A proposal of new concurrency model for Ruby 3

Koichi Sasada
ko1@heroku.com

People love “Concurrency”

dRuby in the last century.
Masatoshi SEKI @m_seki

JA

Improved scalability by relaxing the GVL
Charlie Gracie @crgracie

EN

ErRuby: Ruby on Erlang/OTP
Lin Yu Hsiang @johnlinvc

EN

concurrent-ruby and how it is making Rails concurrent
Vipul A M @vipulnsward

EN

How to create multiprocess server on Windows with Ruby
Ritta Narita @narittan

EN

Ruby Concurrency compared
Anil Wadghule @anildigital

EN

Concurrent RubyKaigi

(at least, there are two parallel sessions)

Why people love (to discuss) “Concurrency”?

- Performance by “Parallel” execution to utilize multiple-cores

About this presentation

- Show “Why difficult multi-threads programs”
- Propose new concurrent and parallel mechanism idea named **“Guild”**
  - For Ruby 3

Koichi Sasada

- A programmer living in Tokyo, Japan
- Ruby core committer since 2007
  - YARV, Fiber, … (Ruby 1.9)
  - RGenGC, RincGC (Ruby 2…)

Koichi is an Employee

heroku

Difficulty of Multi-threads programming

Programming language evolution

- Trade-off: Performance v.s. Safety/Easily
  - Performance: making faster programs
  - Safety: making bug-free programs
  - Easily: making programs happily

Two example
C language

- String manipulation with pointers
- Memory management without GC

String manipulation with pointers

- C: Using raw pointers to manipulate pointers
  - Good: all-purpose and fast
  - Bad: easy to introduce bugs
    - Generate strange behavior
    - Cause abnormal termination
- Ruby: Wrap with String class
  - Good: Easy to use
  - Bad: slower than C in some cases

Object management without GC

- C: Free memory objects manually
  - Good: full control (target, timing and so on)
  - Bad: easy to introduce bugs (double-free/memory-leak, …)
- Ruby: Automatic collection with GC
  - Good: nothing to care about object collection
  - Bad: introduce some overhead

Catch the two examples

- Ruby chose “safety/easily” approach
  - I believe nobody want to use pointers to manipulate String objects and want to manage objects manually anymore
  - Because …
    - Nowadays computers is enough faster.
    - Implementation techniques overcome performance penalties (like generational GC and so on)

Current concurrent primitive Multi-threads programming

- Muilti-threads programming is difficult
  - **Introduce data race, race condition**
  - Introduce deadlock, livelock
  - Difficulty on debugging because of nondeterministic behavior
    - difficult to reproduce same problem

Difficult to make correct (bug-free) programs

- Difficult to tune performance

Difficult to make fast programs

Data race and race condition

- Bank amount transfer example
  - Quoted from Race Condition vs. Data Race http://blog.regehr.org/archives/490

```
def transfer1 (amount, account_from, account_to)
  if (account_from.balance < amount) return NOPE
  account_to.balance += amount
  account_from.balance -= amount
  return YEP
end
```

Data race

- "account_to.balance += amount" has **Data-race**
  - Assume two threads (T1 and T2) invoke this methods with same bank accounts

```
# interleave two threads (T1: amount = 100, T2: amount = 200)
  T1: t1 = account_to.balance # t1 = 10
  T2: t2 = account_to.balance # t2 = 10
  T1: account_to.balance = t1 + 100 #=> 110
  T2: account_to.balance = t2 + 200 #=> 210 (expected: 310)
```

Race condition

- To avoid data-race with the lock
- But there is another problem yet

```
# Lock with “Thread.exclusive”
def transfer2 (amount, account_from, account_to)
  if (account_from.balance < amount) return NOPE
  Thread.exclusive{ account_to.balance += amount }
  Thread.exclusive{ account_from.balance -= amount }
  return YEP
end
```

Race condition

- To avoid data-race with the lock
- But there is another problem yet

```
# T1 amount = 100, T2 amount = 200, account_from.balance = 250
T1: if (account_from.balance (== 250) < 100) return NOPE # OK, go through
T2: if (account_from.balance (== 250) < 200) return NOPE
T2:  Thread.exclusive{ account_to.balance += 200 }
T2:  Thread.exclusive{ account_from.balance -= 200 } #=> 250-200 => 50
T1:  Thread.exclusive{ account_to.balance += 100 }
T1:  Thread.exclusive{ account_from.balance -= 100 } #=> 50 - 100 => negative number!!
```

Final solution

- Lock whole of method

```
def transfer1 (amount, account_from, account_to)
  Thread.exclusive{
    if (account_from.balance < amount) return NOPE
    account_to.balance += amount
    account_from.balance -= amount
    return YEP
  }
end
```

Another example
Multi-thread quiz

- What happen on this program?

```
ary = [1, 2, 3]
t1 = Thread.new{
  ary.concat [4, 5, 6]
}
t2 = Thread.new{
  p ary # what's happen?
}.join
```

(1) [1, 2, 3]
(2) [1, 2, 3, 4, 5, 6]
(3) (1) or (2)

Another example
Multi-thread quiz

- Answer: (4) depends on an interpreter

```
ary = [1, 2, 3]
t1 = Thread.new{
  ary.concat [4, 5, 6]
}
t2 = Thread.new{
  p ary # what's happen?
}.join
```

On MRI, (3) is correct

It will shows
[1, 2, 3] or
[1, 2, 3, 4, 5, 6]
(depends on thread switching timing)

Another example
Multi-thread quiz

- Answer: (4) depends on interpreter

```
ary = [1, 2, 3]
t1 = Thread.new{
  ary.concat [4, 5, 6]
}
t2 = Thread.new{
  p ary # what's happen?
}.join
```

On JRuby:

It can cause Java exception because "Array#concat" is not thread safe

On JRuby …

```
# similar program
h = Hash.new(0)
NA = 1_000
10_000.times{
  ary = []
  (1..10).each{
    Thread.new{
      NA.times{|i|
        ary.concat [i]
      }
    }
  }
  t2 = Thread.new{
    s = ary.dup
  }.join
}
```

Unhandled Java exception: java.lang.NullPointerException

```
java.lang.NullPointerException: null
          rbInspect at org/jruby/RubyBasicObject.java:1105
            inspect at org/jruby/RubyObject.java:516
          inspectAry at org/jruby/RubyArray.java:1469
            inspect at org/jruby/RubyArray.java:1497
        cacheAndCall at org/jruby/runtime/callsite/CachingCallSite.java:293
               call at org/jruby/runtime/callsite/CachingCallSite.java:131
       block in t.rb at t.rb:17
         yieldDirect at org/jruby/runtime/CompiledIRBlockBody.java:156
       yieldSpecific at org/jruby/runtime/IRBlockBody.java:73
       yieldSpecific at org/jruby/runtime/Block.java:136
             times at org/jruby/RubyFixnum.java:291
        cacheAndCall at org/jruby/runtime/callsite/CachingCallSite.java:303
          callBlock at org/jruby/runtime/callsite/CachingCallSite.java:141
               call at org/jruby/runtime/callsite/CachingCallSite.java:145
              <top> at t.rb:3
 invokeWithArguments at java/lang/invoke/MethodHandle.java:599
               load at org/jruby/ir/Compiler.java:111
          runScript at org/jruby/Ruby.java:833
          runScript at org/jruby/Ruby.java:825
         runNormally at org/jruby/Ruby.java:760
         runFromMain at org/jruby/Ruby.java:579
       doRunFromMain at org/jruby/Main.java:425
         internalRun at org/jruby/Main.java:313
                run at org/jruby/Main.java:242
               main at org/jruby/Main.java:204
```

jruby 9.1.2.0 (2.3.0) 2016-05-26 7357c8f OpenJDK 64-Bit Server VM 24.95-b01 on 1.7.0_101-b00 +jit [linux-x86_64]
On 8 hardware threads machine

Difficulty of multi-threads programs

- We need to synchronize all sharing mutable objects correctly (or completely)
  - We need to know which methods are thread-safe and which methods are not.
  - Easy to track all of shared mutable objects on small program (like previous examples)
  - Difficult to track on big programs, especially on programs using gems
- We need to check **all of source codes**, or believe **library documents** (and we need to keep maintain documents correctly)

Difficulty of multi-threads programs (cont.)

- It requires “completeness”
  - If we forget to synchronize for **only one** mutable shared object, it will cause unexpected behavior.
- For debugging, it is difficult to find out the bugs
  - Backtrace may not work well because the problem may be placed on another line.
  - Bugs don’t appear frequently with small data (test-data)
  - Difficult to reproduce issue because of nondeterministic behavior (depends on timing, etc).

FYI:
Why MRI Array#concat is thread-safe?

- MRI uses GVL (Giant/Global VM Lock) to control thread switching timing and C methods (such as Array#concat) are working atomically.
- GVL prohibits parallel thread execution (BAD), however it avoids several severe issues (GOOD).

Thread programming: Performance tuning issue

```
a1 = []; a2 = []
NA = 10_000_000
t1 = Thread.new{
  NA.times{|i| a1 << i }
}.join
t2 = Thread.new{
  NA.times{|i| a2 << i }
}.join
```

Serial program:

real 0m8.568s

user 0m37.816s

sys 0m5.530s

on JRuby

Thread programming: Performance tuning issue

```
a1 = []; a2 = []
NA = 10_000_000
t1 = Thread.new{
  NA.times{|i| a1 << i }
}
t2 = Thread.new{
  NA.times{|i| a2 << i }
}
t1.join; t2.join
```

Parallel program

(2 threads):

real 0m6.411s

user 0m20.527s

sys 0m7.798s

Thread programming: Performance tuning issue

```
a1 = []; a2 = []
NA = 10_000_000
m1, m2  = Mutex.new, Mutex.new
t1 = Thread.new{
  NA.times{|i| m1.synchronize{ a1 << i }}
}
t2 = Thread.new{
  NA.times{|i| m2.synchronize{ a2 << i }}
}
t1.join; t2.join
```

Parallel program with a useless lock 1

(2 threads):

real 0m10.264s

user 0m38.370s

sys 0m4.406s

Thread programming: Performance tuning issue

```
a1 = []; a2 = []
NA = 10_000_000
m = Mutex.new
t1 = Thread.new{
  NA.times{|i| m.synchronize{ a1 << i }}
}
t2 = Thread.new{
  NA.times{|i| m.synchronize{ a2 << i }}
}
t1.join; t2.join
```

Parallel program with a useless lock 2

(2 threads):

real 0m15.163s

user 0m45.317s

sys 0m9.658s

Performance tuning issue

	Execution time
Serial program	**8.568s**
Parallel program	**6.411s**
Parallel program with a useless lock 1	**10.264s**
Parallel program with a useless lock 2	**15.163s**

Thread programming:
Performance tuning issue

We need to use just correct number locks

Not enough → unexpected behavior
Too much → performance penalty

FYI: synchronization mechanism

- Many synchronization mechanisms…
  - Mutual exclusion (Mutex), monitor, critical section
  - Transactional memory (optimistic lock)
  - Atomic instructions
  - Synchronized Queue
  - …
  - Research on many lightweight lock algorithms
- They assume we can use them correctly

Overcome thread difficulty

Key idea

Problem:

Easy to share mutable objects

Idea:

Do not allow to share mutable objects between concurrent entities (like threads), **without any restriction**

Study from other languages

- Shell script with pipes, Racket (Place)
  - Copy mutable data between processes w/ pipes
- Erlang/Elixir
  - Do not allow mutable data
- Clojure
  - Basically do not allow mutable data
  - Special data structure to share mutable objects
  - Note that it can share mutable objects on Java layer

NOTE: we do not list approaches using “type system”

