

Making ***MaNy*** threads on Ruby

Koichi Sasada
Cookpad Inc.
ko1@cookpad.com

RubyKaigi 2022

MaNy Project

! The name “MaNy” is invented by @nobu

- Goal: Make **MANY** threads ($> 100K$)
 - Support massive network concurrent connections
 - HTTP/2, WebSocket, GRPC, ...
 - Like Go, Erlang, ...
 - Lightweight Ractor creation
 - Many actors like Erlang
- Technique: M:N threads
 - **M** native threads (M is about `nproc`) and **N** ($> 100K$) Ruby threads
 - Current: 1:1 model (N Ruby threads on N native threads)
 - Great reference to Go's implementation
 - Two-level scheduling
 - Ractor level M:N scheduling
 - Thread level 1:N scheduling



About Koichi Sasada

- Ruby interpreter developer employed by Cookpad Inc. (2017~) with @mame
 - YARV (Ruby 1.9~)
 - Generational/Incremental GC (Ruby 2.1~)
 - Ractor (Ruby 3.0~)
 - debug.gem (Ruby 3.1~)
 - ...
- Ruby Association Director (2012~)
 - [2022 Call for Grant Proposals](#) (~ Oct 3, 2022)





Motivation

Highly concurrent language Ruby

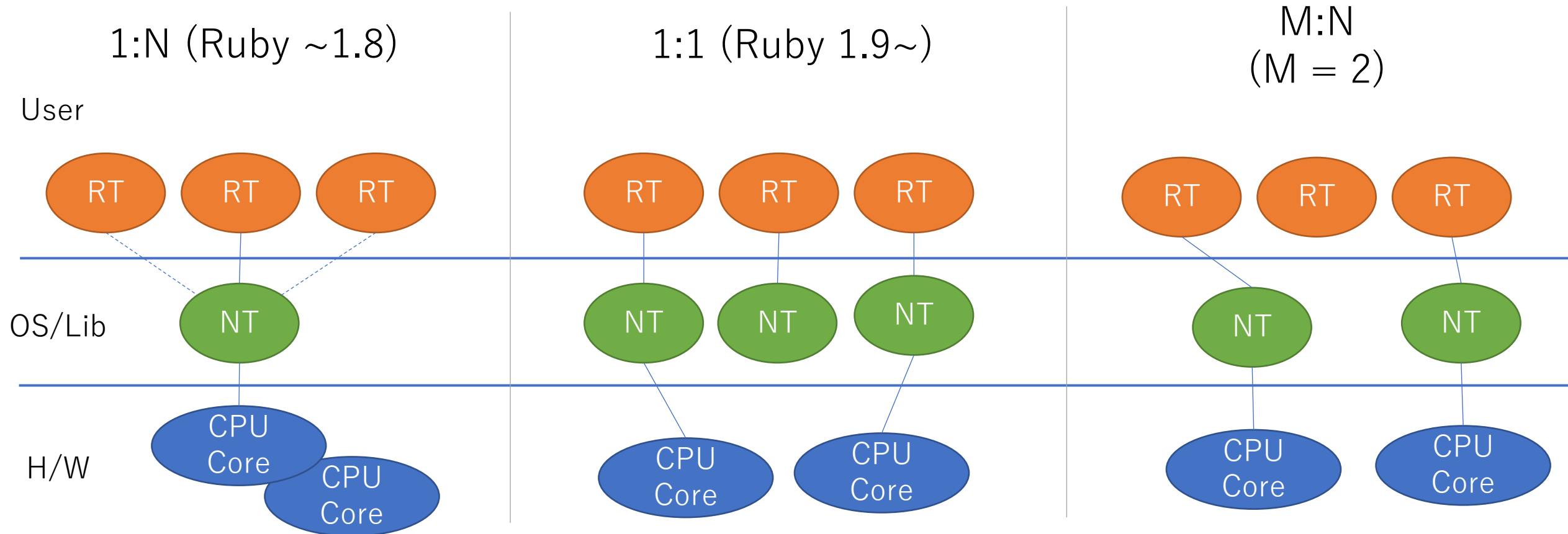
- Support to make a highly concurrent applications by ***scripting*** in Ruby
 - Make **MANY** threads (> 100K)
 - Support massive network concurrent connections
 - HTTP/2, WebSocket, GRPC, ...
 - Like Go, Erlang, ...
 - **Lightweight Ractors**
 - Actor programming with many actors like Erlang
- To encourage **casual** concurrent / parallel programming with Ractor on Ruby

Background

Thread system implementation techniques

 Study in computer science/OS area
 NT: Native thread or kernel thread

How to handle N=3 Ruby threads (RTs) on 2 CPU cores?

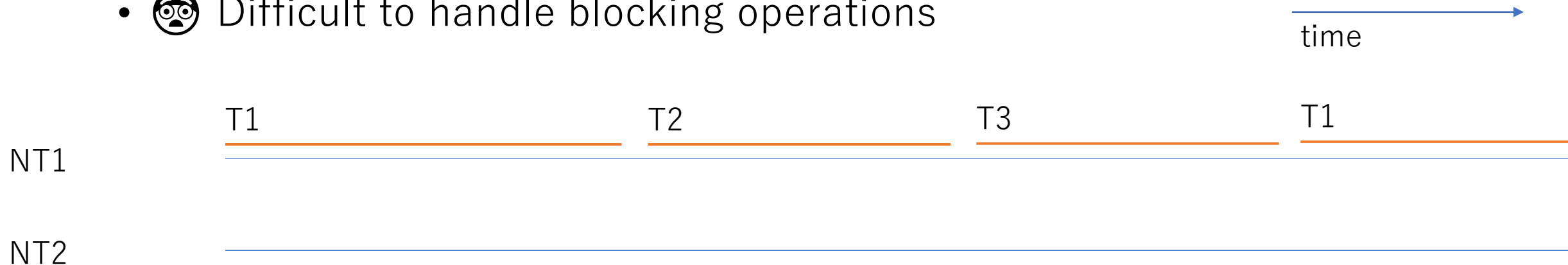




1:N model

Green threads, user level threads, ...

- Only 1 native thread (NT) run multiple threads
 - Ruby ~1.8
 - 😊 Completely controllable
 - 😊 Lightweight (in theory)
 - 😱 Can not run in parallel
 - 😱 Difficult to handle blocking operations



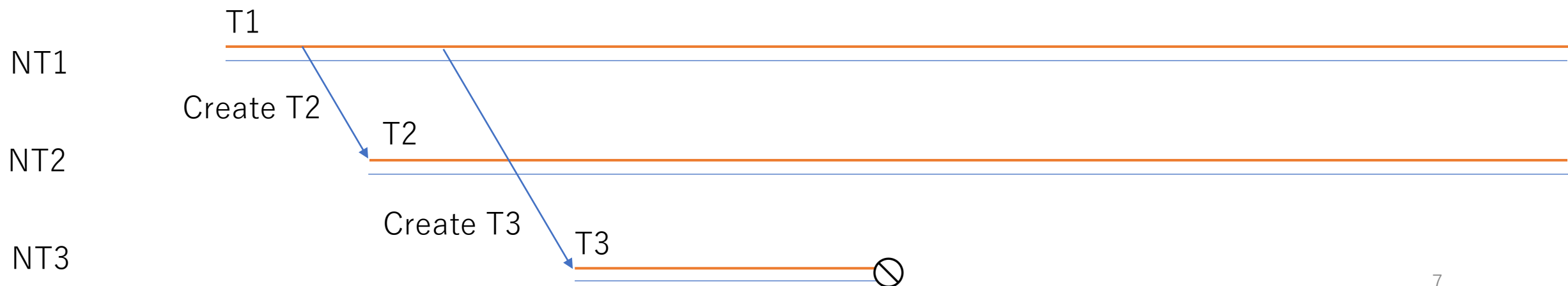
! Only NT1 is used!



1:1 model

Most simplified technique

- 1 native thread per Ruby thread
 - Ruby 1.9~ (has GVL limitation. This page eliminates it)
 - 😊 Simple, easy to handle blocking operations (system does)
 - 😊 Can run in parallel on multi-core systems
 - 😬 More overhead (compare with 1:N, in theory)
 - 😬 Less controllable (only native thread system schedules)



M:N threads

Existing implementations

- Operating systems
 - Solaris, FreeBSD, ... (now they use 1:1 threads)
- Language native
 - Go's goroutines
 - Erlang's (Elixir's) processes
- Extensions
 - Java's virtual threads (JEP 425: Virtual Threads)
 - Rust's tokio, ...

Ruby's case

- Threads run in concurrently, not in parallel
 - **Concurrent**
 - Context switch by time slice (100ms)
 - Switch on blocking operations like blocking I/O
 - **Not in parallel:** A giant lock (GVL) keeps running **1 thread** on a Ractor.
- Ractors can run in parallel
 - A Ractor has at least **1 thread**
 - Threads on different ractors can run in parallel
- 1:1 threads can not make 100K threads/ractors because of some limitations
 - Creation overhead and OS resource limitations
 - M:N technique is used by recent “concurrent” languages and how about on Ruby like Go language without changing anything ...?

⚠ GVL had stood for “**Global VM Lock**” because it is only 1 lock per VM (only 1 thread has GVL can run on a Ruby VM). However, GVLs are available per a ractor. In this talk we continue to use the word GVL and we can say Great Valuable Lock or anything you like.

