Synchronizers

- *Synchronizers*
  - *Objects that coordinate the control flow of threads based on the synchronizer's state*
  - Blocking queues act as synchronizers
    - They cause producers to block when the queue is full
    - They cause consumers to block when the queue is empty
  - There are other types of synchronizers
    - Locks
    - Latches
    - Semaphores
    - Barriers

Locks and Synchronizers

- `java.util.concurrent` provides generally useful implementations
  - **`ReentrantLock, ReentrantReadWriteLock`**
  - **`Semaphore, CountDownLatch, Barrier, Exchanger`**
  - Should meet the needs of most users in most situations
    - Some customization possible in some cases by subclassing
- Otherwise `AbstractQueuedSynchronizer` can be used to build custom locks and synchronizers
  - Within limitations: `int` state and FIFO queuing
- Otherwise build from scratch
  - Atomics
  - Queues
  - **`LockSupport`** for thread parking/unparking, similar to Semaphore

Example: Mutex

- **`Mutex`**: a non-reentrant mutual-exclusion lock
  - Only need to implement exclusive mode methods
- State semantics:
  - State == 0 means lock is free
  - State == 1 means lock is owned
  - Owner field identifies current owning thread
    - Only owner can release, or use associated `Condition`
- Class outline

```
class Mutex implements Lock {
   Thread owner = null;
   class Sync extends AbstractQueuedSynchronizer {
      // AQS method implementations ...
   }
   Sync sync = new Sync();
   // implement Lock methods in terms of sync ...
}
```

Latches

- Synchronizer objects that:
  - Block threads until a terminal condition is met
  - Subsequently release the blocked threads
  - Threads participate in synchronization by executing operations to wait on / modify latch state
- `CountdownLatch`
  - Latch based on counting
    - Terminal condition is that latch has value 0
    - Constructor accepts number to use as initial value
  - Methods
    - `await()`
      Block until latch has value 0
    - `countDown()`
      Decrement latch value by 1

Uses for Latches

- To delay starting of threads until an initial condition is satisfied
  - For example: timing a collection of threads
    - Don't want threads to start until all are created
    - In each thread, use a latch to wait for a "starting signal"
  - In this case, programming would consist of
    - Creation of latch with value 1
    - Creation, starting of threads
    - Decrement of latch using countDown(), which releases threads
- To do a "multi-way join" on thread termination
  - Idea: Initialize latch to number of threads
  - When each thread terminates, have it decrement latch
  - When latch is 0, all threads have terminated

FutureTask<T>

- A synchronization construct for starting computations now, getting the results later
  - A `FutureTask<T>` object is like a method call
    - It is invoked
    - It returns a value of type `T`
  - Unlike a method call, the invocation and return are separate events
    - A thread can start a `FutureTask` …
    - … do other work …
    - … then reconnect with the FutureTask when it needs the results
- The `FutureTask<T>` constructor requires an object matching the `Callable<T>` interface
  - `Callable<T>` like `Runnable`
  - Main method to implement is `public T call()` (as opposed to `void run ()`)
- A `FutureTask` must be embedded in a thread in order to be invoked
  - `Thread` class includes constructor taking a `FutureTask` object, which also implements `Runnable`
  - Starting this thread amounts to "invoking" the FutureTask
- To get result of `FutureTask` object `future`, execute `future.get()`
  - Thread executing this will block until call is complete
  - `future.get()` can throw several exceptions

Counting Semaphores

- Counting semaphores act like bounded counters
  - Initially, a positive value is given to semaphore
  - Operations can atomically decrement (`acquire()`) or increment (`release()`) this value
  - If the semaphore value is 0, then `acquire()` *blocks*

Counting Semaphores

- Why "`acquire()` / `release()`"?
  - Intuition: semaphores dispense "permits"

    Count reflect number of permits available
  - Acquisition of a permit reduces available permits by 1
  - Release increments number of permits by 1
    - Note: you can release even if you have not acquired!
    - So release really means: generate a new permit and add it into pool
  - The permit idea is only for intuition! There are no explicit permit objects

Counting Semaphores

- What are semaphores used for?
  - Resource allocation
    - You have n copies of a resource
    - You can use a semaphore to ensure that when more than n threads need the resource, some of them block
  - Size restrictions for data structures
    - Semaphore records maximum size
    - When you add an element, you need to acquire a permit first
    - When an element is deleted, you release a permit

Barriers

- A synchronizer for blocking a collection of threads until they all are at “the barrier point”
  - Threads wait at the barrier by invoking `barrier.await()`
  - When the number of threads indicated in the barrier object have arrived, all are released
  - Barriers can optionally have a `Runnable` object that is executed right before threads are released
- Uses: simulations
  - Simulations are often “step-by-step”
  - Computation at each step can be done in parallel using threads
  - Don’t want to start next step until current step is complete

Task Execution

Executors

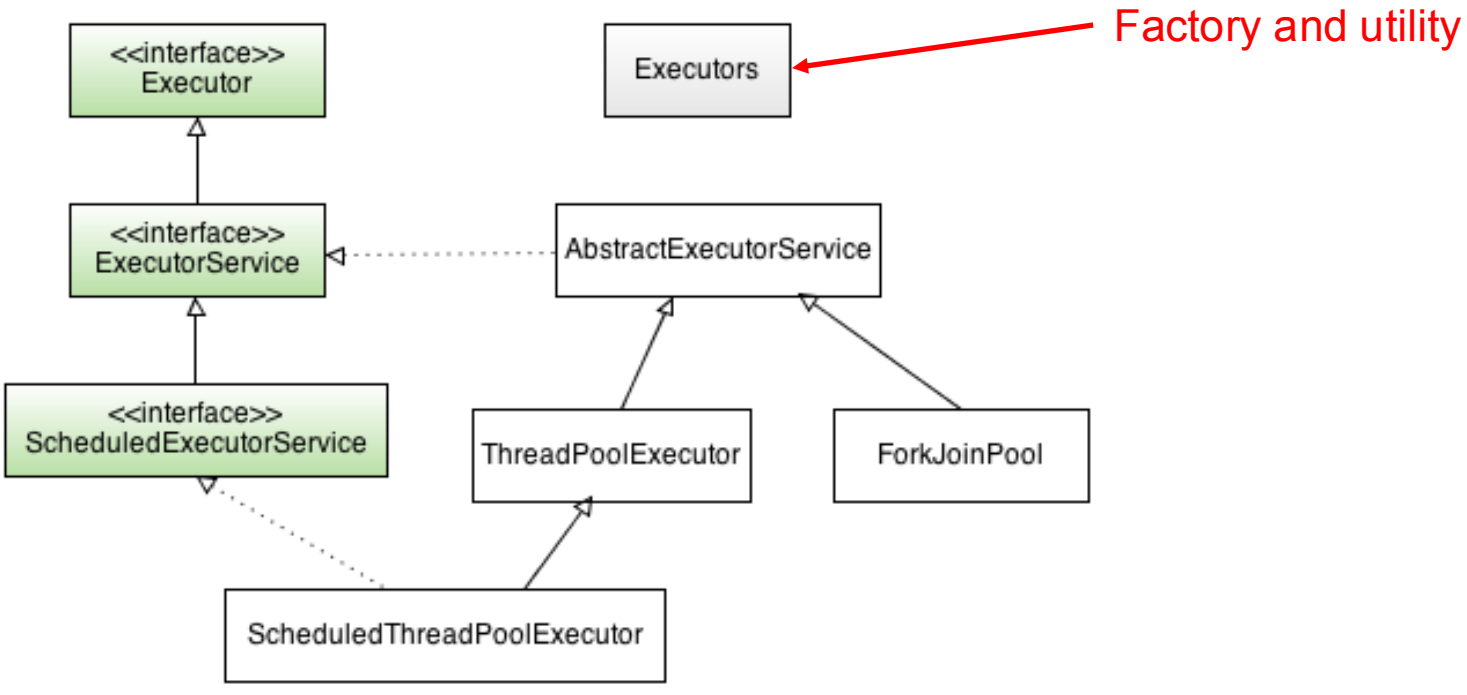
- A middle ground between sequential task processing and thread-per-task processing
- Decouples task submission from task execution

```
public interface Executor {
  void execute(Runnable command);
}
```

Executors

- Executors contain a *thread pool* of *worker threads*
- When a task comes in, and a thread is available, executor gives task to an idle thread
- If no thread is available, executor queues the result for future execution
- Based on producer / consumer pattern
  - Producers: generators of tasks
  - Consumers: threads that execute tasks

Implement Executor: Java 7



Execution Policies

- Executor implementation enables different *execution policies* to be defined
- An execution policy specifies how tasks get executed
  - Which thread?
  - What order (FIFO, LIFO, priority order)?
  - How many concurrent tasks?
  - How many tasks may be queued pending execution?
  - Overload policy? (Which task to kill, and how)
  - Pre- / post-task actions to perform, if any?
- Execution policies are a resource-management tool

  Permit management of concurrency vis a vis number of processors, other resources

Thread Pools

- Contains collection of homogeneous *worker threads*
- Is tightly bound to a work queue holding tasks to be executed
- Worker threads:
  - Request next task from work queue
  - Execute it
  - Return to waiting for next task
- Advantages (vs. creating new thread)
  - No need to wait for creation of new task
  - No overhead associated with task creation, elimination

Thread Pools

- **`Executors`** class contains factory methods for creating thread pools, e.g.
  - `newFixedThreadPool(int nThreads)`
  - `newCachedThreadPool()`
  - `newSingleThreadExecutor()`
  - `newScheduledThreadPool(int corePoolSize)`

`ExecutorService`?

- The factory methods in Executors return objects in `ExecutorService`
- **`ExecutorService`**
  - Is an interface extending `Executor`
  - The new methods include mechanisms for shutting down an executor
    - JVM cannot terminate until all non-daemon threads shut down
    - Worker threads are non-daemon threads
    - Shutting down an executor requires shutting down these threads and dealing with any queued tasks

What About Tasks Submitted After Shutdown?

