Isolates, channels and event streams

for composable distributed programming

Aleksandar Prokopec

Martin Odersky

State of the art

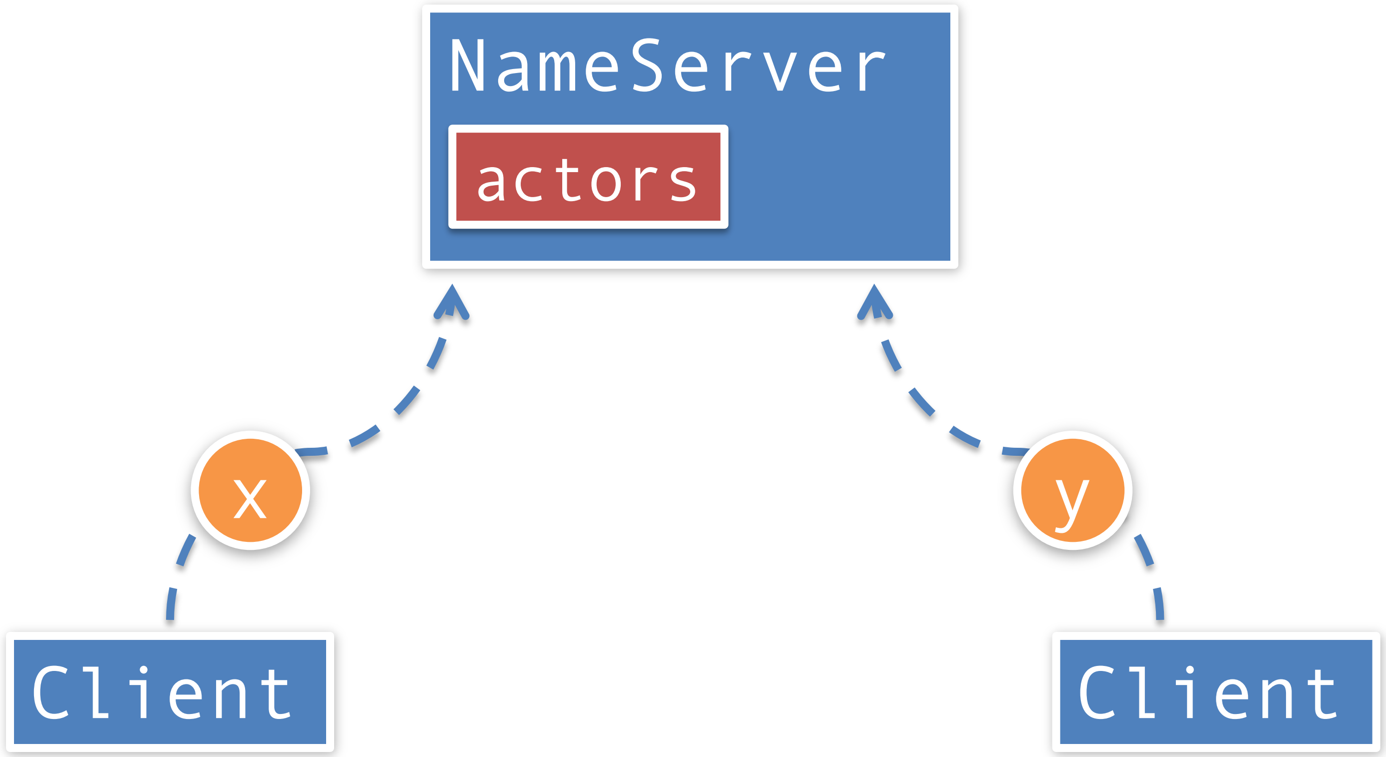
```
class NameServer extends Actor {




}
```

```scala
class NameServer extends Actor {
  val actors = Map[String, ActorRef]()




}
```

```
class NameServer extends Actor {
  val actors = Map[String, ActorRef]()
  def receive = {
    case nm: String => sender ! actors(nm)
  }
}
```

```scala
class NameServer extends Actor {
  val actors = Map[String, ActorRef]()
  def receive = {
    case nm: String => sender ! actors(nm)
  }
}
val ns = actorOf(NameServer)
```

```
class NameServer extends Actor {
  val actors = Map[String, ActorRef]()
  def receive = {
    case nm: String => sender ! actors(nm)
  }
}
val ns = actorOf(NameServer)

class Client extends Actor {
  ns ! “p”
}
for (i <- 0 until 2) actorOf(Client)
```


NameServer
actors
x
y
Client
Client

The problem

```scala
class Server[T, S](f: T => S) extends Actor {

}
```

```scala
class Server[T, S](f: T => S) extends Actor {
  def receive = { case x: T => sender ! f(x) }
}
```

```
class Server[T, S](f: T => S) extends Actor {
  def receive = { case x: T => sender ! f(x) }
}

class Client[T, S]
  (server: ActorRef, req: T, action: S => Unit)
extends Actor {



}
```

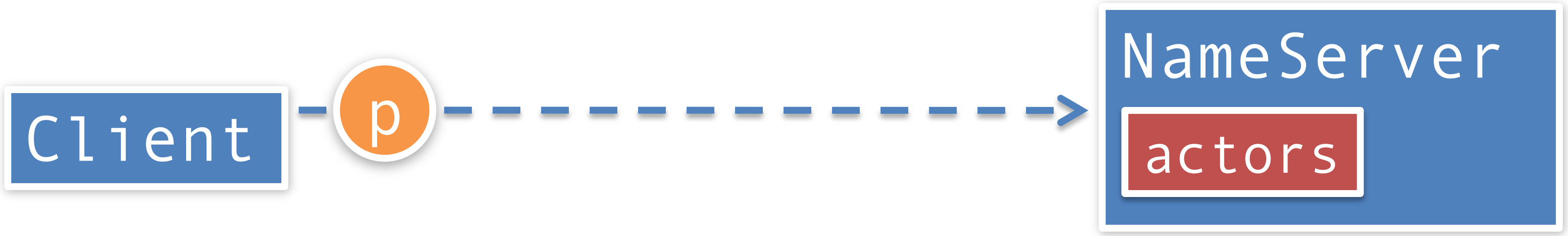
```
class Server[T, S](f: T => S) extends Actor {
  def receive = { case x: T => sender ! f(x) }
}

class Client[T, S]
  (server: ActorRef, req: T, action: S => Unit)
extends Actor {
  server ! req

}
```

```scala
class Server[T, S](f: T => S) extends Actor {
  def receive = { case x: T => sender ! f(x) }
}

class Client[T, S]
  (server: ActorRef, req: T, action: S => Unit)
extends Actor {
  server ! req
  def receive = { case x: S => action(x) }
}
```

