COMP 322: Fundamentals of Parallel Programming

Lecture 33: Combining Distributed and Multithreaded Parallelism (Hybrid Parallelism)

Instructors: Vivek Sarkar, Mack Joyner
Department of Computer Science, Rice University
{vsarkar, mjoyner}@rice.edu

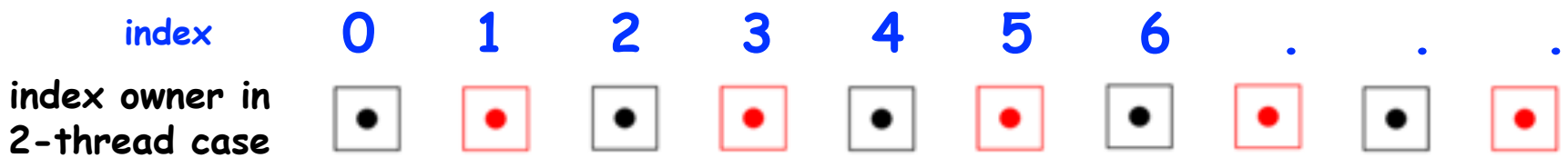
http://comp322.rice.edu/

Worksheet #32 solution: UPC data distributions

In the following example (which is similar to slide 17, but without the blocking), assume that each UPC array is distributed by default across threads with a cyclic distribution. In the space below, a) identify an iteration of the upc_forall construct for which all array accesses are local, and b) an iteration for which all array accesses are non-local (remote). You can assume any values for THREADS in the 2…99 range that you choose for parts a) and b). Explain your answer in each case.

Note that each shared array's distribution always starts with the first element assigned to thread 0 (not where the previous array may have ended).

```
1.  shared int a[100],b[100], c[100];
2.  int i;
3.  upc_forall (i=0; i<100; i++; (i*THREADS)/100)
4.      a[i] = b[i] * c[i];
```



Solution:

- Iteration 0 has affinity with thread 0, and accesses a[0], b[0], c[0], all of which are located locally at thread 0
- Iteration 1 has affinity with thread 0, and accesses a[1], b[1], c[1], all of which are located remotely at thread 1

Processes and Threads (recap from Lecture 18)

- A Java Virtual Machine (JVM) executes in a single *process* with multiple *threads*
- Threads associated with the same process can share the same data
- Processes are the fundamental building block for distributed applications, since a process has to be confined to a single node

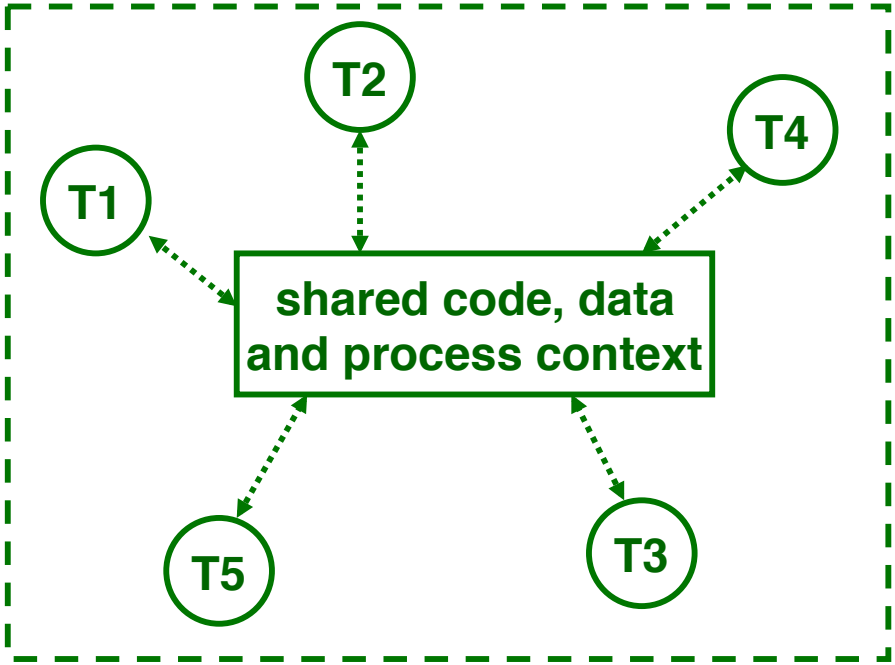


Figure source: COMP 321 lecture on Concurrency (Alan Cox)

