RubyConf 2016

Ruby 3 Concurrency

Koichi Sasada
ko1@heroku.com

RubyKaigi 2016

A proposal of new concurrency model for Ruby 3

Koichi Sasada
ko1@heroku.com

Motivation

Productivity

- Thread programming is very difficult
- Making correct concurrent programs easily

Performance by Parallel execution

- Making parallel programs
- Threads can make concurrent programs, but can’t run them in parallel
- People want to utilize Multi/many CPU cores

RubyKaigi2016 Proposal

Guild: new concurrency abstraction for Ruby 3

- Idea: **DO NOT SHARE** mutable objects between Guilds
- → No data races, no race conditions

Replace Threads to Guilds

RubyKaigi2016 Proposal

Guild: new concurrency abstraction for Ruby 3

- Idea: **DO NOT SHARE** mutable objects between Guilds
- → No data races, no race conditions

Kill Threads

Today's talk

Why is thread programming difficult?

Why does Guild solve this difficulty?

I'll try to shrink this talk (but has 70 pages).
Long version talk at RubyKaigi2016 is available:
http://rubykaigi.org/2016/presentations/ko1.html

NOTE

“Guild” is proposal for Ruby 3.
Specifications
and name of “Guild”
can be changed.

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

Koichi is an Employee

heroku

Visit Heroku booth and discuss more!

Difficulty of 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

- Traditional “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 and race condition

```
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
```

Can you find all bugs?

Data race and race condition

```
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

Data race and race condition

```
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
```

Race Condition

Data race and race condition

- Solution: Lock (synchronize) all over the 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
```

Difficulty of multi-threads programs

- We need to synchronize all sharing mutable objects correctly
  - Easy to share objects, but difficult to recognize
  - We can track on a small program
  - Difficult to track on a **big programs**, especially on **programs using gems**
- We need to check **all of source codes**, or believe **library documents** (but documents should be correct)

Overcome thread difficulty

Key idea

Problem of multi-thread programming:

Easy to share mutable objects

Idea:

Do not allow to share mutable objects without any restriction

Study from other languages

- Do not share mutable objects
  - Copy to send message (shell, druby, …)
    - ☹ Copy everything is slow
  - Prohibit mutable objects (functional lang, Erlang, Elxir)
    - ☹ We can’t accept such big incompatibility
  - Share only immutable objects (Place (Racket))
    - ☹ We want to share other kind of objects
- Allow sharing with restriction
  - Allow mutation only with special protocol (Clojure)
    - ☹ we can’t accept special protocol

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

Our goal for Ruby 3

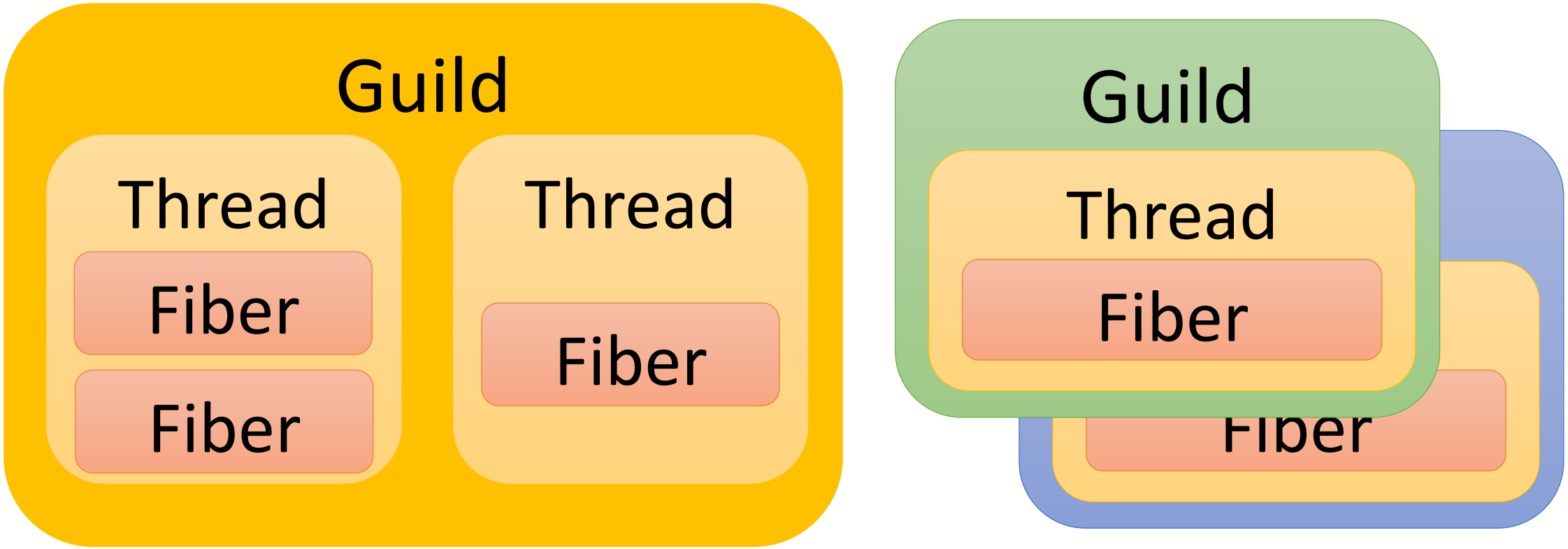
- We need to **keep compatibility** with Ruby 2.
- We can make **parallel program**.
- We **shouldn't consider** about locks any more.
- We **can share** objects with **copy**, but **copy operation should be fast.**
- We **should share immutable objects** if we can.
- We can **provide special objects**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

“Guild”

New concurrency model for Ruby 3

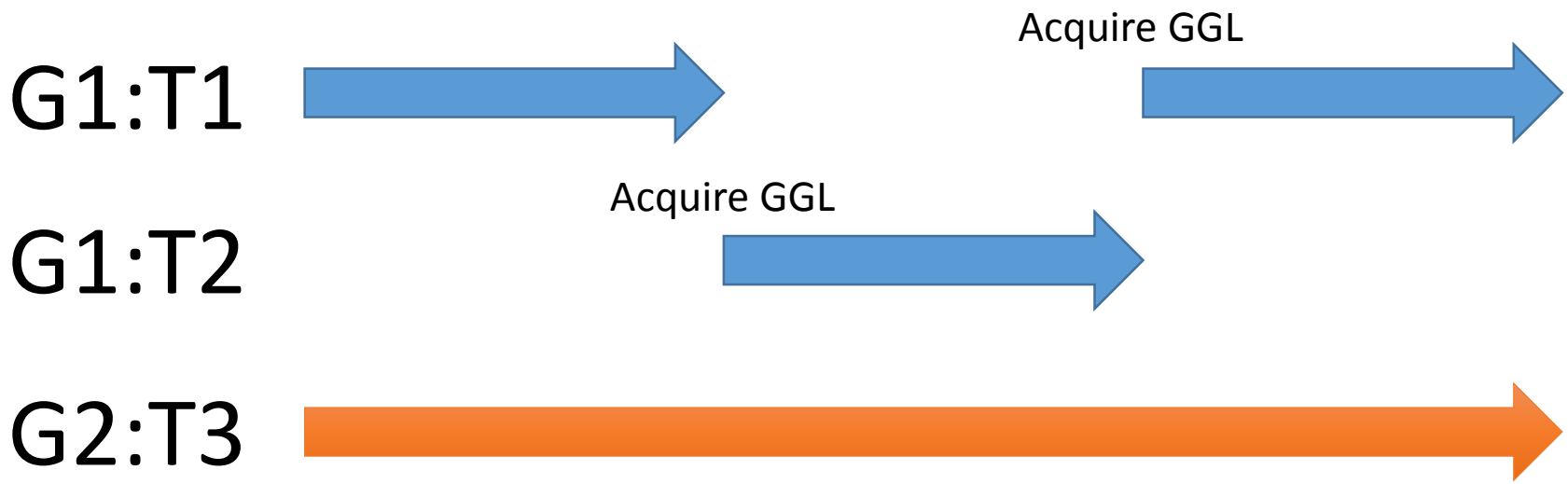
Guild: New concurrency abstraction

- Guild has at least one thread (and a thread has at least one fiber)



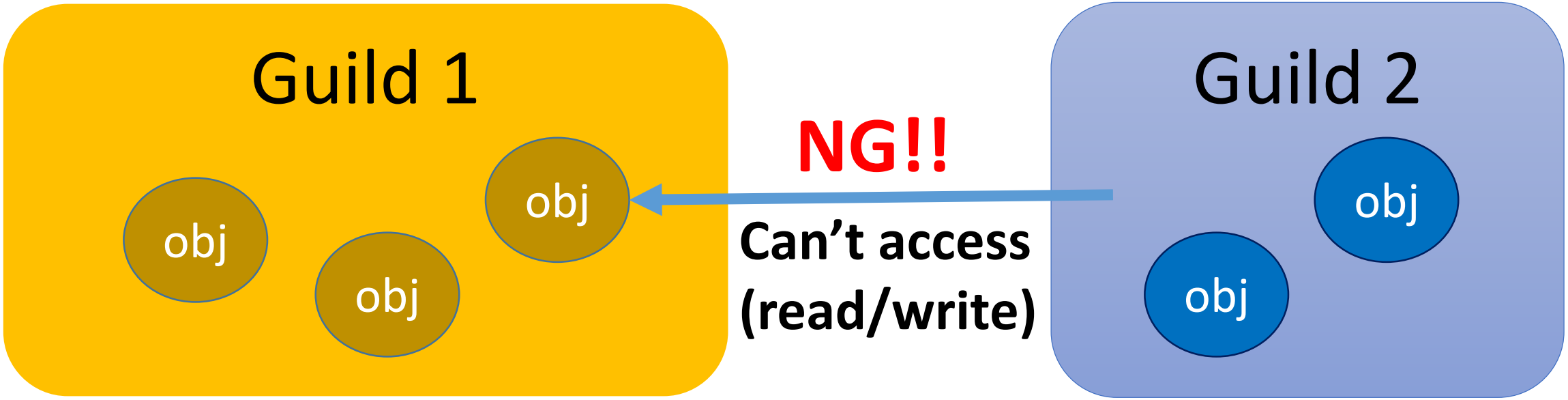
Threads in different guilds can run in Parallel