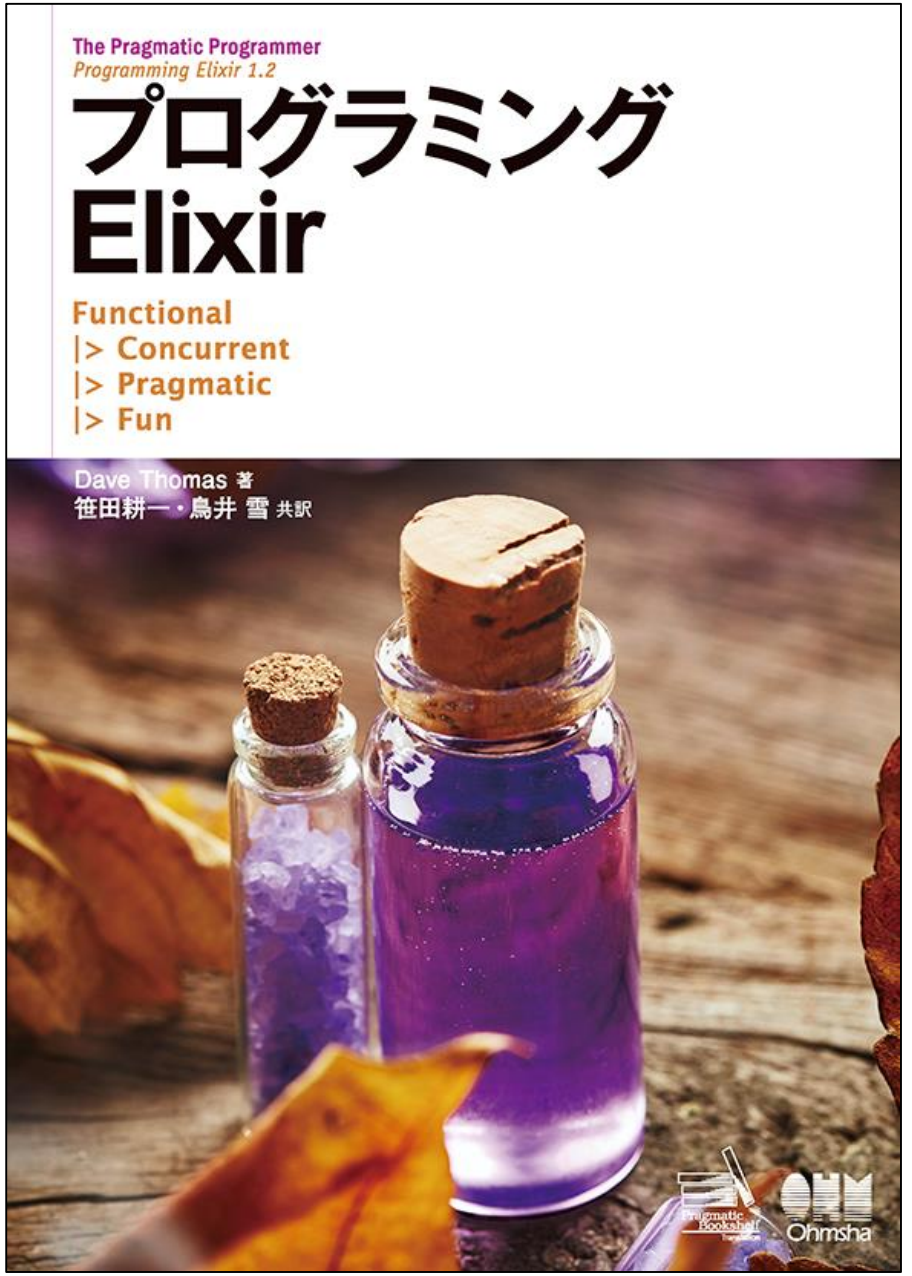
Don’t you know Elixir language?

Programming Elixir 1.2 by Dave Thomas

邦訳:プログラミングElixir
笹田耕一・鳥井雪共訳 2016/08/19

You can buy it TODAY!!

サイン会は明日13時らしいです



Summary of approaches

- Communication with copied data (shell scripts)
  - Good: we don’t need locks
  - Bad: copy everything is **slow**
- Prohibit mutable objects
  - Good: we don’t need locks
  - Bad: Ruby utilizes **many “write” operations**. Unacceptable.
- Provide special data structure to share mutable objects
  - Good: we don’t need locks (who don’t use such special data structures)
  - Bad: Difficult to use special data structures.

Background was finished

Our goal for Ruby 3

- **We need to keep compatibility** with Ruby 2.
- We **shouldn’t consider** about locks any more.
- We **can share** objects with copy, but **copy operation should be fast.**
- We **should share objects** if we can.
- We can **provide special objects** to share mutable objects like Clojure if we really need speed.

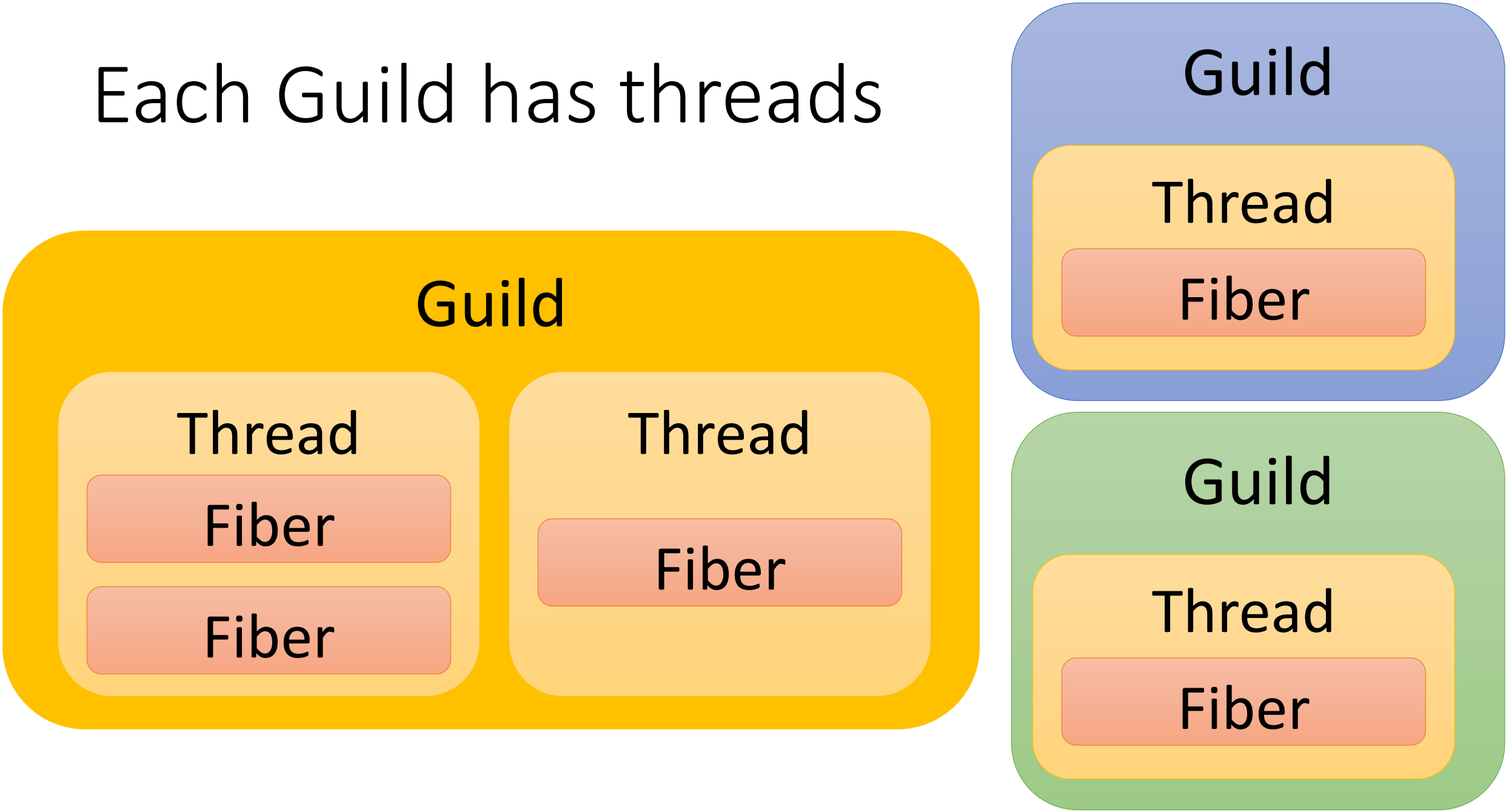
“Guild”

New concurrency model for Ruby 3

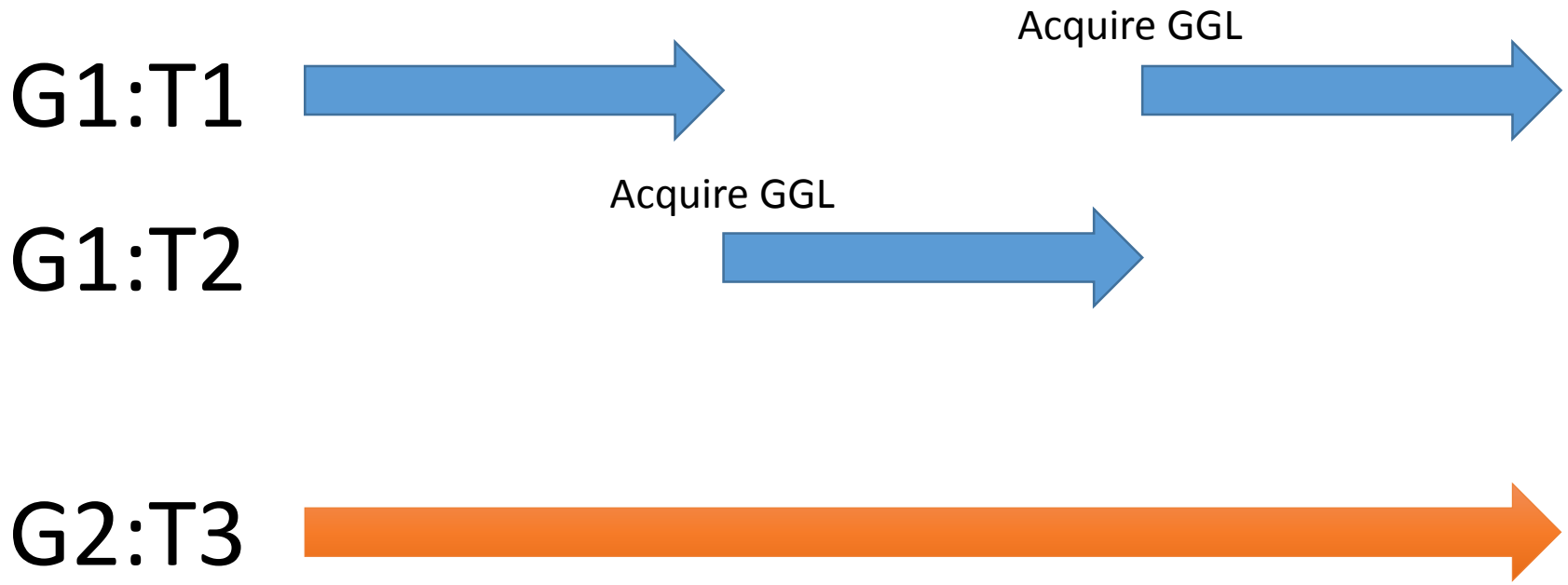
Guild: New concurrency abstraction

- Guild has at least one thread (and a thread has at least one fiber)
  - Threads in different guilds **can run in parallel**
  - Threads in a same guild **can not run in parallel** because of GVL (or GGL: Giant Guild Lock)

