Communicating Process Architectures 2016
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2016

Communicating Generators in JavaScript

Kurt MICALLEF and Kevin VELLA [1]
Department of Computer Science, University of Malta, Msida MSD2080, Malta

Abstract. This paper outlines the design, performance, and use of an application programming interface and library for concurrent programming with CSP in JavaScript. The implementation harnesses ECMAScript 6 Generators to provide co-operative scheduling and channel communication within a single JavaScript engine. External channels lie atop WebSockets, amongst other web technologies, to enable multicore and distributed execution across standard web browsers and Node.js servers. Low-level benchmarks indicate that scheduling and messaging performance is within expectations for this dynamic and diverse execution environment. Sample code snippets highlight the applicability of CSP to contemporary web development in hiding the location of computation and state through the channel abstraction. The "call-back hell" scenario common to many JavaScript applications is alleviated by using channels instead of callbacks, and the possibility of performing parallel and scientific computing is explored with promising results. Finally, the limitations of the present design are discussed, and possible enhancements such as the dynamic migration of state and code are considered.

Keywords. CSP, JavaScript, concurrency, distributed systems, programming tools, world-wide web

Introduction

Love it or hate it, JavaScript [1] dominates today's Web with a presence that extends from desktop and mobile browsers all the way to server-side back-ends. It is a multi-paradigm language featuring first-class functions, and is untyped, dynamic, and usually interpreted. Applications written in JavaScript are event-driven, and tend to feature several levels of nested function call-backs.

JavaScript's ubiquity makes it an attractive target for a CSP-like [2] API and library. Its expressive flexibility combined with recently introduced language features such as generators also make it a convenient target. Moreover, recent and emerging web technologies such as WebSockets [3] and Web Workers [4] make it possible to provide a distributed implementation running across standard browsers and servers.

This paper outlines the design, performance, and use of an application programming interface and library for concurrent programming with CSP in JavaScript ES6 and beyond. Sections 1 and 2 explain how to harness ECMAScript 6 Generators to provide co-operative scheduling and channel communication within a standard JavaScript engine. Subsequently, external channels that lie atop web technologies to enable multicore and distributed execution across unmodified web browsers and Node.js servers are described in Section 3.

[1]Corresponding Author: *Kevin Vella, Department of Computer Science, University of Malta, Msida MSD2080, Malta*; E-mail: `kevin.vella@um.edu.mt`

Low-level benchmarks are presented in Section 4 to characterise the scheduling and messaging overheads introduced by the library. Section 5 highlights the applicability of CSP to contemporary web development in JavaScript. Finally, related work is surveyed, the limitations of the present design are explored, and possible enhancements such as the dynamic migration of state and code are considered in Sections 6 and 7.

1. JavaScript, Concurrency and Web Standards

Current browsers support most of the recent ECMAScript 6 [1] language specification. This iteration of JavaScript introduces generators: functions which can be partially evaluated (in an imperative sense). A generator is capable of suspending its own execution and preserving its state, and its resumption can be triggered in an event-driven fashion.

A generator function is defined by `function* (){...}`, which returns a `Generator` object on initialisation. Its execution is suspended through the `yield` expression, and resumed by invoking the `next()` method on its `Generator` object from its external scope.

The `next()` method continues executing the generator function's code until a `yield` (or any of the other exit expressions such as `return` or `throw`) is encountered. In addition, generators are also capable of sending and receiving data [5], as shown in Listing 1.

```
1  var generatorFunction = function* (){
2    var ret = yield 1;
3    return ret;
4  };
5  var generator = generatorFunction();
6  var x = generator.next().value; // x = 1
7  var y = generator.next(2).value; // y = 2
```

Listing 1. Basic execution of a generator.

Lines 1 to 4 define the `GeneratorFunction` `generatorFunction`. Line 5 instantiates a `Generator` object from the `GeneratorFunction` (`generatorFunction`), which is subsequently invoked with `next()` in line 6. The generator yields in line 2 and passes the value 1 to the caller, which then retrieves it using `.value` and stores it in `x` (line 6 again). Line 7 resumes the generator function from line 2 and passes it the value 2. The generator function stores it in `ret`, and returns it to the caller on exiting in line 3. Back in line 7, the value 2 is received and stored in `y`.

Moreover, the `yield*` expression offers a way to route the aforementioned calls to another generator [5,6]. When invoking a `Generator` object which delegates to another generator, the delegated generator is evaluated, as shown in Listing 2.

```
1  var delegate = function* (){
2    yield 1;
3  };
4  var generator = (function* (){
5    yield* delegate();
6  }());
7  var x = generator.next().value; // x = 1
```

Listing 2. Delegating execution to another generator.

1.1. Distributed JavaScript

As the "language of the web", JavaScript is designed to interact with external entities ranging from DOM elements to remote servers and clients. JavaScript is not only interpreted by most of today's browsers, as server-side JavaScript has become a major trend owing to server run-times such as Node.js [7], which is built on Google Chrome's V8 JavaScript engine. Nowadays it is commonplace to develop front and back-ends for web applications using the same programming language.

A relatively recent technology, WebSockets [3] enable web servers and browsers to communicate using event-driven APIs. Various JavaScript libraries have been implemented on top of this functionality, for instance socket.io [8]. WebRTC [9] is another emerging technology that allows peer-to-peer (browser-to-browser) communication, although an intermediate server is still required for connection establishment. Also of relevance, socket.io-p2p [10] is a library implementing socket.io-like APIs over WebRTC.

JavaScript relies on event-driven function callbacks rather than threading to handle logical concurrency. Multicore processors are harnessed through workers, each operating in its own isolated address space and communicating by passing messages across address space boundaries. Amongst the worker implementations available for JavaScript are Web Workers [4] on browsers, and Cluster [11] workers on Node.js.

The proliferation of JavaScript engines across the Internet poses the challenge of writing reusable code which can execute and move seamlessly in this environment. The opportunity thus arises to envisage, develop, and experiment with software tools to simplify the task at hand.

2. CSP-style Concurrency in JavaScript

The architecture of the system was strongly influenced by the T9000 processor design as described in Networks, Routers, and Transputers [12]. A simple yet efficient software library was produced by adopting this approach.

2.1. The Dispatcher

Generators are unable to send and receive data amongst themselves or synchronise without additional support; they require an intermediate function to handle the routing of data as well as their rescheduling. This necessitates that all such generators, which we have termed *co-generators*, are contained within a special dispatcher function's scope.