They are handled by the *rejected execution handler*

- Could just swallow the tasks
- Could throw **`RejectedExecutionException`**
- Depends on implementation!

- More details in book

ScheduledExecutorService

- An ExecutorService that can schedule commands to run after a given delay, or to execute periodically.

- scheduleAtFixedRate and scheduleWithFixedDelay methods create and execute tasks that run periodically until cancelled.

`CompletionService`

- Extends **`ExecutorService`** with a *blocking completion queue*
  - When a task that has been submitted finishes, a **`Future`** for it is put in completion queue
  - A user of the completion service can extract next finished computation by performing **`take()`** on completion service
- This permits processing of task results in order that they were completed

Designing Thread Pools

- Considerations
  - How big?
  - What execution policy?
- Decisions about these considerations are influenced by several factors
  - Task dependencies
    - Some tasks are independent
    - Some require results of other tasks
    - Some tasks will even spawn other tasks whose results they need
  - Task thread-confinement assumptions
    - Some tasks assume thread-confinement
      - Legacy single-threaded code
      - Efficiency
    - Such tasks should run in a single-threaded thread pool
  - Variability in task execution times, responsiveness requirements
    - Some tasks may run much longer than others
    - Other tasks may need quick turnarounds
  - Tasks that assume thread-specific knowledge
    - Some tasks may make assumptions about the specific thread on which they are running (e.g. if there is a `ThreadLocal` variable)
    - Such tasks must be handled carefully in thread-pool setting

Thread Starvation Deadlock

- An issue affecting pool sizing
- Suppose you have a fixed-size pool (say, 10)
  - Suppose 10 tasks are running, so no free threads
  - Suppose further that each of these tasks submits a task to the pool and then blocks awaiting the result
- Deadlock!
  - Each of 10 task-threads is blocking
  - There are no threads to handle new tasks on which they are blocking
  - No thread can make progress

Sizing Thread Pools

- Want to avoid thread pools that are “too big” or “too small”
  - Too big: contention among threads for memory, other resources
  - Too small: bad throughput
- We have already seen one consideration for sizing thread pools: thread-deadlock starvation
- Other considerations
  - Are tasks compute or I/O intensive?
  - How many processors on system?
  - How much memory do tasks need?
  - What other possibly scarce resources (e.g. JDBC connections) are needed?
- Note
  - Sometimes you have different classes of tasks that must be run, with different profiles
  - You can use multiple thread pools and tune each independently!

Thread-Pool Execution Policies

- Executors include thread-pool execution policy
- Executors returned by **`Executors.newXXXThreadPool()`**, etc. include built-in execution policies
- These methods all use a base implementation given in class `ThreadPoolExecutor`
  - To customize execution policy, you can call the ThreadPoolExecutor constructor yourself
  - The parameters to the constructor allow you to modify the execution policy in a variety of ways

Using ThreadPoolExecutor

- General constructor for this class has following form

```
ThreadPoolExecutor (
        int corePoolSize,
        int maximumPoolSize,
        long keepAliveTime,
        TimeUnit unit,
        BlockingQueue<Runnable> workQueue,
        ThreadFactory threadFactory,
        RejectedExecutionHandler handler )
```

Saturation Policies (cont.)

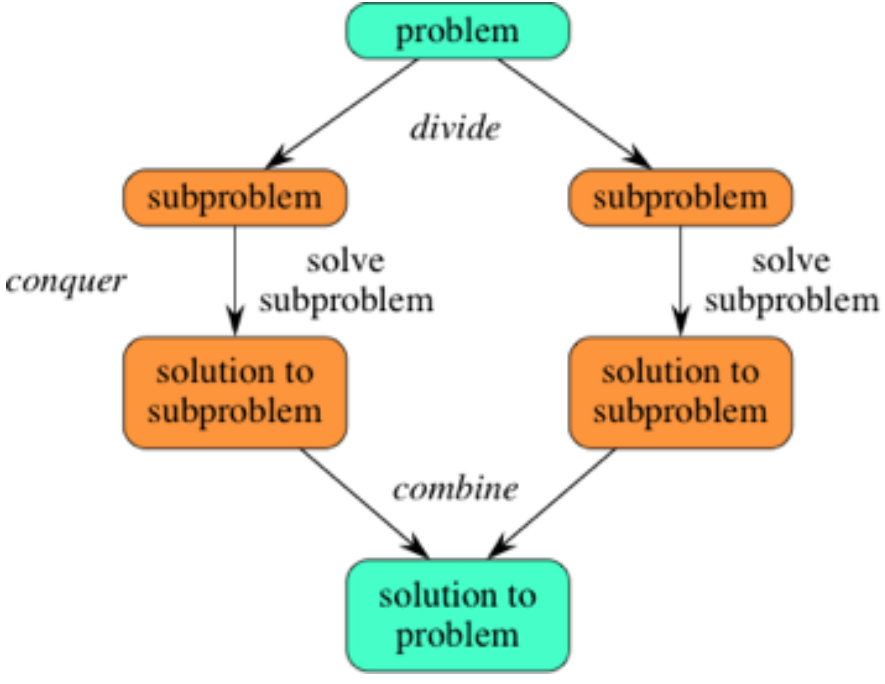
- `ThreadPoolExecutor` implements several saturation policies as (static) classes matching `RejectedExecutionHandler` interface
  - **`AbortPolicy`** (this is the default)
    `execute()` throws RejectedExecutionException if queue is full
  - **`DiscardPolicy`**
    `execute()` silently discards newest task
  - **`DiscardOldestPolicy`**
    - `execute()` discards task at head of work queue (i.e. next one up for execution) and tries to resubmit current task
    - Beware if work queue is a priority queue!
  - **`CallerRunsPolicy`**
    - `execute()` runs the task in the thread calling execute()
    - This helps give worker threads time to catch up, since new invocations of execute will be blocked from that thread!

Fork/Join Parallelism

Divide and Conquer

- Quicksort, Mergesort are examples of *divide-and-conquer* algorithms
  - Basic structure of divide-and-conquer algorithms:
    1. If problem is small enough, solve it directly
    2. Otherwise
       a. Break problem into subproblems
       b. Solve subproblems recursively
       c. Assemble solutions of subproblems into over-all solution
  - If algorithm is tail-recursive, step 2.c. is not necessary
- Other examples
  - Depth-first search
  - Binary search
  - Euclid's algorithm

Divide and Conquer

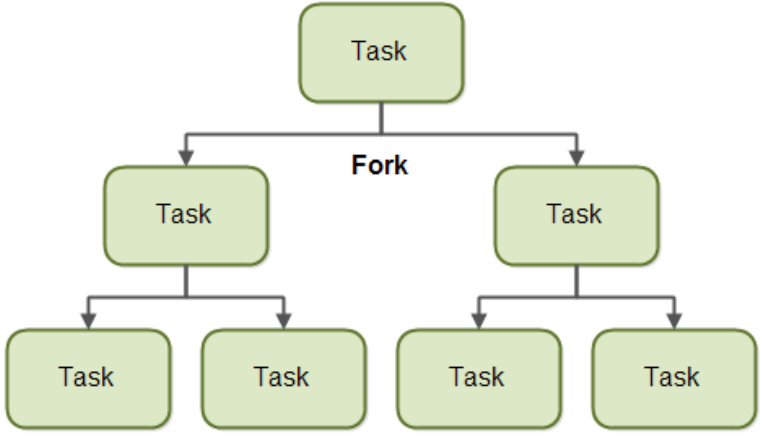


Parallelizing Divide-and-Conquer Algorithms

- The basic strategy: turn recursive calls into tasks
  - Solve the small instances directly
  - For larger instances requiring recursive calls, create tasks for each recursive call
- Performance tuning
  - Use a larger threshold than that specified in base case of original algorithm to switch to sequential solving
  - Threshold should take account of original problem size, number of CPUs

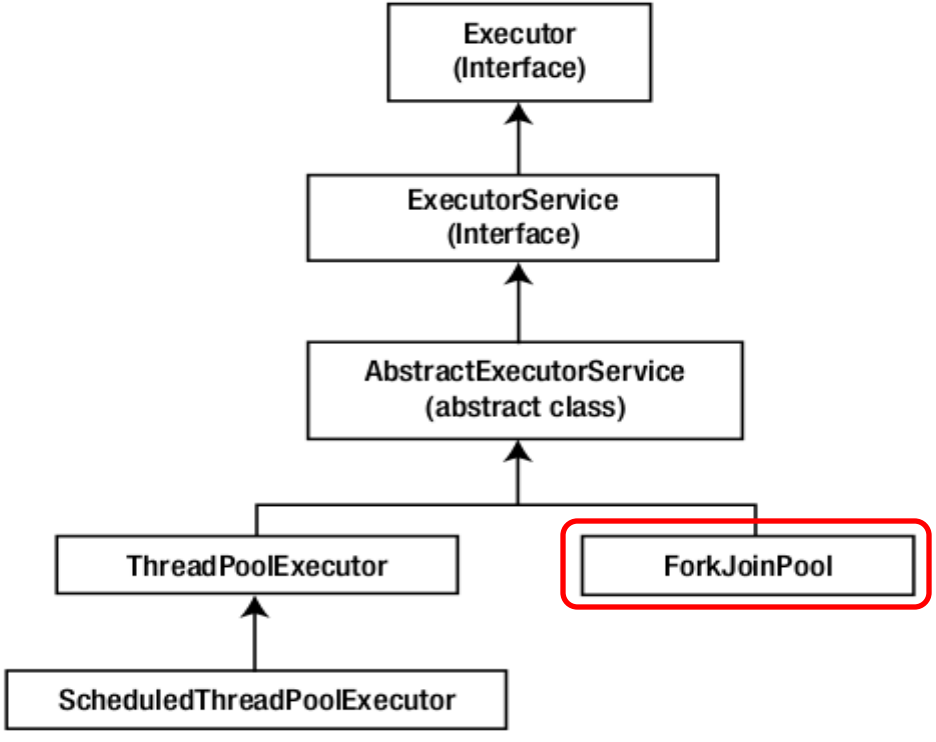
Fork/Join Parallelism

- Parallelizing divide-and-conquer algorithms is frequent enough that Java has specialized support: *Fork/Join parallelism*
- Basic idea: exploit specialized structure of divide-and-conquer dependencies to improve *parallelism* (i.e. execution time)



ForkJoinPool

The *ForkJoinPool* is is an implementation of the ExecutorService.