- Threads in different guilds **can run in parallel**
- Threads in a same guild **can not run in parallel** because of GVL (or GGL: Giant Guild Lock)



Important rule:
Mutable Objects have a membership

- All of mutable objects should belong to **only one Guild** exclusively
- Because Guild is not **“Community”**



Object membership

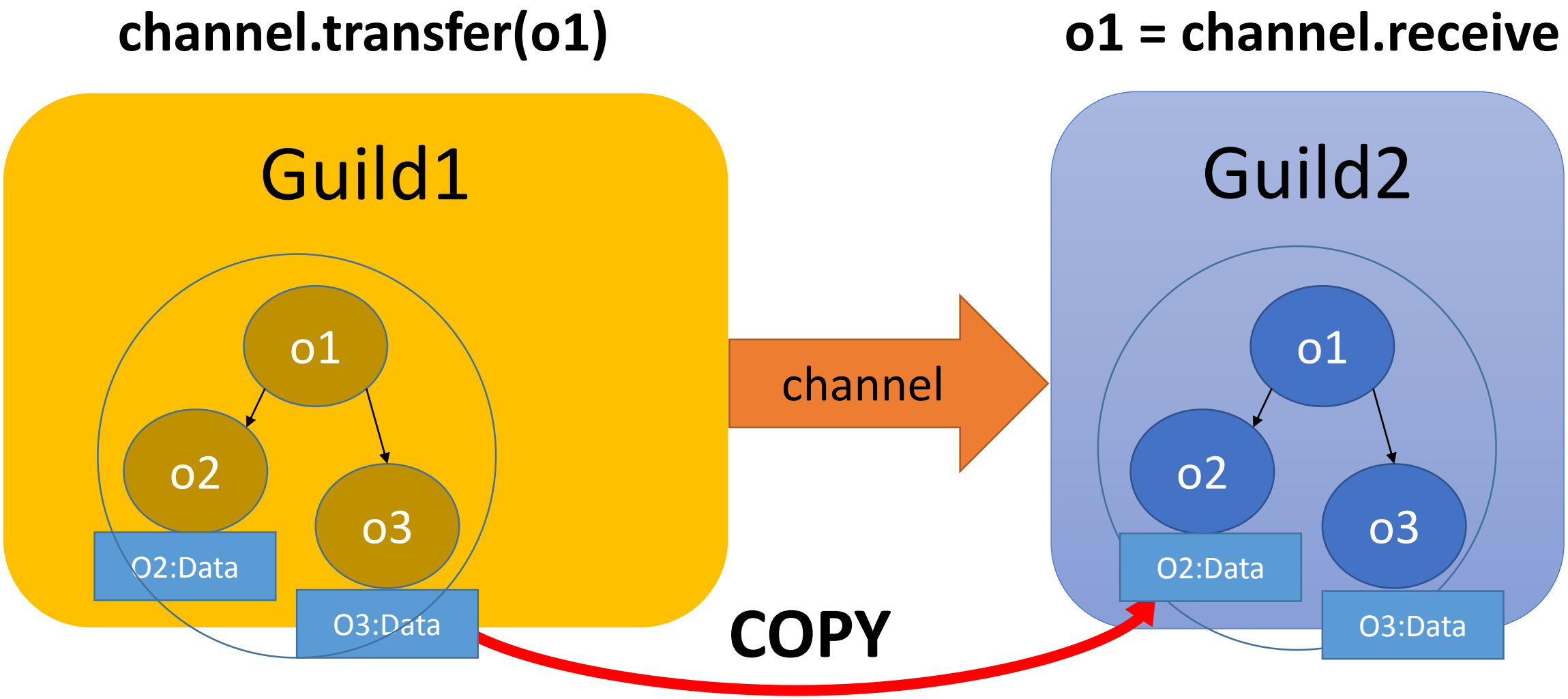
Only one guild can access mutable object

→ **We don’t need to consider about locks**
(if Guild has only one thread)

Inter-guild communication

- **“Guild::Channel”** to communicate each guilds
- Two communication methods
  1. **Copy**
  2. **Move (transfer_membership)**

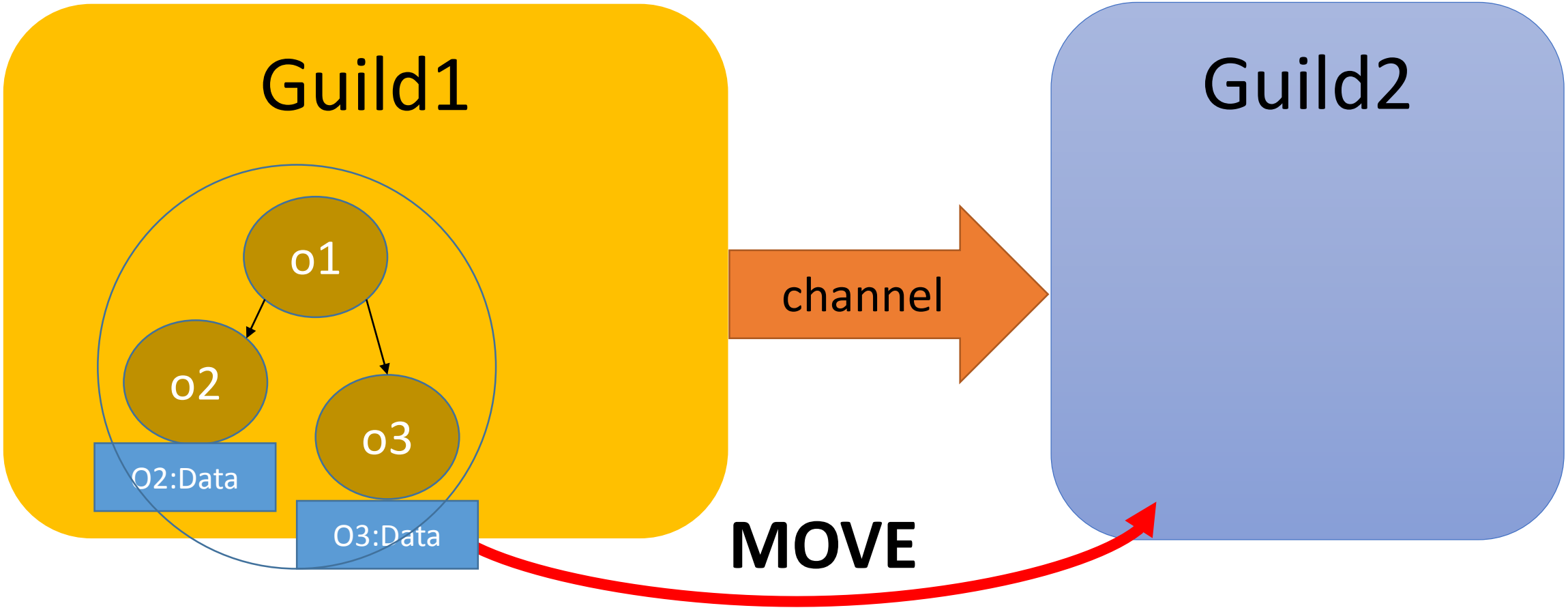
Copy using Channel



Move using Channel

channel.transfer_membership(o1)

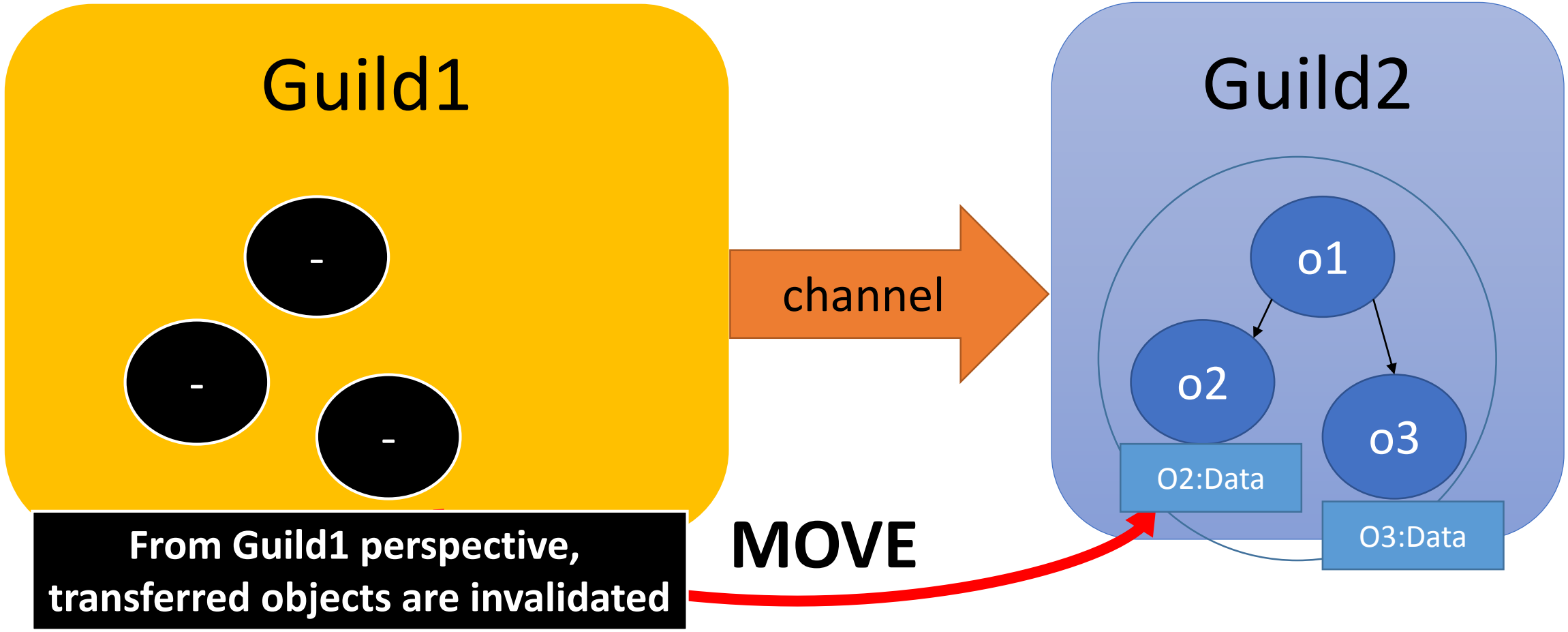
o1 = channel.receive



Move using Channel

channel.transfer_membership(o1)

o1 = channel.receive



Move using Channel

- Prohibit accessing to left objects
  - Cause exceptions and so on
  - ex)
```
obj = “foo”
ch.move (obj)
obj.upcase      #=> Error!!
p(obj)          #=> Error!!
```

