Leader/Followers

A Design Pattern for Efficient Multi-threaded
Event Demultiplexing and Dispatching

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
Electrical Engineering and Computer Science Dept.
Vanderbilt University, Nashville, TN, USA*

Michael Kircher and Frank Buschmann
{Michael.Kircher,Frank.Buschmann}@mchp.siemens.de
Siemens Corporate Technology
Munich, Germany

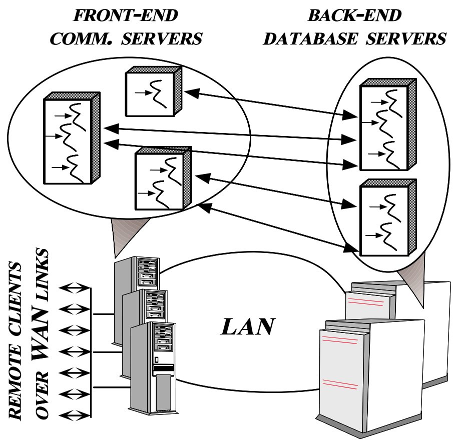
Irfan Pyarali and Carlos O'Ryan
{irfan,coryan}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA

1 Intent

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.

2 Example

Consider the design of a multi-tier, high-volume, on-line transaction processing (OLTP) system shown in the following figure. In this design, front-end communication servers route

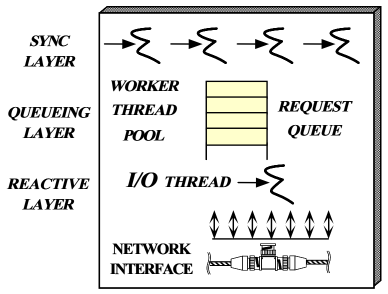


transaction requests from remote clients, such as travel agents, claims processing centers, or point-of-sales terminals, to back-end database servers that process the requests transactionally.

*This work was supported in part by Boeing, NSF grant NCR-9628218, DARPA contract 9701516, SAIC, Siemens, and Sprint.

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.

A common strategy for improving OLTP server performance is to use a multi-threaded concurrency model that processes requests and results simultaneously [1]. One way to multi-thread a OLTP server is to create a thread pool based on the "half-sync/half-reactive" variant of the Half-Sync/Half-Async pattern [2]. In this design, an OLTP server can be designed with a dedicated *network I/O* thread that uses the `select` [3] event demultiplexer to wait for events to occur on a set of socket handles, as shown in the following figure.



When activity occurs on handles in the set, `select` returns control to the network I/O thread and indicates which socket handle(s) in the set have events pending. This I/O thread then reads the transaction request from the designated socket handle, stores it into a dynamically allocated request, and inserts this request into a synchronized message queue implemented using the Monitor Object pattern [2]. This message queue is serviced by a pool of *worker threads*. When a worker thread in the pool is available, it removes the 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 multi-tier OLTP example. For instance, even with a light workload, the half-sync/half-reactive thread pool design 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 [4]. Moreover, if the OLTP server is run on a multi-processor, significant overhead can occur from processor cache coherency protocols required to transfer command objects between threads [5].

If the OLTP 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 [2]. This alternative will reduce some of the 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.

3 Context

An application where events occurring on set of handles must be demultiplexed and processed efficiently by multiple threads that share the handles.

4 Problem

Multi-threading is a common technique to implement applications that process multiple events concurrently. Implementing *high-performance* multi-threaded applications is hard, however. To address this problem effectively, the following *forces* must be addressed:

• **Efficient demultiplexing of handles and threads.** High-performance, multi-threaded applications concurrently process numerous types of events, such as CONNECT, READ, and WRITE events. These events often occur on handles, such as TCP/IP sockets [3], that are allocated for each connected client or server. A key design force, therefore, is determining efficient *demultiplexing associations* between threads and handles.

