Sequence Traces for Object-Oriented Executions

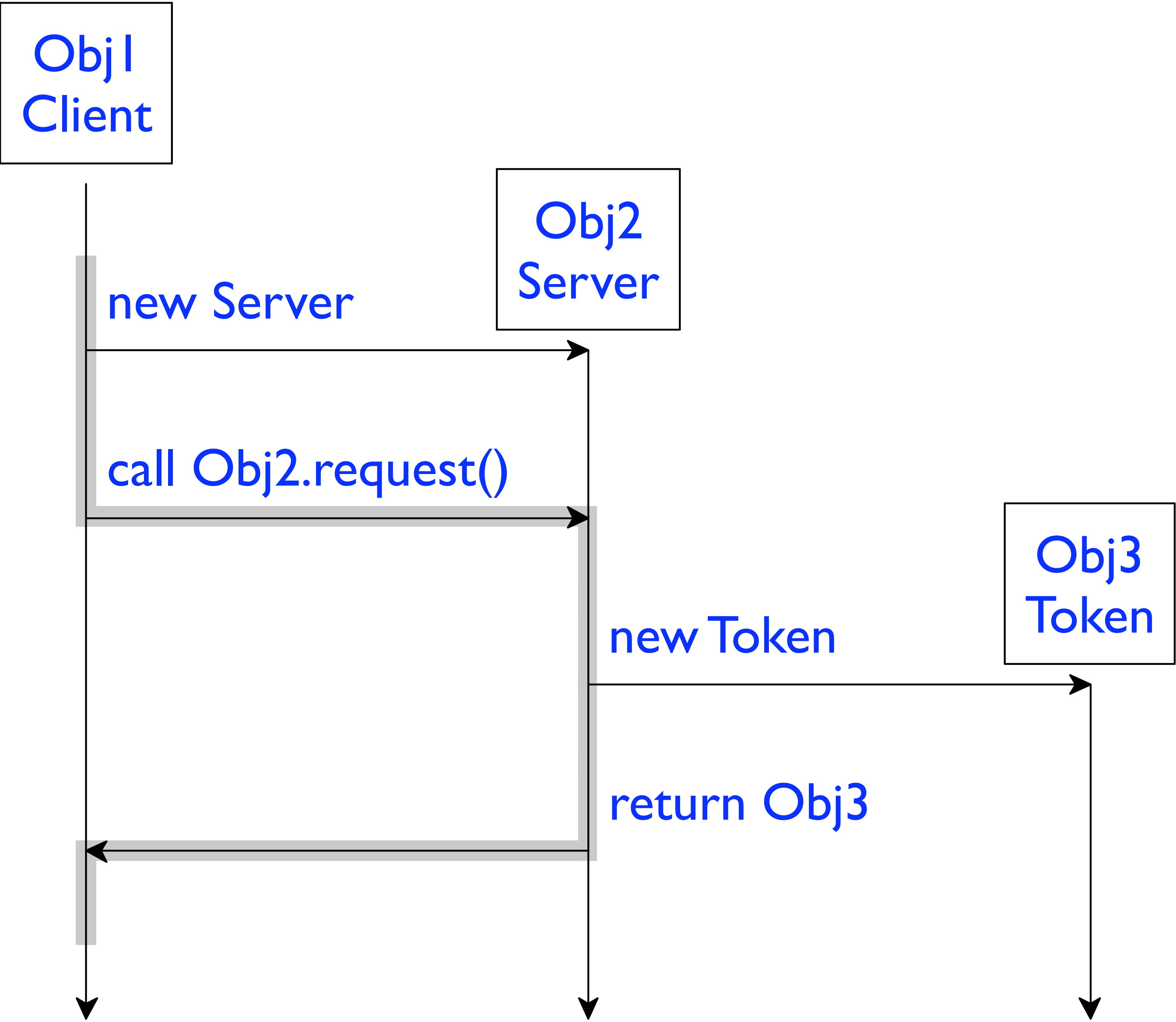
Carl Eastlund and Matthias Felleisen