Use cases for copy and move

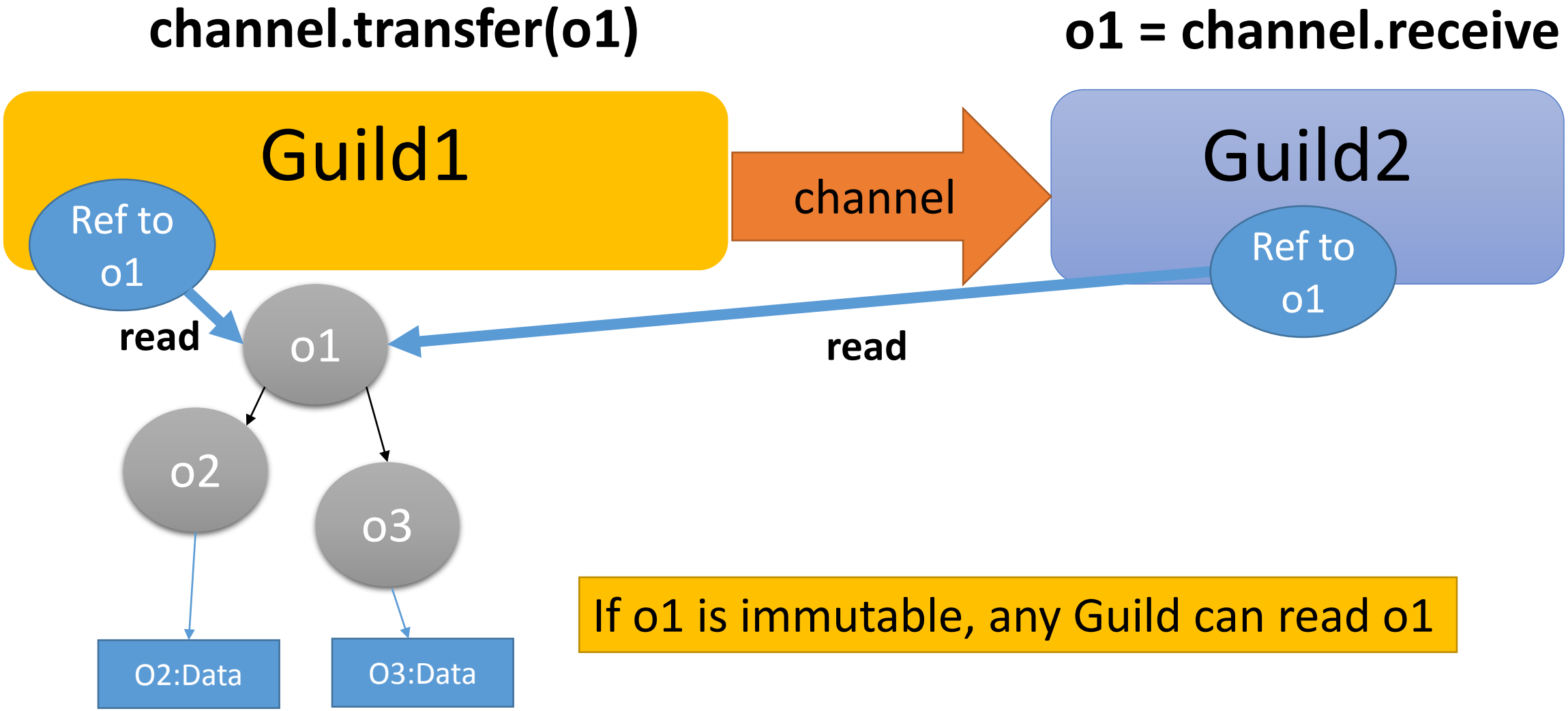
- You can copy small objects (dRuby does)
  - Parameter array (**[:do_foo, 1, 2, 3]**, like Erlang)
- You can move small amount number of objects
  - Move a long string and modify them in parallel

Sharing immutable objects

- **Immutable objects** can be shared with any guilds
  - a1 = [1, 2, 3].freeze: a1 is **Immutable object**
  - a2 = [1, Object.new, 3].freeze: a2 is **not immutable**
- We only need to send references
  - Very lightweight, like thread-programming
- **Numeric objects, symbols, true, false, nil** are immutable (from Ruby 2.0, 2.1, 2.2)

