

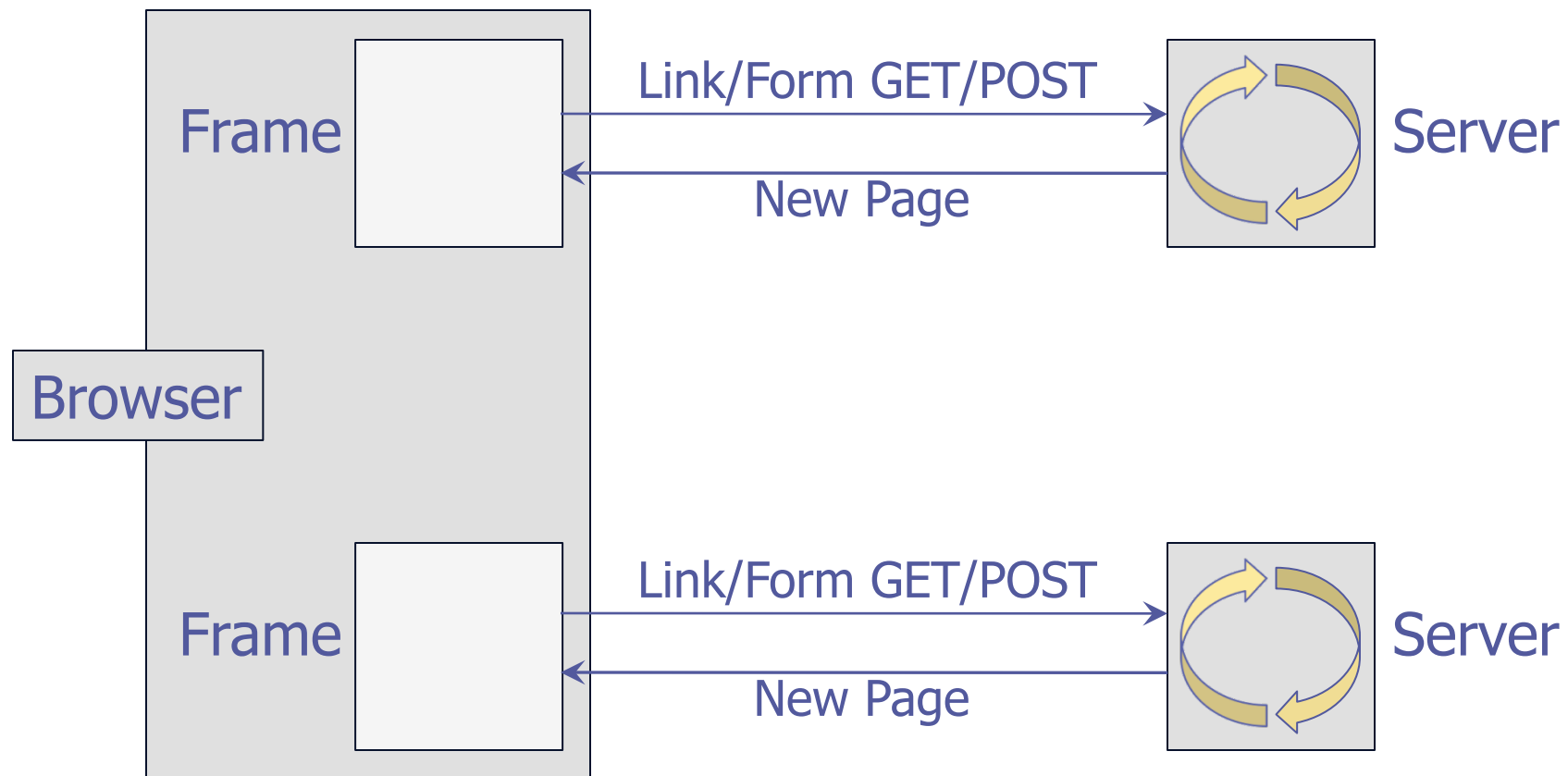
Why is E Merging with JavaScript?

Adventures in Strategic Compromise

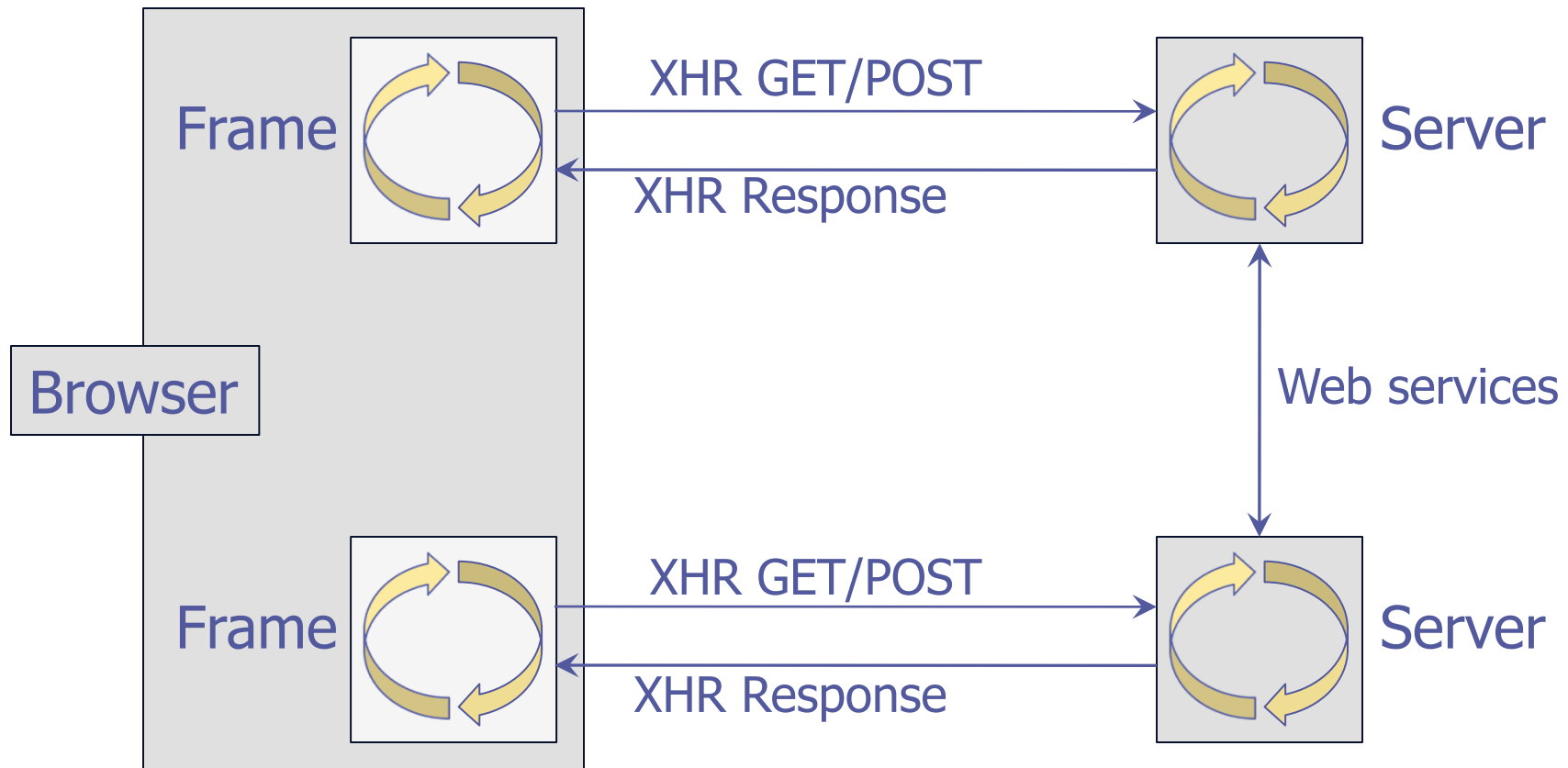
Mark S. Miller and the Cajadores
Emerging Languages, 2010



