invoke the `ACE_TP_Reactor`'s `handle_events()` method, one thread will become the leader and wait for an event. Threads are considered unbound by the ACE thread pool reactor framework. Thus, after the leader thread detects the event, it promotes an arbitrary thread to become the next leader and then demultiplexes the event to its associated event handler.

CORBA ORBs and Web servers. Many CORBA implementations, including Chorus COOL ORB [SMFG00] and TAO [SC99] use the Leaders/Followers pattern for both their client-side connection model and the server-side concurrency model. In addition, The JAWS Web server [HPS99] uses the Leader/Followers thread pool model for operating system platforms that do not allow multiple threads to simultaneously call `accept()` on a passive-mode socket handle.

Transaction monitors. Popular transaction monitors, such as Tuxedo, operate traditionally on a per-process basis, for example, transactions are always associated with a process. Contemporary OLTP systems demand high-performance and scalability, however, and performing transactions on a per-process basis may fail to meet these requirements. Therefore, next-generation transaction services, such as implementations of the CORBA Transaction Service [OMG97b], employ bound Leader/Followers associations between threads and transactions.

Taxi stands. The Leader/Followers pattern is used in everyday life to organize many airport taxi stands. In this use case, taxi cabs play the role of the 'threads,' with the first taxi cab in line being the *leader* and the remaining taxi cabs being the *followers*. Likewise, passengers arriving at the taxi stand constitute the events that must be demultiplexed to the cabs. In general, if any taxi cab can service any passenger, this scenario is equivalent to the *unbound* handle/thread association described in the main *Implementation* section. However, if only certain cabs can service certain passengers,

this scenario is equivalent to the *bound* handle/thread association described in the *Variants* section.

Consequences The Leader/Followers pattern provides the following **benefits**:

Performance enhancements. Compared with the Half-Sync/Half-Reactive thread pool approach described in the *Example* section, the Leader/Followers pattern can improve performance as follows:

- It enhances CPU cache affinity and eliminates the need for dynamic allocation and data buffer sharing between threads. For example, a processing thread can read the request into buffer space allocated on its run-time stack or by using the Thread-Specific Storage pattern [POSA2] to allocate memory.
- It minimizes locking overhead by not exchanging data between threads, thereby reducing thread synchronization. In bound handle/thread associations, the leader thread demultiplexes the event to its event handler based on the value of the handle. The request event is then read from the handle by the follower thread processing the event. In unbound associations, the leader thread itself reads the request event from the handle and processes it.
- It can minimize priority inversion because no extra queueing is introduced in the server. When combined with real-time I/O subsystems [KSL99], the Leader/Followers thread pool model can reduce sources of non-determinism in server request processing significantly.
- It does not require a context switch to handle each event, reducing the event dispatching latency. Note that promoting a follower thread to fulfill the leader role *does* require a context switch. If two events arrive simultaneously this increases the dispatching latency for the second event, but the performance is no worse than Half-Sync/Half-Reactive thread pool implementations.

Programming simplicity. The Leader/Follower pattern simplifies the programming of concurrency models where multiple threads can receive requests, process responses, and demultiplex connections using a shared handle set.

However, the Leader/Followers pattern has the following **liabilities**:

Implementation complexity. The advanced variants of the Leader/Followers pattern are harder to implement than Half-Sync/Half-Reactive thread pools. In particular, when used as a multi-threaded connection multiplexer, the Leader/Followers pattern must maintain a pool of follower threads waiting to process requests. This set must be updated when a follower thread is promoted to a leader and when a thread rejoins the pool of follower threads. All these operations can happen concurrently, in an

unpredictable order. Thus, the Leader/Follower pattern implementation must be efficient, while ensuring operation atomicity.

Lack of flexibility. Thread pool models based on the Half-Sync/Half-Reactive variant of the Half-Sync/Half-Async pattern [POSA2] allow events in the queueing layer to be discarded or re-prioritized. Similarly, the system can maintain multiple separate queues serviced by threads at different priorities to reduce contention and priority inversion between events at different priorities. In the Leader/Followers model, however, it is harder to discard or reorder events because there is no explicit queue. One way to provide this functionality is to offer different levels of service by using multiple Leader/Followers groups in the application, each one serviced by threads at different priorities.

Network I/O bottlenecks. The Leader/Followers pattern, as described in the *Implementation* section, serializes processing by allowing only a single thread at a time to wait on the handle set. In some environments, this design could become a bottleneck because only one thread at a time can demultiplex I/O events. In practice, however, this may not be a problem because most of the I/O-intensive processing is performed by the operating system kernel. Thus, application-level I/O operations can be performed rapidly.