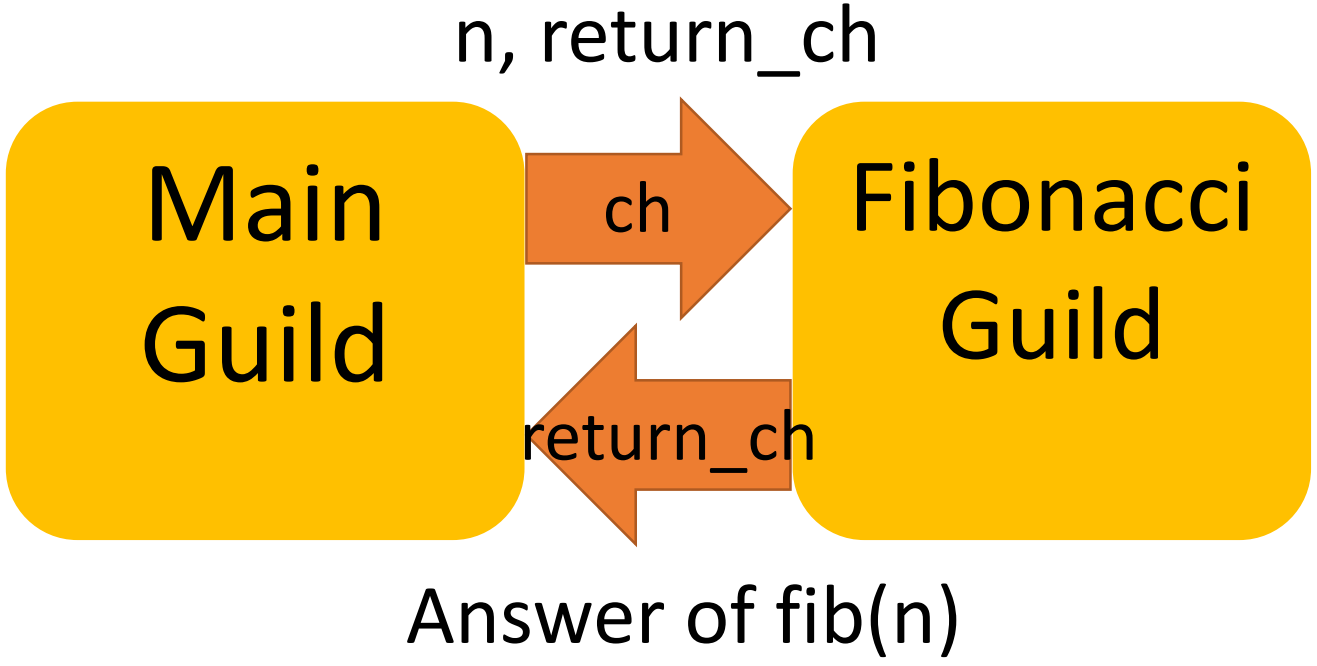
Sharing immutable objects

We can share reference to immutable objects



Use-case 1: master – worker type

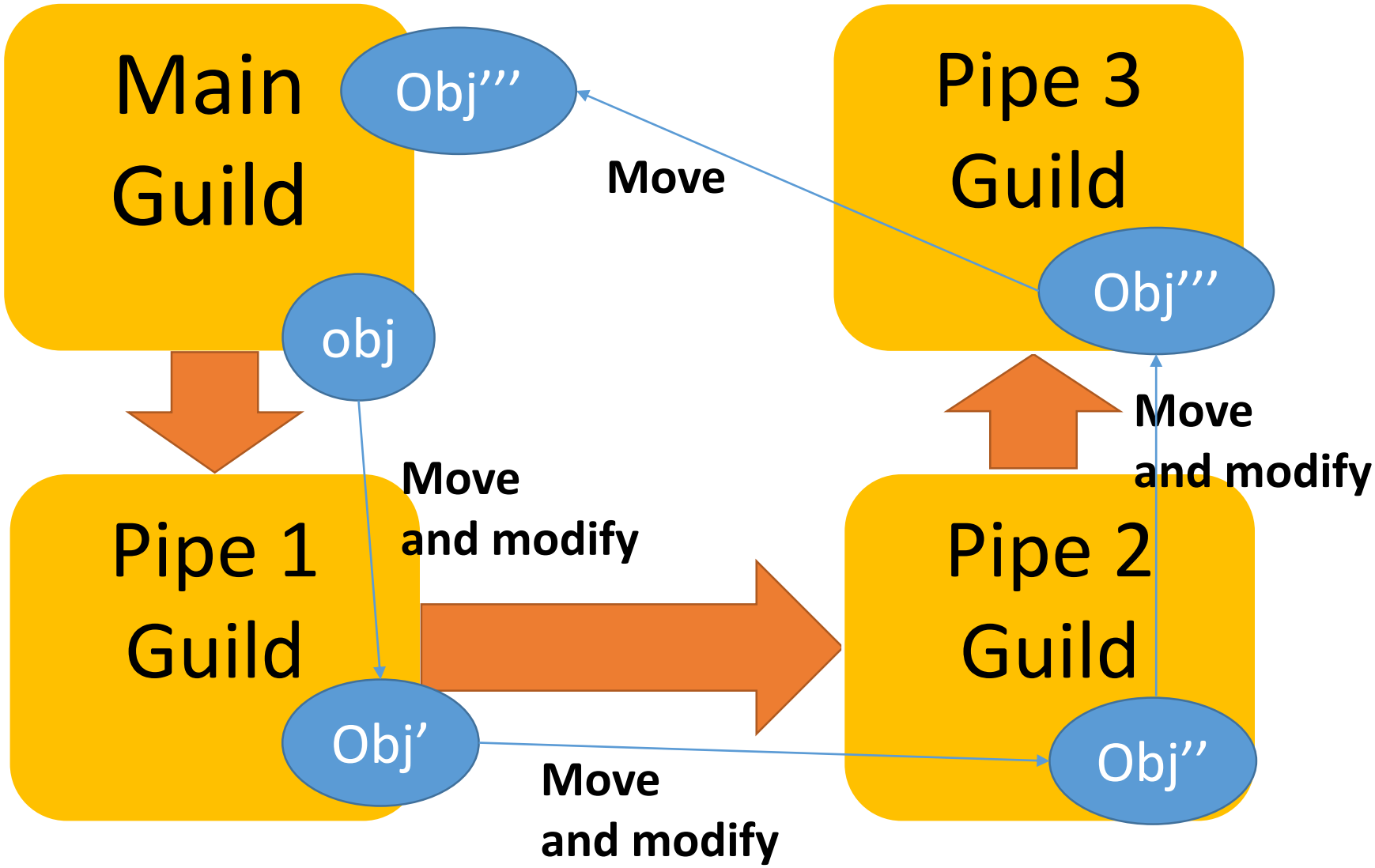
```
def fib(n) ... end
g_fib = Guild.new(script: %q{
  ch = Guild.default_channel
  while n, return_ch = ch.receive
    return_ch.transfer fib(n)
  end
})

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

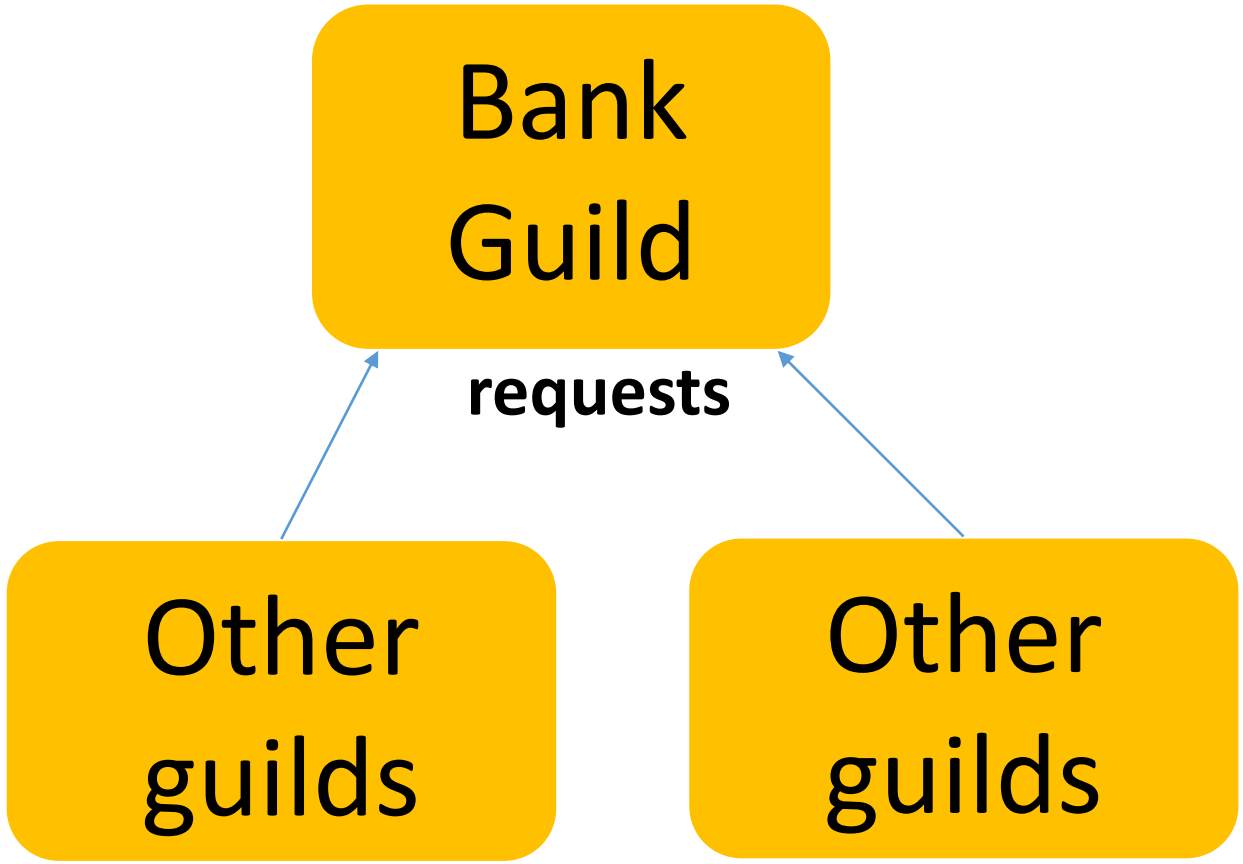
```
result_ch = Guild::Channel.new
g_pipe3 = Guild.new(script: %q{
  while obj = Guild.default_channel.receive
    obj = modify_obj3(obj)
    Guild.argv[0].transfer_membership(obj)
  end
}, argv: [result_ch])
g_pipe2 = Guild.new(script: %q{
  while obj = Guild.default_channel.receive
    obj = modify_obj2(obj)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