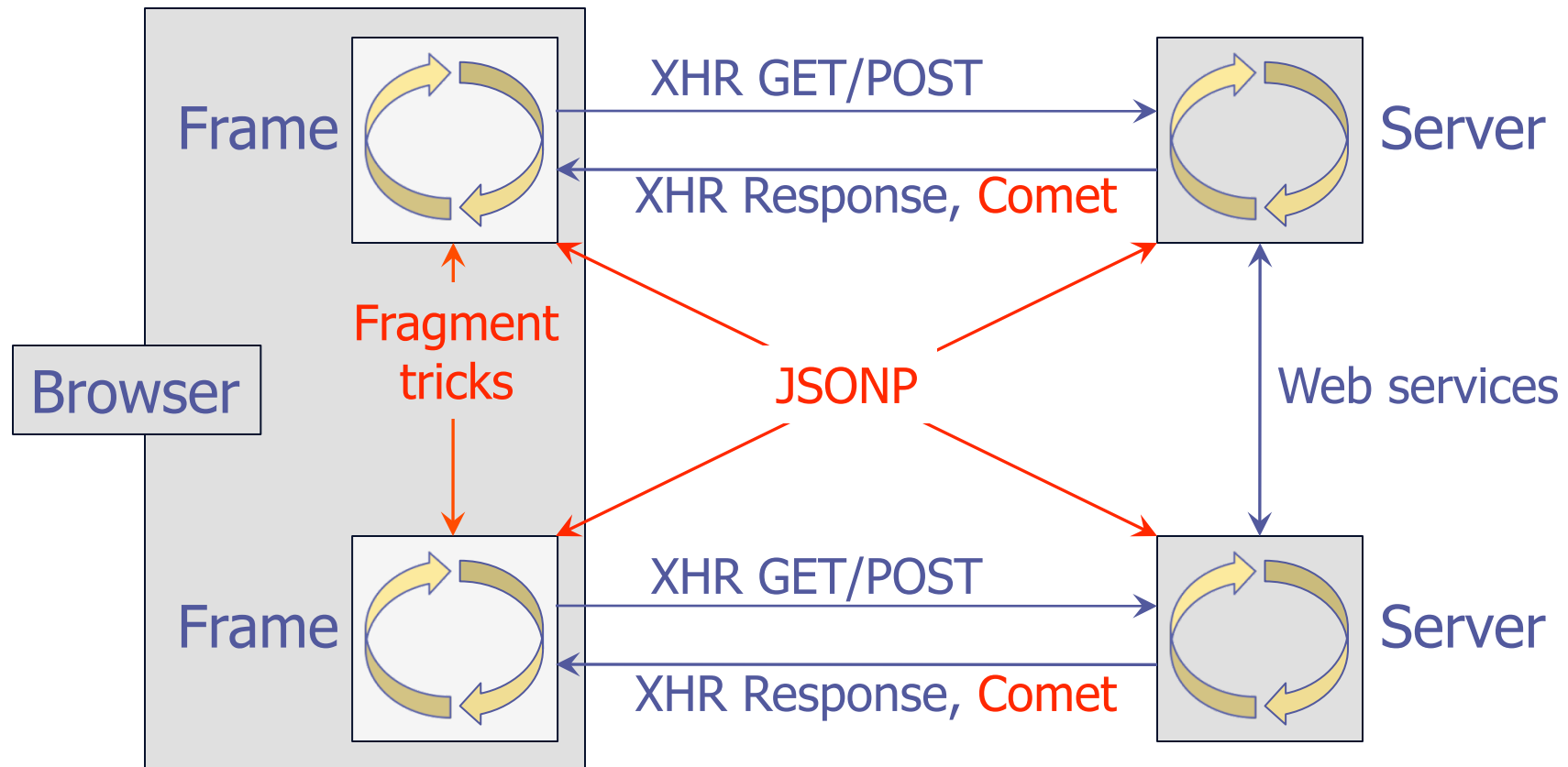
Original Web



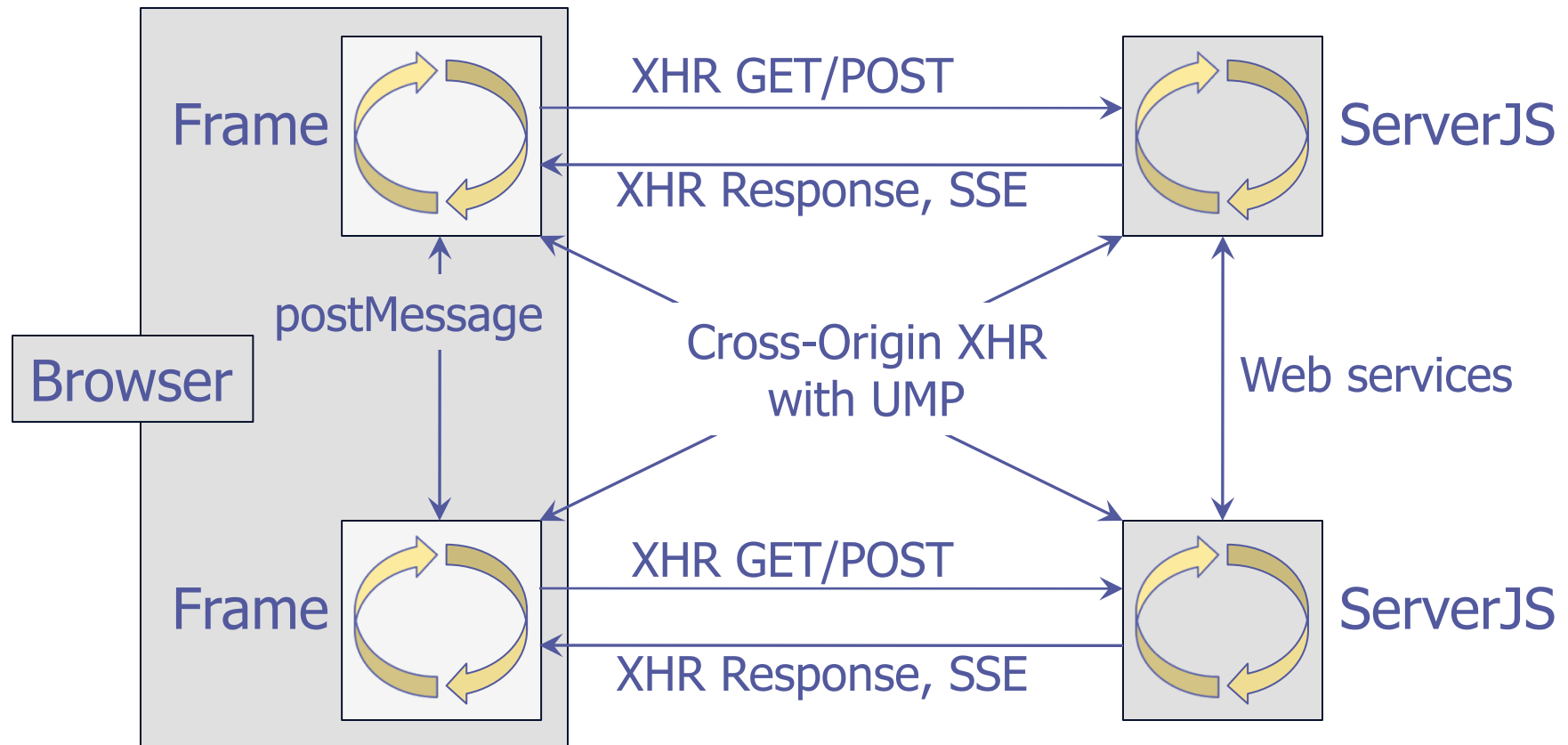
Ajax = Mobile code + async msgs



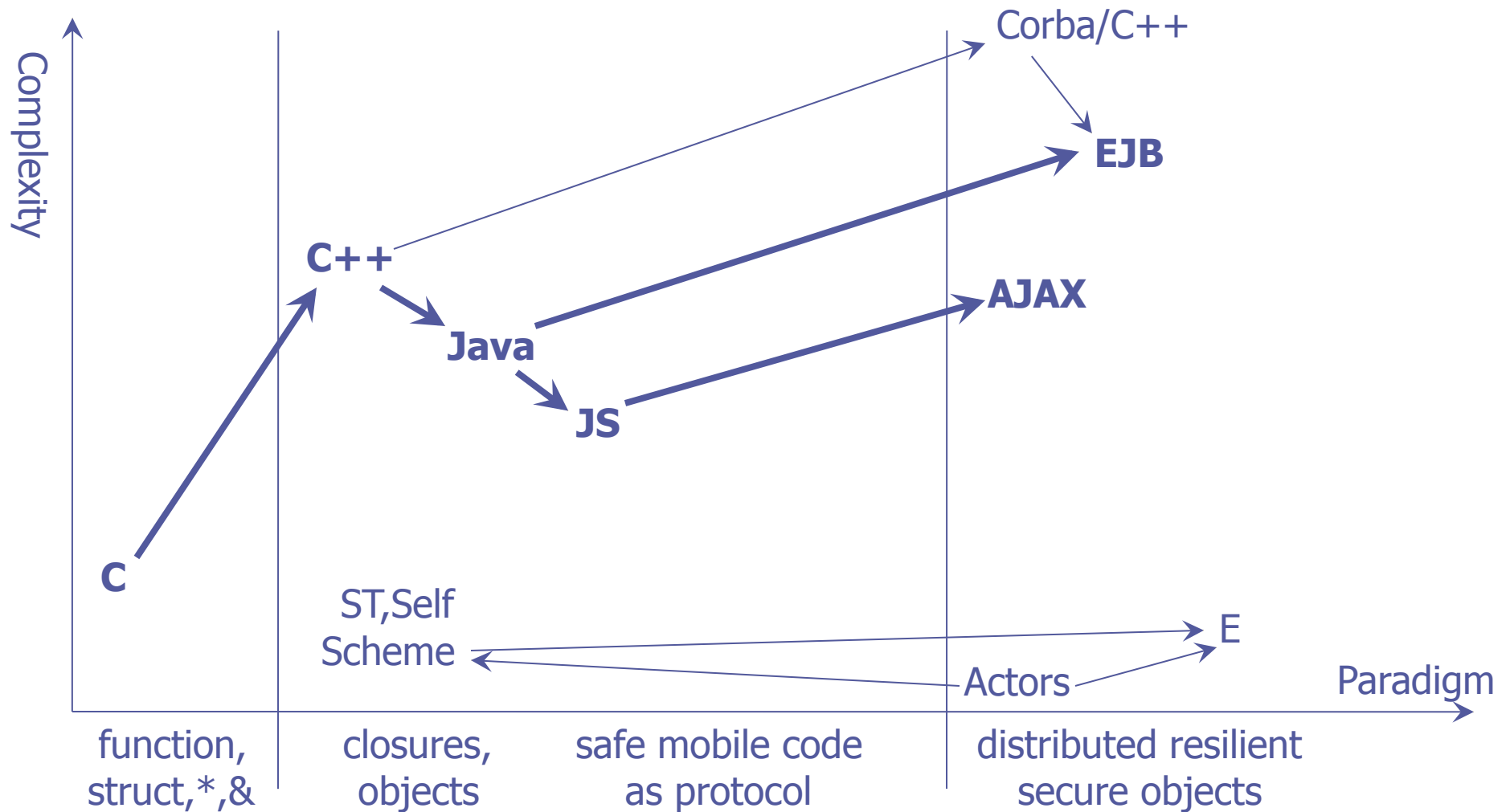
Kludging Towards Distributed Objects



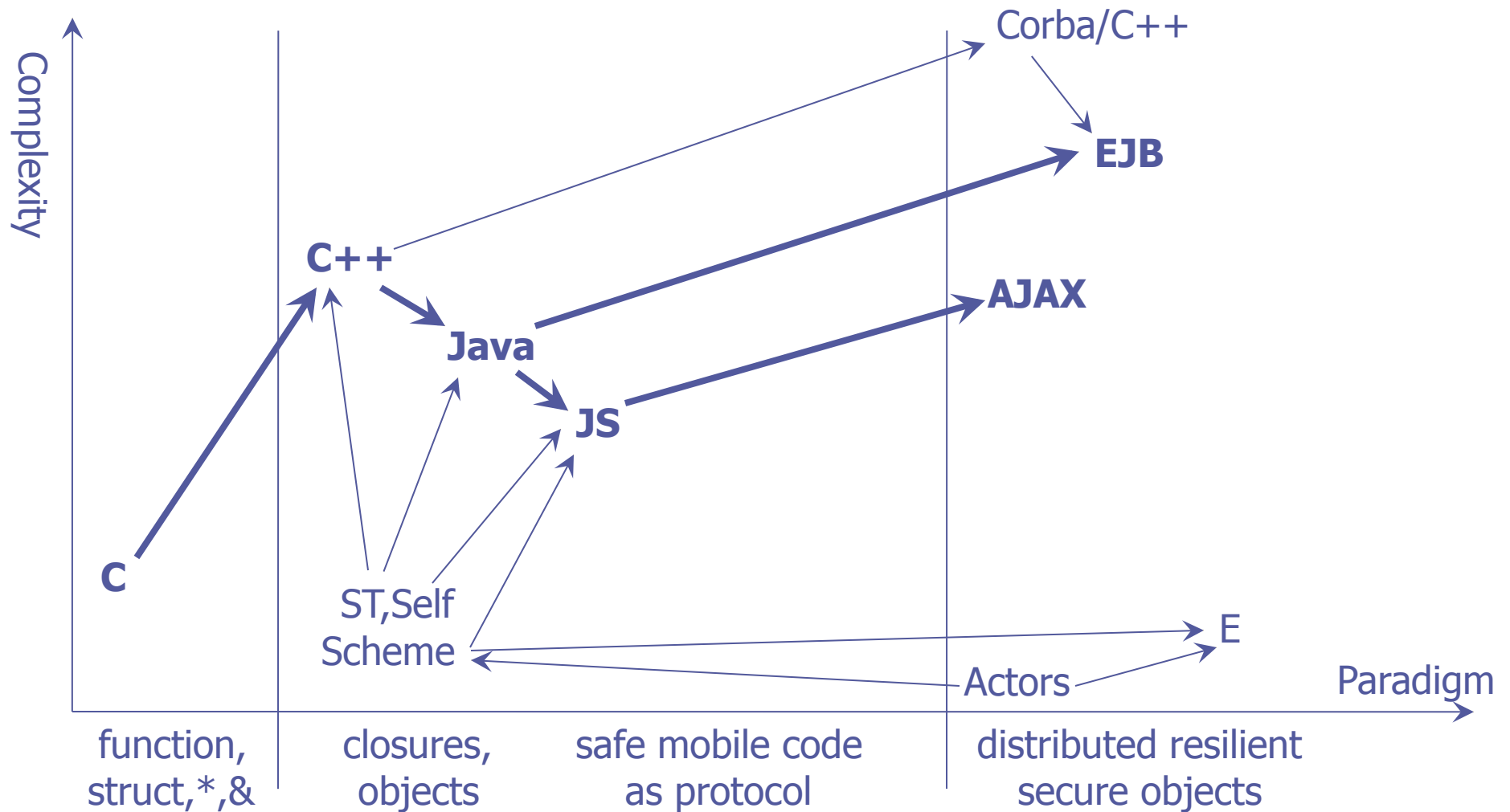
A Web of Distributed Objects



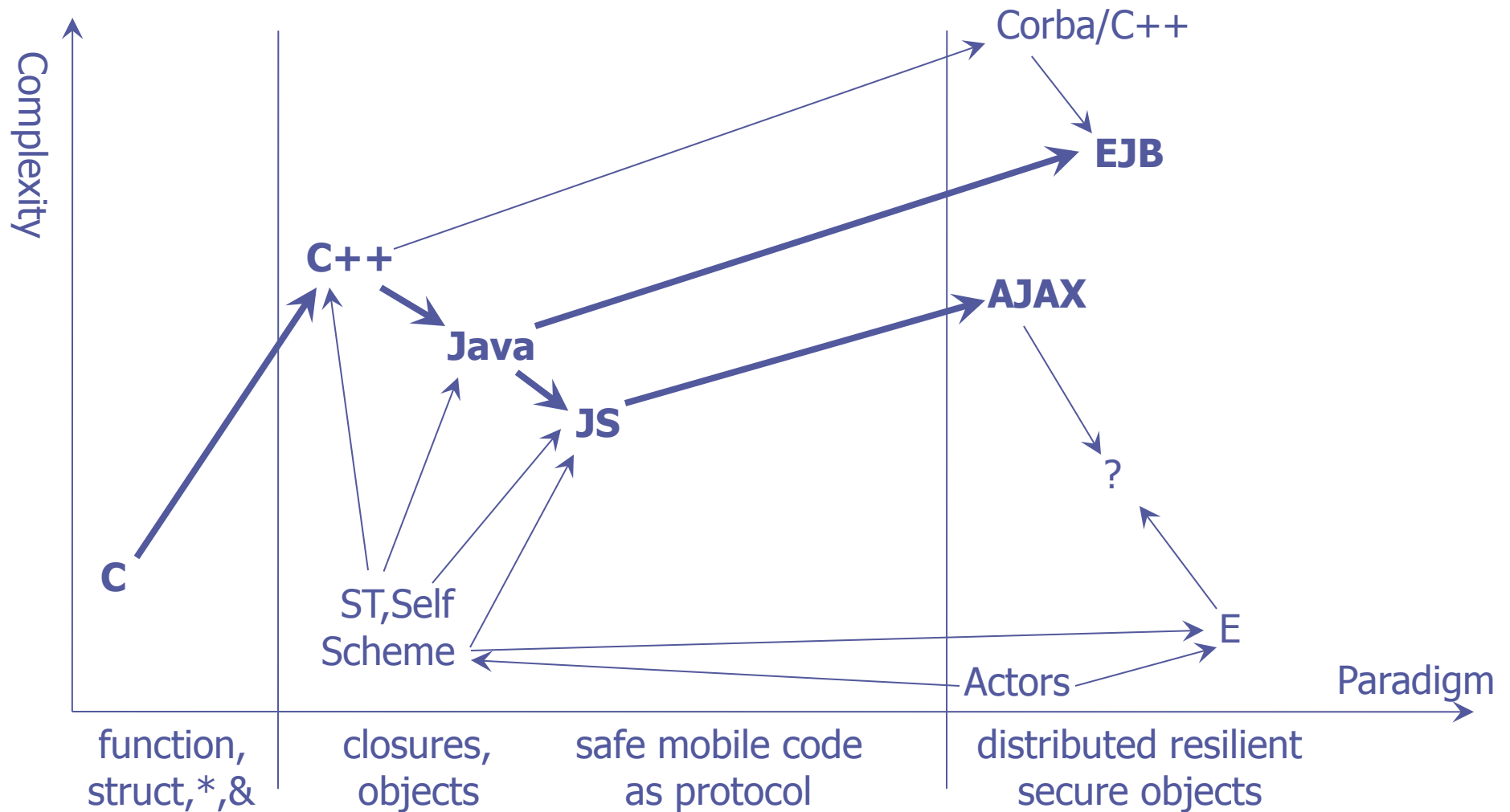
Adoption paths & progress



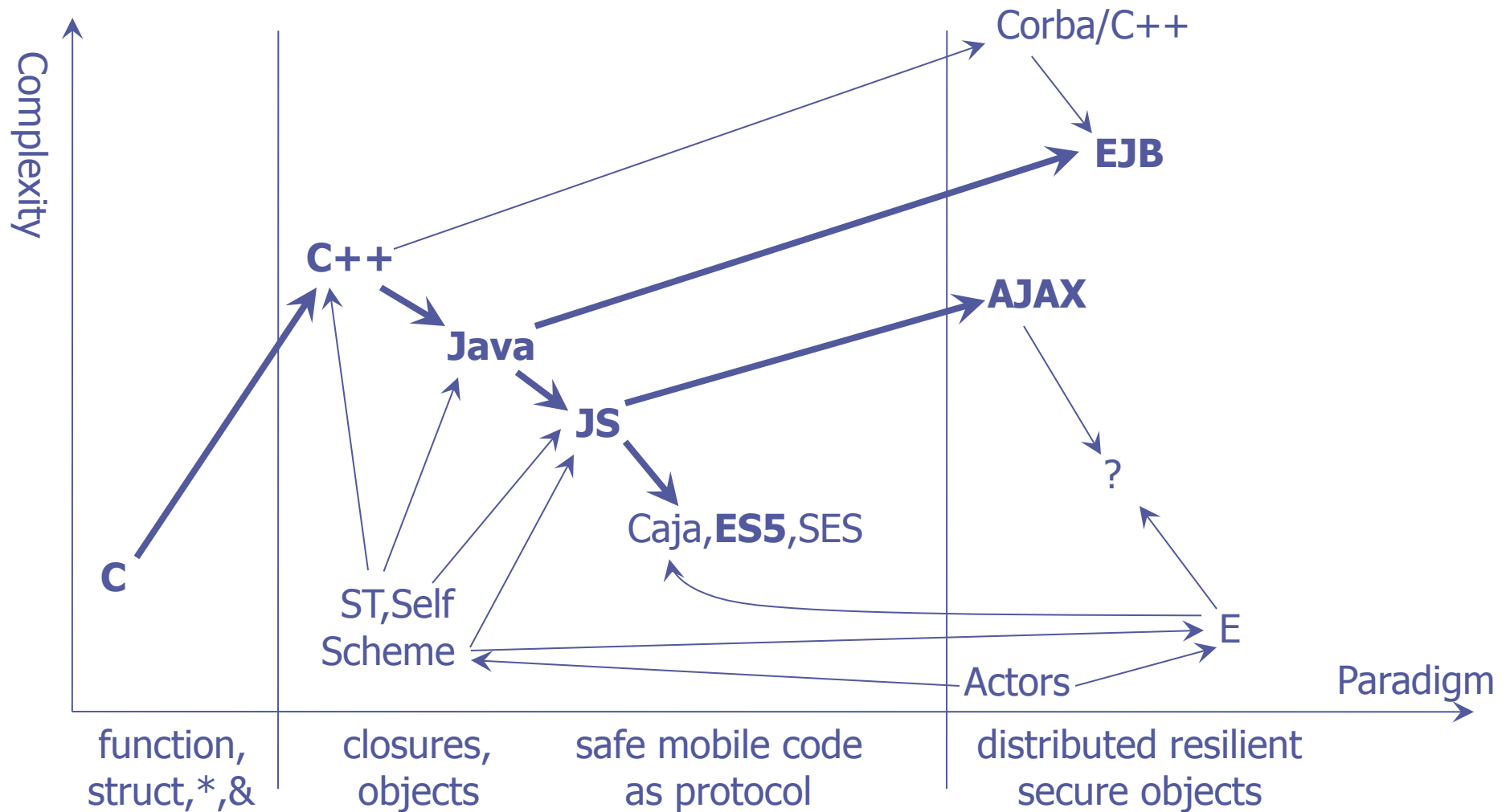
Legacy-free scouts lead way



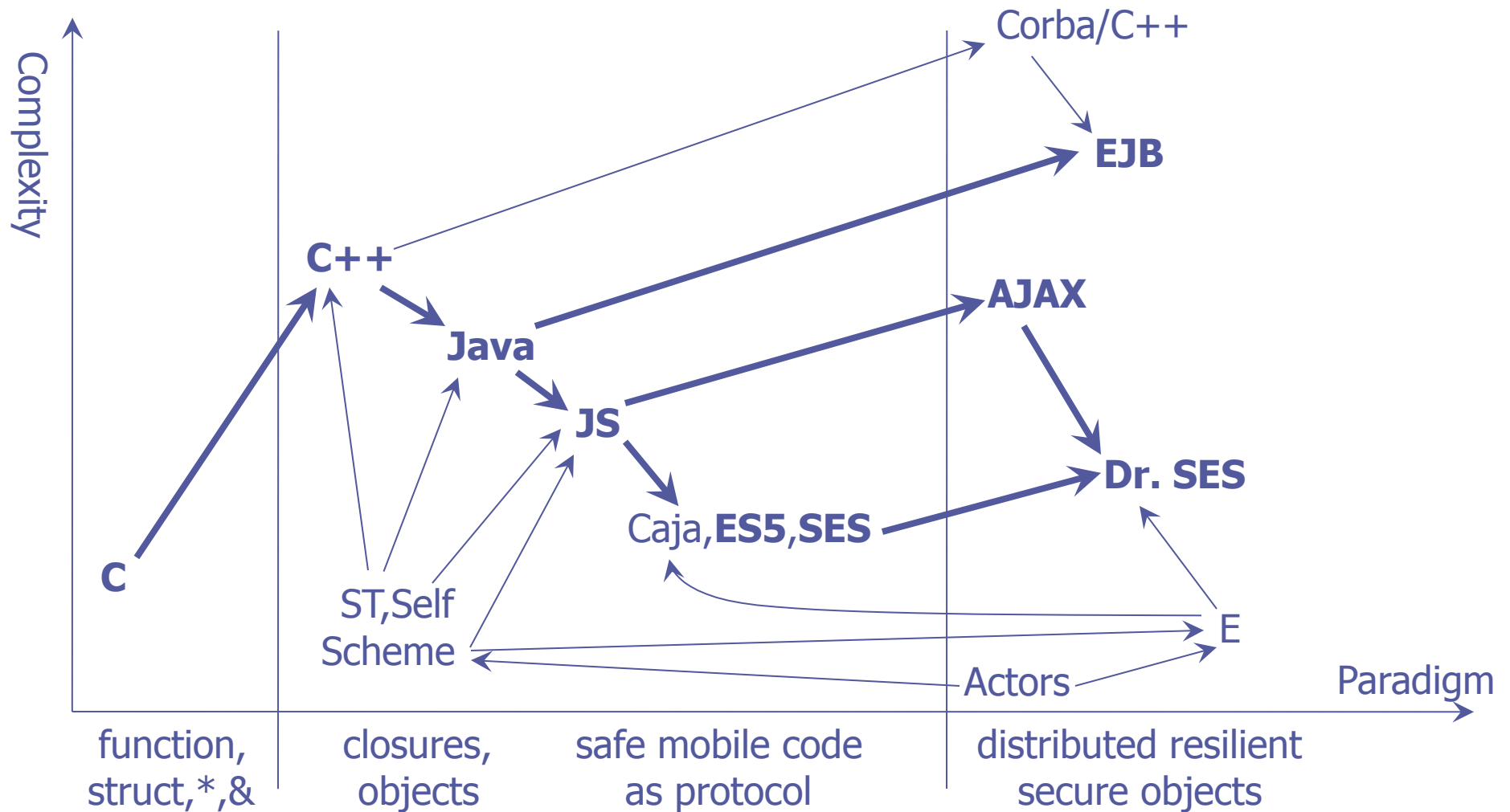
Too big a jump



Today



Tomorrow?



OCaps: Small step from pure objects

- Memory safety and encapsulation
 - + Effects **only** by using held references
 - + No powerful references by default
-
- Reference graph === Access graph

Caja solves the safe mashup problem

Deployed on Yahoo! homepage, Google, Shindig, ...



Corkboard demo

```
<html> <head> <title>Basic Mashup</title>
<script>
  function animate(id) {
    var element = document.getElementById(id);
    var textNode = element.childNodes[0];
    var text = textNode.data;
    var reverse = false;
    element.onclick = function() { reverse = !reverse; };
    setInterval(function() {
      textNode.data = text = reverse ? text.substring(1) + text[0]
        : text[text.length-1] + text.substring(0, text.length-1);
    }, 100);
  }
</script>
</head> <body onload="animate('target')">
  <pre id="target">Hello Programmable World! </pre>
</body> </html>
```



Distributed Concurrency?

	Shared State	Message Passing
Blocking	C++/pthreads Java, C#, Mozart/Oz JoCAML, Polyphonic C#	<u>Blocking receive</u> CSP, Occam, CCS Erlang, Scala, Go
Non-blocking	<u>Soft Transactional Mem</u> Argus, Fortress Clojure, X10	<u>Comm Event Loops</u> Actors, AmbientTalk E, Waterken Ajax



Distributed Defensive Concurrency

	Shared State	Message Passing