- Need to create multiple processes to use multiple nodes
  - Also need to create multiple threads within a process to use multiple cores within a node
  - May also create more than one process in a node

Using multiple threads/processes per node

- Advantages of using multiple threads in a process
  - —Efficiency: more efficient utilization of shared resources compared to one thread per process
  - — Responsiveness: other threads can respond to requests when one thread is blocked e.g., due to an I/O request
  - —Performance: increased throughput, since multiple threads can execute on multiple cores
- Advantages of using multiple processes in a node (each process with multiple threads)
  - —Responsiveness: other processes can respond to requests when one process is blocked e.g., due to garbage collection
  - —Scalability: there is a "sweet spot" for the ideal number of threads to use in a process (which is often less than the number of cores); using additional processes can then help increase performance
  - —Availability/resilience: other processes can respond to requests on node when one process goes down

Example of a Multithreaded Server

```
1.  public void run(final ServerSocket socket, final File rootDir) throws IOException {
2.      /*
3.       * Enter a spin loop for handling client requests to the provided
4.       * ServerSocket object.
5.       */
6.      while (true) {
7.
8.          // TODO Use socket.accept to get a Socket object
9.          Socket s = socket.accept();
10.
11.         Thread thread = new Thread(new RequestHandler(s, rootDir));
12.         thread.start();
13.     }
14. }
```

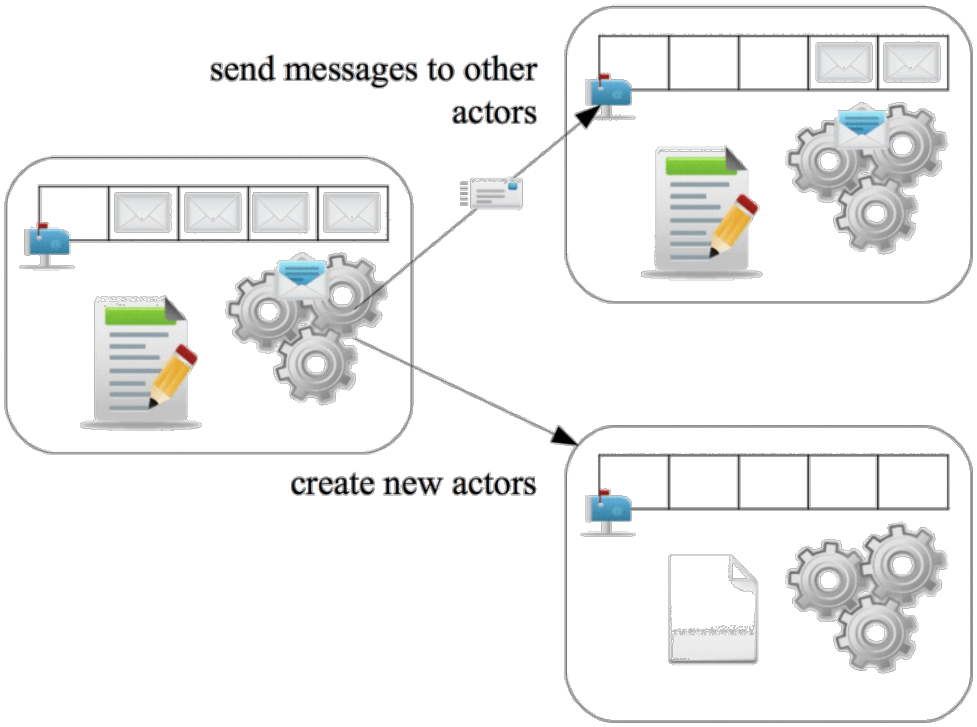

MPI and Threads

- MPI describes parallelism between processes
- Thread parallelism provides a shared-memory model within a process
- MPI-2 defines four levels of thread safety
  - —MPI_THREAD_SINGLE: only one thread
  - —MPI_THREAD_FUNNELED: only one thread that makes MPI calls
  - —MPI_THREAD_SERIALIZED: only one thread at a time makes MPI calls
  - —MPI_THREAD_MULTIPLE: any thread can make MPI calls at any time
- User calls MPI_Init_thread to indicate the level of thread support required; implementation returns the level supported