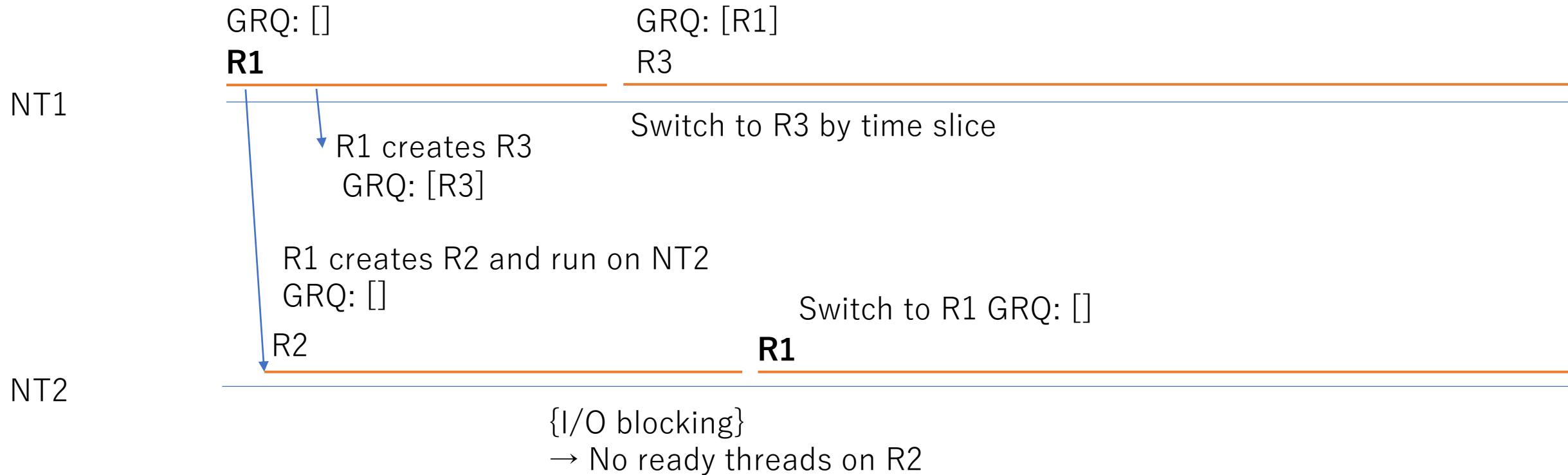
MaNy project



Introduce M:N scheduler into Ruby

Two-level scheduling

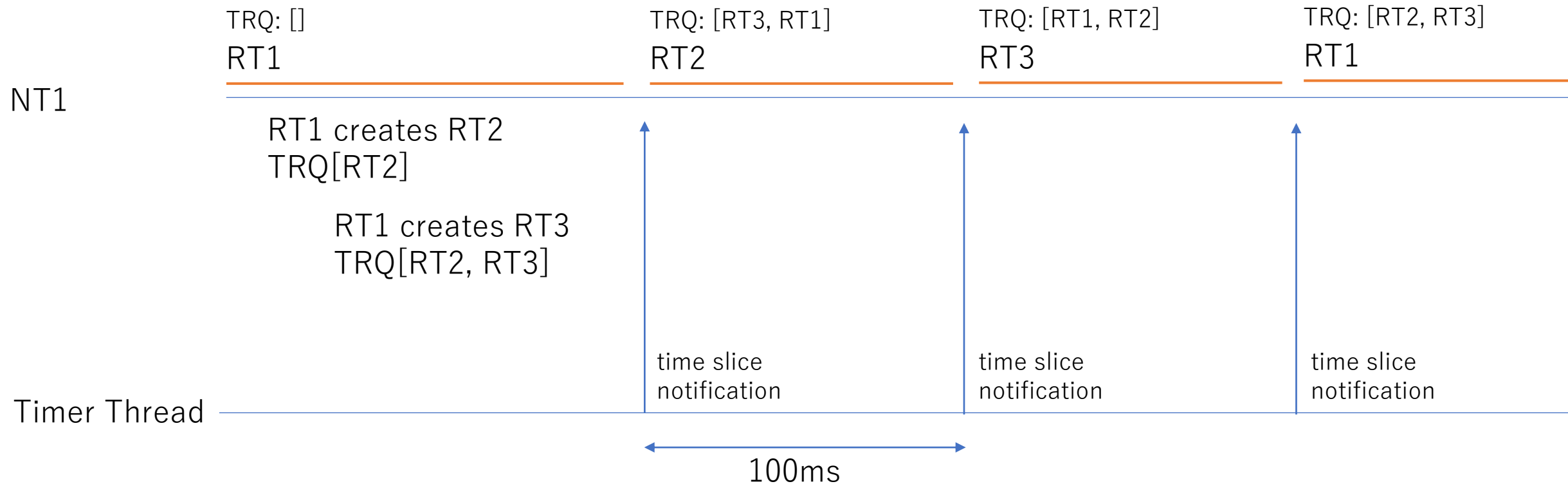
- Ractor level scheduling
 - Run ractors on M native threads (**M:N model**)
 - Global ready queue (GRQ) manages runnable Ractors
- Thread level scheduling
 - Run threads on 1 native thread (**1:N model**)
 - Thread ready queue (TRQ) manages runnable Ruby threads in a Ractor
 - Use multiple native threads to handle unmanaged blocking ops.
 - Support **1:1 model** mode for compatibility
- Misc
 - Both FIFO scheduler (no priority queue(s))
 - Re-introduce a **timer thread** to manage context switch (Go's sysmon)

M:N Ractor level scheduling (M=2)



-  Ractor R1, R2, R3 have 1 thread, respectively.
-  R1 runs on NT1 and NT2 (M:N scheduler)

1:N Thread level scheduling in a Ractor



Handle blocking operations

- We need to handle “Blocking operations” for user level scheduling, otherwise all threads stops
- Blocking operations
 - Managed blocking operations
 - I/O (most of read/write)
 - manage by I/O multiplexing API (select, poll, epoll, kqueue, IOCP, io_uring, ...)
 - Sleeping
 - Synchronization (Mutex, Queue, ...)
 - Unmanaged blocking operations
 - All other blocking operations not listed above, written in C
 - Huge number calculation like Bignum#*
 - DNS lookup
 - I/O (can not detect block-able or not)
 - open on FIFO, close on NFS, ...
 - ...



Handle blocking operations on existing Ruby versions

- Ruby ~1.8 on 1:N model
 - Only handle managed blocking operations.
 - Can not handle unmanaged blocking operations... (stop all threads).
 - ex) system's DNS resolver can stop all threads, but resolve.rb does not stop.
 - 🤪 Difficult to handle blocking operations (only the interpreter can do it)
- Ruby 1.9~ on 1:1 model (2007~, 15 years)
 - Can handle all blocking operations by releasing GVL
 - “`rb_thread_call_without_gvl(func)`” API ← ❤️ Easy to use
 - Other threads can run.
 - Not handled blocking operations can stop all threads, but **after 15 years**, most of critical operations are handled with the API ❤️

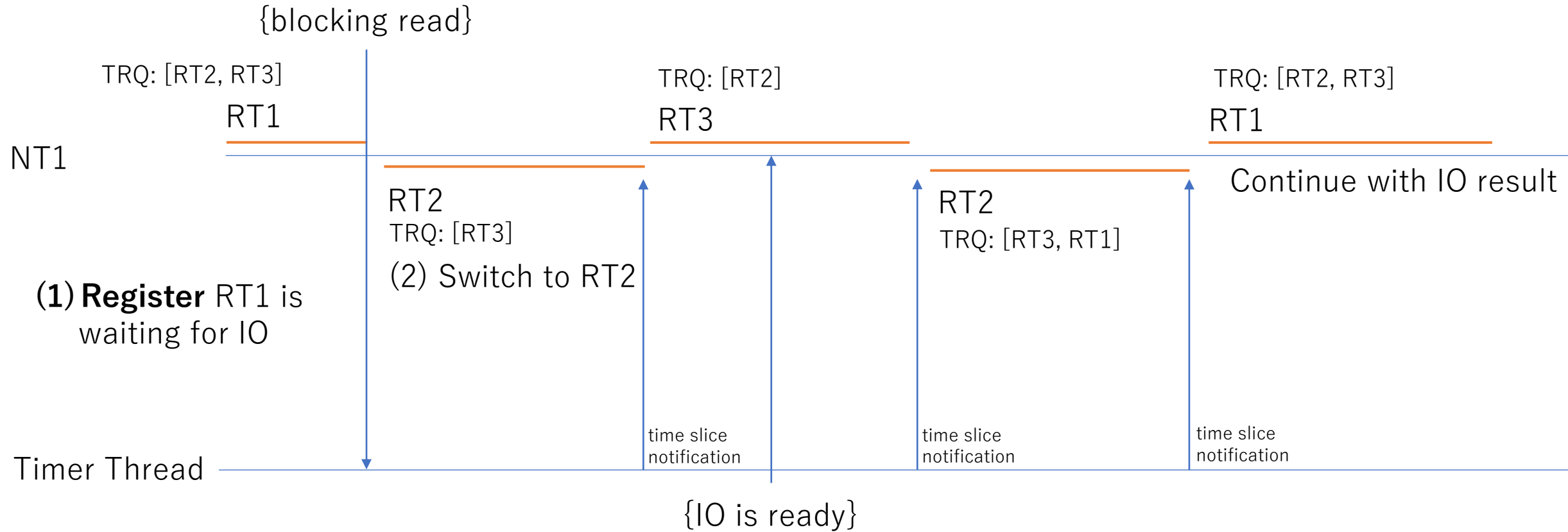
Handle blocking operations on MaNy

- Managed blocking operations
 - Blocking I/O, sleep, synchronizations are managed by a scheduler
 - Timer thread observe I/O events and schedule
 - epoll, kqueue, IOCP (now only epoll is supported)
- Unmanaged blocking operations
 - Run a blocking operations on a native threads marked by **dedicated**
 - DNT (Dedicated native thread) pinned down to a Ruby thread
 - Add a new native thread to run other threads if blocking operations consume a time
 - Timer thread observes native threads count for scheduling





