A Fresh Approach to Concurrency in (server-side) JavaScript

Hannes Wallnöfer
RingoJS

10+ years of server-side JavaScript on the JVM

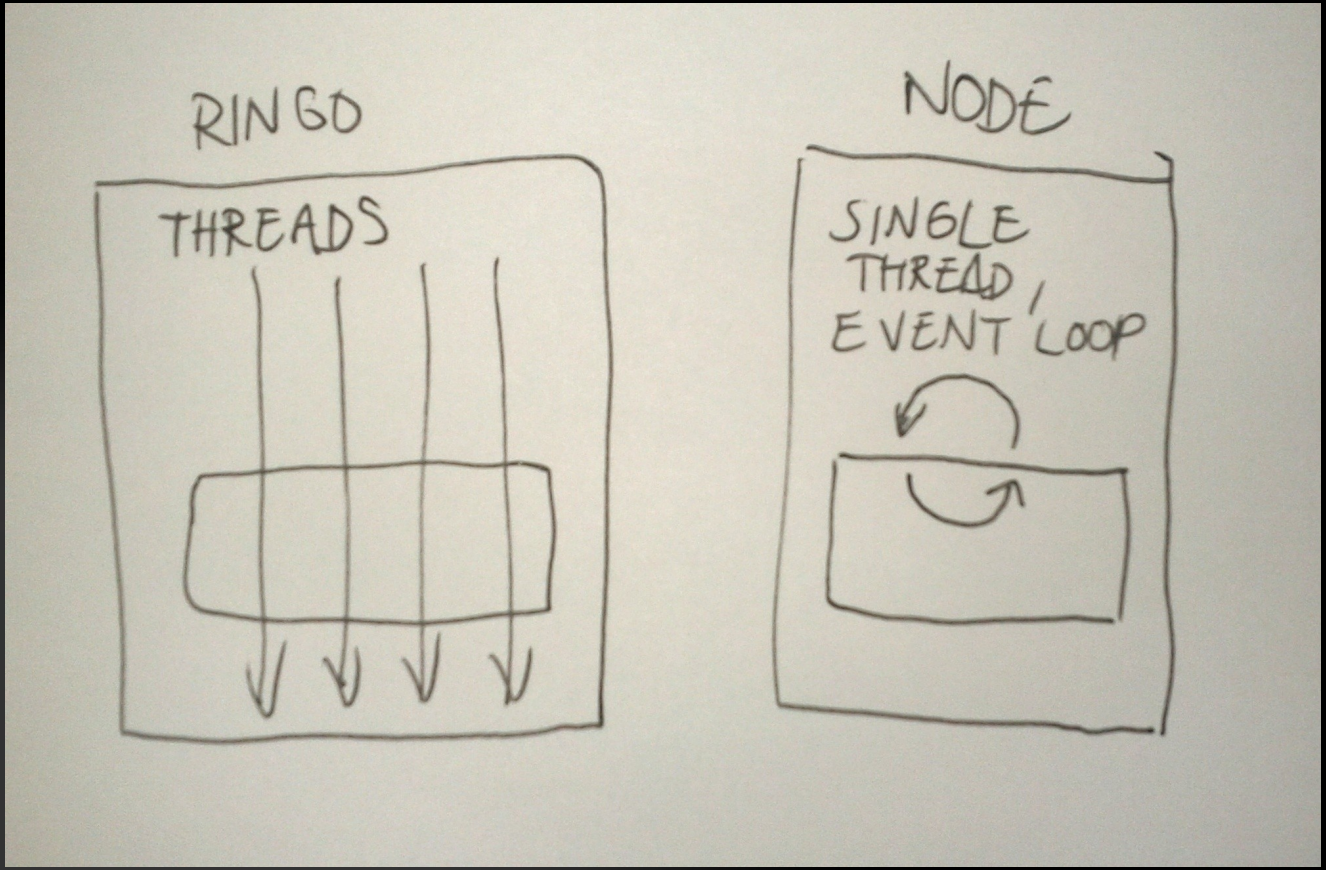
Helma (1998 – 2008)

RingoJS (2008 -)

Both using shared memory multi-threading (that was how you did it back then)

RingoJS moving to isolated workers in next version (will merge to github master soon)

Threading models



What's wrong with shared memory multi-threading?

