CPA 2016
Communicating Generators in JavaScript

Kurt Micallef (kurtmica@live.com)
Kevin Vella (kevin.vella@um.edu.mt)

Department of Computer Science

University of Malta

Problems and Opportunities

1. Single-threaded, event-driven JavaScript limits the scope for concurrency.

Problems and Opportunities

1. Single-threaded, event-driven JavaScript limits the scope for concurrency.

2. JavaScript is a ubiquitous computing technology, running in browsers, server runtimes (Node.js) and worker contexts.

JavaScript Generators

Generators are functions which can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances.[1]

[1] Mozilla Developer Network

JavaScript Generators

Generators are functions which can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances.[1]

```
1 var generatorFunction = function* (){
2   var ret = yield 1;
3   return ret;
4 };
```

[1] Mozilla Developer Network

JavaScript Generators

Generators are functions which can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances.[1]

```
1 var generatorFunction = function* (){
2   var ret = yield 1;
3   return ret;
4 };
5
6 var generator = generatorFunction();
```

[1] Mozilla Developer Network

JavaScript Generators

Generators are functions which can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances.[1]

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

[1] Mozilla Developer Network

JavaScript Generators

Generators are functions which can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances.[1]

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

[1] Mozilla Developer Network

JavaScript Generators

Generators are functions which can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances.[1]

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

[1] Mozilla Developer Network

Generators

```
1 var delegate = function* (){
2   yield 1;
3 };
```

Generators

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

Generators

```
1 var delegate = function* (){
2   yield 1;
3 };
4
5 var generator = (function* (){
6   yield* delegate();
7 }());
8
9 var x = generator.next().value; // x = 1
```

Problems and Opportunities (revisited)

1. Single-threaded, event-driven JavaScript limits the scope for concurrency.

2. JavaScript is a ubiquitous computing technology, running in browsers, server runtimes (Node.js) and worker contexts.

Problems and Opportunities (revisited)

1. Single-threaded, event-driven JavaScript limits the scope for concurrency.
   - However JavaScript generators enable the dynamic execution of a function.

2. JavaScript is a ubiquitous computing technology, running in browsers, server runtimes (Node.js) and worker contexts.

Problems and Opportunities (revisited)

1. Single-threaded, event-driven JavaScript limits the scope for concurrency.
   - However JavaScript generators enable the dynamic execution of a function.
   - These can be repurposed as co-generators to provide co-operative multitasking in a CSP demeanour.
2. JavaScript is a ubiquitous computing technology, running in browsers, server runtimes (Node.js) and worker contexts.

The CSP Environment and Dispatcher

- Generators are initialised in a CSP environment, and execute together as co-generators.
- These are contained within a function scope, the dispatcher.

[2] Except CSP environment creation and channel creation.

The CSP Environment and Dispatcher

- Generators are initialised in a CSP environment, and execute together as co-generators.
- These are contained within a function scope, the dispatcher.

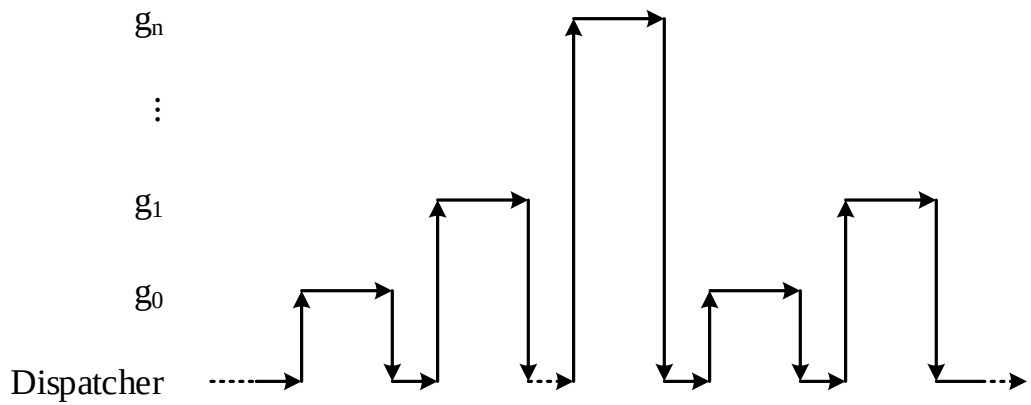


Figure: Execution flow of co-generators.

[2] Except CSP environment creation and channel creation.

The CSP Environment and Dispatcher

- Generators are initialised in a CSP environment, and execute together as co-generators.
- These are contained within a function scope, the dispatcher.

