Leader/Followers

A Design Pattern for Efficient Multi-threaded
I/O Demultiplexing and Dispatching

Douglas C. Schmidt and Carlos O'Ryan
{schmidt,coryan}@uci.edu
Electrical and Computer Engineering Dept.
University of California, Irvine, CA 92697, USA[*]

Michael Kircher
Michael.Kircher@mchp.siemens.de
Siemens Corporate Technology
Munich, Germany

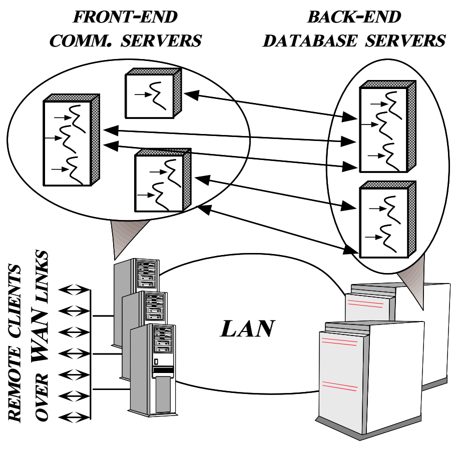
Irfan Pyarali
irfan@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA

1 Intent

The Leader/Followers design pattern provides a concurrency model where multiple threads can efficiently demultiplex events and dispatch event handlers that process I/O handles shared by the threads.

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. This multi-tier architecture is used to improve overall system throughput and reliability via load balancing and redundancy, respectively.

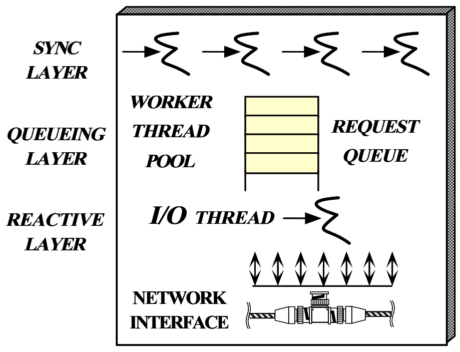
[*]This work was supported in part by Boeing, NSF grant NCR-9628218, DARPA contract 9701516, SAIC, Siemens, and Sprint.

The front-end communication servers are actually "hybrid" client/server applications that perform two primary tasks. First, they receive requests arriving simultaneously from hundreds or thousands of remote clients over wide area communication links, such as X.25 or the TCP/IP. Second, 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 the designated transactions. 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.

The servers in this OLTP system spend most of their time processing various types of I/O operations in response to requests. For instance, front-end servers perform network I/O to receive transaction requests from remote clients, forward them to the appropriate database server, wait for the result, and finally forward the result back to the client. Likewise, back-end 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.

A common strategy for improving OLTP server performance is to use a multi-threaded concurrency model that processes requests and results simultaneously [1]. 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 a OLTP back-end database server is to create a thread pool based on the "half-sync/half-reactive" variant of the Half-Sync/Half-Async pattern [2]. In large-scale OLTP systems, the number of I/O handles may be much larger than the number of threads. In this case, an event demultiplexer, such as `select` [3], `poll` [4], or `WaitForMultipleObjects` [5], can be used to wait for events to occur on a socket handle set. 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 OLTP servers can be designed with a dedicated *network I/O* thread assigned to the event demultiplexer, as shown in the following figure. When activity occurs on handles in the set, the event



demultiplexer returns control to the network I/O thread and indicates which socket handle(s) in the set have events pending. This thread then reads the transaction request from the designated socket handle, stores it into a dynamically allocated command object [6], and inserts the command object into a message queue implemented using the Monitor Object pattern [7]. This message queue is serviced by a pool of *worker threads*. When a worker thread in the pool is available, it removes the command object from the queue, performs the designated transaction, and then returns a response to the front-end communication server.

