Simpler Coordination of JavaScript Web Workers

Marco Krauweel, Sung-Shik Jongmans

Simpler Coordination of JavaScript Web Workers

Marco Krauweel[1] and Sung-Shik T.Q. Jongmans[1,2]

[1] Department of Computer Science, Open University of the Netherlands
[2] Department of Computing, Imperial College London

Abstract. JavaScript is a popular sequential language for implementing Web applications. To enable concurrent execution of JavaScript code, modern JavaScript engines support the Web Workers API. Using this API, developers can spawn concurrent background workers from a distinguished main worker. These workers, which run on the same machine (e.g., to exploit multicore), interact via message-passing.
The Web Workers API is relatively low-level, which makes implementing coordination protocols among background workers laborious and error-prone. To simplify this, we propose to hide the Web Workers API behind a coordination language that provides higher-level constructs. Importantly, developers already use JavaScript together with domain-specific languages HTML (for markup/structure) and CSS (for style/design); another domain-specific language (for coordination) seamlessly fits this practice. Using the coordination language Reo, we demonstrate the advantages and feasibility of this approach by example. We also present the necessary tool support (compiler; runtime library and API; front-end).

1 Introduction

Context. *JavaScript* [7] is a sequential language originally invented to add dynamic behavior to static Web pages. Together with domain-specific languages HTML and CSS, JavaScript has become one of the most widely used languages for implementing Web application clients. In recent years, JavaScript was popularized also for writing server-side code and mobile apps. This makes JavaScript a key language in contemporary software engineering.

Single pieces of JavaScript code are executed sequentially. To concurrently execute multiple pieces of JavaScript code on the same machine (e.g., to exploit a multicore processor), modern JavaScript engines support also the *Web Workers API* [9]. This API "allows Web application authors to spawn background workers running scripts in parallel to their main page", which "allows for thread-like operation with message-passing as the coordination mechanism."

Problem. The Web Workers API is relatively low-level.

First, the Web Workers API provides no constructs for background workers to send messages directly to each other; background workers can send messages only to a distinguished main worker ("hierarchical communication" [21]). Second, the Web Workers API provides no constructs for a background worker to correlate

messages sent to, and later received from, the main worker; workers cannot reply to messages. Third, the Web Workers API provides no constructs for a background worker to pause processing of the current message in anticipation of receiving the next message; background workers cannot block.

Thus, sending messages end-to-end, replying to messages, and awaiting messages, can be implemented only in terms of lower-level constructs. This is laborious and error-prone. For instance, it is already nontrivial to implement a background worker that simply needs to send a message to, and await a reply from, another background worker as part of the same task.

Contribution. To simplify implementing coordination protocols using the low-level Web Workers API, we propose to provide developers additional higher-level constructs. Such constructs have, in fact, been under development already for decades, in *coordination languages*. We therefore propose to hide the Web Workers API behind a coordination language. An observation particularly relevant to this approach is that developers already use JavaScript together with domain-specific languages HTML (for markup/structure) and CSS (for style/design); another domain-specific language (for coordination) seamlessly fits this practice.

In Sect. 2, we present preliminaries on JavaScript and the Web Workers API. In Sect. 3, we illustrate the limitations of the Web Workers API. In Sect. 4, we describe the existing coordination language *Reo* [10,11]. In Sect. 5, we demonstrate how the use of Reo to hide the Web Workers API alleviates its limitations, in theory. In Sect. 6, we present tool support (compiler; runtime library and API; front-end) for developers to implement and run coordination protocols using Reo. In Sect. 7, we demonstrate the feasibility of our approach, in practice. Section 8 concludes this paper, including related work. To our knowledge, this is the first paper that presents high-level constructs to simplify implementing coordination protocols among background workers in JavaScript and the Web Workers API.

2 JavaScript and the Web Workers API

- JavaScript provides first-class functions and the usual control constructs [7].
- The Web Workers API provides constructs for spawning *background workers* from a *main worker* and constructs for sending/receiving messages [9].

Together, JavaScript and the Web Workers API constitute an actor language in the style of *active objects*, in the taxonomy of De Koster et al. [13].

Every worker starts by performing some initial work (e.g., initializing variables). It then checks if, in the meanwhile, *events* have occurred that require processing. Examples include *message events* (i.e., receipt of a message) and *timeout events* (i.e., passage of time). If so, the worker executes *event listeners* in response, until all events have been processed; otherwise, it suspends until the next event occurs. Execution of event listeners is nonblocking and nonpreemptive: event listeners run to completion in one go, uninterleaved with other event listeners of the same worker.

```
1 var ping = new Worker("background.js");
2 var pong = new Worker("background.js");
3
4 ping.onmessage = function(e) {
5   pong.postMessage(e.data); }
6 pong.onmessage = function(e) {
7   ping.postMessage(e.data); }
8
9 ping.postMessage("pong");
```

Fig. 1: main-pp.js

```
1 onmessage = function(e) {
2   if (e.data == "ping")
3     setTimeout(function() {
4       postMessage("pong"); }, 500);
5
6   if (e.data == "pong")
7     setTimeout(function() {
8       postMessage("ping"); }, 1000);
9 };
```

Fig. 2: background.js

A worker has an event handler for every type of event that it responds to. For instance, every background worker has a message event listener to process messages received from the main worker; as background workers cannot receive messages directly from each other, no other message event listeners are necessary. Conversely, the main worker receives messages from all background workers, and it may need to respond differently to each of them; the main worker, therefore, may have multiple message event listeners. Although many types of events exist, only message and timeout events matter in this paper, w.l.o.g.

A worker's cycle of awaiting and processing of events is called its *event loop*, and it is repeated indefinitely; pending event listeners are stored in a private FIFO queue. Furthermore, every worker has its own private heap. No memory is shared; message-passsing is the only means of communication.

Example 1. Figures 1–2 show an example program. This program defines two background workers, called Ping and Pong, who iteratively send `"ping"` and `"pong"` to each other. Syntactically, a message event handler is a function, where parameter `e` represents a message event to be processed; the message is accessed through `e.data`. `postMessage(m)` sends message `m`. `setTimeout(f,t)` generates a timeout after `t` milliseconds; subsequently, function `f` is called.