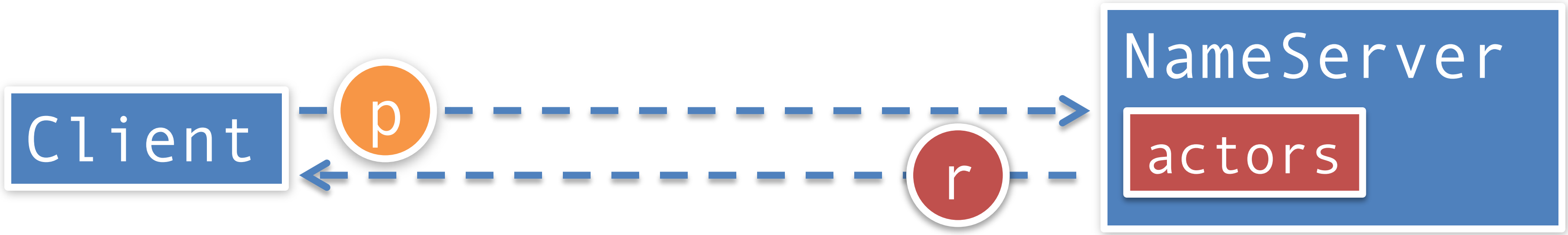
```scala
val actors = Map[String, ActorRef]()
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, "p", println))
```

Client
NameServer
actors

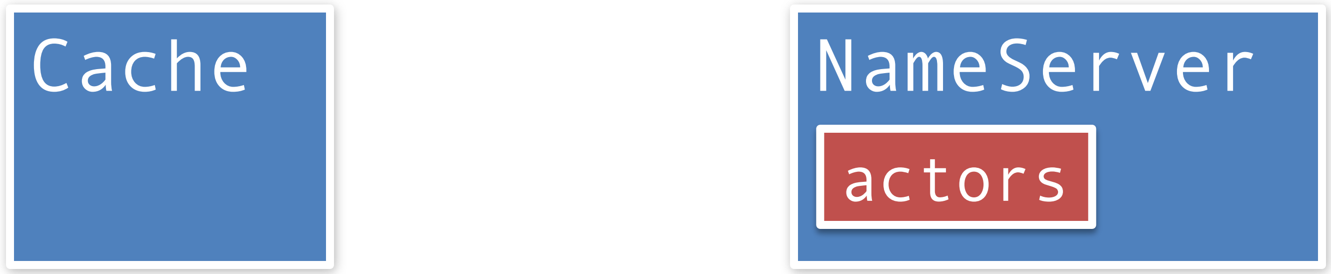
```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))
```



```scala
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))
```

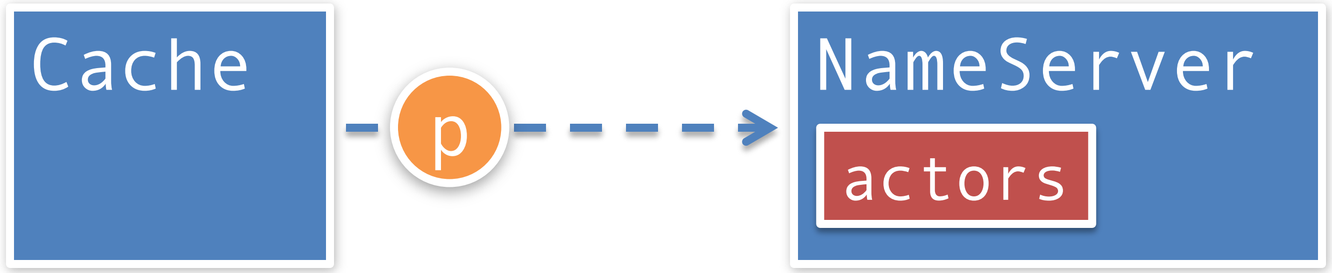


```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))
```



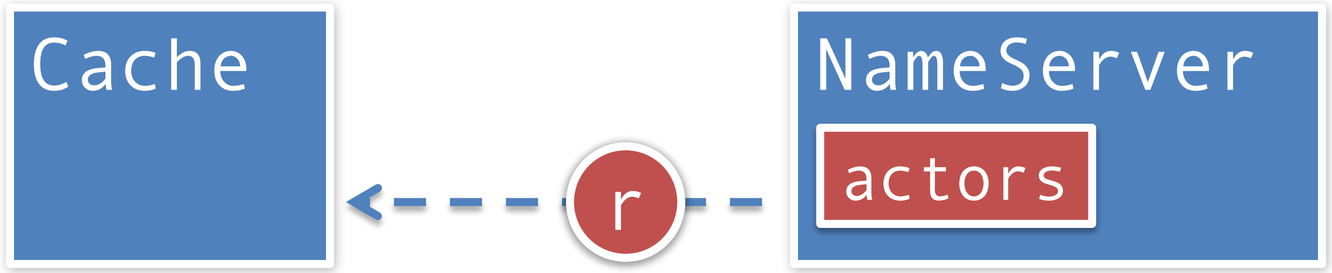
```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))
```

Cache is a client.



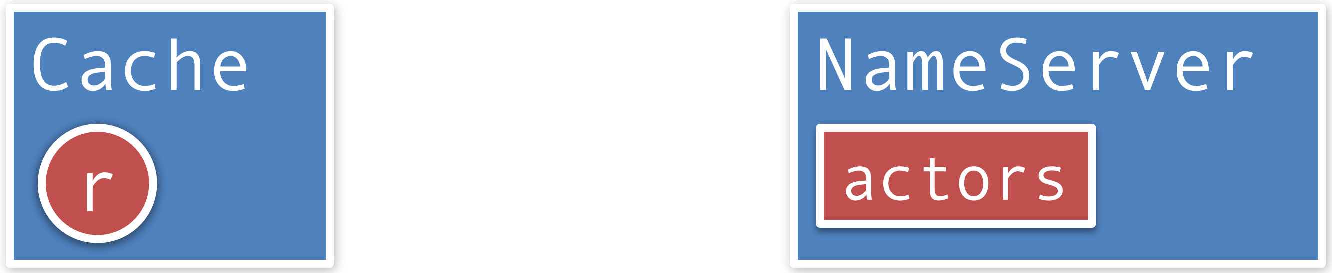
```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))
```

Cache is a client.

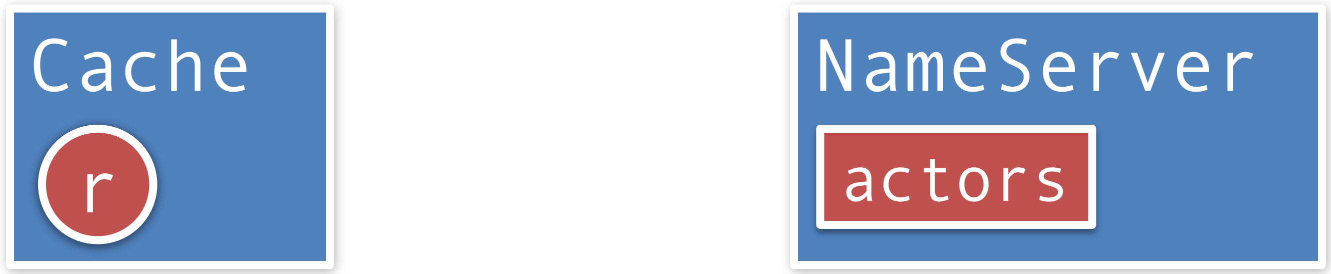


```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))
```

Cache is a client.