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

→ For our OLTP server applications, it is infeasible to associate a separate thread with each socket handle because this design will not scale efficiently as the number of connections increase. □

• **Minimize concurrency-related overhead.** 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 require dynamic memory allocations for each request passed between multiple threads will incur significant overhead on conventional multi-processor operating systems [6].

→ For instance, the "half-sync/half-reactive" thread pool variant [2] employed by our example OLTP servers requires memory to be allocated dynamically in the network I/O thread so that incoming transaction requests can be inserted into the message queue. This design incurs numerous synchronizations and context switches to insert/remove the request into/from the message queue. □

• **Prevent race conditions on handle sets.** Multiple threads that demultiplex events on a shared set of handles must coordinate to prevent *race conditions*. Race conditions can occur if multiple threads try to access or modify certain types of handles simultaneously. This problem often can be prevented by protecting the handles with a synchronizer, such as a mutex, semaphore, or condition variable.

→ For instance, a pool of threads cannot use `select` [3] 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 subset of socket handles [3]. 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. □

5 Solution

Structure a pool of threads to share a set of handles efficiently by taking turns demultiplexing events to event handlers that process the events.

In detail: Allow one thread at a time – the *leader* – to wait for an event to occur on a set of handles. 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 handle set, it first promotes a follower thread to become the new leader and then it plays the role of a *processing* thread, which demultiplexes the event to its designated event handler and perform application-specific event processing. Multiple processing threads can run concurrently while the current leader thread waits for new events on the handle set shared by the threads. After handling its event, a processing thread reverts to a follower that waits for its turn to become the leader thread again.

6 Structure

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

Handles and handle sets. *Handles* identify resources, such as socket connections or open files, which are often implemented and managed by an operating system. A *handle set* is a collection of handles that can be used to wait for 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.

→ 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. Each connection is represented in a server by a separate socket handle. Our OLTP servers use the `select` [3] event demultiplexer, which identifies handles that have events pending, so that applications can invoke I/O operations on these handles *without* blocking the calling threads. □

Event handler. An event handler specifies an interface consisting of one or more hook methods [7, 8]. These methods represent the set of operations available to process application- or service-specific events that occur on handle(s) associated with an event handler.

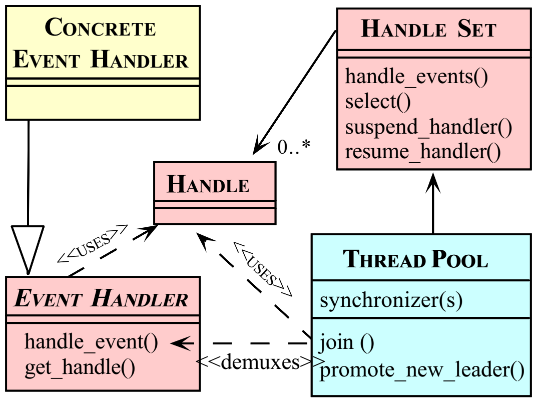
Concrete event handler. Concrete event handlers specialize from the event handler and implement a specific service that the application offers. Each concrete event handler is associated with a handle in the handle set that identifies this service within the application. In addition, concrete event handlers implement the hook method(s) responsible for processing events received through their associated handle.

→ For example, concrete event handlers in front-end OLTP servers receive and validate remote client requests and forward valid 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. □

Thread pool. At the heart of the the Leader/Followers pattern is a pool of threads, which take turns playing various roles. One or more *follower threads* queue up on a synchronizer, such as a semaphore or condition variable, waiting to become the leader thread. 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 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 the event from the handle set to its associated 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 synchronizer for its turn to become the leader thread again.

→ For example, each OLTP server designed using the Leader/Followers pattern can have a pool of threads waiting to process transaction requests. At any point in time, multiple threads in the pool can be processing transaction requests and sending results back to their clients. Up to one thread in the pool is the current *leader*, which waits on the handle set for new CONNECT and READ events to arrive. When this occurs, the leader thread will then play the role of a processing thread, which demultiplexes the event from its handle to its associated event handler and dispatches the handler's hook method to process the event. Any remaining threads in the pool are the *followers*, which wait on a synchronizer for their turn to be promoted to become the leader thread. □

The following figure illustrates the structure of participants in the Leader/Followers pattern.



