

# Concurrency Concerns in Rich Internet Applications

James Ide<sup>1</sup> Rastislav Bodik<sup>1</sup> Doug Kimelman<sup>2</sup>

<sup>1</sup>University of California, Berkeley

<sup>2</sup>IBM T.J. Watson Research Center

**Abstract** Linguistic constructs for concurrent programming, such as atomic regions, typically intend to orchestrate low-level code and strive to support general-purpose programming. We show that concurrency concerns (1) exist also at a higher-level of software design and (2) are somewhat domain-specific in nature. This observation motivates domain-specific high-level abstractions. Specifically, we look at Rich Internet Applications, exemplified by Google Maps and the Web version of Microsoft Outlook, which are written using scripting frameworks such as AJAX and Flash. These frameworks provide event-driven programming models that are single-threaded and non-preemptive. While these models avoid the need for low-level synchronization, they are inadequate for preventing concurrency problems at the higher level of the program. We identify three fundamental sources of concurrency in these applications: asynchronous server data exchange, script execution during loading, and animation. We present preliminary ideas for suitable abstractions that would simplify correctness validation and expose parallelism.

## Introduction

Event-driven programming is considered safer for concurrency than threads because the former guarantees absence of data races when event handlers run to completion on a single thread. Unfortunately, atomic event handling does not eliminate higher-level concurrency concerns such as those arising from asynchronous communication with the server. We examine concurrency and non-determinism present in the increasingly more popular rich internet applications (RIAs) such as Gmail, implemented in event-driven frameworks such as AJAX.

Comforted by event handler atomicity, programmers often consider these applications safe from races. We show that at least three areas of RIA programming exhibit concurrency (sometimes also non-determinism) and may lead to bugs. For instance, these frameworks offer no support for applying multiple animations to a document

element: when an icon is animated by two events, either the animations are serialized or one is lost in a race; in contrast, the desired behavior is to merge the animations.

We identify three sources of concurrency in rich internet applications: asynchronous server data exchange, executing scripts during document loading, and animation. We show why races can occur and how they can result in bugs. We illustrate inadequacy of current programming models and propose higher-level abstractions for addressing the concurrency. We hope that such abstractions will help application of formal methods, such as concise specification and effective verification. Needless to say, we pose more problems than we solve.

## Asynchronous Server Data Exchange

One source of concurrency in RIAs is the asynchronous exchange of data with servers in response to user interactions. For example, in a mapping application, a user can zoom into a map before a previous zoom has completed.

*How do concurrent events arise?* Because the event-driven programming model handles UI events as soon as possible, additional requests to the server might be generated before the server has responded to previous requests, resulting in multiple in-flight requests. These requests are not only asynchronous—*i.e.*, the application does not wait for a response but proceeds and handles the response when it arrives—but the responses can also arrive out of order.

*An example of a resulting race condition.* Consider the following fragment of a simplified mapping application. When a user interaction occurs, the application requests the corresponding map tile from the server and draws the served tile onto the screen. The mapping application also caches the tile.

```
global cache
atomic map(req):
    if (req.coord in cache)
        tile=cache.read(req.coord)
    else
        tile = server.get(req)
        cache.put(req.coord,tile)
    display.draw(tile)
```

The programmer prefers to think of the interaction with the server as a synchronous call, *i.e.* as if the application waits for the response from `server.get(req)` before proceeding. In other words, the programmer wants the call to `map(req)` to be atomic; this atomicity guarantees two important invariants that are relied on in the mapping program:

- *Consistency*: Once all outstanding requests have been serviced, the tile drawn on the display corresponds to the last request from the user.
- *Responsiveness*: The tile drawn on the display is stored in the cache. This invariant ensures that panning back and forth does not result in re-requesting tiles from the server.