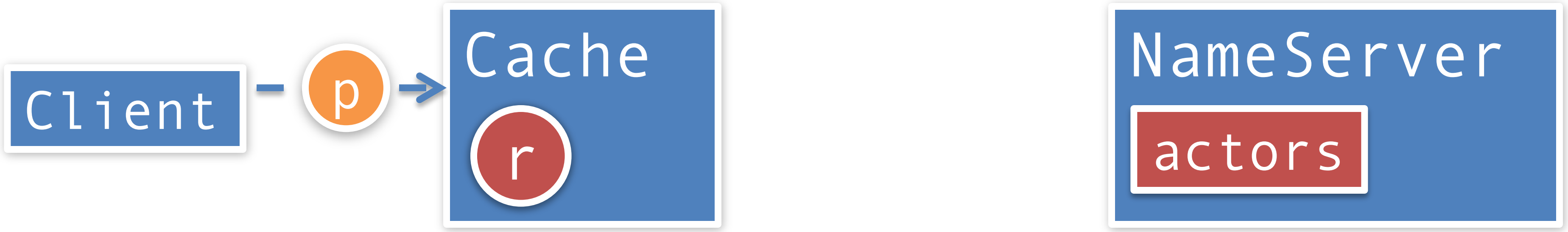


```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))

class Cache(var cached: ActorRef = null)
extends Client(ns, “p”, r => cached = r)
```



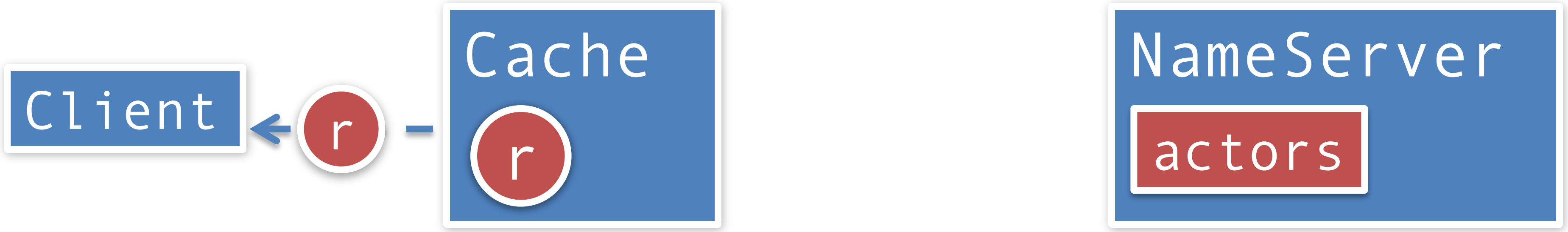
```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))

class Cache(var cached: ActorRef = null)
extends Client(ns, “p”, r => cached = r)
```

Cache is a server.

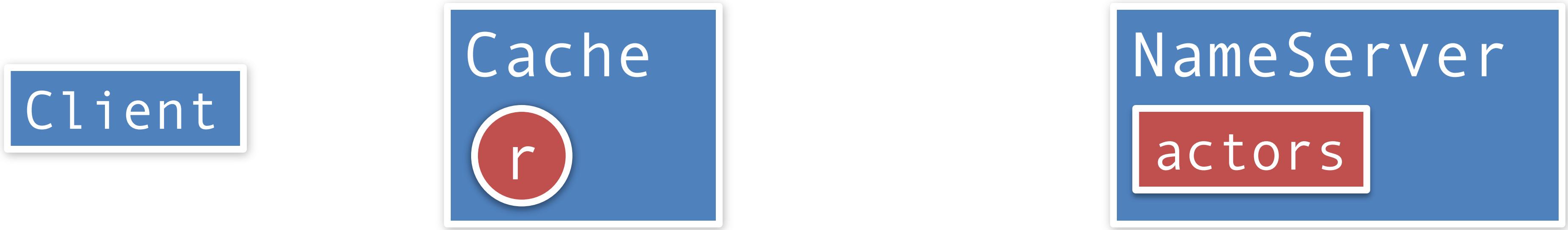


```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))

class Cache(var cached: ActorRef = null)
extends Client(ns, “p”, r => cached = r)
```

Cache is a server.

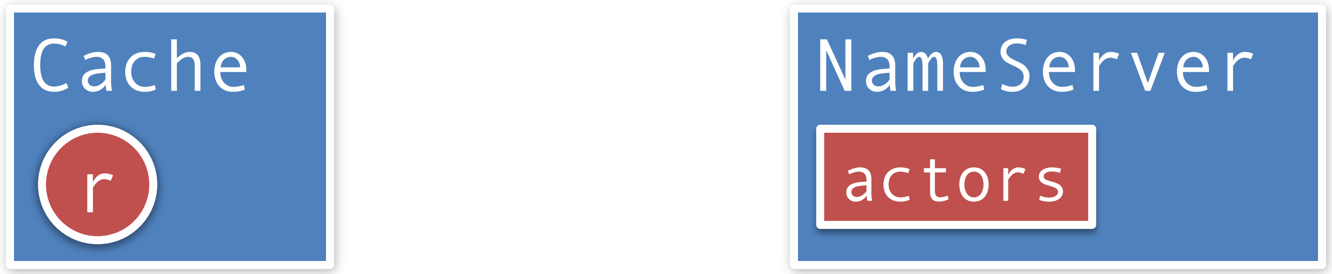


```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))

class Cache(var cached: ActorRef = null)
extends Client(ns, “p”, r => cached = r) {
  def receive = super.receive orElse {
    case “p” => sender ! cached
```



```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))

class Cache(var cached: ActorRef = null)
extends Client(ns, “p”, r => cached = r) {
  def receive = super.receive orElse {
    case “p” => sender ! cached
```

Cache can be refreshed.



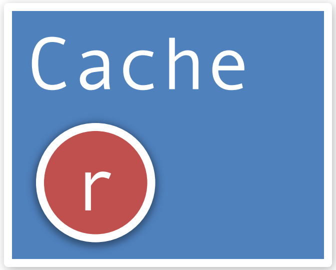
```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))

class Cache(var cached: ActorRef = null)
extends Client(ns, “p”, r => cached = r) {
  def receive = super.receive orElse {
    case “p” => sender ! cached
```

Cache can be refreshed.