Each Guild has threads

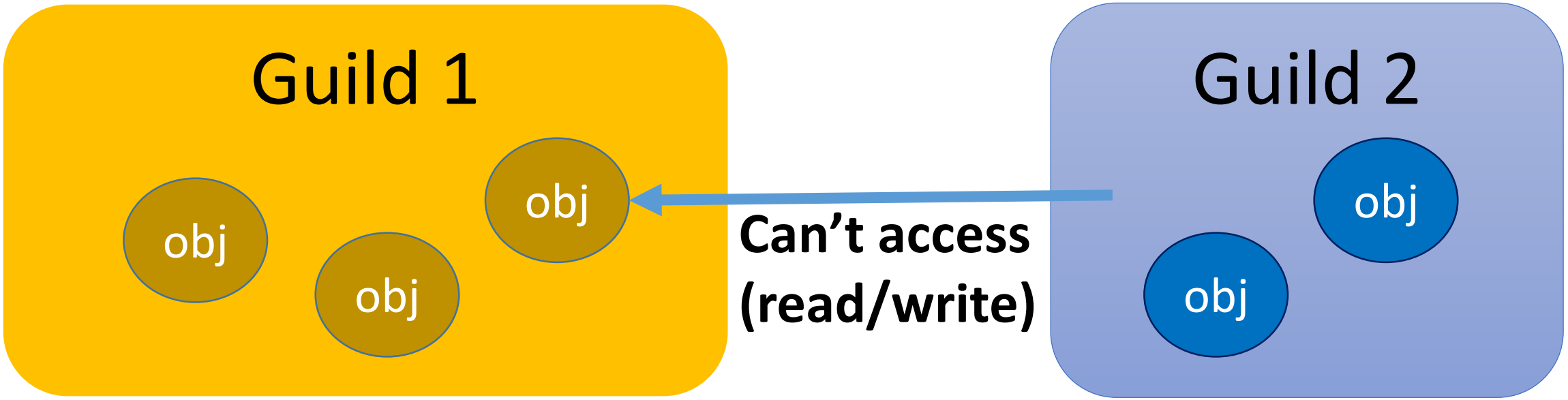


Threads in different guilds can run in Parallel



Guild and objects:
All objects have their own membership

- All of mutable objects should belong to only one Guild (all mutable objects are member of one guild)
- Other guilds can not touch objects



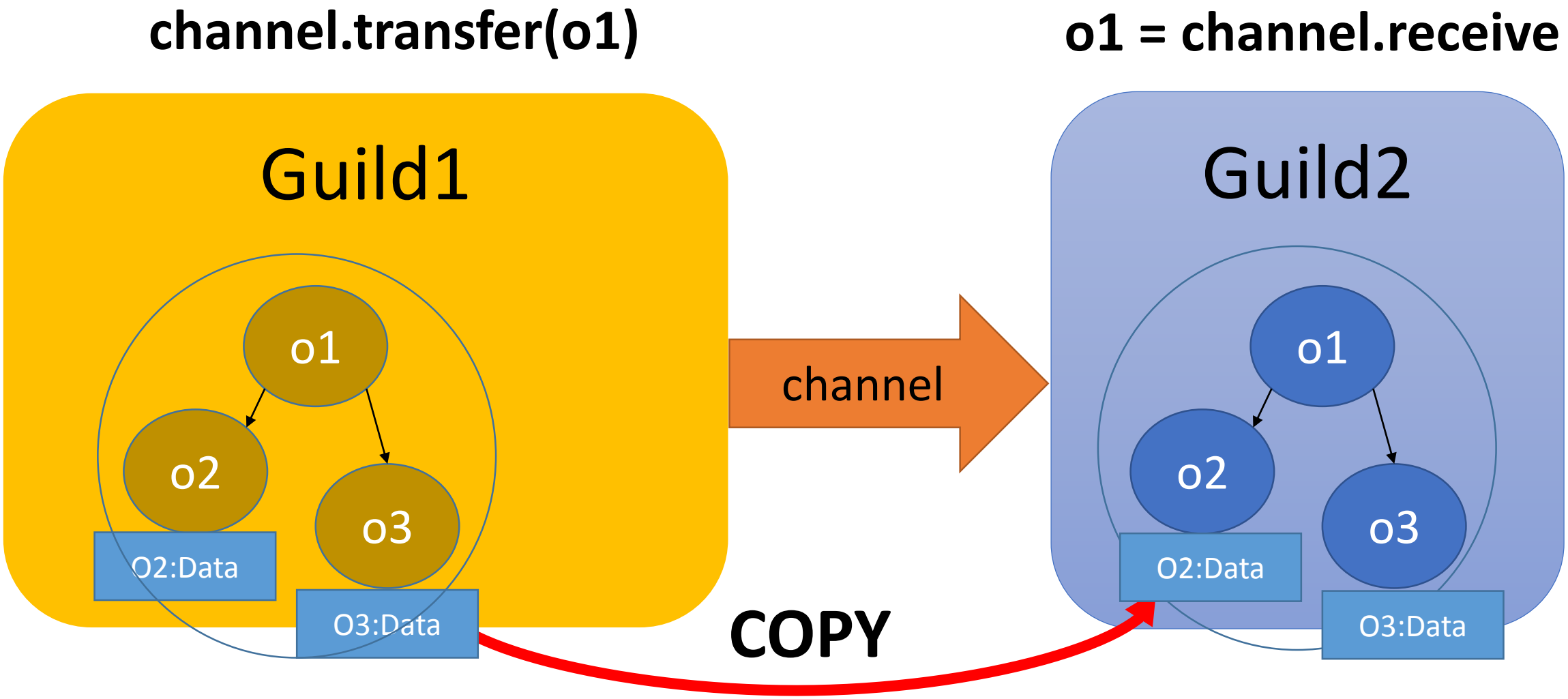
Inter Guilds communication

- We have Guild::Channel to communicate each other
- 2 communication methods
  1. **Copy**
  2. **Transfer membership** or **Move** in short

Copy using Channel

- Guild::Channel#transfer(obj) send **deep copied** object(s) to a destination guild.
- dRuby and multi-process system use this kind of communication

Copy using Channel



Move using Channel [New technique!!]

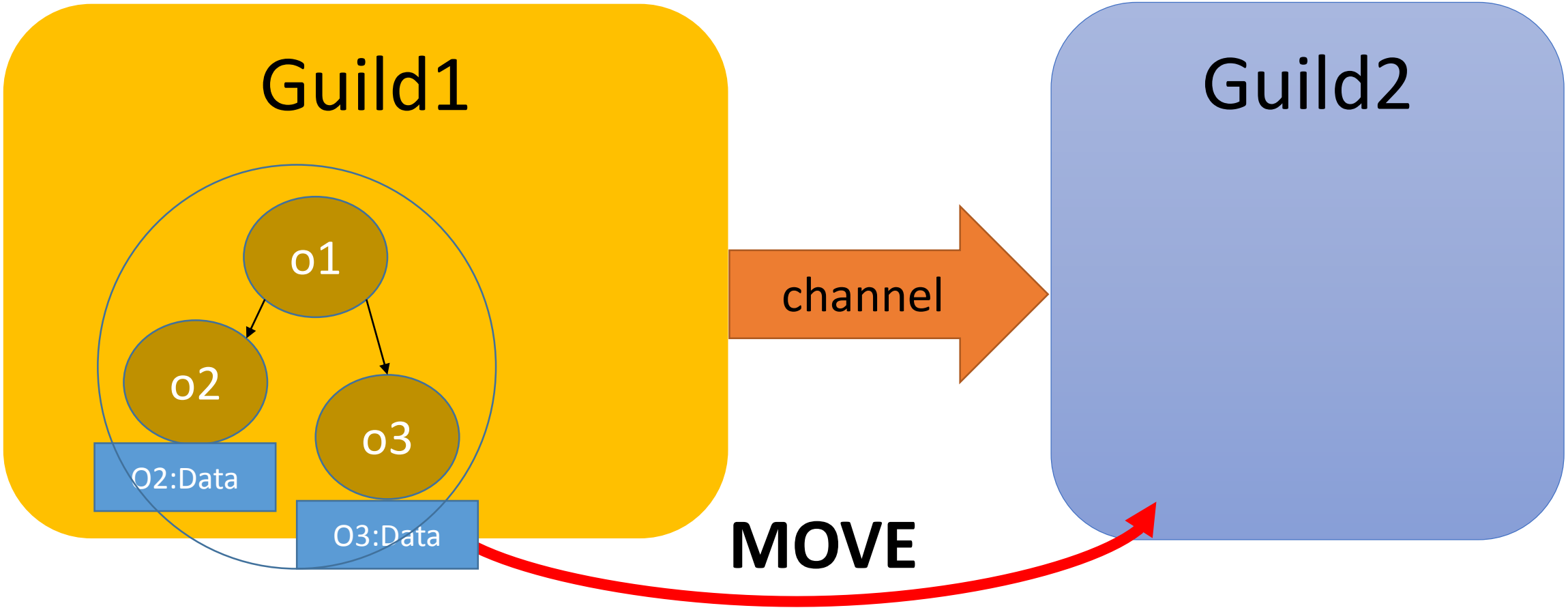
- Guild::Channel#transfer_membership(obj) change the membership of object(s)
  - Leave from the source guild
  - Join to the destination guild
- Accessing from the source guild is invalidated
  - Cause exceptions and so on
  - ex)

```
obj = “foo”
ch.transfer_membership(obj)
obj.upcase        #=> Error!!
p(obj)            #=> Error!!
```

Move using Channel

channel.transfer_membership(o1)

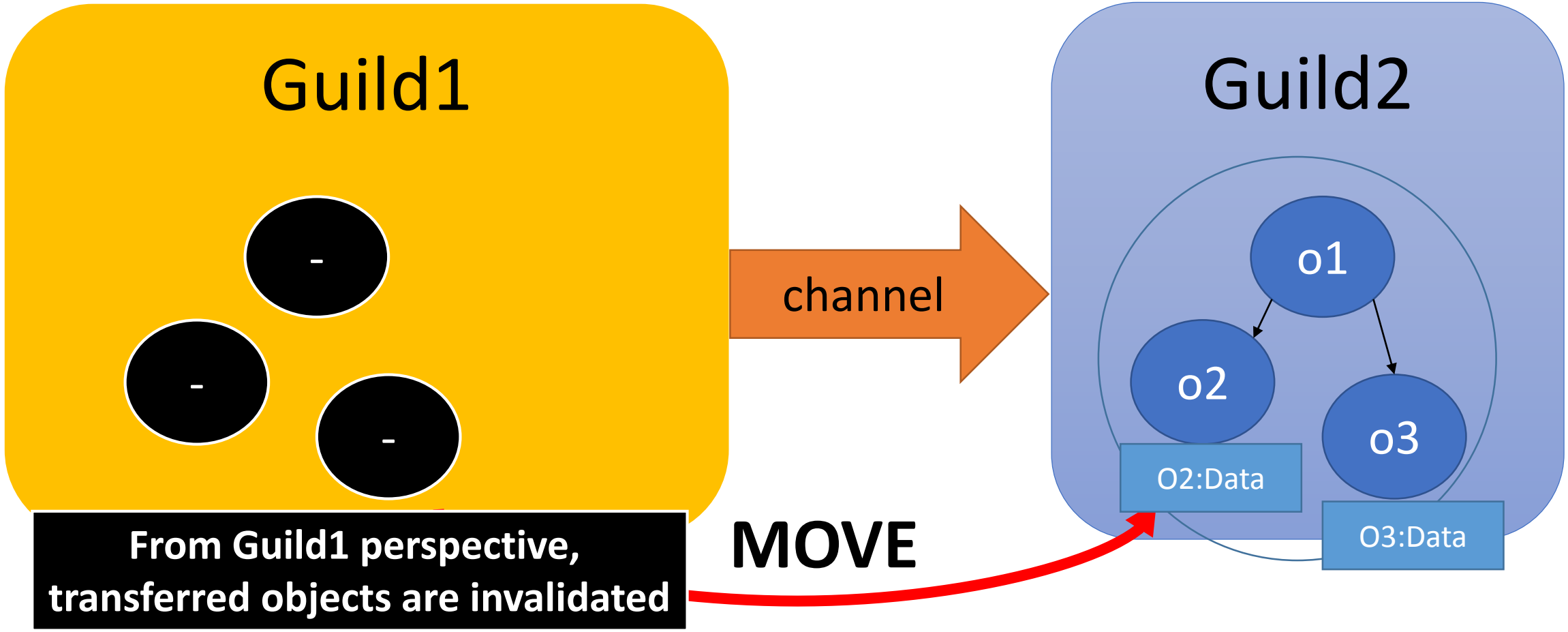
o1 = channel.receive



Move using Channel

channel.transfer_membership(o1)