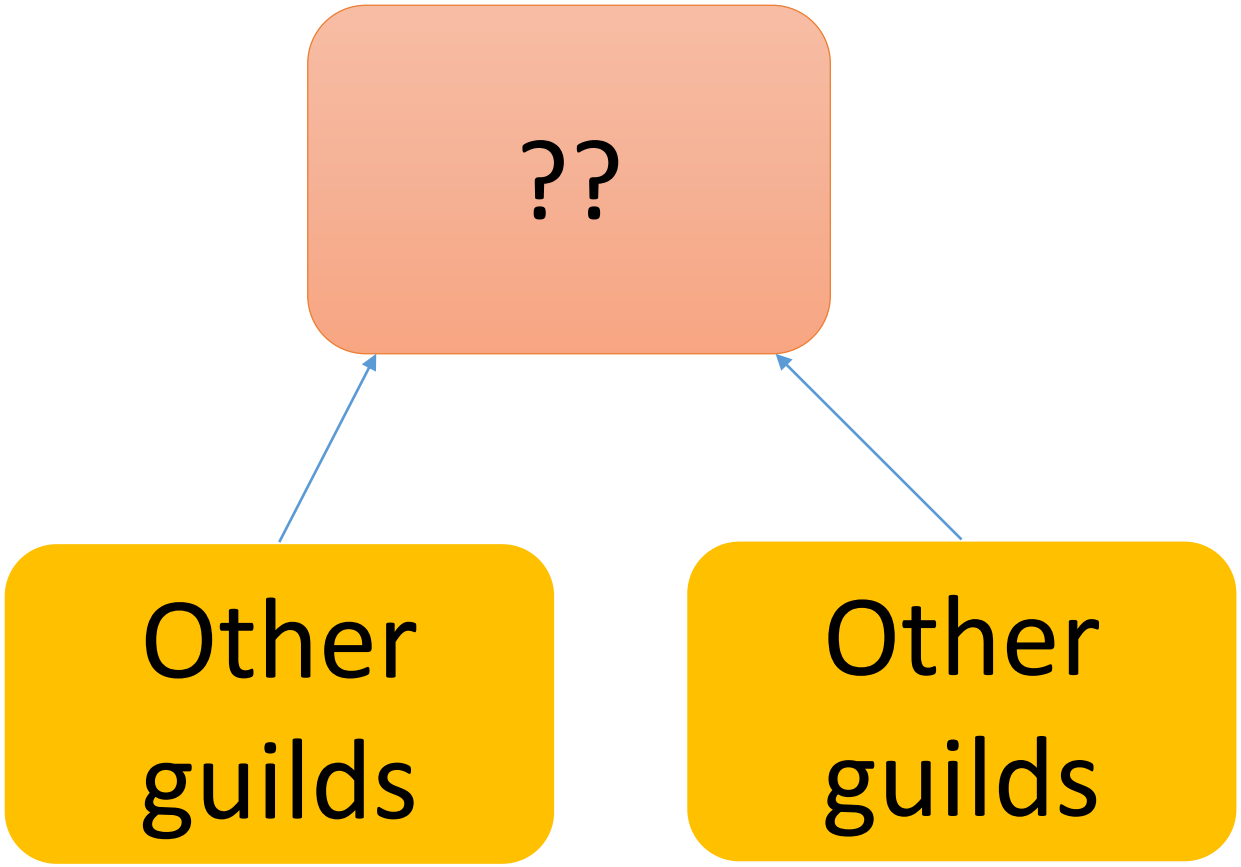
```
g_bank = Guild.new(script: %q{
  while account_from, account_to, amount,
            ch = Guild.default_channel.receive
    if (Bank[account_from].balance < amount)
      ch.transfer :NOPE
    else
      Bank[account_to].balance += amount
      Bank[account_from].balance -= amount
      ch.transfer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



Compare between threads and guilds

- Threads:
  - ☺ Inter threads communication is very fast
  - ☺ We already know thread-programming
  - ☹ Difficult to make correct thread-safe programs
- Guilds:
  - ☹ Inter guilds communication introduces overhead
    - ☺ “Move” technique can reduce this kind of overheads
  - ☹ We need to learn this model
  - ☹ We need to make parallel programs from scratch
  - ☺ We don’t need to care about synchronizations any more

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

Which do you want to choose?

Discussion: The name of “Guild”

- “Guild” is good metaphor for “object’s membership”
- Check duplication
  - Nobody using as programming terminology (maybe)
  - There are no duplicating top-level classes and modules in all of rubygems
  - First letter is not same as other similar abstractions
    - For variable names
    - P is for Processes, T is for Threads, F is for Fibers

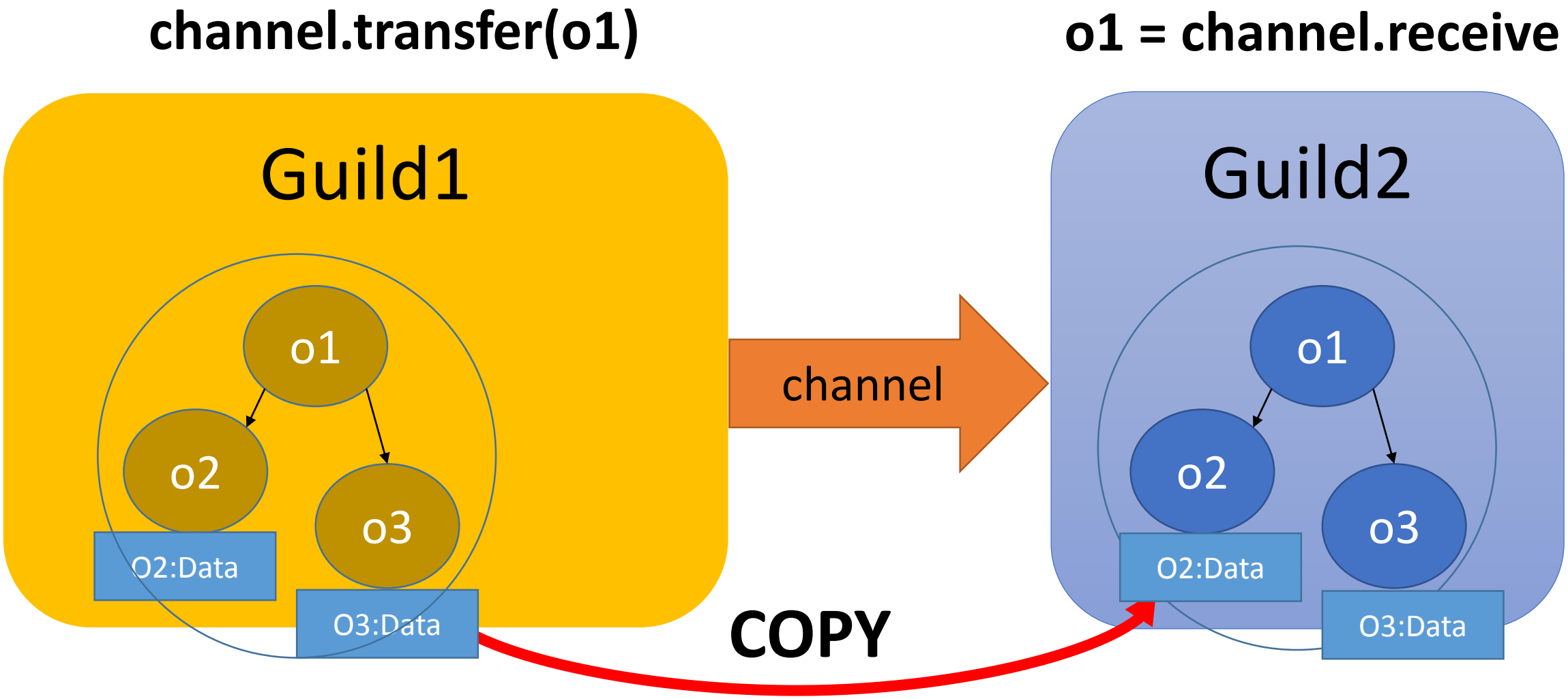
Implementation of “Guild”

- How to achieve **“object membership”**
- How to implement **“Inter Guilds communication”**