Cache
r
NameServer
actors

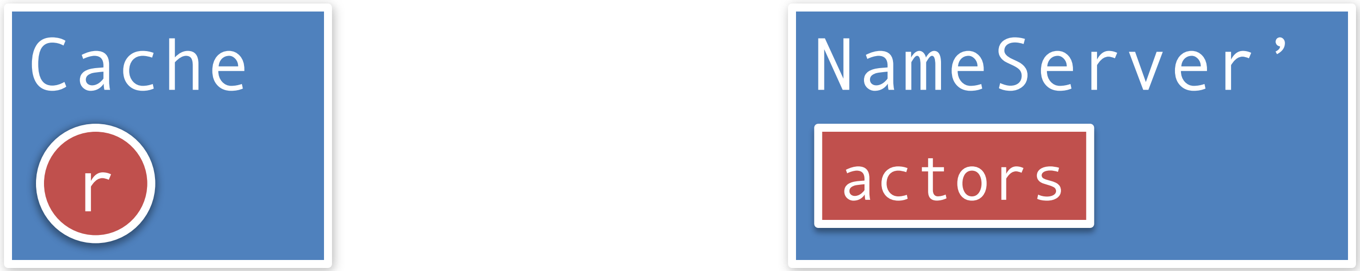
```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))

class Cache(var cached: ActorRef = null)
extends Client(ns, “p”, r => cached = r) {
  def receive = super.receive orElse {
    case “p” => sender ! cached
```

Cache can be refreshed.



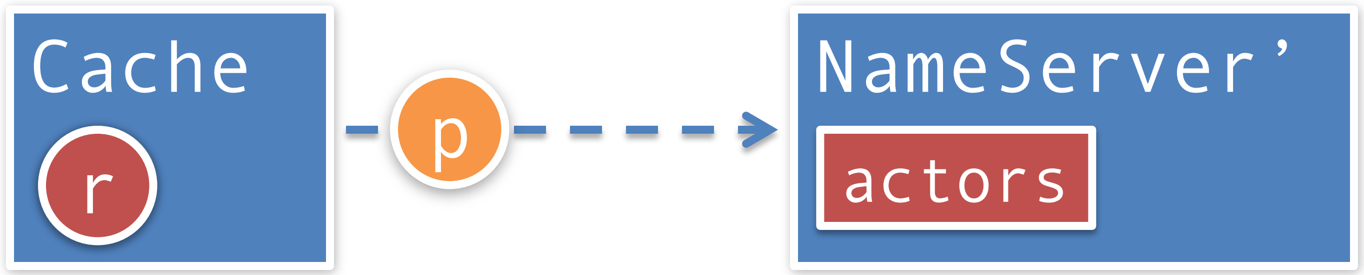
```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))

class Cache(var cached: ActorRef = null)
extends Client(ns, “p”, r => cached = r) {
  def receive = super.receive orElse {
    case “p” => sender ! cached
```

Cache can be refreshed.



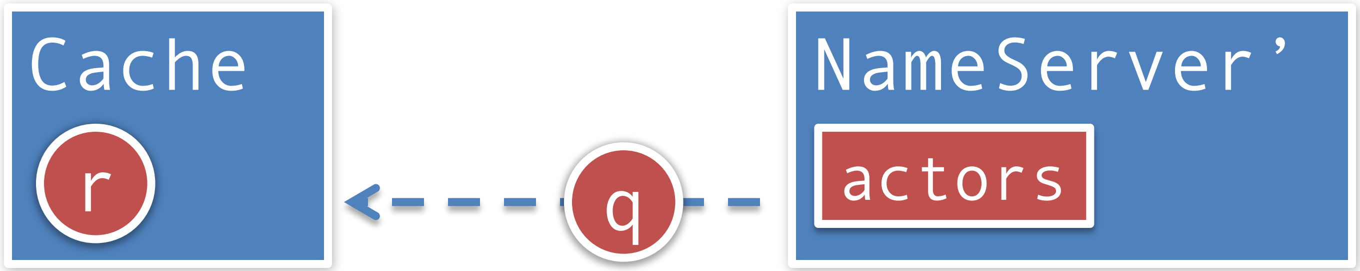
```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))

class Cache(var cached: ActorRef = null)
extends Client(ns, “p”, r => cached = r) {
  def receive = super.receive orElse {
    case “p” => sender ! cached
```

Cache can be refreshed.



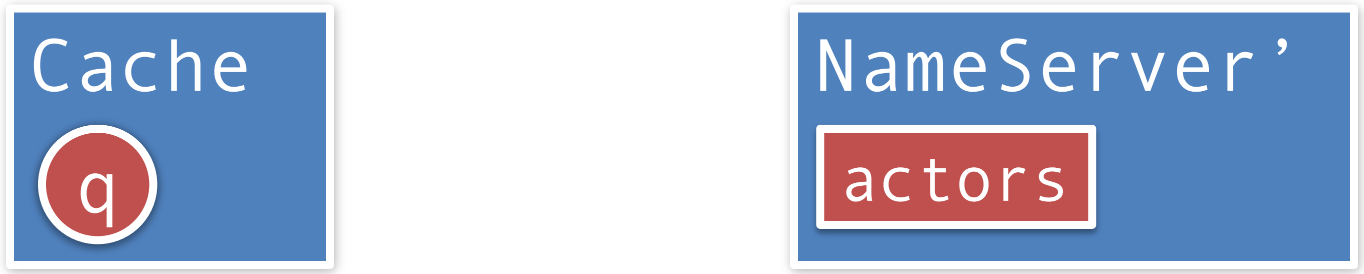
```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))

class Cache(var cached: ActorRef = null)
extends Client(ns, “p”, r => cached = r) {
  def receive = super.receive orElse {
    case “p” => sender ! cached
```

Cache can be refreshed.

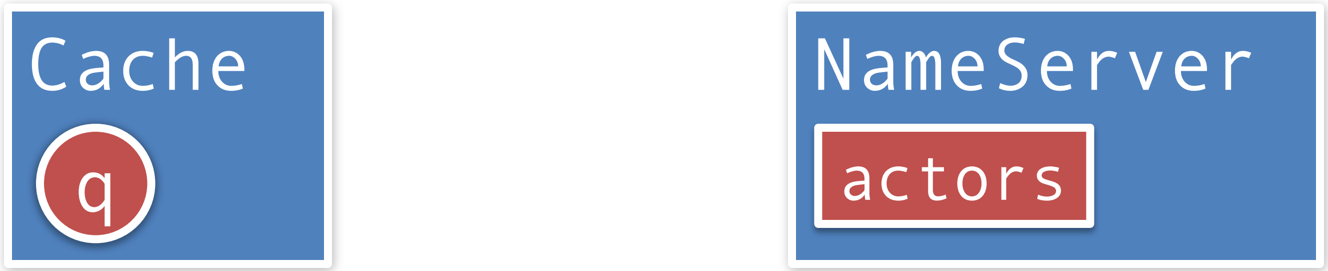


```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, “p”, println))

class Cache(var cached: ActorRef = null)
extends Client(ns, “p”, r => cached = r) {
  def receive = super.receive orElse {
    case “p” => sender ! cached
```

Cache can be refreshed.