Handle managed blocking operations



(1) **Register** RT1 is waiting for IO

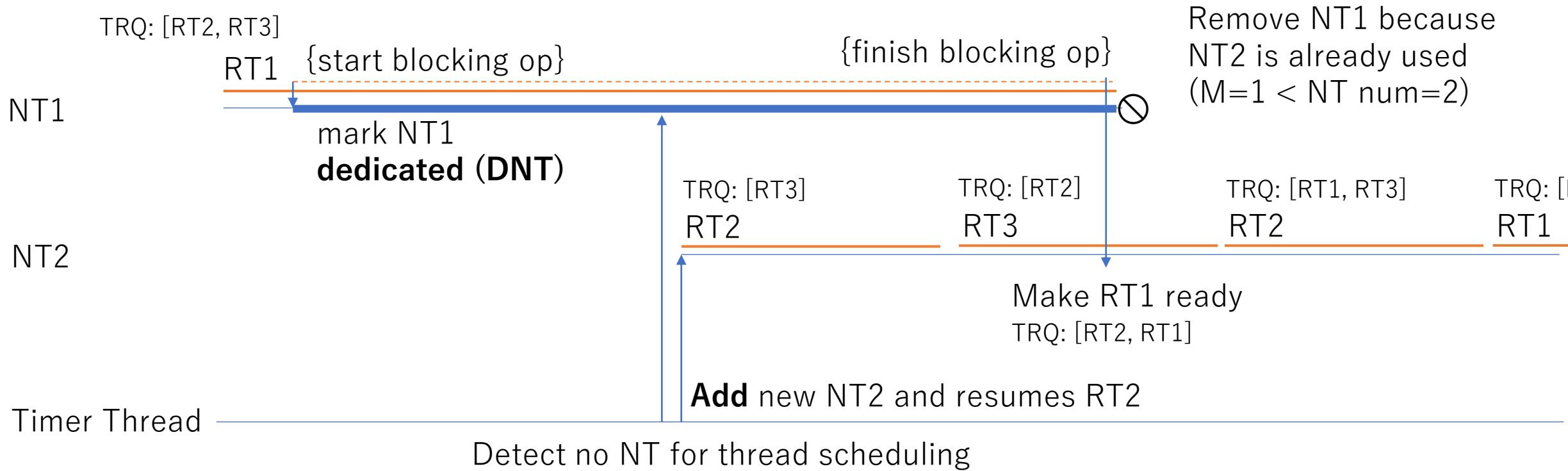
(2) Switch to RT2

(3) Add RT1 to ready queue
→ TRQ: [RT2, RT1]

? RT1 can be scheduled early

Start status:
Ruby threads RT1, RT2, RT3 are there
TRQ (Thread Ready Queue) is [RT2, RT2]

Handle unmanaged blocking operations (M=1)



Start status:

Ruby threads RT1, RT2, RT3 are there

TRQ (Thread Ready Queue) is [RT2, RT2]

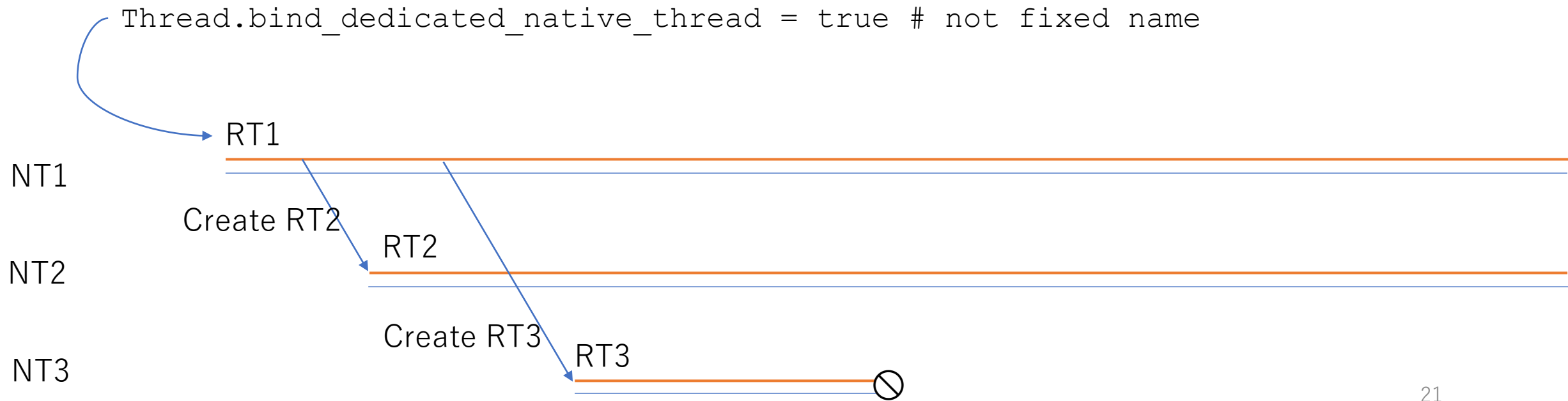
⚠ There are many DNT (\leq RT num)

Compatibility issue

- C-extensions can rely on 1:1 model
 - Doesn't work if it relies on NT's Thread-Local-Storage (TLS)
 - ... other cases? depends on native thread ID?
- Compatibility mode
 - Bind dedicated native thread (DNT) → Force 1:1 model mode
 - Go's `runtime.LockOSThread()`
 - ex) `Thread.bind_dedicated_native_thread = true`
 - Default?
 - Bind DNT if user specifies (Go's approach, because Go hadn't compat. issue)
 - Bind DNT only with the main thread (← Current choice)
 - Bind DNT with threads on the main reactor (safer choice)
 - Support scheduling with normal NT and DNT (complex)



Bound threads for compatibility 1:1 model



Advantages of M:N

- Lightweight threading (and ractor system)
 - Remove impedance mismatch between NT scheduler and Ruby's scheduler especially with GVL
 - No synchronization is needed to control within a Ractor
 - (Slightly) lightweight creation
 - `pthread_create()` is fantastic fast on recent systems, so not big impact

Disadvantage of M:N

- Compatibility issue (already discussed)
- Complication
- Overhead
 - (Slightly) overhead for I/O by checking blockable operation or not
 - (Slightly) introducing a timer native thread for management
- Change visibility outside from Ruby process
 - Metrics tools like ps, top, ...
 - gdb (debugger)
 - hard to debug native threads without not running RTs
 - `rb_bug()` prints huge lines of memory maps (thread count * 2)

Compare with fiber scheduler (Ruby 3.0~)

- Similar motivation (to support huge number of connections)
- Good for fiber scheduler
 - Ruby user can write a scheduler
 - Can mark **best score** with the well-tuned scheduler for the application
 - Non-MRI implementations can use the scheduler written in Ruby
- Good for MaNy Project
 - Ruby users don't need to care about a scheduler
 - Can mark **better score** without changing threaded apps
 - (Hopefully) no hungs on unmanaged blocking operations (like current threads)
 - Timeslice is provided (auto-context switch)
 - Integration with Ractor system → parallel computation



Other interesting topics

- $O(n \dots)$ traps
 - Deadlock detection code can iterate all threads
 - Works well for small number of threads, but superslow for many threads ($O(nm)$)
 - `Thread.list` uses `alloca(thead_num)` and it causes stack overflow
 - `rb_notify_fd_close()` iterates all waiting fds ($O(n^2)$)
 - ...
- Implementation
 - Two level scheduling with dedicated native threads (1:1)
 - Signal handling
 - Context management (stack management, context switching, ...)
 - Time slice provider
 - Debug techniques (logging system)
 - Scheduler functions (naming)
- Others
 - Mysterious OS limitations for native threads
 - `top` (linux /proc) sometimes shows strange values
 - ...