Figure 2 defines Ping (and Pong). Initially, Ping only sets his message event handler (line 1–9); subsequently, he suspends. Whenever Ping receives a `"pong"` message from the main worker, he resumes by sending back a `"ping"` message, after a 1000 milliseconds delay (lines 6–8); subsequently, he suspends again.

Figure 1 defines the main worker. Initially, the main worker spawns Ping and Pong (lines 1–2), then it sets its message event handlers (lines 4–7), then it sends an initial `"pong"` message to Ping (line 9); subsequently, it suspends. Whenever the main worker receives a message from Ping or Pong, it resumes by forwarding that message to Pong or Ping; subsequently, it suspends again.

Note that the main worker must call `postMessage` on a particular background worker, to indicate the receiver. In contrast, background workers do not need to call `postMessage` on the main worker; as background workers can send messages only to the main worker, no confusion can arise about who is the receiver. □

```
1  var i = 0, n = 9;
2  var alice = new Worker("alice.js");
3  var bob   = new Worker("bob.js");
4  var carol = new Worker("carol.js");
5
6  alice.onmessage = function(e) {
7    bob.postMessage(e.data); };
8
9  bob.onmessage = function(e) {
10   var m = e.data;
11   if (i < n) {
12     bob.postMessage({sn: m.sn}); // ack
13     carol.postMessage(m);
14     i++;
15   } else {
16     setTimeout(function() {
17       bob.onmessage(e); }, 0);
18 } };
19
20 carol.onmessage = function(e) {
21   i--; };
```

Fig. 3: main-abc.js

```
1  var nextSn = 0;
2  while (true) {
3    var initVal = ...; // do init work
4    var m = {sn: nextSn++, val: initVal};
5    postMessage(m);
6  }
```

Fig. 4: alice.js

```
1  var pending = [];
2
3  function fstHalf(sn, initVal) {
4    var finalVal = ...; // process initVal
5    var m = {sn: sn, val: finalVal};
6    postMessage(m);
7    pending.push(sn);
8  } // suspend until ack ...
9
10 function sndHalf(sn) { // ... resume!
11   pending.splice(pending.indexOf(sn), 1);
12   ...; // do more work
13 }
14
15 onmessage = function(e) {
16   var m = e.data;
17   if (pending.indexOf(m.sn) == -1)
18     fstHalf(m.sn, m.val);
19   else
20     sndHalf(m.sn);
21 };
```

Fig. 5: bob.js

```
1  onmessage = function(e) {
2    postMessage({}); // ack
3    var m = e.data;
4    var finalVal = m.val;
5    ...; // do final work
6  };
```

Fig. 6: carol.js

3 Limitations and Issues

Example 2. Figures 3–6 show an example program to illustrate the limitations of the Web Workers API, stated in Sect. 1. This program defines three background workers, called Alice, Bob, and Carol, operating in a pipeline: Alice indefinitely produces initial values and sends them to Bob, Bob processes initial values to final values and sends them to Carol, and Carol consumes final values.

Figures 4, 6 define Alice and Carol; they are straightforward. Alice's messages record a serial number (used for bookkeeping by Bob) and an initial value.

Figure 5 defines Bob. Initially, Bob sets an empty list of pending serial numbers (line 1) and sets his message event handler (lines 15–21); subsequently he suspends. Whenever Bob receives a message, he resumes. If Bob does not recognize the serial number in the message (line 17), he starts a new processing cycle for that serial number (line 18); otherwise, he continues a previous processing cycle (line 20). In the former case, Bob processes the initial value to a final value (line 4), then he sends a message to the main worker (lines 5–6), who forwards it to Carol, then he registers the serial number (line 7); subsequently, he suspends again. In the latter case, Bob unregisters the serial number (line 11) and finalizes his processing cycle (line 12); subsequently, he suspends again.

Figure 3 defines the main worker. The main worker unconditionally forwards messages received from Alice to Bob (line 7). As such, Alice communicates to Bob via an unbounded buffer. In contrast, the main worker forwards messages received from Bob to Carol (line 13) only if the number of buffered messages `i` is smaller than a bound `n` (line 11). As such, Bob communicates with Carol via an `n`-capacity buffer (e.g., to avoid excessive memory usage if Carol is slower than Bob, and many large message from Bob would otherwise need to be buffered). To prevent loss of messages, the main worker acknowledges a serial number `sn` to Bob only once the corresponding message can be forwarded (line 12); Bob awaits this acknowledgment before beginning the second half of its processing cycle for `sn`. As such, the `n`-capacity buffer is blocking. If the main worker cannot forward a message yet, it schedules an immediate retry (lines 16–17).[3]

This example program illustrates the limitations of the Web Workers API, stated in Sect. 1. First, it illustrates that background workers cannot directly send to, and receive from, each other. In particular, background workers cannot obtain references to other background workers. Second, this example program illustrates that background workers cannot directly understand replies to messages. In particular, whenever Bob receives from the main worker a reply (acknowledgment) to an earlier message, the context in which he sent that message is gone; to Bob, the reply might just as well be a new forwarded message from Alice. This is why messages need to be explicitly tagged with a serial number and why Bob needs to do extra bookkeeping. Third, this example program illustrates that background workers cannot straightforwardly block. In particular, after Bob sends a message to the main worker, he needs to await an acknowledgment, but the only means of waiting is through suspension and resumption (event loop). As a result, Bob's processing cycle is unnaturally split into two functions. □

The coordination protocol between Bob and Carol in the previous example is among the simplest: asynchronous communication through a blocking FIFO buffer of bounded capacity. Yet, due to limitations of the Web Workers API, the implementation of this simple coordination protocol is not that simple at all: as higher-level constructs are missing (sending message end-to-end, replying to messages, awaiting messages), key characteristics can be expressed only indirectly (through forwarding, through serial numbers, through split functions). This makes development laborious and error-prone.