In the AJAX framework, this application could be implemented with event handlers:

```
global cache
handler for map(req):
  if (req.coord in cache)
    tile = cache.read(req.coord)
    display.draw(tile)
  else
    server.get(req)

handler for server_response(tile):
  cache.put(req.coord,tile)
  display.draw(tile)
```

This straightforward implementation is not correct in the presence of response reordering. For example, the second invariant can be broken by this sequence (we are assuming that the cache has bounded capacity and hence tiles need to be evicted):

1. request 1: misses in the cache, asks server for tile 1, waits...
2. request 2: hits in the cache, displays tile 2
3. request 1: server responds with tile1, which evicts tile2

*A generic solution.* The first solution that comes to mind is to implement a networking layer that guarantees in-order request delivery. This solution is generic but may hurt latency in cases when waiting for the earlier-but-delayed response is not necessary, as is the case in the mapping application (old tiles can be safely dropped). We prefer a

solution that allows application-specific handling of out-of-order requests.

*A domain-specific solution.* A solution that we consider ideal is to allow the programmer to express application-specific correctness conditions, from which a suitable tool can generate code for handling reordered responses. In our mapping example, the programmer may wish to stipulate that, once all requests become quiescent, the application will never display a tile that is not from the most recent request; and he could specify that the cache only contains tiles that have been displayed. How to generate response-handling code from such specifications is an open question, as far as we know.

## Executing Scripts During Document Loading

Another source of concurrency stems from “non-atomic application loading,” which makes it possible for code execution to be non-deterministically interleaved with the loading of document that the code manipulates.

A typical browser application comprises an HTML document and JavaScript handlers that respond to events and modify the document. In order to allow programmers to execute scripts before the document is fully loaded, browser semantics specifies that (1) a script executes as soon as it is encountered in document parsing. In order to prevent non-determinism, the semantics stipulates that (2) document parsing pauses while the script is executing.

It turns out that the second rule is insufficient to prevent non-determinism: When a script registers an event handler while parsing is paused, the handler for the event may subsequently be non-deterministically interleaved with parsing. Because handler execution races with parsing, it is unpredictable what parts of the document have been constructed when the handler executes. An Opera browser developer describes a script in `nytimes.com`, which—assuming it executes before the document completed loading—modified the document data structure [1]. The Opera browser, however, executed the script after the document was completely loaded, and the script overwrote the

completed page. The programmer's intention was probably to delay a script to overlap its execution with image loading, when the browser is idle, but the effect was that the New York Times web page appeared blank in the Opera browser for a few days.

Bringing order to the interleaving of code execution with document loading would solve a problem common in AJAX component programming. A JavaScript library often access a document element created in the HTML file that includes the library. For example, that element may be a frame into which a photo library displays its images. If that document element is created after the library script executes, the library may break, as was the case with an Orkut photo gallery [1]. This situation is analogous to referring to a variable that has not yet been declared. It seems urgent to fix such basic scoping issues in the platform that is rapidly growing in popularity.

Any solution that addresses the loading timing issues will inevitably need to balance application responsiveness (we want to execute the scripts as early as possible) and program complexity (eager execution of scripts will require that we synchronize them properly with document loading). Let us outline two alternative solutions.

1. *The programmer views the application loading as atomic.* This stricter semantics would execute all handlers only after the document is completely loaded. Nonetheless, to improve responsiveness, the browser implementation would be free to execute the scripts early, as long as the semantics is observationally maintained. In some cases, this will prevent early execution of a script: if a script's goal is to count all images in the document, its results cannot be displayed before the document is completely loaded. We believe that this is a rare case; a more typical script would indeed lead to better responsiveness, at the cost of more sophisticated runtime support: for example, if a script decorates images on the page, the effects of the script could be made visible before the document is fully loaded, as long as a deferred execution mechanism executes the script on remaining images as they are loaded.

2. *Scripts execute as they are encountered in the document but are extended with a static interface.* To ensure that eagerly executed scripts refer only to present document elements, one can envision statically typing the libraries. The type would convey which parts of the document the script may refer to. A verifier could check that the document elements will be present when the script executes. This static interface may also enable us to verify that events inserted by eager scripts do not non-deterministically interfere with a loading document, as was the case with nytimes.com on Opera.

## Animation

Another source of concurrency in browser scripting is animations. An animation is the process of transforming the appearance of a document element over time. For example, the closing of a menu may be emphasized by gradual shrinking over a period of one second.

*How do concurrent animation activities arise?* It is not uncommon that multiple animations are ongoing simultaneously. For example, when the user wishes to preview driving directions, the mapping application may play the route, simultaneously panning and zooming in on the map. Panning and zooming are two separate animations that have different effects on the map.