Evaluation

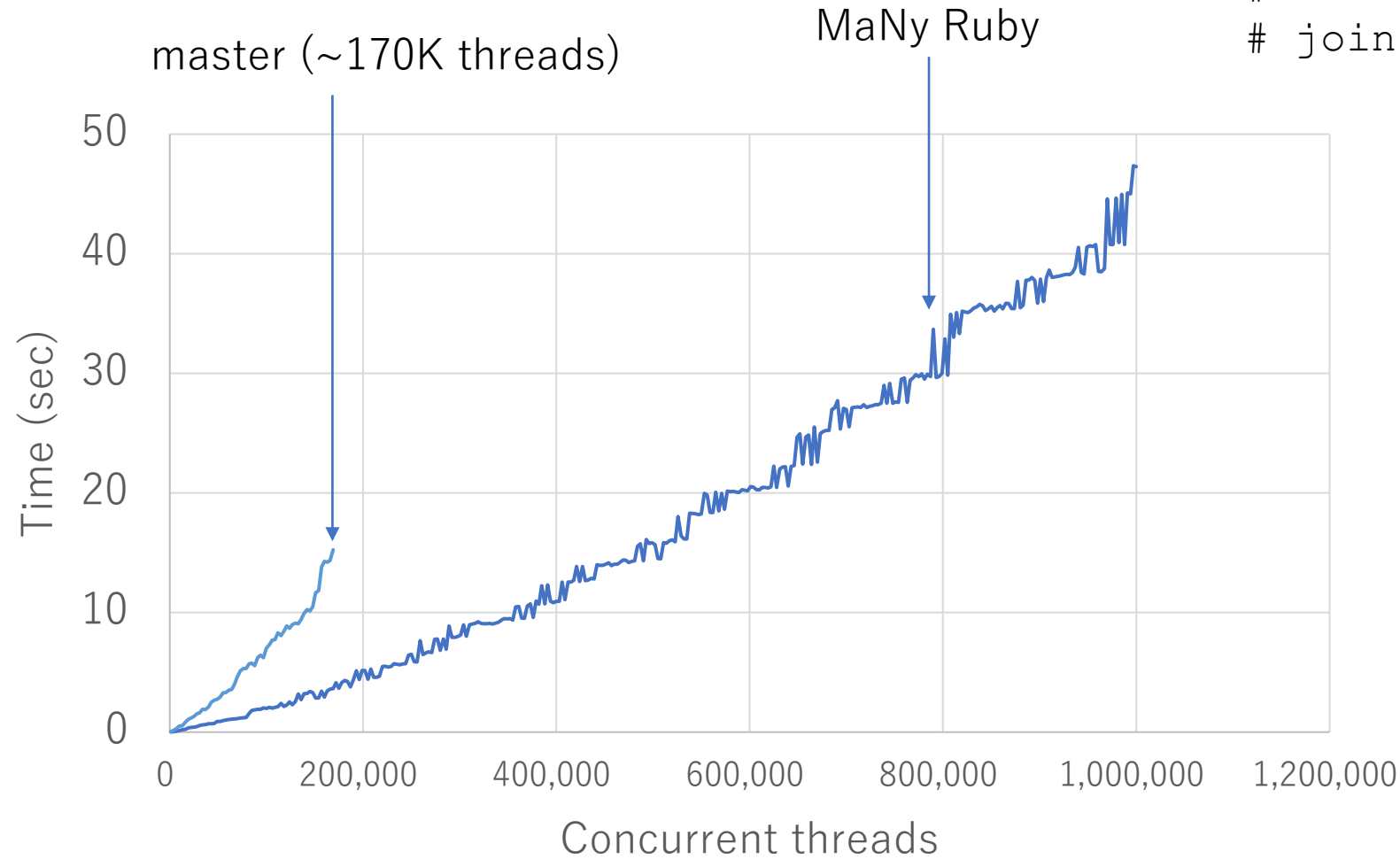


Evaluations environment

- Machine
 - Intel(R) Core(TM) i7-6700 CPU, 4 cores, 8 HW threads, 64GB mem
 - Linux 5.4.0-125-generic (Ubuntu 20.04)
 - `ulimit -n `ulimit -Hn`` (to increase open files for sockets)
 - `vm.max_map_count = 2,000,000` (a thread consumes 2 maps)
 - Base ruby: ruby 3.2.0dev (2022-07-26T07:03:44Z master 9a8f6e392f) [x86_64-linux]
 - `RUBY_THREAD_VM_STACK_SIZE=32768` (32KB, default: 1MB) because master speed decreases in proportion to the VM stack size (will be solved)
- Client machine
 - AMD Ryzen 9 5900HX , 8 cores, 16 HW threads, 16GB mem
- Code
 - MaNy branch: <https://github.com/ko1/ruby/tree/many>
 - Examples: https://github.com/ko1/many_examples

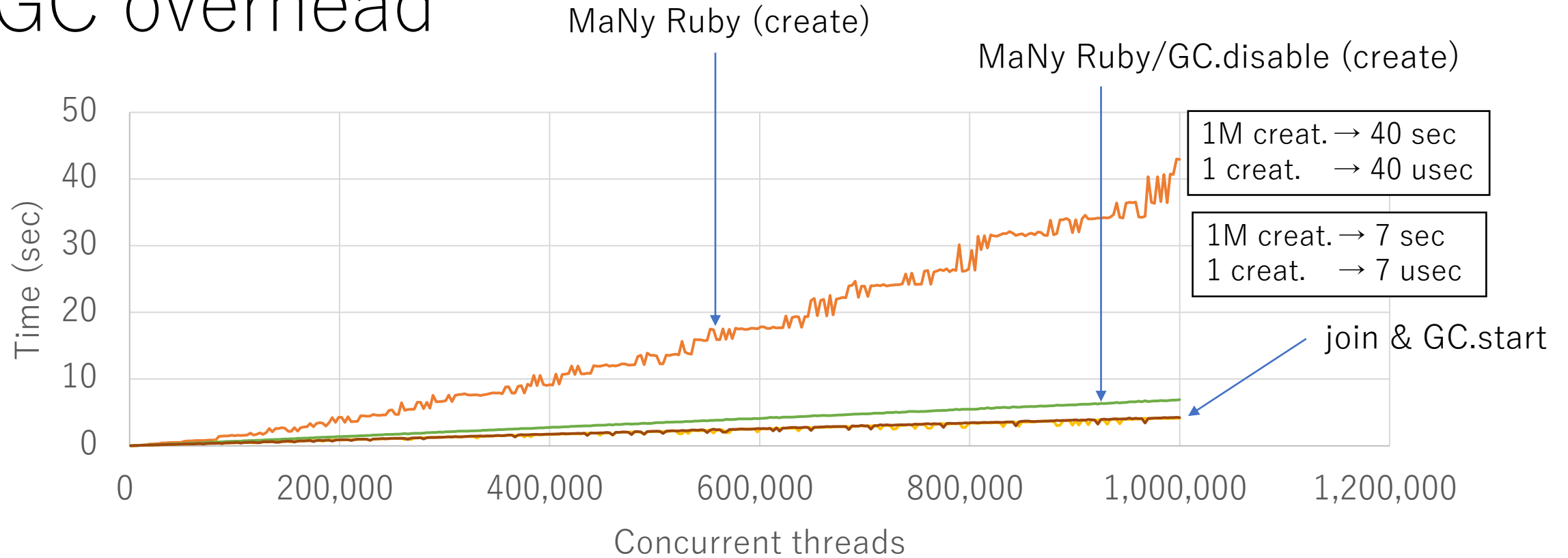
Thread creation

```
# make x threads and stop them  
# restart them  
# join all and GC.start
```



Thread creation GC overhead

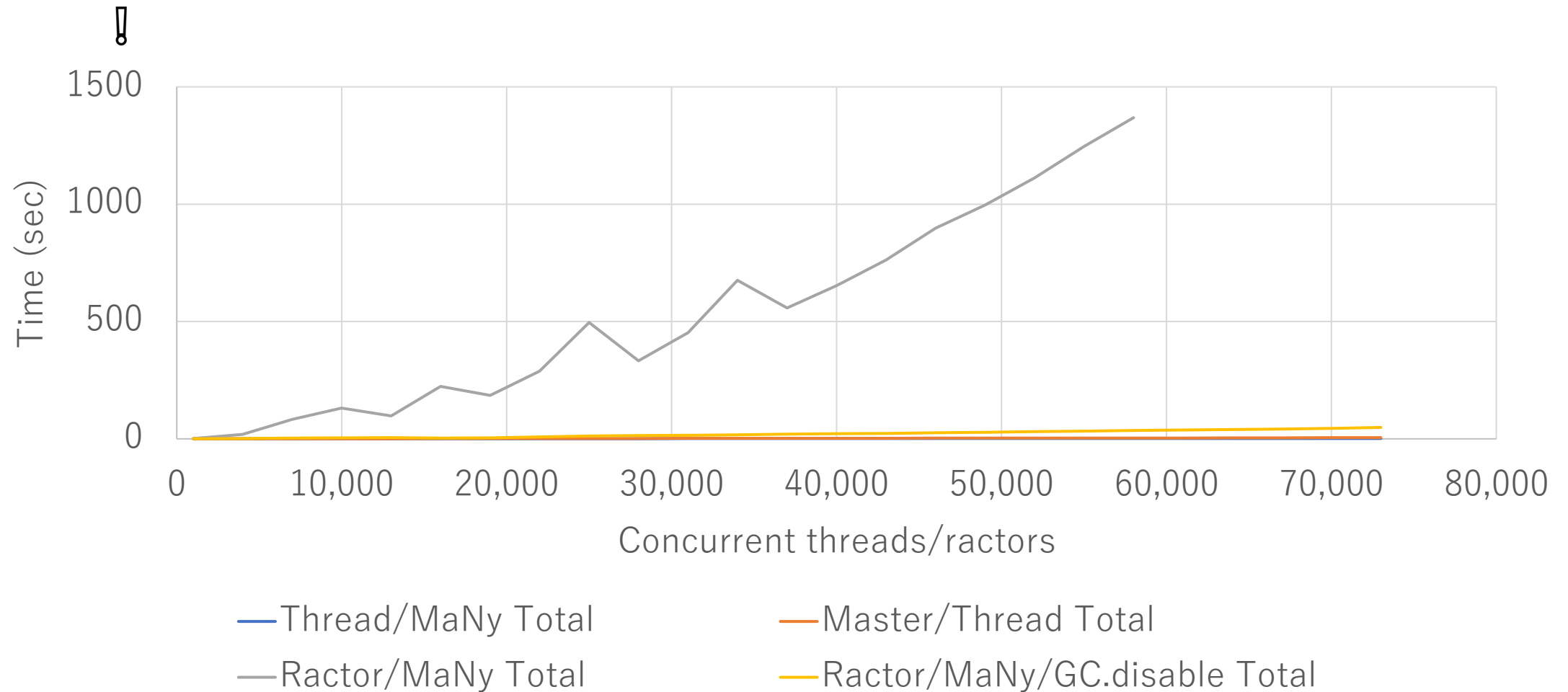
```
# GC.disable  
# make x threads and stop them  
# restart them  
# join all and GC.start
```



- Many/Thread create
- Many/Thread join/GC
- Many/Thread/GC.disablee create
- Many/Thread/GC.disablee join/GC