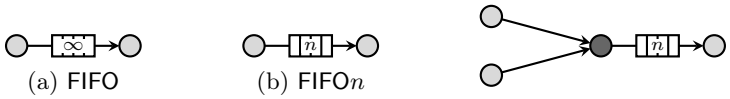
Another issue is that "computation code" is entangled with "coordination code". This lack of separation and modularity between computation and coordination complicates development. For instance, *changing* the implementation of a coordination protocol typically requires global understanding of, and modifications to, larger parts of a program; it cannot be localized, because implementations of coordination protocols are scattered across multiple workers. Moreover, *reusing* the implementation of a coordination protocol in other programs typically requires so much effort that it is practically infeasible.

[3] This is an inefficient implementation, but we aim for simplicity here.

```
1  ...
2  var bob1 = new Worker("bob.js");
3  var bob2 = new Worker("bob.js");
4  ...
5  var pendingBobs = [bob1, bob2];
6
7  alice.onmessage = function(e) {
8    if (pendingBobs.length > 0) {
9      var bob = pendingBobs.shift();
10     bob.postMessage(e.data);
11   } else {
12     setTimeout(function(e) {
13       alice.onmessage(e); }, 0);
14 } };
15
16 ...
```

```
17 bob1.onmessage = function(e) {
18   mFromBob(e.data, bob1); };
19 bob2.onmessage = function(e) {
20   mFromBob(e.data, bob2); };
21
22 function mFromBob(m, bob) {
23   if (m == {}) {
24     pendingBobs.push(bob);
25   } else if (i < n) {
26     bob.postMessage({sn: m.sn}); // ack
27     carol.postMessage(m);
28     i++;
29   } else {
30     setTimeout(function() {
31       mFromBob(m, bob); }, 0);
32 } }
```

Fig. 7: main-abbc.js



(a) FIFO (b) FIFOn (c) MergerFIFOn

Fig. 8: Example connectors

Example 3. Suppose that we want to instantiate two Bobs instead of only one (e.g., because Alice and Carol are twice as fast as Bob). Effectively, then, we need to generalize the coordination protocol between Alice and Bob to multiple readers, and the coordination protocol between Bob and Carol to multiple writers. Figure 7 shows modifications to the main worker to achieve this.[3] Additionally, we *crucially* need to insert a `postMessage({})` call between lines 12 and 13 in Fig. 5, through which a Bob informs the main worker that he is ready for the next message. The latter illustrates that modifications cannot be localized to one specific piece of code; global understanding and modifications are necessary. □

For more complex coordination protocols (e.g., tighter synchronization, many control states), the impact of these limitations and issues becomes only more serious. This is why we propose to provide developers additional higher-level constructs that hide the low-level Web Workers API. Next, we present an existing coordination language that achieves this aim.

4 Reo

Reo [10,11] is a graphical language for compositional construction of coordination protocols, manifested as *circuits*. Briefly, a circuit is a graph-like structure consisting of typed *channels* (decorated edges), through which values flow, and *nodes* (vertices), on which channel ends coincide. Figure 8 shows examples.

Every channel has a type, graphically indicated by (the absence of) a decoration. The type of a channel determines how values flow through it—its behavior.

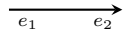
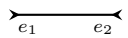
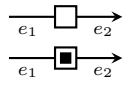
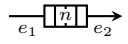
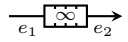
Name	*Syntax*	*Semantics*
sync		Synchronously accepts a value through e_1 and offers that value through e_2.
syncdrain		Synchronously accepts and loses values through e_1 and e_2.
fifo1		Asynchronously [accepts a value v through e_1 and stores v in a buffer], then [offers v through e_2 and clears the buffer]. A token means that a buffer is initially filled with a random v.
fifo$n$		Version of fifo1 with an n-capacity buffer
fifo		Version of fifo1 with an unbounded buffer

Table 1: Common channels

Every channel has two ends, each of which is either a *source end* or a *sink end*. A source end accepts values into its channel, while a *sink end* offers values out of its channel. A channel in Reo has either two source ends, or two sink ends, or a source and a sink end. Table 1 shows common channels types. Users of Reo may extend this set by defining their own channels (modeled in some formalism [18]).

The behavior of a node depends on its coincident channel ends.

- A node with only coincident source ends is a *source node*. A source node is linked to an *output port* of a worker. By performing a `put(`v`)` operation on an output port, a worker attempts to offer value v to the linked source node. As soon as each of the coincident source ends of the source node is ready to accept v, it synchronously *replicates* v into all of them, and the `put` returns; until then, the pending `put` blocks the worker.
- A node with only coincident sink ends is a *sink node*. A sink node is linked to an *input port* of a worker. By performing a `get` operation on an input port, a worker attempts to accept a value from the linked sink node. As soon as at least one of the coincident sink ends of the sink node is ready to offer a value, the sink node *nondeterministically selects* a value v from one of them, and the `get` returns v; until then, the pending `get` blocks the worker.
- The source and sink nodes of a circuit are its *boundary nodes* (light-gray color). A node on which both source and sink ends coincide is a *mixed node* (dark-gray color). Mixed nodes combine the behavior of source and sink nodes: synchronously, a mixed node nondeterministically selects a coincident sink end and replicates its value into all coincident source ends. Nodes cannot temporarily store, generate, or lose values.

Before a value can flow through a circuit, its channels and nodes must first reach consensus about their local behavior to ensure consistent global behavior. For instance, a node should not locally (decide to) replicate a value into the coincident source end of a fifo channel if the buffer of that channel is full.

Example 4. The circuit in Fig. 8a implements the coordination protocol between Alice and Bob in Ex. 2; the circuit in Fig. 8b implements the coordination protocol between Bob and Carol in Ex. 2. This shows that Reo is expressive enough to define both nonblocking coordination protocols (e.g., between Alice and Bob) and blocking ones (e.g., between Bob and Carol); ultimately, the programmer decides what kind of communication is needed. The circuit in Fig. 8c implements the coordination protocol between the two Bobs and Carol in Ex. 3.

The circuit in Fig. 8c works as follows. For a `put(`v`)` operation performed on the top-left source node to return, this node must replicate v into the coincident source end of the sync channel. This is possible only if the sync channel accepts v, which depends on whether it can synchronously offer v through its sink end to the connected mixed node. This is possible only if the mixed node accepts v, which depends both on its nondeterministic selection and on whether it can synchronously replicate v into the coincident source end of the fifo$n$ channel. The latter is possible only if the fifo$n$ channel accepts v, which depends on whether its buffer is not full. Thus, only if the mixed node makes the "right" nondeterministic selection and the buffer is not full, only then flows v *synchronously* from the top-left source node into the buffer (and `put(`v`)` returns).