o1 = channel.receive

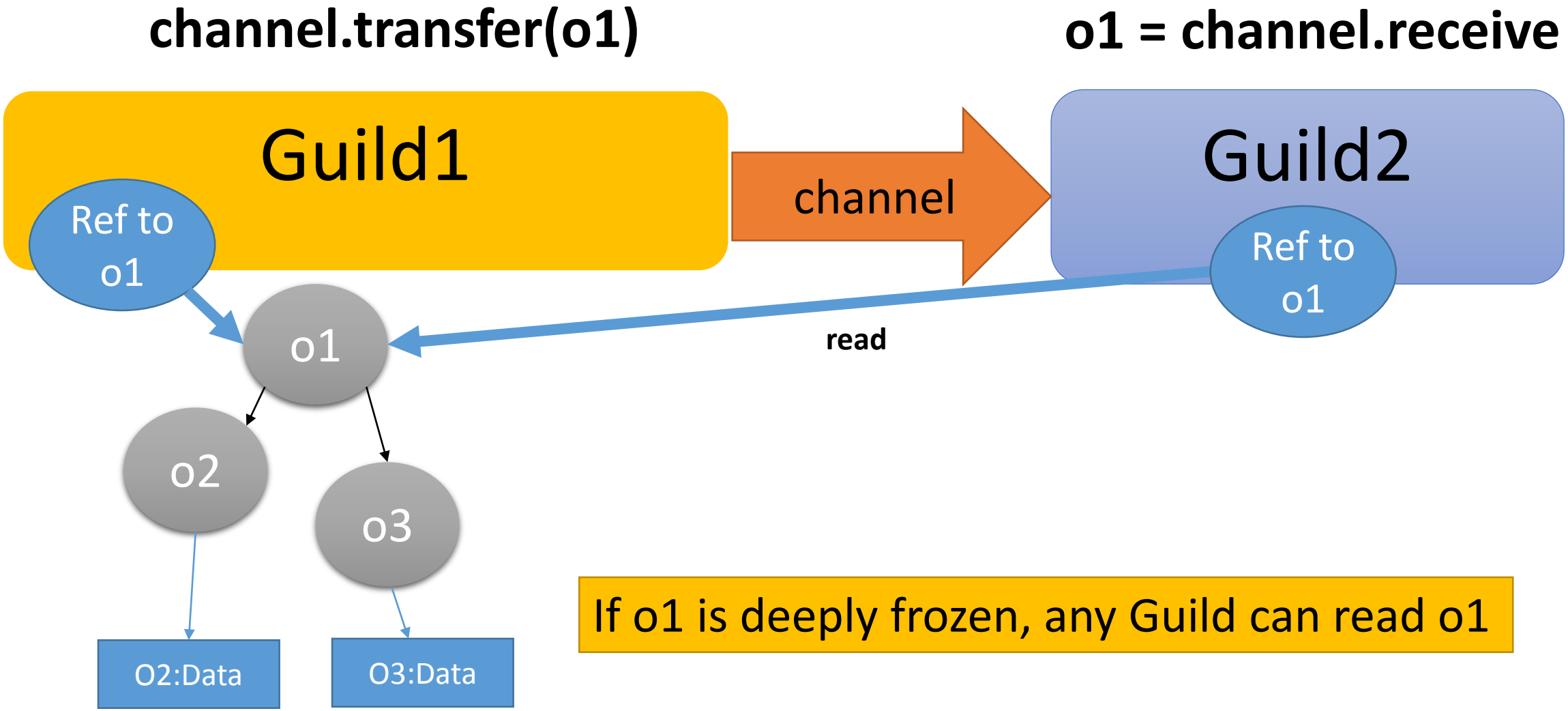


Sharing immutable objects

- Deeply frozen objects can be shared with any guilds
  - [1, 2, 3].freeze: **deeply frozen**
  - [1, Object.new, 3].freeze: not deeply frozen because 2nd element is not frozen
- We only need to send references (very lightweight, like thread-programming)
- **Numeric objects, symbols, true, false, nil** are immutable and deeply frozen (from Ruby 2.0, 2.1, 2.2).

Share immutable objects

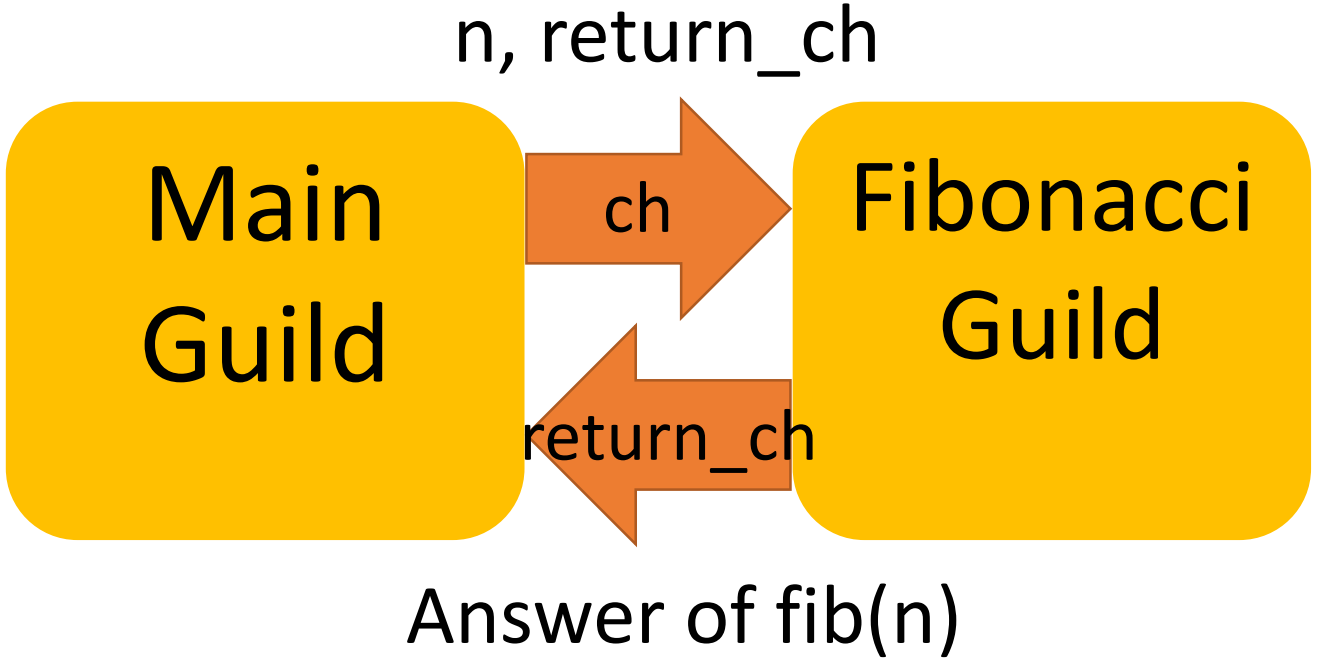
We can share reference to immutable objects



If o1 is deeply frozen, any Guild can read o1

Use-case 1: master – worker type

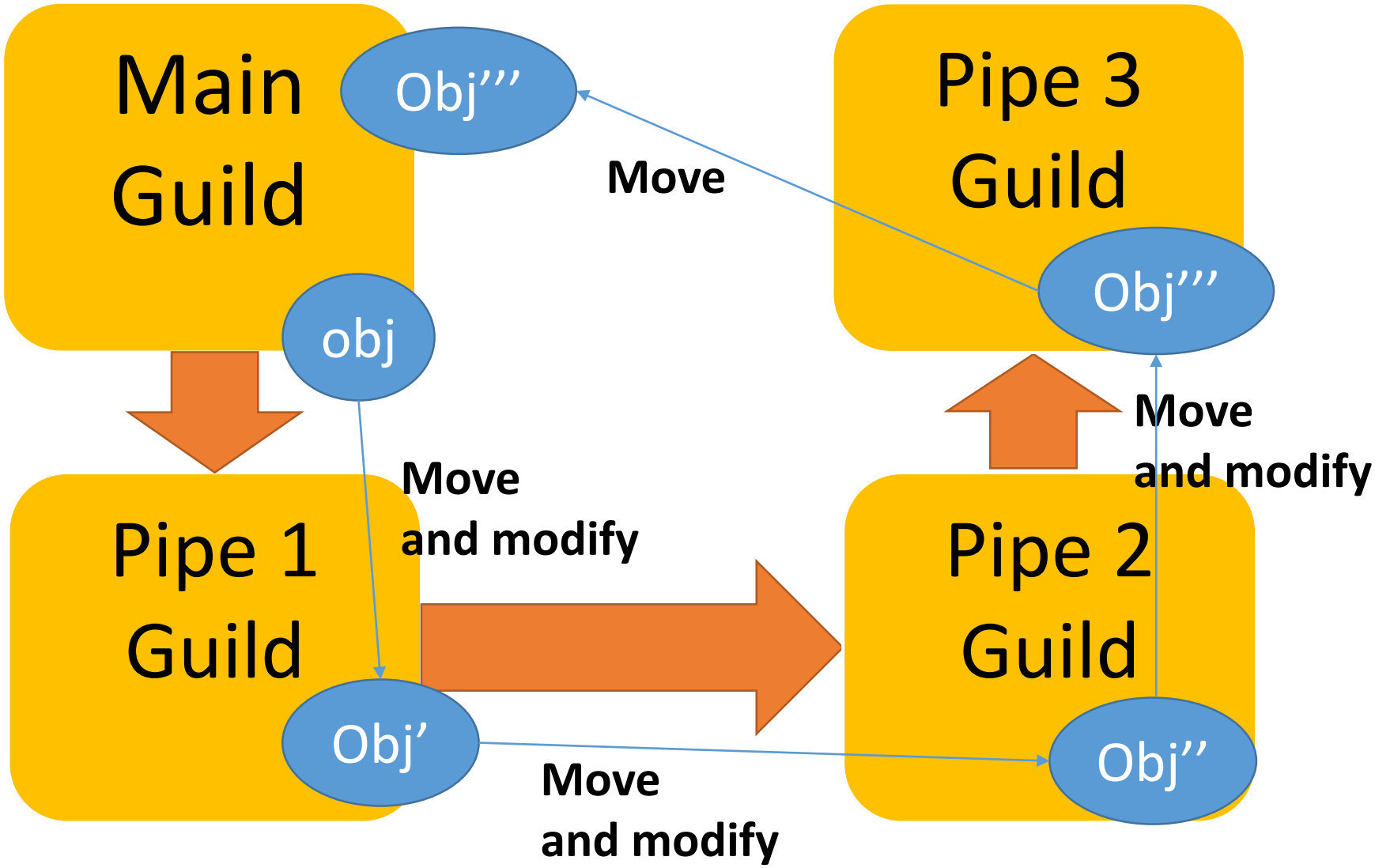
```
def fib(n) ... end
g_fib = Guild.new(script: %q{
  ch = Guild.default_channel
  while n, return_ch = ch.receive
    return_ch.transfer fib(n)
  end
})

ch = Guild::Channel.new
g_fib.transfer([3, ch])
p ch.receive
```



