Half-Sync/Half-Async **

When developing concurrent software, specifically a concurrent ENCAPSULATED IMPLEMENTATION (313) or a network server that employs a REACTOR (259) or PROACTOR (262) event handling infrastructure . . .

. . . we need to make performance efficient and scalable while ensuring that any use of concurrency simplifies programming.

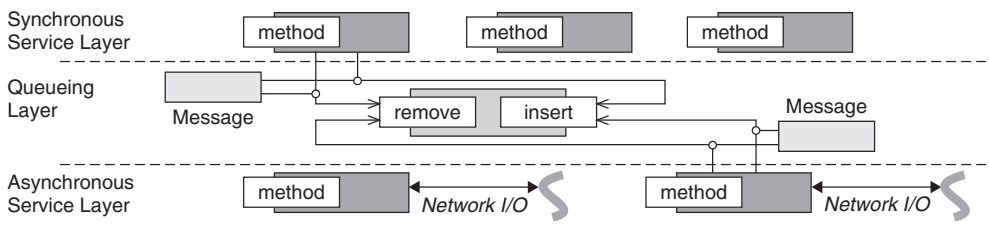
♦♦♦

Concurrent software often performs both asynchronous and synchronous service processing. Asynchrony is used to process low-level system services efficiently, synchrony to simplify application service processing. To benefit from both programming models, however, it is essential to coordinate asynchronous and synchronous service processing efficiently.

Asynchronous and synchronous service processing is usually interrelated. For example, the I/O layer of Web servers often uses asynchronous read operations to obtain HTTP `GET` requests [HPS97]. Conversely, the processing of the `GET` requests at the CGI layer often runs synchronously in separate threads of control. The asynchronous arrival of requests at the I/O layer must therefore be integrated with synchronous processing of the requests at the CGI layer. From a different point of view, similar observations can be made about AJAX—the use of *a*synchronous *Ja*vaScript and *X*ML to increase the perceived responsiveness of Web clients [Gar05]. In general, asynchronous and synchronous services should cooperate effectively and be encapsulated against either other's deficiencies.

Therefore:

Decompose the services of concurrent software into two separated layers—synchronous and asynchronous—and add a queueing layer to mediate communication between them.



Process higher-level services, such as domain functionality, database queries, or file transfers, *synchronously* in separate threads or processes. Conversely, process lower-level system services, such as short-lived protocol handlers driven by interrupts from network hardware, *asynchronously*. If services in the synchronous layer must communicate with services in the asynchronous layer, have them exchange messages via a queueing layer.

♦♦♦

A HALF-SYNC/HALF-ASYNC design enforces a strict separation of concerns between the three layers, which makes concurrent software easier to understand, debug, and evolve. In addition, asynchronous and synchronous services do not suffer from each other's liabilities: asynchronous service performance does not degrade due to blocking synchronous services, and the simplicity of programming synchronous services is unaffected by asynchronous complexities such as explicit state management. Finally, using a queueing layer avoids hard-coded dependencies between the asynchronous and synchronous service layers, as well as making it easy to reprioritize the order in which messages are processed. The strict decoupling of the asynchronous layer from the synchronous layer, however, requires that data exchanged between the two layers must be either communicated as COPIED VALUES, which can introduce performance penalties and resource management overhead the more data there is to pass, or represented as IMMUTABLE VALUES, which are lighter in weight but perhaps more intricate to construct.

In general, a HALF-SYNC/HALF-ASYNC arrangement employs LAYERS (185) to keep its three distinct execution and communication models independent and encapsulated.

Services in the synchronous layer, such as database queries, file transfers, or domain functionality, generally run in their own threads, which allows concurrent execution of multiple services. If realized as an ACTIVE OBJECT (365) a service can also handle multiple service requests simultaneously, which in turn can improve the performance and throughput of an application.