Once the buffer is full, no value can flow from either of the two source nodes into the buffer. The only thing that can happen at this point is the flow of a value out of the buffer to the sink node. Subsequently, because the buffer is not full anymore, a value can flow from one of the two source nodes into the buffer. □

For Java developers to use Reo to implement and run coordination protocols, a *Reo-to-Java compiler*, a *Reo@Java runtime library*, and a *Reo@Java API* exist [16,17]. The idea is that developers implement workers as Java classes and coordination protocols as Reo circuits. Then, the Reo-to-Java compiler computes the semantics of the Reo circuits as state machines (more precisely, as *constraint automata* [12,16]), and it generates Java classes for their runtime simulation.

At runtime, every hand-written worker object, and every compiler-generated state machine object, runs in its own Java thread. Through the Reo@Java API, every worker object has access to (data structures for) ports, on which it can perform `put` and `get` operations. State machine objects monitor those ports: whenever a worker object performs a `put` or `get` on one of its ports, a designated state machine object checks whether this operation enables a transition out of its current state; every transition represents a synchronous flow of values among ports. If so, the state machine object makes the transition, distributes values accordingly among participating ports, and completes all `put` and `get` operations involved; if not, the state machine object does nothing and awaits the next `put` or `get`. In accordance with the blocking nature of `put` and `get`, the execution of worker objects is suspended so long as their `put`s and `get`s are pending. The Reo@Java runtime library provides an implementation of the Reo@Java API.

```
function* alice(out) {
  while (true) {
    var initVal = ...; // do init work
    yield out.put(initVal);
} }

function* carol(in) {
  while (true) {
    var finVal = yield in.get();
    ...; // do final work
} }
```

```
// (continued)

function* bob(in, out) {
  while (true) {
    var initVal = yield in.get();
    var finVal = ...; // process initVal
    yield out.put(finVal);
    ...; // do more work
} }
```

Fig. 9: workers-reo.js

Fig. 10: main-abc.reo

5 Hiding the Web Workers API behind Reo – *In Theory*

Reo enables developers to implement coordination protocols in terms of domain-specific abstractions and higher-level constructs. As a coordination language that hides the low-level Web Workers API, use of Reo has two main implications.

The first implication is the adoption of a syntactic separation between computation and coordination: developers should implement workers in JavaScript, while they should implement coordination protocols purely in Reo. This separation leads to modularization (of workers and coordination protocols), which has well-known advantages [23], including improved changeability and reusability. The second implication is that programmers should no longer implement communications with nonblocking `postMessage` from the existing Web Workers API, but with blocking `put` and `get` from our new *Reo@JS API*. If necessary, as demonstrated in Ex. 4, `put` and `get` can effectively be made nonblocking by connecting them to a circuit with fifo buffers. Of course, nothing prevents programmers from implementing additional "covert communications" among background workers in an indirect way (e.g., using message channels or sockets), but this is against the philosophy of our approach and therefore discouraged.

Example 5. Figure 9 shows implementations of Alice, Bob, and Carol in the example program from Ex. 2, using the Reo@JS API instead of the Web Workers API. Alice, Bob, and Carol are now implemented as functions, with output ports and input ports as parameters. As `put` and `get` are blocking, especially function `bob` is significantly simpler than the code in Fig. 5 (fewer details to worry about).

Figure 10 shows implementations of the coordination protocols between Alice and Bob, and between Bob and Carol. This diagram also defines which workers the program consists of, by which functions those workers are defined, and how workers' ports are linked to boundary nodes. Using higher-level constructs

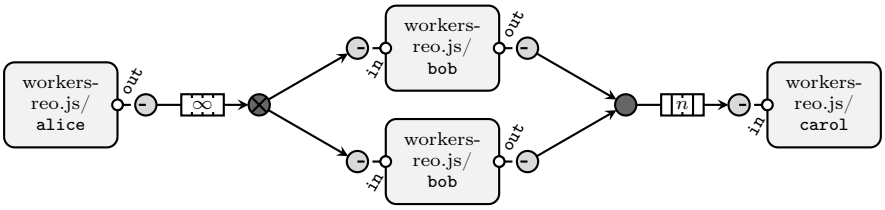


Fig. 11: main-abbc.reo

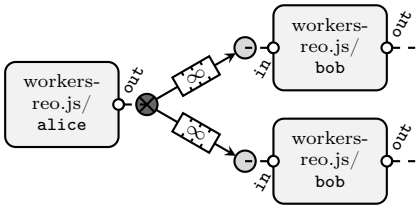


Fig. 12: main-abbc2.reo (fragment)

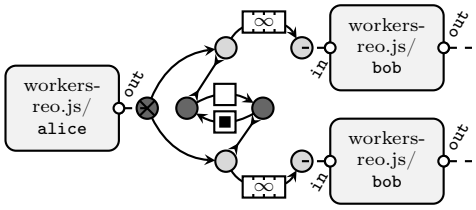


Fig. 13: main-abbc3.reo (fragment)

to express coordination, as in Reo, developers are relieved from the burden of working with low-level constructs provided by the Web Workers API. □

Example 6. Figure 11 shows implementations of the coordination protocols between Alice and the two Bobs, and between the two Bobs and Carol, in the example program from Ex. 3. The "crossed" mixed node replicates a value into only one of its coincident source ends (instead of all), selected nondeterministically.[4] Notably, Alice, the two Bobs, and Carol are defined by *exactly the same functions* as those in Fig. 9. Such reusability is one of the advantages of separating computation from coordination. Figure 12 shows an implementation of a different coordination protocol between Alice and the two Bobs, where every Bob now has its own unbounded buffer (instead of them sharing one). Making these modifications at this level of abstraction is simple; at the level of abstraction of the Web Workers API, it is not. Figure 13 shows an implementation of yet a different coordination protocol between Alice and the two Bobs, where values from Alice are divided evenly over the two buffers (not guaranteed with Fig. 12).