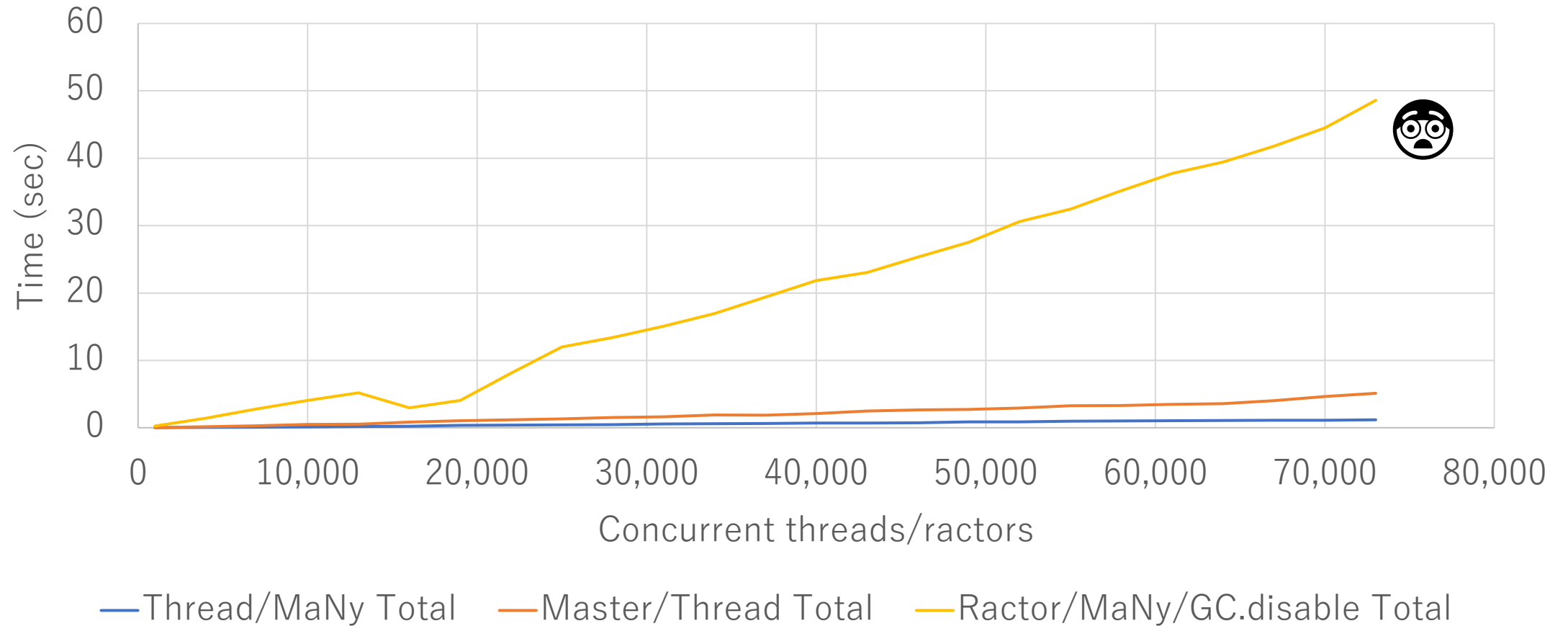
Ractor creation





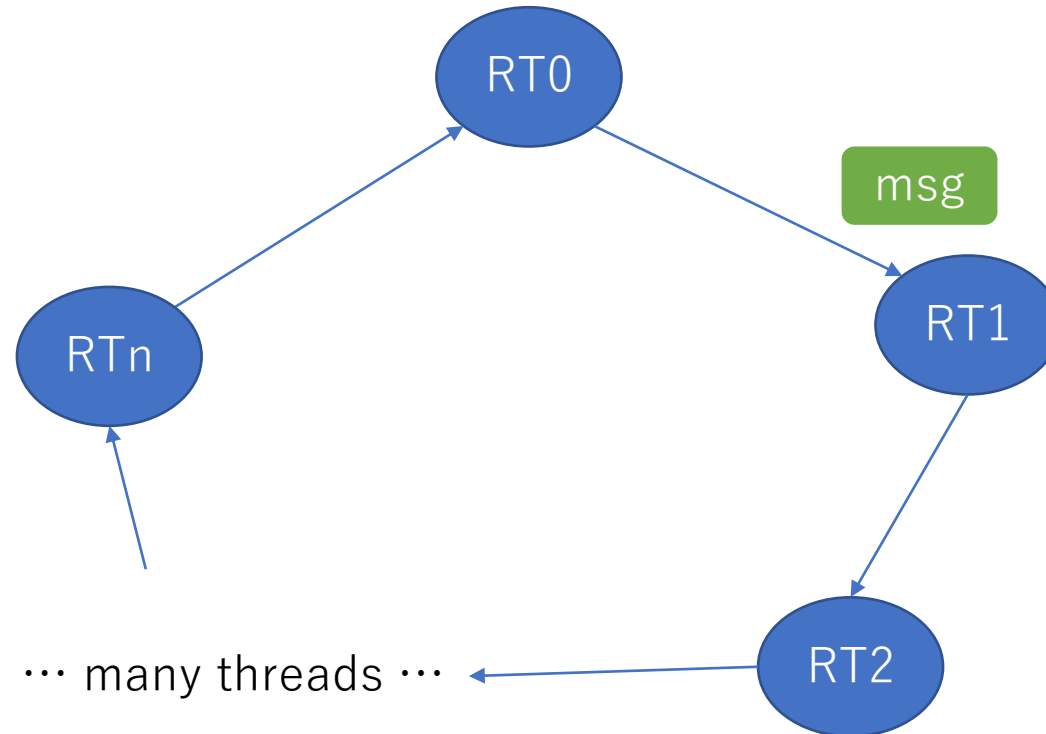
Ractor creation

⚠ Synchronization issue...?



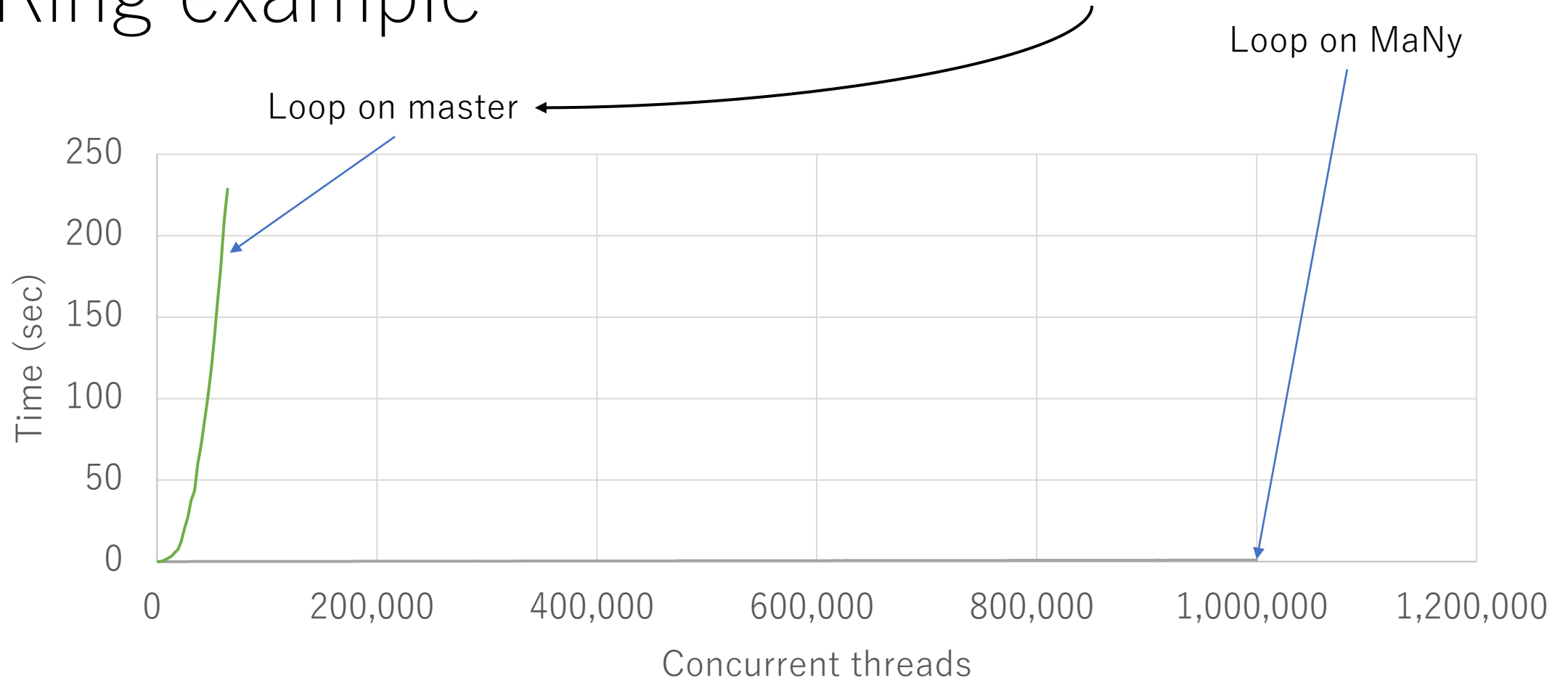
Ring example

- Prepare n threads ordered sequentially
- Pass a message to the next Ruby thread with Queue



! Kernel/Ruby scheduler's impedance mismatch (GVL handling)