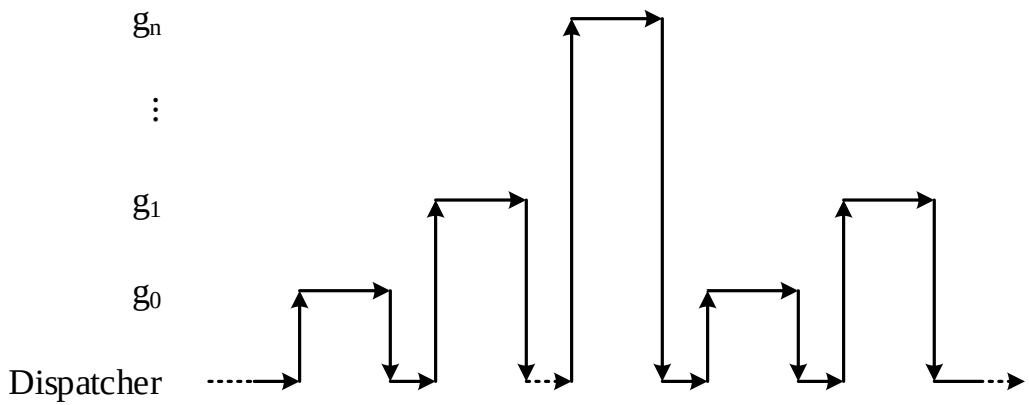


Figure: Execution flow of co-generators.

- All API functions[2] **must** be:
  - Called within a CSP environment.
  - Prefixed with a `yield`.

[2] Except CSP environment creation and channel creation.

The CSP Environment and Dispatcher

- Generators are initialised in a CSP environment, and execute together as co-generators.
- These are contained within a function scope, the dispatcher.

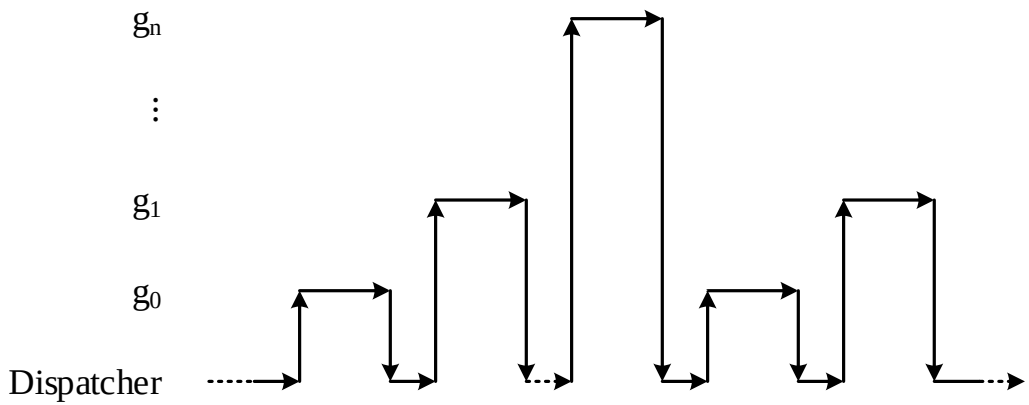


Figure: Execution flow of co-generators.

- All API functions[2] **must** be:
  - Called within a CSP environment.
  - Prefixed with a `yield`.
    - `yield` on its own is effectively a part of the API.

[2] Except CSP environment creation and channel creation.

API functions: process creation

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

Similar to occam's top-level `PAR`.

API functions: process creation

```
csp.csp(
  function* (){ },
  // ...
  function* (){ }
);
```

Similar to occam's top-level `PAR`.

```
csp.csp(function* (){
  yield csp.fork(
    function* (){ },
    // ...
    function* (){ }
  );
});
```

API functions: process creation

```
csp.csp(
  function* (){ },
  // ...
  function* (){ }
);
```

Similar to occam's top-level PAR.

```
csp.csp(function* (){
  yield csp.fork(
    function* (){ },
    // ...
    function* (){ }
  );
});
```

```
csp.csp(function* (){
  yield csp.co(
    function* (){ },
    // ...
    function* (){ }
  );
});
```

Similar to occam's PAR.

API functions: Channel communication

```
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 });
```

API functions: Timeouts

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

Similar behaviour to occam's `TIMER`s.

API functions: Timeouts

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

Similar behaviour to occam's `TIMER`s.

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

Similar to popular programming languages' `Thread.sleep()`.

API functions: Choice

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

Similar to occam's `ALT`.

Problems and Opportunities (re-revisited)