Blocking	C++/pthreads Java, C#, Mozart/Oz JoCAML, Polyphonic C#	<u>Blocking receive</u> CSP, Occam, CCS Erlang, Scala, Go
Non-blocking	<u>Soft Transactional Mem</u> Argus, Fortress Clojure, X10	<u>Comm Event Loops</u> Actors, AmbientTalk E, Waterken Ajax

No conventional deadlocks or memory races



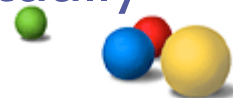
Distributed Secure Concurrency

	Shared State	Message Passing
Blocking	C++/pthreads Java, C#, Mozart/Oz JoCAML, Polyphonic C#	<i>Blocking receive</i> CSP, Occam, CCS Erlang, Scala, Go
Non-blocking	<i>Soft Transactional Mem</i> Argus, Fortress Clojure, X10	<i>Comm Event Loops</i> Actors, AmbientTalk E, Waterken Ajax, Dr. SES

No conventional deadlocks or memory races

`var good = bob.buy(desc, payment);` `// do it immediately`

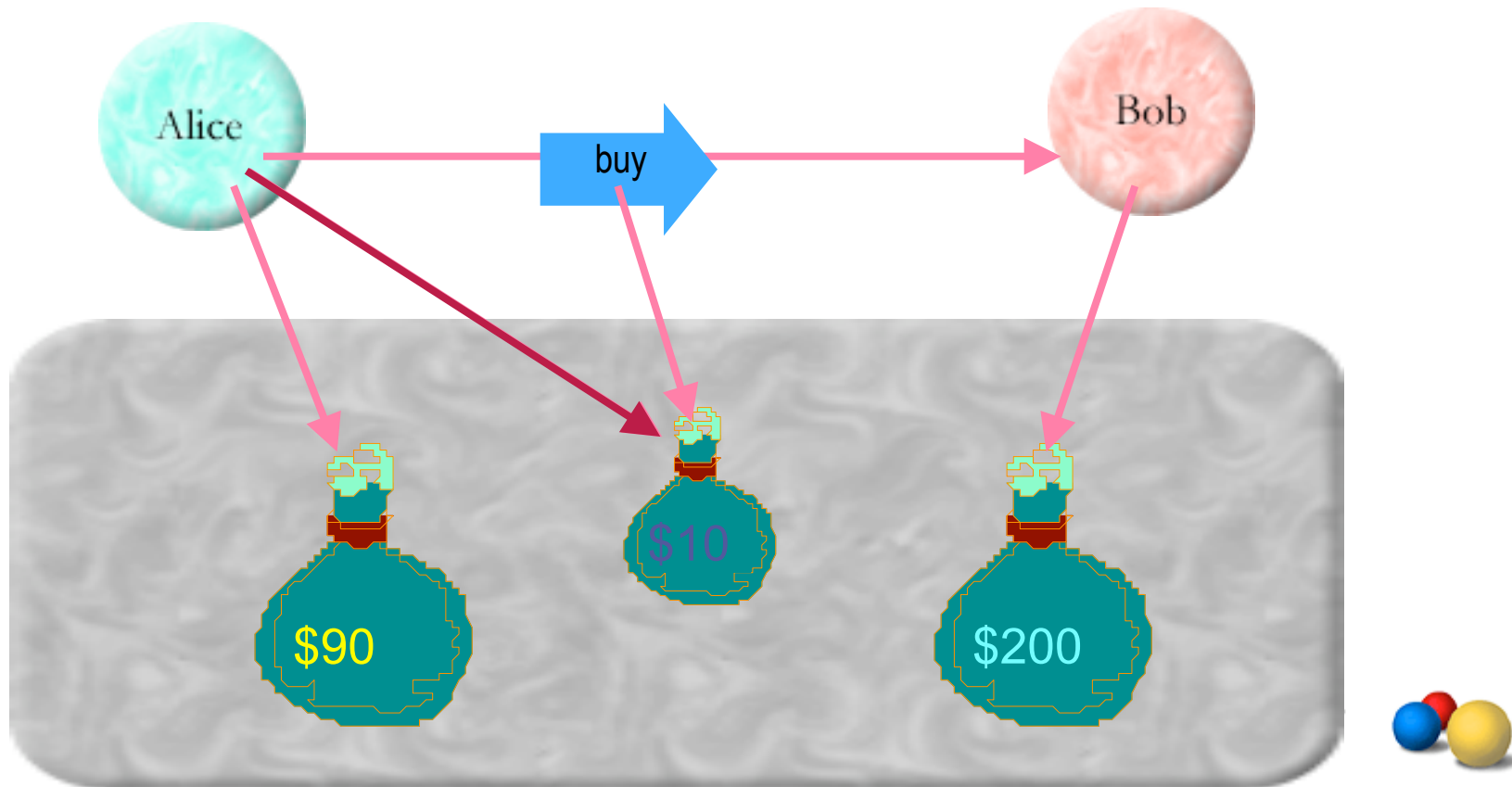
`var goodP = bobP ! buy(desc, payment);` `// do it eventually`



Distributed Secure Currency

```
const paymentP = myPurse ! makePurse();  
paymentP ! deposit(10, myPurse);  
const goodP = bobP ! buy(desc, paymentP);
```

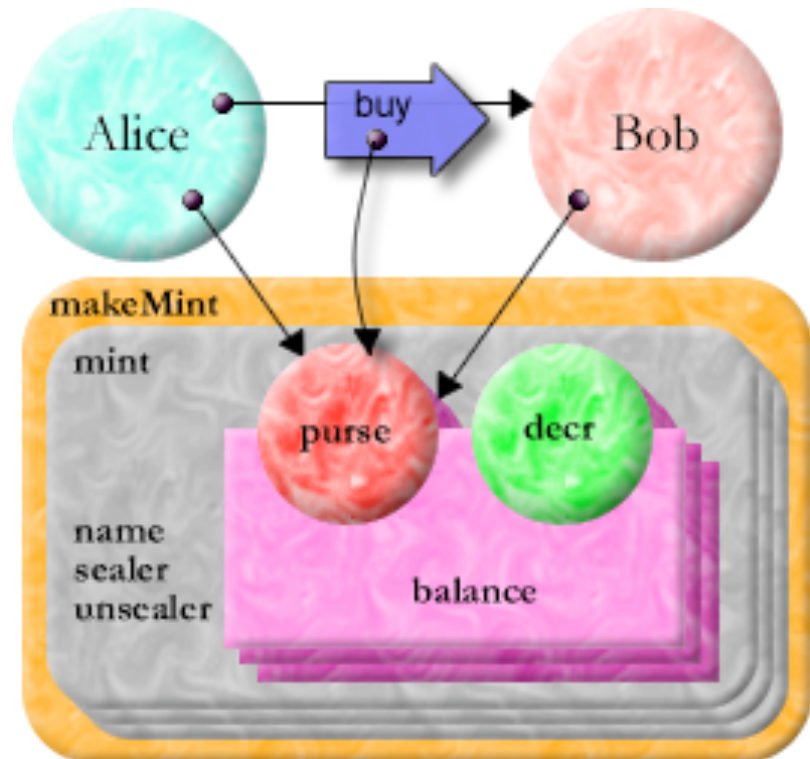
```
return Q.when(paymentP, const(p) {  
  return Q.when(myPurse ! deposit(10, p), const(_) {  
    return good; }, ...
```



Money as “factorial” of secure coding

```
const makeMint() {  
  const decr = WeakMap();  
  const mint(balance :: Nat) {  
    const purse = Q.def({  
      getBalance: const() { return balance; },  
      makePurse: const() { return mint(0); },  
      deposit: const(amount :: Nat, src) {  
        const newBal :: Nat = balance + amount;  
        decr.get(src)(amount);  
        balance = newBal;  
      }  
    });  
    decr.set(purse, const(amount) {  
      balance = balance - amount;  
    });  
    return purse;  
  }  
  return mint;  
}
```

// No explicit crypto
// Compares well with E



Questions?