A similar concurrency design can be applied to front-end communication servers, where a separate network I/O thread and a pool of worker threads can validate and forward client requests to the appropriate back-end servers. In this design, the front-end communication servers also play the role of "clients" to back-end servers. Thus, they wait for the back-end servers to return transaction results. After a front-end server receives a result from a back-end server, the result must be dispatched to the appropriate worker thread. Moreover, in multi-tier systems, front-end servers may need to respond to requests generated by back-ends while they are processing front-end requests. Therefore, front-end servers must always be able to process incoming requests and send responses, which implies that worker threads in front-end servers cannot all block simultaneously.

Although the threading models described above are used in many concurrent applications, they 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 command object between the network I/O thread and a worker thread, which makes even the best-case latency unnecessarily high [8]. 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 [9].

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 [10]. 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 multiple threads.

3 Context

An application where events occurring on set of I/O handles must be demultiplexed and dispatched efficiently by multiple threads.

4 Problem

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

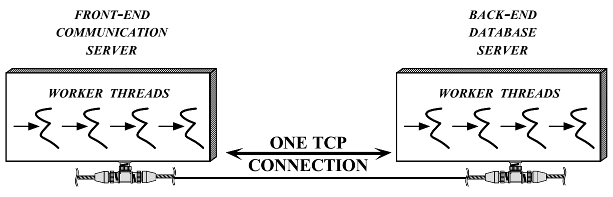
• **Efficient demultiplexing of I/O handles and threads:** High-performance multi-threaded applications process numerous types of events, such as connection, read, and write events, concurrently. These events often occur on I/O handles, such as TCP/IP sockets [3], that are allocated for each connected client or server. A key design challenge, therefore, is determining efficient *(de)multiplexing associations* between threads and I/O handles.

For server applications, it is often infeasible to associate a separate thread with each I/O handle because this design may not scale efficiently as the number of handles increases. It may be necessary, therefore, to have a small, fixed number of

[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.

threads *demultiplex* events from a larger number of handles. Conversely, a client application may have a large number of threads that are communicating with the same server. In this case, however, allocating a connection-per-thread may consume excessive operating system resources. Thus, it may be necessary to *multiplex* events generated by many client threads onto a smaller number of connections, *e.g.*, by maintaining a single connection from a client process to each server process [11] with which it communicates.

→ For example, one possible OLTP server concurrency model could allocate a separate thread for each client connection. However, this thread-per-connection concurrency model may not handle hundreds or thousands of simultaneous connections scalably. Therefore, our OLTP servers employ a demultiplexing model that uses a thread pool to align the number of server threads to the available processing resources, such as the number of CPUs, rather than to the number of active connections. Likewise, to conserve system resources, multiple threads in each of our front-end communication servers send requests to the same back-end server over a single *multiplexed connection*, as shown in the following figure. Thus, when a



front-end server receives a result from a back-end server, it must demultiplex the result to the corresponding thread that is blocked waiting to process it. □

• **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, a concurrency model that requires memory to be allocated dynamically for each request and passed between multiple threads will incur significant overhead on conventional multiprocessor operating systems [12].

→ For instance, our example OLTP servers employ a thread pool concurrency model based on the "half-sync/half-reactive" variant of the Half-Sync/Half-Async pattern [2]. This model uses a message queue to decouple the *network I/O thread*, which receives client request events, from the pool of *worker threads*, which process these events and return responses to clients. Unfortunately, this design requires memory to be allocated dynamically, from either the heap or a global pool, in the network I/O thread so that incoming event requests can be inserted into the message queue. In addition, it requires numerous synchronizations and context switches to insert/remove the request into/from the message queue. □

• **Prevent race conditions:** Multiple threads that demultiplex events on a set of I/O handles must coordinate to prevent *race conditions*. Race conditions can occur if multiple threads try to access or modify certain types of I/O 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 handles [3]. Thus, the thread pool would need to re-synchronize to avoid having multiple threads `read` from the same handle. Moreover, for bytestream-oriented protocols, such as TCP, having multiple threads invoking `read` on the same socket handle will corrupt or lose data. Likewise, multiple simultaneous `write`s to a socket handle can "scramble" the data in the bytestream. □

5 Solution

Allow one thread at a time – the leader – to wait for an event to occur on a set of I/O handles. Meanwhile, other threads – the followers – can queue up waiting their turn to become the leader. After the current leader thread demultiplexes an event from the I/O handle set, it promotes a follower thread to become the new leader and then dispatches the event to a designated event handler, which processes the event. At this point, the former leader and the new leader thread can execute concurrently.