Fork/Join Parallelism

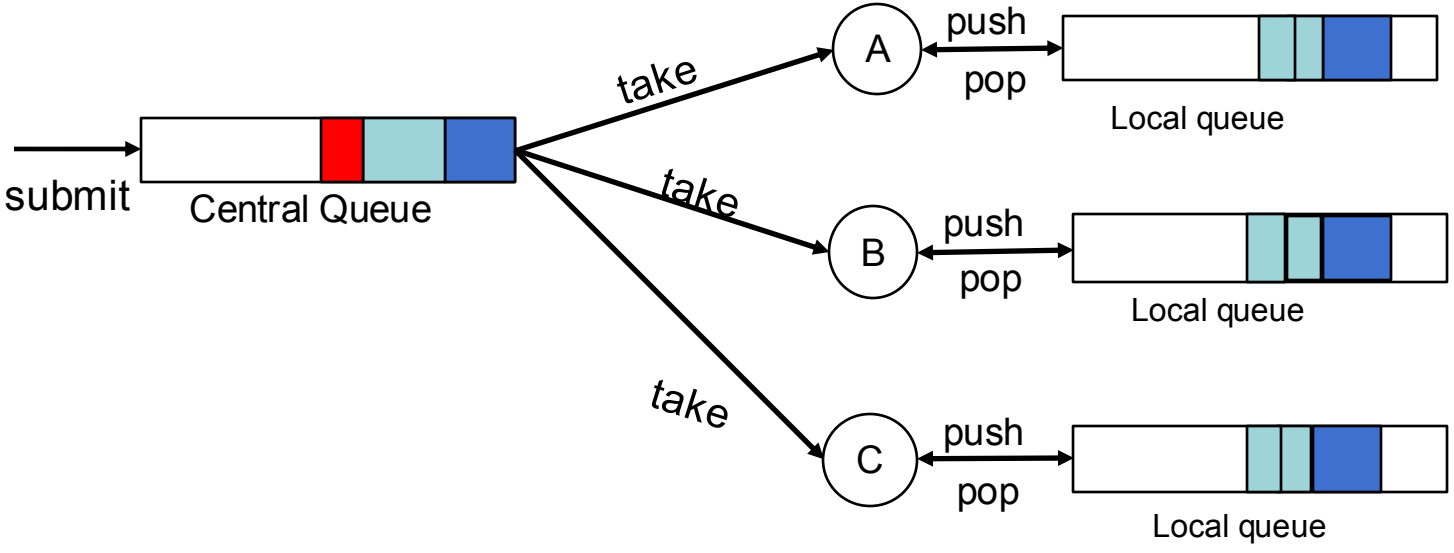
- Components of Fork/Join framework
  - Specialized executor class: **`ForkJoinPool`**
    - Implements **`ExecutorService`** interface
    - Uses specialized thread-pool management, work distribution strategies tuned for divide and conquer
  - Specialized task class: **`ForkJoinTask<V>`**
    - Implements **`Future<V>`** interface
    - Has numerous specialized operations
    - Lighter weight than a Java thread. A large number of ForkJoinTasks can run in a small number of worker threads in a fork-join pool
    - Two important subclasses
      - RecursiveTask: like Callable in that value is returned
      - RecursiveAction: like Runnable in that no value is returned

ForkJoinPool

- The executor for fork-join tasks
  - Maintains thread pool
  - Allocates work among worker threads
- Key attributes
  - Limits number of workers to number of CPUs (default) or user-specified number
  - Workers that are waiting for subtasks to complete are put to work on other subtasks
  - *Work-stealing* used to keep workers busy
    - Each worker has its own work queue (actually, a work *deque*)
    - When a workers deque is empty, it takes work from another workers deque

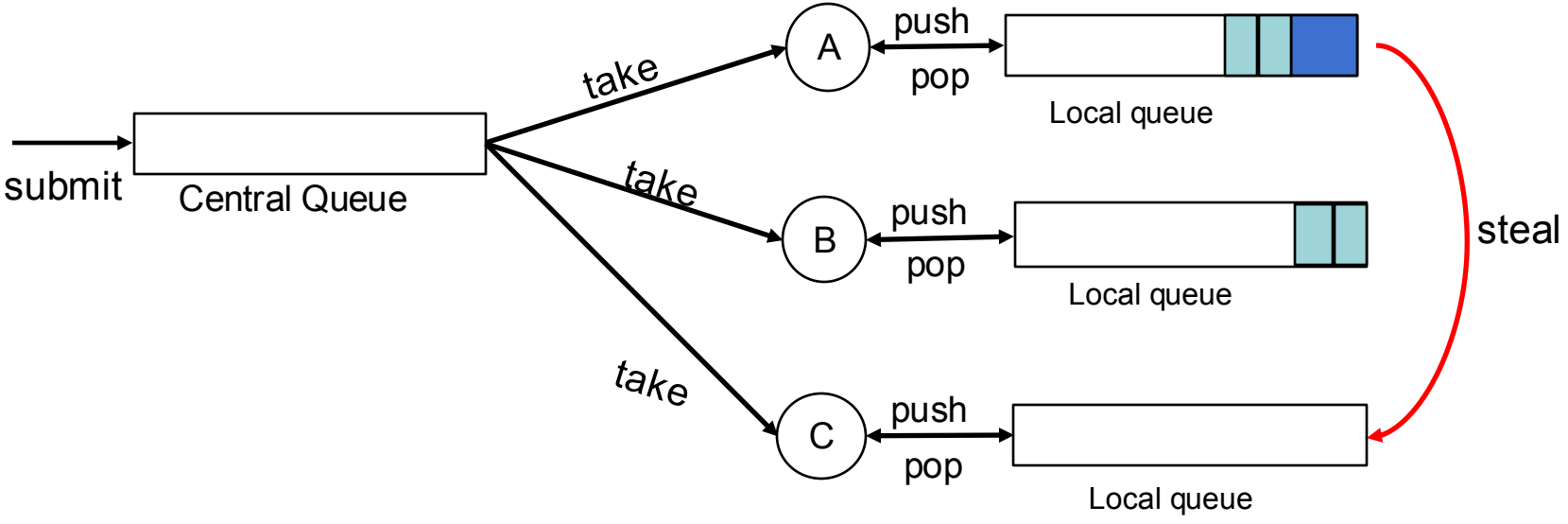
The Fork-Join Framework Structure

A ForkJoinPool with three worker threads, A, B, and C.



- The ForkJoinPool uses a central inbound queue and an internal thread pool.
- Each worker thread has its own task queue, to which it can add new tasks

Work Stealing



- Idle worker C steals a task from the tail of A's queue
  - Tail has the larger tasks
  - A takes from head, C steaks from tail. They do not contend for lock

Deque? Work Stealing?

- allows adding / removing elements from both ends
- Each worker thread has a deque containing tasks to work on
  - When new tasks are forked, they are “pushed” onto the front of the deque (i.e. opposite of what you do with a queue)
  - When a worker finishes a task, or blocks on the current one, it “pops” the next task from the front of its deque
- When a worker’s deque is empty it tries to steal a task from the *back* of one of the other workers’ deques
  - If it is successful it works on this task, using its own deque to push / pop subtasks
  - `Future`-like feature of `join()` ensures results of “stolen” tasks are available to original task owner

Quiz 1:

Worker 1 queue

Worker 2 is idle and wants to steal a task from worker 1. Worker 2 will steal the task

A. a
B. b
C. c
D. d

ForkJoinTask<V>

- Tasks that are managed by ForkJoinPool
- Besides usual Future methods (e.g. get()), other key methods are:
  - `ForkJoinTask<V> fork()`
    Arranges to asynchronously execute this task
  - `V join()`
    Returns the result of the computation when it is done.
  - `V invoke()`
    Commences performing this task, awaits its completion if necessary, and returns its result, or throws an (unchecked) RuntimeException or Error if the underlying computation did so.
  - `static ForkJoinPool getPool()`
    Returns the pool hosting the current task execution, or null if this task is executing outside of any ForkJoinPool
- `getPool()`?
  - ForkJoinTasks contain internal reference to the ForkJoinPool they belong to
  - When a ForkJoinTask forks another task, the new task inherits the ForkJoinPool from the caller

More on `fork()`, `join()`

- **`fork()`** has effect of submitting task to ForkJoinPool
  - Task is placed in deque of "parent task" (i.e. one that performed `fork()`)
  - Task performing `fork()` keeps executing
- **`join()`** has effect like `get()` in Future<V>
  - Task performing `join()` waits until result of subtask is available
  - While it is waiting it may start work on other tasks in its deque or engage in work-stealing
  - Note: unlike `get()`, `join()` is not a "blocking operation" in the standard sense: no InterruptedException can be thrown!
  - Using `join()` also forestalls thread-starvation deadlock
    - Although number of worker threads is fixed …
    - … **`join()` doesn't block**

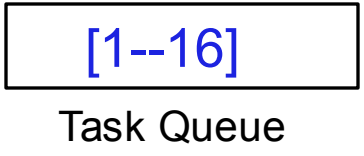
Structure of a Fork/Join Application

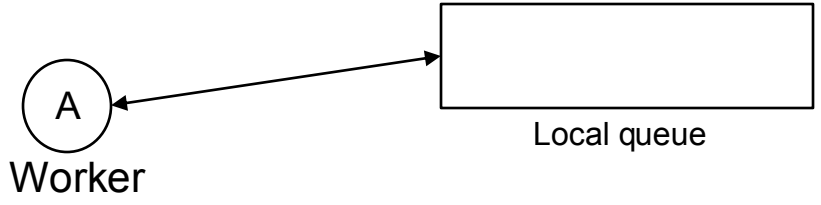
- Define class of ForkJoinTasks

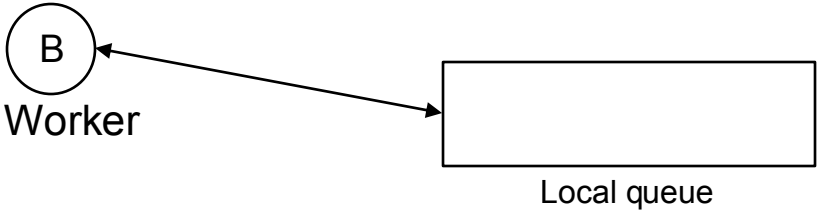
  ForkJoinTasks create subtasks, call fork, join, etc.
- Client application (i.e. one calling Fork/Join application) does this:
  - Create ForkJoinPool
  - Create task for entire problem to be solved
  - Call `execute() / submit() / invoke()` method of ForkJoinPool with this task
- Note that ForkJoinTasks do not usually call invoke method of ForkJoinPool!