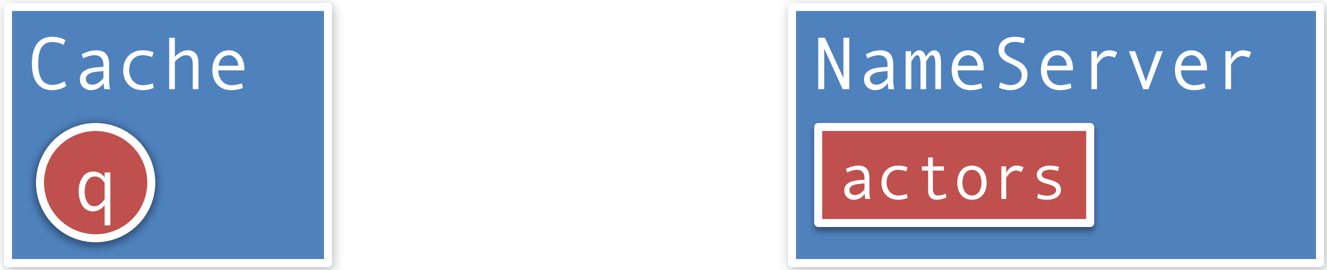


```
val actors = Map[String, ActorRef]()
val ns = actorOf(Server(actors))
val client = actorOf(Client(ns, "p", println))

class Cache(var cached: ActorRef = null)
extends Client(ns, "p", r => cached = r) {
  def receive = super.receive orElse {
    case "p" => sender ! cached
    case newNs: ActorRef => newNs ! "p"
  }
}
```

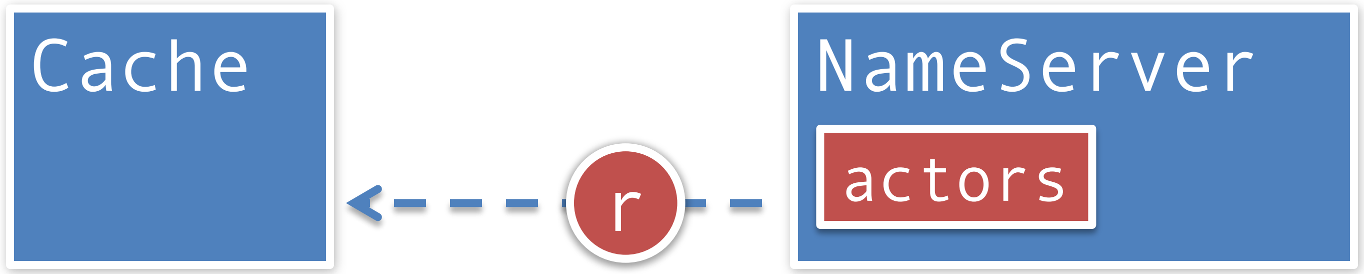


Not just ugly, but also incorrect.

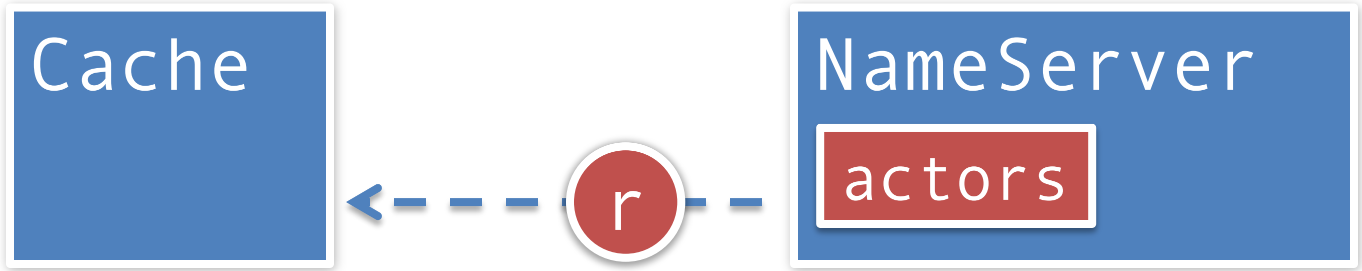
```
class Cache(var cached: ActorRef = null)
extends Client(ns, “p”, r => cached = r) {
  def receive = super.receive orElse {
    case “p” => sender ! cached
    case newNs: ActorRef => newNs ! “p”
  }
}
```



```
...
def receive = { case x: S => action(x) }
...

class Cache(var cached: ActorRef = null)
extends Client(ns, “p”, r => cached = r) {
  def receive = super.receive orElse {
    case “p” => sender ! cached
    case newNs: ActorRef => newNs ! “p”
  }
}
```



```
...
def receive = { case x: S => action(x) }
...

class Cache(var cached: ActorRef = null)
extends Client(reg, “p”, r => cached = r) {
  def receive = {
    case r: ActorRef => cached = r
    case “p” => sender ! cached
    case newNs: ActorRef => newNs ! “p”
  }
}
```



Fundamental problem

Implementer needs to be aware of all the protocols running in the actor.

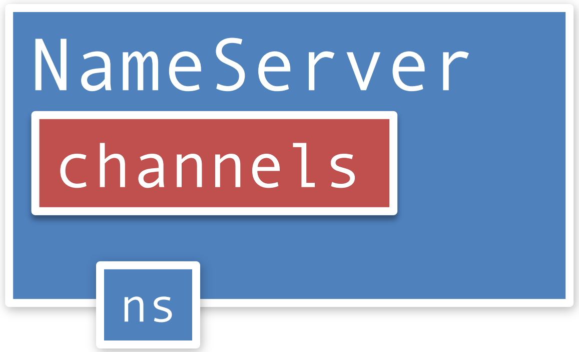
First ingredient

Express concurrency in the system

```scala
class NameServer
extends Iso[String] {
  val channels = Map[String, Channel[_]]()




}
```

NameServer
channels

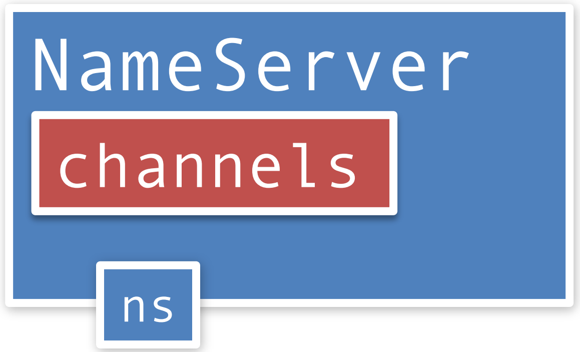
```
class NameServer
extends Iso[String] {
  val channels = Map[String, Channel[_]]()




}
val ns: Channel[String] = isolate(NameServer)
```