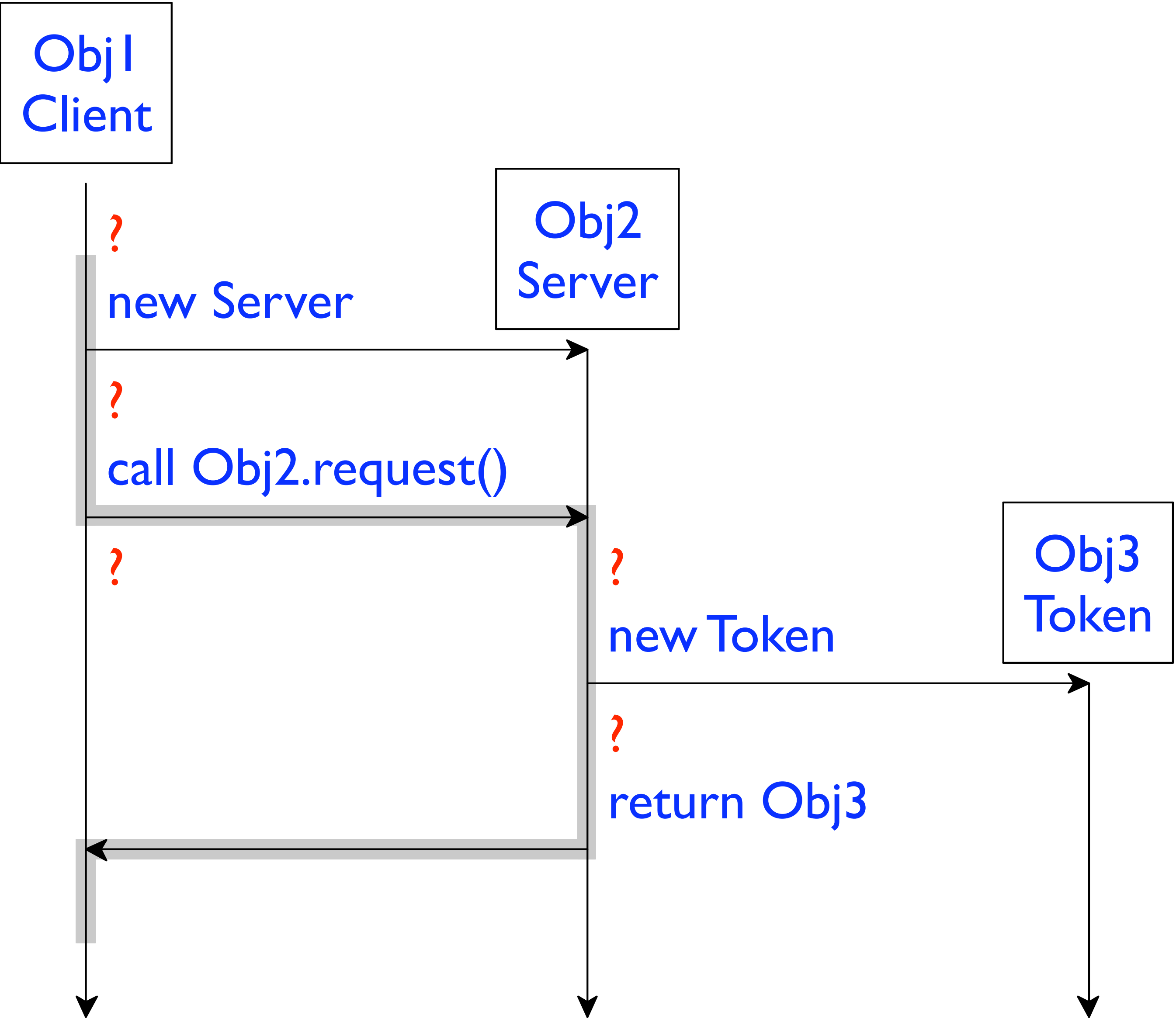
Northeastern University

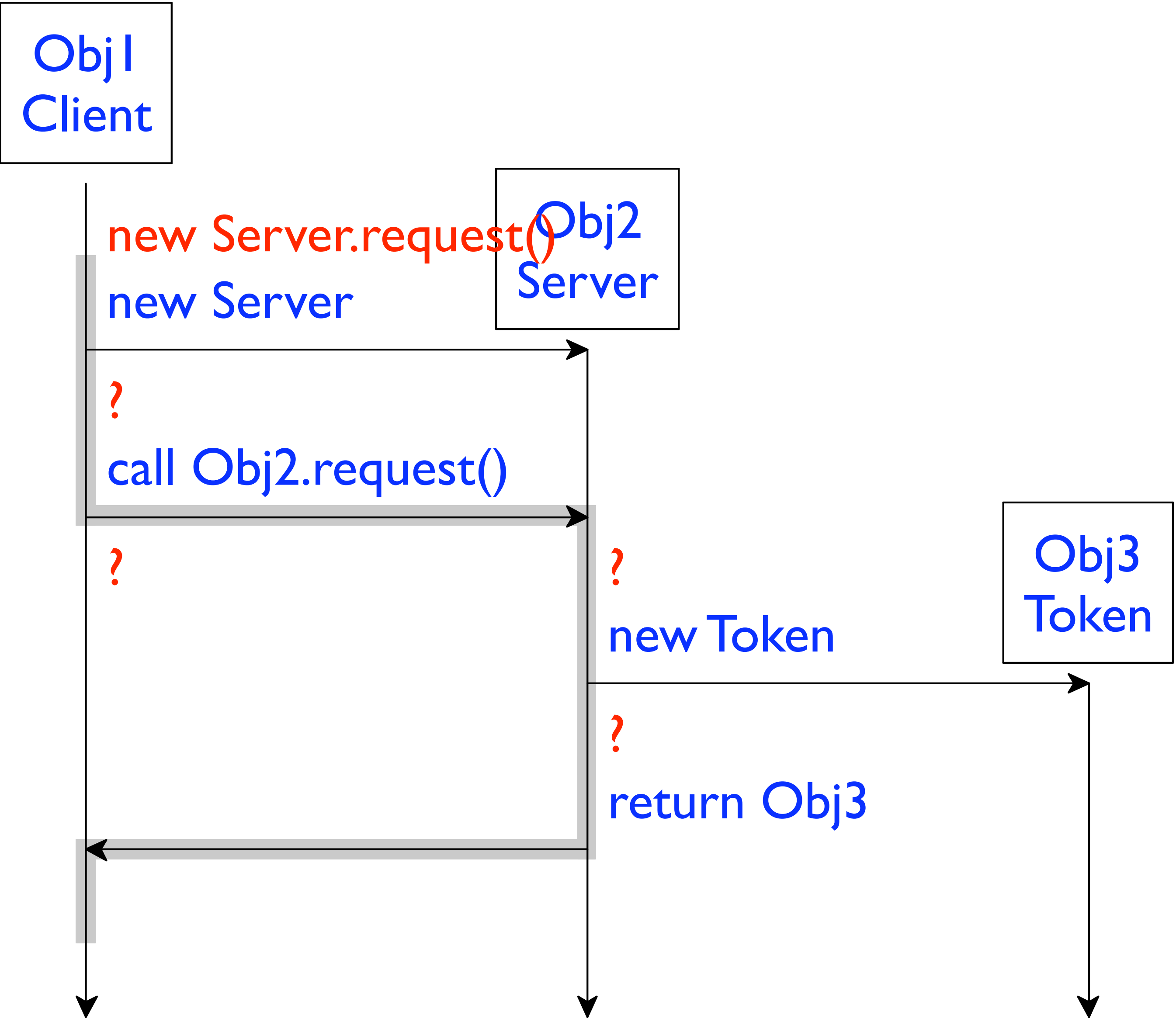
```
class Client
{ Object run () { new Server.request ()}}

class Server
{ Token request () { new Token}}

class Token {}

new Client.run ()
```

