
JActor2

JActor2 Revisited

JActor2 is a robust and high-performance alternative to threads and locks.

JActor2 Revisited focuses on a subset of the API that is easy to learn but reasonably comprehensive.

By Bill la Forge, 2014

The Problem with Threads

- The problems are well known: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf
- Acutal performance gains can be difficult to achieve.
- Testing is inadequate in the face of race conditions, live locks, deadlocks.
- Taking advantage of additional threads can be difficult.

The Problem with Actors

- Actors often block processing of messages relating to new activities until older activities are completed.
- For actors there is no concept equivalent to locking order. Deadlock avoidance is left entirely in the hands of the developer.
- Actors are coupled. When the protocol used by an actor is changed, other actors are effected.

JActor2 Model: Message Passing

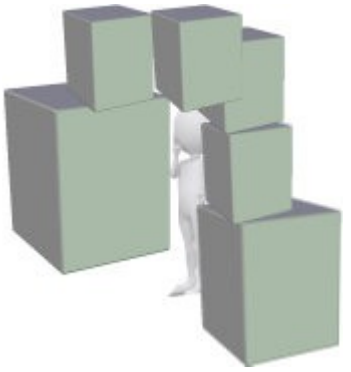
- *Reactors* are light-weight threads that process the messages one at a time.
- *Blades* are application objects that perform the actual message processing. Every *Blade* has an associated *Reactor* for sending and receiving messages.
- Race conditions can not occur when variables are only updated during message processing. Easily confirmed by a code review.
- Vertical scaling is a natural consequence, providing care is taken to avoid bottlenecks.

The JActor2 Model: Message Types

- The three types of messages are requests, responses and signals.
- Requests are analogous to OO method calls in that control is always returned eventually. Requests always return a response or an exception.
- Signals do not return control. Uncaught exceptions raised during signal processing are logged.

JActor2 Model: Isolation of Requests

- Once processing has begun for a request, only responses and signals are processed until a response is returned for that request. Requests are completely isolated. (The I in ACID)
- Blades can process requests from multiple sources, so a request that employs other Blades may not be fully atomic. (Shared dependencies.)



JActor2 Model: The Work Queue

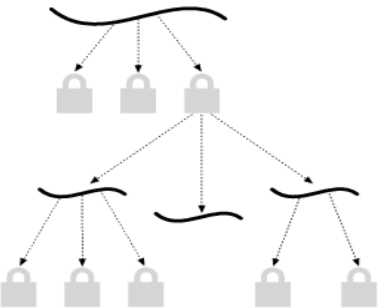
- When a message is passed to a *Reactor* and the *Reactor* is not active (has no thread), the *Reactor* is added to a work queue.
- A pool of threads all try to read *Reactor*s from the same work queue. When read, if the *Reactor* is already active then the thread just reads another *Reactor*.
- The *Reactor*'s messages are processed until the only messages that are left are requests awaiting the completion of the current request, after which the thread reads the next reactor.

JActor2 Model: Message Buffering

- Requests and responses are not sent immediately, but grouped into buffers and passed after the current message is processed to reduce the cost of message passing.
- When the last buffer is passed, if the destination *Reactor* is not active, the current *reactor* is added to the work queue and the destination *Reactor* is processed instead for an improvement in overall performance.
- Signals are not buffered, but passed immediately to their target *Reactor*'s input queue.

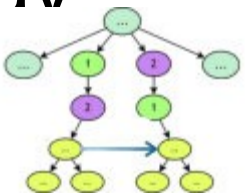
Deadlocks

- When two threads each hold a lock and try to acquire the lock held by the other thread, you get a deadlock.
- Similarly you can have two *Reactor*s which send requests to each other but will not process any subsequent requests until they get a response from the reactor.
- The same happens with actors, only nobody calls them deadlocks.



Partial Ordering

- One way to avoid deadlocks is to observing a partial ordering. So for a given set of locks, they will always be acquired in the same order.
- Partial orderings are addressed when designing the software. Problems arise when maintaining a large program over an extended period of time. Locking order documents may not be maintained, may not be consulted for every change or may not even exist.
- Testing doesn't help either, as deadlocks may occur infrequently.



JActor2 Model: Partial Ordering

- If *Reactor*s never send requests to other *Reactor*s which have sent them a request, even indirectly, then there will be no deadlocks.
- This partial ordering in which *Reactor*s send to which other reactors is tracked at runtime. And any attempt to send a request which violates the partial ordering observed to date raises a runtime exception.
- System tests with reasonable coverage will now detect partial ordering failures, adding significantly to overall robustness.

https://github.com/laforge49/JActor2