In detail: multiple former leader threads can process events concurrently while the current leader thread waits on the handle set. After its event processing completes, an idle follower thread waits its turn to become the leader. If requests arrive faster than the available threads can service them, the underlying I/O system can queue events internally until a leader thread becomes available. The leader thread may need to handoff an event to a follower thread if the leader does not have the necessary context to process the event. This scenario is particularly relevant in high-volume, multi-tier distributed systems, where results often arrive in a different order than requests were initiated. For example, if threads use the Thread-Specific Storage pattern [22] to reduce lock contention, the thread that processes a result must be the same one that invoked the request.

6 Structure

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

Handles and handle sets: *Handles* identify I/O 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 I/O 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 – CONNECTION 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 handles *without* blocking calling threads. However, multiple threads cannot call `select` on the same handle set simultaneously because more than one thread will be notified erroneously that I/O events are pending. □

Event handler: An event handler specifies an interface consisting of one or more hook methods [6, 13]. 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.

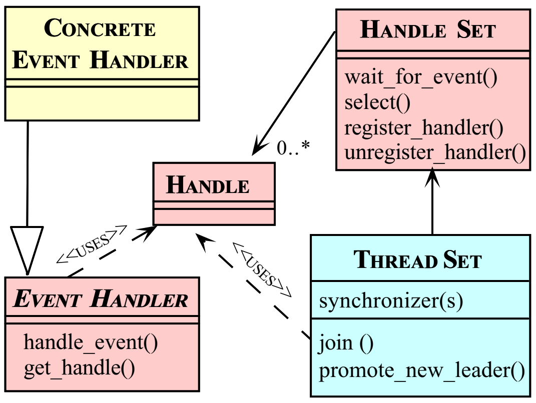
Concrete event handler: Concrete event handlers specialize from the event handler and implement a specific service that the application offers. Each concrete event handler is associated with a handle that identifies this service within the application. In addition, concrete event handlers implement the 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. □

Leader/followers thread set: Threads play several roles in the Leader/Followers pattern. A leader thread waits for an event to occur on a handle set, as which point it processes the event and performs some type of service. After a thread is finished processing an event it can wait on a *thread set*. Zero or more follower threads can queue up waiting to become the leader or to receive event dispatches from the current leader thread. When the current leader receives an event from its handle set it promotes a *follower* thread to become the new leader and then processes the event itself. The follower set can be maintained implicitly, for example, using 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 handoffs.

→ For example, each OLTP back-end database server has 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 front-end communication servers. 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. Any remaining threads in the pool are the *followers*, which wait on the *thread set* for their turn to be promoted to become the leader thread or to receive event handoffs from the current leader. □

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



7 Dynamics

Two types of collaborations can occur between participants in the Leader/Followers pattern, depending on whether there is a *bound* or *unbound* association between I/O handles in a handle set and threads, as described below.

Unbound handle/thread association: In this use case, there is no fixed association between threads and I/O handles. Thus, any thread can process any event that occurs on any I/O handle in a handle set. Unbound associations are often used when a pool of threads take turns sharing a handle set.

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

Bound handle/thread association: In this use case, each thread is bound to its own I/O 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.[2] In this case, the client application thread expects to process the response event on this I/O 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* [11]. 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]. □

Given the two types of associations described above, the following collaborations occur in the Leader/Followers pattern.

1: Leader demultiplexing: The leader thread waits for an event to occur on the handle set.