Code Example:

Task “Sum 1 to 16” arrives

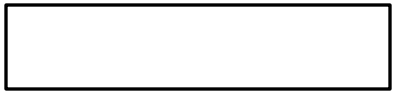


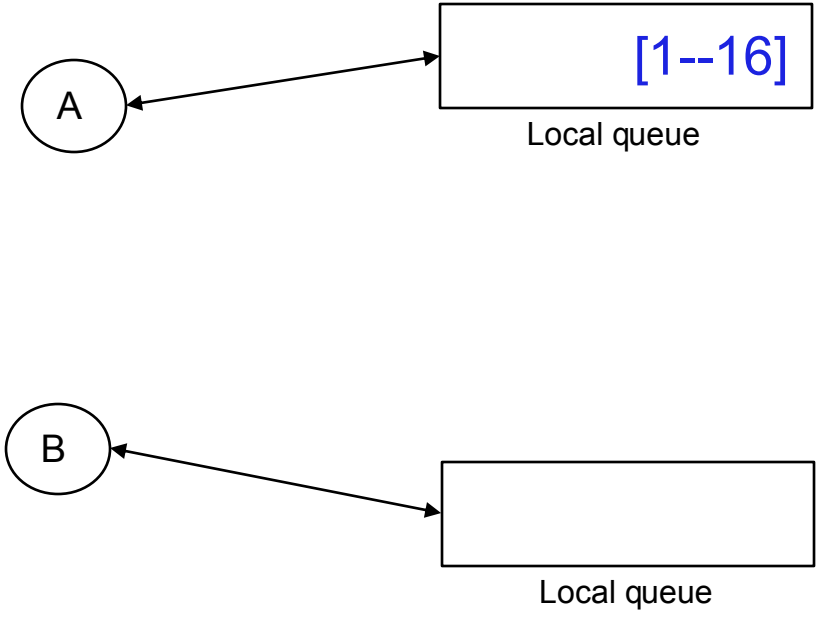




Code Example:

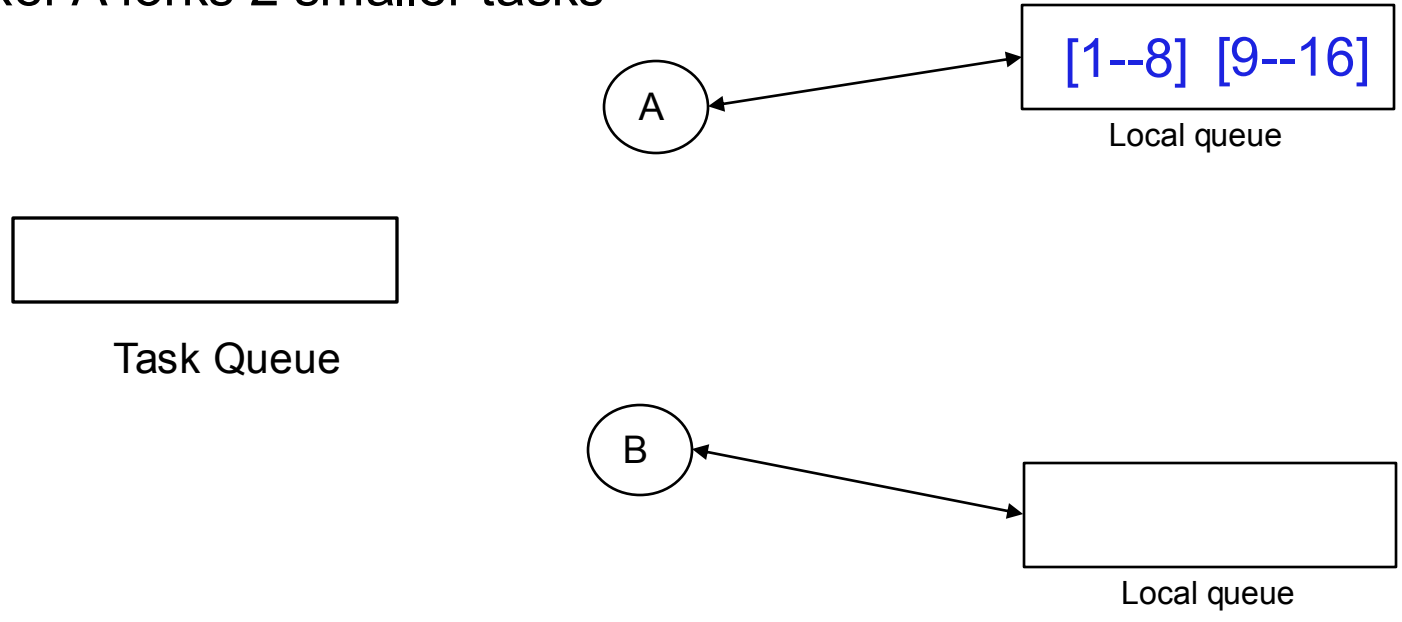
Worker “A” takes the task





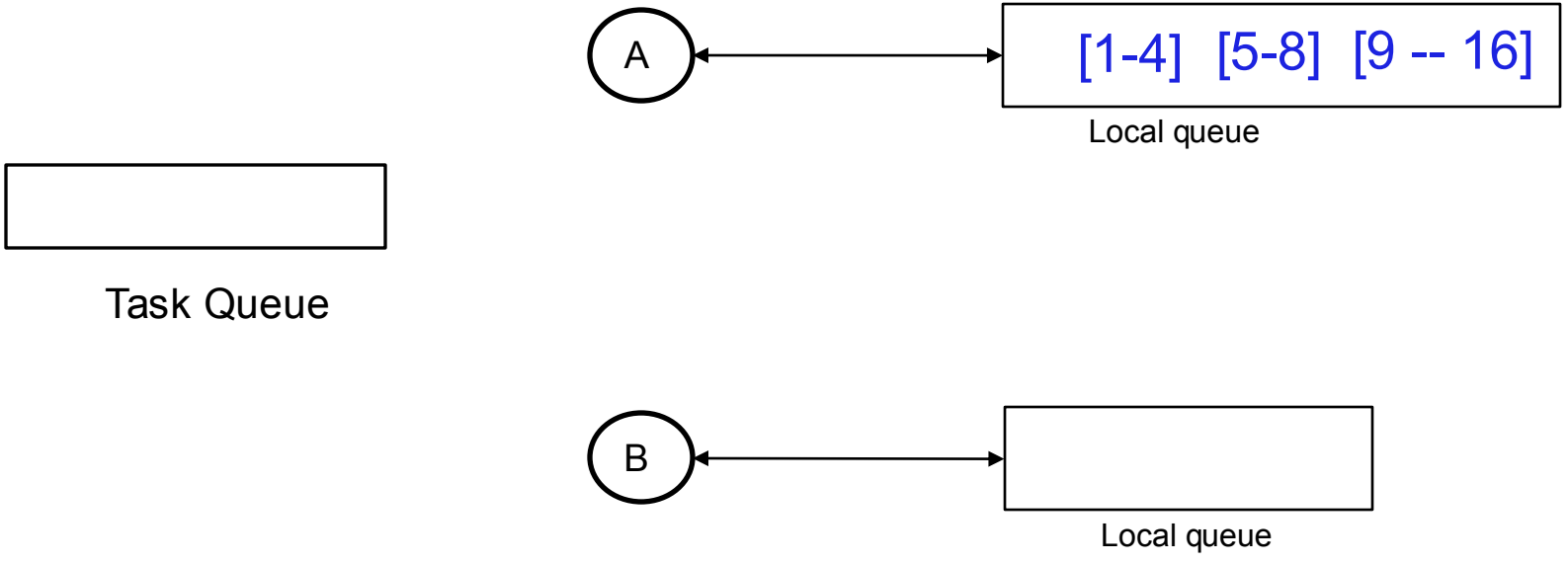
Code Example:

Worker A forks 2 smaller tasks



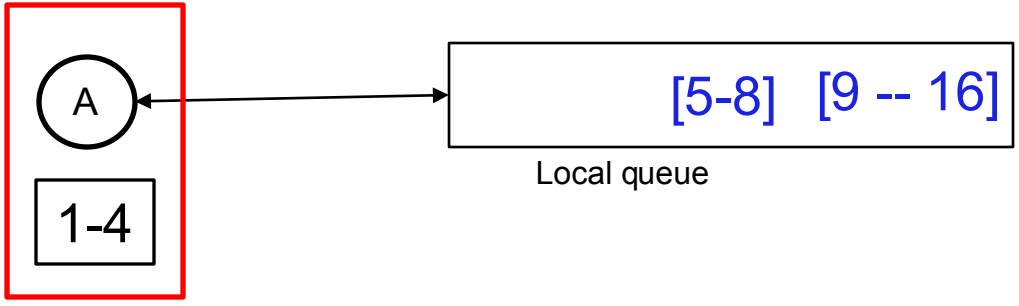
Code Example:

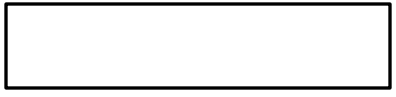
A takes the task 1-8, and forks 2 smaller tasks



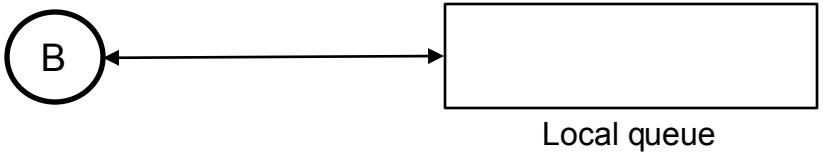
Code Example:

A computes the task 1-4



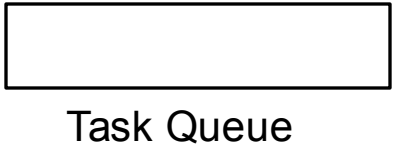


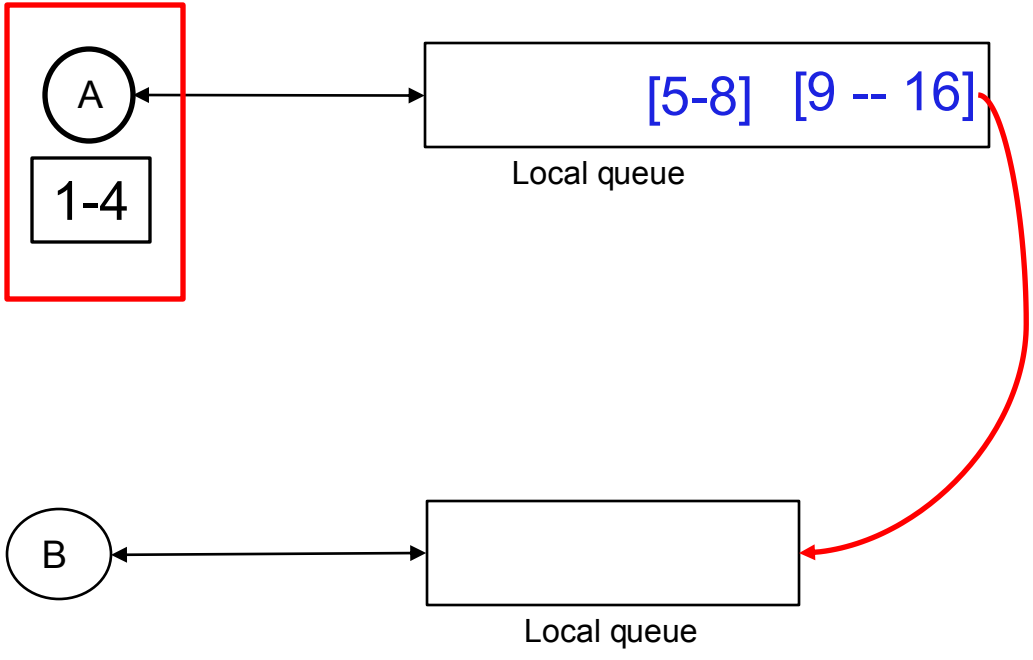
Task Queue



Code Example:

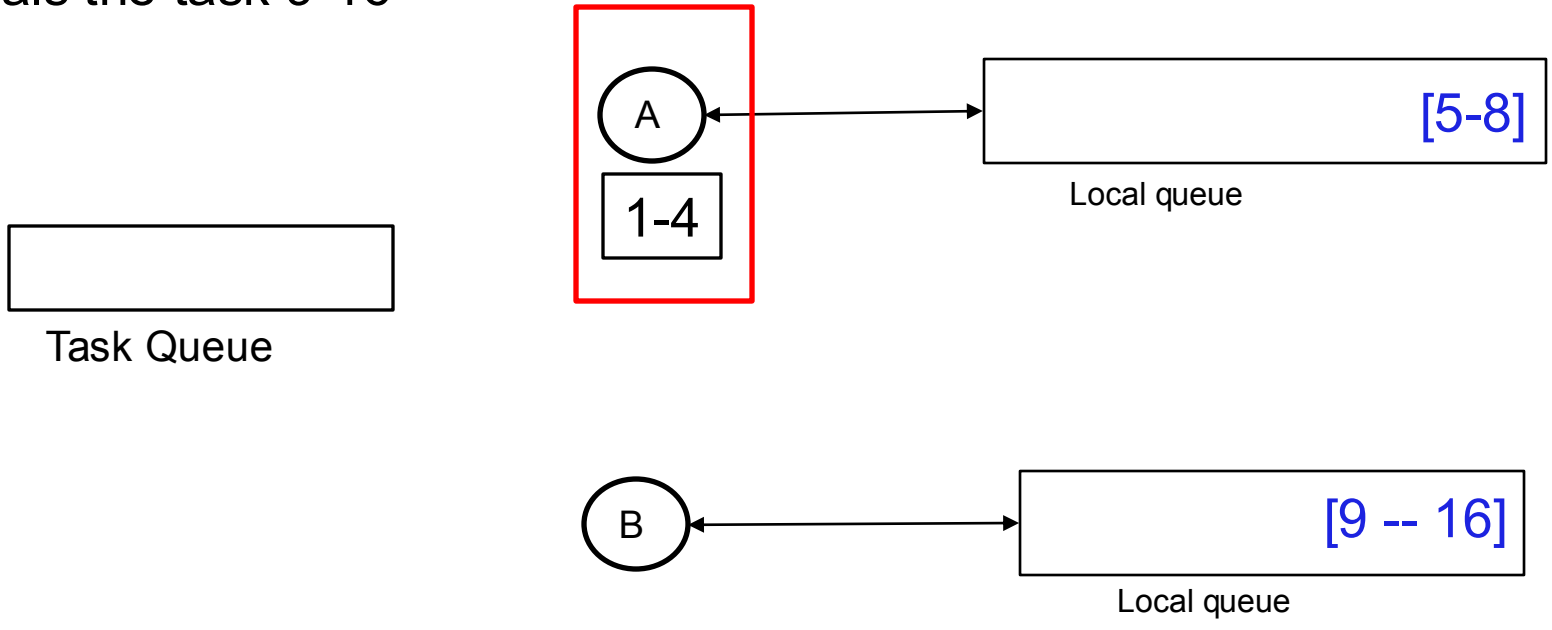
B steals the task 9-16





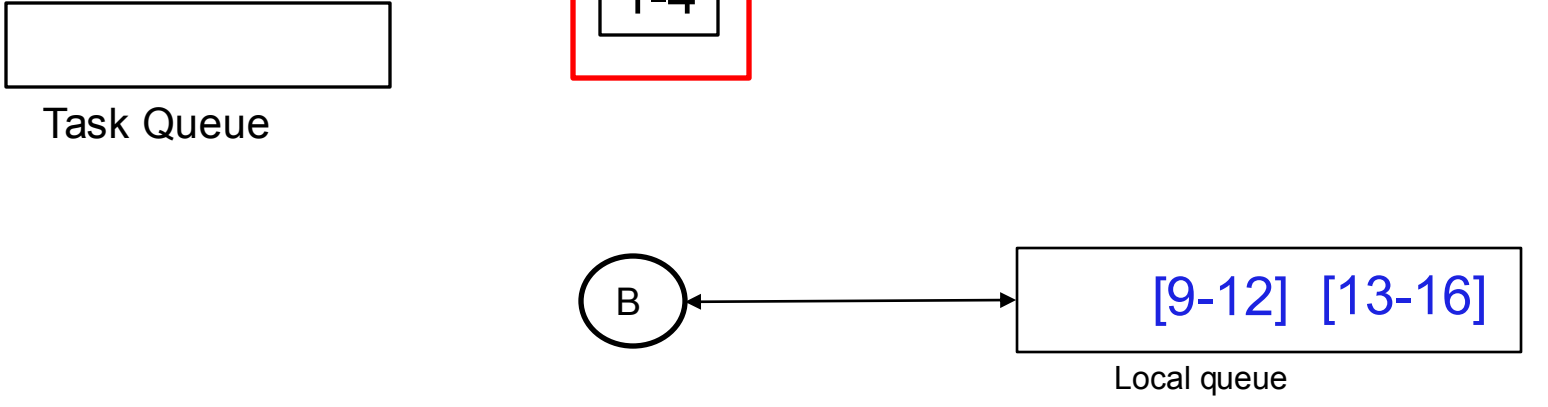
Code Example:

B steals the task 9-16



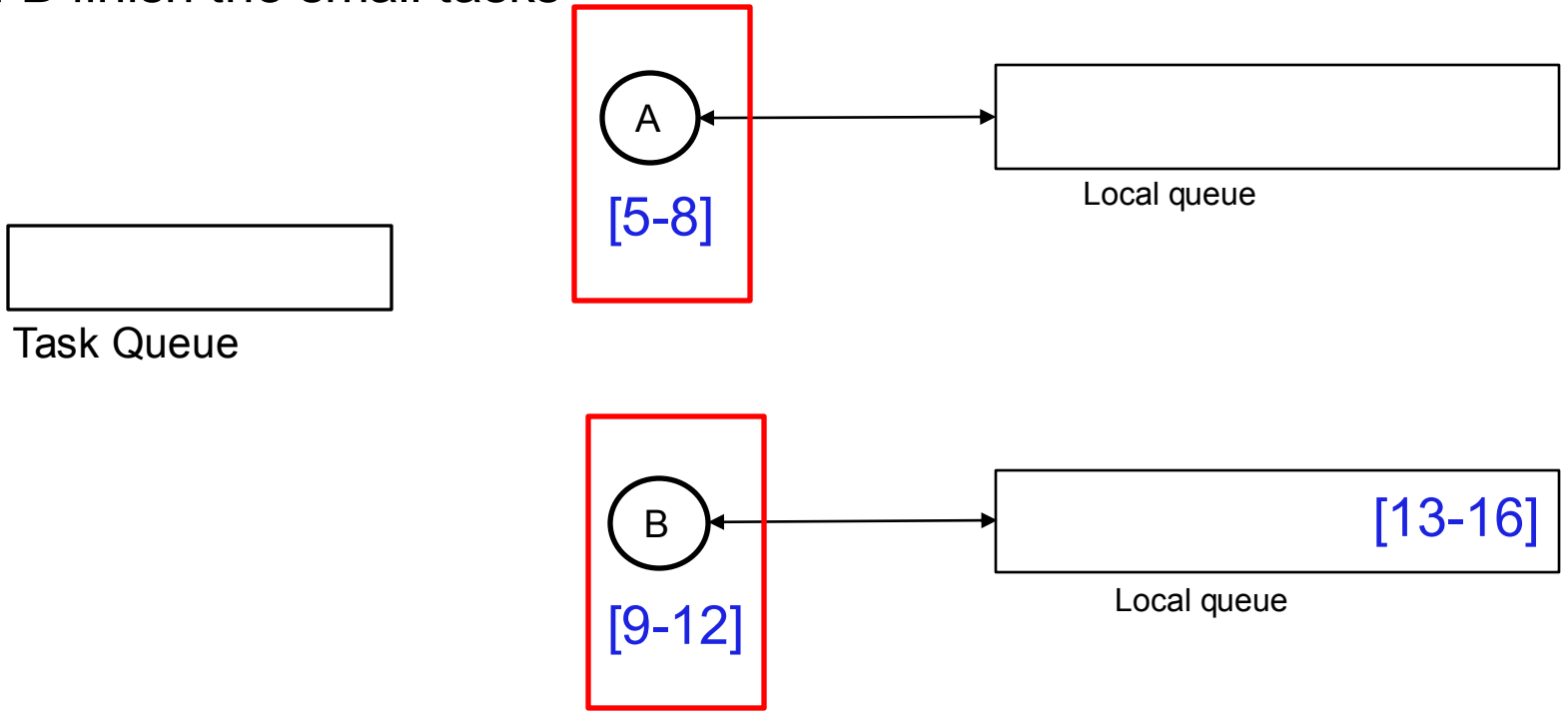
Code Example:

B takes 9-16, and forks 2 smaller tasks



Code Example:

A and B finish the small tasks



Performance Tuning of Fork/Join Applications

- ForkJoinPools automatically manage number of threads to try to maximize parallelism
- Application must manage task-creation overhead
  - Use thresholds, just like with other executor-based parallelizing approaches
  - Thresholds determine when to create new tasks vs. using sequential solutions
  - Because Fork/Join uses lock-free implementations of deques, it is more forgiving about task boundaries

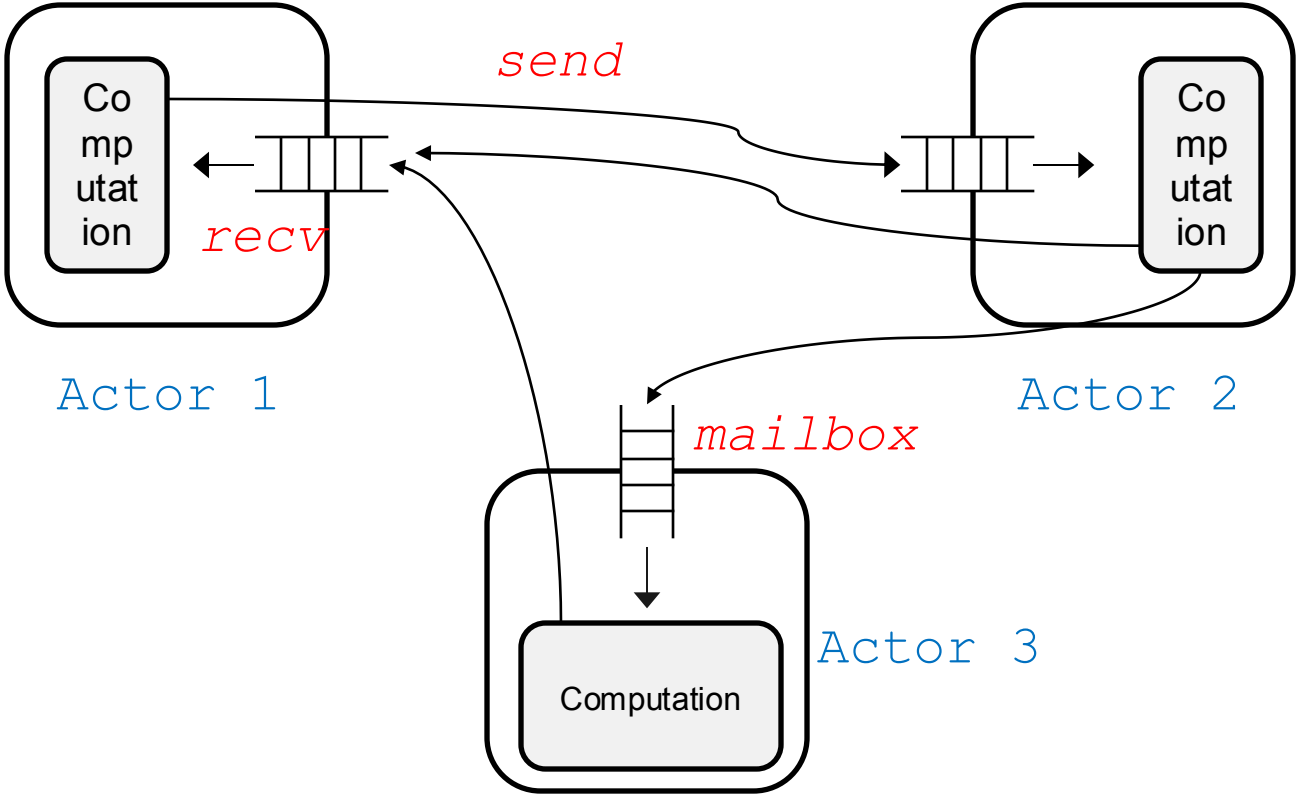
When To Use Fork/Join?

- Fork/Join in Java tuned for maximizing parallelism
  - Idea is to give solutions to big problems fast
  - Algorithms should be in a divide-and-conquer style
- Fork/Join in Java also handle dependencies between tasks well
  - `join()` does not block!
  - If `join()` is only means for inducing task dependencies, then thread-starvation deadlock cannot happen
  - This also simplifies termination detection, since there is no penalty for waiting for other tasks to finish

The Actors Model

- A system model supporting a multi-process programming paradigm
  - Model assumes no shared memory
  - No assumptions about distributed / non-distributed
- Systems consist of multiple *actors*
  - An actor is an independent sequential (“= single-threaded”) computation
  - Each actor has a “mailbox” from which it extracts messages that it then processes
  - Actors communicate by sending each other messages

An Actor System



General Actor Behavior

- How actors execute
  - They wait until there is a message in their mailbox
  - They remove message from mailbox and process it
  - Processing may involve sending of messages to other actors
  - When processing is complete, they retrieve next message from mailbox and repeat
- This style of programming / system is sometimes called *reactive*
  - Actors compute by reacting to messages that have arrived
  - Otherwise, they are idle

Message Passing

- Recall: actors communicate via message passing
- Different actor frameworks provide different guarantees about message delivery.
- Here are the ones we will use (conform to akka)
  - *Asynchronous*: senders do not know when messages are received
  - *At-most-once delivery*: every message sent is eventually received at most once (could be lost, but not duplicated)
  - *Locally FIFO*: messages sent by one actor directly to another are received in the order sent, lost messages excepted

Actor History

- Originally proposed by Carl Hewitt in 1970s as basic model of distributed computing
- Theory studied in 1980s / early 1990s by researchers
- Mid-1990s: first serious language implementation (*Erlang*, Ericsson)
  - Used in implementation of telephone switches
  - Key features: light-weight (more like tasks than threads), high degree of concurrency, resiliency in face of failure
- Mid-2000s: Scala language (for JVM!) includes actors
- Late 2000s: akka open-source library for Scala, Java

akka Java Library

- Provides implementation of actor model for Java
- Key features
  - Basic actor framework
    - Special actor objects
    - Communication via message-passing methods
  - Lightweight
    - Actors resemble tasks more than threads
    - ~300 bytes of overhead per actor
  - Location transparency
    - Actors programmed identically, whether local or on remote host
    - Differences captured in configuration file
  - Fault tolerance via hierarchy
    - Actors arranged in parent/child hierarchy
    - Parents handle failures of children

akka Documentation

- General: http://doc.akka.io/
  - General overview of actor model as realized in akka
  - There are also links for the full documentation of Java version of akka
  - The API / language reference documentation is specific for Version 2.5.x
- For Version 2.5 https://doc.akka.io/docs/akka/2.5.3/java/index.html
  - You'll get a splash screen saying "upgrade"
  - Ignore this!

Basics of akka Java

- akka actors live in an *actor system*
  - Actor system provides actor execution (think “threads”), message-passing infrastructure
  - To create actors, you must first create an actor system
  - The relevant Java class: `ActorSystem`
- So, first line of Hello World `main()` method is:

```
ActorSystem actorSystem =
ActorSystem.create("Message_Printer");
```

  - “Message_Printer” is name of actor system (required)
  - akka actor system names must not have spaces or punctuation other than - or _ !