For MPI_THREAD_MULTIPLE

- When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
- Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions
- It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls
- User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads
- *Often simpler to limit all MPI calls to a single thread …*

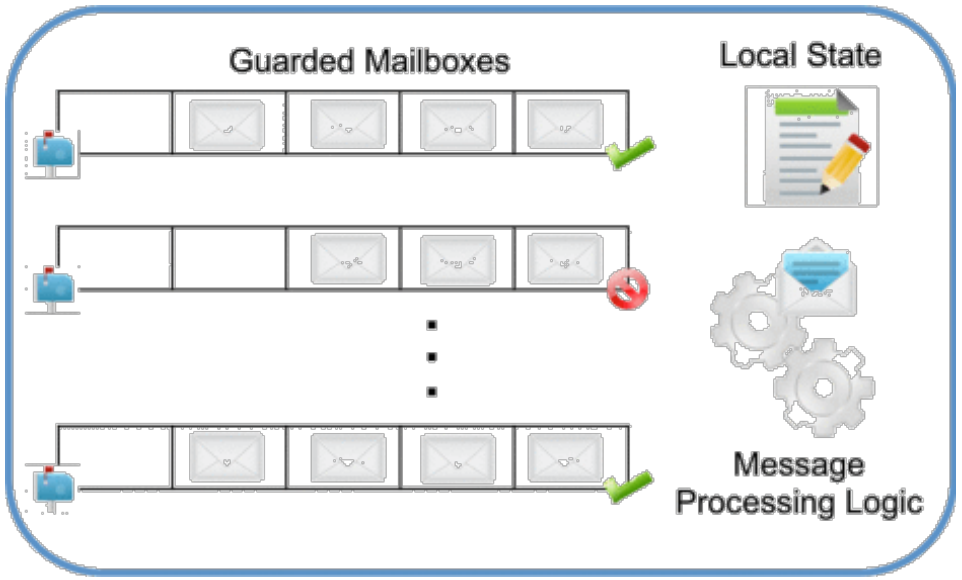
Recap of Actor Model



Source : *S. Imam and V. Sarkar. Integrating Task Parallelism with Actors, OOPSLA '12*

- Logical active thread with isolated data; all inter-actor interactions must be performed via messages
- Inherently concurrent (alternative to transactions)
- Lightweight
- Highly scalable
- Promising for both manycore *and distributed parallelism*
- HJlib extends the actor model by integrating it with task parallelism e.g., by supporting finish construct with actors

