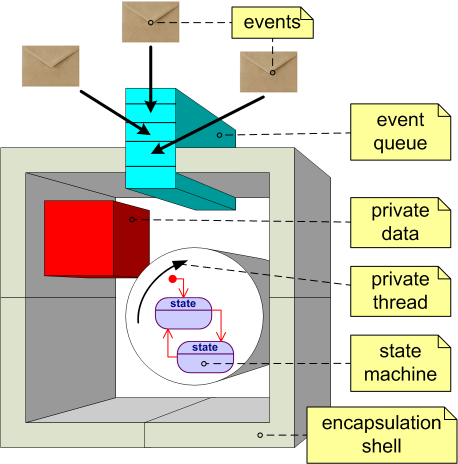
## Active Object (Actor) Design Pattern

The [best practices](https://www.state-machine.com/active-object#BestPractices) just outlined are collectively know as the **Active Object (or Actor) design pattern**. In this pattern, Active Objects (Actors) are event-driven, strictly [encapsulated](https://www.state-machine.com/active-object#Encapsulation) software objects running in their own threads of control that communicate with one another [asynchronously](https://www.state-machine.com/active-object#Asynchronous) by exchanging events.

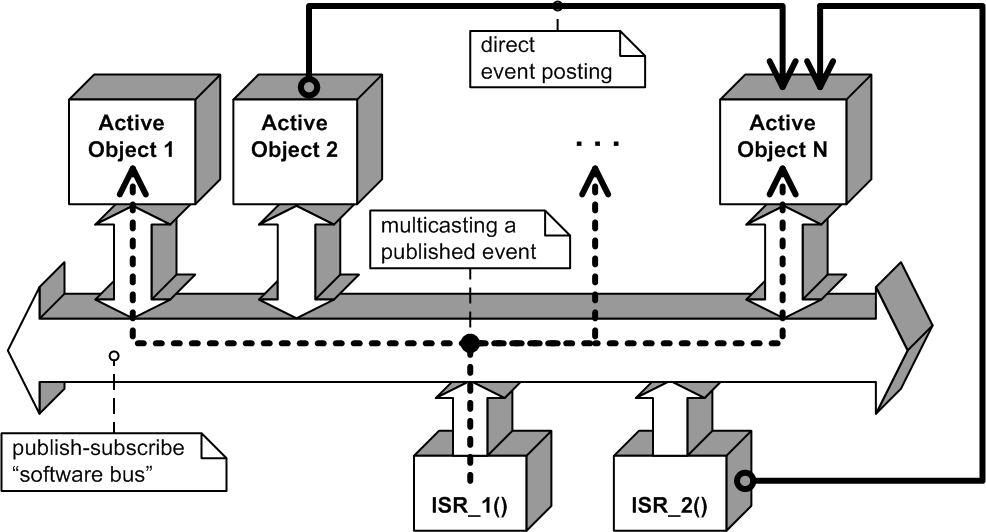


### True Encapsulation for Concurrency

In a sense Active Objects are the most stringent form of [object-oriented programming (OOP)](https://www.youtube.com/playlist?list=PLPW8O6W-1chzoLFm2eLy11AoGiYbApjc4), because the asynchronous communication enables active objects to be truly **encapsulated**. In contrast, the traditional OOP encapsulation, as provided by C++, C# or Java, does not really encapsulate anything in terms of concurrency. Any operation on an object runs in the caller’s thread and the attributes of the object are subject to the same race conditions as global data, not encapsulated at all. To become thread-safe, operations need to be explicitly protected by a mutual exclusion mechanism, such as a mutex or a monitor, but this [causes contention, reduces responsiveness (blocking), and often leads to missed real-time deadlines](https://www.state-machine.com/doc/Sutter2009a.pdf).