Services in the asynchronous layer can be realized with the help of asynchronous interrupts or operating system APIs that support asynchronous I/O, for example Windows overlapped I/O and

I/O completion ports [Sol98], or the POSIX aio_* family of asynchronous I/O system calls [POSIX95]. WRAPPER FACADES (459) help to encapsulate platform-specific asynchronous I/O functions behind a uniform interface, which simplifies their correct use and the portability of the asynchronous layer to another operating system. Alternatively, if a HALF-SYNC/HALF-ASYNC arrangement is designed in conjunction with a PROACTOR or REACTOR event-handling infrastructure, this event-handling infrastructure *is* the asynchronous layer. Although a REACTOR is not truly asynchronous, it shares key properties with asynchrony if its services implement short-duration operations that do not block for long periods of time.

The queueing layer often consists of a message queue shared by all services in the synchronous and asynchronous layers. Sophisticated queueing layers can provide multiple message queues, for example one message queue for every message priority or communication endpoint. Message queues can be implemented as MONITOR OBJECTS (368) to serialize access to the message queues transparently for asynchronous and synchronous services. TEMPLATE METHODS (453) and STRATEGIES (455) support the setting of various aspects of the message queues, for example configuring their behavior for ordering, serialization, notification, and flow control. STRATEGIES are the more flexible option, offering loose coupling and runtime configuration and re-configuration of the message queues. TEMPLATE METHODS can be appropriate if only compile-time flexibility is needed. The information routed by the queueing layer is encapsulated within MESSAGES (420).

Leader/Followers **

When developing concurrent software, specifically a concurrent ENCAPSULATED IMPLEMENTATION (313) or a network server that employs a REACTOR (259) event-handling infrastructure . . .

. . . we must often react on and process multiple events from multiple event sources both concurrently and efficiently.

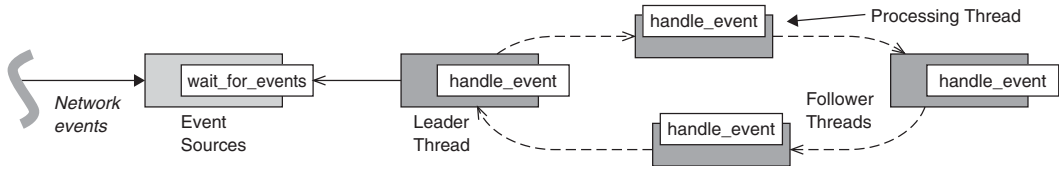
♦♦♦

Most event-driven software uses multi-threading to process multiple events concurrently. It is surprisingly hard, however, to allocate work to threads in an efficient, predictable, and simple manner.

In event-driven software, particularly server software, it is often necessary to define efficient demultiplexing associations between threads and event sources. It is also necessary to prevent race conditions if multiple threads demultiplex events on a shared set of event sources. For example, a Web server may use multiple threads to service multiple `GET` requests scalably on multiple I/O handles. Some methods of associating threads and event sources are inefficient because they incur high levels of overhead. For example, creating a thread for each request, or dedicating a separate thread for each event source, can be inefficient due to scalability limitations of operating systems and hardware. What is needed is an architecture for concurrent reactive software that is both easy to use and efficient.

Therefore:

Use a pre-allocated pool of threads to coordinate the detection, demultiplexing, dispatching, and processing of events. In this pool only one thread at a time—the *leader*—may wait for an event on a set of shared event sources. When an event arrives, the leader promotes another thread in the pool to become the new leader and then processes the event concurrently.



While the *leader* is listening on the event sources for an event to occur, other threads—the *followers*—can queue up and sleep until they are promoted to be the leader. When the current leader thread detects an event from the event sources it does two things. It first promotes a follower thread to become the new leader, then it morphs itself into a *processor* thread that demultiplexes and dispatches the event to a designated event handler that runs in the same thread that received the event. Multiple processing threads can handle events concurrently while the current leader thread waits for new events to occur on the shared event sources. After handling its event, a processing thread reverts to the follower role and sleeps until it becomes the leader again.