Selector Model

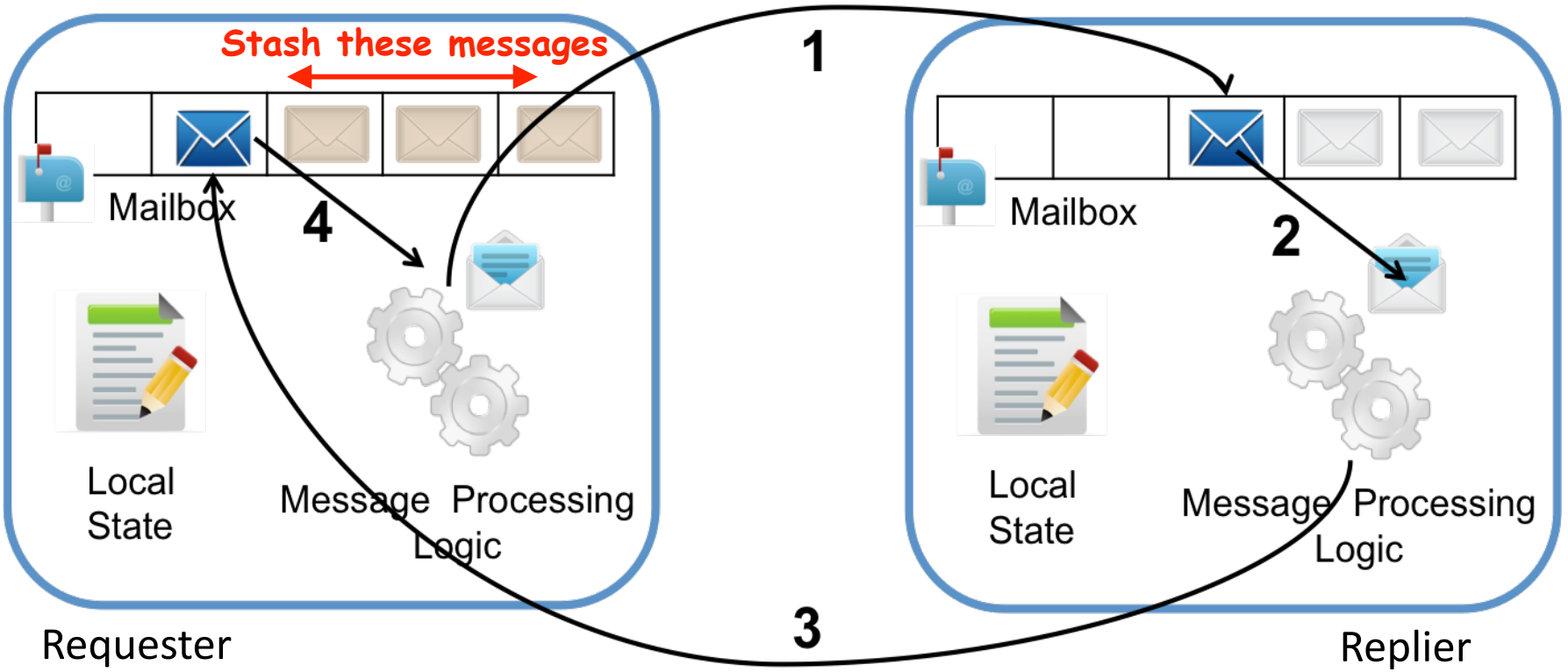


Source: S. Imam and V. Sarkar. ,Selectors : Actors with multiple guarded mailboxes, AGERE'14

- Extend actors with multiple 'guarded' mailboxes
- Actor can enable / disable mailboxes to implement different protocols
- Easy synchronization and co-ordination between multiple selectors e.g.,
  - Synchronous request-reply
  - Join Patterns
- Modularity and Data locality of Actor model preserved

Synchronous Request-Response Pattern

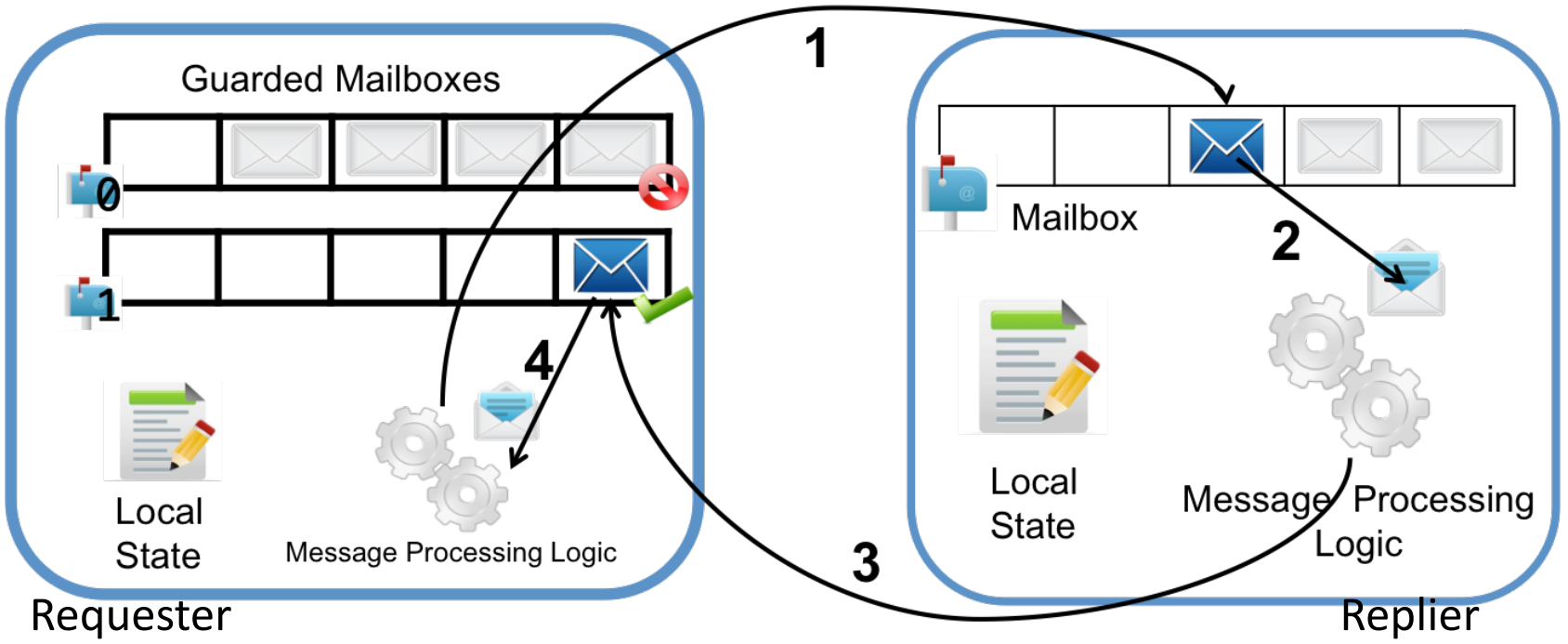
1. Requester sends a message to a replier system
2. Replier *eventually* receives and processes the request
3. Replier returns a message in response
4. Requester can only make further progress after receiving response (must "stash" all intervening messages