It works most of the time

When it fails, it fails in weird and unpredictable ways

You can't see it in the code

Any other way beside the old way and a single-threaded event loop?

Want to allow blocking

Want to leverage JVM threads

There's a third way (with many names): Actors, workers, shared-nothing threads, lightweight processes, message passing...

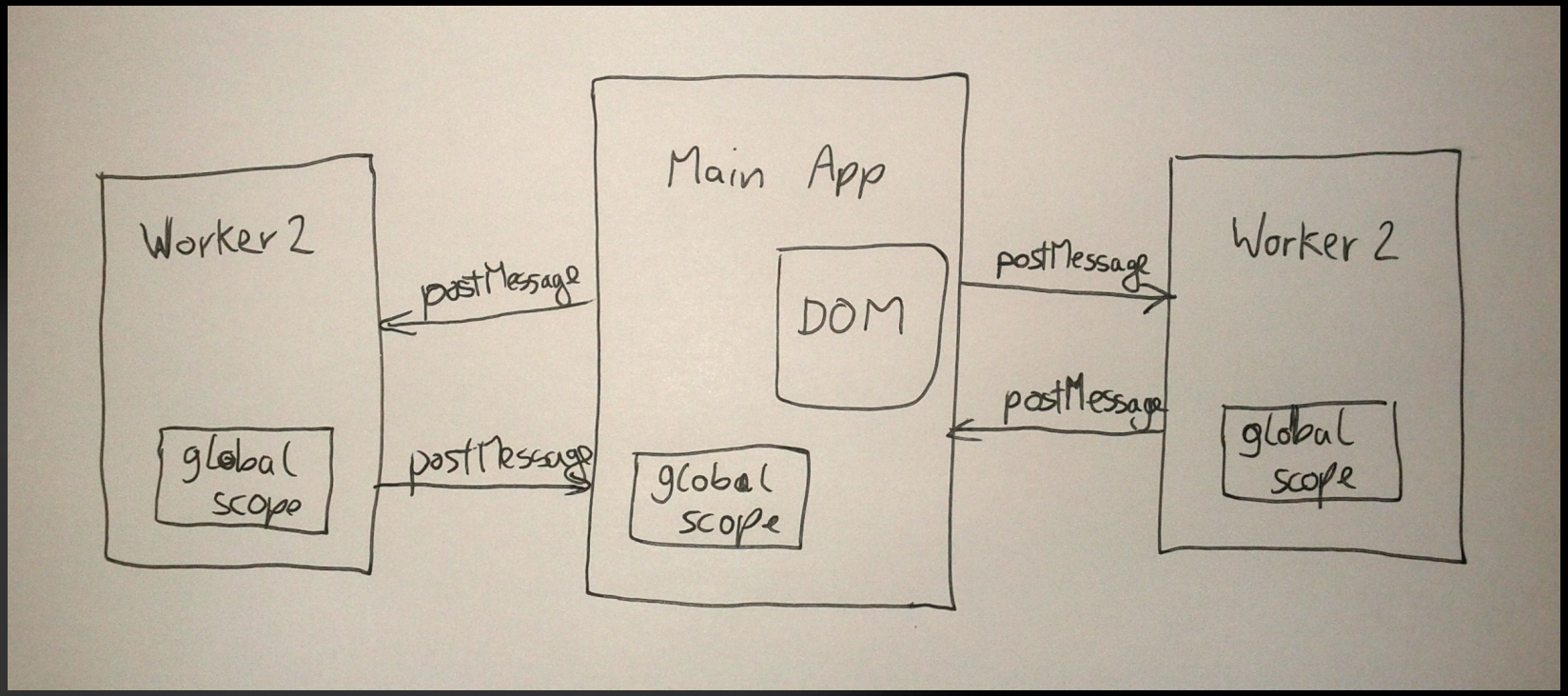
W3C Web Workers