The circuits in Fig. 11 are built from (the two simple) circuits in Fig. 10. This shows that not only can coordination protocols be reused, but in fact, they can be further *composed* into more complex ones. In plain JavaScript and the Web Workers API, reusing and composing existing implementations of coordination protocols is practically infeasible. □

Examples 5–6 show how the limitations and issues of the Web Workers API (Sect. 3) are alleviated when a coordination language, such as Reo, is used. Hiding the Web Workers API behind Reo makes both workers and coordination

[4] This node is, in fact, syntactic sugar for a complexer circuit (i.e., it can be composed).

protocols simpler to implement, easier to change, and easier to reuse, *in theory*. To actually reap these advantages *in practice*, developers need tool support. We present such tools in the next section, to show that our approach is also feasible.

6 Tool Support

Overview. For developers to use Reo to implement and run coordination protocols, they need a compiler and a runtime library. The "easy part" was developing our new *Reo-to-JS compiler*, which works much in the same way as the existing Reo-to-Java compiler. The "hard part" was developing our new *Reo@JS runtime library*, which differs significantly from its Java counterpart.

Compiler. Most of the internals of the new Reo-to-JS compiler are reused from the existing Reo-to-Java compiler (see Sect. 4). In particular, computation and optimization of state machines for Reo circuits is independent of the target language and works in exactly the same way for JavaScript as for Java.

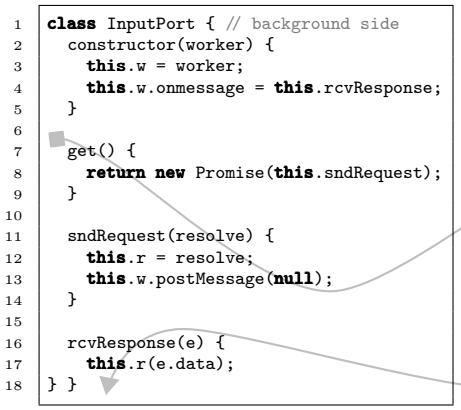
The JavaScript code generated by the Reo-to-JS compiler is also conceptually similar to the Java code generated by the Reo-to-Java compiler: in the same event-driven way as explained in Sect. 4, the generated JavaScript code simulates a computed state machine. This state machine is executed by the main worker.

The Reo-to-JS compiler also generates "wrappers" around the functions that are linked to the boundary nodes of the circuit; these wrappers are executed in separate background workers, using the Web Workers API. All necessary initialization code is also generated. Thus, the only code that developers need to write by hand are the functions that link to the boundary nodes (e.g., Fig. 9).

Runtime library. The new Reo@JS runtime library contains auxiliary code that simplifies and reduces the amount of code that needs to be generated. Most of this code is similar to the code in the existing Reo@Java runtime library. The two runtime libraries significantly differ, however, in their implementation of ports, and particularly, the implementations of `put` and `get`. This is because Java differs from JavaScript and the Web Workers API in two significant ways: Java supports shared-memory concurrency and blocking constructs, which JavaScript and the Web Workers API do not support. Shared memory in Java enables straightforwardly linking worker objects to state machine objects by sharing the same port objects between them; this is not possible in JavaScript. Blocking operations in Java make implementing the blocking semantics of `put` and `get` straightforward; this is also not possible in JavaScript.

We first explain our solution to the unshared memory complication. The idea is to use *two* sets of output ports and input ports: one set for background workers, and another set for the main worker. Figures 14–15 show simplified[5]

[5] Compared to Figs. 14–15, the actual implementation also supports multiple ports per workers, and it is protected against workers that perform the next `get` (or `put`) before the previous one is completed. Details appear elsewhere [20].



```
class InputPort { // background side
  constructor(worker) {
    this.w = worker;
    this.w.onmessage = this.rcvResponse;
  }

  get() {
    return new Promise(this.sndRequest);
  }

  sndRequest(resolve) {
    this.r = resolve;
    this.w.postMessage(null);
  }

  rcvResponse(e) {
    this.r(e.data);
} }
```

Fig. 14: input-port.js (simplified)

```
class InputPortProxy { // main side
  constructor(worker) {
    this.w = worker;
    this.w.onmessage = this.rcvRequest;
  }

  rcvRequest(e) {
    this.get().then(this.sndResponse);
  }

  get() {
    // ...
    // (similar to Java version)
  }

  sndResponse(val) {
    this.w.postMessage(val);
} }
```

Fig. 15: input-port-proxy.js (simpl.)

versions of the code for input ports; the code for output ports is similar. Fig. 14 shows class `InputPort`, instances of which are used by background workers. At the "background side", whenever `get` is called, `sndRequest` is subsequently called (through a *promise*; see below). `sndRequest` essentially relays the `get` call to the proxy of the main worker, by sending a null message. Then, at the "main side", this null message is processed by `rcvRequest`, which calls the real `get` that is monitored by the state machine. The resulting value is eventually sent back to the background worker, where it is processed by `rcvResponse`.

The nonblocking operations complication is trickier to solve, because it seems impossible to implement blocking operations in JavaScript without exposing the programmer to some of the required logic. The argument is as follows. JavaScript has no blocking constructs, so we need to implement them ourselves. One option is *busy-waiting*, where an event listener that needs to block repeatedly checks the value of a variable until it is changed by another event listener of the same worker (e.g., in response to a message receipt). But, as event listeners are never preempted, they are never interleaved. This entails a causal loop that causes a busy-waiting worker to busy-wait forever: the second event listener is not executed until the busy-waiting is over, which does not happen until the variable is updated, which is what the second event listener should do. The only viable alternative to implement blocking, then, seems suspending the worker in its event loop, by ending the current event listener. Importantly, just before suspending, bookkeeping is required to register a *continuation*; otherwise, the worker does not know how to proceed later on. It seems inevitable to place the burden of bookkeeping for continuations on the developer. The main challenge is to minimize this burden, encapsulating it in the Reo@JS runtime library wherever possible.