1. Single-threaded, event-driven JavaScript limits the scope for concurrency.

2. JavaScript is a ubiquitous computing technology, running in browsers, server runtimes (Node.js) and worker contexts.

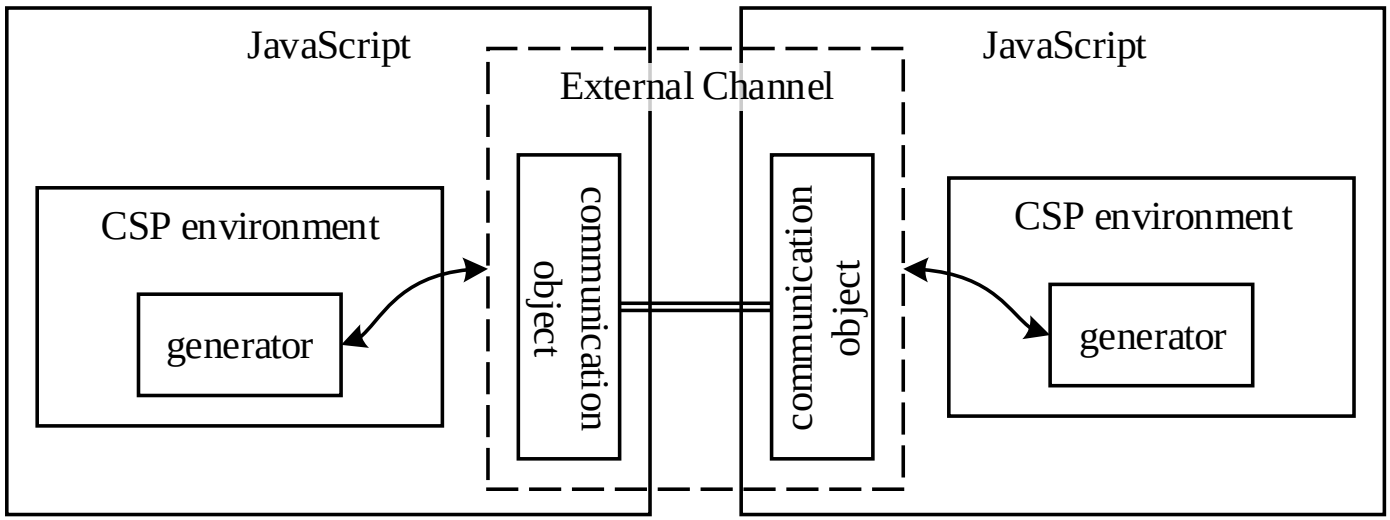
Problems and Opportunities (re-revisited)

1. Single-threaded, event-driven JavaScript limits the scope for concurrency.

2. JavaScript is a ubiquitous computing technology, running in browsers, server runtimes (Node.js) and worker contexts.
   - CSP environments can be distributed over several distinct JavaScript instances to achieve parallel execution.

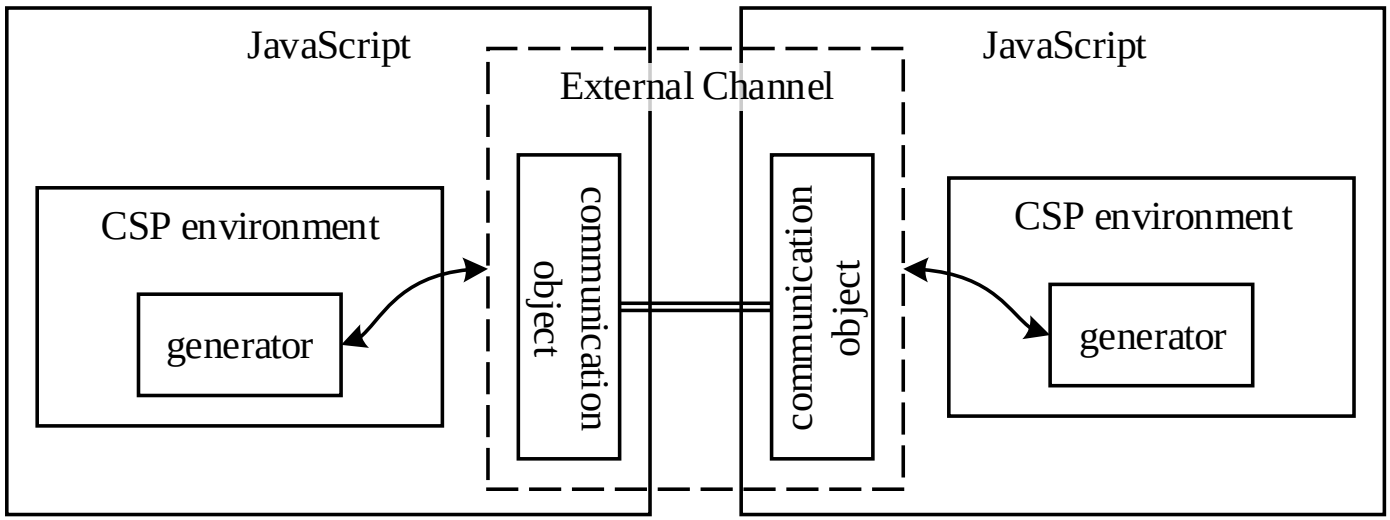
External Channels

- External channels extend across JavaScript instances by overlying various communication mechanisms.



External Channels

- External channels extend across JavaScript instances by overlying various communication mechanisms.



- JavaScript environments investigated: browsers, Node.js, and workers.
  - Transport mechanisms used: socket.io (over WebSockets), Web Workers, and Cluster Workers.

External Channels – `DistributedChannel`

External channel implementation over socket.io (WebSocket).

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

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

Listing: Channel communication between distributed co-generators.