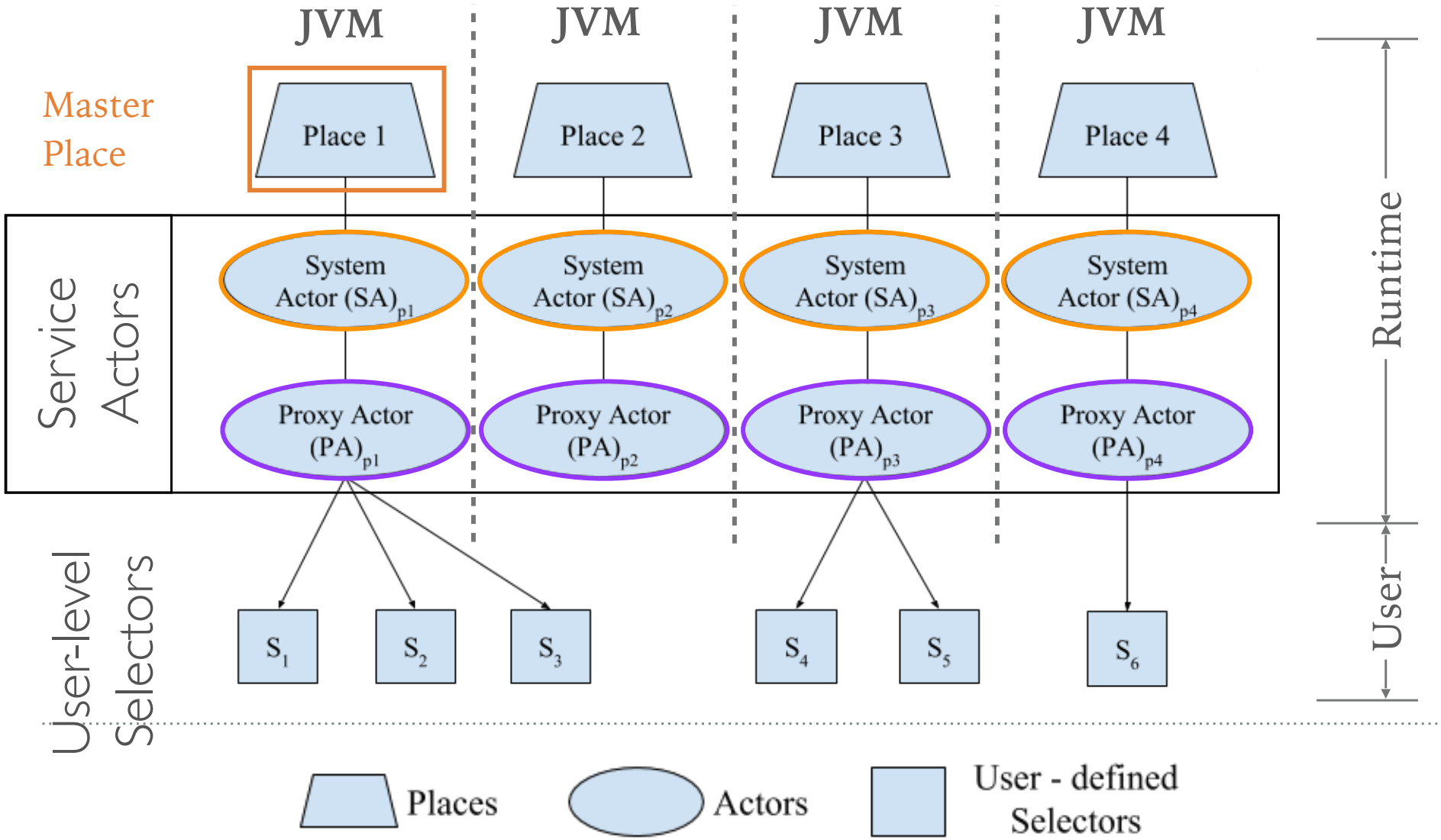
Synchronous Request/Response: Selector-based Solution