*Can race condition arise between two animations?* Animations that modify different elements typically do not lead to race conditions. Similarly, absence of a race is typically guaranteed when two animations modify distinct attributes of an element. For example, it is safe for one animation to transform the size of a bouncing ball and another to transform the opacity of the ball. However, if two animations modify the same attribute of an element, a careless implementation may cause a race on accesses to the attribute. The effect may be either undesirable behavior (*e.g.*, one of the animations does not occur) or the element may be corrupted.

*A race scenario.* Let us illustrate the race on a hypothetical icon dock, similar to that in Mac OS X. The dock is a bar that displays icons of favorite applications. The dock can be transformed by two

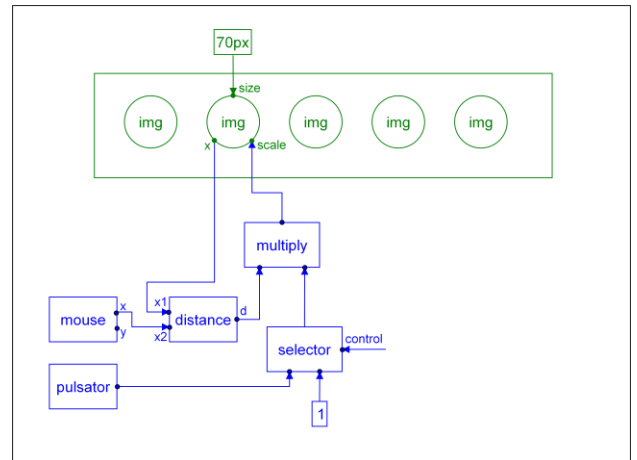
animations, both writing the *size* attribute of an icon: the first animation enlarges icons that are close to the mouse; the second animation causes pulsation (repeated expansion and contraction) of icons that are in alert state, e.g., the mail application icon when new mail has arrived. The two animations can simultaneously affect the same icon; the desired behavior might be to combine the scaling factors of the two animations, e.g. an icon that is both close to the mouse and in alert state would pulsate with respect to an enlarged size.

*A race bug:* Animations are typically implemented by transforming the element at frame granularity. Once per 30ms or so, each active animation updates the pertinent attributes, after which the document goes through layout and rendering. If the implementation allows both of the animations to proceed independently, the following race bug might occur: Both animations update the size of the icon element in each frame; the second animation overwrites the effects of the first one, the result of which is that the first animation has no effect while the second one is active. Once the second animation finishes, the effects of the first one reappear. This is the animation semantics in jQuery, a popular JavaScript framework [2]. Note that this behavior does not combine the animations as we desired (which was combining the scaling effects of the two animations). One might therefore consider jQuery to allow races among animations.

*Composing animations.* If the animation library does not support *compositions of animations* that transform the same attribute, two workarounds offer themselves. First, the programmer may apply the first animation on the icon element, and the second animation on the icon's containing box. If the size of the icon is given as percentage of its containing box, the two animations will combine as desired. Most programmers will consider this a crude hack, but the second workaround is much trickier: whenever the mouse moves, the script can stop the pulsating animation and restart it with an icon size adjusted for the mouse position. This solution is tricky because restart of the animation must appear not to interrupt the pulsating cycle.

*Proposed solution.* Suitable abstractions for animation should not only prevent races among animations on the same element, but it should also

allow a semantic merge of the animations. Our example below suggests a dataflow model in the spirit of Max/MSP [3] or LabView. Races are detected syntactically, for example by checking whether two distinct sources control the *scale* port of an image. These races are resolved by merging, in this case with the *multiply* operator.



## Conclusion

Concurrency in rich internet applications is often overlooked. We have described three sources of concurrency and non-determinism: animation, asynchronous server data exchange, and eager script execution during document loading. We wish to argue that these issues can only be addressed effectively at a higher semantic level than is currently provided.

## References

- [1] Hallvord R. M. Steen, "Websites Playing Timing Roulette," <http://my.opera.com/hallvors/blog/2009/03/07/websites-playing-timing-roulette>
- [2] jQuery, <http://jquery.com/>
- [3] Max/MSP, <http://www.cycling74.com/products/max5>