External Channels – `WorkerChannel`

External Channel implementation over workers: Web Workers and Node.js Cluster Workers.

```
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 });
```

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

Listing: Channel communication between co-generators across Web Workers.

Recall Channels

Syntactic and semantic equivalence across channels over all types of communication mechanisms!

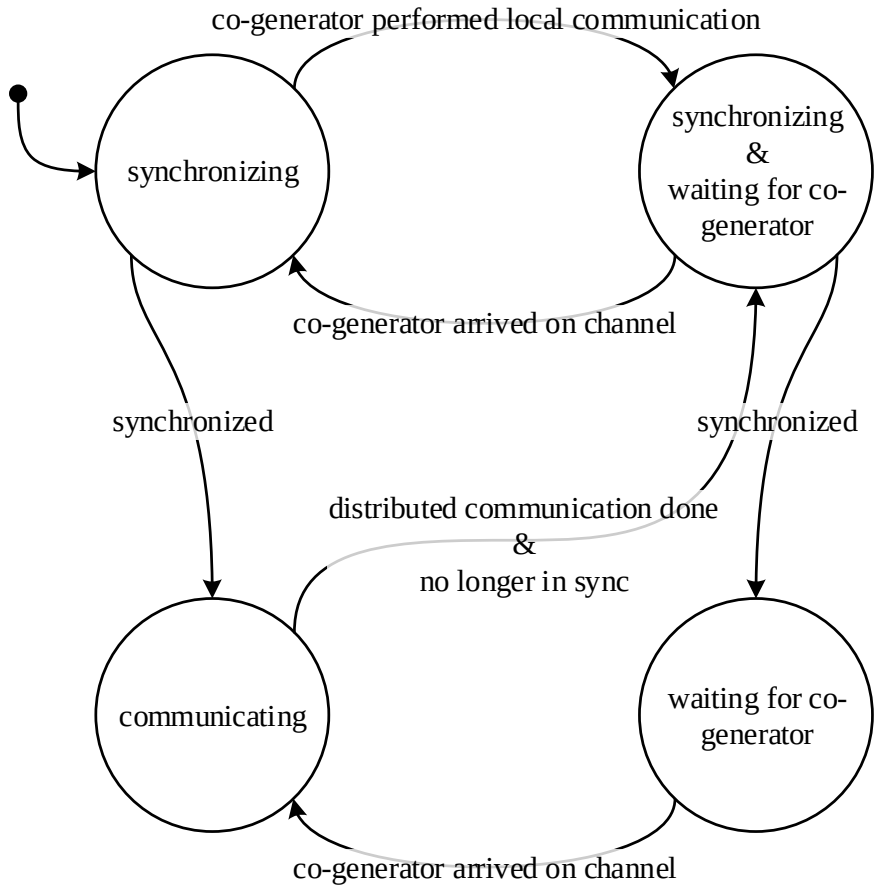
```
1 var channel = new csp.Channel();
2
3 csp.csp(function* (){
4   var x = yield channel.recv(); // x = 1
5 }, function* (){
6   yield channel.send(1);
7 });
```

External Channels – communication protocol

- Synchronize-then-communicate protocol used to alleviate any race conditions.

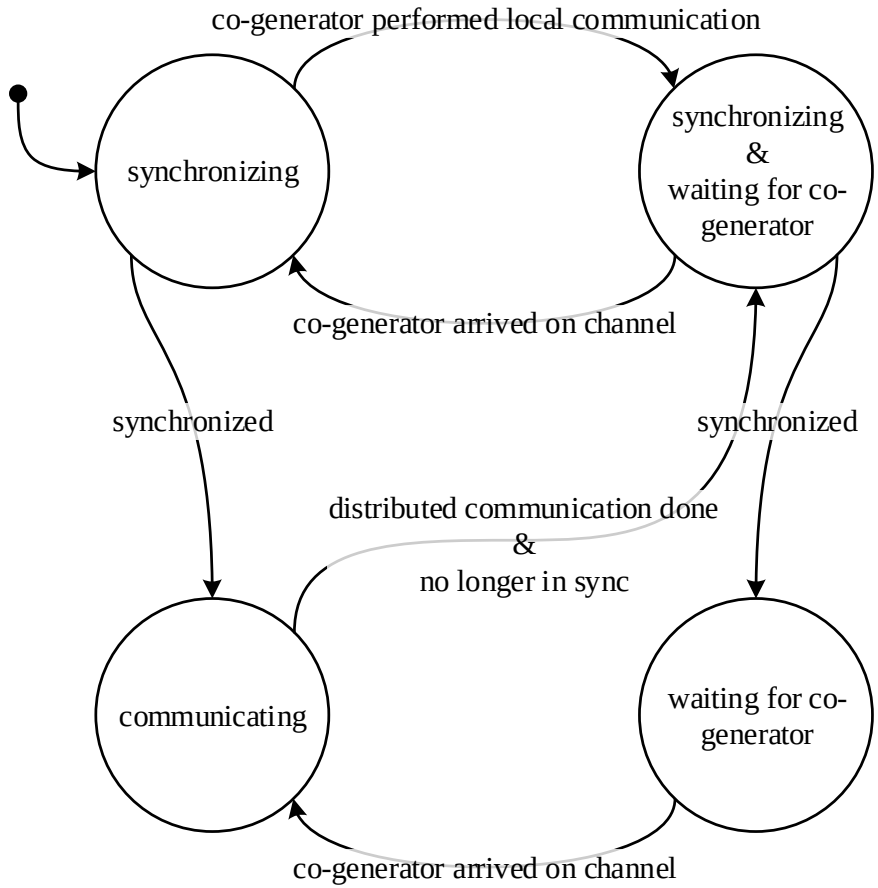
External Channels – communication protocol