7 Dynamics

The following collaborations occur in the Leader/Followers pattern.

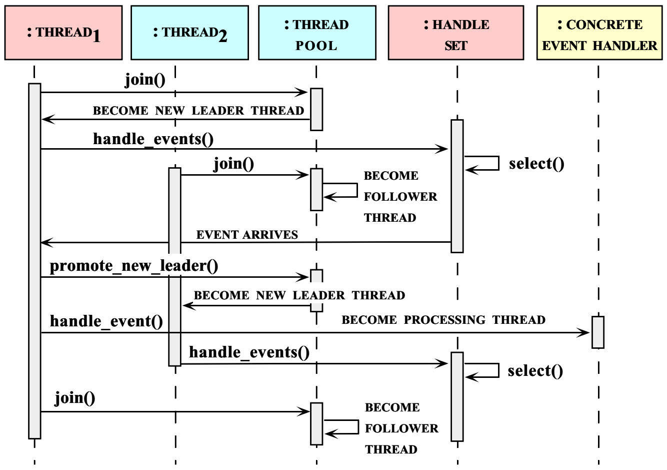
• **Leader thread demultiplexing.** The leader thread waits for an event to occur on any handle in the handle set. If there is no current leader thread, *e.g.*, due to events arriving faster than the available threads can service them, the underlying operating system can queue events internally until a leader thread is available.

• **Follower thread promotion.** After the leader thread has detected an new event, it chooses 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 promoting a follower to become the new leader, the former leader plays the role of a processing thread, which concurrently demultiplexes the event it detected to its associated event handler and then dispatches the handler's hook method to handle the event.

• **Rejoining the thread pool.** To demultiplex handles in a handle set, a processing thread must first rejoin the thread pool after it is complete and can 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 and waits until it is promoted by a leader thread.

The following figure illustrates the collaborations among participants in the Leader/Followers pattern for a pool of two threads.



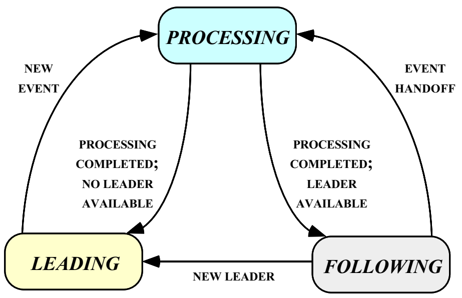
At any point in time, a thread participating in the Leader/Followers pattern is in one of following three states:

• **Leader.** A thread in this state is currently the leader, waiting for an event to occur on the handle set. A thread in the leader state can transition into the processing state when it detects a new event.

• **Processing.** A thread in this state can execute concurrently with the leader thread and any other threads that are in the processing state. A thread in the processing state typically transitions to the follower state, though it can transition to the leader state immediately if there is no current leader thread when it finishes its processing.

• **Follower.** A thread in this state waits as a follower in the thread pool. A follower thread can transition to the leader state when promoted by the current leader.

The figure below illustrates the states and the valid transitions in the Leader/Followers pattern.


PROCESSING
NEW EVENT
EVENT HANDOFF
PROCESSING COMPLETED; NO LEADER AVAILABLE
PROCESSING COMPLETED; LEADER AVAILABLE
LEADING
NEW LEADER
FOLLOWING

8 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 can be used to wait for events to occur. Developers often choose the handles and handle set mechanisms provided by an 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 of handle allows multiple threads to access the handle concurrently without incurring race conditions that can corrupt, lose, or scramble the data [3]. 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 of handle requires multiple threads to access the handle iteratively because concurrent access will cause race conditions. For instance, the Socket API for bytestream-oriented protocols, such as TCP, does not guarantee that `read` or `write` operations are atomic. 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 of handle set can be called concurrently, *e.g.*, 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 [9] supports concurrent handle sets by allowing a pool of threads to wait on the same set of handles simultaneously.