NOTE: Making other Fibonacci guilds, you can compute fib(n) in parallel

Use-case 2: pipeline

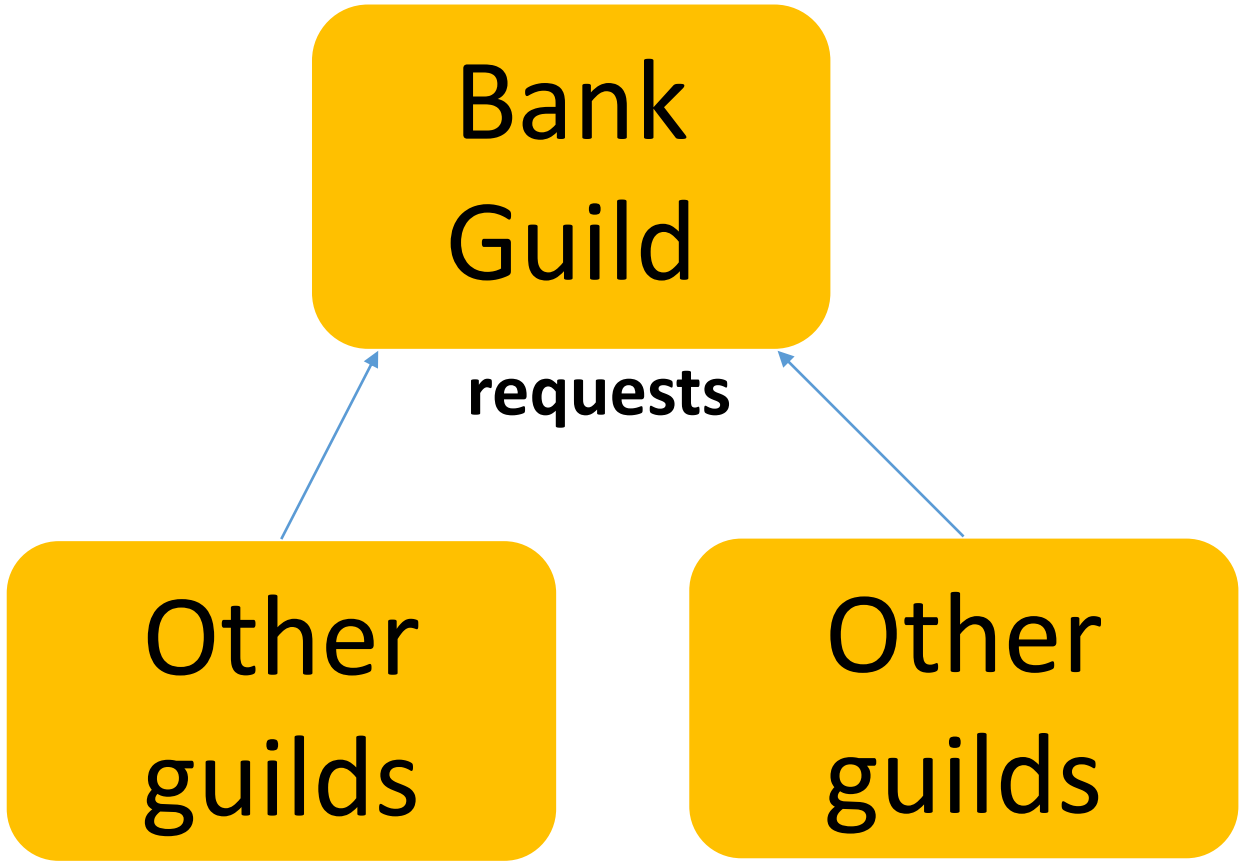
```
result_ch = Guild::Channel.new
g_pipe3 = Guild.new(script: %q{
  while obj = Guild.default_channel.receive
    obj = modify_obj3(obj)
    Guild.argv[0].transfer_membership(obj)
  end
}, argv: [result_channel])
g_pipe2 = Guild.new(script: %q{
  while obj = Guild.default_channel.receive
    obj = modify_obj2(obj)
    Guild.argv[0].transfer_membership(obj)
  end
}, argv: [g_pipe3])
g_pipe1 = Guild.new(script: %q{
  while obj = Guild.default_channel.receive
    obj = modify_obj1(obj)
    Guild.argv[0].transfer_membership(obj)
  end
}, argv: [g_pipe2])

obj = SomeClass.new

g_pipe1.transfer_membership(obj)
obj = result_ch.receive
```



Use-case:
Bank example

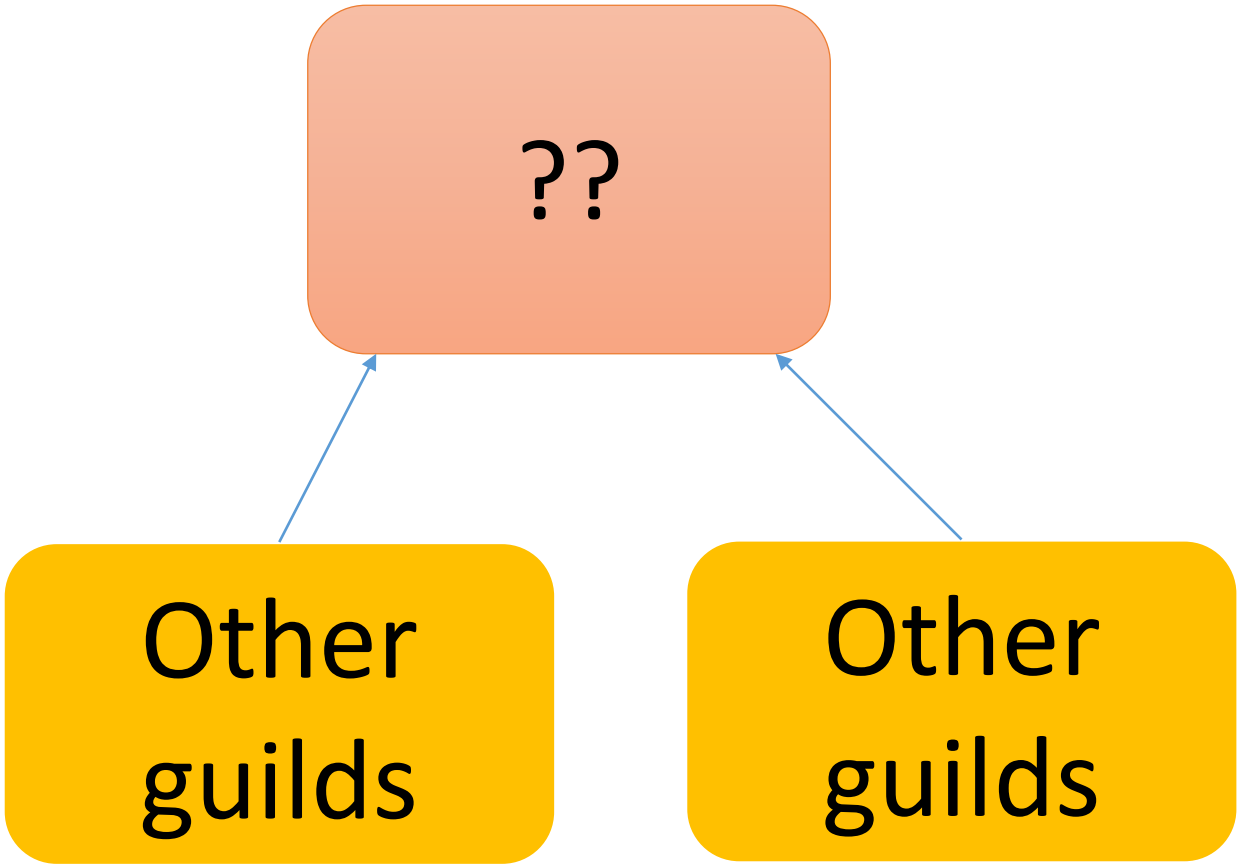
```
g_bank = Guild.new(script: %q{
  while account_from, account_to, amount,
            ch = Guild.default_channel.receive
    if (Bank[account_from].balance < amount)
      ch.transfer :NOPE
    else
      Bank[account_to].balance += amount
      Bank[account_from].balance -= amount
      ch.transfer :YEP
    end
  end
})
…
```

Only bank guild maintains bank data



Use-case:
Introduce special data structure

- Ideas of special data structure to share mutable objects
  - Use external RDB
  - In process/external Key/value store
  - Software transactional memory
  - …



Summary of use cases

- Making multiple workers and compute in parallel
  - Requests and responses are communicate via channels
  - You can send it with copy or move
  - Maybe web application can employ this model
- Making Pipeline structures and compute in parallel
  - Each task has own Guild
  - Receive target object, modify it and send it next pipeline
  - You will send it with move (transfer membership)
  - It will help applications like applying several filters for input data
- Own responsibility by one Guild
  - All accesses are managed by one responsible Guild
  - If you want to share mutable objects, we need special data structures
  - External RDBs or key/value stores are also good idea for this purpose

Communication strategy

[Upper is better]

- Passing deeply frozen objects
- Copy mutable objects
- If you have performance problem, move (transfer membership) mutable objects
- If you have performance problem too, use special data structure to share mutable objects

Compare between Thread model and Guild model

- On threads, it is **difficult to find out** which objects are shared mutable objects
- On Guilds, there are **no shared mutable objects**
  - If there are special data structure to share mutable objects, we only need to check around this code
  - Encourage “Safe” and “Easy” programming

Compare between Thread model and Guild model

- On threads, inter threads communication is very fast.
- On Guilds, inter guilds communication introduce overhead
  - “Move” (transfer membership) technique can reduce this kind of overheads

Trade-off: Performance v.s. Safety/Easily

Which do you want to choose?

Digression: The name of “Guild”

- “Guild” is good metaphor for “object’s membership”
- Check duplication
  - First letter is not same as other similar abstractions
    - For variable names
    - P is for Processes, T is for Threads, F is for Fibers
  - There are no duplicating top-level classes and modules in all of rubygems

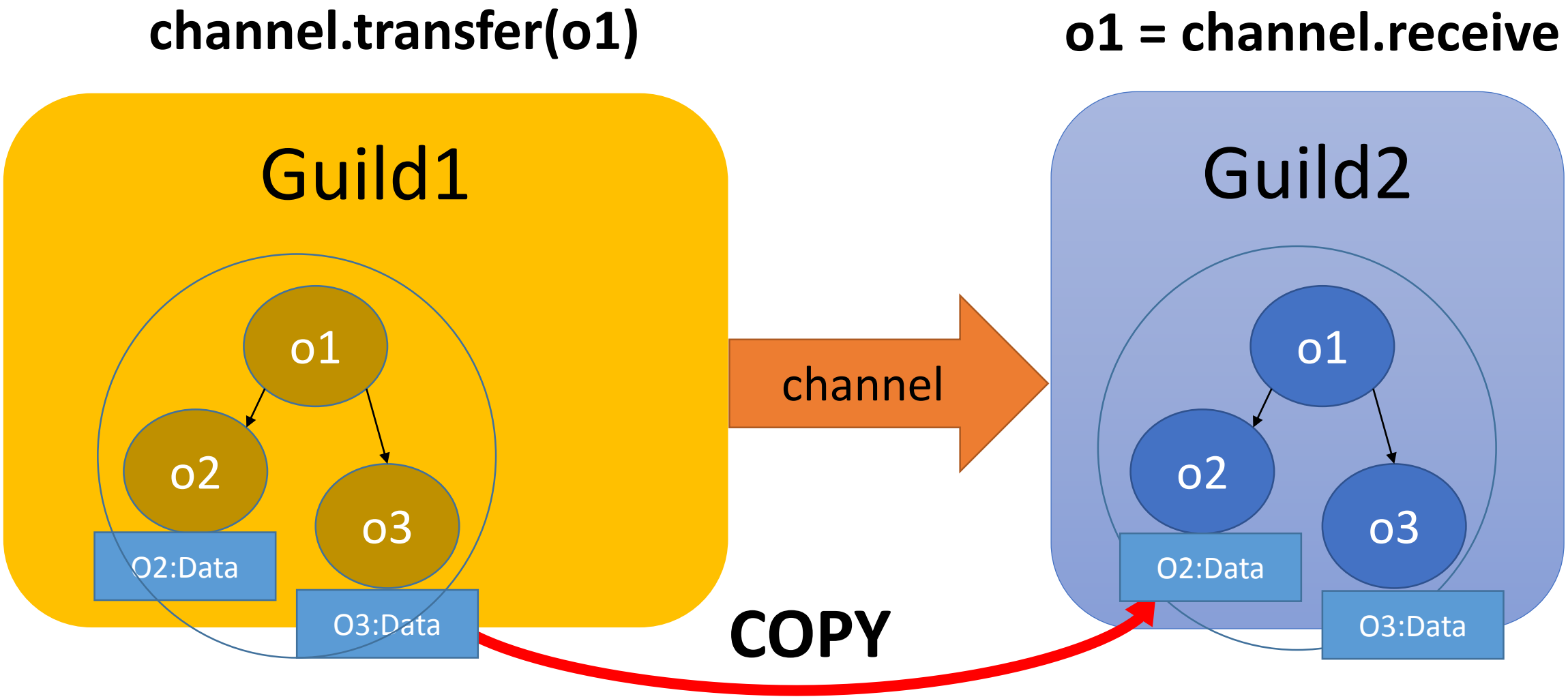
Implementation of “Guild”