The dispatcher is created on initialisation of a CSP environment through `csp()`, the single top-level scope for all CSP code running within a JavaScript engine. This is equivalent to the top-level `PAR` in an occam [13] program. The API accepts multiple co-generators as individual comma-separated parameters or as an array, and schedules them for concurrent execution by the dispatcher. These top-level co-generators can create and schedule additional co-generators for execution by the dispatcher as explained in Section 2.2. In turn the `csp()` function itself returns once all the co-generators under its management have no more processing to do on the event-loop, either because they are waiting for some event, or because they have terminated. Listing 3 illustrates `csp()` in use.

```
1  csp.csp(
2    function* (){ /* ... */ },
3    // ...
4    function* (){ /* ... */ }
5  );
```

Listing 3. Executing several co-generators within a CSP environment.

Fairness is not guaranteed across multiple CSP environments running in the same JavaScript engine, as each dispatcher is not aware of co-generators in the other CSP environments.

On initialisation the dispatcher schedules these co-generators on the FIFO run queue for starvation-free execution. Each queue node is an object containing the `Generator` object itself and an associated value. Initially `undefined`, this value is updated as the co-generator executes, and is passed to the co-generator whenever it resumes execution through `next()`.

When an executing co-generator pauses with a `yield`, the returned value is examined by the dispatcher, which invoked the co-generator in the first place. The returned value is typically an object created by the API function called after `yield`, wrapping data relevant to the co-generator's state and the API itself. Based on this data, the dispatcher performs the corresponding action and switches to the next co-generator on the run queue, as portrayed in Figure 1, until the run queue is empty. For this reason, the following rules must be observed when calling API functions (other than CSP environment and channel creation) to avoid undefined behaviour.

1. An API function must be prefixed with `yield`. This is necessary because API functions wrap any essential data in an `Object` which is to be interpreted by the dispatcher. Not doing so will corrupt the state of the CSP environment, as the co-generator would not pause and the API does not function as supposed to.
2. CSP constructs are to be called from the scope of a generator contained by a CSP environment, `csp()`, directly or indirectly. Should they be invoked outside of a `csp()` call a dispatcher would not be available to schedule execution and communication.

Since `yield` on its own does not call a specific API function, nothing is returned to the dispatcher, and it simply enqueues the paused co-generator at the end of the queue. Judicious use of `yield` is conducive to fairness in this co-operative scheduling regime.

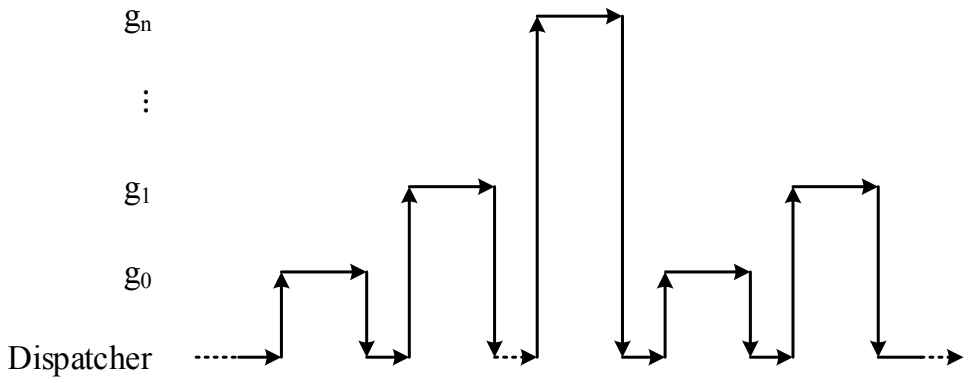

Figure 1. Execution flow of co-generators.

2.2. Co-Generator Creation and Termination

The CSP environment created with `csp()` initialises and concurrently executes the top-level co-generators listed in its arguments. The API functions described here create new co-generators, making use of the enclosing CSP environment's FIFO dispatcher and run queue to provide starvation-free execution in a deadlock-free CSP environment whose constituent co-generators yield regularly.

The `fork()` API function schedules the passed co-generators on the run queue. The created co-generators do not resynchronise with their parent co-generator. On the other hand, the `co()` API function, demonstrated in Listing 4, is similar to occam's `PAR` construct. The caller is not executed again after calling `co()` until all co-generators in that concurrent section are executed to completion. This is achieved by creating a barrier across the given co-generators, and resuming the caller once these co-generators have all joined the barrier on completion of their execution. If the dispatcher queue is empty, and there are still barriers waiting on co-generators, this is an indication that deadlock has occurred. An `Error` is thrown by the dispatcher in such cases, allowing exceptional situations to be handled appropriately within JavaScript but outside the CSP environment.

```
1  csp.csp(function* (){
2    yield csp.co(
3      function* (){ /* ... */ },
4      // ...
5      function* (){ /* ... */ }
6    );
7  });
```

Listing 4. Dynamically creating co-generators.

2.3. Channels

Apart from being an object which co-generators reference to communicate data, the `Channel` also serves as a temporary home for suspended co-generators. When either a `send()` or `recv()` is called on a `Channel`, the API function returns the `Channel` object and any data stored on it to the dispatcher, which in turn deals with them appropriately.

At its inception, a `Channel` is empty. The first co-generator to perform `send()` or `recv()` on the channel is suspended, waiting for a second co-generator to communicate on the channel. The dispatcher hangs the suspended co-generator on the `Channel` in the meantime. When the second co-generator communicates on the `Channel`, the corresponding API function (`send()` or `recv()`) returns the `Channel` object to the dispatcher. Upon examining the `Channel` object returned, the dispatcher discovers the suspended co-generator, and thus schedules both communicating co-generators at the back of the run queue while also passing the data to the receiving co-generator.

Currently, run-time checks are performed to ensure that no more than two co-generators use a particular channel to communicate: one sender and one receiver. In future this constraint may be relaxed to favour the convenience of any-to-any channels [14]. An example of channel communication is shown in Listing 5.

```
1  var channel = new csp.Channel();
2
3  csp.csp(function* (){
4    var x = yield channel.recv(); // x = 1
5  }, function* (){
6    yield channel.send(1);
7  });
```

Listing 5. Channel communication between co-generators.