Ring example

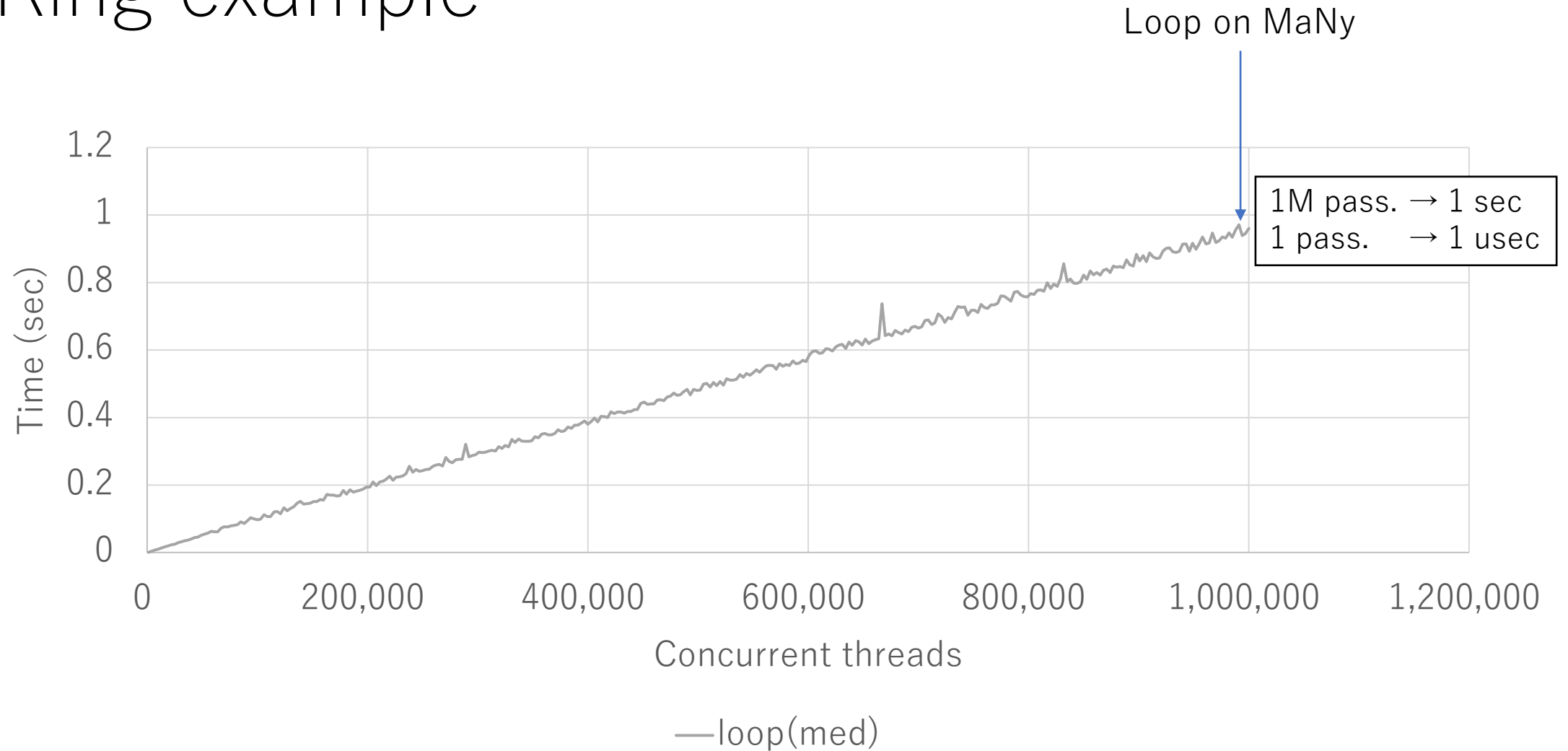


— loop(med) — loop(med)/master

(med: median value of 3 trials)



Ring example

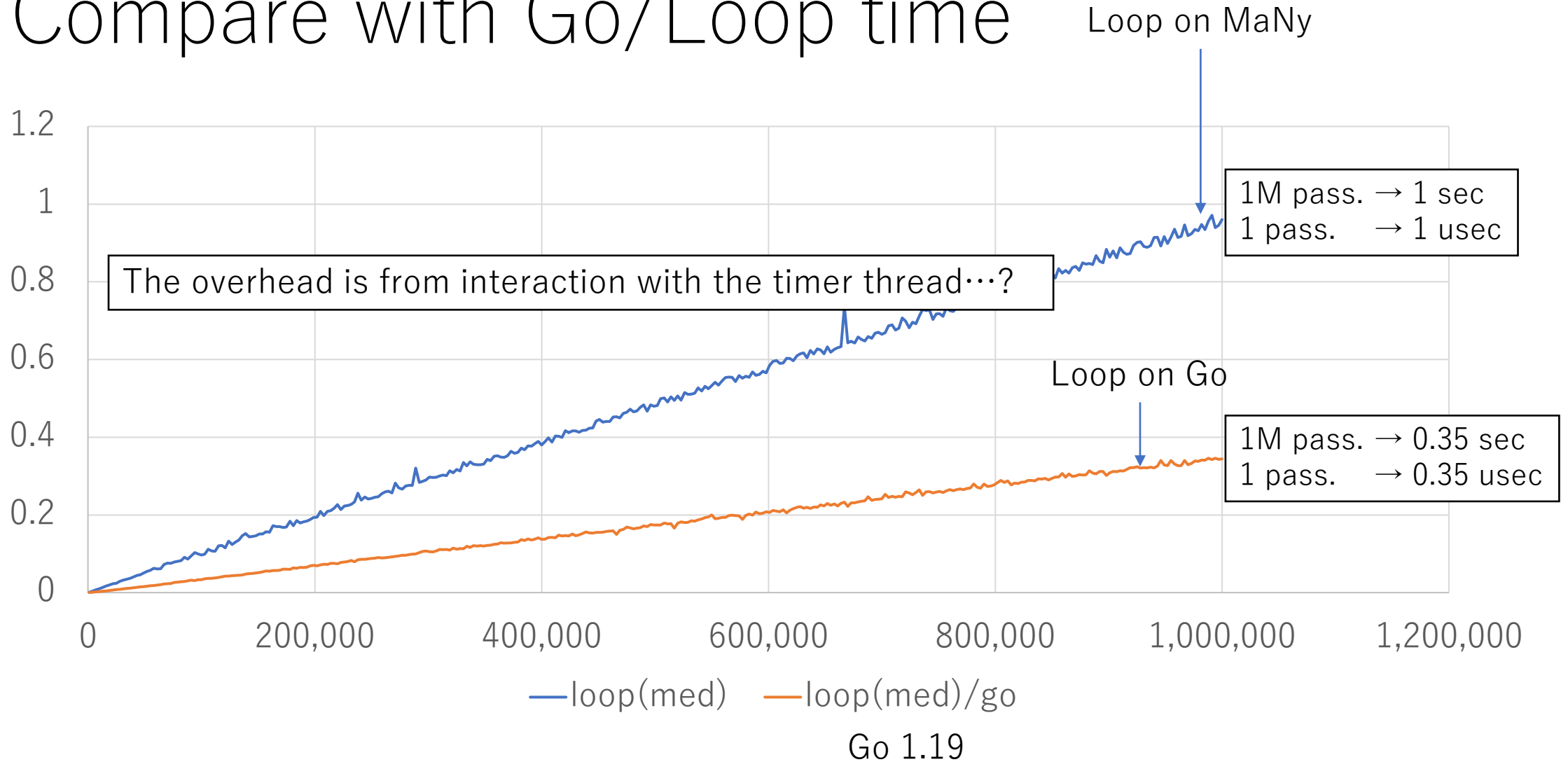


(med: median value of 3 trials)