• **Iterative handle set.** This type of handle set 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` [3] and `poll` [10] functions only 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.

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`s 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 implementation activities can be skipped.

For instance, certain network programming APIs, such as UDP support in Sockets, support concurrent multiple I/O operations on a shared handle. Thus, 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 [9], 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 system's thread scheduler to (de)multiplex threads, handle sets, and handles robustly, in which case, *implementation activities* 2 through 6 can be skipped.

1.4. Implement an event handler demultiplexing mechanism. In addition to calling an event demultiplexer to wait for indication events to occur on its handle set, a Leader/Followers pattern implementation must demultiplex events to event handlers and dispatch their hook methods to process the events. The following are two strategies for implementing this mechanism:

• **Implement a demultiplexing table.** In this strategy, the handle set demultiplexing mechanisms provided by the operating system are used directly. Thus, a Leader/Follower implementation must maintain a demultiplexing table that is a manager [PLoPD3] containing a set of <*handle, event handler, event registration types*> tuples. Each handle is a "key" used to associate handles with event handlers in its demultiplexing table, which also stores the type of indication event(s), such as CONNECT and READ, that each event handler is registered for on its handle. The contents of this table are converted into handle sets passed to the native event demultiplexing mechanism, such as `select` or `WaitForMultipleObjects`,

→ *Implementation activity* 3.3 of the Reactor pattern [2] illustrates how to implement a demultiplexing table. □

• **Apply a higher-level event demultiplexing pattern.** In this strategy, developers leverage higher-level patterns, such as Reactor [2], Proactor [2], and Wrapper Facade [2]. 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.

→ 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 [2] 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. □

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. Temporarily deactivating the handle from the handle set 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 handle set during this interval, it could falsely dispatch the event a second time. After the event is processed, the handle is reactivated in the handle set, which allows the leader thread to wait for events to occur on it and any other activated handles in the set.

→ In our OLTP example, this handle (de)activation protocol can be provided by extending the `Reactor` interface defined

[2]However, `WaitForMultipleObjects` does not by itself address the problem of notifying a particular thread when an event is available.

in *implementation activity* 2 of the Reactor pattern [2], 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);
  // ...
};
```

□

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. All follower threads in the set can 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 multiple threads sharing a handle set are serialized.

→ 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_Bound_Thread` caches the reactor passed to it. By default, this reactor implementation uses `select`, which only supports iterative handle sets. Therefore, `LF_Thread_Pool` is responsible for serializing multiple threads that take turns calling `select` on the reactor's handle set.

Application threads invoke the `join` method 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 a timeout occurs. 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` and `reactive_handle` methods temporarily deactivate and activate handles within a reactor's handle set. The implementations of these methods simply forward to the same methods defined on the `Reactor` interface shown in *implementation activity* 2.

Note that a single condition variable synchronizer is shared by all threads in this set. As shown in *implementation activities* 4 and 5, the implementation of `LF_Thread_Pool` is designed using the Monitor Object pattern [2]. □

4. Implement a protocol to allow threads to initially join (and rejoin) the thread pool. This protocol is used when event processing completes and a thread is available to process another event. If no leader thread is available, a follower 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. The following two subactivities can be used to implement this protocol.

→ 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 I/O 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 it can be a leader, at which point it uses the reactor to wait for an I/O 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 reassumes its follower role. These steps continue looping until the application terminates or a timeout occurs.

□

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. 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 restored once the event has been dispatched.

→ The Decorator pattern [7] can be applied to allow the Leader/Followers pattern implementation to promote a new leader *before* invoking the `handle_event` hook, as follows:

```
class LF_Event_Handler : public Event_Handler
  // This use of <Event_Handler> plays the
  // <Component> role in the Decorator pattern.
{
  // 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, Event_Type et) {
    // Temporarily deactivate the handler in the
    // reactor to prevent race conditions.
    thread_pool_->deactivate_handle
      (this->get_handle (), et);

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

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

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

As shown above, an application can implement concrete event handlers and the Leader/Followers implementation can use the Decorator pattern to promote a new leader transparently. □

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 [9], which is an example of the "Fresh Work Before Stale" pattern [11]. 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 [12], 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 [13]. 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 [12].