Fully isolated JavaScript environments

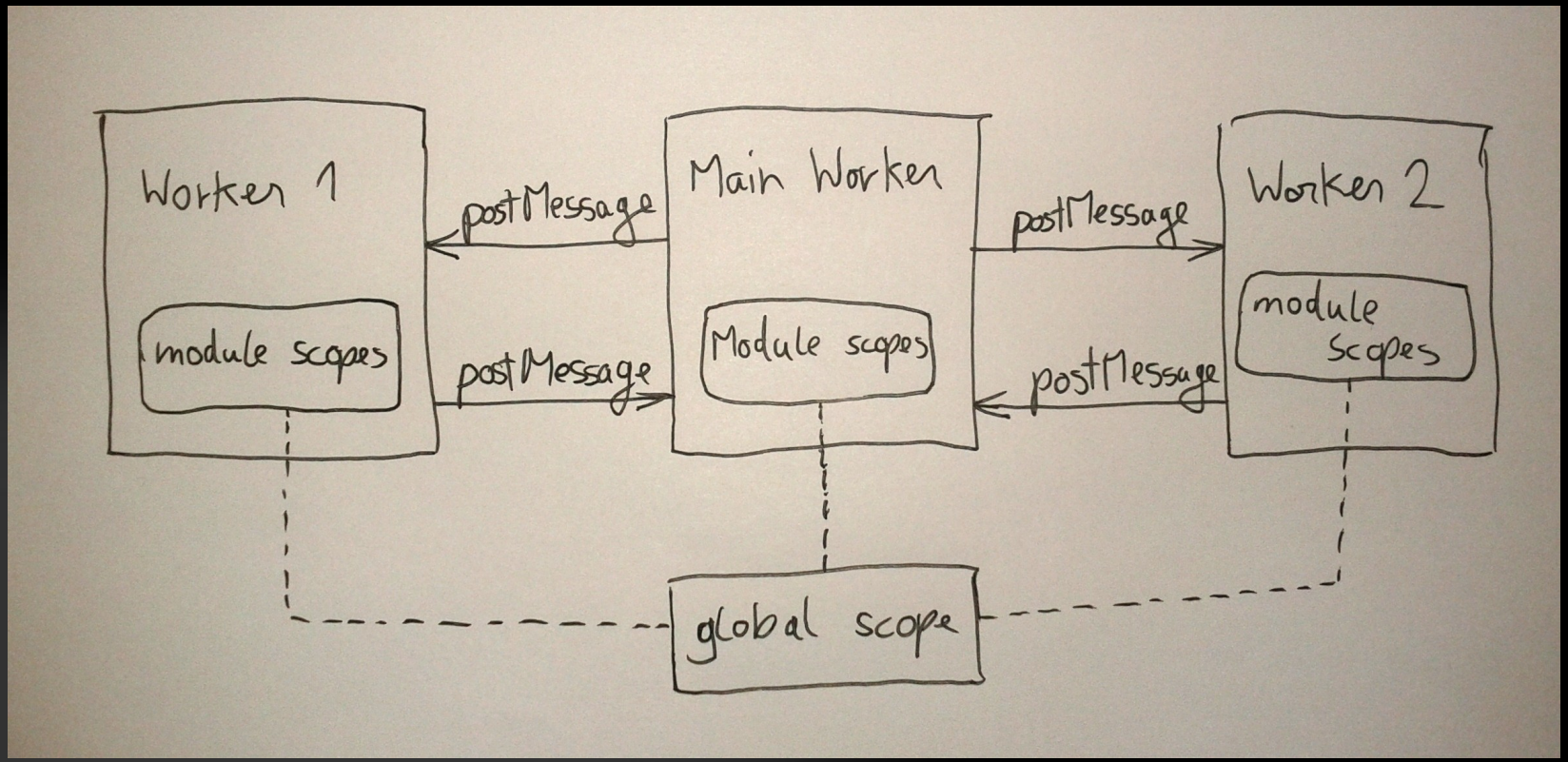
No access to DOM

Asynchronous message passing using JSON serialization

Web Workers in the browser



Workers in Ringo



Shared global scope in Ringo

- greatly reduces worker instantiation overhead
- allows us *not* to JSON-serialize parameters between workers (unless we want to)
- provides a simple way to opt-in to shared data:

```
global.foo = "bar";
```