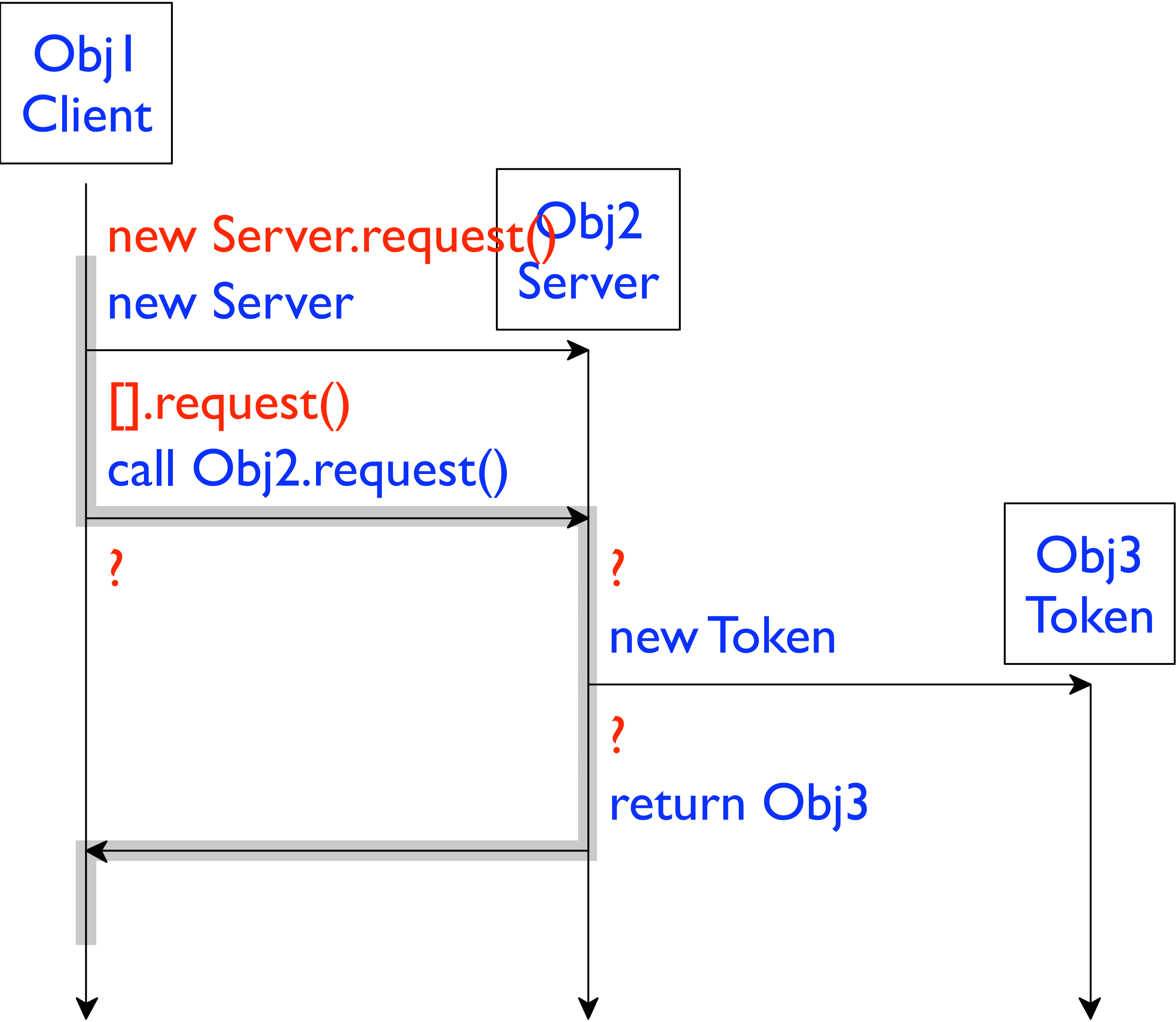
```
(define client%
  (class object%
    (super-new)
    (define/public (run)
      (send (new server%) request))))

(define server%
  (class object%
    (super-new)
    (define/public (request)
      (new token%))))

(define token% (class object% (super-new)))

(send (new client%) run)
```

