Liquefy the cloud

Etienne Brodu
PhD student, poster presenter
DICE (CITI Insa Lyon / INRIA Grenoble Rhône-Alpes) & Worldline
etienne.brodu@insa-lyon.fr

Stéphane Frénot
DICE (CITI Insa Lyon / INRIA Grenoble Rhône-Alpes)
stephane.frenot@insa-lyon.fr

Fabien Cellier
Worldline
fabien.cellier@worldline.com

Frédéric Oblé
Worldline
frederic.oble@worldline.com

Abstract

Web services handle a fluctuating number of connected users implying variation of resources usage. These connections are usually processed by web services as a stream of user requests.

To be able to handle these variations in the stream, classical solutions offer developers an API to split the web service into parts, and dynamically distribute them onto multiple machines to balance the load.

We offer to automatically transpile a web service from regular code to multiple atomic stateless parts instead of constraining developers to adhere to a specific syntax.

Our destination model of transpilation is composed of atomic stateless parts containing the logic, listening for messages, and sending modified messages to other parts. A messaging system delivers messages between the parts.

To allow these stateless parts to store states, the messaging system binds a memory context with the message just before delivering it to the parts, and stores back modifications.

The messaging system then would be able to distribute these atomic parts between machines to balance the load from each part, and move memory contexts accordingly.

Using transpilation instead of a specific API, this approach keeps the scalability distinct from development, allowing developers to focus on business logic, and not en scalability issue, thus allowing small business to rapidly grow a user base.

Presentation of this poster

Etienne Brodu is a PhD student working part time in the DICE team, in the INRIA - CITI laboratory, and in the High Processing and Volumes (HPV) team at Worldline. He will present the poster, and the results we worked on during his first year as a PhD student.

Liquefy the cloud

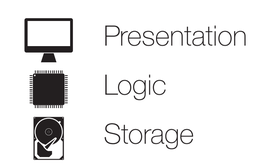
Etienne Brodu, Stéphane Frenot, Frédéric Oblé, Fabien Cellier

etienne.brodu@insa-lyon.fr, stephane.frenot@insa-lyon.fr, frederic.oble@worldline.com, fabien.cellier@worldline.com

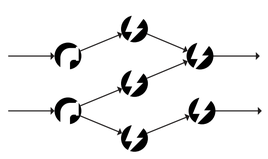
How to abstract web services' usage variation from developpement ?

A popular web service might grow from thousands to millions of users in a matters of days.
To react to such variation of usage, they have to be scalable.

The classical approaches - the **three tiers** architecture, frameworks like **storm** or **languages** like Erlang - allow developer to split web services into well defined parts in order to be scalable.



Three tiers architecture


Storm

Instead we want to **automatically** split a web service into **stateless parts**, and make them communicate by **volatile data streams**.

The persistence is decoupled from the logic and managed into the messaging system.

Statelessness and volatility assure the web service to be **frictionless***, and the association with this fine decomposition, make it **scalable**.

***Frictionless** : independent from the hardware allowing it to be moved at runtime without relocation of ressources.

In a context of **data-stream oriented** web services written in **javascript**, we want
+ to create a **frictionless*** execution model and
+ to **transpile** standard web services into this model

Example : a visit counter

// simple javascript

```
function store(userId, req, res, callback) {
  // Internal call : synchronous
  var result = localProcess(userId, req);

 // External Call : asynchronous
  callExternalDB(userId, result, function(result) {
    if (result.condition)
      return callback(result);
  });
}

// Entry point
app.get('/:id', function(req, res){
  return store(req.params.id, res, function(result) {
    res.send(result);
  });
});
```

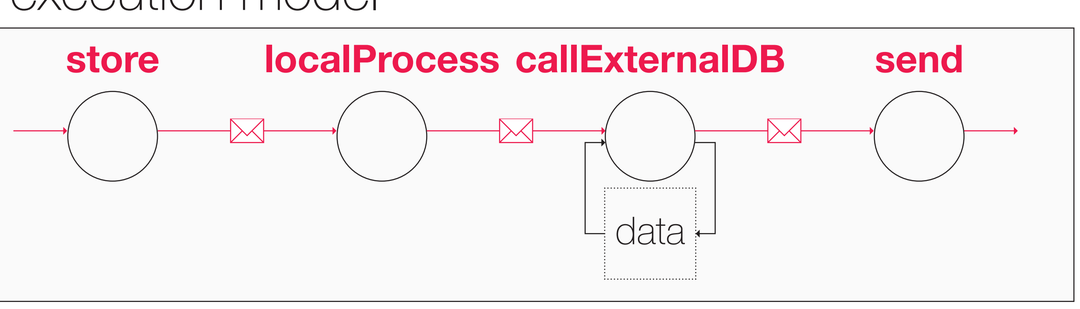
Extraction

// frictionless model

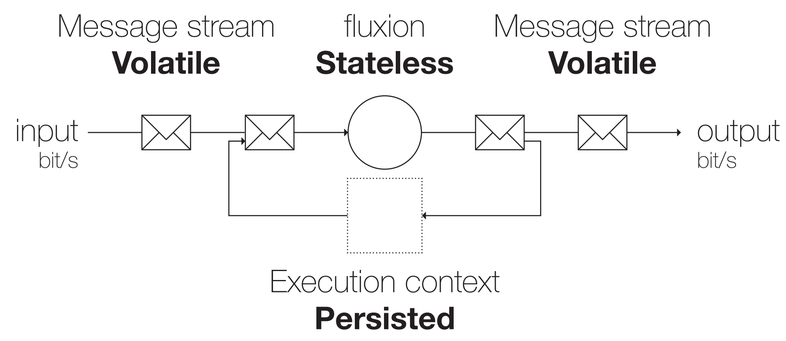
```
input >> store
store >> localProcess
localProcess >> callExternalDB
callExternalDB >> send
send >> output
```

Execution

// execution model



Execution model



Our **execution model** is composed of :

Fluxion, stateless parts, listens for, modifies and sends messages to other fluxions.

Execution context are persisted memory states needed by fluxions.

Messaging system keeps tracks of fluxions, and delivers volatile message streams.
+ Binds context execution and messages.
+ Moves fluxions and contexts to balance load.

Progression & Objectives

We aim to transpile any javascript web service into one which can adapt dynamically to load.
Without the development constraints imposed by other approaches.

DONE a javascript library to express and execute a fluxional program written in javascript.

TODO enhance this library with the automatic migration of fluxions, and create a langage with a compiler to javascript.