- Synchronize-then-communicate protocol used to alleviate any race conditions.



External Channels – communication protocol

- Synchronize-then-communicate protocol used to alleviate any race conditions.



- This protocol allows further external channel implementations!

Performance: Co-generator Execution

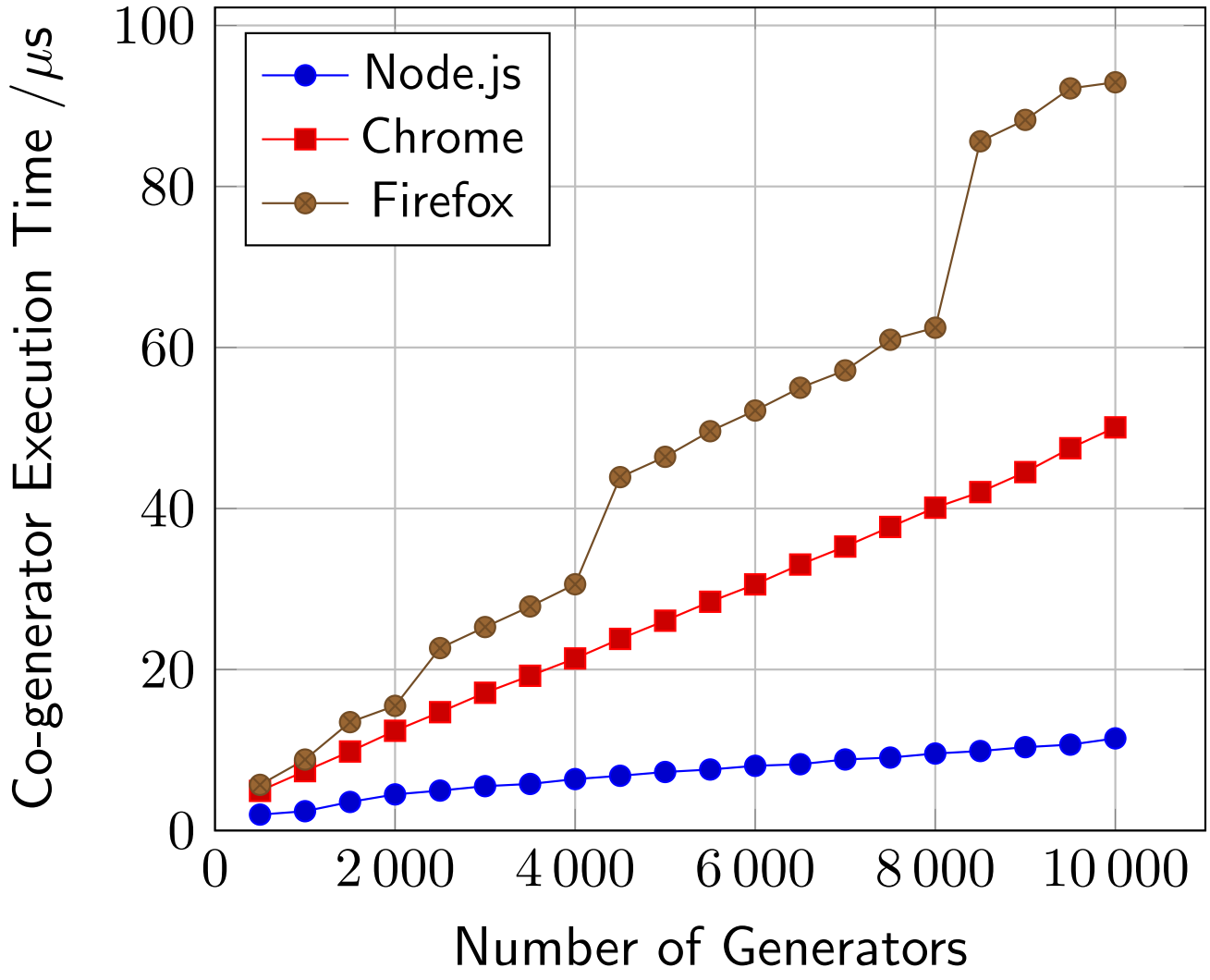


Figure: Scaling up co-generators in a CSP environment.