1. Two mailboxes
   1. one to receive regular messages
   2. one to receives synchronous response messages

- Whenever expecting a synchronous response
  - disables the regular mailbox
  - ensures next message processed is from reply mailbox



Distributed selector system



Source: A Distributed Selectors Runtime System for Java Applications. Arghya Chatterjee, Branko Gvoka, Bing Xue, Zoran Budimlic, Shams Imam, Vivek Sarkar. , PPPJ'16.

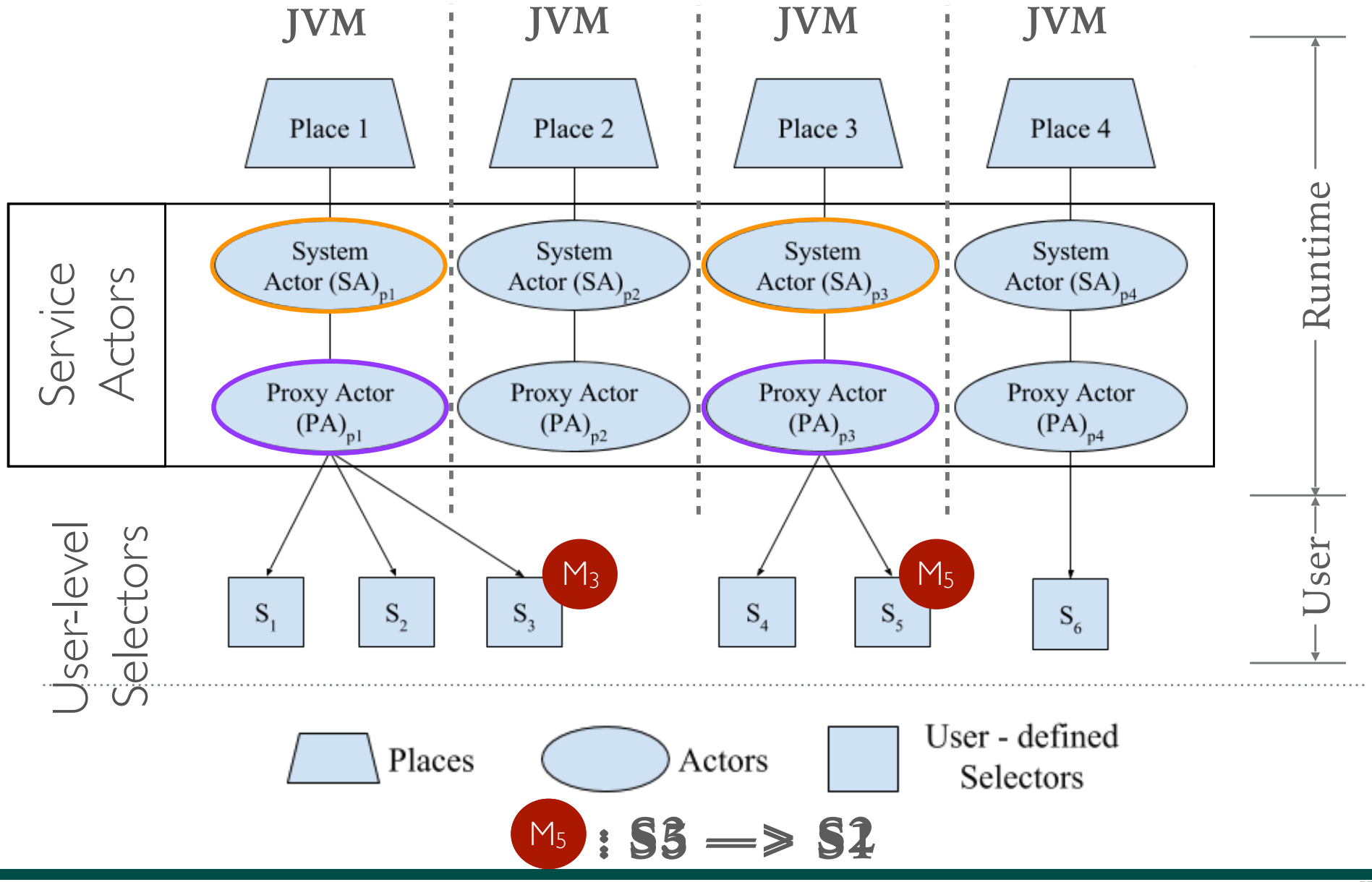
Automatic bootstrap and initialization

- No configuration file ➔ program runs in shared memory