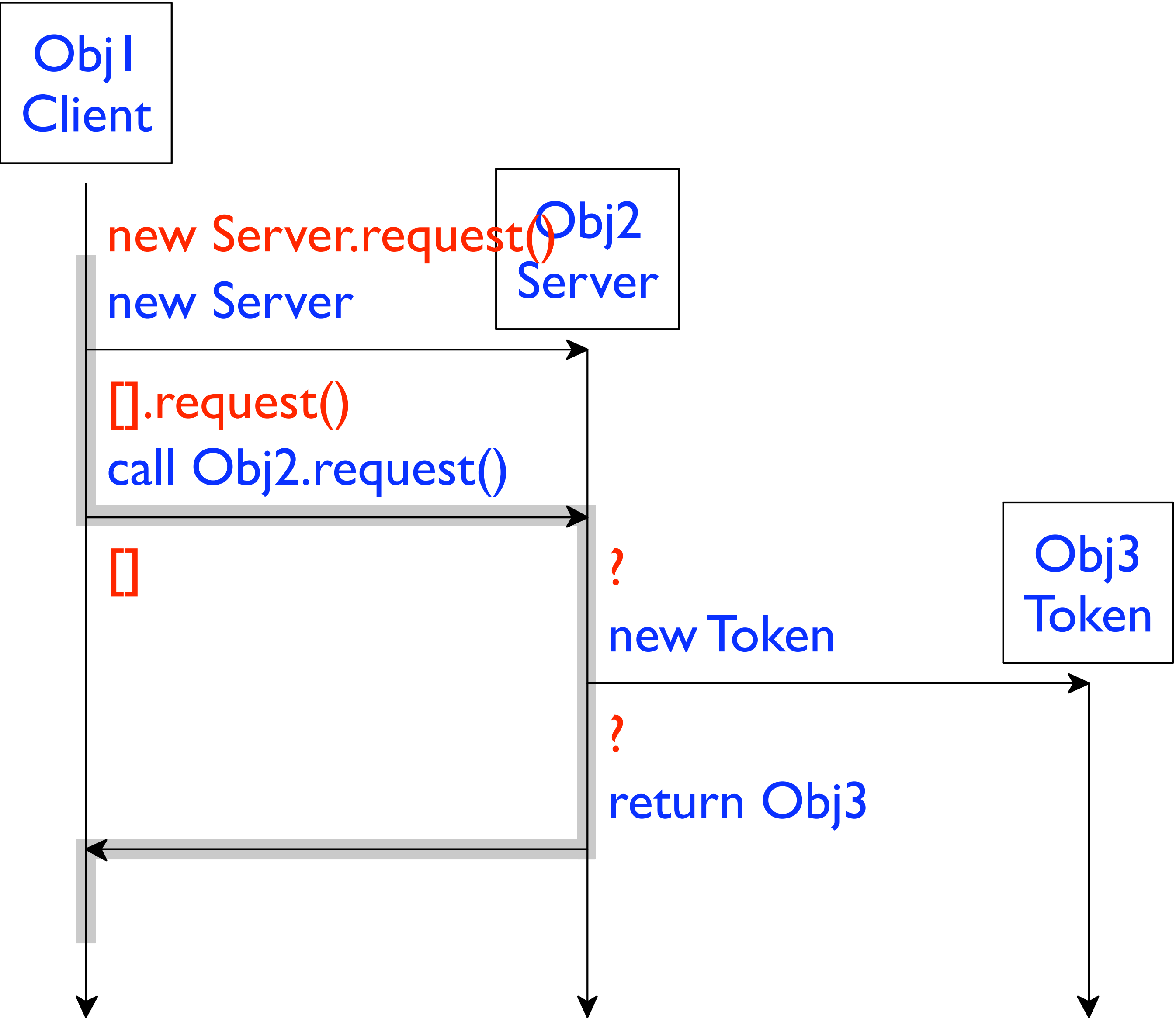
```
[run = ς (c) [request = ς (s) []] .request] .run
```