♦♦♦

By pre-allocating a pool of threads, a LEADER/FOLLOWERS design avoids the overhead of dynamic thread creation and deletion. Having threads in the pool self-organize and not exchange data between themselves also minimizes the overhead of context switching, synchronization, data movement, and dynamic memory management. Moreover, letting the leader thread perform the promotion of the next follower prevents performance bottlenecks arising from having a centralized manager make the promotion decisions.

The price to pay for such performance optimizations is limited applicability. A LEADER/FOLLOWERS configuration only pays off for short-duration, atomic, repetitive, and event-based actions, such as receiving and dispatching network events or storing high-volume data records in a database. The more services the event handlers offer, the larger they are in size, while the longer they need to execute a request, the more resources a thread in the pool occupies and the more threads are needed in the pool. Correspondingly fewer resources are available for other functionality in the application, which can have a negative impact on the application's overall performance, throughput, scalability, and availability.

In most LEADER/FOLLOWERS designs the shared event sources are encapsulated within a dispatcher component. If a LEADER/FOLLOWERS arrangement is designed in conjunction with a REACTOR event-handling infrastructure, its reactor component *is* the dispatcher. Encapsulating the event sources separates the event demultiplexing and

dispatching mechanism from the event handlers. Providing the dispatcher with methods for deactivating and reactivating a specific event source avoids race conditions if a new leader thread is selected simultaneously with completion of processing of the most recent event.

Specify the threads as a Resource Pool (503), and use a Monitor Object (368) to maintain the dispatcher and synchronize access to the shared event sources. This design enhances performance by using a self-organizing concurrency model that avoids the overhead of a separate queueing layer between event sources and event handlers.

Inside the thread pool the monitor object offer two methods to its threads. A join method allows newly initialized threads to join the pool. The joining thread first waits to become the new leader by suspending its own execution on the thread pool's monitor condition. After it becomes the leader, it accesses the shared event sources to wait for and process an incoming event. A `promote_new_leader` method allows the current leader thread to promote a new leader by notifying a sleeping follower via the thread pool's monitor condition. The notified follower resumes execution of the thread pool's join method and accesses the shared event sources to wait for the next event to occur.

Multiple promotion protocols, such as last-in/first-out, first-in/first-out, and highest priority, can be supported via Template Methods (453) and Strategies (455).

Active Object **

When developing an ENCAPSULATED IMPLEMENTATION (313), the synchronous service layer in a HALF-SYNC/HALF-ASYNC (359) architecture, or service handlers in an ACCEPTOR-CONNECTOR (265) configuration . . .

. . . we must often ensure that the operations of components can run concurrently within their own threads of control.

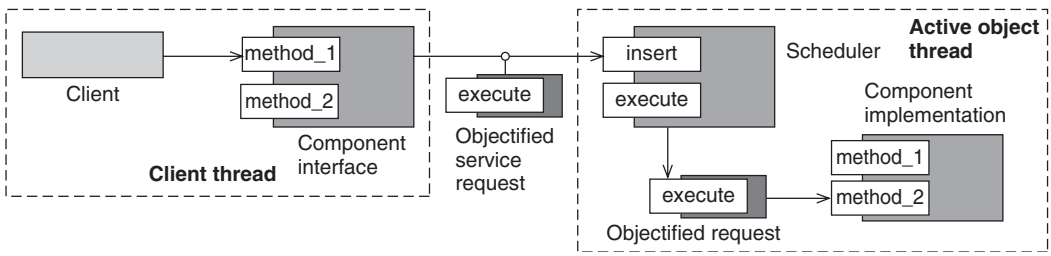
♦♦♦

Concurrency can improve software quality of service, for example by allowing components to process multiple client requests simultaneously without blocking. Developers, however, must decide how to express the units of concurrency in their software and how to interact with them as they run.