Change JavaScript's Trajectory

ES3 core idea: records and closures

A record of closures → object with methods



Change JavaScript's Trajectory

ES3 core idea: records and closures

A record of closures → object with methods

Took us **2 years** to secure by server translation

ES5 ratified 12/09. All browsers implementing

Object.freeze(obj) → stable records

"use strict"; → lexical scope + encapsulation

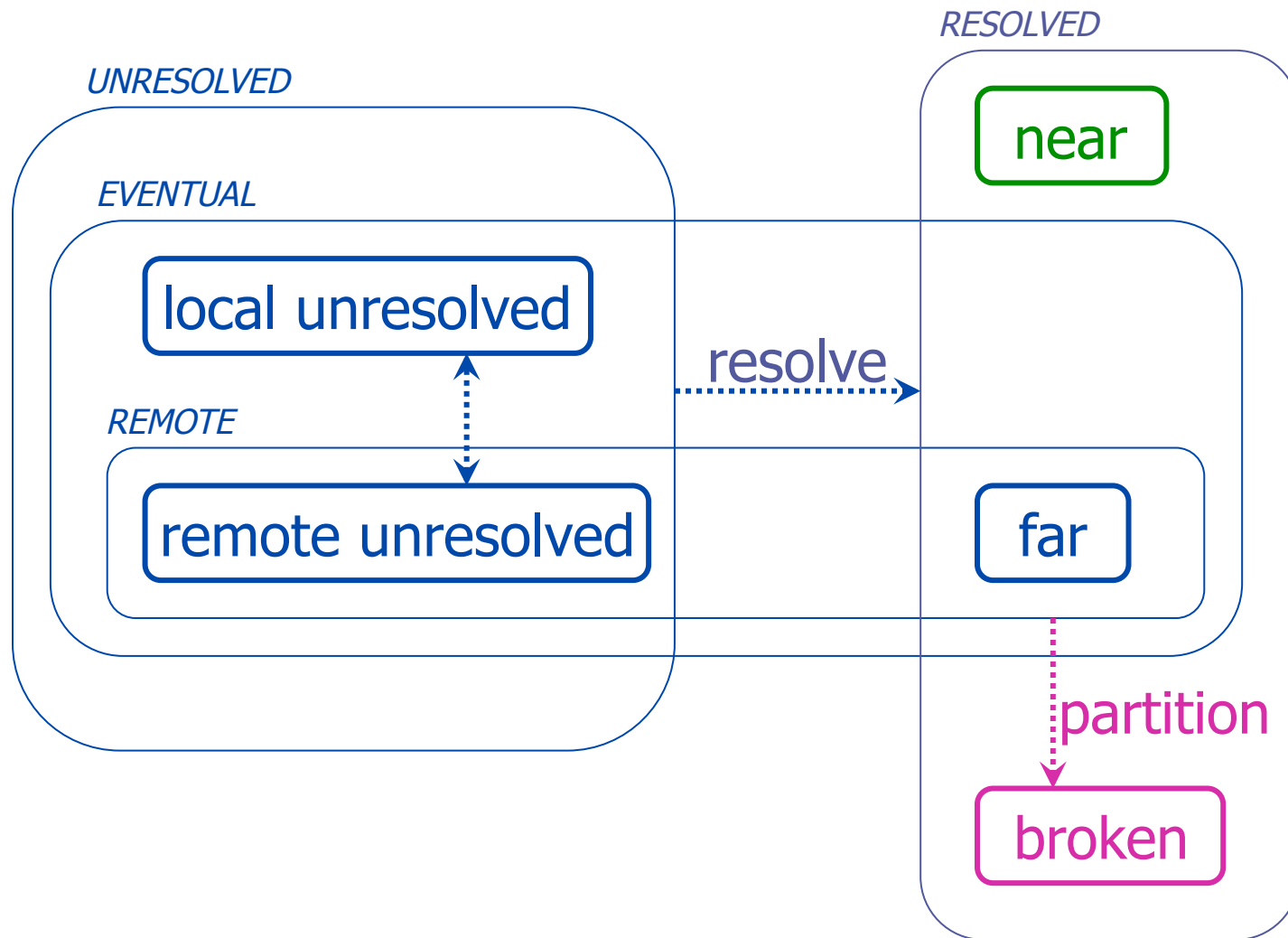
A stable record of encapsulated closures

→ a defensible object

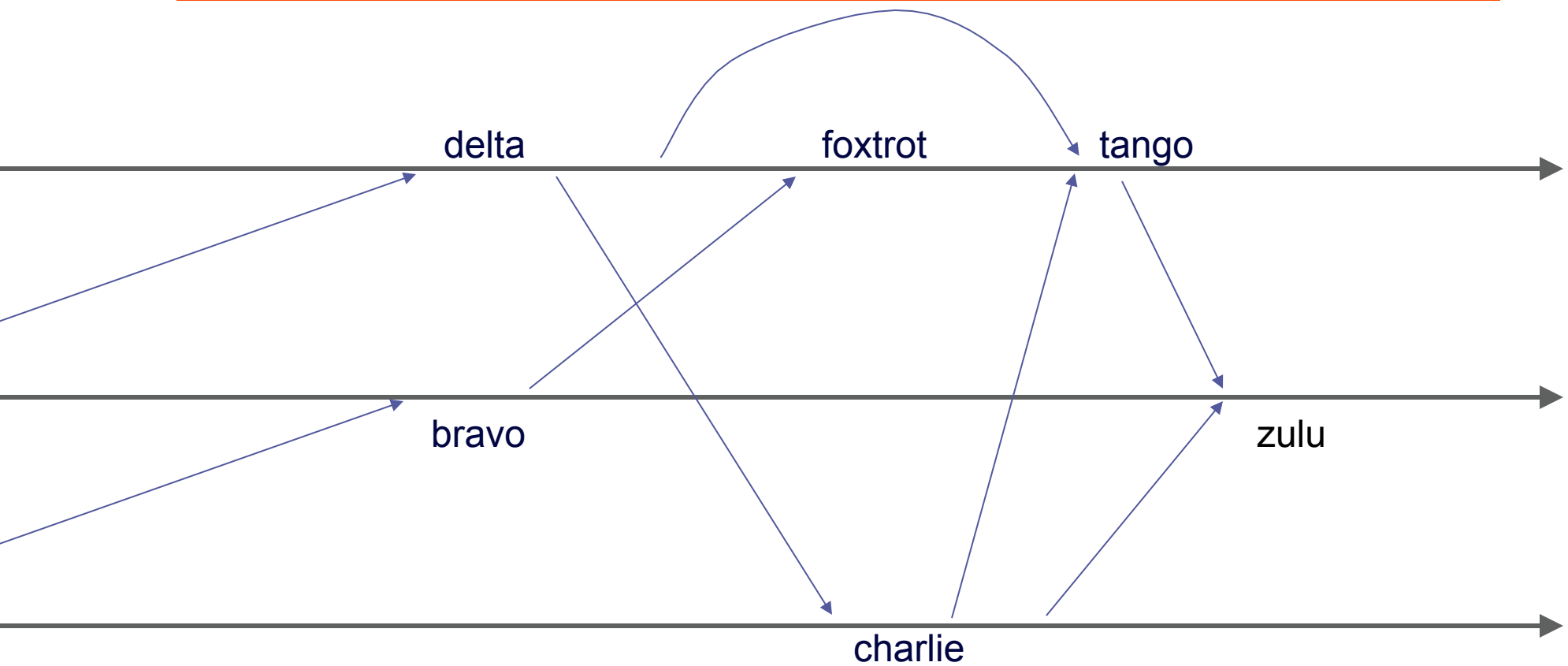
Took us **2 afternoons** to secure by client verification