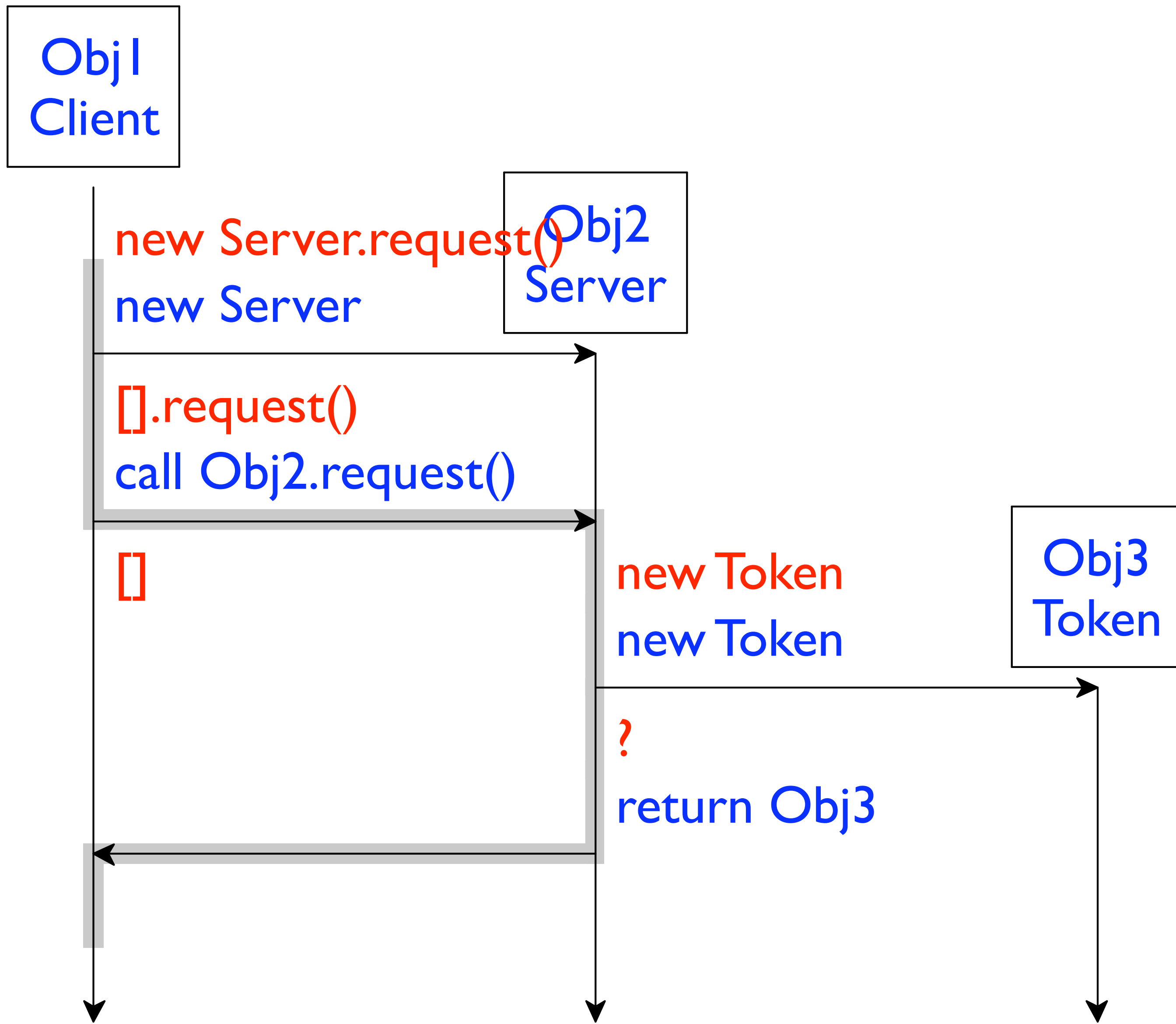

Obj1
Client
Obj2
Server
Obj3
Token
new Server
call Obj2.request()
new Token
return Obj3