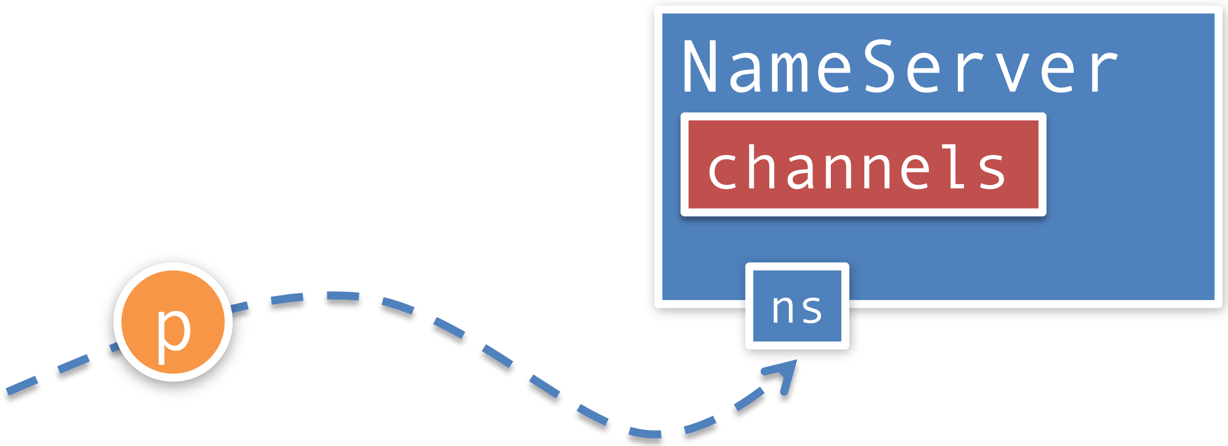
Second ingredient

Send information to other processes

```
class NameServer
extends Iso[String] {
  val channels = Map[String, Channel[_]]()




}
val ns: Channel[String] = isolate(NameServer)
ns ! "p"
```



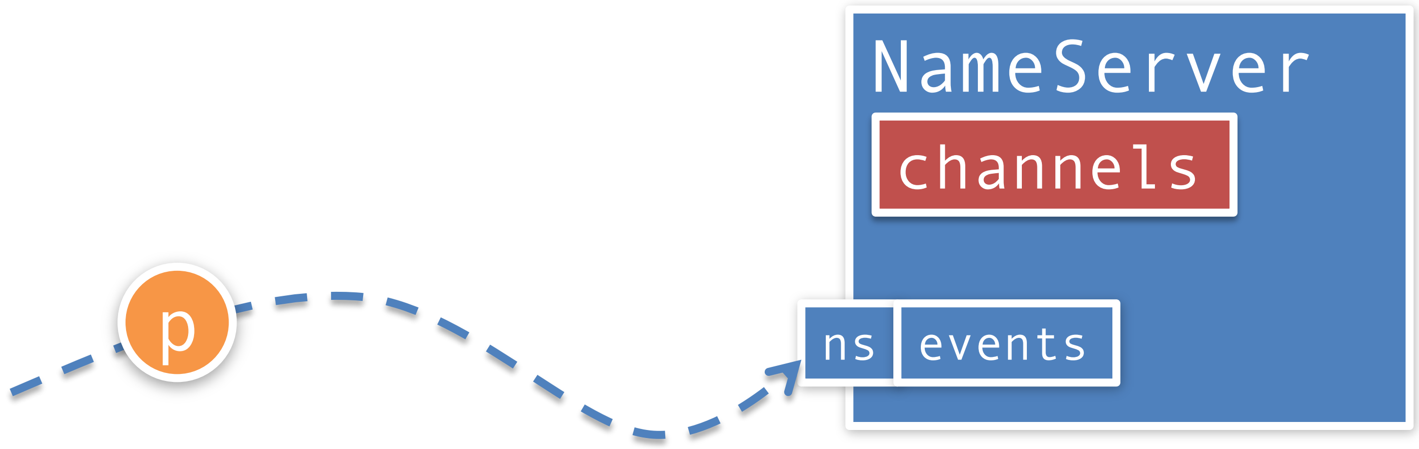
```
class NameServer
extends Iso[String] {
  val channels = Map[String, Channel[_]]()




}
val ns: Channel[String] = isolate(NameServer)
ns ! “p”
```