Creating Actors in akka Java 2.5 (1/4)

- Actors are objects (of course!)
- Objects are typically in a subclass of the akka library class `AbstractActor`
- **Step 1 in creating actors:** define class of actors
  - Here is the relevant import / class declaration

```
import akka.actor.AbstractActor;
public class MessagePrinterActor extends AbstractActor …
```

Creating Actors in akka Java 2.5 (2/4)

- **Step 2 in creating actors:** finish implementation of actor class
  - akka `AbstractActor` needs instance method: `public Receive createReceive()`
  - This method describes how a message object should be processed
- Hello World example

```
@Override
public Receive createReceive() {
    return receiveBuilder()
        .match(Double.class, d -> {
            sender().tell(d.isNaN() ? 0 : d, self()); }
        )
        .match(Integer.class, i -> {
            sender().tell(i * 10, self()); }
        )
```

Creating Actors in akka Java 2.5 (3/4)

- In akka, actors can only be created in the context of an `ActorSystem`
  - Relevant `ActorSystem` method : **`ActorRef actorOf(Props p, String name);`**
  - Return type **`ActorRef`** is class of “references to actors” (more later on this notion)
  - `String` parameter is actor name (no spaces, non-alphanumeric characters other than -,_!)
  - “`Props`”?

Creating Actors in akka Java 2.5 (3/4)

- In akka, actors have various configuration information
  - Type of mailbox data structure
  - How messages actually get delivered to mailbox (“dispatching”)
  - Etc.
- This information is encapsulated in a `Props` object for a given class of actors
- To create actors in a class, a `Props` object for the class must be constructed

Creating Actors in akka Java 2.5 (3/4)

- **Step 3 in creating actors:** create `Props` object for actors class.
  - This is done in the Hello World `main()` using factory method `create()` in akka `Props` class
  - `create()` builds `Props` object with reasonable defaults (unbounded queues for mailboxes, etc.)
  - Relevant Hello World code:

```
Props mpProps = Props.create(MessagePrinterActor.class);
```

Creating Actors in akka Java 2.5 (4/4)

- **Step 4 in creating actors:** call `actorOf()` method in relevant `ActorSystem`
- In Hello World example:

```
ActorRef mpNode = actorSystem.actorOf(mpProps, "Name");
```

  - This creates and launches a single actor in `actorSystem`
  - Actor is now ready to receive, process messages

Communicating with Actors

- Actors compute by processing messages
- To send a message to an actor, use `ActorRef` instance method **`tell(Object msg, ActorRef sender)`**
  - `tell()` takes message (payload) and sender as arguments
    - sender parameter allows return communication, although it should really be called "replyTo"
    - If no return communication desired, specify null for sender field
  - `tell()` is often said to implement "fire and forget" communication
    - Method call returns as soon as message handed off to infrastructure
    - No waiting to see if recipient actually receives it
- In Hello World example:

```
mpNode.tell("Hello World", null);
```

Shutting Down an ActorSystem

- `ActorSystem` objects use worker threads internally to execute actors
- These threads must be killed off before an actor-based application can terminate
- To do this: call `ActorSystem` instance method `terminate()`
- From Hello World example:

```
actorSystem.terminate();
```

Moving Information from ActorSystem to Java

- The `tell()` method permits messages to be sent to actors
  - In Hello World, this was how information was passed from “rest of Java” into actor
  - Actors can also send messages to each other inside an actor system
- How can actors communicate with outside world?

  Outside world (i.e. “rest of Java”) is not an actor, so `tell()` cannot be used!
- Solution: **`Patterns.ask()`**

MessageAcknowledgerActor.java

```
public class MessageAcknowledgerActor extends AbstractActor {
  …
  public void onReceive(String msg) throws Exception {
      ActorRef sender = getSender();
      System.out.printf("Message is:  %s%n", msg);
      sender.tell(msg + " message received", sender);
    }
  }
}
```

`getSender()`?

- Instance method in `AbstractActor`
- Returns `ActorRef` for sender of current message being processed in `Receive`
  - The sender is the second parameter of the tell() method call corresponding to the current message
  - A more accurate characterization: rather than thinking of this as message sender (it may not be!) think of it as “Reply-To”, as in e-mail

Actor Communication

- Actor(Ref)s communicate by sending each other messages
- To send a message to recipient `r`, a sender `s` needs to invoke `r.tell()`
- This means the sender needs to know `r`!
- Different ways to do this
  - Send a message to `s` containing `r` as payload
  - Send message to `s` with `r` as sender
  - In constructor associated with `s`, include `r` as parameter

Messages

- Messages are objects
- Valid classes of messages must match `Serializable` interface
  - Serializable objects can be converted into bytes
  - This is needed for actors to communicate over communication networks, which just transmit bytes
- They should also be *immutable*
  - Objects are properly constructed
  - Fields are private, final
  - State never changes

Quiz 1

In Actor model, actors communicate through

A. Shared Memory
B. Pipe
C. Message
D. Shared Files

Quiz 2

When Actors work concurrently, we don't need locking and blocking

A. True
B. False

Dynamic Actor Creation

- In Java we saw that tasks can create other tasks
- In akka Java, actors can also create other actors!
  - Actor creation so far has been done using calls to `actorOf()` method of `ActorSystem` object
  - It may also be done by calling `actorOf()` method of `ActorContext` object
    - An `ActorContext` object is the environment surrounding an actor
    - To get the `ActorContext` of an `AbstractActor` actor, call `getContext()` instance method

Supervision

- Every actor has exactly one supervising actor
  - When one actor creates another using first actor's context, first actor is *supervisor* of second
    - First actor often also called *parent*
    - Second usually called *child* or *subordinate*
  - What about actors created via ActorSystem actorOf()?
    - Every actor system three top-level actors (called *guardians*) that are started automatically
      - / The root guardian
      - /system The System guardian (child of /)
      - /user The Guardian Actor (child of /)
    - When an object is created using actorOf() in ActorSystem, it is by default made a child of `/user`
- What supervisors do
  - Delegate tasks to children
  - Take remedial action when children fail
- Supervision is basis of *fault tolerance* in akka

Getting Supervisory Information

- `ActorContext` has methods for retrieving parent, child information
  - `ActorRef parent()`

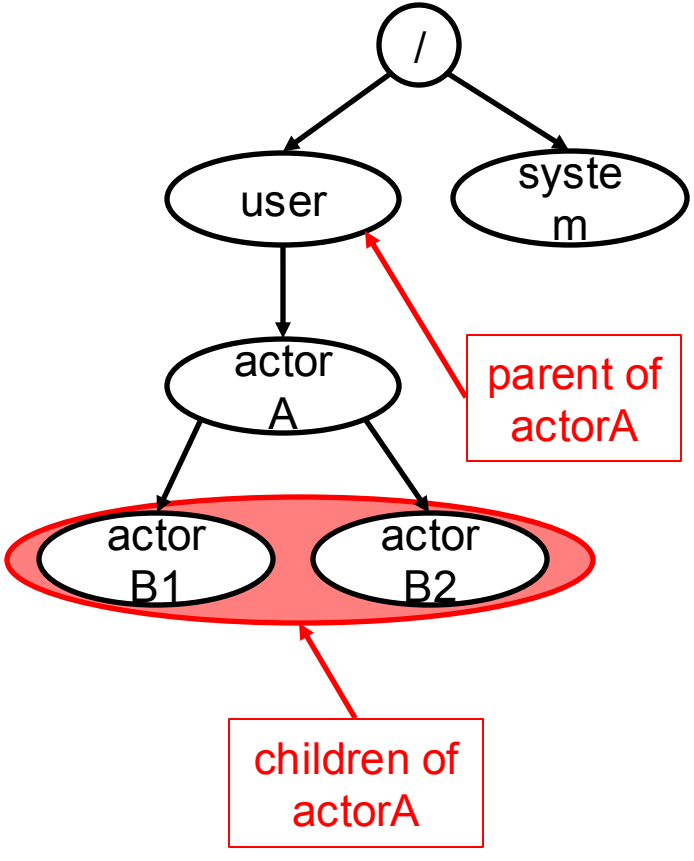
    Return parent of actor associated with context
  - `java.lang.Iterable<ActorRef> getChildren()`

    Return children as a Java Iterable
  - `ActorRef getChild(String name)`

    Return child having given name, or null if there is no such child
- To find parent of given actor, invoke following in body of actor definition: `getContext().parent()`

Supervisory Hierarchy

- Supervision relationship induces a tree
  - Every actor (except /) has exactly one parent
  - Every actor has ≥ 0 children
- Every actor can be identified via path (`ActorPath`) in tree
- To get path of `ActorRef`, use `path()` instance method
- For actorA
  - Parent: user
  - Children: actorB1, actorB2
  - Path: /user/actorA



How an Actor Can Find Its Name

- `getName()`? `name()`? No
  No such instance methods in `AbstractActor`
- `getSelf().getName()`? `getSelf().name()`? No
  No such instance methods in `ActorRef`
- `getContext().getName()`?
  `getContext().name()`? No
  No such instance methods in `ActorContext`
- Solution: go through `ActorPath`
  - `ActorPath` objects have `name()` method returning name (`String`) of actor at that path
  - So, `getSelf().path().name()` returns name of yourself

Supervision in Detail

- When an actor fails (i.e. throws an exception) a special system message is sent to its parent
  - Systems messages have their own message queue; they are not handled by `createReceive()`
  - No guarantees about precedence of system messages over regular messages
- Parent actor has four choices in akka
  1. *Resume* the failed child in child's accumulated internal state
  2. *Restart* the failed child in its initial state
  3. *Stop* the failed child permanently
  4. *Escalate* (i.e. fail itself, handing off responsibility to its own parent)
- Communication associated with these choices is via system messages that are handled by special system-message queue