2.4. Timeouts

Timeouts, as shown in Listing 6, provide similar functionality to occam's `TIMER`. The time elapsed in milliseconds since the creation of the entire CSP environment is obtained using `csp.clock()`, while `csp.timeout()` only reschedules the calling co-generator once the specified number of milliseconds have passed since the creation of the CSP environment. Alternatively, `csp.sleep()` reschedules the co-generator after the specified interval in milliseconds. No co-generator will resume execution before its timeout expires, and multiple waiting co-generators will resume execution in timeout order.

```
1  csp.csp(function* (){
2    // ...
3    yield csp.timeout(csp.clock() + 1000);
4    // continue after current time + 1 second
5  });
```

Listing 6. Using a timeout within a co-generator.

The specified timeout order is preserved by keeping waiting co-generators and their respective expiry times in a queue ordered by expiry times. The dispatcher polls the current time at every yield point, and returns any waiting co-generators with times in the past to the run queue in order of expiry. This maintains the expected scheduling order for waiting co-generators even if their timeouts have already expired, but not for co-generators whose timeouts expired before their relative `csp.timeout()` invocations.

If the dispatcher queue is empty but there are still unexpired timeouts, a JavaScript timeout is set with the time remaining to the nearest timeout. This callback function resumes the dispatcher, which proceeds to reschedule the co-generators whose timeouts have since expired.

2.5. Choice

The current implementation of choice supports channel input guards, timeout guards as well as boolean guards. Since each guard is associated with an action, the choice API function takes a list of pairs containing the guard and the guarded action. The choice API function wraps the guards in an object and returns it to the dispatcher, which then carries out the choosing procedure. Listing 7 shows the `csp.choice()` function in use.

```
1  var channel = new csp.Channel();
2
3  csp.csp(function* (){
4    yield csp.choice(
5      {
6        "recv": channel,
7        "action": function* (x) { /* ... */ }
8      },
9      {
10       "timeout": 1000,
11       "action": function* () { /* ... */ }
12     },
13     {
14       "boolean": true,
15       "action": function* () { /* ... */ }
16     }
17   );
18 });
```

Listing 7. Choice on a number of guards.

A choice consists of the following sequential phases: enabling, waiting, and disabling (start and end are not relevant here because there is no shared state between co-generators). The enabling co-generator determines whether a guard is ready, in which case a flag is set in the dispatcher. In the case of a channel guard, it is ready whenever a sending co-generator is waiting on the channel. A timeout guard is never ready instantaneously, and only the timeout expiring first is considered, as the others will never be ready before the first one. Lastly, a boolean guard is ready if and only if its condition evaluates to `true`.

When all guards are enabled, the dispatcher inspects whether the ready flag was set. If it was not set, the dispatcher is said to be in the waiting phase for that choice, and proceeds to execute the next co-generator on queue. On the other hand, if the flag was set to `true` then it knows that one of the guards is ready, so it begins the disabling procedure.

Disabling the guards can also happen as the choice is satisfied in time, for instance when a timeout expires, or when a sending co-generator arrives on an enabled channel. The guards are temporarily stored on the channel so that when a sending co-generator arrives on the channel, the dispatcher has a reference to the guards to disable. Guarded channels are disabled by clearing any data stored from the enabling stage, whilst a guarded timeout is disabled by clearing the timeout.

Whilst iterating over the guards to disable them, the first ready one is ultimately chosen. Although previously it was said that a choice is made non-deterministically, the order of how guards are disabled influences the way they are chosen. As a result, the `choice()` implementation is, by design, a prioritised one, since ready guards that are listed before are chosen over the latter ones.

If a channel guard is chosen, a generator helper is scheduled to actually perform the input. This generator receives the data from the channel satisfying the `choice()`, which is passed as an argument to the corresponding action, and called through the `yield*` expression. When the action is executed to completion, the helper reschedules the co-generator performing the `choice()`. `ALT` in occam permits prefixing *any* guard with a boolean condition, which must be `true` before the guard may be considered for selection; at present our API excludes this functionality. Moreover, `choice()` does not allow output guards.

3. External Channels

A number of external channel implementations are provided for communication between co-generators operating in separate JavaScript engine instances. This includes instances dispersed across distributed Node.js servers and browsers, as well as instances operating in disjoint address spaces on the same machine.

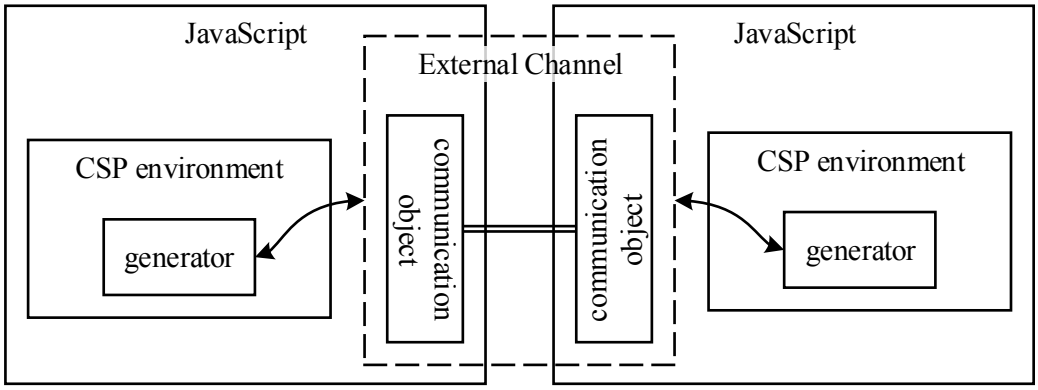

Figure 2. External Channel abstraction.

When an external channel is created and connected across two JavaScript engine instances, an object is obtained on each end which is then used as an end-point over which to communicate, as depicted in Figure 2. Once connected, external channels expose the usual `Channel` functionality so that no changes to the actual application code are required. A communicating co-generator is oblivious as to whether it is performing local or external communication.

3.1. Transport for External Channels

External channel implementations over JavaScript socket libraries such as socket.io [8] and WebSocket [3] are provided so that Node.js instances can communicate with browsers as well as with each other. This enables channel communication across physically distributed co-generators on the Internet. A socket object is obtained when a connection is established through socket.io or WebSocket, and this is passed as an argument to the channel creation API function. Exchanged messages are tagged with unique identifiers so as to multiplex `DistributedChannels` over the same socket. The distributed version of the local channel communication example which was presented in Listing 5 follows, in Listings 8 and 9.

```
1 http.createServer().listen(8000);
2 io.on("connection",function(socket) {
3   var channel = new csp.DistributedChannel(socket, "id1");
4
5   csp.csp(function* (){
6     var x = yield channel.recv();
7   });
8 });
```

Listing 8. Channel communication between distributed co-generators (server).

```
1 var socket = io.connect("http://serverhost:8000/");
2 var channel = new csp.DistributedChannel(socket, "id1");
3
4 csp.csp(function* (){
5   yield channel.send(1);
6 });
```

Listing 9. Channel communication between distributed co-generators (client).