In contrast, all private attributes of an active object are truly encapsulated without any mutual exclusion mechanism, because they can be only accessed from the active object’s own thread. Note that this **encapsulation for concurrency** is not a programming language feature, so it is no more difficult to achieve in C as in C++, but it requires a programming discipline to avoid sharing resources ([shared-nothing principle](https://en.wikipedia.org/wiki/Shared_nothing_architecture)). However, the event-based communication helps immensely, because instead of sharing a resource, a dedicated Active Object can become the manager/broker of the resource and the rest of the system can access the resource only via events posted to this broker Active Object.

### Asynchronous Communication



Each Active Object has its own event queue and receives all events exclusively through this queue. Events are delivered **asynchronously**, meaning that an event producer merely posts an event to the event queue of the recipient active object but does **not** wait ([block](https://www.state-machine.com/active-object#Blocking)) in line during the actual processing of the event. The event processing occurs always in the thread context of the recipient active object. The active object infrastructure (framework), such as the [QP RTEF](https://www.state-machine.com/products/qp/), is responsible for delivering and queuing the events in a thread-safe and deterministic manner.

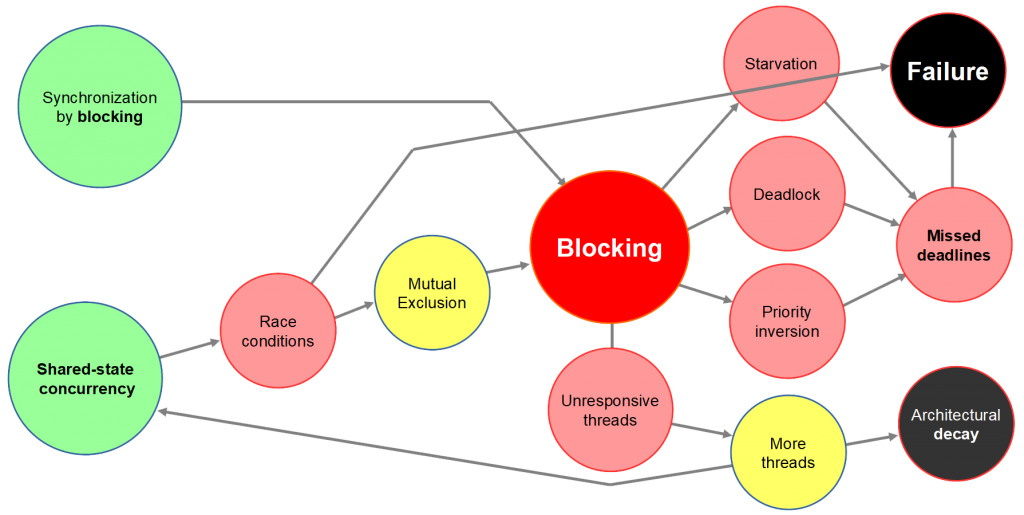
### Run-to-Completion (RTC)

Each Active Object handles events in run-to-completion (RTC) fashion. RTC simply means that an Active Object handles one event at a time, that is, the Active Object must always complete the processing of the previous event before it can start processing the next event from its queue.

For example, under a **preemptive** kernel an RTC step can be preempted by another thread executing on the same CPU. This is determined by the scheduling policy of the underlying kernel, not by the Active Object model. When the suspended thread is assigned CPU time again, it resumes from the point of preemption and, eventually, completes its event processing. As long as the preempting and the preempted threads do not share any resources, there are **no concurrency hazards**.

### No Blocking

Most conventional operating systems manage the threads and all inter-thread communication based on **blocking**, such as waiting on a semaphore or a time-delay. However, blocking is the root cause of the whole slew of problems. The central issue is that while a thread is blocked waiting for one type of event, the thread is not doing any other work and is **not responsive** to other events. Such a thread cannot be easily extended to handle new events.



In contrast, event-driven Active Objects don’t need to block, because in event-driven systems the control is [inverted](https://www.state-machine.com/event-driven-programming/#Inversion) compared to traditional threads. Instead of blocking to wait for an event, an active object simply finishes its RTC step and returns to the supervisory event-driven framework to be activated when the next event arrives. This arrangement allows active objects to remain **responsive** to events of all types, which is central to the unprecedented flexibility and extensibility of Active Object systems.