2: Follower promotion: There are two cases, depending on which type of thread/handle associations are used:

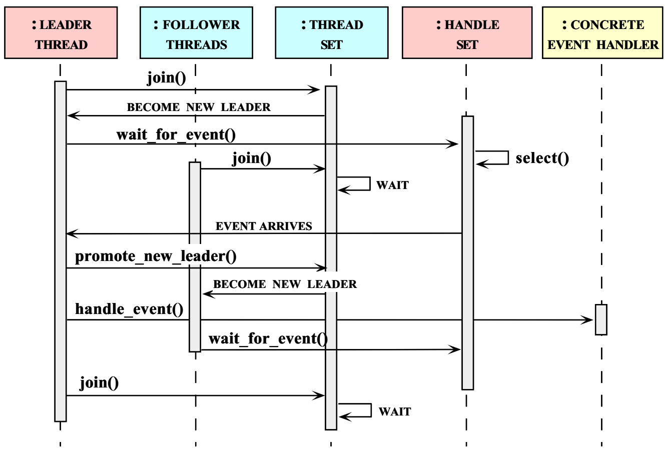
• For *unbound* thread/handle associations, after the leader thread has demultiplexed an event, it chooses a follower thread to become the new leader using one of the promotion protocols described in *implementation* activity 4.

• For *bound* thread/handle associations, after the leader demultiplexes one event, it checks the I/O 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 used for unbound thread/handle associations. Conversely, if the event is intended for another thread, the leader must handoff the event to the designated follower thread. This follower thread then unregisters itself from the leader/followers thread set and processes the incoming event concurrently. Meanwhile, the current leader thread continues to wait for another event to occur on the handle set.

3: Event processing: For unbound handle/thread associations, the former leader thread concurrently processes the event it demultiplexed after promoting a follower to become the new leader. For bound handle/thread associations, either the former leader continues to process the event it demultiplexed or a follower thread processes the event that the leader thread handed off to it.

4: Rejoining the leader/followers thread set: To demultiplex the I/O handles in a handle set, a thread must first (re)join the leader/followers thread set. This action often occurs when event processing is completed and a thread is available to process another event. A thread can become the leader immediately if there is no current leader. Otherwise, the thread becomes a follower and must wait until it is promoted by a leader.

The following figure illustrates the collaborations among participants in the Leader/Followers pattern.



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

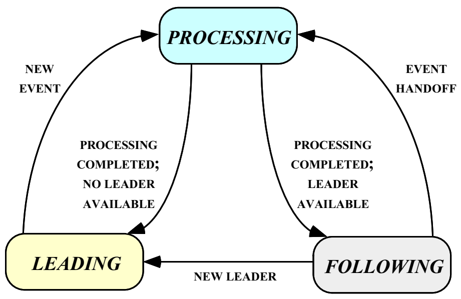
• **Leader:** A thread in this state is currently the leader, waiting for an event to occur on the handle set. A thread in the leader state can transition into the processing state when it receives an event.

• **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 in the follow thread set. A follower thread can transition to the leader state when promoted by the current leader or it can move directly to the processing state if it receives an event handoff from the leader.

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

[2]Note that multiple threads can be bound to the same I/O handle if connections are multiplexed.


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 I/O handle and handle set mechanisms: A handle set is a collection of I/O handles that can be used to wait for events to occur on handles in the set. Developers often choose the I/O 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 I/O handle and handle set mechanisms.

1.1. Determine the type of I/O handle: There are two general types of I/O handles:

- **Concurrent handles:** This type of handle allows multiple threads to access the handle concurrently without incurring race conditions that can corrupt, loose, 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, if I/O operations on the socket are not serialized properly, corrupted or lost data can result.

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 [5] 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` [4] functions only support iterative handle sets. Thus, a pool of threads cannot use `select` or `poll` to demultiplex events on the same handle set.