Using the underlying socket library, the HTTP server in Listing 8 listens on port 8000 (lines 1 and 2) until the client in Listing 9 connects to it (line 1). On connection, both sides create their ends of a distributed channel over socket objects, with `id1` as the shared identifier for channel multiplexing over a socket. Execution continues as per Listing 5. It should also be possible to drop in socket objects obtained from socket.io-p2p [10], an API abstraction over WebRTC [9], but unexpected behaviour was observed in practice, possibly due to WebRTC not being finalised in current JavaScript engines.

Another external channel implementation enables communication between workers, which typically execute in separate operating system processes to harness multicore processors. Since browser and Node.js workers are incompatible [4,11], separate handlers were implemented. Nonetheless, only one `WorkerChannel` type is provided to handle both cases. Any overhead resulting from having to identify the worker type at run-time is restricted to the channel creation phase.

```
1 var worker = new Worker("worker.js");
2 var channel = new csp.WorkerChannel(worker);
3
4 csp.csp(function* (){
5   var x = yield channel.recv();
6 });
```

Listing 10. Channel communication between co-generators across workers (master).

```
1 var channel = new csp.WorkerChannel(self);
2
3 csp.csp(function* (){
4   yield channel.send(1);
5 });
```

Listing 11. Channel communication between co-generators across workers (worker – `worker.js`).

As before, worker creation is externalised, and the worker object obtained is passed as an argument when creating the `WorkerChannel`. An adaptation of Listings 8 and 9 for Web Workers is shown in Listings 10 and 11. A similar worker object is obtained when forking a Cluster Worker, and is used in the same manner.

3.2. External Channel Protocol

External channel communication involves first synchronizing the channel end-points and then performing the actual communication, as shown in Figure 3. This two-phase protocol is necessary since race conditions may result in sending data both to a local co-generator and a remote one. Thus, whenever synchronization occurs, the co-generators on the external channels

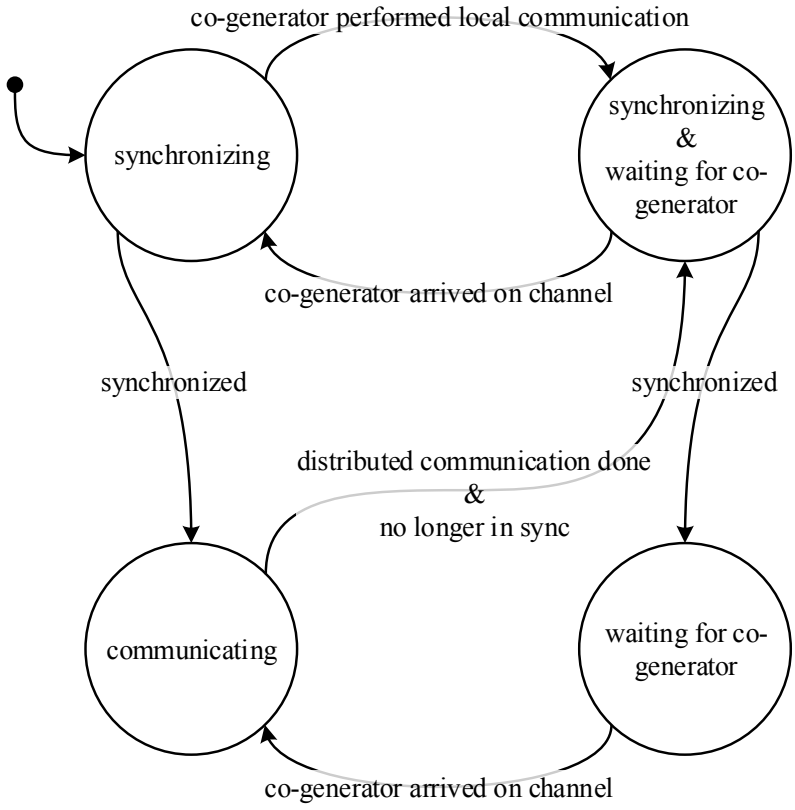

Figure 3. State diagram of an external channel.

are tied to the communication taking place. Once communication is complete, the external channel is in an idle state again.

The synchronisation phase involves sending an empty message, which could be avoided by conflating the first packet of data to be sent with the synchronization message. It should also be possible to optimistically kick off the message sending phase before synchronisation is complete, while passing on responsibility for dynamic buffer allocation to JavaScript itself.

3.3. Implementation Details

Very few alterations to the dispatcher were needed to accommodate external channels. Because the external channels implemented here use event listeners to receive data, it was necessary to pass the event listener to the dispatcher to initialise it on first use of the said channel. This is not done on the channel's creation, since a reference to the dispatcher is needed by the external channels without exposing the dispatcher to the user. Moreover, when performing a `choice()` on an external channels, the sending co-generator would not be stored on the channel as it would if it were a local channel. External channels are ready as soon as the synchronisation phase completes, hence this case needs to be considered when enabling a guarded external channel.

It should be noted that the design of the system allows new external channel types to be added without requiring internal modifications. Provided that the synchronize-then-communicate protocol is observed, no changes to the dispatcher are required.

Both the `DistributedChannel` and `WorkerChannel` implementations utilise a single listener to wait for messages independently of `recv()` calls on the said channel. Messages are sent over the corresponding object asynchronously. Hence, whilst the receiver is handling a message, the JavaScript engine hosting the sender is not stalled waiting for the receiver's acknowledgement.

Callbacks are used internally by the `socket.io` implementation to send an acknowledgement to the sender on message receipt. Since synchronization does not occur until a co-generator involved in a choice is actually committed to the channel, when a co-generator is not already suspended on the channel the callback is instead stored to be called later.

Node.js Cluster workers also utilise callbacks, but these are invoked automatically. Because of this, synchronising involves sending an actual message instead of invoking a callback function, so in this case the listener includes logic which would otherwise be in the sender callback. Web Workers are handled in a similar way. Since callback hooks are not provided when sending the actual data the sender assumes the receiver has received the data, and continues execution immediately.