First, we use *promises*. A `Promise` object represents the eventual result of an asynchronous computation, which may not have finished yet when the `Promise` object is created. Upon calling the constructor of a `Promise` object, an *executor (function)* is passed that defines the asynchronous computation to be performed;

```
1  function runMultiplier(
2    inputPort1, inputPort2, outputPort) {
3
4    var lhs, rhs, product;
5
6    inputPort1.get().then(
7      function(val) {
8        lhs = val;
9        return inputPort2.get();
10   }).then(
11     function(val) {
12       rhs = val;
13       product = lhs * rhs;
14       return outputPort.put(product);
15   }).then(
16     function() {
17       console.log("result: " + product);
18   });
19 }
```

Fig. 16: Promises, explicitly

```
1  function* runMultiplier(
2    inputPort1, inputPort2, outputPort) {
3
4    var lhs = yield inputPort1.get();
5    var rhs = yield inputPort2.get();
6    var product = lhs * rhs;
7    yield outputPort.put(product);
8    console.log("result: " + product);
9  }
```

Fig. 17: Promises, implicitly

```
1  function loop(generator, val) {
2    var r = generator.next(val);
3    if (!r.done) { // gener. not yet done
4     Promise.resolve(r.value).then(
5       function(nextVal) {
6         loop(generator, nextVal);
7  } );} }
8
9  loop(runMultiplier(/* ports */), null);
```

Fig. 18: Looping on promises

inside the constructor, the executor is called with a *resolve (function)* as parameter. The resolve is called once the asynchronous computation finishes. Initially, the resolve is empty. To "fill" the resolve, `then` can be called on the created `Promise` object with a *callback (function)* as parameter. If the resolve has already run at that point, the callback is immediately run; otherwise, the resolve is filled with a call to the callback (which runs as soon as the asynchronous computation finishes). Technically, `then` returns another `Promise` object, making it possible to express a sequence of seemingly blocking computations by successively calling `then`. Such a chain of `then` calls nests promises in promises, which asynchronously—interrupted in the event loop—resolve one after the other.

Example 7. As shown in Fig. 14, `get` returns a new `Promise` object; the same holds for `put`. Figure 16 shows what a worker looks like if developers were to express computations directly with promises. Note that the callbacks for the two `get` operations have a formal parameter `val` (which contains, conceptually, the value offered by the circuit); the actual parameter is passed on line 17, Fig. 14 (at which point `this.r` refers to the callback). □

The previous example shows that using promises to implement blocking, still places a heavy burden on developers. To simplify this, we use *generators*. A generator is a function that can be exited and re-entered: whenever a `yield` is encountered during the execution of a generator, the context is saved, and the generator is exited (optionally yielding a result). When the generator is later re-entered, its saved context is restored, and it proceeds from where it left off.

We require that workers are implemented as generators instead of as normal functions (indicated with an asterix), and that `put` and `get` are always used together with `yield`. Subsequently, we can apply *promise-based asynchronous*

task running [28], where every background worker calls its generator, waits until a promise from `put` or `get` is yielded, provides this promise a callback in which the generator is re-entered, and suspends; the provided callback ensures that this process repeats itself until the generator is done. Figure 18 shows this approach in code. In this way, every background worker effectively runs a loop (but with continuous interruptions through suspension and resumption) in which all code is executed in callbacks, from one `put`/`get` to the next `put`/`get`. For instance, "unrolling" the code in Fig. 17 results in the code in Fig. 16.

Using promises and generators, we largely relieve developers from the burden of bookkeeping for continuations, with concise code as a result (e.g., Fig. 9).

Front-end. We hooked our Reo-to-JS compiler into *PrDK* [16,17]. PrDK consists of plugins for Eclipse, including a graphical editor for Reo circuits that allows developers to draw Reo diagrams as the ones in Figs. 10–13, using a drag-and-drop interface. From this editor, the Reo-to-JS compiler can directly be run to generate and package all JavaScript code, for client-side execution in browsers or for server-side execution in *Node.js* (using the *tiny-worker* library [6]).

7 Hiding the Web Workers API behind Reo – *In Practice*

Examples 5–6 (Sect. 5) showed that, in theory, the limitations and issues of the Web Workers API (Sect. 3) are alleviated when a coordination language, such as Reo, is used. Our example programs were still abstract, though. In this section, complementary, we report on a concrete example program that we implemented using the tools presented in Sect. 6.

Example 8. To avoid bias, we took an existing program [1] (not ours) that uses the Web Workers API. This program performs a nontrivial numerical computation, where n background workers cooperatively compute $a^b \ mod \ c$. This calculation is also performed, for instance, in RSA decryption, where a is the encrypted message, and where b and c constitute an agent's private key.

In the original program, first, the main worker sends a message (a, b_i, c)—a "work package"—to every background worker $1 \leq i \leq n$, such that $\sum b_i = b$. Subsequently, every background worker i computes $a^{b_i} \ mod \ c$ and sends back the result. Finally, the main worker aggregates these results into the final outcome. Communication between the main worker and the background workers thus follows a typical master–slaves pattern.

Using our tools, we adapted the original program to use the Reo@JS API, effectively by replacing all `postMessage` calls with `put` and `get` calls on ports. We also relieved the main worker from its original tasks of dividing the work and aggregating the results, and placed these responsibilities with two new background workers. As a result, the background workers now exactly fit Alice, the Bobs, and Carol in Ex. 3: Alice divides the work, the Bobs perform the work and compute the results, and Carol aggregates the results. By instantiating Alice, the Bobs,

and Carol in this way, Fig. 11 is directly applicable in this case study, including all its previously explained advantages (Exs. 5–6).