- **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.

→ For example, back-end database servers could use the following simple protocol to promote follower thread in whatever order they are queued by an operating system's condition variable:

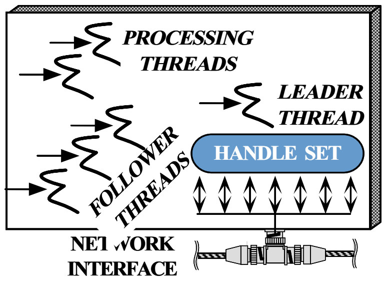
```
int LF_Thread_Pool::promote_new_leader (void)
{
  // Use Scoped Locking idiom to acquire mutex
  // automatically in the constructor.
  Guard<Thread_Mutex> guard (mutex_);

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

  // Indicate that we're 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 to process a request. □

6. Implement the concrete event handlers. Application developers must decide what actions to perform when the hook method of concrete event handler is invoked by a processing thread in the Leader/Followers pattern implementation. *Implementation activity* 6 in the Reactor pattern [2] describes various issues associated with implementing concrete event handlers.

9 Example Resolved

In this section, we illustrate how the Leader/Followers pattern can be applied to our OLTP back-end database servers.[3] Back-end database servers can use the Leader/Followers pattern to implement a thread pool that demultiplexes I/O events from socket handles to their event handlers efficiently. As illustrated in the following figure, there is no designated network I/O thread. Instead, a pool of threads is pre-allocated during



database server initialization, as shown in the following `main` function:

```
const int MAX_THREADS = ...;

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);

  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 the network I/O thread by calling the `LF_Thread_Pool::join` method, as follows:

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

  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 servers. If requests arrive when all threads are busy, they will be queued in socket handles until a thread in the pool is available to execute the requests.

When a request event arrives, the leader thread temporarily deactivates the socket handle 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 [2].[4] All OLTP activities occur in the processing thread. Thus, no further context switching, synchronization, or data movement is necessary. When it finishes handling a request, the processing thread returns to playing the role of a follower and waits in the thread pool. Moreover, the socket handle it was processing is reactivated in the handle set so that `select` can wait for I/O events to occur on it, along with other sockets in the handle set.

10 Variants

10.1 Bound Handle/Thread Associations

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 threads take turns demultiplexing a shared handle set.

→ 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. □

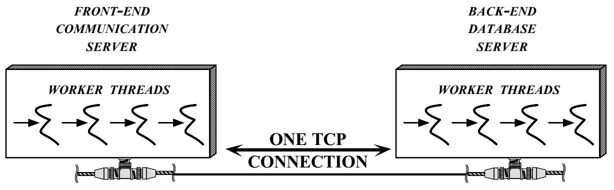
However, an important variant of this pattern uses *bound handle/thread associations*. In this use case, each thread is

[3] The *Variants* section describes how this pattern can be applied to our OLTP front-end communication servers.

[4] 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.

bound to its own handle, which it uses to process particular events. Bound associations are often used when a client application 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 a specific thread, *i.e.*, the thread that sent the original request.

→ 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 back-end servers using *multiplexed connections* [12], 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 [14]. □

As described below, supporting bound thread/handle associations requires changes to the following sections of the Leader/Followers pattern.

Structure

Thread pool. In the bound handle/thread association model, the leader thread 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 collection class. 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 model variant of the Leader/Followers pattern requires changes to the following two collaboration:

• **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 then unregisters itself from the thread pool and processes the incoming event concurrently. 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.

Implementation

The first two *implementation activities* in the earlier *Implementation* section require no changes. However, the following changes are required to support bound handle/thread associations in subsequent *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 results 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.