- How to implement inter Guilds communication
- How to isolate process global data

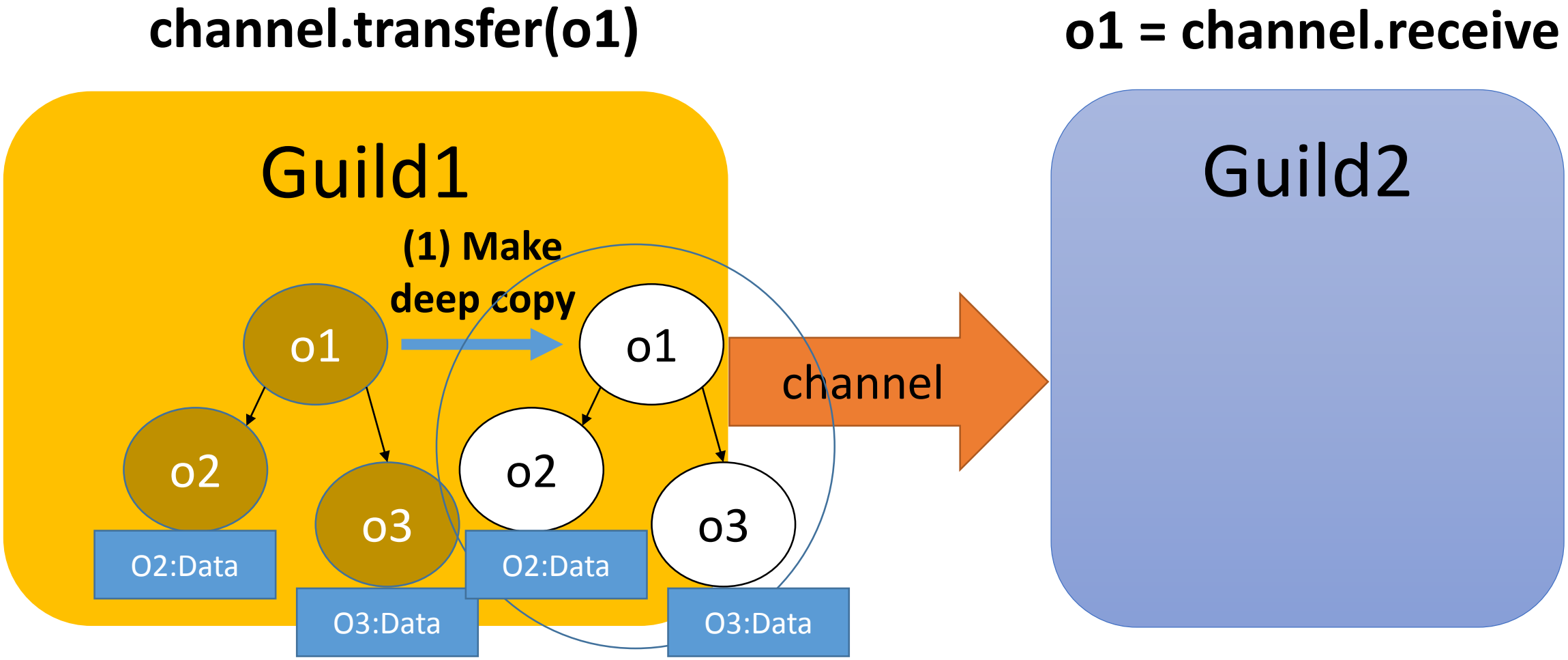
How to implement inter Guilds communication

- Copy
- Move (transfer membership)

Copy using Channel



Copy using Channel Implementation

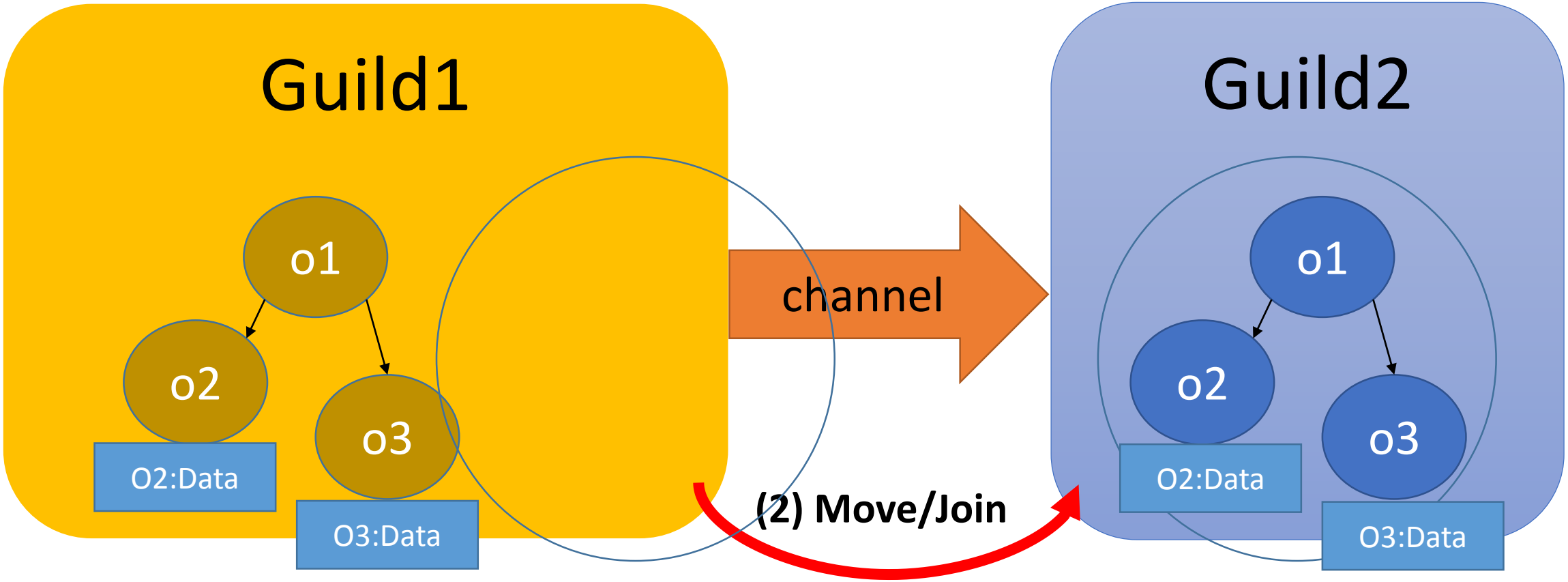


Copy using Channel Implementation

We can use CoW technique for data

channel.transfer(o1)

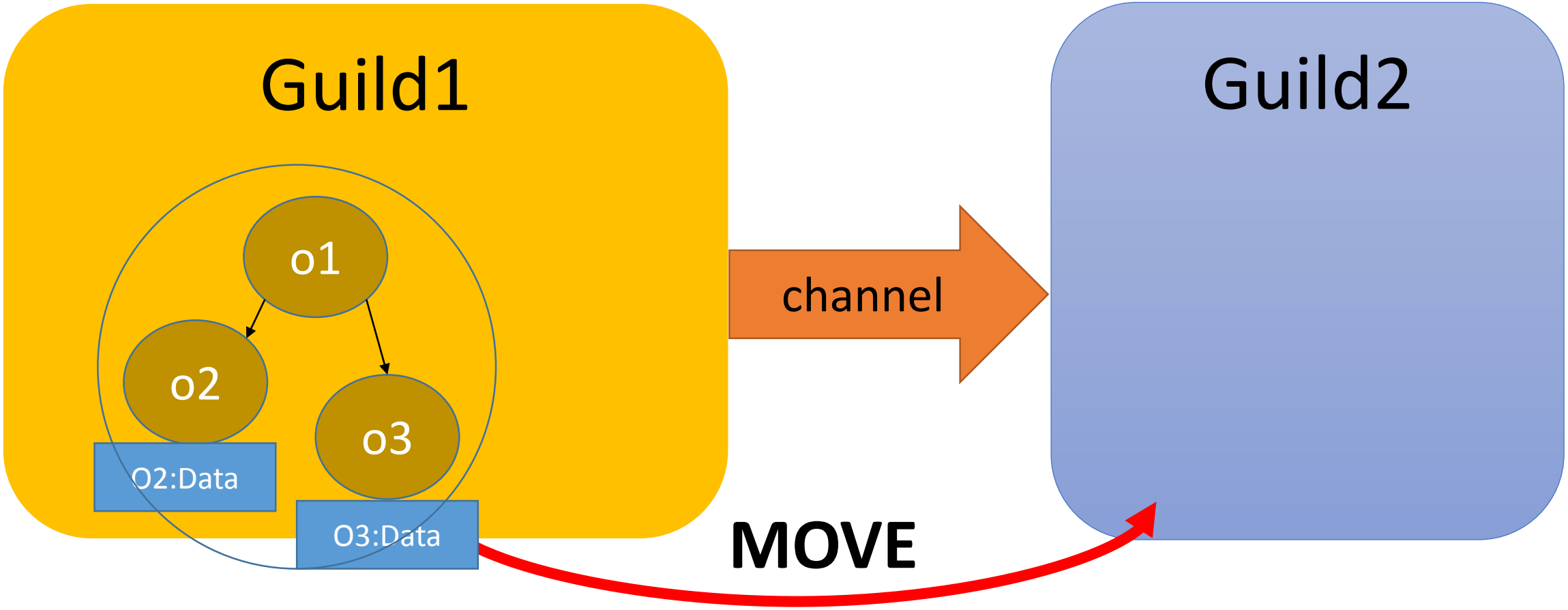
o1 = channel.receive



Move using Channel

channel.transfer_membership(o1)

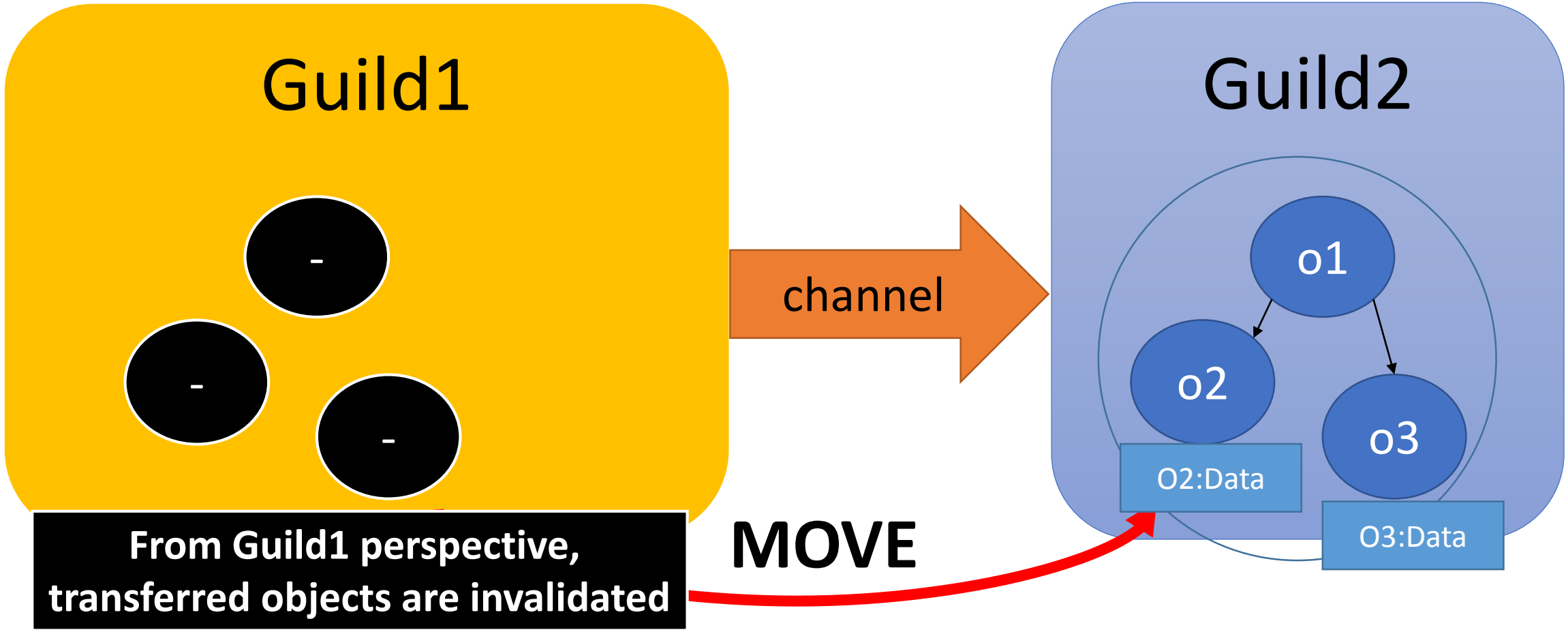
o1 = channel.receive



Move using Channel

channel.transfer_membership(o1)