- Configuration file ➔ program runs in distributed mode (without any changes in the source code!)
- Master place — startup confirmation from all remote places and executes user code
- Users can start selectors remotely

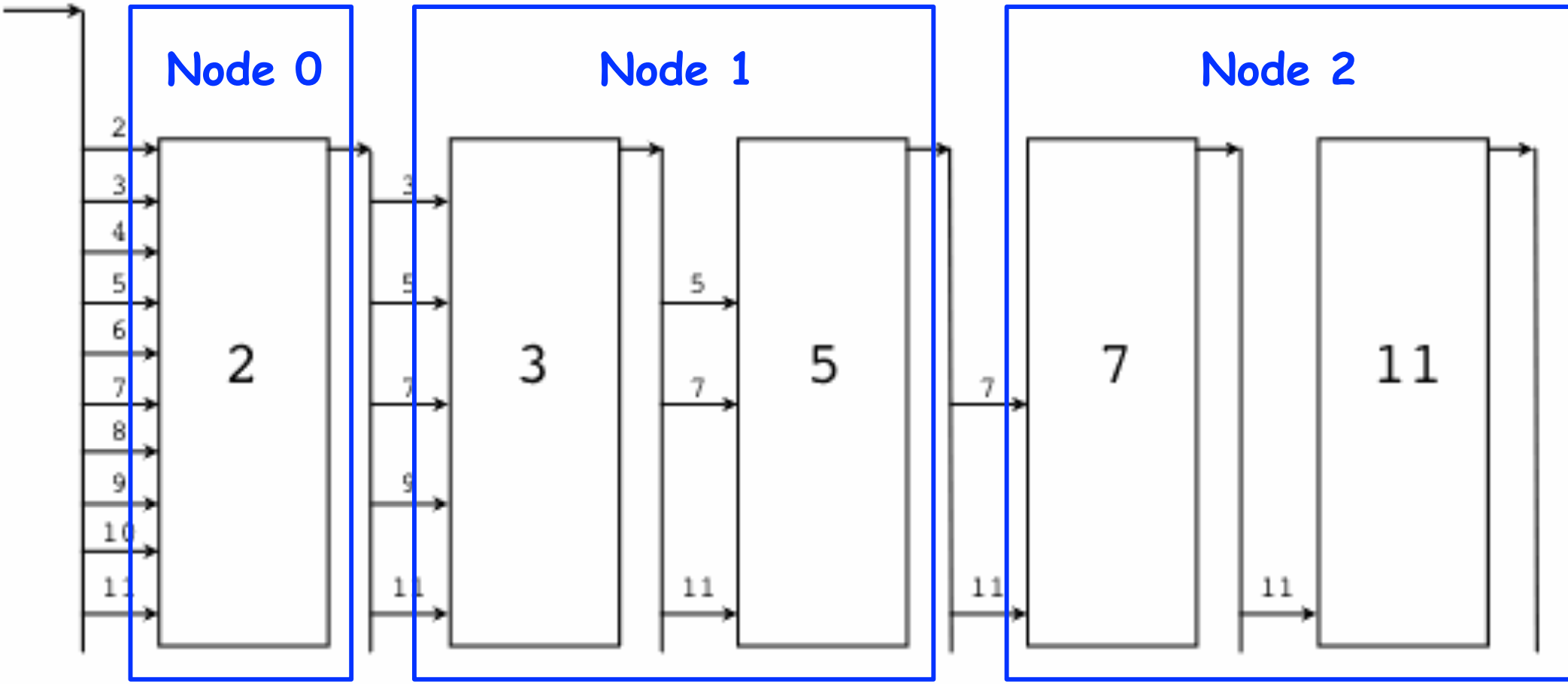
Configuration file

```
1  selectorSystem {
2      init {
3      place : p0,
4      hostname:cn16.davinci.rice.edu,
5      port: 5000,
6   }
7   remote : [
8     {
9        place : p1
10       hostname:cn20.davinci.rice.edu,
11       port: 5000,
12    },
13    {
14       place : p2
15       hostname:cn20.davinci.rice.edu,
16       port: 5002,
17    }
18   ]
19 }
```