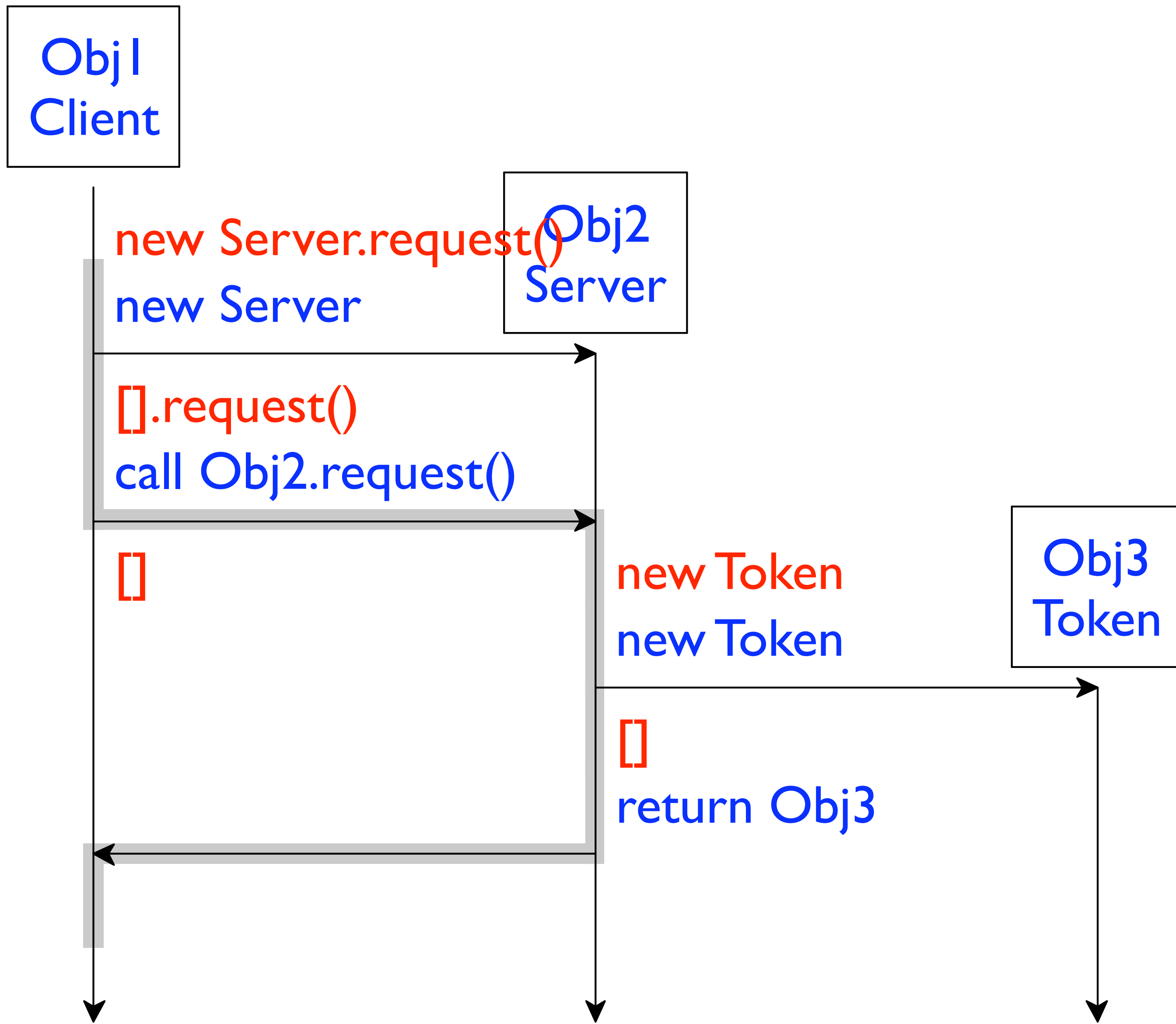

Obj1
Client
Obj2
Server
Obj3
Token
?
new Server
?
call Obj2.request()
?
?
new Token
?
return Obj3


Obj1
Client
Obj2
Server
new Server.request()
new Server
?
call Obj2.request()
?
Obj3
Token
?
new Token
?
return Obj3


Obj1
Client
Obj2
Server
Obj3
Token
new Server.request()
new Server
[].request()
call Obj2.request()
?
?
new Token
?
return Obj3


Obj1
Client
Obj2
Server
Obj3
Token
new Server.request()
new Server
[].request()
call Obj2.request()
[]
?
new Token
?
return Obj3


Obj1
Client
new Server.request()
new Server
Obj2
Server
[].request()
call Obj2.request()
[]
new Token
new Token
Obj3
Token
?
return Obj3


Obj1
Client
new Server.request()
new Server
Obj2
Server
[].request()
call Obj2.request()
[]
new Token
new Token
Obj3
Token
[]
return Obj3

Two-Level Framework

```
State  = ⟨Pool,Stack,Ref,Action⟩
Object = ⟨static,Dynamic⟩
Stack  = ⟨Ref, cont⟩ *
Action = new Object; cont
       | inspect Ref; cont
       | get Ref.field; cont
       | set Ref.field := Value; cont
       | call Ref.method(Value *); cont
       | return Value
       | ERR
```

$\text{init(program)} = \langle \text{Pool}, \text{Stack}, \text{Ref}, \text{Action} \rangle$

$\langle \text{Pool, Stack, Ref, new Object; cont} \rangle \rightarrow$
$\langle \text{Pool[Ref}' \mapsto \text{Object], Stack, Ref, resume(cont,Ref}')\rangle$