Performance: Message Transmission

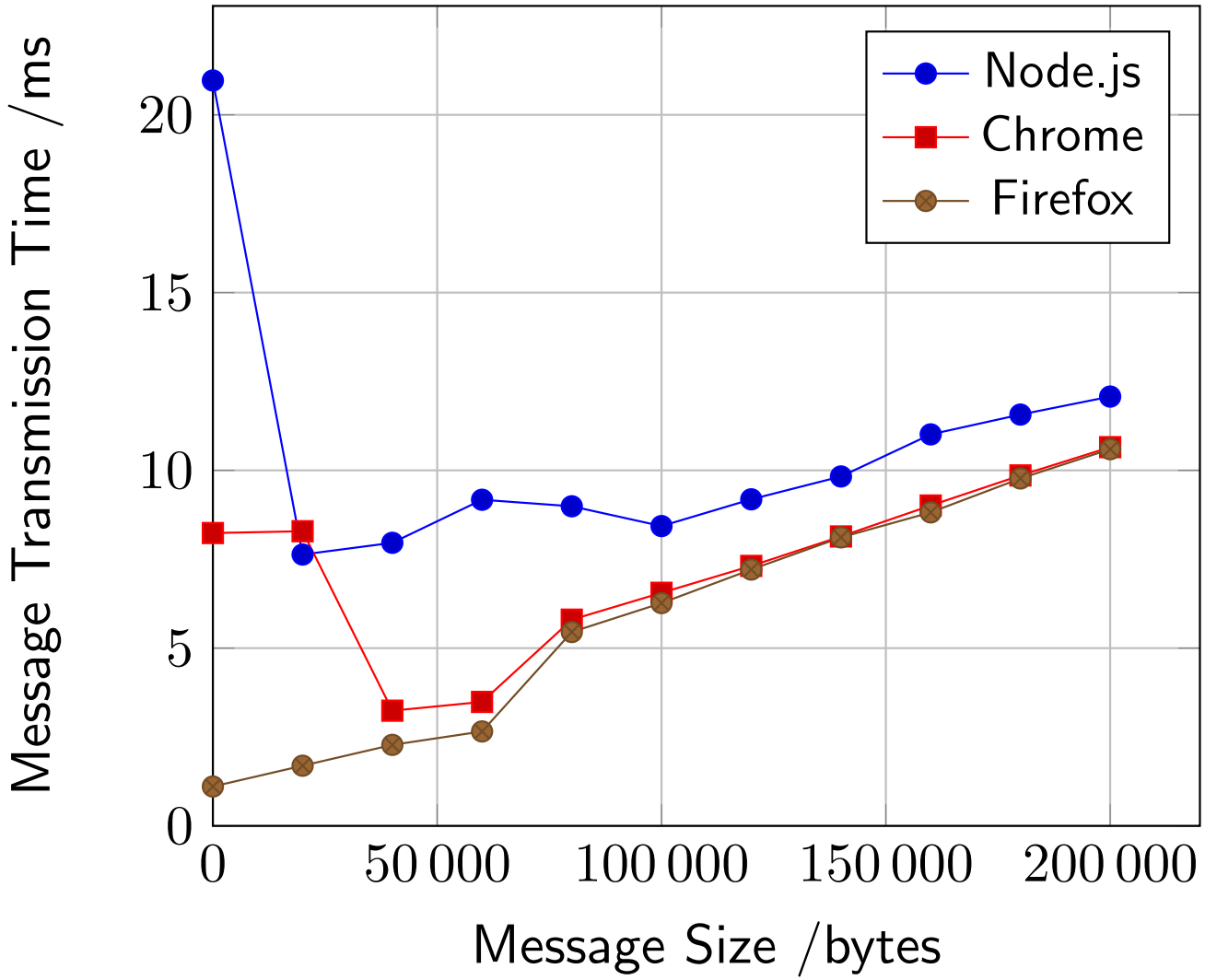


Figure: Scaling up message size over distributed channels.

Use Cases – Synchronous JavaScript

```
1 var promise = new Promise(function (resolve,reject)
   {
2   setTimeout(function callback(){
3     resolve("csp");
4   }, 1000);
5 });
```

Use Cases – Synchronous JavaScript

```
1 var promise = new Promise(function (resolve,reject)
   {
2   setTimeout(function callback(){
3     resolve("csp");
4   }, 1000);
5 });
6
7 promise.then(function (x){
8   console.log(x); // "csp"
9 });
```

Use Cases – Synchronous JavaScript

```
1 var promise = new Promise(function (resolve,reject)
   {
2   setTimeout(function callback(){
3     resolve("csp");
4   }, 1000);
5 });
6
7 promise.then(function (x){
8   console.log(x); // "csp"
9 });
```

```
1 var channel = csp.Channel();
2
3 csp.csp(function* (){
4   yield csp.sleep(1000);
5   yield channel.send("csp");
6 }
```