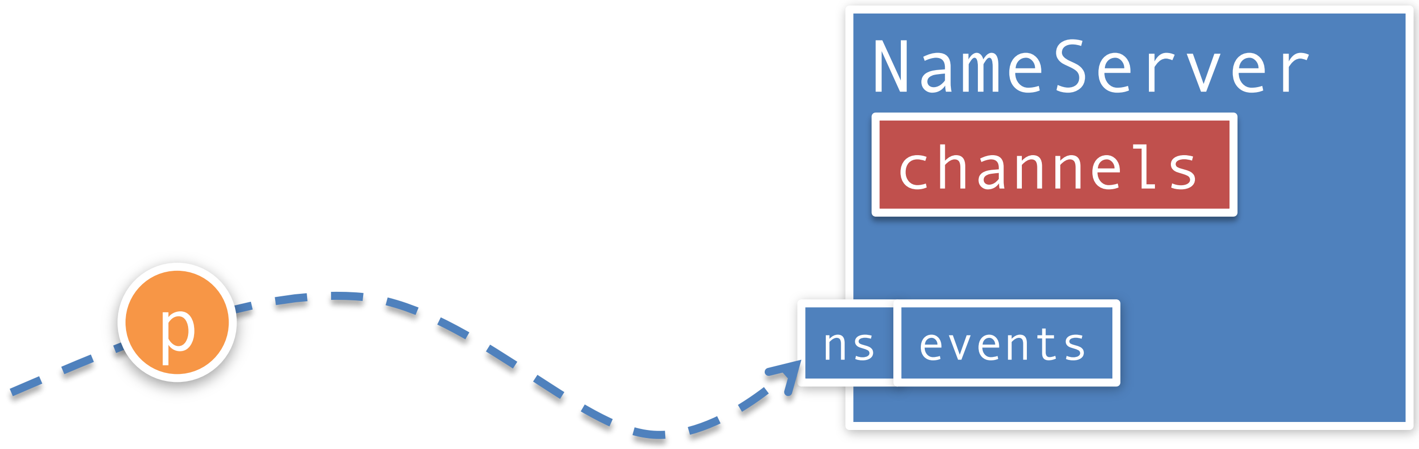
Third ingredient

Receive information from other processes

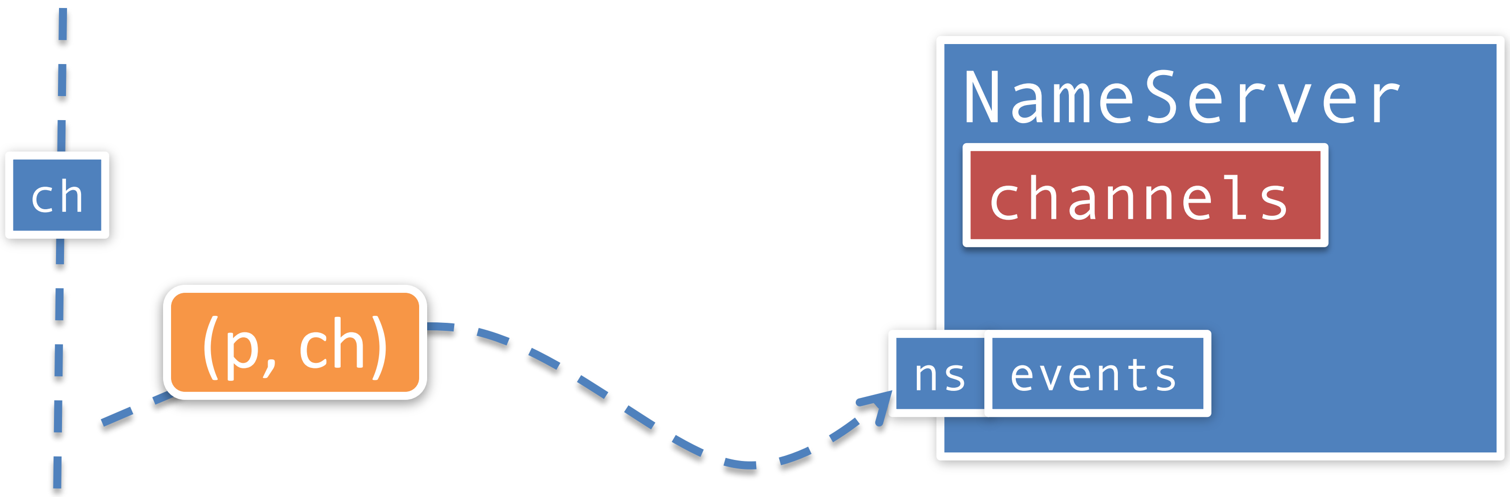
```
class NameServer
extends Iso[String] {
  val channels = Map[String, Channel[_]]()
  events onMatch {
    case name => sender ! channels(name)
  }
}
```



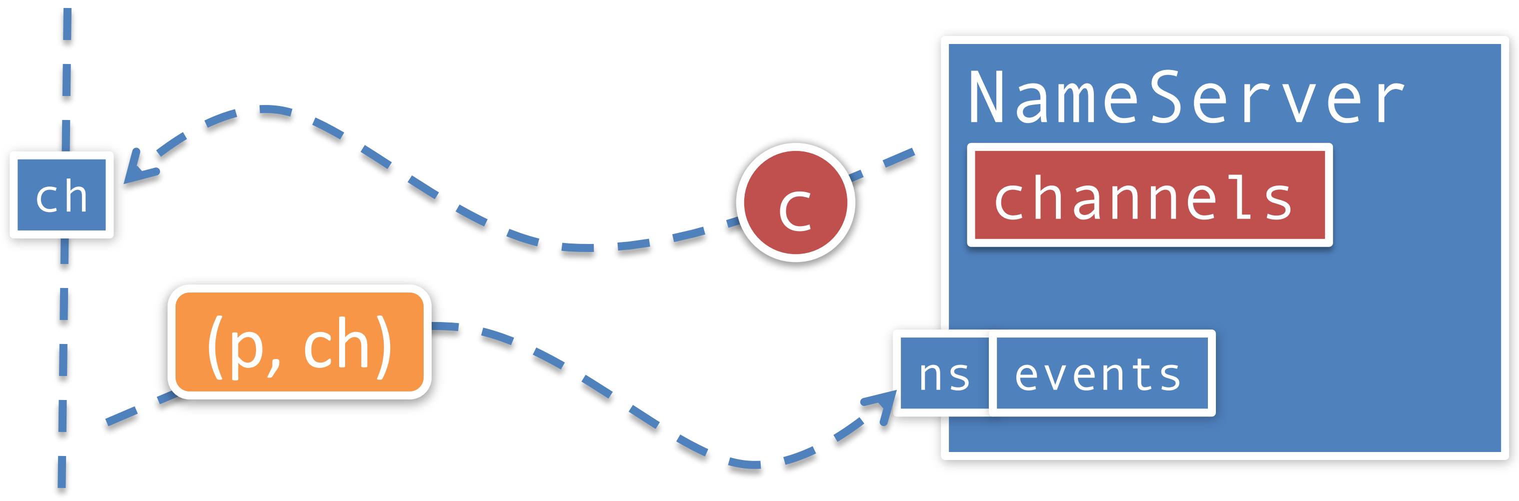
```
class NameServer
extends Iso[String] {
  val channels = Map[String, Channel[_]]()
  events onMatch {
    case name => sender ! channels(name)
  }
}
```



```
class NameServer
extends Iso[(String, Channel[Channel[_]])] {
  val channels = Map[String, Channel[_]]()
  events onMatch {
    case (name, ch) => ch ! channels(name)
  }
}
```



```
class NameServer
extends Iso[(String, Channel[Channel[_]])] {
  val channels = Map[String, Channel[_]]()
  events onMatch {
    case (name, ch) => ch ! channels(name)
  }
}
```



```scala
def open[T]: (Channel[T], Events[T])

trait Channel[T] {
  def !(x: T): Unit
}

trait Events[T] {
  def onEvent(f: T => Unit)
  def onMatch(f: PartialFunction[T, Unit])
  def forward(c: Channel[T])
}
```

```
class NameServer
extends Iso[(String, Channel[Channel[_]])] {
  val channels = Map[String, Channel[_]]()
  events onMatch {
    case (name, ch) => ch ! channels(name)
  }
}

type Req[T, S] = Channel[(T, Channel[S])]
```

```
class NameServer
extends Iso[(String, Channel[Channel[_]])] {
  val channels = Map[String, Channel[_]]()
  events onMatch {
    case (name, ch) => ch ! channels(name)
  }
}

type Req[T, S] = Channel[(T, Channel[S])]
def server[T, S](f: T => S): Req[T, S]
```

```scala
class NameServer
extends Iso[(String, Channel[Channel[_]])] {
  val channels = Map[String, Channel[_]]()
  events onMatch {
    case (name, ch) => ch ! channels(name)
  }
}

type Req[T, S] = Channel[(T, Channel[S])]
def server[T, S](f: T => S): Req[T, S] = {
  val (ch, events) = open[(T, Channel[S])]



}
```

```
class NameServer
extends Iso[(String, Channel[Channel[_]])] {
  val channels = Map[String, Channel[_]]()
  events onMatch {
    case (name, ch) => ch ! channels(name)
  }
}

type Req[T, S] = Channel[(T, Channel[S])]
def server[T, S](f: T => S): Req[T, S] = {
  val (ch, events) = open[(T, Channel[S])]
  events onMatch { case (x, c) => c ! f(x) }

}
```

```scala
class NameServer
extends Iso[(String, Channel[Channel[_]])] {
  val channels = Map[String, Channel[_]]()
  events onMatch {
    case (name, ch) => ch ! channels(name)
  }
}

type Req[T, S] = Channel[(T, Channel[S])]
def server[T, S](f: T => S): Req[T, S] = {
  val (ch, events) = open[(T, Channel[S])]
  events onMatch { case (x, c) => c ! f(x) }
  ch
}
```

```scala
class NameServer
extends Iso[(String, Channel[Channel[_]])] {
  val s = server(Map[String, Channel[_]]())
  events.forward(s)
}
```

```scala
type Req[T, S] = Channel[(T, Channel[S])]
def server[T, S](f: T => S): Req[T, S] = {
  val (ch, events) = open[(T, Channel[S])]
  events onMatch { case (x, c) => c ! f(x) }
  ch
}
```

```
type Req[T, S] = Channel[(T, Channel[S])]
def ?[T, S](r: Req[T, S], x: T): Events[S]
```

```
type Req[T, S] = Channel[(T, Channel[S])]
def ?[T, S](r: Req[T, S], x: T): Events[S] = {
  val (ch, events) = open[S]


}
```

```
type Req[T, S] = Channel[(T, Channel[S])]
def ?[T, S](r: Req[T, S], x: T): Events[S] = {
  val (ch, events) = open[S]
  r ! (x, ch)

}
```

```
type Req[T, S] = Channel[(T, Channel[S])]
def ?[T, S](r: Req[T, S], x: T): Events[S] = {
  val (ch, events) = open[S]
  r ! (x, ch)
  events
}
```

```
class Client(val ns: Req[String, Channel[_]])
extends Iso[Unit] {
  val response = server ? “p”
  response.onEvent(println)
}
```