4. Performance Analysis

In this section a number of performance measures are presented and analysed. A number of low-level benchmarks, including CommsTime [15], were deployed to estimate the performance of channel communication, co-generator creation and context-switching. Unless otherwise stated, the readings were taken on a desktop computer with the following specifications: Core i5-3210M 2.50GHz, with Turbo Boost up to 3.1GHz (virtualisation enabled); 8GB RAM; Windows 10 Home 64-bit. Readings were taken for the JavaScript engines shipped with clean installations of the following products: Node.js v4 64-bit, Google Chrome v49 64-bit and Mozilla Firefox v45 64-bit. Each experiment was designed to measure the time taken to complete a large number of operations, from which the mean time for a single operation was calculated. Furthermore, every experiment was repeated on three separate occasions to obtain the mean times that are presented here.

4.1. External Channel Overhead

The time it takes to send a number of empty messages over an external channel connecting a co-generator on one JavaScript instance to another co-generator running on a second instance was measured. The measurement was retaken, this time using the channel's underlying transport mechanism directly and hence bypassing the channel API. The overhead incurred by using external channels was taken as the difference between the two measurements divided by the number of messages. This experiment was repeated for channels over WebSockets and Web Workers running on all three JavaScript engines that were considered.

A Node.js server and a browser were deployed on distributed locations in order to conduct the experiment for external channels over WebSockets. The server-side instance is mandatory because of the client-server WebSocket constraint, at least until a working WebRTC channel is available. The receiver was located on the server, and the sender on the browser. Sending an empty message over a channel still necessitates a rendezvous, which is accomplished with a ping-pong message pattern. This is evident in Figure 4, since the total communication time on the channel is twice as expensive when compared to socket.io transmission. Later in this section it is shown that the synchronisation incurs a fixed overhead that fades into insignificance as the message gets larger.

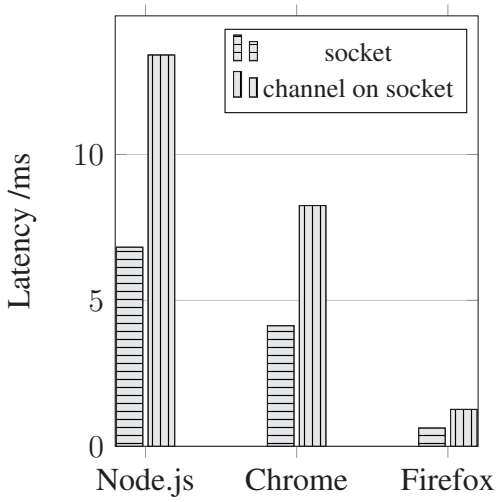

Figure 4. Latency introduced by channels over TCP sockets.

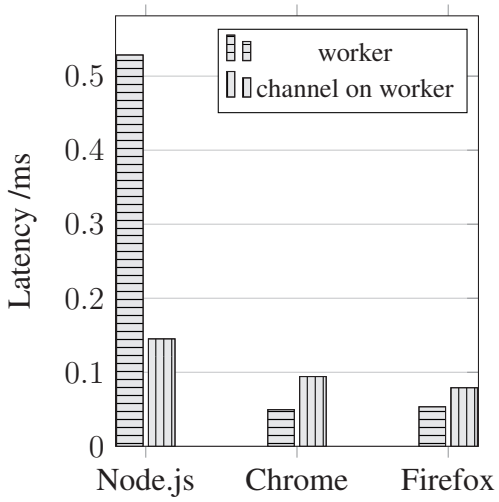

Figure 5. Latency introduced by channels over workers.

Separate experiments were conducted for worker channels on Node.js and browsers, since different worker objects (Node.js Cluster workers versus browser workers) are used by the underlying implementations. Both are evaluated by setting up the receiver on a worker and the sender on the master. Similar to distributed channels, there is an penalty associated with using worker channels. However, as seen in Figure 5, channel transmission time per message is less than twice that of the underlying implementation. This is a consequence of using the built-in callbacks instead of a reply message when notifying the sender on message receipt.

An anomaly observed in Node.js was that channels outperformed the underlying transport library. This might be due to the sender waiting for the receiver to process the previous data, thus consuming more time. This case requires further investigation, as merely adding a timeout before sending each message yielded the expected ratio. As expected, Node.js workers were generally more processor-hungry than their browser counterpart, since Node.js workers are operating system processes while browser workers employ multithreading.

4.2. Scaling Up

The time taken to perform a `yield` was measured for n co-generators running concurrently within a CSP environment, for values of n up to $10,000$. Each co-generator performed m `yield`s in a tight loop, and the time to perform one `yield` was reached by dividing the total execution time by the product of n and m. Figure 6 shows that `yield` execution time increased with the number of co-generators n instead of remaining constant.

In order to establish the source of this behaviour, a separate test was conducted to evaluate if the execution time for invoking `next()` on a generator remains the same when calling it n times on a single generator versus invoking it once on each of n generators. Single invocations on multiple generators incurred increasing overhead compared to the single generator case as n was increased. This phenomenon was studied further using a profiler, and was found to be a consequence of increasing garbage collection overhead. As it is evident that this issue is external to the library implementation, one hopes that future implementations of JavaScript will improve on generator performance at scale.

Next, producer and consumer co-generators were set up on both ends of an external channel overlying a WebSocket, and the time taken to send messages of various sizes was

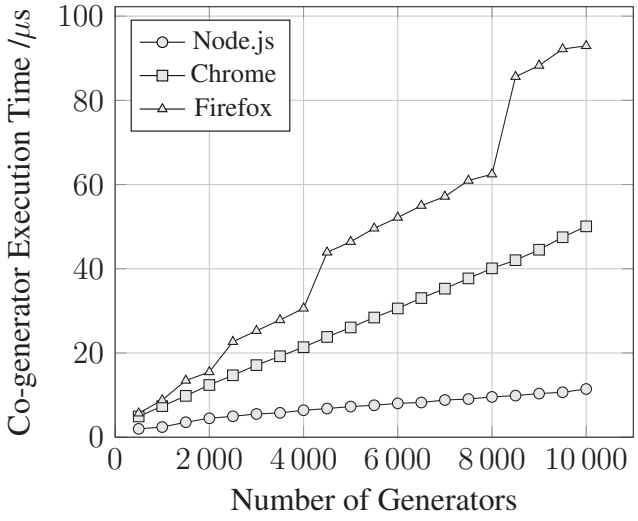

Figure 6. Scaling up co-generators in a CSP environment.