1.3. Determine the consequences of selecting certain I/O 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 applications need not guard against these use cases. In particular, if the I/O handle and handle set mechanisms are both concurrent, many of the remaining 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 by one thread or another, without risk of partial `read`s or interleaved data corruption. Likewise, certain handle set mechanisms, such as the Win32 `WaitForMultipleObjects` function [5], return a single handle per call, which allows them to be called concurrently by a pool of threads.[3] 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. Such implementations can skip *implementation* activity 2. Moreover, if in *implementation activity* 3 it is determined that the threads are unbound, there is no need to implement any subsequent implementation activities.

1.4. Encapsulate the lower-level handle set mechanisms with s higher-level patterns (optional): One way to implement the Leader/Followers pattern is to use native operating system handle set event demultiplexing mechanisms, such as `select` and `WaitForMultipleObjects`. Conversely, developers can leverage higher-level patterns, such as Reactor [15], Proactor [10], and Wrapper Facade [16]. These patterns simplify the Leader/Followers implementation and reduce the effort needed to address the accidental complexities of programming to low-level native handle set 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.

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

→ For example, in our OLTP server example, I/O events must be demultiplexed to concrete event handlers, which are dispatched according to which I/O handle received the event. The Reactor pattern [15] supports this activity, thereby simplifying the implementation of the Leader/Followers pattern. In the context of the Leader/Followers pattern, however, a reactor only dispatches *one* concrete event handler during its demultiplexing phase, 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 I/O <HANDLE>.
  virtual HANDLE get_handle (void) const = 0;
};
```

□

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 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 is provided by the reactor implementation, as follows:

```
class Reactor {
public:
  // Temporarily remove the <Event_Handler> from the
  // internal handle set.
  int suspend_handler (Event_Handler *,
                       Event_Type et);
  // Restore a previously suspended <Event_Handler>
  // to the internal handle set.
  int resume_handler (Event_Handler *,
                      Event_Type et);
};
```

□

3. Implement the thread set: To promote a follower thread to the leader role, as well as determine which thread is the current leader, an implementation of the Leader/Followers pattern must manage a set of threads. The two general strategies for implementing thread sets – *unbound* and *bound* – are described below.

• **Unbound thread set:** In this design, all follower threads in the set simply wait on a single synchronizer, such as a semaphore or condition variable. This strategy is designed for unbound handle/thread associations, where it does not matter which thread processes an event, as long as multiple threads sharing a handle set are serialized.

In certain applications, the leader thread need not handoff events to specific follower threads. For example, a "pure" server that receives requests from the network and sends responses through the same connection can assign any thread to process each new event. In this case, only threads engaged in server processing must participate in the unbound leader/followers thread set. These requirements can yield a smaller, simpler Leader/Followers pattern implementations.

→ For example, the unbound thread set class shown below can be used for the back-end database servers in our OLTP example.

```
class Unbound_Thread_Set {
public:
```

Application threads invoke the `join` method to wait to demultiplex and dispatch new I/O events.

```
  int join (Time_Value *timeout = 0);
```

This method blocks until the application terminates or a timeout occurs. *Implementation activity* 4 illustrates how this method can be defined.

The `promote_new_leader` method is invoked by server concrete event handlers before they perform application-specific event processing.

```
  int promote_new_leader (void);
```

This method promotes one of the follower threads in the set to become the new leader. *Implementation activity* 6 illustrates how this method can be defined.

The constructor caches the reactor passed to it. This reactor implementation uses `select`, which only supports iterative handle sets. Therefore, `Unbound_Handle_Set` is responsible for allowing multiple threads to call `select` on the reactor's handle set serially.