Ring example

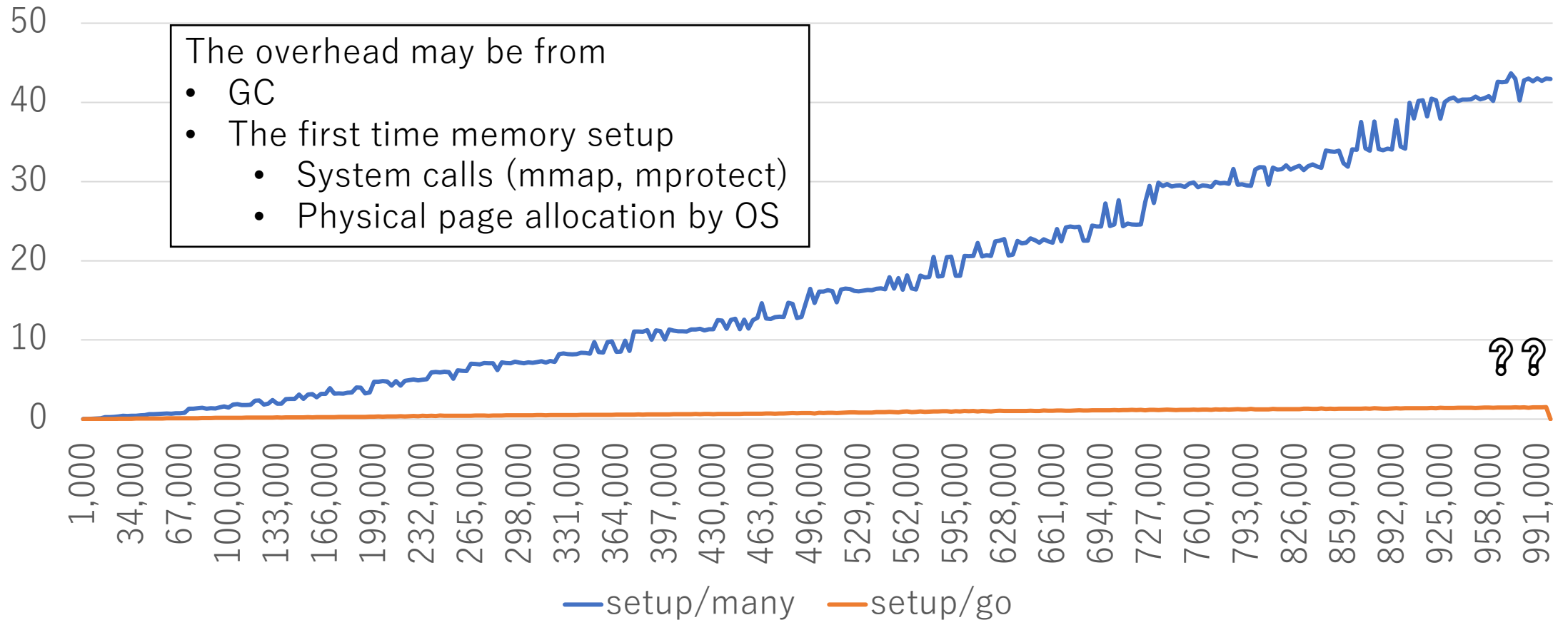
Compare with Go/Loop time



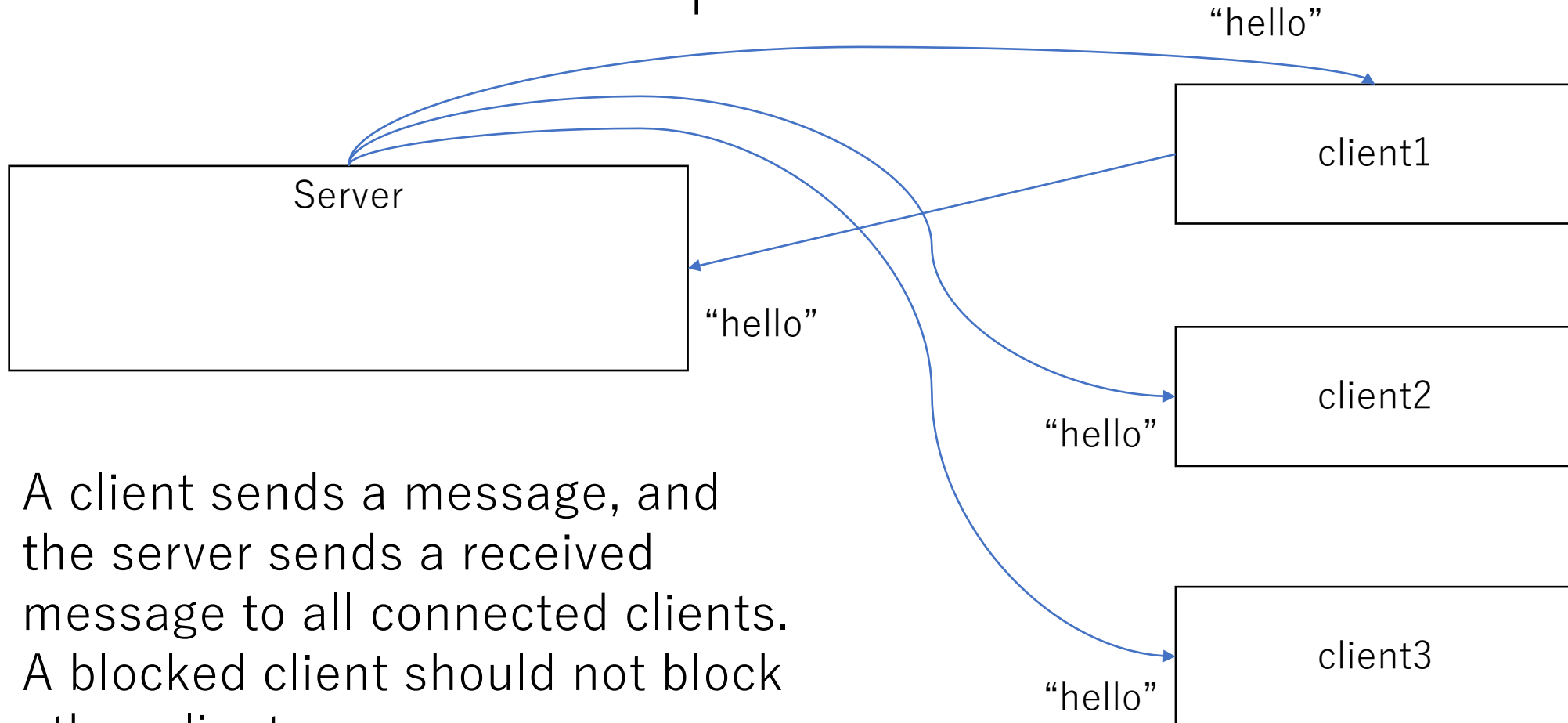


Ring example

Compare with Go/Creation time



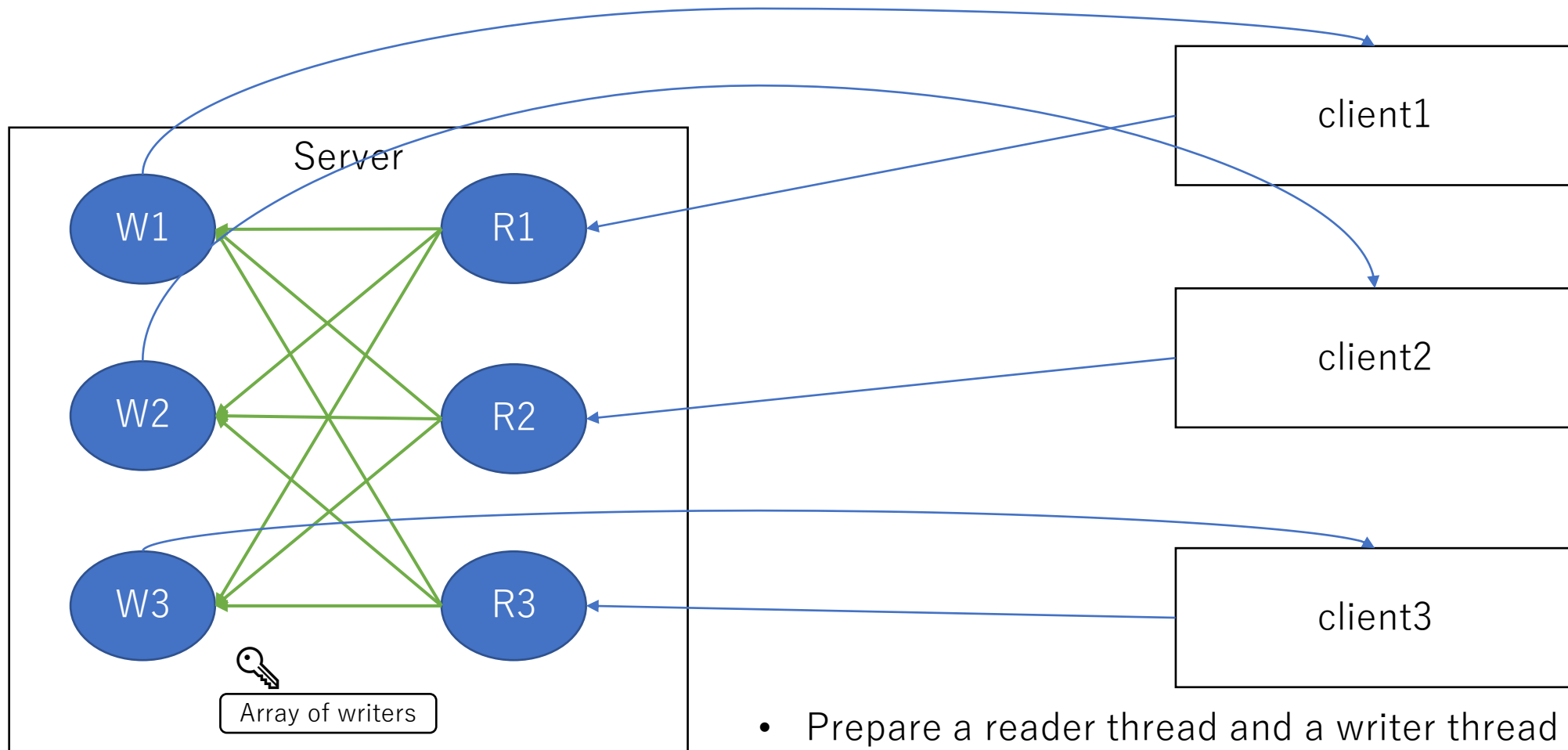
Chat server example



- A client sends a message, and the server sends a received message to all connected clients.
- A blocked client should not block other clients
- The server and clients are connected with TCP/IP