Use Cases – Synchronous JavaScript

```
1 var promise = new Promise(function (resolve,reject)
   {
2   setTimeout(function callback(){
3     resolve("csp");
4   }, 1000);
5 });
6
7 promise.then(function (x){
8   console.log(x); // "csp"
9 });
```

```
1 var channel = csp.Channel();
2
3 csp.csp(function* (){
4   yield csp.sleep(1000);
5   yield channel.send("csp");
6 }, function* (){
7   var x = yield channel.recv(); // "csp"
8 });
```

Use Cases – Parallel Computing

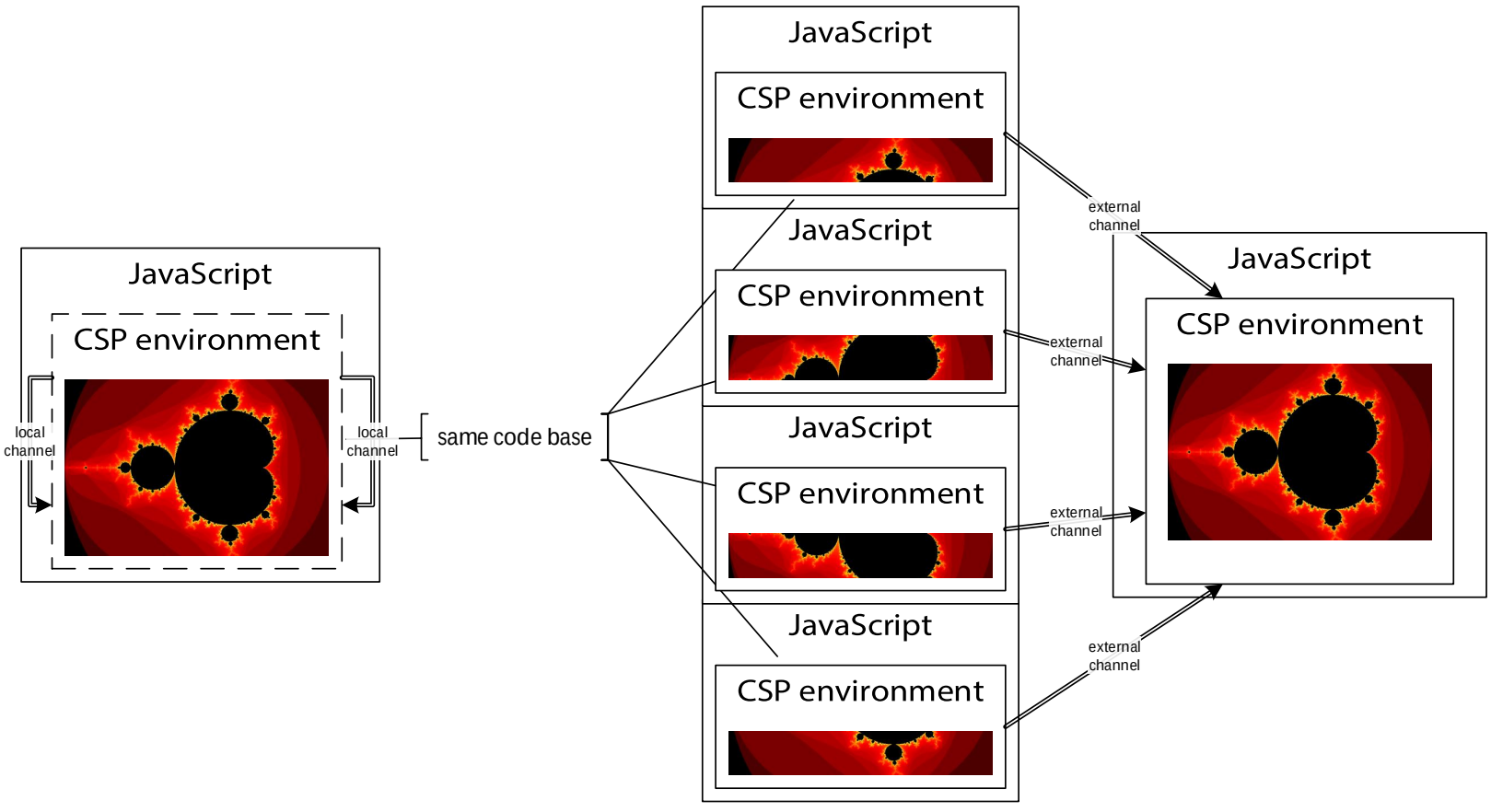


Figure: Concurrent code is reused in different distributed configurations.