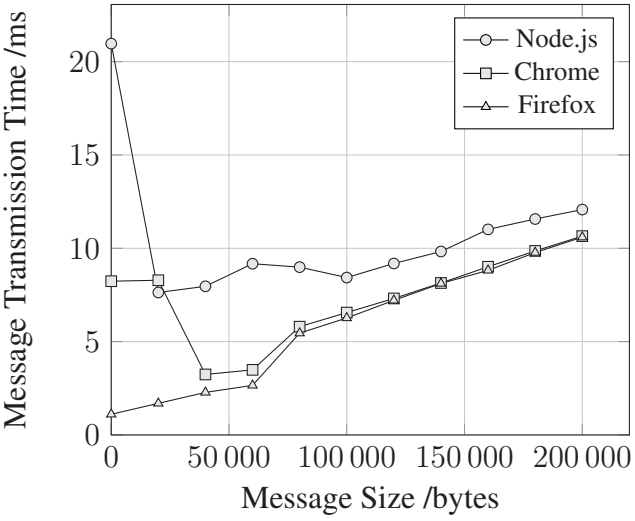

Figure 7. Scaling up message size over distributed channels.

measured. The consumer was deployed on a Node.js instance and the browser hosted the producer. Figure 7 indicates that communication time for large messages (> 100000 bytes) increased linearly with the message size. Less decipherable behaviour was observed around and below the default TCP socket send window size of 64KB. Firefox performed as expected all the way down to zero-sized messages, but Node.js observations for small messages ballooned out of control. Having checked that Nagle's algorithm [16] had already been disabled by the underlying socket library, eyes turned to the garbage collector. This hypothesis was strengthened when the expected result was sporadically observed after disabling garbage collection, but ultimately not confirmed with confidence.

4.3. CommsTime and Co-generator Creation

The CommsTime benchmark [15] derives performance metrics for communication and context-switching times in CSP-like systems. The implementation, which is reproduced in Listing 12 in its entirety, consists of four co-generators which communicate through channels for a number of iterations.

```
var csp = /*...*/; // import

function* Prefix(n, in, out){
  yield out.send(n);
  var value;
  while (true){
    value = yield in.recv();
    yield out.send(value);
  }
}

function* Delta(in, out0, out1){
  var value;
  while(true){
    value = yield in.recv();
    csp.co(function* (){
      yield out0.send(value);
    }, function* (){
      yield out1.send(value);
    });
  }
}

function* Successor(in, out){
  var value;
  while (true){
    value = yield in.recv();
    yield out.send(value + 1);
  }
}

function* Consume(loops, in){
  var t0, t1, value;
  // warm-up loop
  for (var i=0; i<1000; i++){
    value = yield in.recv();
  }
  while (true){
    t0 = Date.now();
    // benchmark loop
    for (var i=0; i<loops; i++){
      value=yield in.recv();
    }
    t1 = Date.now();
    // print results here!
  }
}

(function Commstime(){
  var a = new csp.Channel(),
    b = new csp.Channel(),
    c = new csp.Channel(),
    d = new csp.Channel();
  csp.csp(
    Prefix(0, b, a),
    Delta(a, c, d),
    Successor(c, b),
    Consume(1000000, d)
  );
}());
```

Listing 12. CommsTime benchmark.

CommsTime was executed on a single JavaScript instance, and the results are shown in Figure 8. Communication is considerably more expensive than context switching due to run-time checks to ensure a single sender and receiver on each channel, data routing between the co-generators, and rescheduling both co-generators involved in the communication on the run queue.

Measured co-generator creation times for `csp()`, `fork()` and `co()` are shown in Figure 9. The `co()` API function is the most expensive, due to barrier creation and resynchronisation of each created co-generator on its completion. The `fork()` API function, which merely schedules the co-generators at the end of the run queue, is the least expensive. The CSP environment creator, `csp()`, falls between the former two constructs in terms of overhead. It is responsible for initialising the API functions accessible within the environment, creating the dispatcher and run queue, and scheduling the passed co-generators.

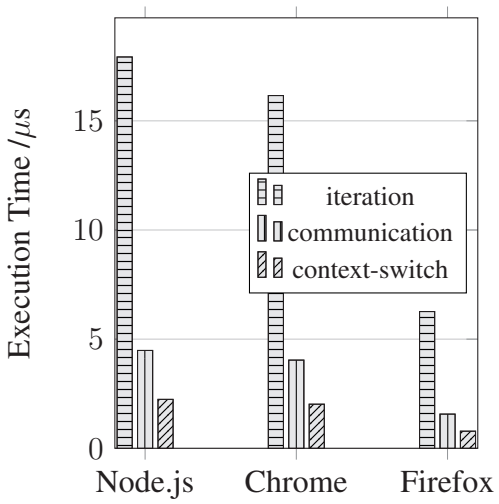

Figure 8. CommsTime performance.

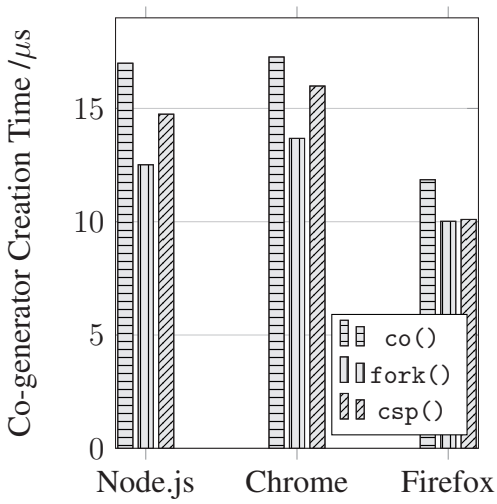

Figure 9. Co-generator creation performance.

5. Case Studies

5.1. Synchronous JavaScript

JavaScript code is typically asynchronous, with callback functions handling the deferred result. This matches its intended application, as HTML elements remain responsive instead of blocking on network operations. Unfortunately, control flow in asynchronous code is not immediately evident, especially with intricately nested callback functions. Control skips from the calling function to the called asynchronous function and immediately back to the caller. The callback function, which was passed as an argument to the asynchronous function, is typically invoked when the asynchronous call completes.