Chat server example: Queue version

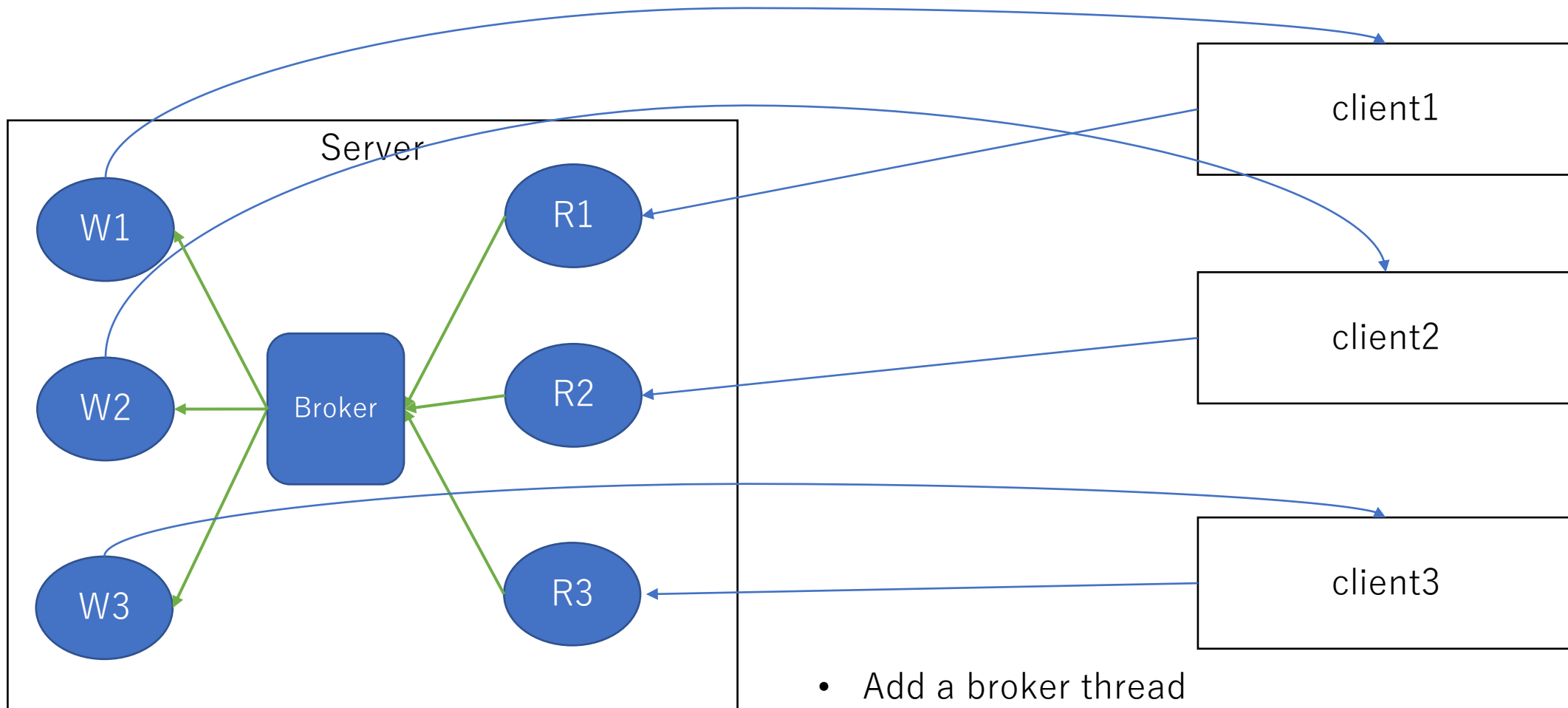


Readers and writers are connected with Queues (green lines)

- Prepare a reader thread and a writer thread for each connection. → $2 \cdot n$ threads for n connections
- Readers and writers should be separated because writing to the socket can block.



Chat server example: Broker version



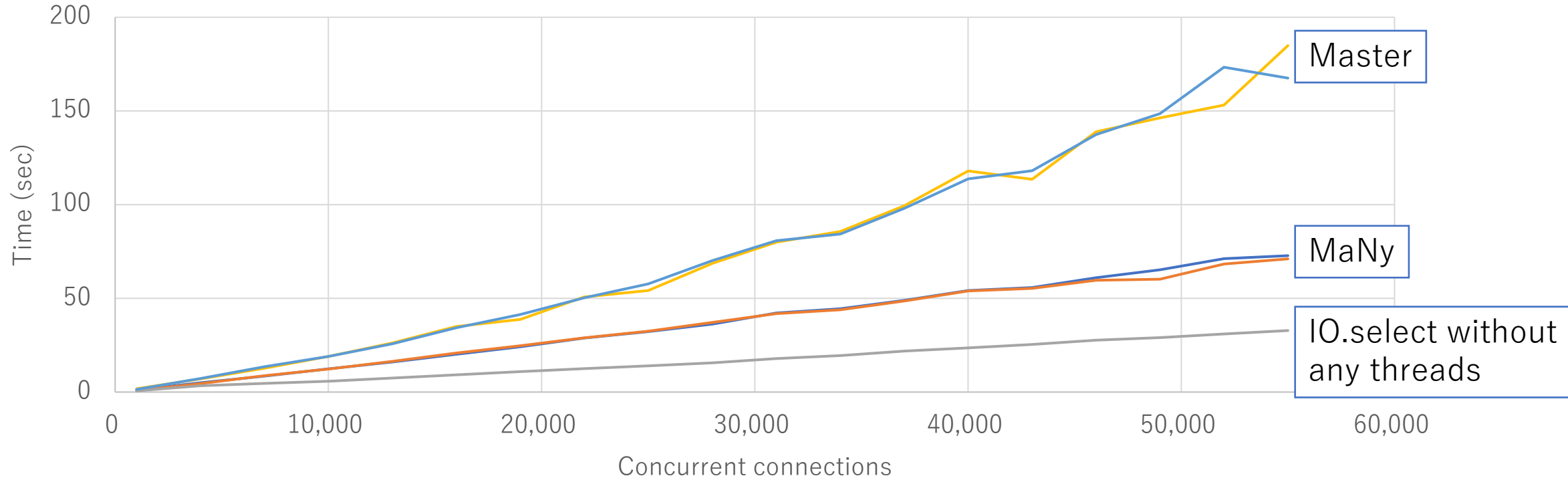
Readers and writers are connected with Queues (green lines)

- Add a broker thread
- Readers send message to the broker, and it broadcast to writers. No Mutex🔑!

Chat server example

RTT x 100

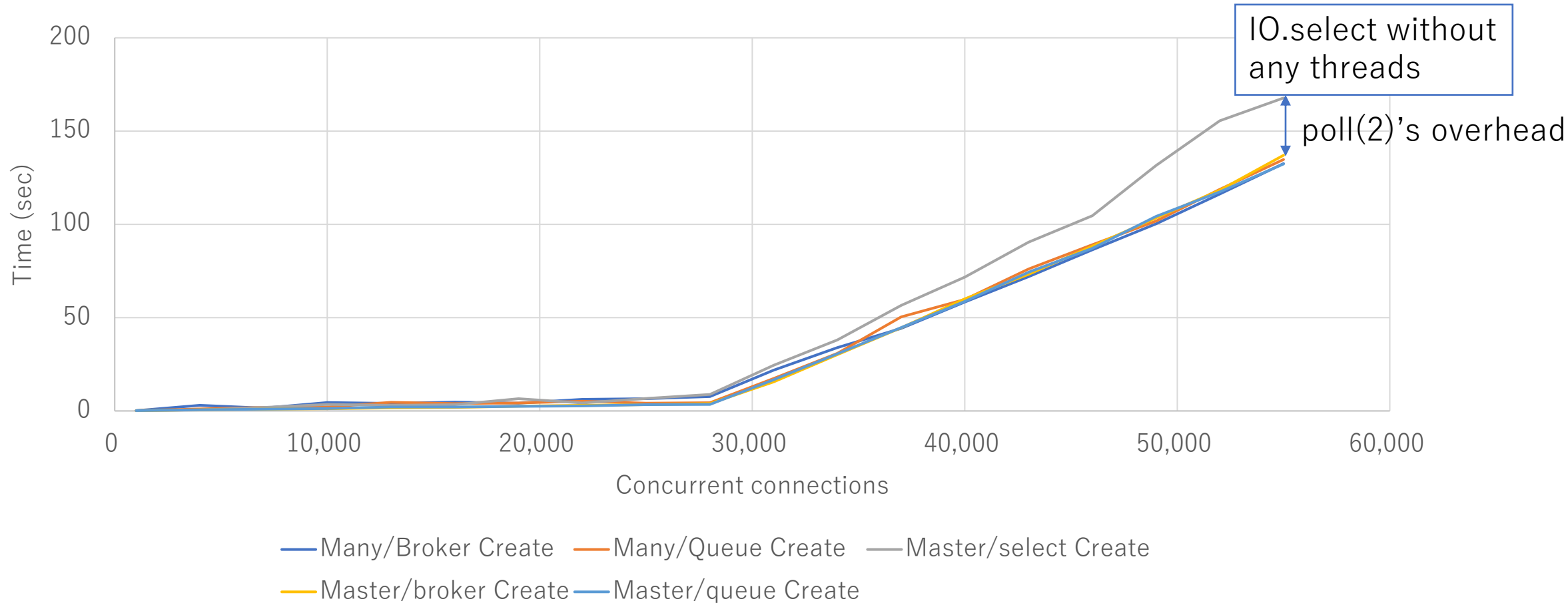
```
# from clients
100.times{
  socks[n/2] << "msg"
  socks.each{|s| s.gets}
}
```



— Many/Broker 100*(post+get*n) — Many/Queue 100*(post+get*n) — Master/select 100*(post+get*n)
— Master/broker 100*(post+get*n) — Master/queue 100*(post+get*n)

Chat server example

Connection time



Future work

- MaNy project
 - Complete implementations
 - Ractor support
 - Increase supported platforms (Mac, Windows, ... Now Linux only)
 - Merge into master (3.2...?)
- Ractor local GC (working with a GSoC student)
- Feasible debugger for massive concurrent application

→ Make Ruby as a **casual** concurrent language

Acknowledgements

- Matz, mame and other reviewers of our design
- Yuki Torii helps me a lot to attend RubyKaigi 2022

MaNy Project

- Goal: Make **MANY** threads ($> 100K$)
 - Support massive network concurrent connections
 - HTTP/2, WebSocket, GRPC, ...
 - Like Go, Erlang, ...
 - Lightweight Ractor creation
 - Many actors like Erlang
- Technique: M:N threads
 - **M** native threads (M is about `nproc`) and **N** ($> 100K$) Ruby threads
 - Current: 1:1 model (N Ruby threads on N native threads)
 - Great reference to Go's implementation
 - Two-level scheduling
 - Ractor level M:N scheduling
 - Thread level 1:N scheduling