- How to design **“shared mutable data”**
- How to isolate **“process global data”**

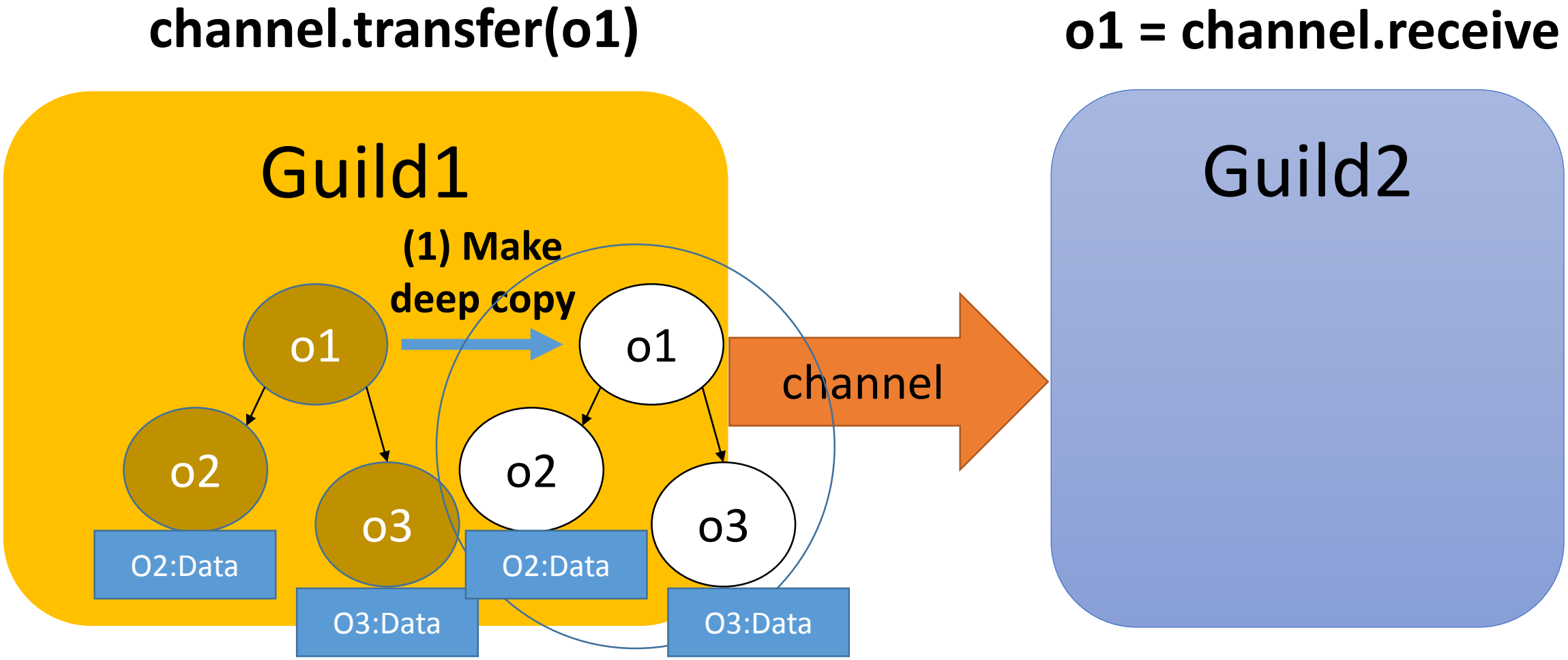
How to implement inter Guilds communication

- Copy
- Move (transfer membership)

Copy using Channel



Copy using Channel Implementation

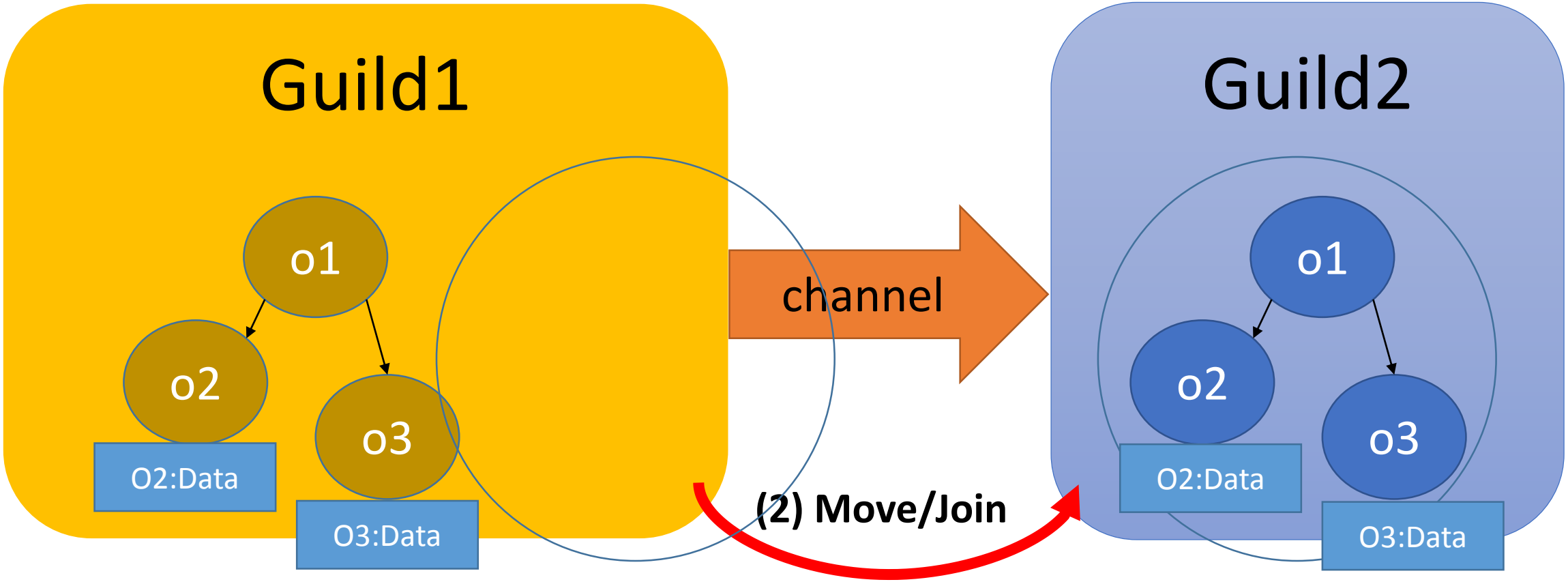


Copy using Channel Implementation

We can use CoW technique for data

channel.transfer(o1)

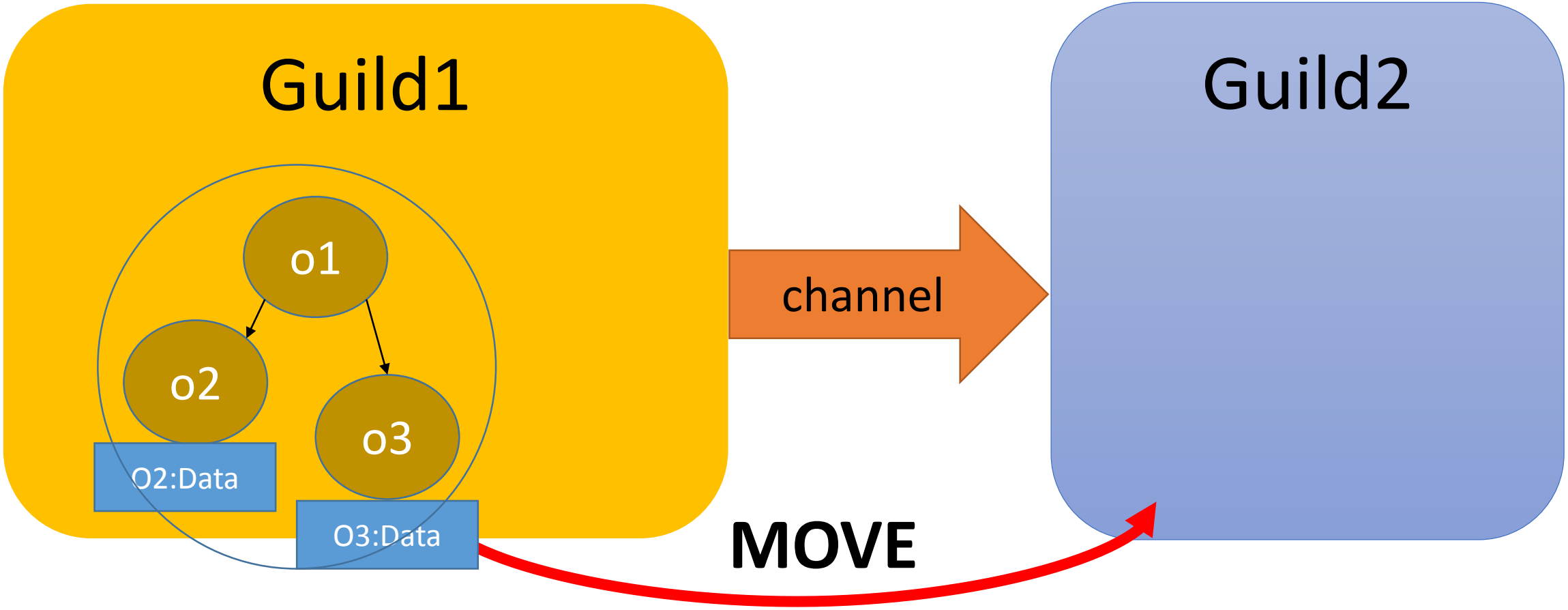
o1 = channel.receive



Move using Channel

channel.transfer_membership(o1)

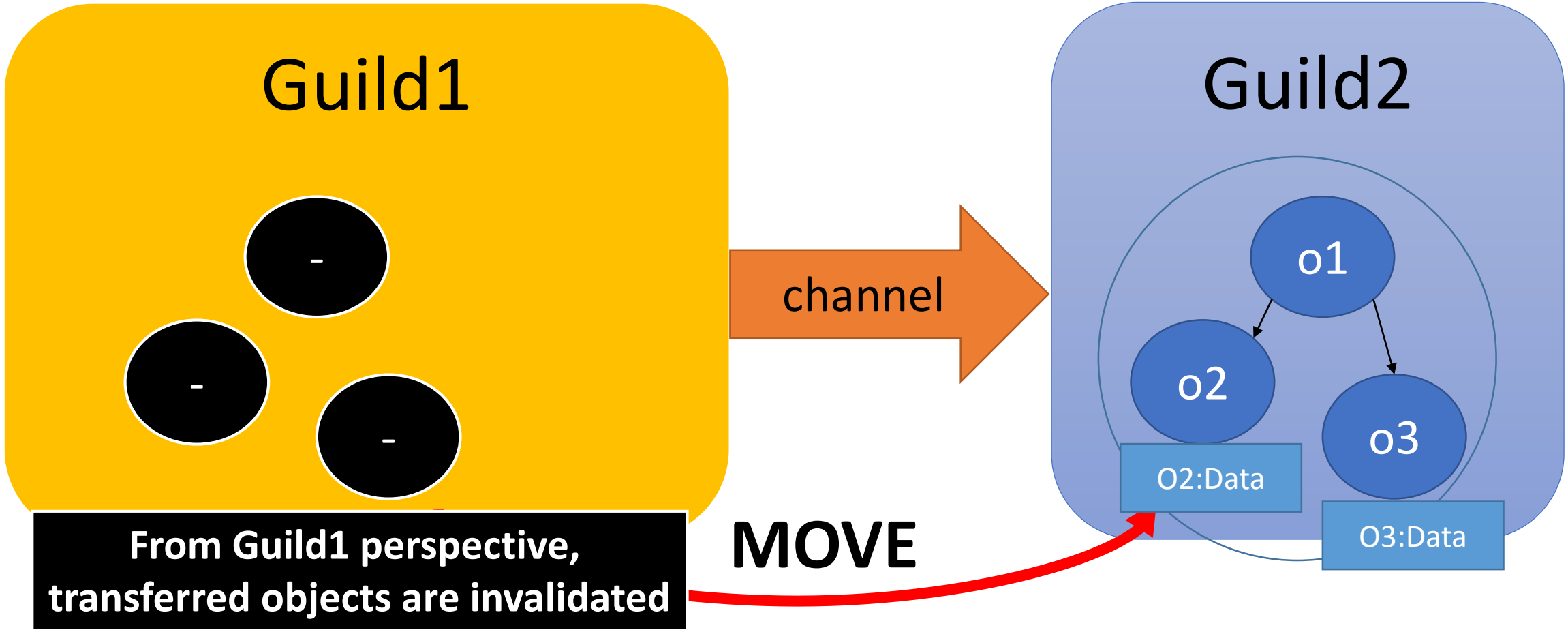
o1 = channel.receive



Move using Channel

channel.transfer_membership(o1)

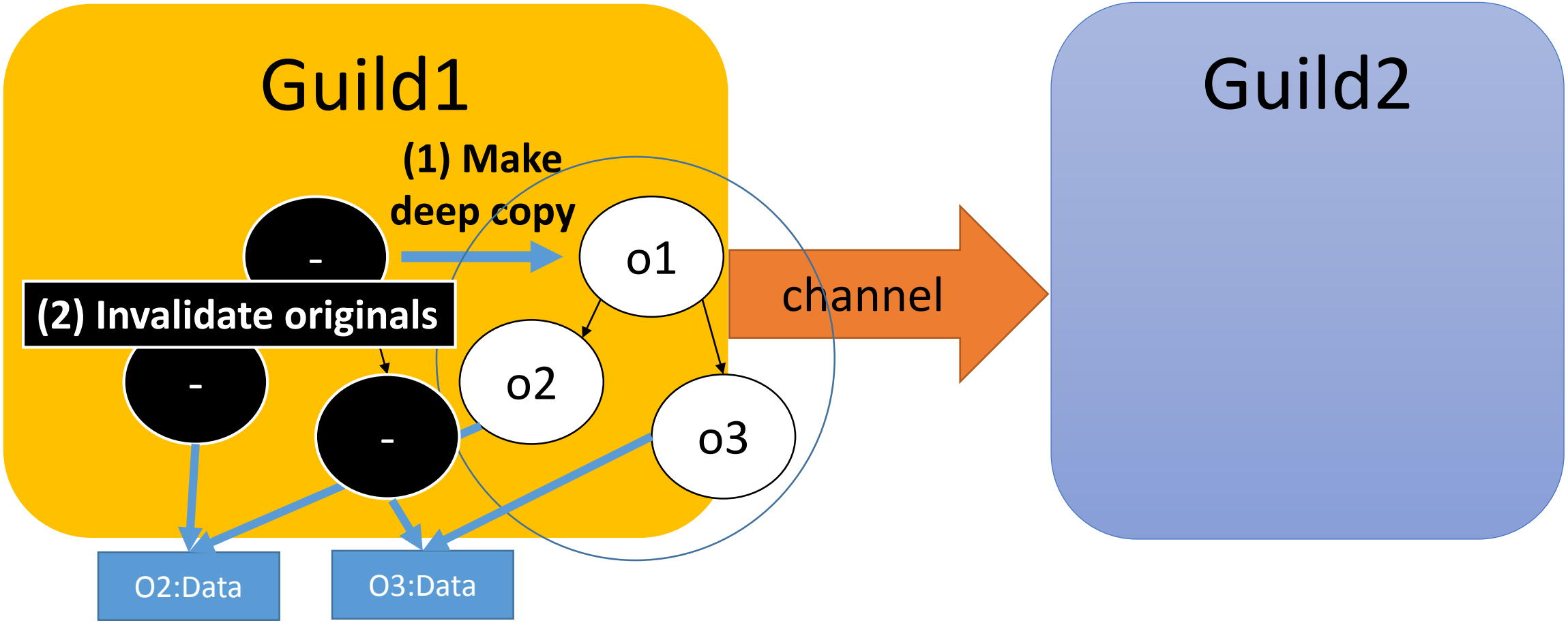
o1 = channel.receive



Move using Channel Implementation

channel.transfer_membership(o1)

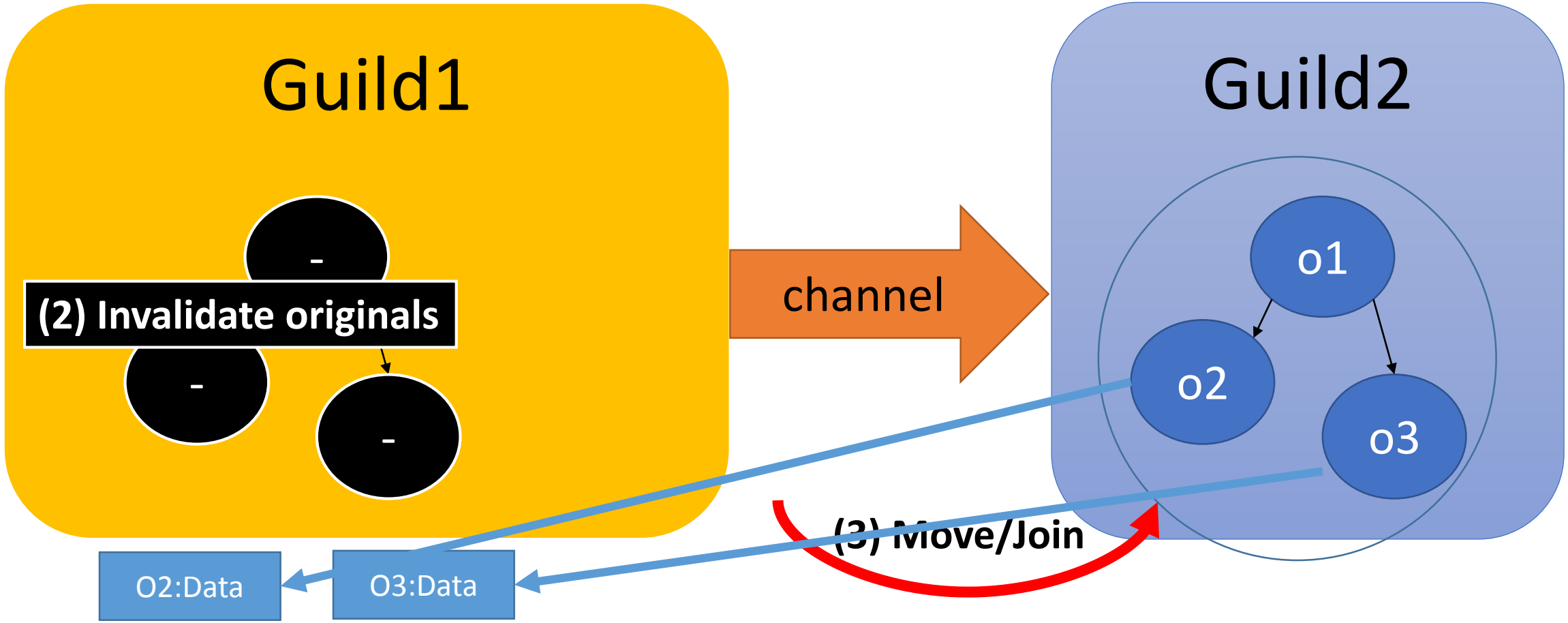
o1 = channel.receive



Move using Channel Implementation

channel.transfer_membership(o1)

o1 = channel.receive



Move using Channel Implementation

- "Move" is not a reference passing,
  but a **copy object headers**
    → Objects don't need to know own guild
    → Interpreter doesn't need to check guilds
- Mutable objects live in same guild their entire life

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 immutable 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

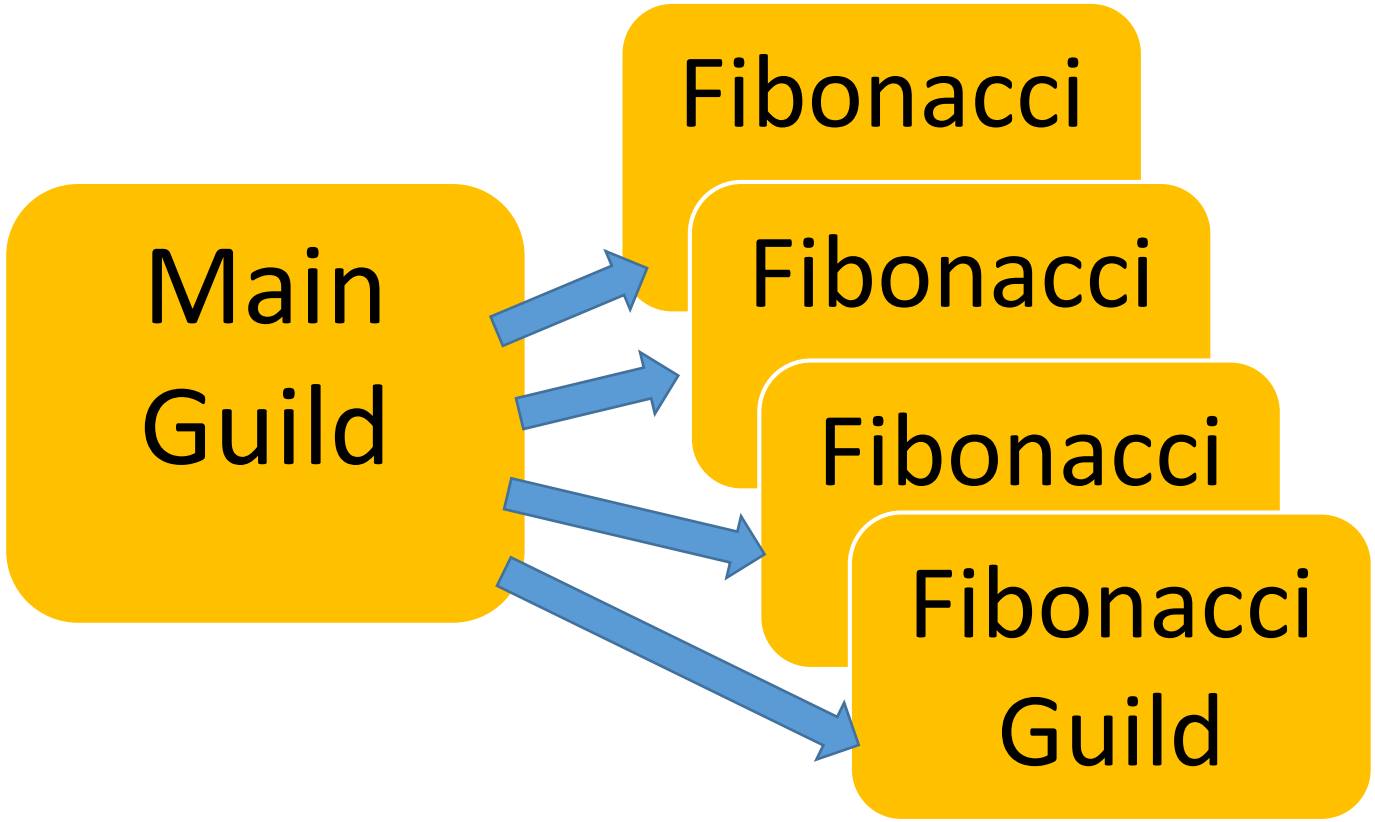
Interpreter process global data

- 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

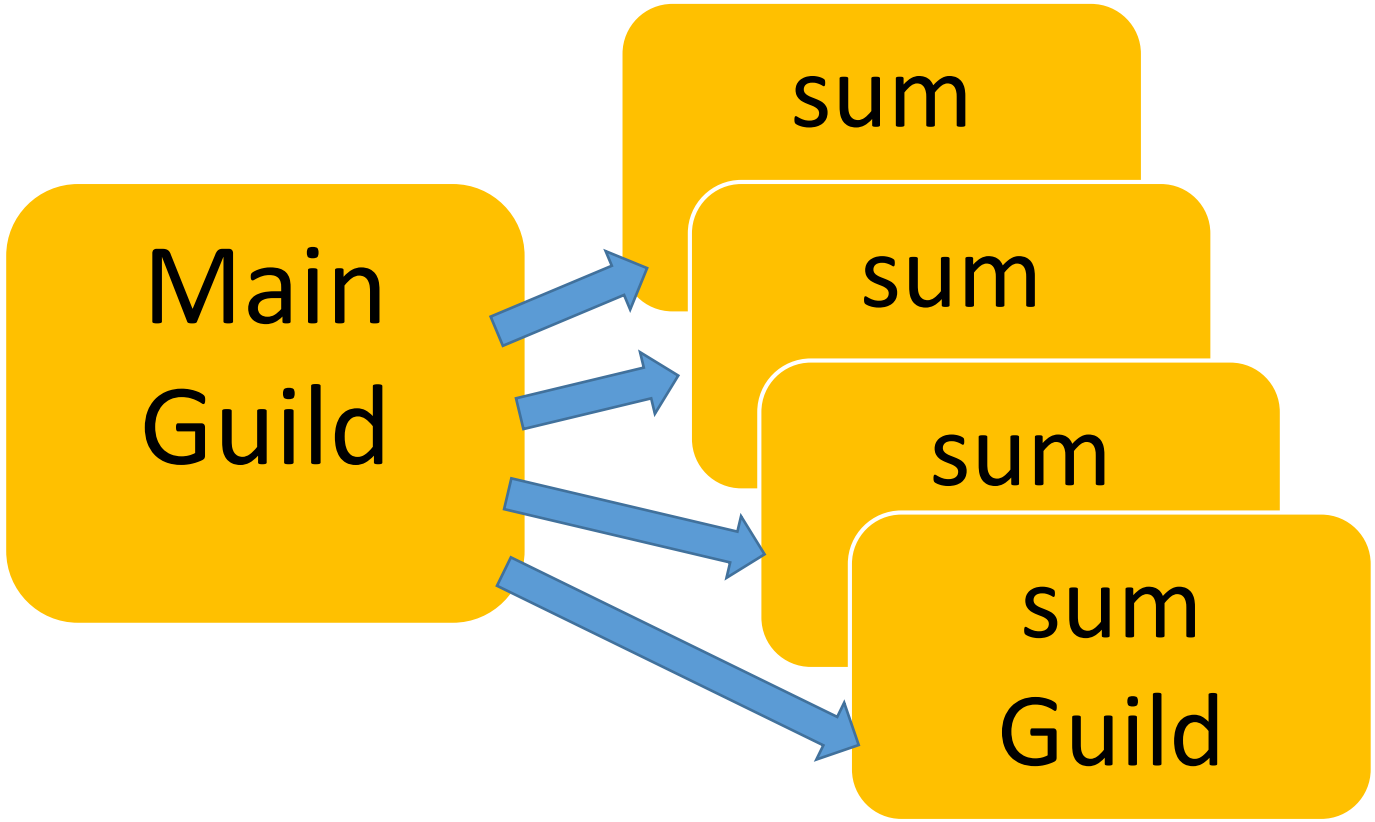
Performance evaluation
Simple numeric task in parallel