In particular, clients should be able to issue requests on components without blocking until the requests execute. It should also be possible to schedule the execution of client requests according to specific criteria, such as request priorities or deadlines. To keep service requests independent, they should be serialized and scheduled transparently to the component and its clients, thereby enabling the reuse of software implementations that require different synchronization strategies.

Therefore:

Define the units of concurrency to be service requests on components, and run service requests on a component in a different thread from the requesting client thread. Enable the client and component to interact asynchronously to produce and consume service results.



Clients can initiate a service request on the component by calling a method on its interface, which is exposed to the clients' thread(s). Design the component's interface without subjecting it to synchronization constraints and return control immediately to clients after they issue their requests. In addition, objectify the requests, pass them to the component implementation running in one or more separate threads, and let the implementation schedule the execution of the requests independently of the point in time at which they were initiated. Provide a way for the component to return results to the client when the service is complete.

♦♦♦

An ACTIVE OBJECT design enhances concurrency in an application by allowing clients threads and the execution of service requests to run simultaneously: clients are not blocked while their service requests are executed. In addition, synchronization complexity is reduced by using a scheduler that evaluates synchronization constraints to serialize access to the component's implementation. The separation of request scheduling from the actual component implementation also supports the reuse of the component in scenarios that do not require synchronization, as well as enhancing legacy components that are not designed for concurrent access to be adapted for use in a concurrent application. Finally, the order of service request execution can differ from service request invocation, which can better account for priorities, deadlines, and other synchronization constraints—but at the cost of complicated debugging. An ACTIVE OBJECT arrangement also introduces a heavyweight request handling and execution infrastructure, which can cause performance penalties for components that only implement short-duration methods.

Use an EXPLICIT INTERFACE (281) to expose the component's interface to client threads. This design allows clients to access the component is if it were collocated in their own thread. Design the interface so that its method signatures do not include synchronization parameters. Clients therefore appear to have an exclusive access to the concurrent component even if it shared by multiple client threads.

At runtime, let the component interface objectify all method invocations into service requests, which are typically realized as COMMANDS (435) that convey the necessary synchronization constraints of their corresponding method invocations. This request objectification

decouples service requests from service execution in time and space, so that each client can invoke services on the component without blocking itself or other clients. Store the created service requests into a shared activation list that maintains all pending service requests on the concurrent component. Implementing the activation list as a MONITOR OBJECT (368) helps to ensure thread-safe concurrent access.

One or more threads host the component's implementation. Within each thread, a servant implements the component's functionality. A scheduler dequeues pending service request objects from the shared activation list and executes them on the servants. Such a design allows service requests and executions to run concurrently, that is, service requests are invoked in client threads, while service executions run in different threads. In addition, the scheduler separates component functionality from scheduling and synchronization mechanisms, which supports independent realization and evolution of both concerns.

Design the scheduler as a COMMAND PROCESSOR (343) that implements the component's event loop. This monitors the activation list to identify service requests that become executable, removes these requests from the activation list, and executes them on their servant. TEMPLATE METHODS (453) and STRATEGIES (455) can support multiple scheduling policies within the scheduler. TEMPLATE METHODS are most appropriate if the configuration of the scheduler is possible at compile time. STRATEGIES, in contrast, support runtime configuration and reconfiguration of scheduling policies.

Clients can obtain the result of a service request on the concurrent component via a FUTURE (382). The interface of the concurrent component returns the future to the client after the service's invocation, while the associated service request fills the future after the servant has finished with the service's execution. If the client accesses the future before it contains the service's result, the client can block or poll until the result is available. When futures are no longer needed they can be reclaimed safely via AUTOMATED GARBAGE COLLECTION (517) if this strategy is supported by the programming language, or via a COUNTING HANDLE (522) if the reclaim must be coded manually.