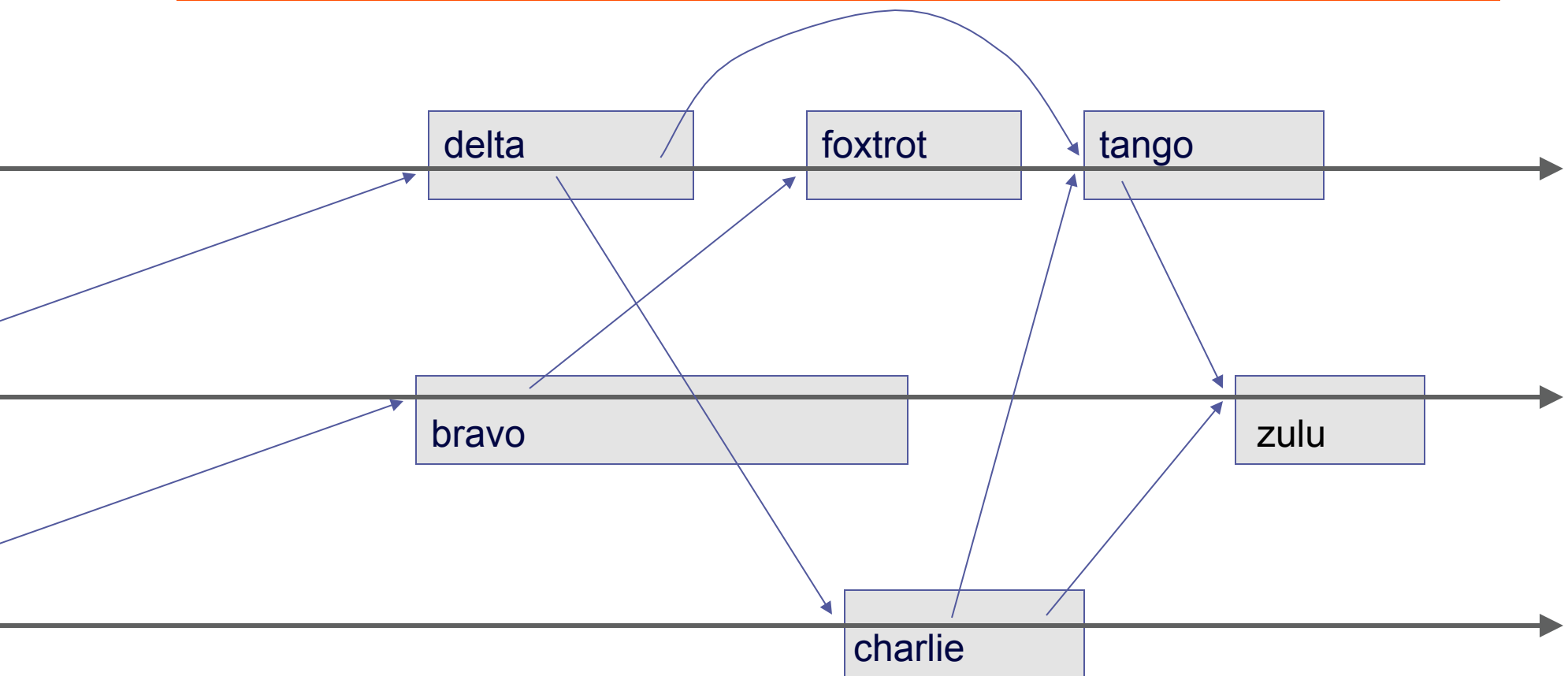
Promise states and transitions



Lamport's distributed causality



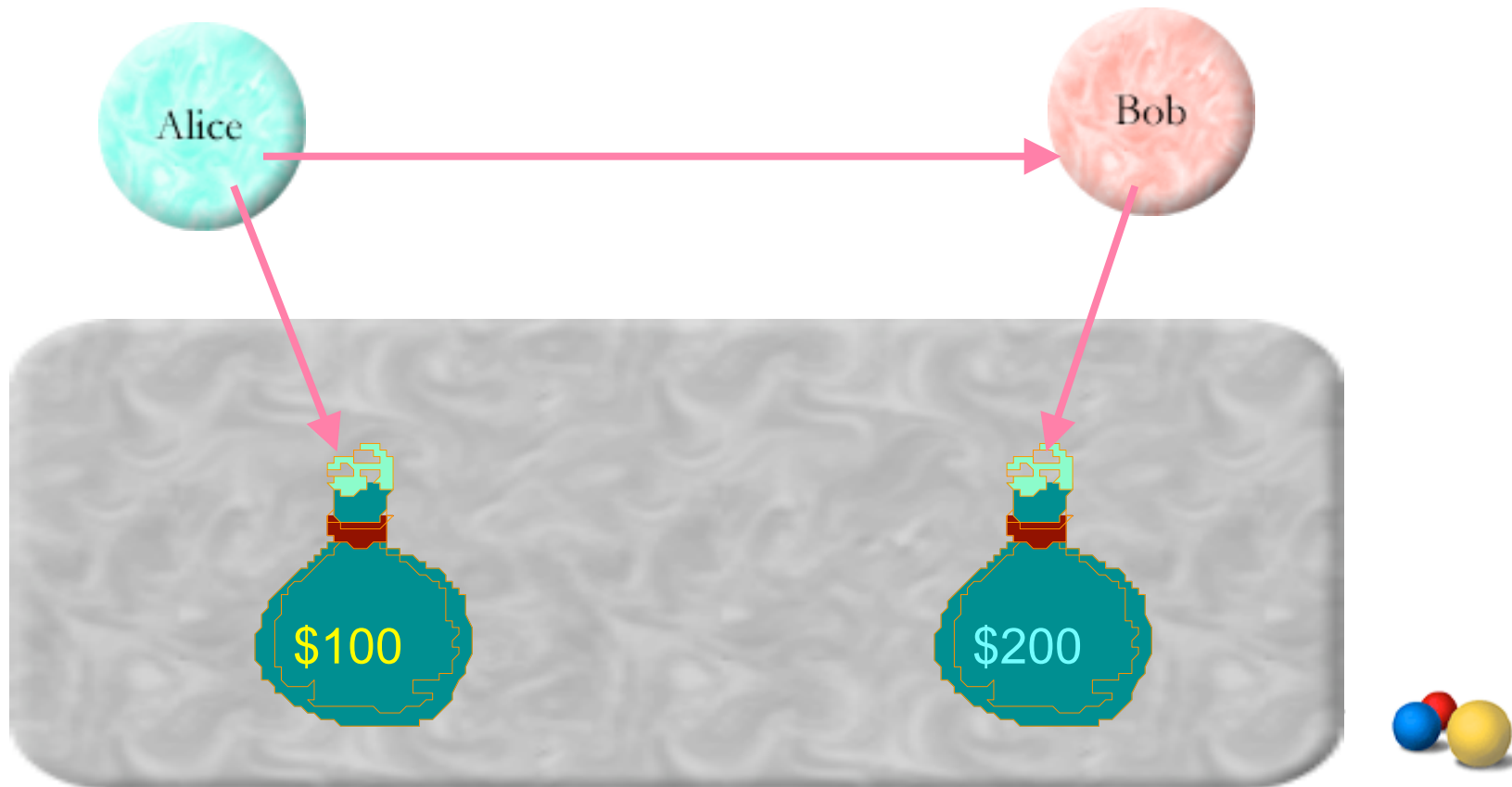
Communicating Event Loops



Turns are isolated units of operation

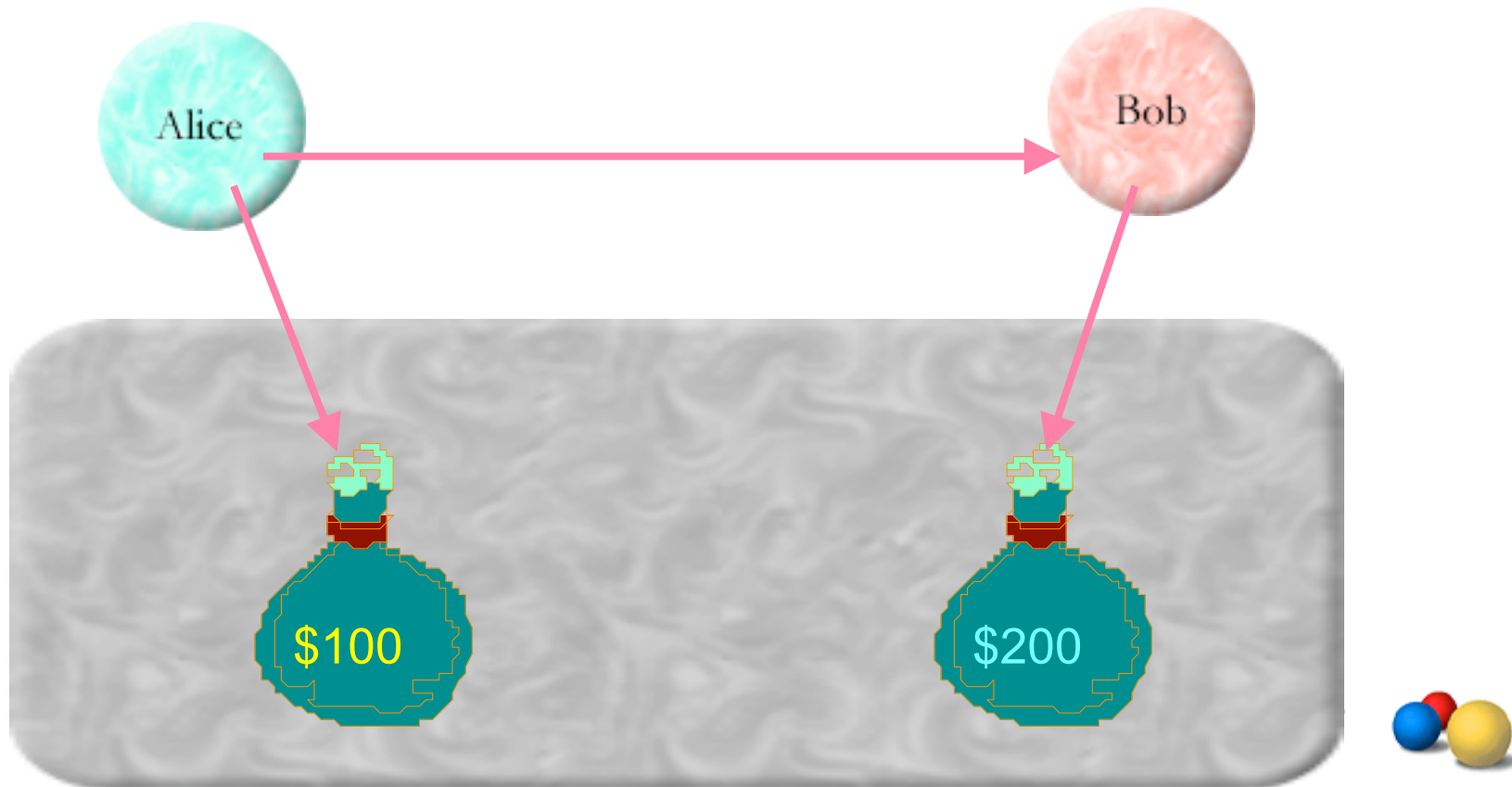


Distributed Secure Currency



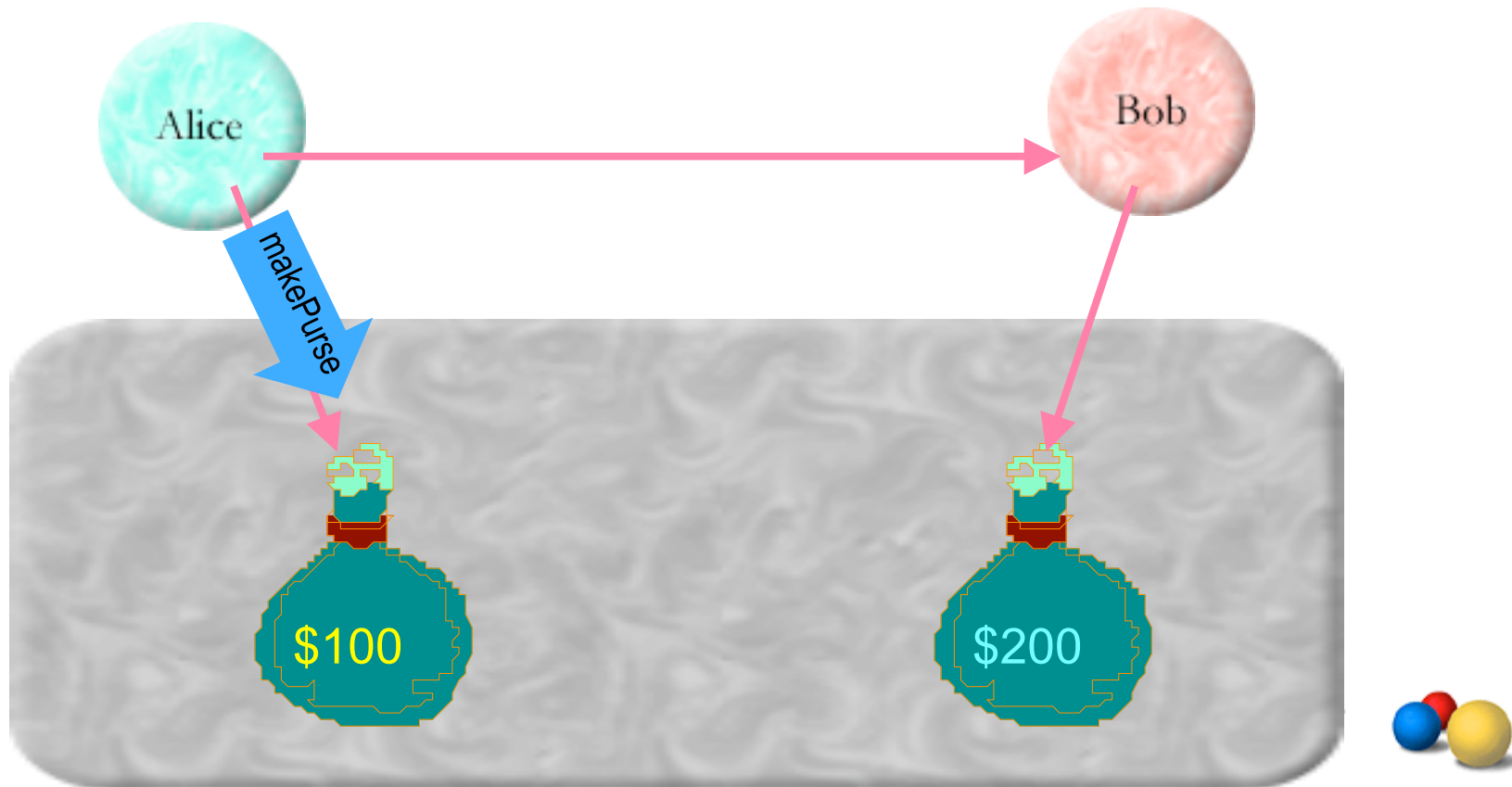
Distributed Secure Currency

```
const paymentP = myPurse ! makePurse();
```



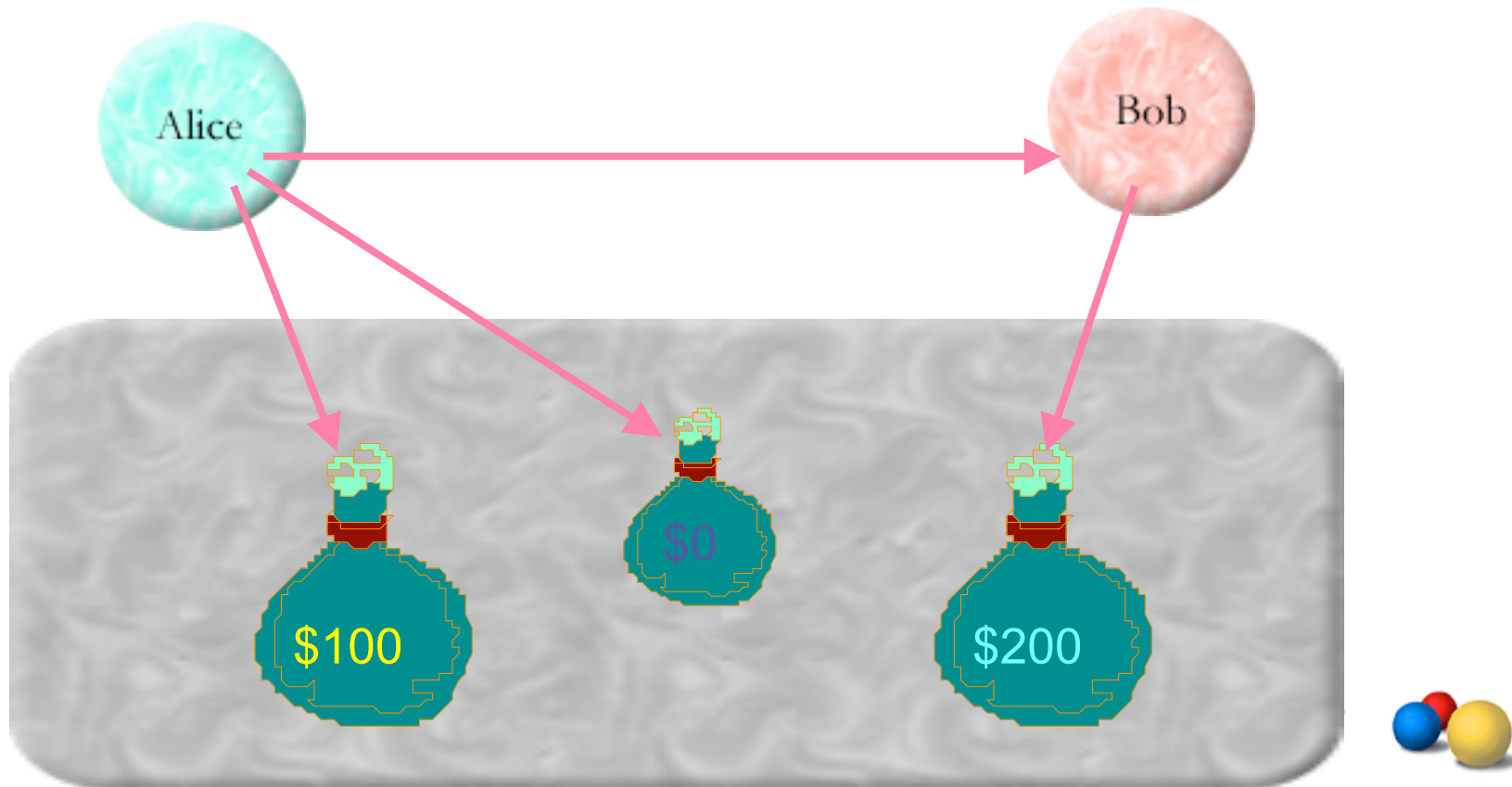
Distributed Secure Currency

```
const paymentP = myPurse ! makePurse();
```



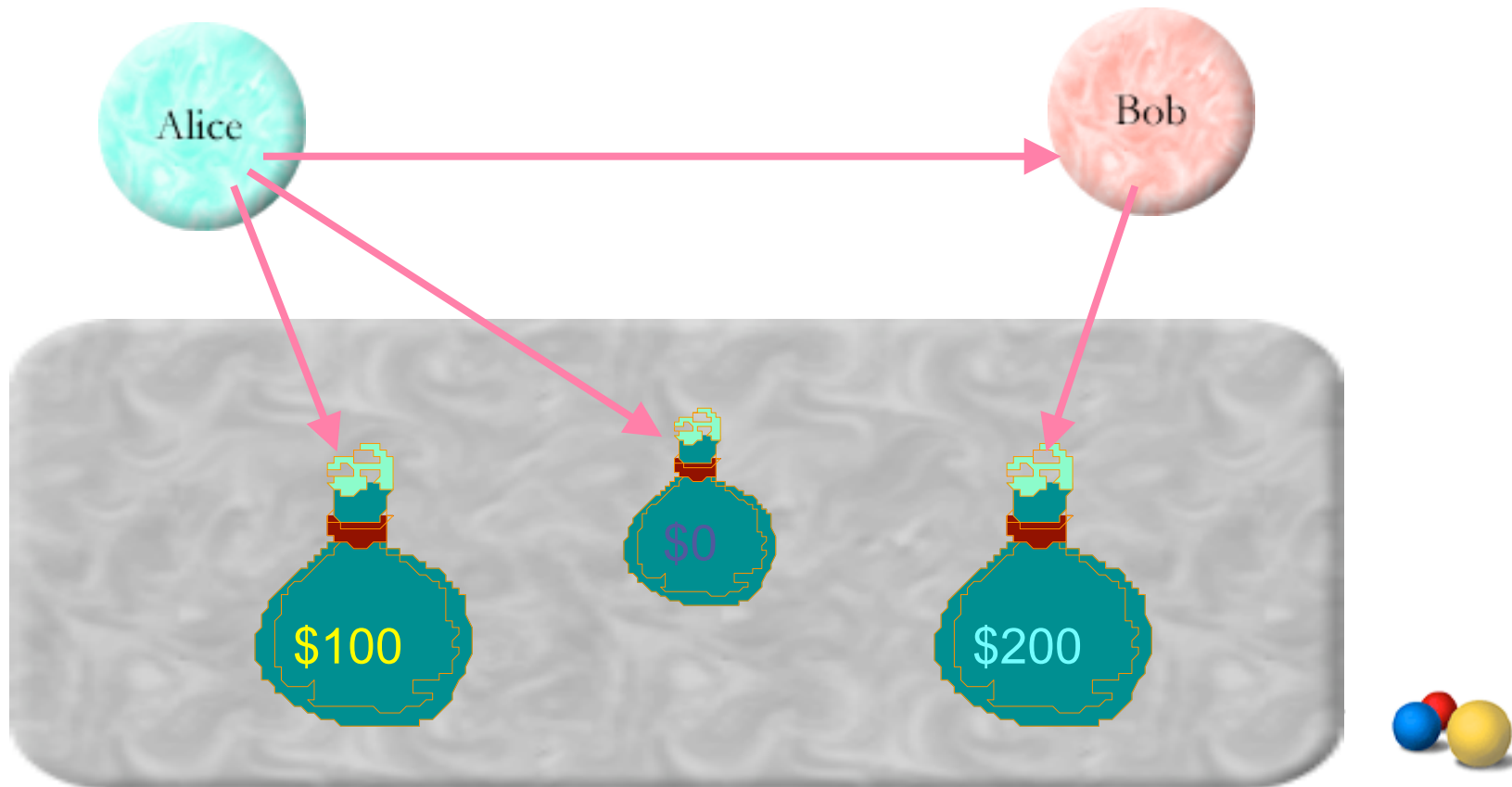
Distributed Secure Currency

```
const paymentP = myPurse ! makePurse();
```



