A compiler providing incremental scalability for web applications

Etienne Brodu
etienne.brodu@insa-lyon.fr
IXXI – ENS Lyon
15 parvis René Descartes – BP 7000
69342 Lyon Cedex 07 FRANCE

Stéphane Frénot
stephane.frenot@insa-lyon.fr
IXXI – ENS Lyon
15 parvis René Descartes – BP 7000
69342 Lyon Cedex 07 FRANCE

Fabien Cellier
fabien.cellier@worldline.com
Worldline
Bât. Le Mirage
53 avenue Paul Krüger
CS 60195
69624 Villeurbanne Cedex

ABSTRACT
To develop a web application, one needs to choose between two programming models. The monolithic one favors features improvements, while the decentralized one favors performance improvements. To avoid this choice, we compile monolithic web applications into a high-level language compliant with a distributed model.

Categories and Subject Descriptors
Software and its engineering [**Software notations and tools**]: Compilers—*Runtime environments*

General Terms
Compilation

Keywords
Flow programming, Web, Javascript

1. INTRODUCTION
A community quickly adopts a web application if it complies with user requirements. To do so, the monolithic programming approach facilitates quick enhancements in features. Java is a very popular choice using this approach. To cope with the community growth, the resources shall grow proportionally. The threading approach to scale an application on growing resources is error-prone. Eventually this growth requires to replace the initial approach for models providing incremental scalability. These models generally distribute application parts on a cluster of commodity machines[4]. However these models are specific and require the development team to be trained and to start over the initial code base. These modifications imply to spend development resources in background without adding visible value for the users.

Node.js provides a more efficient approach for network applications than the threading approach. Its monolithic approach facilitates quick enhancements in features, but lacks incremental scalability. It is based on an event-loop consuming messages, similarly the models described above. Because of this similarity, we think it is possible to identify autonomous parts communicating through data streams.

To lift the risks described above, we maintain the *Node.js* monolithic approach and propose a tool to compile it into a high-level language compatible with a more scalable model. It extracts autonomous parts by searching for rupture points marking them out.

We present rupture points in section 2, the high-level language in section 3, and a compilation example in 4. Finally, we cite related works in section 5 and conclude this presentation.

2. RUPTURE POINTS
A **rupture point** is a call of a loosely coupled function. It indicates an interface between two application parts along a data stream. In *Node.js*, I/O operations are asynchronous functions. That is a function call that resumes immediately, with a function to process later the result of the operation : the callback. The callback is loosely coupled with the initial caller. An asynchronous call indicates a rupture point. To detect rupture points, the compiler uses a predefined list of asynchronous functions. For example `app.get` and `fs.readFile` in listing 1, lines 5 and 6.

```
1  var app = require('express')(),
2      fs = require('fs'),
3      count = 0;
4
5  app.get('/', function handler(req, res){
6    fs.readFile(__filename, function reply(err, data) {
7      count += 1;
8      var code = ('' + data).replace(/\n/g, '<br>').
           replace(/ /g, ' ');
9      res.send(err || 'downloaded ' + count + ' times<
           br><br><code>' + code + '</code>');
10   });
11 });
12
13 app.listen(8080);
14 console.log('>> listening 8080');
```

Listing 1: A simple application presenting two asynchronous functions : `app.get` and `fs.readFile`

In *Node.js*, there is a single thread of execution to avoid concurrent memory access. The compiler needs to control the memory to assure independence between application parts. If an application part reads a variable from another part, it sends the variable downstream. If an application part modifies a variable from another part, it persists the variable. If several application parts modify the same variable, the compiler merges these parts.

3. HIGH-LEVEL LANGUAGE

⟨program⟩ ⊨ ⟨flx⟩ | ⟨flx⟩ *eol* ⟨program⟩
⟨flx⟩ ⊨ `flx` ⟨id⟩ ⟨ctx⟩ *eol* ⟨streams⟩ *eol* ⟨fn⟩
⟨streams⟩ ⊨ `null` | ⟨stream⟩ | ⟨stream⟩ *eol* ⟨streams⟩
⟨stream⟩ ⊨ ⟨op⟩ ⟨dest⟩ [⟨msg⟩]
⟨dest⟩ ⊨ ⟨list⟩
⟨ctx⟩ ⊨ `{`⟨list⟩`}`
⟨msg⟩ ⊨ `[`⟨list⟩`]`
⟨list⟩ ⊨ ⟨id⟩ | ⟨id⟩ , ⟨list⟩
⟨op⟩ ⊨ `>>` | `->`
⟨id⟩ ⊨ *Javascript identifier*
⟨fn⟩ ⊨ *Javascript and stream syntax*

The compiler encapsulates an application part in a *fluxion* ⟨flx⟩. It is an autonomous entity of execution with a unique name ⟨id⟩ and a persisted memory, called *context* ⟨ctx⟩. At a message reception, it executes its function ⟨fn⟩. This function uses the Javascript syntax augmented with ⟨stream⟩ to indicate an output stream to other fluxions ⟨dest⟩. The function ⟨fn⟩ is indented to be demarcated from the rest.