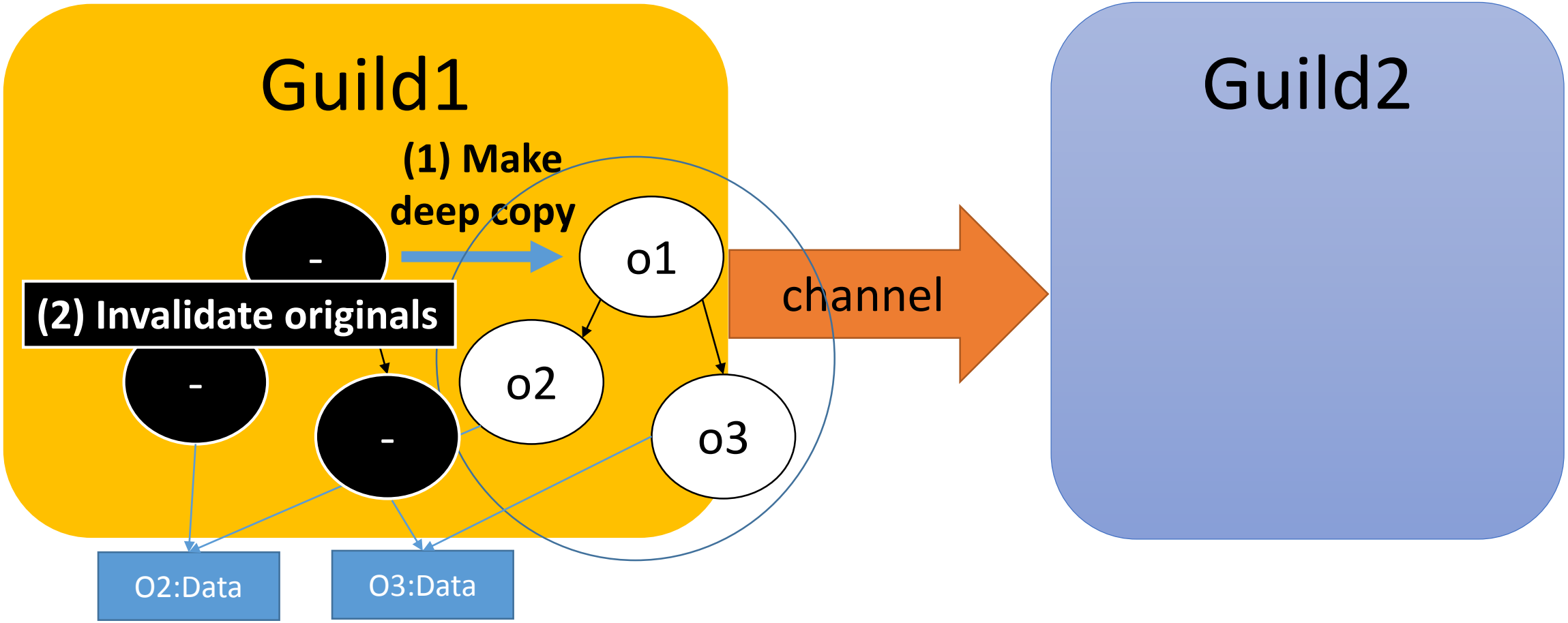
o1 = channel.receive



Move using Channel Implementation

channel.transfer_membership(o1)

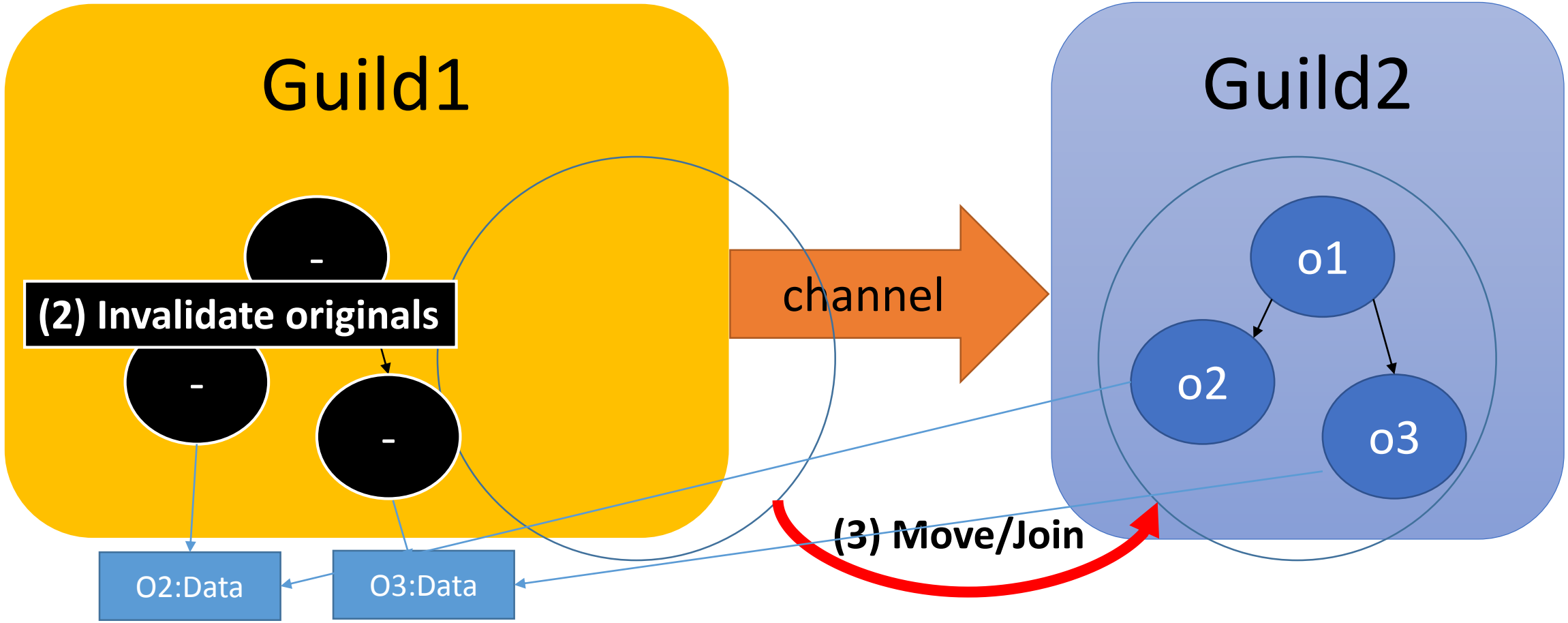
o1 = channel.receive



Move using Channel Implementation

channel.transfer_membership(o1)

o1 = channel.receive



Ruby global data

Keep compatibility with Ruby 2

- Global variables ($foo)
  - Change them to Guild local variables
- Class and module objects
  - Share between guilds
- Class variables
  - Change them to guild local. So that it is guild/class local variables
- Constants
  - Share between guilds
  - However if assigned object is not a deeply frozen object, this constant is accessed only by setting guilds. If other guilds try to access it, them cause error.
- Instance variables of class and module objects
  - Difficult. There are several approaches.
- Proc/Binding objects
  - Make it copy-able with env objects or env independent objects
- ObjectSpace.each_object
  - OMG

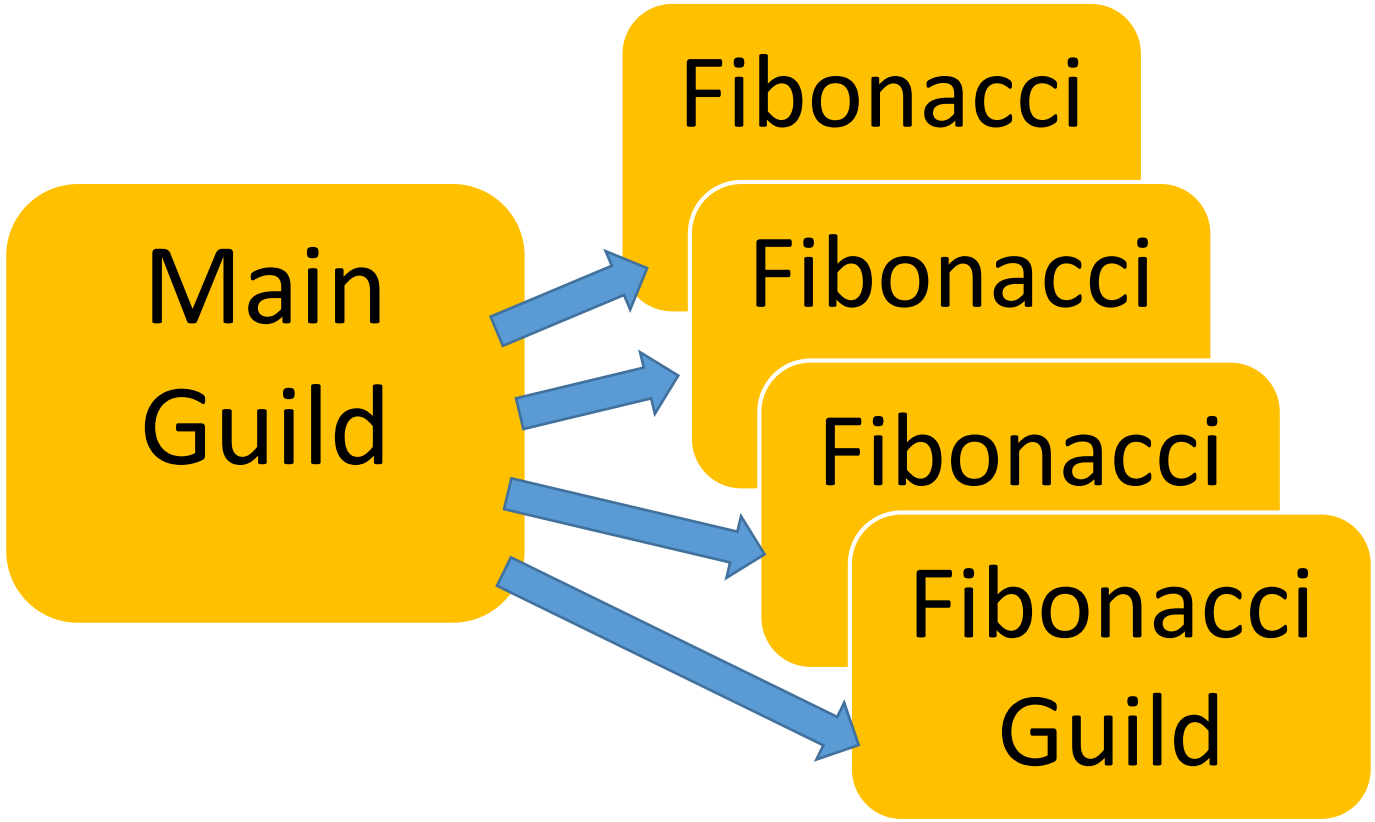
Interpreter process global data

- GC/Heap
  - Share it. Do stop the world parallel marking- and lazy concurrent sweeping.
  - Synchronize only at page acquire timing. No any synchronization at creation time.
- Inline method cache
  - To fill new entry, create an inline cache object and update atomically.
- Tables (such as method tables and constant tables)
  - Introduce mutual exclusions.
- Current working directory (cwd)
  - Each guild should have own cwd (using openat and so on).
- Signal
  - Design new signal delivery protocol and mechanism
- C level global variables
  - Avoid them.
  - Main guild can use C extensions depends on them
- Current thread
  - Use TLS (temporary), but we will change all of C APIs to receive context data as first parameter in the future.