See Also The Proactor pattern [POSA2] can be used as an alternative to the Leader/Followers pattern when an operating system supports asynchronous I/O efficiently and programmers are comfortable with the asynchronous inversion of control associated with this pattern. The Half-Sync/Half-Async [POSA2] and Active Object [POSA] patterns are alternatives to the Leader/Followers pattern when there are additional synchronization or ordering constraints that must be addressed before requests can be processed by threads in the pool. Moreover, these patterns may be necessary if event sources cannot be waited for by a single event demultiplexer.

Acknowledgement Thanks to Hans Rohert for many excellent comments on this paper.

References

[BaLee98] R.E. Barkley, T.P.Lee: *A Heap-Based Callout Implementation to Meet Real-Time Needs*, Proceedings of the USENIX Summer Conference, USENIX Association, pp. 213 - 222, June 1998

[BaMo98] G. Banga, J.C. Mogul: *Scalable Kernel Performance for Internet Servers Under Realistic Loads*, Proceedings of the USENIX 1998 Annual Technical Conference, USENIX, New Orleans, Louisiana, June 1998

[GoF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[HPS99] J.C. Hu, I. Pyarali, D.C. Schmidt: *The Object-Oriented Design and Performance of JAWS: A High-performance Web Server Optimized for High-speed Networks*, Parallel and Distributed Computing Practices Journal, special issue on Distributed Object-Oriented Systems, 1999

[HS98] J.C. Hu, D.C. Schmidt: JAWS: *A Framework for High-performance Web Servers*, Domain-Specific Application Frameworks: Frameworks Experience by Industry, Wiley & Sons, 1999. Eds: Mohamed Fayad and Ralph Johnson

[KSL99] F. Kuhns, D.C. Schmidt and D.L. Levine: *The Design and Performance of a Real-time I/O Subsystem*, Proceedings of the IEEE Real-Time Technology and Applications Symposium, IEEE, Vancouver, British Columbia, Canada, pp. 154 - 163, June 1999

[Lea99a] D. Lea: *Concurrent Programming in Java, Design Principles and Patterns*, 2nd edition, Addison-Wesley, 1999.

[Mes96] G. Meszaros: *A Pattern Language for Improving the Capacity of Reactive Systems*, in [PLoPD2], 1996

[OMG97b] Object Management Group: *CORBA Services - Transactions Service*, TC Document formal/ 97-12-17, 1997

[PLoPD3] R.C. Martin, D. Riehle, F. Buschmann (Eds.): *Pattern Languages of Program Design 3*, Addison-Wesley, 1997 (a book publishing selected papers from the Third International Conference on Pattern Languages of Programming, Monticello, Illinois, USA, 1996, the First European Conference on Pattern Languages of Programming, Irsee, Bavaria, Germany, 1996, and the Telecommunication Pattern Workshop at OOPSLA ‘96, San Jose, California, USA, 1996)

[POSA2] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann: *Pattern-Oriented Software Architecture—Concurrent and Networked Objects*, John Wiley and Sons, 2000

[Pree95] W. Pree: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995

[PRS+99] I. Pyarali, C. O’Ryan, D.C. Schmidt, N. Wang, V. Kachroo, A. Gokhale: *Applying Optimization Patterns to the Design of Real-Time ORBs*, Proceedings of the 5th conference on Object-Oriented Technologies and Systems, San Diego, CA, USENIX, 1999

[Rago93] S. Rago: *UNIX System V Network Programming*, Addison-Wesley, 1993.

[[SC99] D.C. Schmidt, C. Cleeland: *Applying Patterns to Develop Extensible ORB Middleware*, IEEE Communications Magazine Special Issue on Design Patterns, April, 1999.

[Sch97a] D.C. Schmidt: *The ACE Framework*, http://www.cs.wustl.edu/~schmidt/ACE.html, 1997

[SKT96] J.D. Salehi, J.F. Kurose, D. Towsley: *The Effetiveness of Affinity-Based Scheduling in Multiprocessor Nertworks*, IEEE INFOCOM, IEEE Computer Society Press, March, 1996

[SMFG00] D.C. Schmidt, S. Mungee, S. Flores-Gaitan, A. Gokhale: *Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers*, Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet, Ed.: A. Stoyen, Kluwer, 2000

[Sol98] D.A. Solomon: Inside Windows NT, 2nd Edition, Microsoft Press, 1998

[Ste97] W.R. Stevens: *Unix Network Programming, Second Edition*, Prentice Hall, 1997