```
  Unbound_Thread_Set (Reactor *reactor =
                      Reactor::instance ()):
    reactor_ (reactor) {}
```

By default, our `Unbound_Thread_Set` uses the reactor singleton, though this choice can be overridden easily by an application.

The implementation of `Unbound_Thread_Set` uses the following data members:

```
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_;
};
```

Note that a single condition variable synchronizer is shared by all threads in this set. Moreover, the implementation of the `Unbound_Handle_Set` is designed using the Monitor Object pattern [2]. □

• **Bound thread set:** In this design, each follower thread waits on its own synchronizer. This strategy is well-suited for bound handle/thread associations, where a leader thread may need to handoff I/O 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.

In addition, a bound thread set may be necessary if an application multiplexes connections among two or more threads, in which case the bound thread set 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.

$\rightarrow$ For example, below we illustrate how a bound handle set 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 unbound thread set above, where we illustrated how to use the Leader/Followers pattern to demultiplex an iterative *handle set*.

We first define a `Thread_Context` class that is nested within the `Bound_Thread_Set` class:

```
class Bound_Thread_Set {
public:
  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 ...
  };
```

`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. Thus, a thread that wants to send a request uses the following method to register its associated `Thread_Context` with the bound thread set to inform the set that it expects a response.

```
  int expecting_response (Thread_Context *context);
```

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

After the request is sent, the client thread invokes the `wait` method defined in the `Bound_Thread_Set` class to wait for the response:

```
  int wait (Thread_Context *context) {
    // step (a): wait as a follower or become a leader
    // step (b): dispatch event to bound thread
    // step (c): elect new leader
  }
```

The definition of steps (a), (b) and (c) in the `wait` method of `Bound_Thread_Set` are illustrated in *implementation activities* 4, 5, and 6, respectively.

```
public:
  Bound_Thread_Set (HANDLE mutex_stream_handle)
    : muxed_stream_ (muxed_stream_handle) {}
```

The implementation of `Bound_Thread_Set` uses the following data members:

```
private:
  // Wrapper facade for the the multiplexed
  // connection stream.
  SOCK_Stream muxed_stream_;

  // The thread id of the leader thread.
  // Set to NO_CURRENT_LEADER if there
  // is no current leader.
  Thread_Id leader_thread_;

  // The set 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_;
};
```

By comparing the data members of `Bound_Thread_Set` and `Unbound_Thread_Set`, it is clear that the primary differences are that `Unbound_Thread_Set` contains an *explicit* collection of threads, represented by the `Thread_Context` objects, and a multiplexed `SOCK_Stream` wrapper facade object. In contrast, the collection of threads in the `Unbound_Thread_Set` is implicit, namely, the queue of waiting threads blocked on its condition variable. Thus, each follower thread can wait on a separate condition variable until they are promoted to become the leader thread or receive an event handoff from the current leader. □

4. Implement a protocol to allow threads to initially join (and rejoin) the leader/followers thread set: This protocol is used when event processing is completed 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 set. The protocol implementation depends on which strategy – unbound or bound – is used to implement thread sets, as described below.

• **Unbound thread set:** For unbound thread/handle associations, the join protocol can be implemented by simply calling `wait` on the condition variable used by the thread set.

→ For example, our back-end database servers can implement the `join` method of the `Unbound_Thread_Set` as follows:

```
int Unbound_Thread_Set::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.
      followers_condition_.wait (timeout);

    // Become leader.
    leader_thread_ = Thread::self ();

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

    // Run the reactor to dispatch the <handle_event>
    // hook method associated with the next event.
    if (reactor_->handle_events () == -1)
      return;
    // Reenter monitor to serialize the test
    // for <leader_thread_> in the while loop.
    guard.acquire ();
  }
}
```

Note how the thread alternates between its role as a leader and a follower. In the first part of the loop, the thread waits until it becomes a leader. After becoming the leader, it waits for I/O events. After dispatching an I/O event via its reactor, it re-assumes a follower role. This process continues until the application terminates or a timeout occurs.

Before the thread can process the event it must promote a new leader to handle any other incoming I/O events. The following concrete event handler is responsible for initiating this task, as follows:

```
class LF_Event_Handler : public Event_Handler {
private:
  // Instance of an <Unbound_Thread_Set>.
  Unbounded_Thread_Set unbound_thread_set_;
  // ...
public:
  // Hook method dispatched by a <Reactor> to
  // handle events of a particular type.
  virtual int handle_event
       (HANDLE, Event_Type et) {
    unbound_thread_set_->suspend_handler (this, et);
    unbound_thread_set_->promote_new_leader ();

    // ...application-specific event processing code...

    unbound_thread_set_->resume_handler (this, et);
  }
};
```

□

• **Bound thread set:** For bound thread/handle associations, the follower must first add its condition variable to the map in the thread set and then call `wait` on it. This allows the leader to use the Specific Notification pattern [17, 18] if it must handoff an event to a specific follower thread.

→ For example, our front-end communication servers must maintain a bound set of follower threads. This set is updated when a new leader is promoted, as follows:

```
int
Bound_Thread_Set::wait (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 set.
    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 ();
      ) {
    char buffer[HEADER_SIZE];

    // Leave monitor temporarily to allow other
    // follower threads to join the set.
    guard.release ();
    if (muxed_stream_.recv (handle, buffer,
        HEADER_SIZE) == -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. □

5. Implement the event handoff mechanism: Unbound handle/thread associations do not require event handoffs between leader and follower threads. For bound handle/thread associations, however, the leader thread must be prepared to handoff an event to a designated follower thread. The Specific Notification pattern [17, 18] can be used to implement this handoff 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 set. 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 handoff an event to the thread designated to process the event:

```
int
Bound_Thread_Set::wait (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 ...
}
```

□

6. Implement the follower promotion protocol: Immediately after a leader thread demultiplexes an event, but before it processes the event, it must promote a follower thread to become the new leader. The following protocols can be used to determine which follower thread to promote.

- **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 handoffs 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 [9, 19]. 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.

- **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 [5], which is an example of the "Fresh Work Before Stale" pattern [20]. Implementing a LIFO promotion protocol requires a more complex 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 [11], 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 [21]. Although this protocol is more complex than the FIFO and LIFO protocols, it may be necessary to promote follower threads according to their priorities to minimize priority inversion [11].

• **Implementation-defined order:** This ordering is most common when implementing unbound 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 condition variable:

```
int Unbound_Thread_Set::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 ())
    return -1; // Only leader thread can invoke this.

  leader_thread_ = NO_CURRENT_LEADER;
  followers_condition_.notify ();
  // Release mutex automatically in destructor.
}
```

The `promote_new_leader` method is invoked by concrete event handlers before they start processing a request. □

• **Specific order:** This ordering is common when implementing a bound thread set, where it is necessary to handoff 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_Thread_Set`'s `wait` method to promote a new leader, as follows:

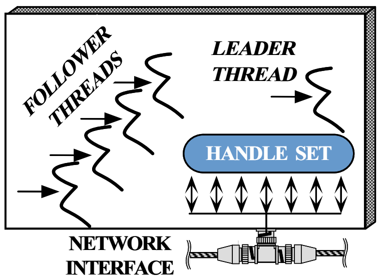
```
int Bound_Thread_Set::wait (Thread_Context *context)
{
  // ... details omitted ...

  // 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 ();
}
```

□

9 Example Resolved

Our OLTP system can apply the Leader/Followers pattern for front-end communication servers and back-end database servers. Below, we illustrate how this pattern can be applied for both use cases.

OLTP back-end database servers: These servers can use the unbound handle set version of the Leader/Followers pattern to implement a thread pool that demultiplexes I/O events 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. These threads are not bound to any particular I/O handle. Thus, all threads in this pool take turns playing the role of the network I/O thread by having the current leader thread `select` on a shared handle set of sockets connected to OLTP front-end servers.

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 process the request event. The former leader thread then reads the request into a buffer that resides in the run-time stack or is allocated using the Thread-Specific

Storage pattern [22].[4] All processing of the data occurs in the former leader thread; thus, no further context switching, synchronization, or data movement is necessary. After processing completes, the former leader thread becomes a follower and returns to 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.