Use Cases – Parallel Computing

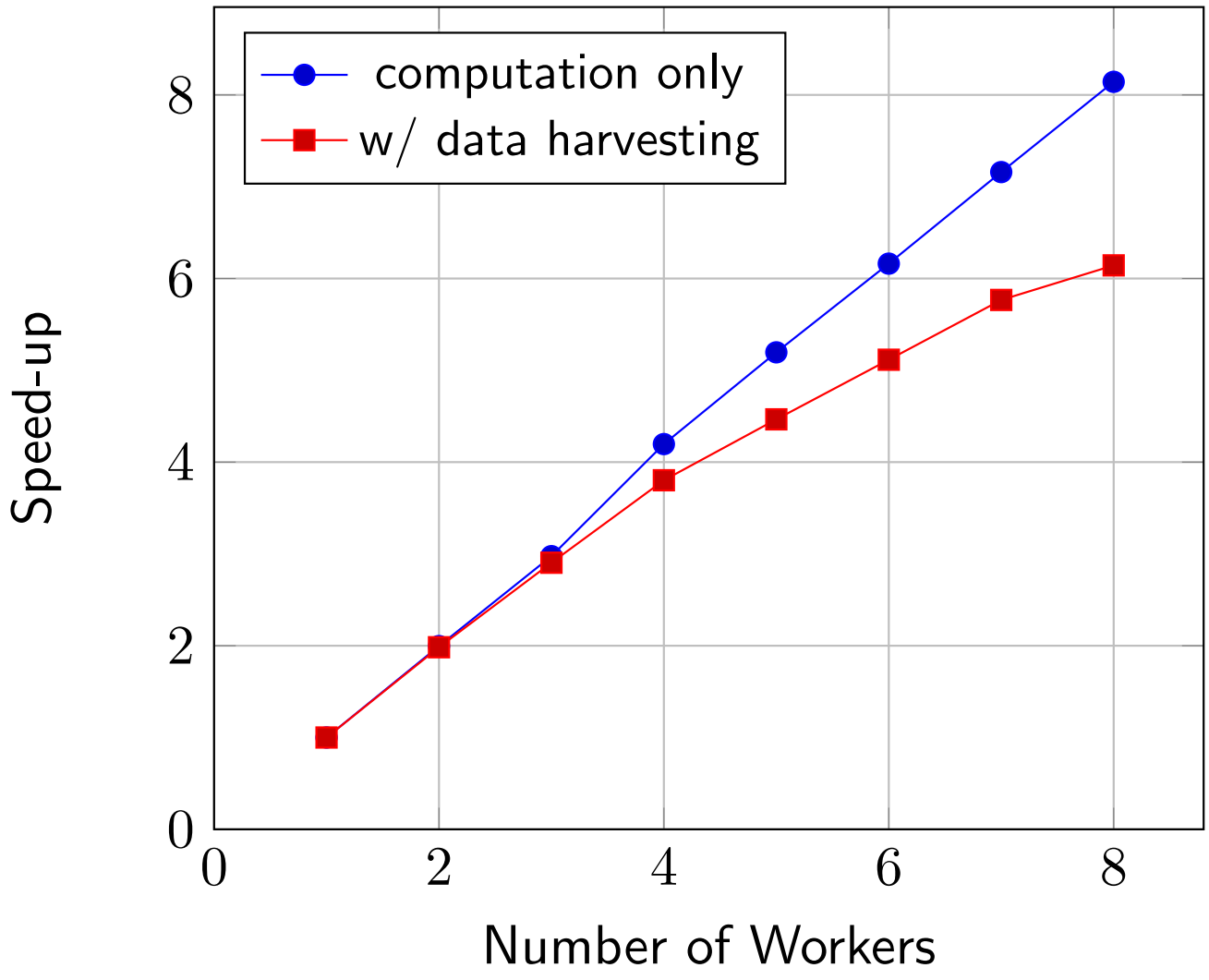


Figure: Mandelbrot set computation speed-up.

Conclusions

- A straightforward CSP library implementation in JavaScript was achieved by following the occam language and the 'Networks, Routers, and Transputers' design.

Conclusions

- A straightforward CSP library implementation in JavaScript was achieved by following the occam language and the 'Networks, Routers, and Transputers' design.
- Extending the implementation with external channels is useful because:
  - The transport mechanism is abstracted away, alleviating the need to tailor code to its location.
  - JavaScript's parallel computing capabilities can be harnessed at a higher level of abstraction.

Conclusions

- A straightforward CSP library implementation in JavaScript was achieved by following the occam language and the 'Networks, Routers, and Transputers' design.
- Extending the implementation with external channels is useful because:
  - The transport mechanism is abstracted away, alleviating the need to tailor code to its location.
  - JavaScript's parallel computing capabilities can be harnessed at a higher level of abstraction.
- By using `eval()`, simple run-time code mobility can be achieved since co-generators already use transport-agnostic channels.

Conclusions

- A straightforward CSP library implementation in JavaScript was achieved by following the occam language and the 'Networks, Routers, and Transputers' design.
- Extending the implementation with external channels is useful because:
  - The transport mechanism is abstracted away, alleviating the need to tailor code to its location.
  - JavaScript's parallel computing capabilities can be harnessed at a higher level of abstraction.
- By using `eval()`, simple run-time code mobility can be achieved since co-generators already use transport-agnostic channels.
- Distributed failures: how best to handle them in CSP-like systems?