With the introduction of co-generator functions that can be suspended and resumed, callbacks are no longer necessary in JavaScript. A co-generator function that communicates through channels handles asynchronicity and blocking just as well while maintaining an intuitive control flow. The reader is referred to [17] for a detailed discussion and concrete examples. With external channels over WebSockets and eventually WebRTC the same concept applies across distributed locations.

5.2. Parallel Computing

Naive computation of the Mandelbrot set [18] was favoured as a simple and embarrassingly parallel example to demonstrate that linear speed-up is achievable when using the library for such systems. A task farming pattern was adopted: a farmer co-generator located on a Node.js server continuously supplies worker co-generators located on Cluster [11] worker nodes with tasks (individual horizontal lines of a 3000×2000 pixel canvas) to compute.

The experiment was conducted on Ubuntu 14.04.1 64-bit with Linux kernel 4.2.0-34, running on a pair of quad core Intel Xeon E5620s clocked at 2.40GHz and 24GB of RAM, with hyperthreading disabled. Figure 10 illustrates the speed-up obtained with up to eight workers, both with and without result harvesting through channels.

Linear speed-up is predictably achieved when result harvesting is not performed but the situation deteriorates when harvesting, with a top speed-up of 6 obtained on 8 cores. This may be due to `WorkerChannels` crossing address space boundaries, as Node.js Cluster workers are

individually hosted in operating system processes. However it must be noted that no profiling was attempted at this stage.

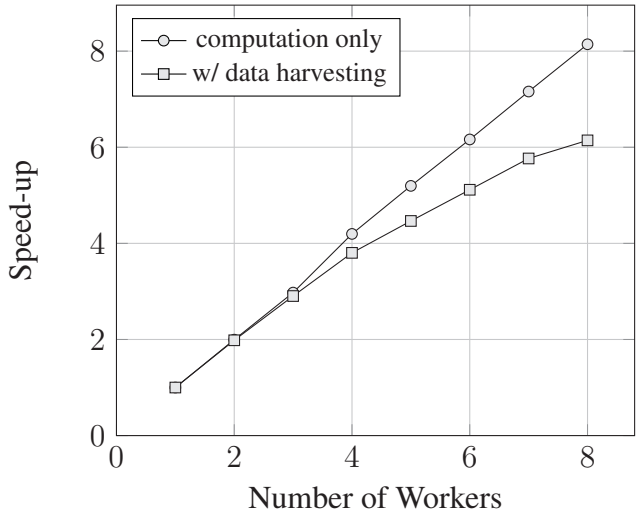

Figure 10. Mandelbrot set computation speed-up.

6. Related Work

Preceding this work and indeed inspiring it, js-csp [19] uses JavaScript generators to provide CSP-like features that are constrained to operate within a single JavaScript engine. In common with Google's Go language [20] and Clojure [21]'s core.aync [22] library, js-csp omits a `PAR`-like operation that synchronises completed generators before returning. A Clojure-to-JavaScript transpiler, ClojureScript [23], enables the use of the CSP functionality offered by core.async in JavaScript.

ECMAScript 6 also includes promises, which offer an alternative path towards CSP-like behaviour in tandem with generators, as seen in asynquence [24]. The adoption of `async` and `await` in ECMAScript 7 presents another alternative, as showcased in the async-csp [25] library.

CSP libraries are available for several popular programming languages: CCSP [26], CTC [27], and libcsp [28] for C; CTC++ [27] and C++CSP [29] for C++; JVMCSP [30,31] (a target for the ProcessJ programming language), CTJ (formerly CJT) [32] and JCSP [33] for Java and JVM languages; CSP.NET [34] and CSP for .NET [35] for the .NET framework; PyCSP [36] for Python; CSO [37] for Scala; and CHP [38] for Haskell. In particular, CCSP, JCSP, CTC++, CSP.NET, PyCSP and CSO support some form of distributed execution.

7. Conclusions

A straightforward distributed implementation of CSP-style concurrency in a JavaScript library was achieved, principally influenced by the occam language [13] and the T9000 processor [12]. It was shown that it is possible to eliminate callbacks in JavaScript with minimal effort using channels. Moreover, the use of channels was extended across distributed locations using standard technologies such as WebSockets and workers, to ultimately harness the combined processing resources of servers and browsers. The channel abstraction hides whether

the computation on the other end is local or remote; this allows the relocation of computation with significantly fewer modifications to the application code.

This implementation or elements of its design could be harnessed by process-oriented programming languages such as ProcessJ to target JavaScript. The opportunity to provide online e-learning services using an in-browser process-oriented programming environment is of particular interest [39].

Looking forward, it should be possible to implement external channels using `async` functions coupled with promises with even less boiler-plate code. One might envisage taking a library such as async-csp [25] in this direction once ECMAScript 7 is commonplace.

The library can be extended to support additional features such as shared channels as showcased in occam-π [40]. Sending channel endpoints over channels would require a suitable underlying link to be automatically identified or established at the migrated channel endpoint's new location. At present, manual establishment of the underlying link is required prior to channel creation, which can easily be performed outside the CSP environment for static communication patterns. Since link establishment is external to channel creation and communication API functions, one could conceive an unobtrusive extension to the library that would be responsible for automatically synthesizing underlying links for channels based on the resources available.

The interpreted and dynamic nature of JavaScript opens up further possibilities: any individual function's source code and associated state is available at run-time, and can be passed to `eval()` at a remote location to implement the base functionality for dynamic code mobility across distributed locations. Thus, code and execution state would be able to automatically migrate between JavaScript environments on the basis of available processing and memory capacity. Optimistic message delivery on channels can cut down waiting times while preserving synchronization semantics; the associated buffer management is easily delegated to the JavaScript engine, for better or for worse. On the flip-side, run-time performance is generally unpredictable and far below what is achievable with compiled languages.

In closing, it is noted that a significant part of this paper has focused on distributing CSP-like code across the Web. Partial failure is unavoidable over a distributed system's lifetime, and it can be argued that synchronous external channels are not a perfect match for such situations. Indeed, vanilla CSP does not consider partial failure, and neither does the library in its present state. In vague CAP [41] terms, such concurrency models fall within CA territory: Consistent and Available, but not Partition tolerant. One might justifiably ask, how should emerging libraries and languages in the CSP vein behave under network partition conditions?

Acknowledgements

The authors would like to express their gratitude to the reviewers for their valuable feedback on an earlier draft of this paper.