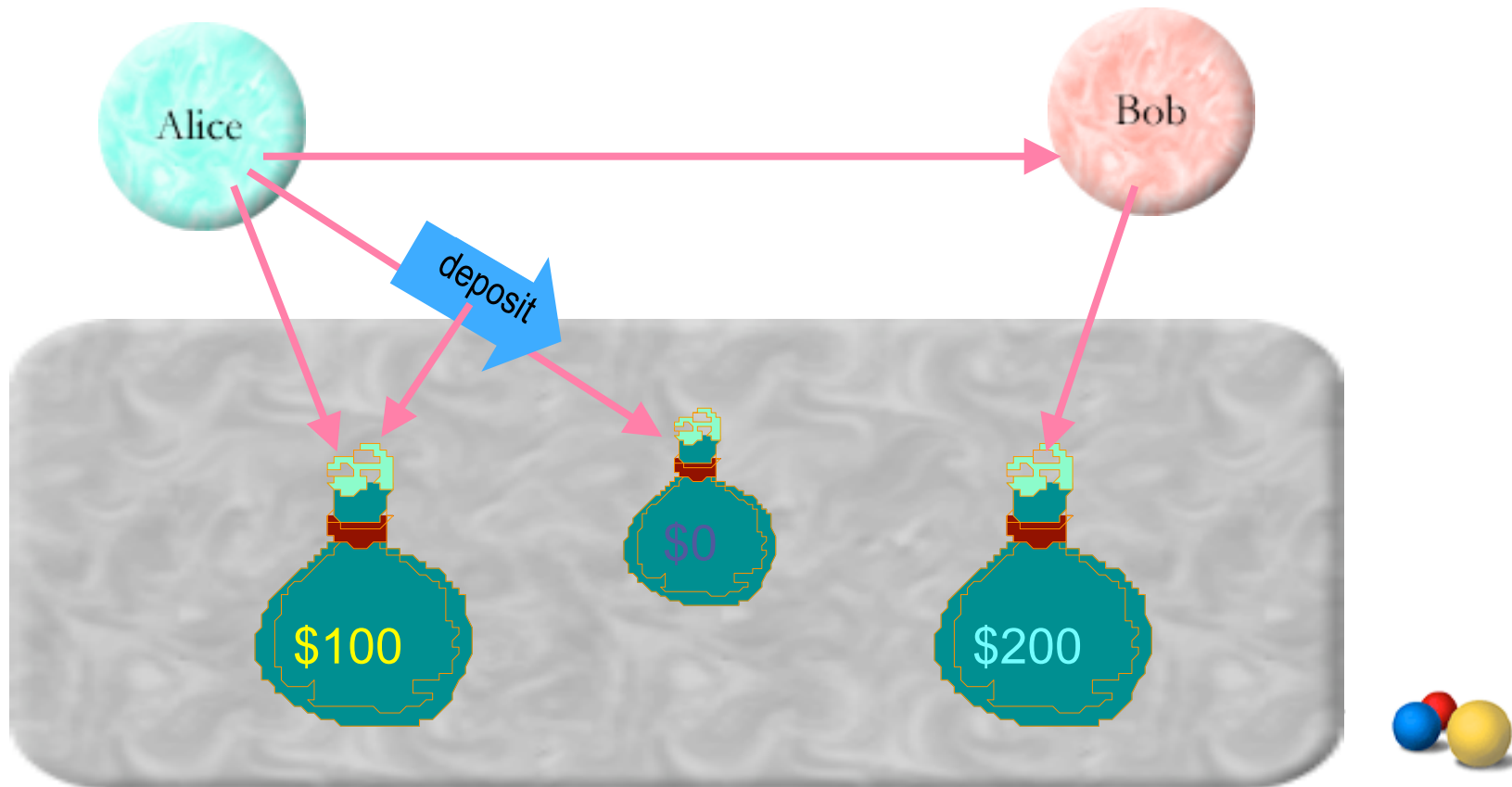
Distributed Secure Currency

```
const paymentP = myPurse ! makePurse();  
paymentP ! deposit(10, myPurse);
```



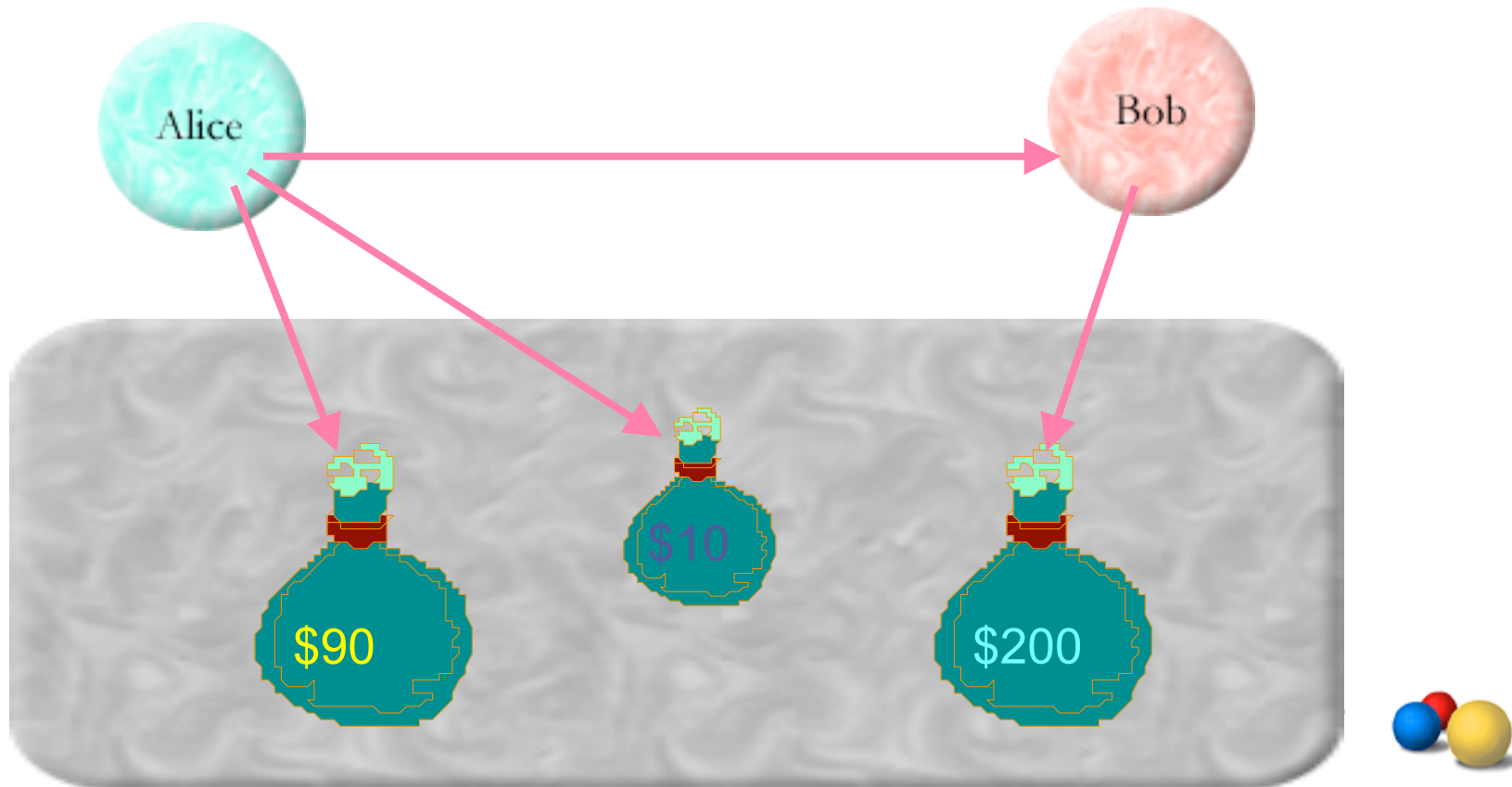
Distributed Secure Currency

```
const paymentP = myPurse ! makePurse();  
paymentP ! deposit(10, myPurse);
```



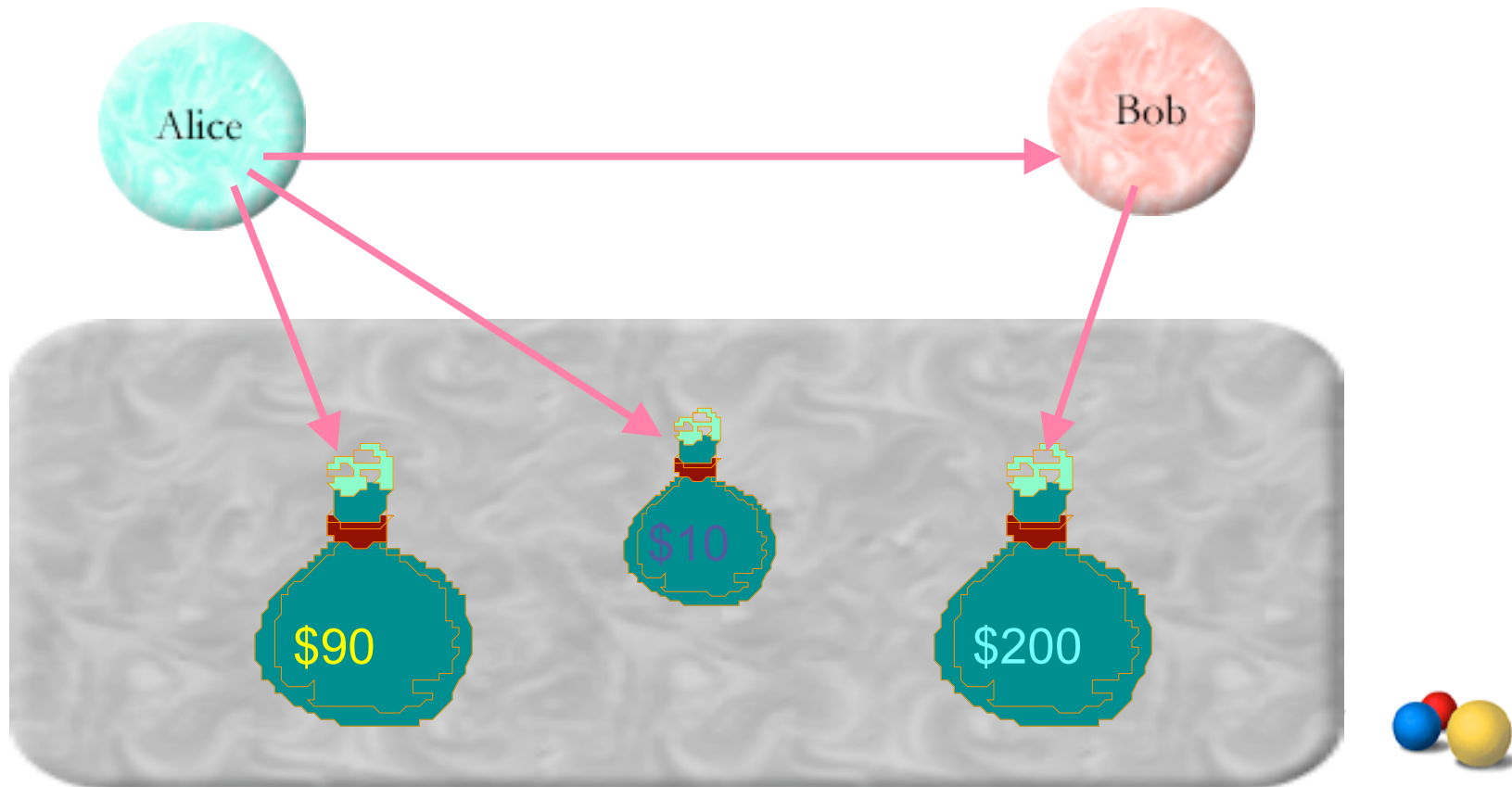
Distributed Secure Currency

```
const paymentP = myPurse ! makePurse();  
paymentP ! deposit(10, myPurse);
```



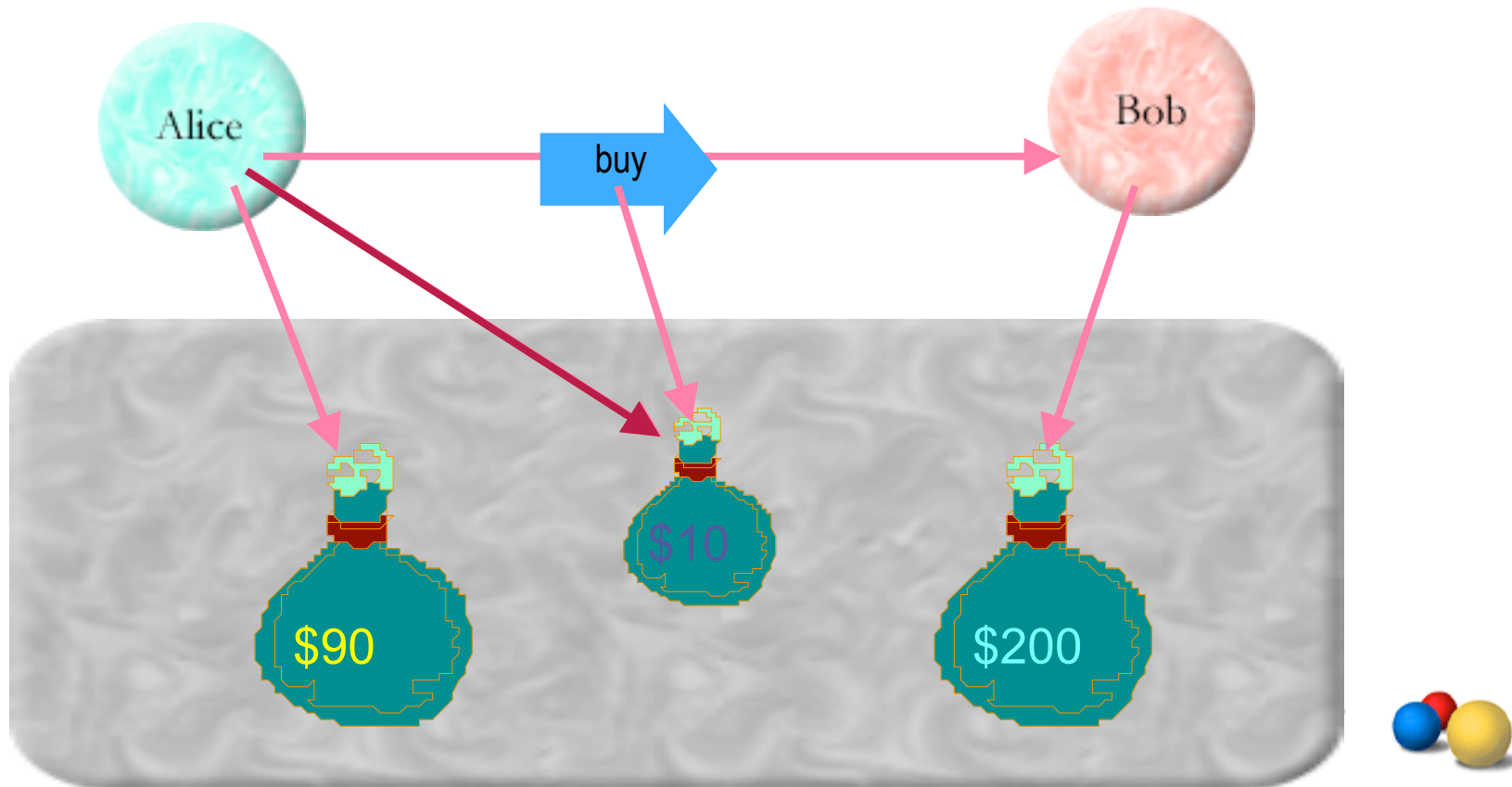
Distributed Secure Currency

```
const paymentP = myPurse ! makePurse();  
paymentP ! deposit(10, myPurse);  
const goodP = bobP ! buy(desc, paymentP);
```



