Communicating Event Loops
An exploration in Javascript

Tom Van Cutsem
Vrije Universiteit Brussel

Mark S. Miller
Google Research
Kludging Towards Distributed Objects

Browser → JSONP → Server

Frame → JSONP → Server

XHR GET/POST → XHR Response, Comet

Fragment tricks
A Web of Distributed Objects

Browser

Frame

postMessage

Frame

Server

XHR GET/POST

XHR Response, SSE

Cross-Origin XHR with UMP

XHR GET/POST

XHR Response, SSE

Web services

Server
A Web of Distributed Objects

Mobile messages, code, objects
A Web of Distributed Objects
Communicating Event Loops

Events = asynchronous messages
Event handlers = methods
Vats

- A vat is a container for objects, consisting of:
  - A heap of objects
  - A LIFO call stack of method invocations
  - A FIFO queue of pending message deliveries
  - Incoming and outgoing references (see later)
Turns

- A vat processes messages in its queue sequentially
- Each such message triggers a method invocation on a local object
- This method is run to completion
  - Determines a single execution "turn"
  - No preemption, no interleaving
- Computation proceeds
  - From stack top to bottom
  - From queue left to right
Properties

• Within a vat: plain sequential OOP
• Between vats: strictly asynchronous messaging
  o no conventional deadlocks
  o hides latency
• Explicit unit of interleaving (turns)
  o turns run to completion: no races on vat-local state
  o easy to add new events without breaking existing code
  o control flow across turns "inverted"
• Explicit locality boundaries (vats)
  o no synchronously accessible shared state
  o easy to add new vats without breaking existing vats
  o partitioning state across vats requires consideration
• Non-determinism is restricted to message arrival order
Immediate call vs. eventual send

o.m(1,2,3); // immediate call

o ! m(1,2,3); // eventual send
Promise

- A placeholder for an asynchronous value
- Either resolved with a value or broken with an exception

```
let p = o ! m(x);
```
Example

```javascript
let calculator = {
    add: function(x, y) { return x + y; },
    ...
};

let sumP = calculator ! add(3, 5);
// sumP resolves to 8 in a later turn
```
Promise

It is *not* possible to block a vat to await the value of a promise.

```
let p = o ! m(x);
p.get();
```
**When** (control-flow synchronization)

How to get at the resolved value?

```javascript
Q.when(expr, function(v) {
    // "callback"
}, function(err) {
    // "errback"
})
```

- `expr` may evaluate to a promise
- `v` will be bound to the promise's fulfilled value
- callback or errback *guaranteed* only to execute in a later turn
- *either* the callback *or* the errback triggers at most once
Example

```javascript
let calculator = {
    add: function(x, y) { return x + y; },
    ...
};

Q.when(calculator.add(3, 5), function(sum) {
    console.log(sum); // logs 8 in a later turn
});
```
Promise chaining
(dataflow synchronization)

Dependent promises form a dataflow network

```javascript
let p1 = o ! m(x);
let p2 = p1 ! n(y); // p2 depends on p1

let o2 = {
    f: function() { let p3 = o ! m(x); return p3; }
};
let p4 = o2 ! f(); // p4 depends on p3
```
Promise chaining
(dataflow synchronization)

- when-expression evaluates to a promise itself

```javascript
// p2 depends on p1
let p2 = Q.when(p1, function(x) {
    return x + 1;
}, function(err) {
    throw err;
});
```

Q.when reconciles asynchronous programming with functional programming style.
Possible syntactic sugar

\[
\text{await } = \text{ shallow continuation + Q.when}
\]

```javascript
let p2 = function() {
    try {
        return (await p1) + 1;
    } catch (err) {
        throw err;
    }
}(());
```
Explicit promise creation

Required when promise resolution should be postponed based on conditions other than message passing

```javascript
function delay(millis, answer = undefined) {
  let {promise, resolve} = Q.defer();
  setTimeout(() => {
    resolve(answer);
  }, millis);
  return promise;
}
```
Broken promise contagion

A promise that depends on a broken promise itself becomes broken, with the same exception

```javascript
let o = {
    m: function() { throw "an exception"; }
};

Q.when(o ! m(x) ! n(y), function(x) {
    // ...
}, function(err) {
    // will trigger with err = "an exception"
}
```
Asynchrony contagion

