Should “Fireability” be Replaced by Pinning?

Peter Dibble

Fireability was introduced in RTSJ 1.0.2. Primarily, fireability addresses a problem with reference counting a scoped, non-default initial memory area for an async event handler. Briefly, the spec requires the reference count to be non-zero from the first release through the last release, but when is the last release? Unlike threads, async event handlers don’t have a well-defined lifecycle. A permissive approach could leave the reference count non-zero when the AEH is gone, resulting in scoped memory leakage.

Broadly speaking, the fireability semantic says the reference count should be non-zero only while the async event handler can be “fired” – that is: released. This includes while it is attached to an async event, or is a miss handler or overrun handler for a live thread or fireable async event handler. In practice fireability is messy to implement and hostile to the application developer. (This could be the topic for a paper of its own.) Becoming non-fireable entails finalization which has a hard-to-predict cost, and finalization may render the async event handler fireable again (leading to complex and ill-defined sequences of interactions). Becoming fireable is usually cheap, but it may block while an async event handler is becoming non-fireable, so even this is not safe. Becoming non-fireable requires that the implementation track all actions that can affect the fireable condition, and the set of actions and their exact timing is hard to define.

In a complex application, the cost of changing the fireability of async event handlers, and the point at which fireability-related costs accrue may be hard to predict. In short: to avoid a potential memory leak, fireability imposes heavy burdens on both the implementation and the application developer.

The fireability semantics only apply to applications that construct async event handlers with an initial memory that is a scoped memory other than the current memory area – a non-default initial memory area. Since the scoped memory leakage problem addressed by fireability was only discovered through a thought experiment, never the subject of a bug report, it appears that this situation is rare.

The root of the fireability problem is the RTSJ semantic that dictates that the initial scoped memory area of an async event handler must not be cleared between releases (assuming no other uses of the scope). This semantic has two perceived advantages:

1. The async event handler can store persistent data in its initial memory area. (Regardless of the lifetime of objects in the handler’s initial memory area, the handler can store persistent data in the object executing the handler’s logic.)

2. The async event handler may defer the cost of finalizing objects in its initial memory area. Instead of executing finalizers at the end of each release it executes them when the handler becomes non-fireable.

Set against these are the above disadvantages of fireability, plus the problem that determining the correct size for the handler's initial memory may be difficult when its fireability state becomes complex, as when multiple asynchronous activities contribute to a handler's fireability. These are the kind of problems that may not appear under testing or even in early deployed versions of software. This type of problem tends to lie dormant until the software is in the field or has been updated and extended.

In RTSJ 1.0.2, a developer can circumvent problems related to the treatment of a handler's initial memory area by not using a scoped non-default initial memory area for async event handlers or by entering another scope in the async event handler before doing any allocation. In short, there are ways to achieve very nearly the same semantics as a scoped-non-default initial memory area, without actually using one and incurring all of the fireability complexity.

RTSJ 1.1

The JSR-282 Experts Group is considering either modifying the behavior of scoped non-default initial memory areas for async event handlers such that they are entered on each release, or modifying the fireability semantics to support an extended lifetime for objects in a handler's initial memory area in a way that is transparent to the application.

The least dramatic change as part of the JSR-282 JSR proposal, as currently conceived, will include a constructor switch for async event handlers that allows them to use existing fireability semantics to control the initial memory area's reference count, or simply have the handler enter and leave the initial memory area on each release.

In addition, the Experts Group has created an API for "pinning" scoped memories: a pinned scope cleared is not cleared when its reference count goes to zero. The pinning facility could be used to generate nearly the same behavior as fireability by specifying that an async event handler pins its initial memory area the first time it is released. The initial memory area may be unpinned at any time under application control, or (roughly speaking) it will be automatically unpinned when the initial memory area becomes unreachable or its parent scope is about to become unreferenced.

Summary of Alternatives

The alternatives currently on the table are:

1. Add an option for async event handlers to enter the initial memory area on each release but otherwise make no changes.

2. Give async event handlers a constructor switch that selects whether the JVM pins the scoped non-default initial memory area on the first release, or uses simple enter per release with no pinning. Within this alternative is the question of whether pinning should be the default behavior.

3. Change the semantics for async event handler releases to enter-per-release. Pinnable scopes (and other tools that already exist in RTSJ 1.0.2) are available for the developer to use if they want to control the life-time of objects in the initial scoped memory.

Compatibility Considerations

Option (1) maintains compatibility with the existing spec, but does little to improve the situation. The constructive effect of option (1) could be improved by deprecating the old behavior.

Option (2) breaks compatibility with the existing spec for applications that depend on the details of when async event handlers become unfireable.

> Such an application would depend on a handler's initial memory area being de-referenced when it is detached from all sources of fireability instead of letting the de-reference be deferred until the program automatically unpins the initial memory area. More specifically:
>
> The initial memory of a handler, `aeh`, persists across multiple fires, but can be cleared by
>
> ```
> ae.setHandler(null);
> ```
>
> ```
> ae.setHandler(aeh);
> ```

The incompatibility introduced by option (2) can be worked around by explicitly unpinning the handler's initial memory area when the handler would have become non-fireable.

Option (3) breaks compatibility most aggressively since it requires a source code change (adding pinning) to get the behavior of option (2) and otherwise manages the reference count of the initial memory area in a fashion unlike previous versions of the RTSJ. It is, however, probable that most RTSJ users would prefer the new behavior.

The Experts Group's Opinions

The Expert Group believes that:

- The fireability semantic is painfully complicated, and has a tendency to introduce hard-to-control CPU consumption.
- Few applications take advantage of the persistence of objects in async event handlers' initial memory. If there are such applications, they will probably want to make other source changes to capitalize on RTSJ 1.1 features and could therefore adjust to either option (2) or (3) with little pain.
- Most likely no applications, other than artificial constructs such as the RTSJ test suite, depend on the details of the fireability semantics, so option (2)'s incompatibility with version 1.0.2's behavior will probably effect no applications.

- Both option (2) and (3) simplify RTSJ implementation and remove a distracting bit of complexity from the specification. Option (3) is the better of the options in both of these dimensions.

The Expert Group would like to select option (3), but does not want to break compatibility in a way that harms existing applications. Option (2) would be a fallback, with less impact on compatibility. We hope to measure the compatibility issue by soliciting input from the community of RTSJ users.