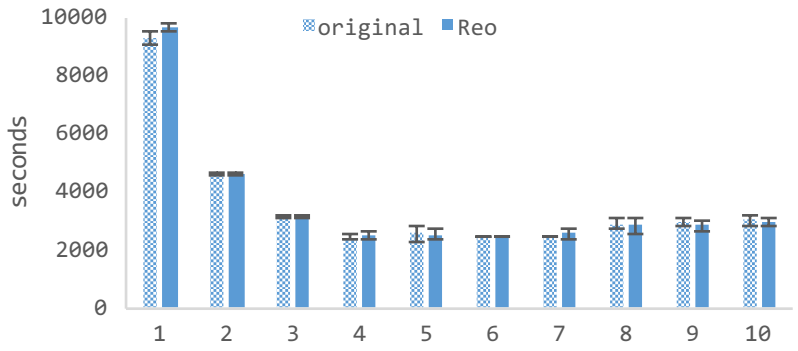
As a result of these changes, the coordination code that needed to be written manually was reduced from 145 lines to 46 lines (reduction of nearly 70%). Moreover, by implementing the coordination protocol in a separate module as a Reo circuit, it became amenable to reuse; essentially, the changes to the original program turned a specific implementation of master–slaves coordination into a reusable generic one. Thus, the effort of designing the circuit (less than an hour by a Reo expert) need not be remade in the future. □

Example 9. Suppose that the original program "accidentally" (e.g., as the result of a programming mistake) lets the main worker send all work packages to the same background worker, so losing concurrency. As the implementation of the coordination protocol is not an explicit module in the original program, there is no obvious place to enforce that work packages should be evenly distributed among background workers.

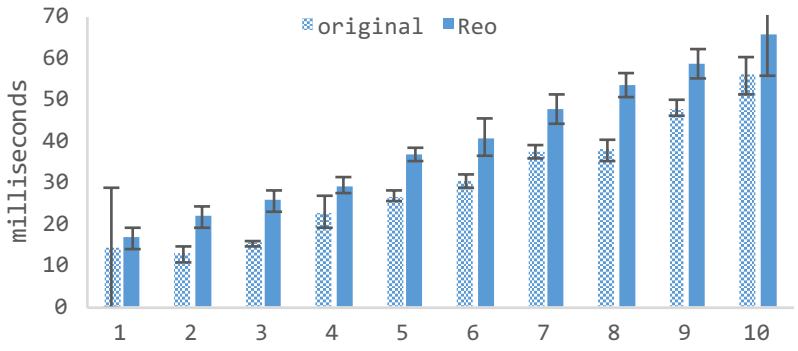
In contrast, using Reo, we can encode such constraints in the circuit, and the compiler-generated code automatically enforces them. Figure 13 already showed such a circuit for two Bobs, which can be generalized to n. Using this circuit, if a faulty Bob performs multiple `get` operations, the circuit still ensures that the other Bobs receive a work package before the faulty Bob receives its second one, so preserving concurrency. Thus, new constraints (e.g., to improve robustness) can directly be added in our approach, due to improved changeability. □

Example 10. As a first indication of performance, we conducted the following experiment. We took a synchronous variant of the circuit in Fig. 13 and a synchronous variant of the circuit between the two Bobs and Carol in Fig. 11, then we used the Reo-to-JS compiler to generate code, and finally we ran the resulting programs in Firefox on a machine with four hardware threads, processed by two physical cores. We experimented with synchronous circuits, because we wanted to test a worst-case scenario (asynchrony is cheaper than synchrony). The computation to be performed was always $2^{1024 \cdot 10^6} \bmod 97777$. We repeated this for $1 \leq n \leq 10$ Bobs, and we averaged our timing measurements over ten runs per n; we did the same for the original program to make a comparison.

Figure 19a shows the begin-to-end execution times (error bars indicate standard deviation). The figure shows that scalability is quite well, so long as there is hardware parallelism to harness (up to $n = 4$). Evaluating the effectiveness of the parallelization is actually not our main concern, though; we are primarily interested in the performance of the compiler-generated protocol implementation *relative to the hand-written one in the original program*. Figure 19b is, thus, more interesting: it shows the execution times of *only* the coordination overhead



(a) Begin-to-end



(b) Coordination overhead

Fig. 19: Experimental results

(measured by commenting out the computations of the workers).[6] This figure shows that the compiler-generated code is on average 20% slower. Given that we have not seriously optimized our compiler and runtime library yet, we consider this a promising result: it suggests that we can have the advantages in Sect. 5 without prohibitive performance costs. □

The examples in Sect. 5 and in this section, combined with the tools in Sect. 6, provide first evidence that hiding the Web Workers API behind a coordination language, such as Reo, is advantageous in theory *and* feasible in practice.

[6] Ideally, coordination overhead is completely independent of workload. In practice, however, this is not always so. The memory footprint of a workload can, for instance, affect the size of coordination overhead [16]. Generally, the larger a workload, the smaller the ratio $\frac{\text{coordination}}{\text{computation}}$, the less impact coordination overhead has on performance, and the less important minimizing such overhead becomes; at that point, other software qualities (changeability, reusability, etc.) may take precedence.

8 Conclusion

Related work. Beside the Web Workers API, other proposals to incorporate actor-based concurrency in JavaScript have been made. For instance, Stivan et al. [26] ported the JVM-based implementation of the Akka framework to JavaScript, called *Akka.js*. One of the differences between the Web Workers API and Akka.js is that Akka.js allows actors on different JavaScript engines to communicate with each other. For Myter et al. [21], supporting both parallelism and distribution was a key design consideration in developing the *Spiders.js* framework. Spiders.js places particular emphasis on ease of programming, but not on coordination protocols. Welc et al. [27] proposed *generic workers* to support both parallelism and distribution in terms of an API very similar to the Web Workers API. These approaches are complementary to ours: in this paper, we simplify implementing coordination protocols among background workers running on the same machine, but a generalization to distributed actors would be interesting.

There has also been work on incorporating data parallelism in JavaScript, including the *River Trail* framework by Herhut et al. [14] and the *WebCL* initiative [8], which constitutes a JavaScript binding to OpenCL. In an emperical study of twelve Web applications, Radoi et al. [25] found "a surprisingly large quantity of compute-intensive loops of which many were latently parallel."

Outside academia, several libraries have been developed to simplify programming with the Web Workers API, including *q-connection* [5], *parallel.js* [4], *Hamsters.js* [2], and *operative* [3], but without emphasizing coordination protocols.

Technical differences aside, the main conceptual difference between all this related work and our proposed approach is that we emphasize the importance of coordination protocols as explicit programming artifacts, thereby enforcing syntactic separation of coordination and computation. Such a separation makes it easier to change and reuse coordination protocols.