Asynchrony cannot be hidden by functional abstraction

```javascript
function f() {
    ...
    return v;
}

function g() {
    /*A*/
    let val = f();
    /*B*/
    return v2;
}
```

```javascript
function f() {
    ...
    return p;
}

function g() {
    /*A*/
    let p2 = Q.when(f(),
        function(val) {
            /*B*/
            return p2;
        });
    return p2;
}
```
Asynchrony contagion

Asynchrony cannot be hidden by functional abstraction

```javascript
function f() {
    ...
    return v;
}

function g() {
    /*A*/
    let val = f();
    /*B*/
    return v2;
}
```

```javascript
function f() {
    ...
    return p;
}

function g() {
    /*A*/
    let val = await f();
    /*B*/
    return v2;
}
```
Communicating event loops

- Objects can be spread across multiple vats
  - May or may not be distributed across multiple machines
  - Near vs Far references
  - Far reference points to an *individual* object within another vat, *not* to the vat as a whole (!)
Distributed parameter passing semantics

• By default, objects are passed "by far reference"
  o invoked method is given a far reference to the object
• Primitive values are passed by copy
• Can easily pass objects "by copy" by serializing them into a JSON string

// in vat A
let arg = {...};
obj ! m(arg);

// in vat B
function m(param) {
  // param is far
}
Assume o is a far reference, pointing to a vat serving the URL https://...

let $p = o \rightarrow m$;

GET https://...?q=m

let $p = o \rightarrow m(x)$;

POST https://...?q=m body

where

body = JSON.stringify(x, ...)

Eventual references

- Promises and far references are *eventual*.
- An eventual reference enforces eventual (asynchronous) access to its target.

<table>
<thead>
<tr>
<th></th>
<th>Near reference</th>
<th>Eventual reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate call</td>
<td>Method invocation</td>
<td>Error</td>
</tr>
<tr>
<td>o.m()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eventual send</td>
<td>Enqueue message in own vat</td>
<td>Enqueue message in target's vat</td>
</tr>
<tr>
<td>o! m()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
State diagram of a promise, revisited
Failures

What if the target of an eventual reference becomes disconnected (e.g. due to a network partition)?

Model allows for different failure semantics. For example:

- **E**: partial failure permanently breaks the reference
- **AmbientTalk**: messages are buffered on disconnected refs, may reconnect, may also "expire" ("leased" references)
- **Waterken**: partial failures never break references

What model to support in Javascript?
Experimental feature: where

- Javascript scripts are routinely exchanged between machines (mobile code)
- Why not provide direct linguistic support for this idiom?

```javascript
let local = { ... };
Q.when(o!m(), function(v) {
  // v is eventual
  // local is near
}, function (err) {
  // local is near
});
```

```javascript
let local = { ... };
Q.where(o!m(), function(v) {
  // v is near
  // local is eventual
}, function (err) {
  // local is near
});
```
Example: MapReduce Lite

// initValue = value of T'
// elemPs = array of promise<T>
// mapper = closed, mobile function T -> T'
// reducer = function T' x T' -> T'
// returns promise<T'> | T'

function mapReduce(initValue, elemPs, mapper, reducer) {
    let countDown = elemPs.length;
    if (countDown === 0) { return initValue; }
    let result = initValue;
    let {promise, resolve} = Q.defer();

    elemPs.forEach(function(elemP) {
        let mappedP = Q.where(elemP, mapper);
        Q.when(mappedP, function(mapped) {
            result = reducer(result, mapped);
            if (--countDown === 0) { resolve(result); }
        }, resolve);
    });
    return promise;
}
History

Hewitt's Actor model

Liskov's Argus: guardians ~ vats, (blocking) promises

E: Original-E, Joule, Vulcan

AmbientTalk: E, Yonezawa's ABCL
Communicating Event Loops (Recap)

- Concurrency model that:
  - Blends well with objects and messages
  - Scales to distributed programs
- Explicit locality boundaries (*vats*)
  - Within a vat: plain sequential OOP
  - Between vats: strictly asynchronous messaging
- Explicit unit of interleaving (*turns*)
  - Promises stitch turns together
  - Reconcile asynchronous and functional programming