	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

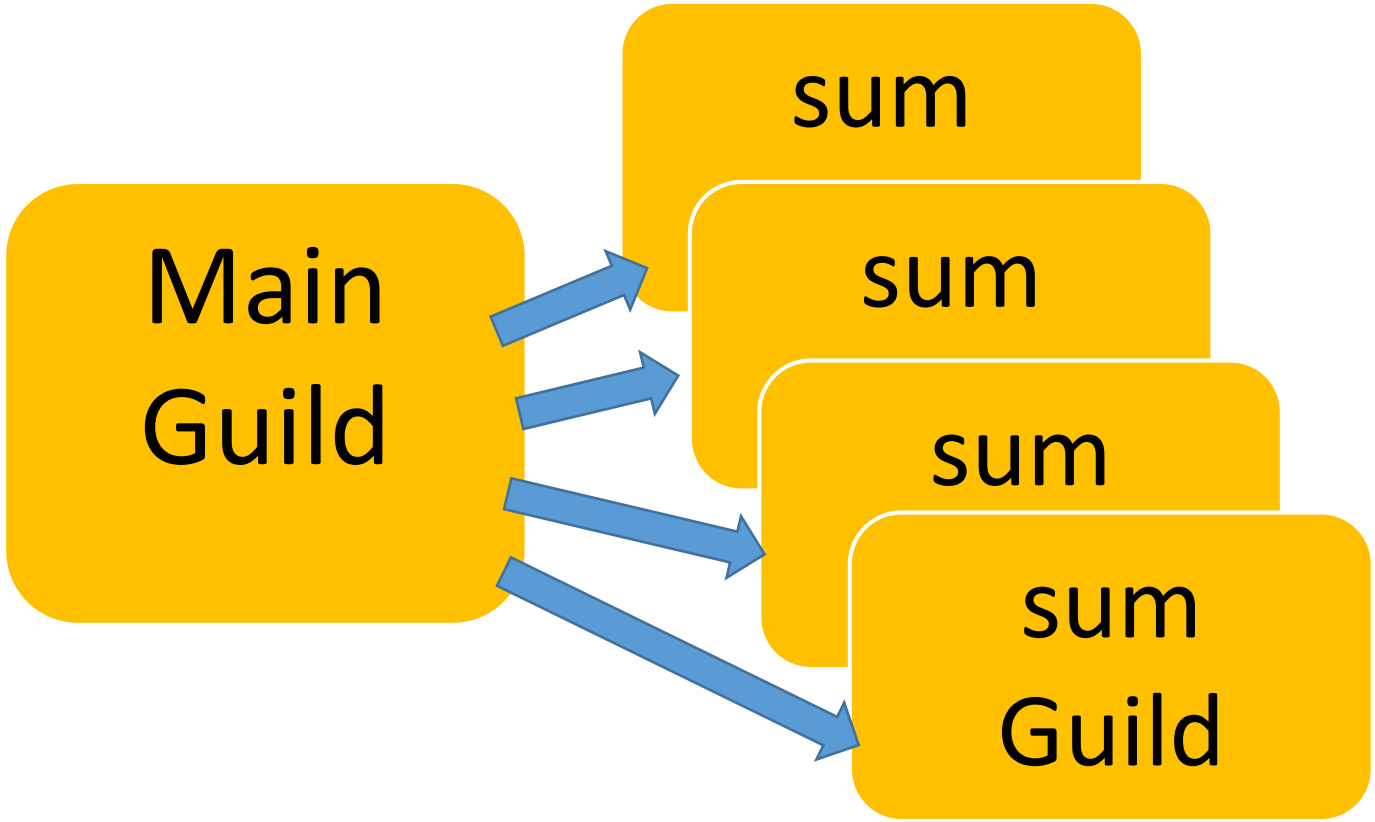


	Execution time (sec)
Single-Guild	1.00
Multi/ref	0.64
Multi/move	4.29
Multi/copy	5.16

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

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

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 immutable 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.
  - **OK:** Only in main guild, it is compatible.
- We can make **parallel program**.
  - **OK:** Guilds can run in parallel.
- We **shouldn't consider** about locks any more.
  - **OK:** Only using copy and move, we don't need to care locks.
- We **can share** objects with copy, but **copy operation should be fast.**
  - **OK:** Move (transfer membership) idea can reduce overhead.
- We **should share objects** if we can.
  - **OK:** We can share immutable objects fast and easily.
- We can **provide special objects** to share mutable objects like Clojure if we really need speed.
  - **OK:** Yes, we can provide.

Satisfied!

FAQ

- Q: Can we try Guild now?
- A: No.
  - Implementation on MRI is big project. Not yet.
    - Supporting this project is welcome.
  - Some guys are trying to implement it on JRuby.

FAQ

- Q: Should we wait Guild for Ruby 3?
- A: Not sure.
  - 2.6? 2.7? 2.8?
  - I want to implement it next year.

FAQ

- Q: Can Guild replace **ALL** of Thread programs?
- A: No.
  - To utilize Guild, you need to rewrite your programs.