Performance evaluation

- On 2 core virtual machine
  - Linux on VirtualBox on Windows 7
- Now, we can’t run Ruby program on other than main guild, so other guilds are implemented by C code

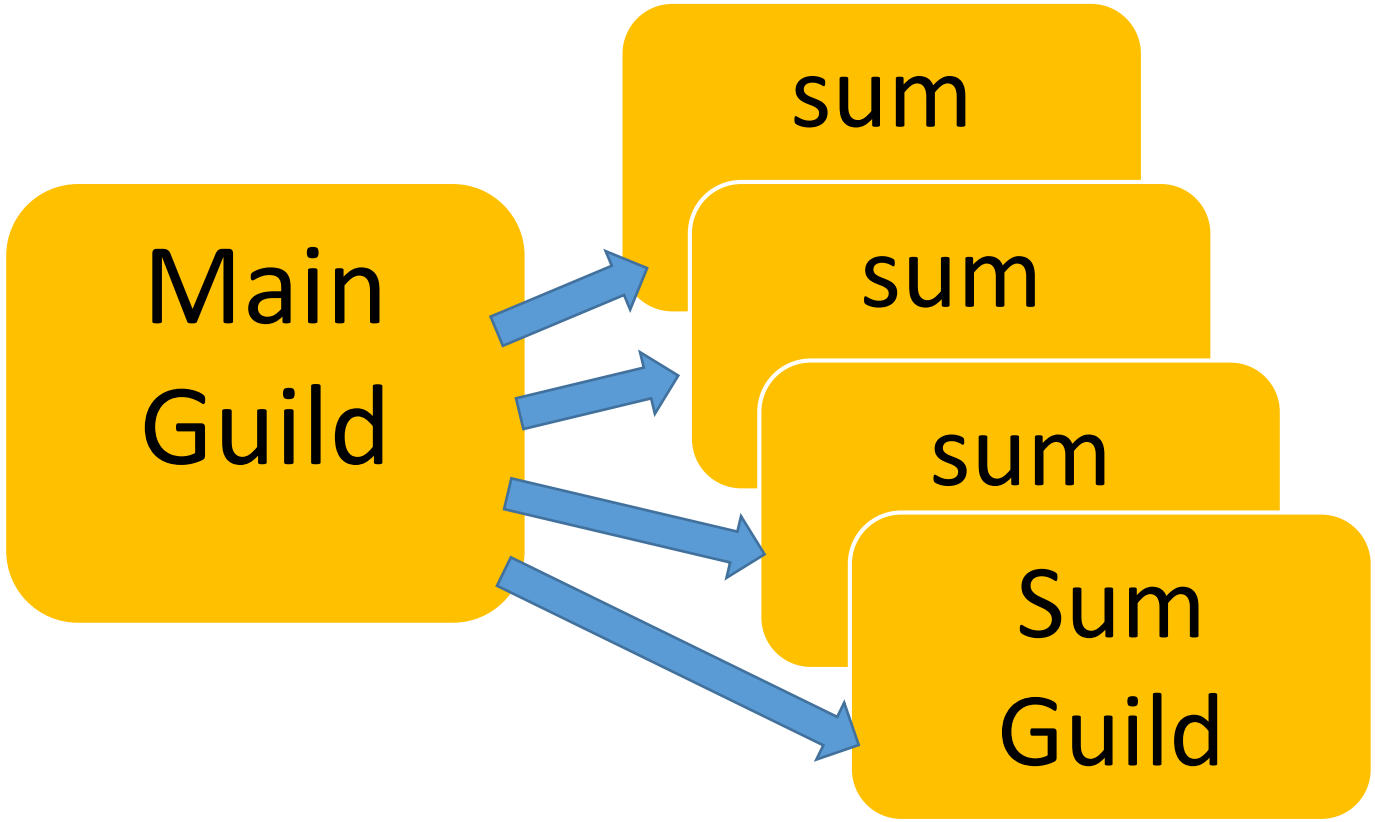
Performance evaluation
Simple numeric task in parallel



	Execution time (sec)
Single-Guild	19.45
Multi-Guild	10.45

Total 50 requests to compute fib(40)
Send 40 (integer) in each request

Performance evaluation Copy/Move

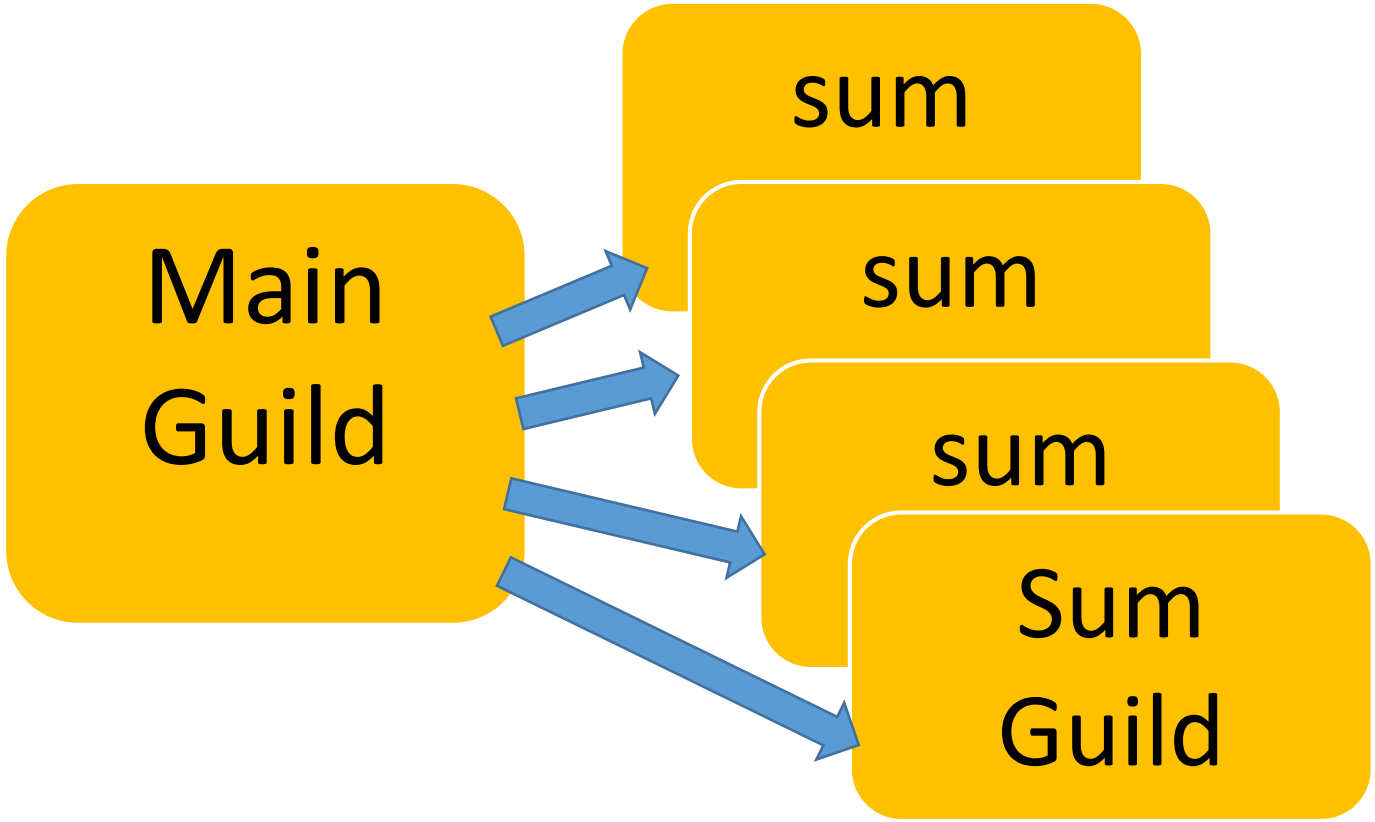


	Execution time (sec)
Single-Guild	1.00
Multi/ref	0.64
Multi/move	4.29
Multi/copy	5.16

Too slow!!
Because “move” need to check all of elements

Total 100 requests to compute sum of array
Send (1..10_000_000).to_a in each request

Performance evaluation Copy/Move



	Execution time (sec)
Single-Guild	1.00
Multi/ref	0.64
Multi/move	0.64

If we know this array only has deeply frozen objects,
we don’t need to check all elements => special data structure

Check our goal for Ruby 3

- **We need to keep compatibility** with Ruby 2.
  - Only in main guild, it is compatible.
- We **shouldn’t consider** about locks any more.
  - Only using copy and move, we don’t need to care locks
- We **can share** objects with copy, but **copy operation should be fast.**
  - Move (transfer membership) idea can reduce overhead
- We **should share objects** if we can.
  - We can share deeply frozen objects easily
- We can **provide special objects** to share mutable objects like Clojure if we really need speed.
  - Yes, we can provide.

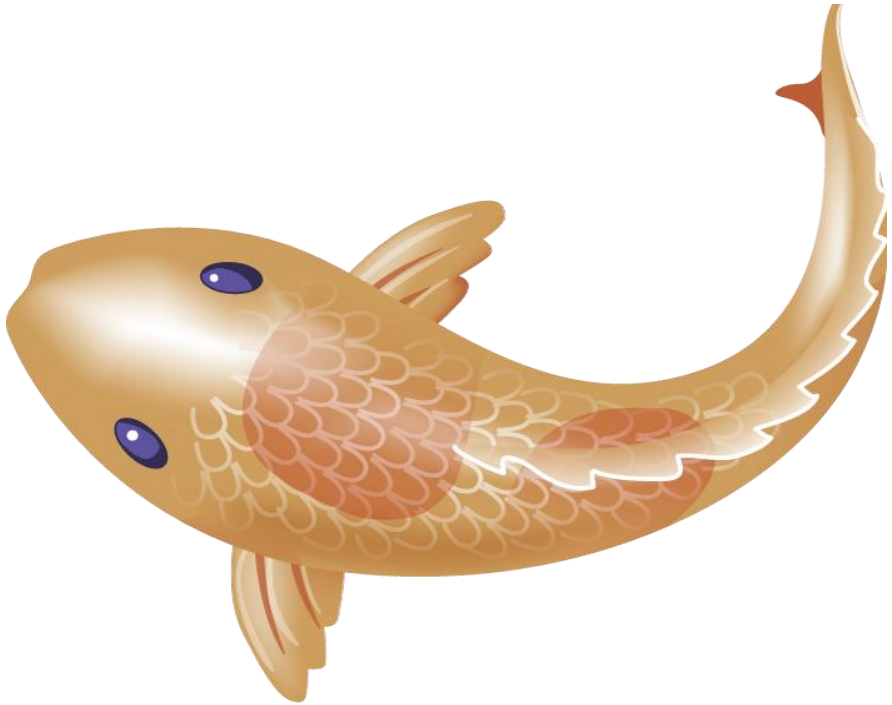
Summary

- Introduce “why threads are very difficult”
- Propose new concurrency abstraction “Guild” for Ruby 3
  - Not implemented yet, but I show key ideas

Thank you for your attention

Koichi Sasada
<ko1@heroku.com>

salesforce heroku



Approach comparison

	Process/MVM	Place (Racket)	Guild (copy/move)	Thread
Heap	Separate	Separate	Share	Share
Communication Mutable objects	Copy	Copy	Copy/Move	Share
Communication Frozen object	Copy	Share (maybe)	Share	Share
Lock	Don’t need	Don’t need	(mostly) Don’t need	Required
ISeq	Copy	Share	Share	Share
Class/Module	Copy	Copy (fork)	Share	Share