4. COMPILATION EXAMPLE

To illustrate the compiler features, we compiled the example from listing 1 into listing 2. Source and result are available on github[2]. The two asynchronous functions from the source are detected, which result in three *fluxions*. Callbacks are replaced by output *streams*. The variable `res` is read by *fluxion* `reply-1001`, and is forwarded along the message stream. The variable `count` is modified only by *fluxion* `reply-1001`, and is persisted in its *context*.

```
1  flx source.js {}
2  >> handler-1000 [res]
3    var app = require('express')(), fs = require('fs'),
         count = 0;
4    app.get('/', >> handler-1000);
5    app.listen(8080);
6    console.log('>> listening 8080');
7
8  flx handler-1000 {fs}
9  -> reply-1001 [res]
10   function handler(req, res) {
11     fs.readFile(__filename, -> reply-1001);
12   }
13
14 flx reply-1001 {count}
15 -> null
16   function reply(err, data) {
17     count += 1;
18     var code = ('' + data).replace(/\n/g, '<br>').
           replace(/ /g, ' ');
19     res.send(err || 'downloaded ' + count + ' times<
           br><br><code>' + code + '</code>');
20   }
```

Listing 2: Compilation result for listing 1

5. RELATED WORKS

Our work is based on *Node.js*[1] by Ryan Dahl. It is also inspired by works on incremental scalability, like the Staged Event-Driven Architecture of Matt Welsh[9], System S developped in the IBM T. J. Watson research center[5]. More recent projects are Spark Streaming[10], MillWheel[1], Timestream[8] and Storm[2]. The idea is also inspired by works on DataFlow leading up to Flow-Based programming (FBP)[7] and Functional Reactive Programming (FRP)[3]. Both FBP and FRP, recently got some attention in the Javascript community, respectively with the projects *NoFlo*[3] and *Bacon.js*[4]. Promises[6] are related to our work. Like callbacks, they bring an asynchronous execution model to a synchronous programming style.

6. CONCLUSION

We present in this paper a compiler to transform a *Node.js* web application into independent parts communicating by message streams. The compiler spots rupture points in the application indicating an interface between two parts. We believe this compilation can enable incremental scalability for web applications without discarding the monolithic programming model favoring enhancements in features.

Références

[1] T Akidau et A Balikov. ■ MillWheel : Fault-Tolerant Stream Processing at Internet Scale ■. In : *Proceedings of the VLDB Endowment 6.11* (2013).

[2] Etienne Brodu. *flx-example*. doi : `10.5281/zenodo.11945`.

[3] C Elliott et Paul Hudak. ■ Functional reactive animation ■. In : *ACM SIGPLAN Notices* (1997).

[4] A Fox, SD Gribble, Y Chawathe, EA Brewer et P Gauthier. *Cluster-based scalable network services*. 1997.

[5] N Jain, L Amini, H Andrade et R King. ■ Design, implementation, and evaluation of the linear road benchmark on the stream processing core ■. In : *Proceedings of the ...* (2006).

[6] B Liskov et L Shrira. *Promises : linguistic support for efficient asynchronous procedure calls in distributed systems*. 1988.

[7] JP Morrison. *Flow-Based Programming*. 1994, p. 1–377.

[8] Z Qian, Y He, C Su, Z Wu et H Zhu. ■ Timestream : Reliable stream computation in the cloud ■. In : *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)* (2013).

[9] M Welsh, SD Gribble, EA Brewer et D Culler. *A design framework for highly concurrent systems*. 2000.

[10] M Zaharia, T Das, H Li, S Shenker et I Stoica. ■ Discretized streams : an efficient and fault-tolerant model for stream processing on large clusters ■. In : *Proceedings of the 4th ...* (2012).

1. http ://nodejs.org/
2. http ://storm-project.net/
3. https ://github.com/bergie/noflo
4. https ://github.com/raimohanska/bacon.js