  - I assume 90% of programs are easy to replace.
  - For example, “moving” IO object is easy to understand, so that web application server is easy to implement.

FAQ

- Q: Membership seems “**ownership”**. Right?
- A: Yes.
  - Actually, we call this idea “ownership” before.
  - We named **“membership”** because “Guild” is not owner of members.

FAQ

- Q: “Moving” cause huge overhead for big object graph (like big Hash object). Right?
- A: Yes.
  - We need to move all of objects (e.g. Hash entries).
  - We need to introduce special data structures for such big object graph (like Clojure).
  - I believe people can change their mind to fit this model.

FAQ

- Q: Can we share Proc object?
- A: No.
  - Good question. I’m thinking several options:
    - Allow to copy local environment (variables)
    - Allow to move local environment (variables)
    - Introduce isolated Proc

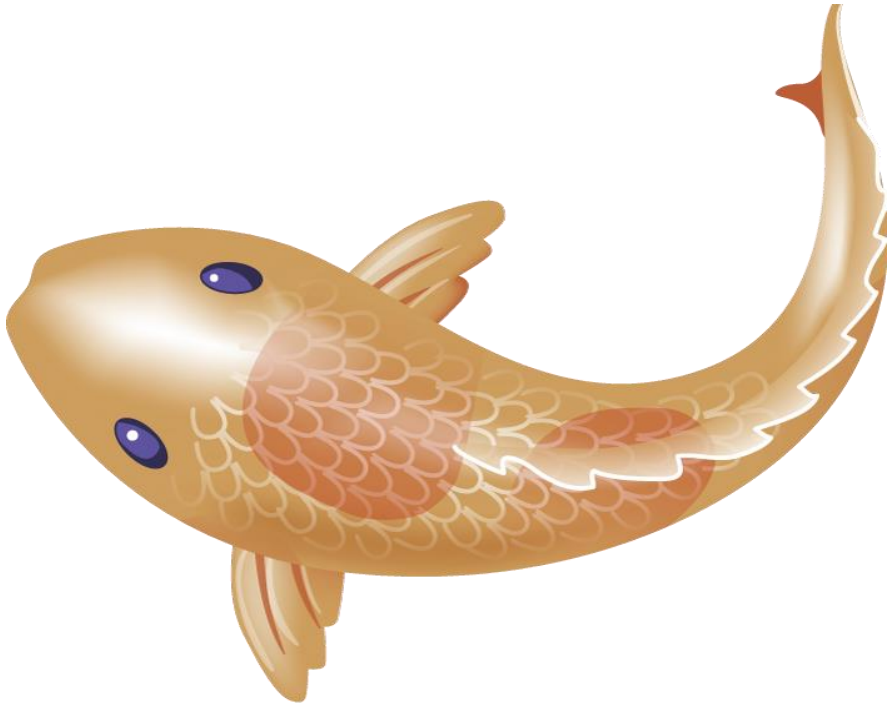
Summary

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

Thank you for your attention

Koichi Sasada
<ko1@heroku.com>

salesforce heroku



Approach comparison

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

Related work

- ***"Membership transfer"*** is proposed by [Nakagawa 2012], but not completed
- Alias analysis with type systems
  - Ruby doesn't support static type checking
- Dynamic alias analysis with runtime checking
  - We need to reduce dynamic check overhead
  - We can't insert dynamic checking completely (this is why I found ***"membership transfer"***)