This work. We showed that implementing coordination protocols among background workers is difficult, because of limitations and issues with the low-level Web Workers API (Sects. 2–3). To make this simpler, we proposed to hide the Web Workers API behind a coordination language that provides higher-level constructs. Using Reo (Sect. 4), we demonstrated the advantages and feasibility of our approach by example (Sects. 5, 7), and we presented the necessary tool support in terms of a compiler, a runtime library and API, and a front-end (Sect. 6). As Web application developers have a tradition of separating concerns (HTML, CSS, JavaScript), Web applications may constitute a fertile new application domain for existing coordination languages.

Future work. We argued, by example, that use of a coordination language makes implementing coordination protocols among background workers simpler (i.e., it has particular advantages over directly using the Web Workers API). Complementary proof for this claim would consist of a set of successful real projects; we therefore aim to empirically evaluate the merits of our proposed

approach, and improve our tools in the process. Optimizing the Reo-to-JS compiler and Reo@JS runtime library is also high on our list, including a study of impact on performance, which we barely touched upon in this paper.

An important aspect of the previously mentioned future work is studying the scalability of our proposed approach, both in terms of usability and performance. We expect modularity (separation of coordination and computation) to play an enabling role in dealing with complex systems, but it requires a new way of working from programmers; the consequences of this are unclear and should be better studied. Also, tools (notably, the compiler) need to ensure performance scalability as coordination protocols grow larger (i.e., involve more participants), which generally is a nontrivial technical challenge [19].

We chose to use Reo in this work, because it is strongly rooted in separation of computation and coordination. Reo, however, also has its limitations. For instance, run-time parametrization in the number of workers is not yet possible. It is therefore interesting to see if a JavaScript variant of, for instance, *Pabble* [22] (based on *multiparty session types* [15]) is more suitable.

A special case of a background worker is one that only calls an asynchronous API (e.g., the Geolocation API) and processes the result in a callback. In another experiment with our tools [20] (omitted from this paper to save space), we encountered several such distinguished background workers and already added support for them in our tool (the tool automatically generates code to make asynchronous API calls from our framework; the programmer does not need to write such boilerplate code her/himself). Further research is necessary, however, to study to what extent coordination languages can also be used to orchestrate asynchronous API calls as a solution to "callback hells", and whether there are advantages compared to existing approaches; see Philips et al. [24].

References

1. Javascript Web Workers Test. `http://pmav.eu/stuff/javascript-webworkers/`
2. [library]: Hamsters.js. `https://github.com/austinksmith/Hamsters.js`
3. [library]: operative. `https://github.com/padolsey/operative`
4. [library]: parallel.js. `https://github.com/parallel-js/parallel.js`
5. [library]: q-connection. `https://github.com/kriskowal/q-connection`
6. [library]: tiny-worker. `https://github.com/avoidwork/tiny-worker`
7. [standard] Ecma International: ECMA-262. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`
8. [standard] Khronos Group: WebCL. `https://www.khronos.org/webcl`
9. [standard] W3C: Web Workers. `https://www.w3.org/TR/workers`
10. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comp. Sci. 14(3), 329–366 (2004)
11. Arbab, F.: Puff, The Magic Protocol. In: Formal Modeling: Actors, Open Systems, Biological Systems, LNCS, vol. 7000, pp. 169–206. Springer (2011)
12. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. 61(2), 75–113 (2006)
13. De Koster, J., Van Cutsem, T., De Meuter, W.: 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In: Proceedings of AGERE 2016. pp. 31–40. ACM (2016)

14. Herhut, S., Hudson, R., Shpeisman, T., Sreeram, J.: River Trail: A Path to Parallelism in JavaScript. In: Proceedings of OOPSLA 2013. pp. 729–744. ACM (2013)
15. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. ACM SIGPLAN Notices (Proceedings of POPL 2008) 43(1), 273–284 (2008)
16. Jongmans, S.S.: Automata-Theoretic Protocol Programming. Ph.D. thesis, Leiden University (2016)
17. Jongmans, S.S.: PrDK: Protocol Programming with Automata. In: Proceedings of TACAS 2016, LNCS, vol. 9636, pp. 547–552. Springer (2016)
18. Jongmans, S.S., Arbab, F.: Overview of Thirty Semantic Formalisms for Reo. Scientific Annals of Computer Science 22(1), 201–251 (2012)
19. Jongmans, S.S., Arbab, F.: Can High Throughput Atone for High Latency in Compiler-Generated Protocol Code? In: Fundamentals of Software Engineering (Proceedings of FSEN 2015), LNCS, vol. 9392, pp. 238–258. Springer (2015)
20. Krauweel, M.: Concurrent and asynchronous JavaScript programming using Reo. Master's thesis, Open University of the Netherlands (2017)
21. Myter, F., Scholliers, C., De Meuter, W.: Many Spiders Make a Better Web: A Unified Web-Based Actor Framework. In: Proceedings of AGERE 2016. pp. 51–60. ACM (2016)
22. Ng, N., Yoshida, N.: Pabble: parameterised Scribble. Service Oriented Computing and Applications 9(3-4) (2015)
23. Parnas, D.: On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM 15(12), 1053–1058 (1972)
24. Philips, L., De Koster, J., De Meuter, W., De Roover, C.: Dependence-driven delimited CPS transformation for JavaScript. In: Proceedings of GPCE 2016. pp. 59–69. ACM (2016)
25. Radoi, C., Herhut, S., Sreeram, J., Dig, D.: Are web applications ready for parallelism? In: Proceedings of PPoPP 2015. ACM (2015)
26. Stivan, G., Peruffo, A., Haller, P.: Akka. js: Towards a portable actor runtime environment. In: Proceedings of AGERE! 2015. pp. 57–64. ACM (2015)
27. Welc, A., Hudson, R., Shpeisman, T., Adl-Tabatabai, A.R.: Generic Workers: Towards Unified Distributed and Parallel JavaScript Programming Model. In: Proceedings of PSI EtA 2010. ACM (2010)
28. Zakas, N.: Promises and Asynchronous Programming. In: Understanding ECMAScript 6, chap. 11, pp. 213–241. No Starch Press, 1st edn. (2016)