→ 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_;
};
```

A thread that wants to send a request uses the expecting_response method to register its associated Thread_Context with the bound thread set to inform the set that it expects a response:

```
void new_request (Bound_LF_Thread_Pool *tp
  ... /* request args */)
{
  // 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

The definition of steps (a), (b) and (c) in the join method of Bound_LF_Thread_Pool are illustrated in the updated *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 *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 SOCK_Stream wrapper facade 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. □

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 [15, 16] to hand-off an event to a specific follower thread.

→ For example, our front-end communication servers must maintain a bound pool of follower threads. This set is updated when a new leader is promoted, 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;
  }
  // No leader, 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 ();
    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);
  }
}
```

Unlike the unbound case we cannot apply the Decorator pattern to augment any user defined event handler to participate in the Leader/Follower pattern. The thread pool needs to parse the request to extract the response id and match it to the corresponding `Thread_Context`. Consequently the thread pool must perform at least part of the I/O, and it cannot be completely encapsulated by the `Event_Handler`.
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 handle,
                                      Event_Type)
{
  // Parse the response header and
  // get the response id.
  int response_id = parse_header (handle);

  // 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 *destination_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
  destination_context->read_response_body (handle);
  // Reenter monitor.
  guard.acquire ();

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

Application developers are responsible for implementing the `parse_header` and `read_response_body` methods, which apply the Template Method pattern [7].

6. 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 [5, 17]. 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 more complex because it must maintain a collection of synchronizers.

→ For 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 ();
}
```

□

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 [15, 16] 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.

→ 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 the 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 *destination_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.
    destination_context->read_response_body (handle);
    // Reenter monitor.
    guard.acquire ();

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

□

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 main *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.

10.2 Relaxing Serialization Constraints

There are operating system platforms where multiple leader threads can wait simultaneously on a handle set. For example, the Win32 `WaitForMultipleObjects` function [9] supports concurrent handle sets that allow a pool of threads to wait on the same set of handles simultaneously. Thus, a thread pool designed using this function can take advantage of multi-processor hardware to perform useful computations while other threads wait for events. In such cases, the conventional Leader/Followers pattern implementation serializes thread access to handle sets, which can overly restrict application concurrency. To relax this constrain, the following variants of the Leader/Followers pattern can allow multiple leader threads to be active simultaneously:

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

Multiple leaders and multiple followers. In this variant, 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 leaders/followers thread 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.

10.3 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.

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 to dispatch new incoming requests, issuing requests to the back-end servers to satisfy the client application requirements and then waiting for the responses from the back-end server.

10.4 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 has all its threads blocked waiting for responses.

In this variant, the binding of threads and handles changes dynamically, for example, initially a thread may be unbound, during processing of an incoming request the application requires services provided by other peers in the distributed system. In that case the unbound thread dispatches new requests 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.

In such an implementation, the `Bound_LF_Thread_Pool` cannot simply demultiplex events for a single handle. As with the `LF_Thread_Pool` class, the unbound version must be extended to support a full handle set. In particular the `wait()` method in Step 4 cannot perform the I/O directly. Instead the `Event_Handler` performs all the I/O, and it informs the `Bound_LF_Thread_Pool` to dispatch the message to the correct thread.

10.5 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:

1. A leader thread is assigned to each source of events (as opposed to a single leader thread for the complete system).
2. After the event is received, but before processing the event, a leader thread can select any follower thread to wait on the leader's event source.

A drawback with this variant, however, is that the number of participating thread must always be greater than the number of event sources. Therefore, it can be hard to scale it as the number of event sources increases.

11 Known Uses

ACE Thread Pool Reactor framework [18]. The ACE framework provides an object-oriented framework implementation of the Leader/Followers pattern called the "thread pool reactor" (`ACE_TP_Reactor`) to demultiplex 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 `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, once the leader leader thread detects the event, it promotes an arbitrary thread to become the next leader it and then demultiplexes the event to its associated event handler.

CORBA ORBs. Many CORBA implementations, including Chorus COOL ORB [12] and TAO [19] use the Leaders/Followers pattern for both their client-side connection model and the server-side concurrency model.