Demo

The most inefficient Fibonacci implementation ever, spawns 1000s of workers

fib(17): 6 seconds in Ringo, about 50 seconds in Firefox 7

Each Ringo worker has its own single-threaded event loop

Works with setTimout(), setInterval(), and things built on top of that such as promises

Does not work yet with events triggered from external sources (e.g. Java libraries)

Ringo still supports blocking

Synchronous I/O

```
var bytes = file.read();
```

Semaphores (introduced with workers)

```
semaphore.signal();
semaphore.tryWait(2000, 3);
```

Mixing blocking and event loops

Short answer: Don't!

Long answer: It depends on how long you block and how much liveness you need. But better to run your blocking code in a separate worker!

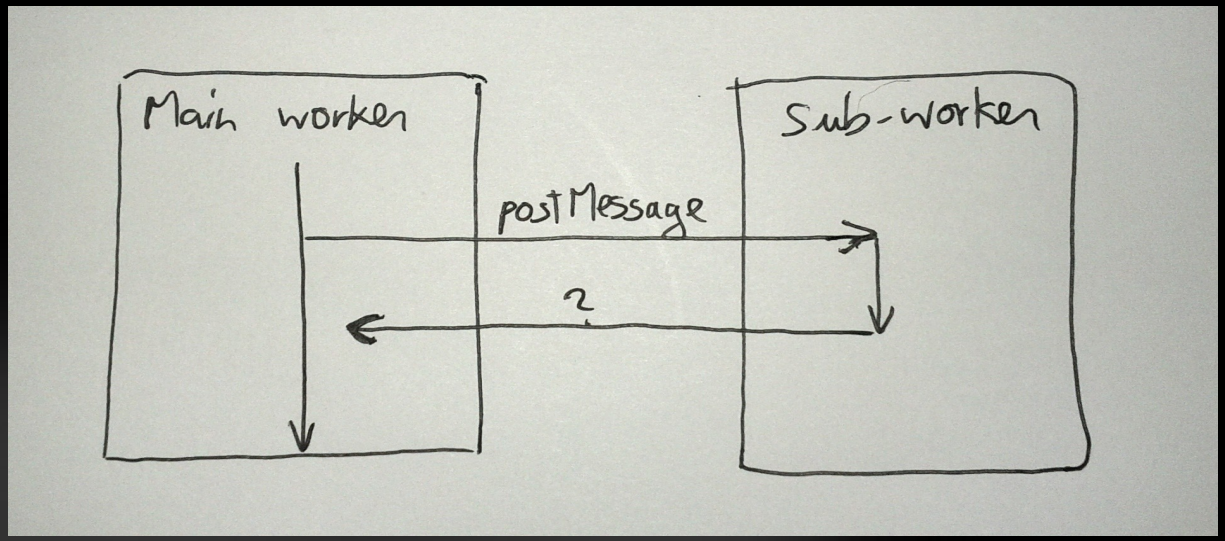
Suggested usage patterns

Separate long running, synchronous code from short running, asynchronous code

(that's basically what workers were intvented for)

(but in Ringo it goes both ways)

Problem: how to receive callbacks from workers in long-running, sync code?



Remember, callbacks won't fire until the code launching the worker has terminated, which may be never.

Semaphores to the rescue

```
var worker = new Worker("foo.js");
var result, error, semaphore = new Semaphore();
worker.onmessage = function(e) {
   result = e.data;
   semaphore.signal();
};
worker.onerror = function(e) {
   error = e.data;
   semaphore.signal();
};
worker.postMessage("foo", true);
semaphore.wait();
```

Unsolved problems

Pooling and reusing "dirty" workers

- Different worker pools for different purposes?
- Better support for leasing/releasing workers

Make sure events from Java libraries run within the event loop

- Event-loop aware event dispatcher

Questions?

@hannesw
http://ringojs.org