Distributed Secure Currency

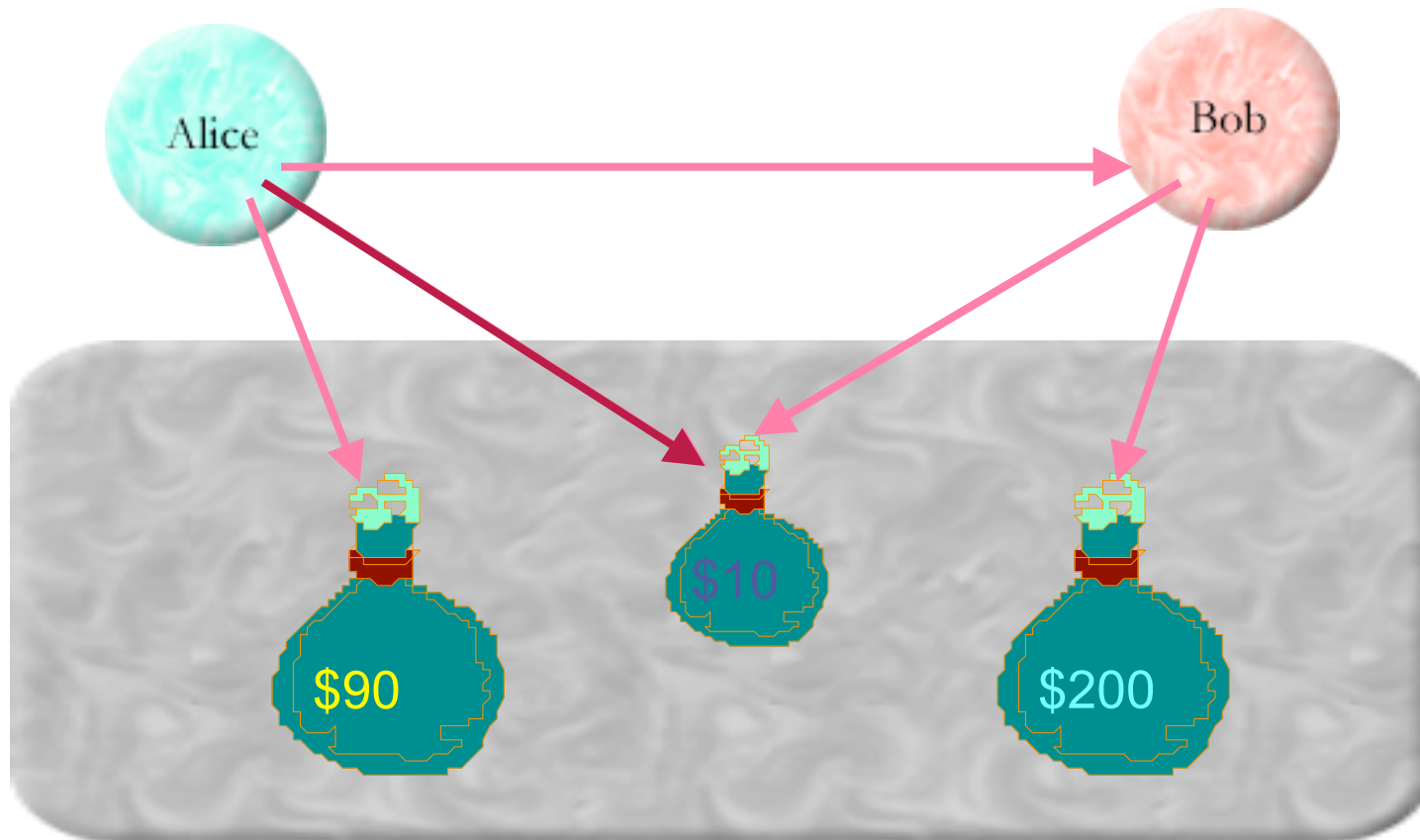
```
const paymentP = myPurse ! makePurse();  
paymentP ! deposit(10, myPurse);  
const goodP = bobP ! buy(desc, paymentP);
```



Distributed Secure Currency

```
const paymentP = myPurse ! makePurse();  
paymentP ! deposit(10, myPurse);  
const goodP = bobP ! buy(desc, paymentP);
```

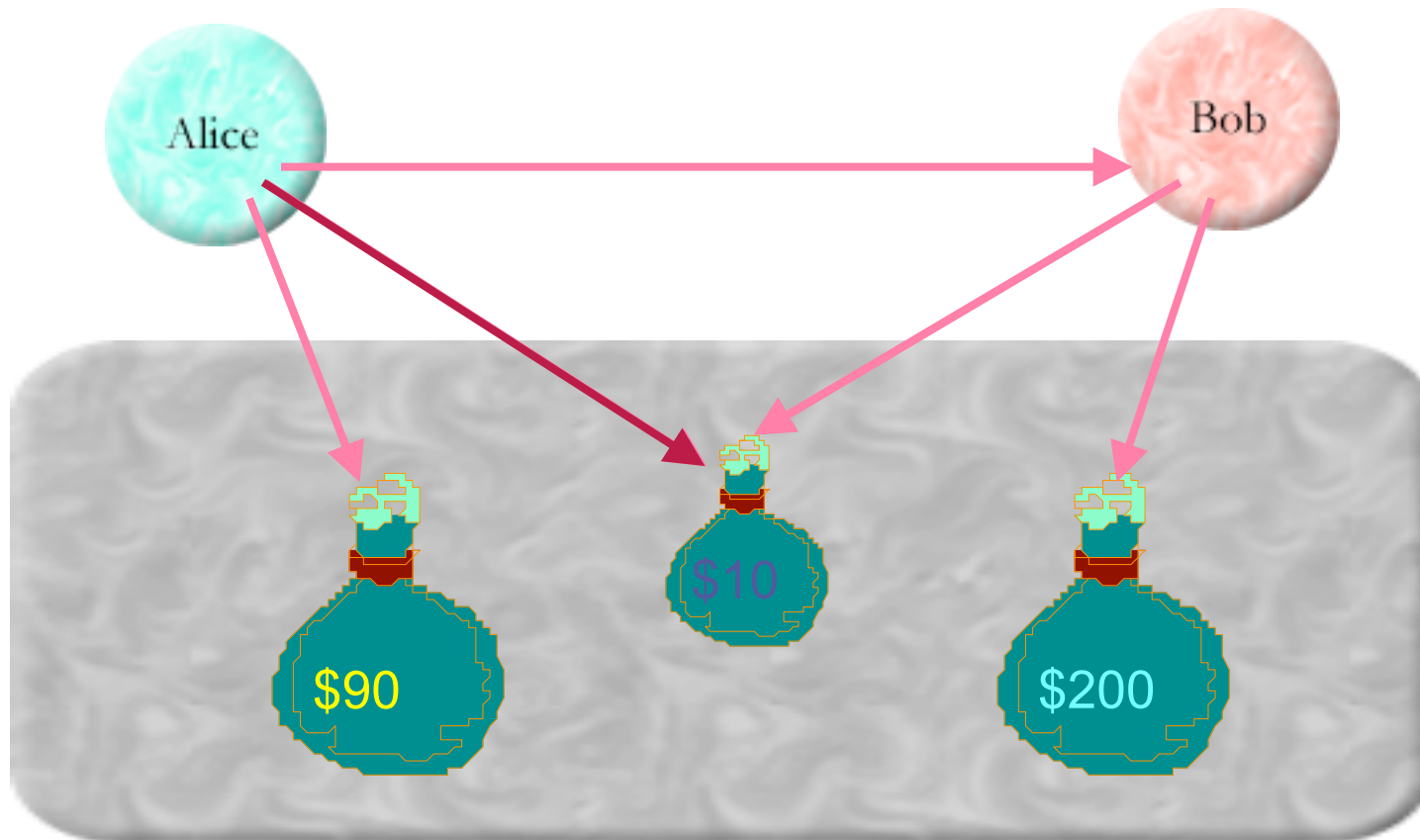
```
return Q.when(paymentP, const(p) {
```



Distributed Secure Currency

```
const paymentP = myPurse ! makePurse();  
paymentP ! deposit(10, myPurse);  
const goodP = bobP ! buy(desc, paymentP);
```

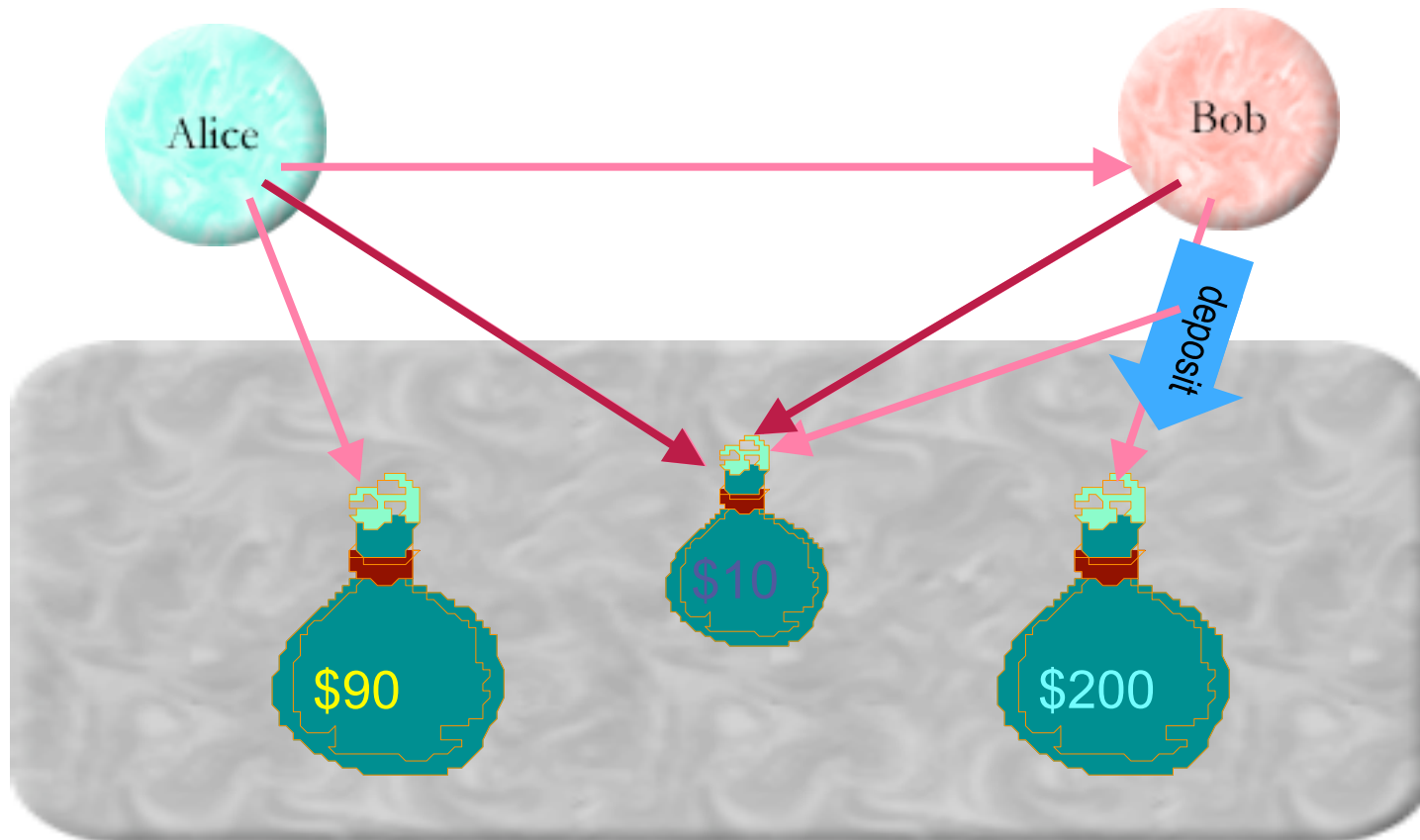
```
return Q.when(paymentP, const(p) {  
  return Q.when(myPurse ! deposit(10, p), const(_) {
```



Distributed Secure Currency

```
const paymentP = myPurse ! makePurse();  
paymentP ! deposit(10, myPurse);  
const goodP = bobP ! buy(desc, paymentP);
```

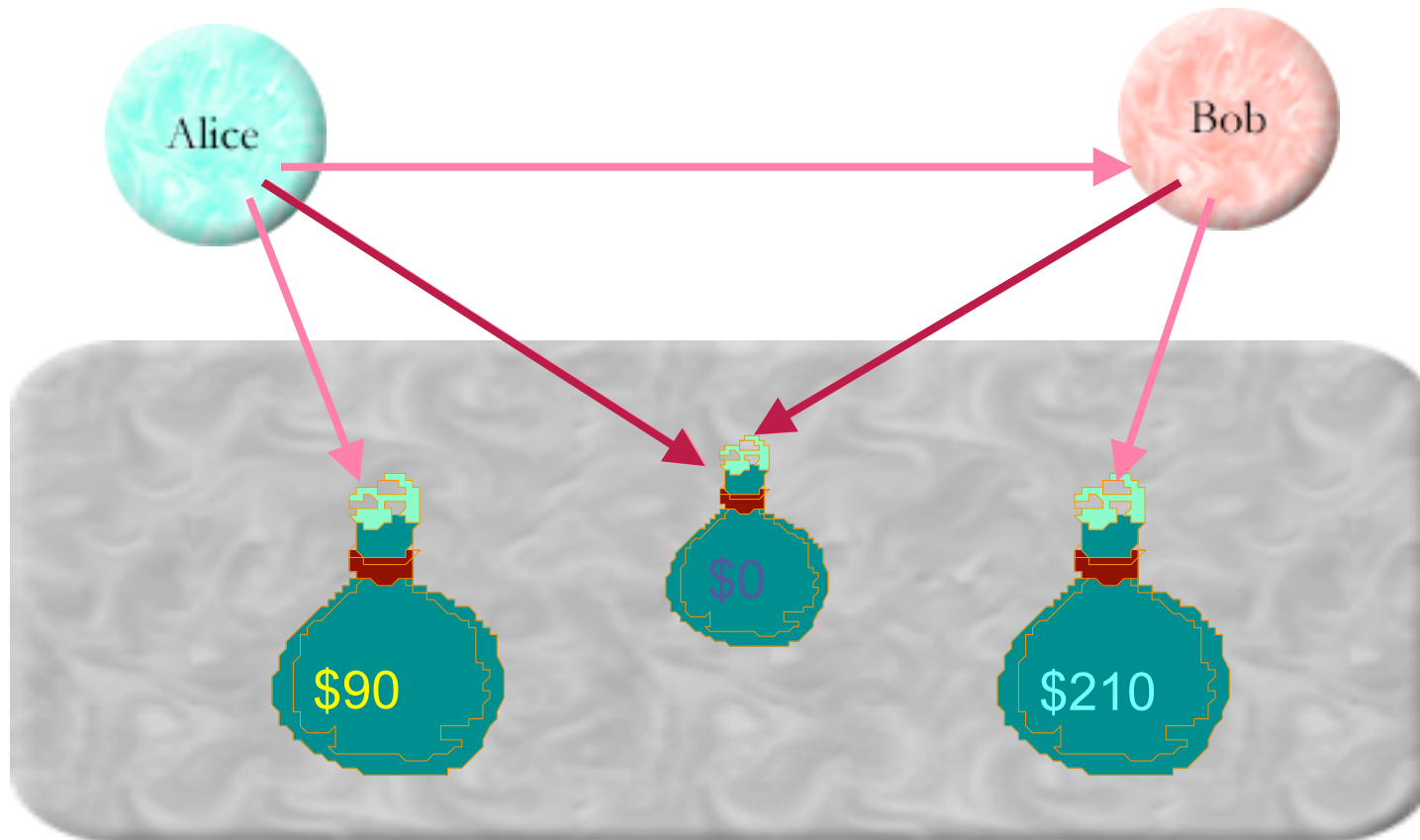
```
return Q.when(paymentP, const(p) {  
  return Q.when(myPurse ! deposit(10, p), const(_)) {
```