In our OLTP back-end database server example, the current leader thread promotes a follower thread to become the new leader after it demultiplexes a request event. The new leader waits on the synchronous event demultiplexer, while all former leader threads process their request events concurrently. All remaining follower threads queue up on the synchronizer, waiting their turn to be promoted to the leader. 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 the former leader finishes processing its request, it sends the result to the front-end and re-queues itself into the follower thread set.

OLTP front-end communication servers: Front-end communication servers can use the bound handle set version of the Leader/Follower pattern to wait for both requests from remote clients and responses from back-end servers. This design can be structured much like the back-end servers described above. The main difference is that the front-end server threads are "bound" to particular I/O handles once they forward a request to a back-end server. For example, they can wait on a condition variable until the response is received. After the response is received, the front-end server uses the request id to handoff 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.

[4]In contrast, the half-sync/half-reactive thread pool must allocate each request dynamically from a shared heap because the request is passed between threads.

10 Variants

Relaxing serialization constraints: There are platforms where multiple leader threads can wait simultaneously on a handle set. For example, a thread pool can take advantage of multi-processor hardware to perform useful computations while other threads wait for I/O 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 set 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 I/O handles in an application may have dedicated threads to handle certain events, whereas other I/O handles can be processed by any thread. Thus, one variant of the Leader/Follower pattern uses its event handoff mechanism to notify certain subsets of threads, according to the I/O 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 will 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.

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_Thread_Set` cannot simply demultiplex events for a single handle. As with the `Unbound_Thread_Set` 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_Thread_Set` to dispatch the message to the correct thread.

11 Known Uses

ACE Thread Pool Reactor framework [23]. 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 among 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 an event is received the leader thread handles the event and promotes an arbitrary thread to become the next leader.

CORBA ORBs. Many CORBA implementations, including Chorus COOL ORB [11] and TAO [24] 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 I/O 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.

12 See Also

The Proactor pattern [10] 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 pattern [2] is an alternative 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 [22] 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 dispatches the event based on the I/O 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 [25], 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 requires 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 set 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 set of follower threads. All those operations can happen concurrently, in an unpredictable order, thus, the 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 is demultiplexing 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.

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*, to appear.

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

[3] W. R. Stevens, *UNIX Network Programming, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

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

[5] D. A. Solomon, *Inside Windows NT, 2nd Ed.* Redmond, Washington: Microsoft Press, 2nd ed., 1998.

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

[7] D. C. Schmidt, "Monitor Object – an Object Behavior Pattern for Concurrent Programming," *C++ Report*, vol. 12, May 2000.

[8] 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), USENIX, May 1999.

[9] 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.

[10] I. Pyarali, T. H. Harrison, D. C. Schmidt, and T. D. Jordan, "Proactor – An Architectural Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events," in *Pattern Languages of Program Design* (B. Foote, N. Harrison, and H. Rohnert, eds.), Reading, MA: Addison-Wesley, 1999.

[11] 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*, To appear 2000.

[12] 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, MA), pp. 624–633, IEEE, April 1995.

[13] W. Pree, *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley, 1994.

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

[15] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[16] D. C. Schmidt, "Wrapper Facade: A Structural Pattern for Encapsulating Functions within Classes," *C++ Report*, vol. 11, February 1999.

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

[18] D. Lea, *Concurrent Java: Design Principles and Patterns, Second Edition*. Reading, MA: Addison-Wesley, 1999.

[19] 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.

[20] 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, MA: Addison-Wesley, 1996.

[21] 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.

[22] T. Harrison and D. C. Schmidt, "Thread-Specific Storage: A Pattern for Reducing Locking Overhead in Concurrent Programs," in *OOPSLA Workshop on Design Patterns for Concurrent, Parallel, and Distributed Systems*, ACM, October 1995.

[23] 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.

[24] 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.

[25] 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.