Distributed selector system: Message routing

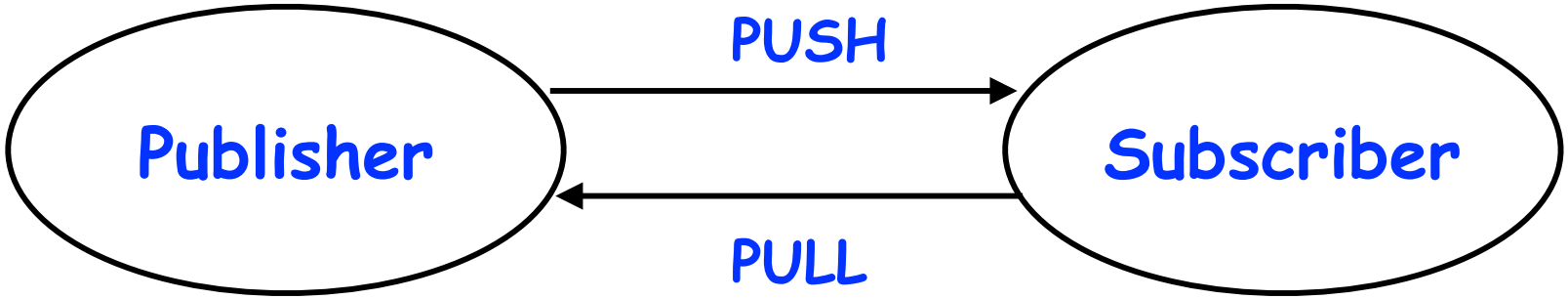


Example of Distributed Selectors: Sieve of Eratosthenes



Distributed Reactive Programming

- Reactive programming model supports both push (as in actors) and pull (as in streams) models



- Four interfaces available in Java
  - Flow.Publisher
  - Flow.Subscriber
  - FLow.Process
  - Flow.Subscription

Sample Publisher

```
// Source: https://community.oracle.com/docs/DOC-1006738

01.  import java.util.concurrent.SubmissionPublisher;
02.  ...
03.      //Create Publisher
04.      SubmissionPublisher<String> publisher =
             new SubmissionPublisher<>();
05.
06.      //Register Subscriber
07.      MySubscriber<String> subscriber = new MySubscriber<>();
08.      publisher.subscribe(subscriber);
09.
10.      //Publish items
11.      System.out.println("Publishing Items...");
12.      String[] items = {"1", "x", "2", "x", "3", "x"};
13.      Arrays.asList(items).stream().forEach(i ->
                                              publisher.submit(i));
14.      publisher.close();
```


Sample Subscriber

```
// Source: https://community.oracle.com/docs/DOC-1006738

01.  import java.util.concurrent.Flow.*;
02.  ...
03.
04.  public class MySubscriber<T> implements Subscriber<T> {
05.    private Subscription subscription;
06.
07.    @Override
08.    public void onSubscribe(Subscription subscription) {
09.      this.subscription = subscription;
10.      subscription.request(1); //a value of  Long.MAX_VALUE may be considered as effectively
unbounded   11.    }
12.
13.    @Override
14.    public void onNext(T item) {
15.      System.out.println("Got : " + item);
16.      subscription.request(1); //a value of  Long.MAX_VALUE may be considered as effectively
unbounded   17.    }
18.
19.    @Override
20.    public void onError(Throwable t) {
21.      t.printStackTrace();
22.    }
23.
24.    @Override
25.    public void onComplete() {
26.      System.out.println("Done");
27.    }
```