Distributed Secure Currency

```
const paymentP = myPurse ! makePurse();  
paymentP ! deposit(10, myPurse);  
const goodP = bobP ! buy(desc, paymentP);
```

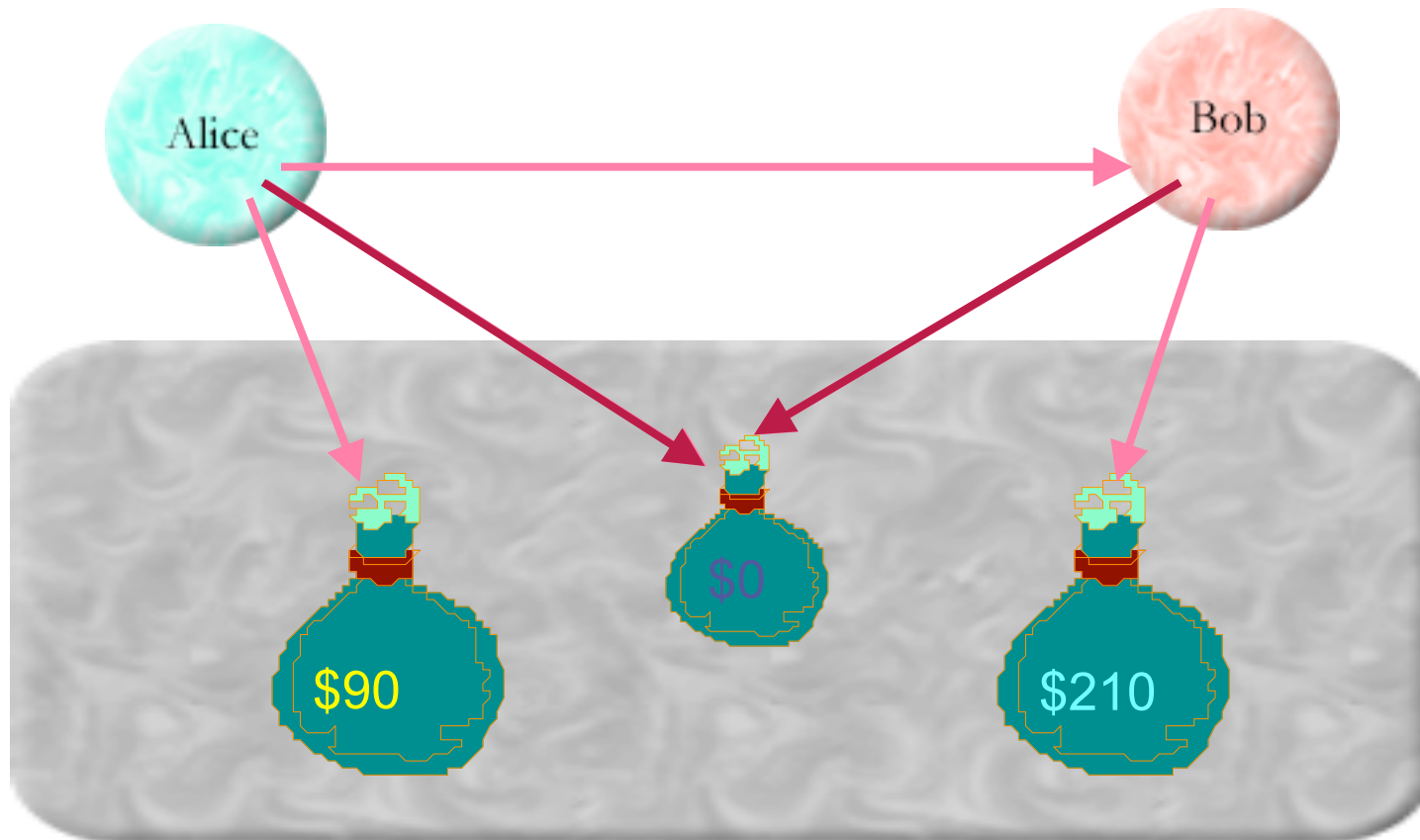
```
return Q.when(paymentP, const(p) {  
  return Q.when(myPurse ! deposit(10, p), const(_)) {
```



Distributed Secure Currency

```
const paymentP = myPurse ! makePurse();  
paymentP ! deposit(10, myPurse);  
const goodP = bobP ! buy(desc, paymentP);
```

```
return Q.when(paymentP, const(p) {  
  return Q.when(myPurse ! deposit(10, p), const(_) {  
    return good; }, ...
```



How to lose an arms race

Properties of Interpreters or the Browser Environment that allow Privilege Escalation

Below is a list of known attack vectors. We discuss the EcmaScript 3 language, quirks of existing interpreters, and browser specific extensions that could allow privilege escalation so that we can come up with tests for a safe JavaScript rewriter or verifier.

Attack Vectors at the EcmaScript/JavaScript level

- [GlobalObjectPoisoning](#) -- Global object poisoning
- [EvalArbitraryCodeExecution](#) -- eval and the Function constructor allow arbitrary code execution
- [ArgumentsMaskedByVar](#) -- function arguments array masked by var arguments on Opera
- [CrossScopeParameterModification](#) -- arguments array allows modification of parameters
- [ArgumentsExposesCaller](#) -- arguments Array and function object expose caller
- [FunctionMemberCrossScopeParameterAccess](#) -- function object's arguments array expose arguments while call in progress
- [TypeofInconsistent](#) -- typeof inconsistent for regular expressions
- [InaccessibleLocalVariables](#) -- Inaccessible local variables
- [CatchBlocksScopeBleed](#) -- catch blocks may cause global assignment, or local scope creep
- [GlobalScopeViaThis](#) -- Global scope reachable via this from functions not invoked as methods
- [DeleteUnmasksGlobals](#) -- Delete can unmask globals
- [FunctionConstructor](#) -- Function constructor accessible via the 'constructor' property
- [ObjectEvalArbitraryCodeExecution](#) -- Object.eval allows execution of unsanitized code on Firefox.
- [ObjectWatch](#) -- Object.watch allows stealing and poisoning of otherwise restricted data
- [ObjectToSourceLeaksPrivates](#) -- Object.toSource and uneval allow access to private fields
- [FunctionMethodsLeakGlobalScope](#) -- Function.call or Function.apply can leak window with certain this-values.
- [ConditionalCompilationComments](#) -- Conditional compilation may allow disabling of runtime checks.
- [StringObfuscationIsEasy](#) -- Approaches that rely on detecting code for other languages in string literals is easy to defeat
- [ParentCircumventsScoping](#) -- The javascript1.2 feature __parent__ circumvents normal scoping.
- [JsControlFormatChars](#) -- [\C&:] can be used to hide code in string or comments.
- [InconsistentlyReservedKeywords](#) -- Different reserved keyword set can cause parser ambiguity
- [ErrorExposesParameterValues](#) -- The stack property of Error includes parameter values.
- [HiddenControlFlowHazard](#) -- Seemingly safe Caja data computations may result in a control-flow transfer to a potential adversary.
- [RegexpsLeakMatchGlobally](#) -- Any regular expression can match against the last string passed to any other
- [EvalBreaksClosureEncapsulation](#) -- Eval extensions allow reaching into the scope chain of closures
- [PostIncrementAndDecrementCanReturnNonNumber](#) -- Incorrect implementations of postincrement and postdecrement can cause confusion as to which property is being accessed
- [MisOptimizations](#) -- Some interpreters try to optimize javascript before execution subtly changing the semantics of builtin operators ([PostIncrementAndDecrementCanReturnNonNumber](#) is a specific example)
- [CompoundAssignmentsCanReturnNonNumber](#) -- The type of assignment expressions may not be correct.
- [FinallySkipped](#) -- An exception that is thrown not inside a try/catch caught skips finally blocks.

Attack Vectors at the Browser Environment, DOM, HTML, or CSS levels

- [ScriptInHtml](#) -- HTML Tags in JavaScript Strings can allow Unsanitized Script Execution
- [SetTimeoutArbitraryCodeExecution](#) -- setTimeout & setInterval allow arbitrary code execution
- [DOMNodeAllowArbitraryCodeExecution](#) -- ActiveXObject, document.createElement, document allow arbitrary code execution
- [InnerHTMLYieldsCdata](#) -- script, style, xmp and listing elements' innerHTML cannot be safely inserted into another element's innerHTML
- [DomAllowsXsrf](#) -- document object allows arbitrary XSRF with the user's credentials
- [DomAllowsKeylogging](#) -- DOM access allows keylogging
- [XsrfViaXxe](#) -- XMLHttpRequest and DOMParser parsing allow arbitrary XSRF via XXE
- [CssAllowsArbitraryCodeExecution](#) -- Some CSS properties allow execution of unsanitized javascript
- [CssImportsAllowUnsanitizedCodeExecution](#) -- @import can import unsanitized CSS which can execute unsanitized javascript
- [NullCharEscapes](#) -- Null characters in URL can disguise protocols such as javascript:
- [ConfusedHtmlParsers](#) -- Differences in the way HTML parsers parse malformed HTML can hide unsanitized scripts
- [EventHandlersEvalWithDom](#) -- The scope that event handlers are executed in may expose DOM properties as globals
- [DocTypesCanInjectUnsanitizedContent](#) -- DOCTYPEs can define entities which can inject unsanitized script or markup.
- [EventChecksCircumventableByInfLoops](#) -- Invariants preserved by event handlers can be circumvented by causing the browser to turn off javascript.
- [IdAndNameMasking](#) -- Members of HtmlCollection, HTMLFormElement, etc. masked by ids&names
- [UriFetchingSideChannel](#) -- Side-channels from unproxied connections leak information across closed networks
- [HistoryMining](#) -- CSS can be used to determine whether a user has visited a URL.
- [RedirectWithoutUserAction](#) -- JS and HTML both allow redirection with user interaction.
- [PhishingViaCrossSiteHttpAuth](#) -- An attacker can display an HTTP authorization dialog that looks like it may have come from another site.



Distributed **Resilient** Secure EcmaScript

```
p1 = farBob ! foo(carol);           // queue request for Bob  
p3 = p1 ! bar(p2);                 // left dataflow chaining  
p5 = try when (i = p3, j = p4) { => i + j };    // gather results  
b5 = try whenever (i = b3, j = b4) { => i + j }; // perpetual  
p6 = try (f = farF, x = farX) in (farEval) { => f(x) }; // mobile
```



Distributed **Resilient** Secure EcmaScript

Remaining Open Resilience Problems

Persistence: How orthogonal?

Waterken, KeyKOS, E, Workers

Disconnected Operation: How to reconcile?

Dominant partition, Wave OT, Una, Ambient references

Upgrade: When instances outlive their class

Co-existence: When versions collide

Each presents new security challenges



x86: We've heard this tune before

Dumb terminals talk to mainframes	Early browsers talk to servers
Smart terminals do forms locally	HTML forms
Growth of terminal featuritis	Cancerous growth of HTML tags
High school student designs 8008 arch to save Datapoint chip count	Intern designs & builds JS to script HTML
"In two years I learned enough CS to realize the horror of what I'd created, but by then it was too late." –Harry Pyle	
Without anyone thinking it's good, achieves worldwide domination.	
Displaces competitors everyone agrees was better	
x86 now dominates on servers	Server-side JS growing fast
The last instruction set?	The last programming language?



Distributed Resilient Secure EcmaScript

Beautiful Simple Core: Scheme, Self

Objects as records. Functions as lexical closures.

Records of lexical closures => objects with methods

```
function makeCounter(count) {  
  return {  
    incr: function() { return ++count; }  
  };  
}
```



Distributed Resilient Secure EcmaScript

EcmaScript 5 Strict

Tamper-proof (frozen) objects. Encapsulated closures.

Frozen records of protected closures => High integrity

'use strict';

```
const makeCounter = Object.freeze(function(count) {  
  return Object.freeze({  
    incr: Object.freeze(function() { return ++count; });  
  });  
});
```



Distributed Resilient Secure EcmaScript

EcmaScript Harmony

Makes high integrity convenient

Faithful virtualization by interposition

Modular modules with lexical scoping

```
const makeCounter(count) {  
  return Object.freeze({  
    incr: const() { return ++count; };  
  });  
};
```



Distributed Resilient **Secure** EcmaScript

When Alice asks: `bob.foo(carol)`

Alice grants Bob access to Carol, as needed for `foo`

Memory-safe encapsulated objects

Protect objects from their outside world



Distributed Resilient **Secure** EcmaScript

When Alice asks: `bob.foo(carol)`

Alice grants Bob access to Carol, as needed for `foo`

Memory-safe encapsulated objects

Protect objects from their outside world

OCaps: Causality only by references

No powerful references by default

Protect world from objects

Reference graph === Access graph

Deny authority by withholding connectivity



Distributed Resilient **Secure** EcmaScript

Java : Joe-E :: EcmaScript : SES

Defensive Consistency & Natural POLA

SES \subset (ES5 Strict + a bit of ES-Harmony)

Deny access to global variables, global object

Delete non-whitelisted properties

Freeze accessible primordials (Object, Array, Array.prototype,...)

Restrict eval() and Function() to SES



Distributed Resilient **Secure** EcmaScript

Easy Secure JavaScript Mashups Impossible?



Distributed Resilient **Secure** EcmaScript

Easy Secure JavaScript Mashups Impossible?

The counter example:

```
const bobEndowments = Object.freeze({counter: makeCounter(0)});  
const bobMakerCode = //... fetch potentially malicious code ...  
const bob = eval(bobMakerCode).make(bobEndowments);
```

Bob can *only* count.



Distributed **Resilient** Secure EcmaScript

Between machines...

*There is no **do**, there is only **try**.*

--with apologies to Yoda

```
p1 = farBob ! foo(carol); // Bob throws, breaking p1  
p3 = p1 ! bar(p2);       // broken promise contagion
```



Distributed **Resilient** Secure EcmaScript

Between machines...

*There is no **do**, there is only **try**.*

--with apologies to Yoda

```
p1 = farBob ! foo(carol); // Bob throws, breaking p1
p3 = p1 ! bar(p2);        // broken promise contagion
p4 = try when (r3 = p3) { // delayed error handling
    => "ok: " + r3
} catch (ex) { => "bad: " + ex };
```



Caja Roadmap

	Cajita	SES5/3	SES/ES5-strict
+	Valija	ES5/3	ES5-strict
+	ref_send / server-proxy	→	ref_send / UMP
+		server-server captp	captp / web-sockets
+		"!" "in" expression sugar	→
Subtotal:		Dr. SES5/3	Dr. SES
+	Sanitize HTML & CSS	→	
+	Domita / uncajoled JS	Domado / SES	→
=	Caja Today	Caja Tomorrow	Caja on ES5,HTML5