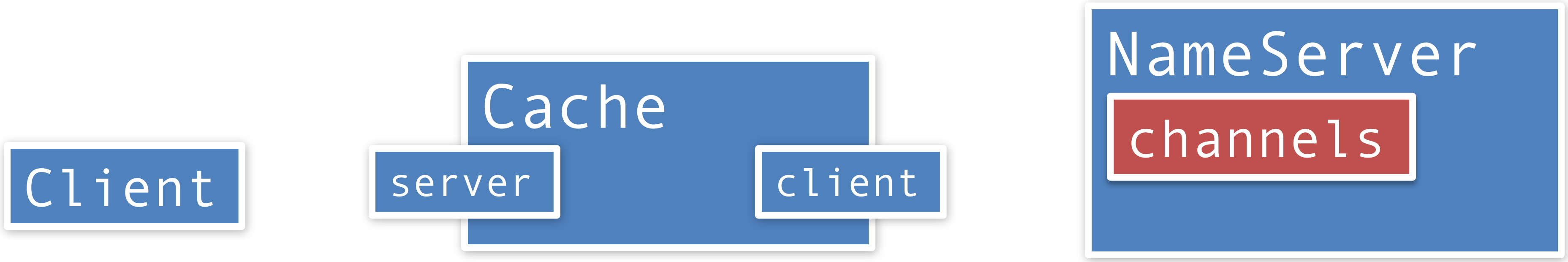
```
type Req[T, S] = Channel[(T, Channel[S])]
def ?[T, S](r: Req[T, S], x: T): Events[S] = {
  val (ch, events) = open[S]
  r ! (x, ch)
  events
}
```

```scala
class Cache(val ns: Req[String, Channel[_]])
extends Iso[(String, Channel[Channel[_]])] {







}
```

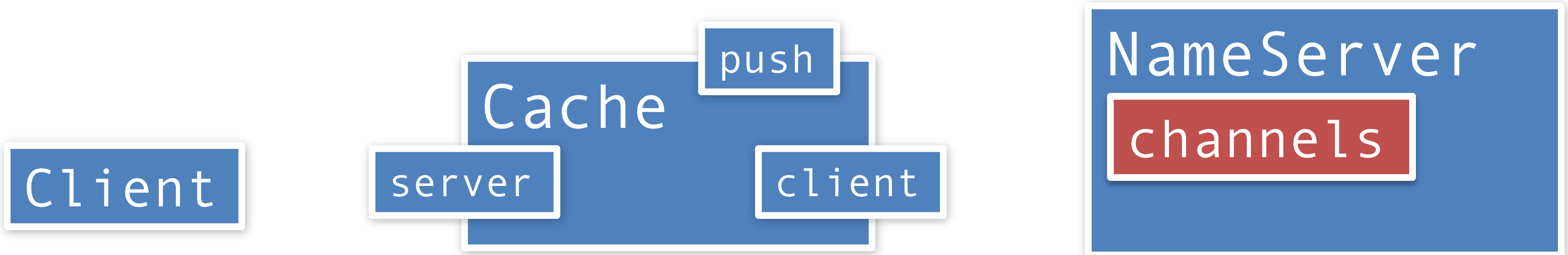
```
class Cache(val ns: Req[String, Channel[_]])
extends Iso[(String, Channel[Channel[_]])] {
  var cached: Channel[_] = null
  events.forward(server(x => cached))









}
```

Cache
Client
server

```
class Cache(val ns: Req[String, Channel[_]])
extends Iso[(String, Channel[Channel[_]])] {
  var cached: Channel[_] = null
  events.forward(server(x => cached))

  val response = server ? "p"
  response.onEvent(c => cached = c)



}
```



```
class Cache(val ns: Req[String, Channel[_]])
extends Iso[(String, Channel[Channel[_]])] {
  var cached: Channel[_] = null
  events.forward(server(x => cached))

  val response = server ? “p”
  response.onEvent(ch => cached(“p”) = ch)

  open[Req[String, Channel[_]]].events.onEvent(
    ns => (ns ? “p”).onEvent(c => cached = c))
}
```



Systems can compose

Broadcast

```
def bcst[T](s: Set[Channel[T]]): Channel[T]
```

Systems can compose

Broadcast

```
def bcst[T](s: Set[Channel[T]]): Channel[T]
```

CRDT

```
def crdt[T, D](bcst: Channel[T],
  update: (D, T) => D): (T => Unit, () => D)
```

Thank you!