$\langle \text{Pool, Stack, Ref, call Ref}'\text{.method(Value *); cont} \rangle \rightarrow$
$\langle \text{Pool}, \langle \text{cont,Ref} \rangle \text{ Stack, Ref, invoke(Ref}',\text{Pool(Ref}'),\text{method,Value *)} \rangle$

. . .

```
static    = class
primitive = null
cont      = []
          | { type x = cont; type x = e; * e }
          | (type) cont
          | (cont <: type) Ref
          | cont : class . fieldCJ
          | cont : class . fieldCJ = expr
          | . . .
```

```
init(def * expr)                              =   eval(expr)
resume(cont, Value)                           =   eval(cont[Value])
invoke(Ref, Object, method, Value *)          =   ... eval(...) ...
```

```
eval({ type' x' = Value'; type x = expr; * expr'}) =
    eval({ type x = expr[x' := Value']; * expr'[x' := Value']})
eval({ expr }) = eval(expr)
eval(Value) = return Value
eval(cont[new class]) = new construct(class); cont
. . .
```

Type Soundness

If init, invoke, and resume are total and produce appropriately typed outputs for their inputs, then for every program P of type T, either P diverges or init(P) →* R and R : T.

type	:	Value types
objtype	:	Object Types (subset of type)
<:	:	Subtyping (partial order)
fields	:	Fields of an objtype
methods	:	Methods of an objtype
metatype	:	Static record of an objtype

Object Debugger

Thank you.