Web servers. The JAWS Web server [1] uses the Leader/Followers thread pool model...

Transaction monitors. Popular transaction monitors, such as Tuxedo, have traditionally operated on a per-process basis, *i.e.*, 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 the CORBA Transaction Service [14], employ bound associations between threads and transactions. The Leader/Followers pattern supports this architecture with bound associations between threads and handles.

Taxi stands. The Leader/Followers pattern is used in everyday life to organize many airport taxi stands. In this case, taxi cabs are 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 is equivalent to the *unbound* handle/thread association described in the main *Implementation* section. However, if only certain cabs can service certain passengers, this is is equivalent to the *bound* handle/thread association described in the variant's *Implementation* section.

12 See Also

The Proactor pattern [2] can be used as an alternative to the Leader/Followers pattern when an operating system supports asynchronous I/O efficiently.

The Half-Sync/Half-Async [2] and Active Object [2] 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.

13 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 unbound allocation and data buffer sharing between threads by reading the request into buffer space allocated on the stack of the leader or by using the Thread-Specific Storage pattern [2] 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 [20], the Leader/Followers thread pool model can significantly reduce sources of non-determinism in server request processing.

- 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 it 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 [2] 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 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 I/O-intensive processing is performed by the operating system kernel. Thus, the I/O operations can be performed rapidly.

Acknowledgements

Thanks to Hans Rohnert for providing insightful comments that helped to improve this paper significantly.

References

[1] J. Hu, I. Pyarali, and 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*, vol. 3, Mar. 2000.

[2] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2.* New York: Wiley & Sons, 2000.

[3] W. R. Stevens, *UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI, 2nd Edition*. Englewood Cliffs, NJ: Prentice Hall, 1998.

[4] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), pp. 145–159, USENIX, May 1999.

[5] J. D. Salehi, J. F. Kurose, and D. Towsley, "The Effectiveness of Affinity-Based Scheduling in Multiprocessor Networking," in *IEEE INFOCOM*, (San Francisco, USA), IEEE Computer Society Press, Mar. 1996.

[6] D. C. Schmidt and T. Suda, "Measuring the Performance of Parallel Message-based Process Architectures," in *Proceedings of the Conference on Computer Communications (INFOCOM)*, (Boston), pp. 624–633, IEEE, Apr. 1995.

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

[8] W. Pree, *Design Patterns for Object-oriented Software Development.* Reading, MA: Addison-Wesley, 1995.

[9] D. A. Solomon, *Inside Windows NT, 2nd Edition*. Redmond, WA: Microsoft Press, 1998.

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

[11] G. Meszaros, "A Pattern Language for Improving the Capacity of Reactive Systems," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, Massachusetts: Addison-Wesley, 1996.

[12] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and 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*, vol. 21, no. 2, 2001.

[13] R. E. Barkley and T. P. Lee, "A Heap-based Callout Implementation to Meet Real-time Needs," in *Proceedings of the USENIX Summer Conference*, pp. 213–222, USENIX Association, June 1988.

[14] Object Management Group, *Transaction Services Specification*, OMG Document formal/97-12-17 ed., Dec. 1997.

[15] T. Cargill, "Specific Notification for Java Thread Synchronization," in *Pattern Languages of Programming Conference (PLoP)*, Sept. 1996.

[16] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns, Second Edition.* Boston: Addison-Wesley, 2000.

[17] J. C. Mogul and A. Borg, "The Effects of Context Switches on Cache Performance," in *Proceedings of the* 4^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Santa Clara, CA), ACM, Apr. 1991.

[18] D. C. Schmidt, "Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software," in *Handbook of Programming Languages* (P. Salus, ed.), MacMillan Computer Publishing, 1997.

[19] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the* 3^{rd} *Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

[20] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of RIO – A Real-time I/O Subsystem for ORB Endsystems," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, (Edinburgh, Scotland), OMG, Sept. 1999.