Monitor Object **

When developing a SHARED REPOSITORY (202) architecture, a REQUESTOR (242), CLIENT REQUEST HANDLER (246), MESSAGE CHANNEL (224), MESSAGE ROUTER (231) distribution infrastructure, ENCAPSULATED IMPLEMENTATION (313), ACCEPTOR-CONNECTOR (265) arrangement, HALF-SYNC/HALF-ASYNC (359), LEADER/FOLLOWERS (362), or ACTIVE OBJECT (368) concurrency model . . .

. . . we must consider that objects can be shared between threads.

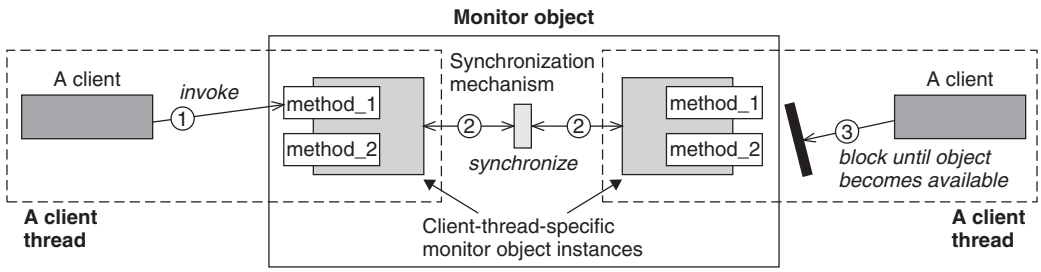
♦♦♦

Concurrent software often contains objects whose methods are invoked by multiple client threads. To protect the internal state of shared objects, it is necessary to synchronize and schedule client access to them. To simplify programming, however, clients should not need to distinguish programmatically between accessing shared and non-shared components.

Instead, each object accessed by multiple client threads should ensure that its methods are serialized transparently without requiring explicit client intervention. To ensure the quality of service of its clients, a shared object should also relinquish its thread of control voluntarily if any of its methods must block during execution, leaving the component in a stable state so that other client threads can access it safely.

Therefore:

Execute a shared object in each of its client threads, and let it self-coordinate a serialized, yet interleaved, execution sequence. Access the shared object only through synchronized methods that allow execution of only one method at a time.



Each monitor object contains a monitor lock that it uses to serialize access to the object's state. Within a synchronized method, first acquire the monitor lock to ensure no other synchronized methods can execute. Once the lock is held, evaluate whether the shared object's current state allows the synchronized method to run. If it does, execute it, otherwise suspend the execution of the synchronized method on a condition. If called, a monitor condition should suspend the thread of its caller until it is notified to wake it. When suspending a thread, the monitor condition should also release the monitor lock, and when resuming this thread, re-acquire the monitor lock.

Suspending a synchronized method allows other client threads to access the shared object via its synchronized methods. Any synchronized method that executes, completing execution, may affect the validity of monitor conditions, in which case it should notify the corresponding monitor condition so that suspended method invocations can resume execution. Before terminating a synchronized method, release the monitor lock so that other synchronized methods called by other threads can execute.

♦♦♦

Designing a shared object type as a MONITOR OBJECT simplifies concurrency control by sharing the object among cooperating threads and combining state synchronization with method invocation. A MONITOR OBJECT also helps in implementing a cooperative method execution sequence that ensures the availability of the shared object to its clients, and maximizes its availability within the constraints of serialization, to ensure that state changes are complete and free of race conditions. A liability of this pattern, however, is that it couples domain functionality tightly with synchronization aspects. It can be hard to compose or nest monitor objects without risking deadlock, for example, if one monitor object makes callbacks into objects that in turn use other monitor objects.

A monitor object can use a THREAD-SAFE INTERFACE (384) to decouple synchronization from its functionality. Such a design also allows both concerns to vary independently. GUARDED SUSPENSION (380) can be used to coordinate threads running in the object. The execution of methods is scheduled via monitor conditions and monitor locks that determine the circumstances under which they should suspend or resume their execution and that of collaborating components.