References

[1] Ecma International. *Standard ECMA-262 - ECMAScript Language Specification*. 6th edition, June 2015.
[2] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
[3] I. Hickson. The WebSocket API. W3C recommendation, W3C, September 2012.
[4] Web workers. WHATWG living standard, WHATWG, March 2016.
[5] A. Rauschmayer. ES6 generators in depth. http://www.2ality.com/2015/03/es6-generators.html, March 2015.

[6] K. Simpson. Diving deeper with ES6 generators. https://davidwalsh.name/es6-generators-dive, July 2014.
[7] L. M. Surhone, M. T. Tennoe, and S. F. Henssonow. *Node.Js*. Betascript Publishing, Mauritius, 2010.
[8] G. Rauch. socket.io. https://www.npmjs.com/package/socket.io, 2016. npm Module.
[9] A. Bergkvist, D. C. Burnett, C. Jennings, A. Narayanan, and B. Aboba. WebRTC 1.0: Real-time Communication Between Browsers. W3C recommendation, W3C, January 2016.
[10] G. Rauch. socket.io-p2p. https://www.npmjs.com/package/socket.io-p2p, 2016. npm Module.
[11] Cluster. https://nodejs.org/api/cluster.html, 2016. npm Module.
[12] INMOS Limited. *Networks, Routers and Transputers: Function, Performance, and Applications*. IOS Press, Netherlands, 1993.
[13] INMOS Limited. *Occam 2 Reference Manual*. Prentice Hall, May 1988.
[14] M. O. Larsen and B. Vinter. Exception Handling and Checkpointing in CSP. In *Communicating Process Architectures 2012*, pages 201–212, Dundee, Scotland, August 2012. Open Channel Publishing.
[15] D. C. Wood and P. H. Welch. The Kent Retargetable Occam Compiler. In *Proceedings of the 19th World Occam and Transputer User Group Technical Meeting on Parallel Processing Developments*, WoTUG '96, pages 143–166, Amsterdam, The Netherlands, The Netherlands, 1996. IOS Press, Netherlands.
[16] J. Nagle. RFC-896: Congestion Control in IP/TCP Internetworks. *Internet Engineering Task Force*, January 1984.
[17] J. Long. Taming the asynchronous beast with CSP channels in JavaScript. http://jlongster.com/Taming-the-Asynchronous-Beast-with-CSP-in-JavaScript, September 2014.
[18] B. B. Mandelbrot. *Fractals: Form, Chance and Dimension*. W. H. Freedman and Co., New York, NY, USA, 1977.
[19] T. A. Nguyễn. js-csp. https://github.com/ubolonton/js-csp, 2015. GitHub repository.
[20] A. A. A. Donovan and B. W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 1 edition, November 2015.
[21] R. Hickey. The Clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, July 8, 2008, Paphos, Cyprus*, page 1, New York, NY, USA, 2008. ACM.
[22] Clojure. core.async. https://github.com/clojure/core.async, 2016. GitHub repository.
[23] S. Sierra and L. VanderHart. *ClojureScript: Up and Running*. O'Reilly Media, Inc., 2012.
[24] K. Simpson. asynquence: The promises you don't know yet. https://davidwalsh.name/asynquence-part-1, June 2014.
[25] D. Lesage. async-csp. https://github.com/dvlsg/async-csp, 2016. GitHub repository.
[26] J. Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 182–196, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press, Netherlands.
[27] D. S. Jovanović, G. H. Hilderink, and J. F. Broenink. Controlling a Mechatronic Set-up using Real-time Linux and CTC++. pages 1323–1331. Drebbel Insitute for Mechatronics, University of Twente, Netherlands, 8th Mechatronics Forum International Conference, June 24-, 2002.
[28] R. D. Beton. libcsp - a Building mechanism for CSP Communication and Synchronisation in multithreaded C programs. In *Communicating Process Architectures 2000*, pages 239–250, September 2000.
[29] N. C. C. Brown and P. H. Welch. An Introduction to the Kent C++CSP library. In *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 182–196, Amsterdam, the Netherlands, September 2003. IOS Press, Netherlands.
[30] J. B. Pedersen and A. Stefik. Towards Millions of Processes on the JVM. In *Communicating Process Architectures 2014*, pages 139–168, Oxford, UK, August 2014. Open Channel Publishing.
[31] C. Shrestha and J. B. Pedersen. JVMCSP - Approaching Billions of Processes on a Single-Core JVM. In *Communicating Process Architectures 2016*, Copenhagen, Denmark, August 2016. Open Channel Publishing.
[32] G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. Communicating Java Threads. In *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent systems engineering series*, pages 48–76, Amsterdam, the Netherlands, 1997. IOS Press, Netherlands.
[33] P. H. Welch and J. M. R. Martin. A CSP Model for Java Multithreading. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, PDSE '00, pages 114–123, Washington, DC, USA, 2000. IEEE Computer Society.
[34] A. A. Lehmberg and M. N. Olsen. An Introduction to CSP.NET. In *Communicating Process Architectures 2006*, volume 64 of *Concurrent Systems Engineering Series*, pages 13–30. IOS Press, Netherlands, September 2006.

[35] K. Chalmers and S. Clayton. CSP for .NET Based on JCSP. In *Communicating Process Architectures 2006*, volume 64 of *Concurrent Systems Engineering Series*, pages 59–76. IOS Press, Netherlands, September 2006.

[36] J. M. Bjørndalen, B. Vinter, and O. J. Anshus. PyCSP - Communicating Sequential Processes for Python. In *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering Series*, pages 229–248. IOS Press, Netherlands, July 2007.

[37] B. Sufrin. Communicating Scala Objects. In *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering Series*, pages 35–54. IOS Press, Netherlands, September 2008.

[38] N. C. C. Brown. Communicating Haskell Processes: Composable Explicit Concurrency using Monads. In *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering Series*, pages 67–83. IOS Press, Netherlands, September 2008.

[39] J. B. Pedersen and M. A. Smith. ProcessJ: A Possible Future of Process-Oriented Design. In *Communicating Process Architectures 2013*, pages 133–156, Edinburgh, Scotland, August 2013. Open Channel Publishing.

[40] P. H. Welch and F. R. M. Barnes. Communicating Mobile Processes: introducing occam-pi. In *Communicating Sequential Processes: The First 25 Years*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Berlin Heidelberg, April 2005.

[41] E. Brewer. CAP twelve years later: How the “rules” have changed. *Computer*, 45(2):23–29, 2012.