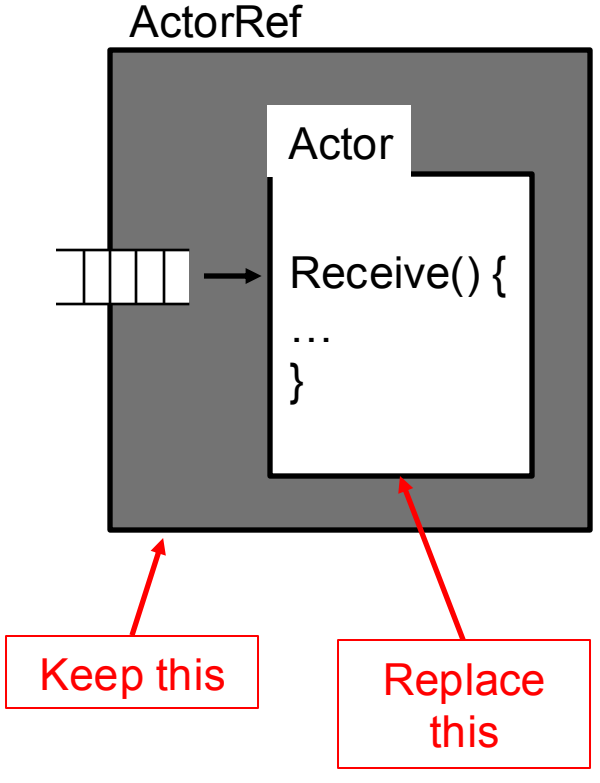
  This queue is only used for supervision (i.e. parent-child) communication

Resumption of Failed Child

- `createReceive()` method in child is re-invoked
  - Message being processed when failure occurred is lost
  - Processing of messages in child's message queue resumes
- When to do this?
  - Maybe if transient system fault caused failure
  - Maybe if there is a bug in child that doesn't affect its ability to process future messages

Restarting a Failed Child

- Idea
  - Create new actor instance
  - Replace actor instance in ActorRef for failed child with new instance
    - Path unchanged
    - So is name
  - Invoke Receive() method of new actor instance to start processing messages in message queue
- Message processed during failure is lost, but no pending messages in failed child's mailbox are



Stopping an Actor

- Stopping a child during supervision involves a general actor-stopping technique
- `ActorContext` objects include following method

  `void stop(ActorRef actor)`
  - Stops actor
  - Processing of current message completes first, however
- What about messages in mailbox when actor is stopped? And those sent to stopped actor?
  - These are called *dead letters*
  - akka uses a special actor (`/deadLetters`) to handle these
  - There are also mechanisms for retrieving them
- What about children?
  - They are stopped also,
  - This percolates downwards through supervision hierarchy, to children's children, children's children's children, etc.

Actors Can Stop Other Actors …

- … even themselves!
- If following is executed in AbstractActor …

```
getContext().stop(getSelf())
```

- … then it stops itself! (And consequently its children, grandchildren, etc.)
  - When an actor is stopped, its supervisor is notified
  - So are other actors that are monitoring this actor
  - akka buzzwords for this: DeathWatch, DeathPact
    - Special **`Terminated`** messages (these are not system messages, so are delivered to regular mailboxes) are sent to actors that have registered with stopped actor
    - Registration is done via **`watch()`** method in **`ActorContext`**
    - De-registration: **`unwatch()`** method in same class

Failure Escalation

- As name suggests, escalation in response to child failure means that parent fails by throwing same exception as child
- Parent's parent then must handle failure

Details of Supervision

- Each `AbstractActor` object contains a `SupervisorStrategy` object
  - To obtain `SupervisorStrategy` object, execute actor's `supervisorStrategy()` instance method
  - This method may be overridden in order to customize supervision approach
- The `SupervisorStrategy` determines how failures of children will be handled

Two Kinds of `SupervisorStrategy`

- `AllForOneStrategy (subclass of SupervisorStrategy)`
  - If one child fails, apply supervision strategy to all of the children, not just the failing one
  - Used if children are tightly coupled
- `OneForOneStrategy (also subclass of SupervisorStrategy)`
  - Apply supervision strategy only to failing child; other children left unaffected
  - Used if children are largely independent

Deciders

- Core of a `SupervisionStrategy`: *decider*
  - A decider maps exception classes to directives, which describe which of four mechanisms to use to recover
  - A directive has one of four forms: Escalate, Restart, Resume, Stop
- You may customize a `SupervisionStrategy` by changing the decider
- There is also a default decider

akka and the Java Memory Model

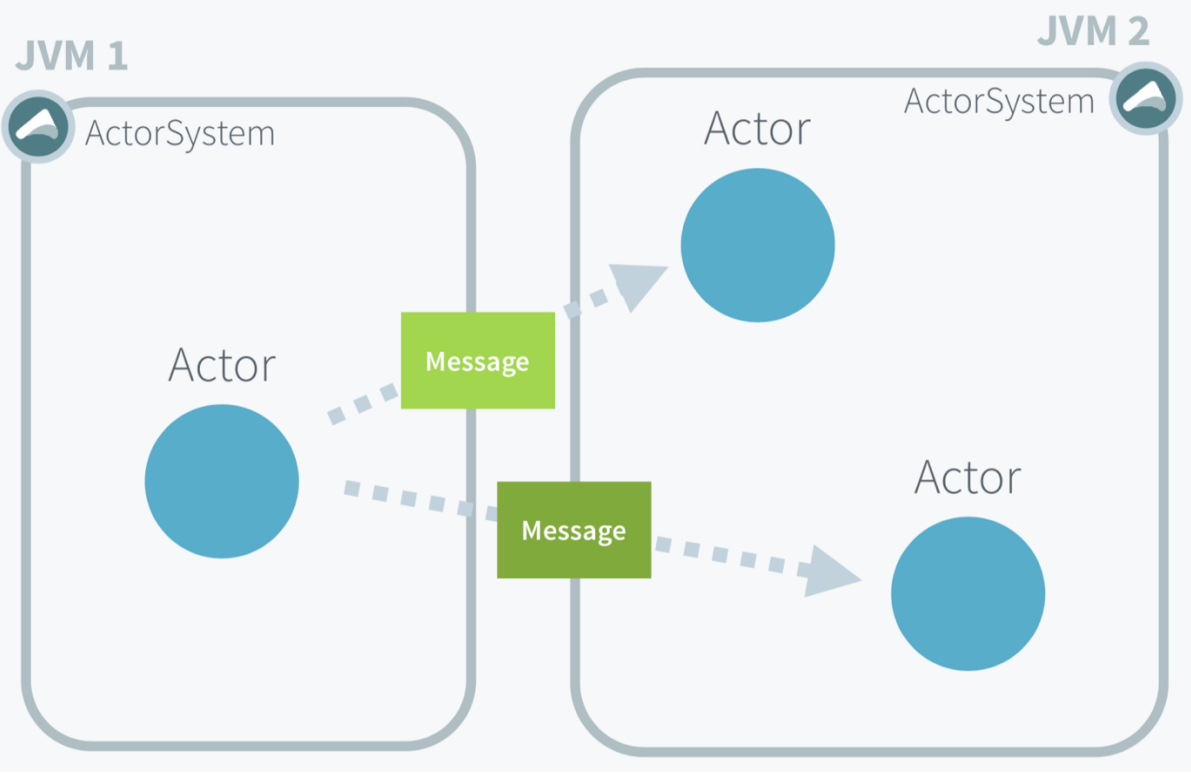
- Actors do not (intentionally) share memory
- In a local application (single JVM), one still needs to worry about visibility

- akka guarantees the following
  - If one actor sends a message to another, then pending writes before the send are guaranteed to be visible after the receipt

Akka “happens before” rules

- To prevent visibility and reordering problems on actors, Akka guarantees the following two “happens before” rules:
  - **The actor send rule:** the send of the message to an actor happens before the receive of that message by the same actor.
  - **The actor subsequent processing rule:** processing of one message happens before processing of the next message by the same actor.

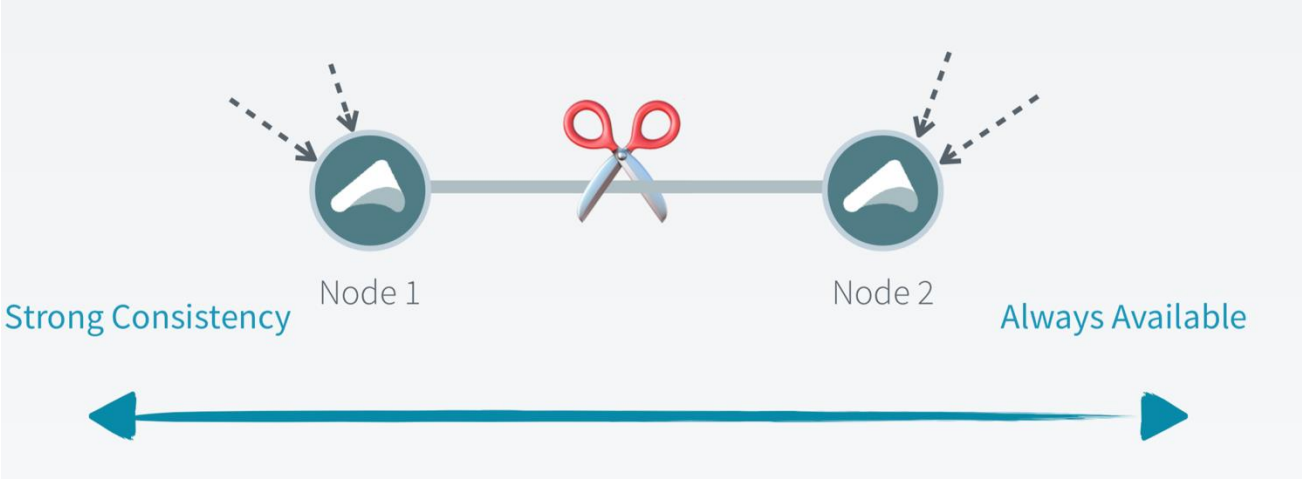
Distributed



Distributed: It is hard

- Reliability:
  - power failure, old network equipment, network congestion, coffee in router, rodents, DDOS attacks...
- Latency:
  - loopback vs local net vs shared congested local net vs internet

Consistency vs Availability



CAP Theorem

- It is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees
  - Consistency
    - Every read receives the most recent write or an error
  - Availability
    - Every request receives a (non-error) response
  - Partition tolerance
    - The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

CAP Theorem

- No distributed system is safe from network failures, thus network partitioning generally has to be tolerated.

- We left with two options:
  - Consistency
  - Availability.

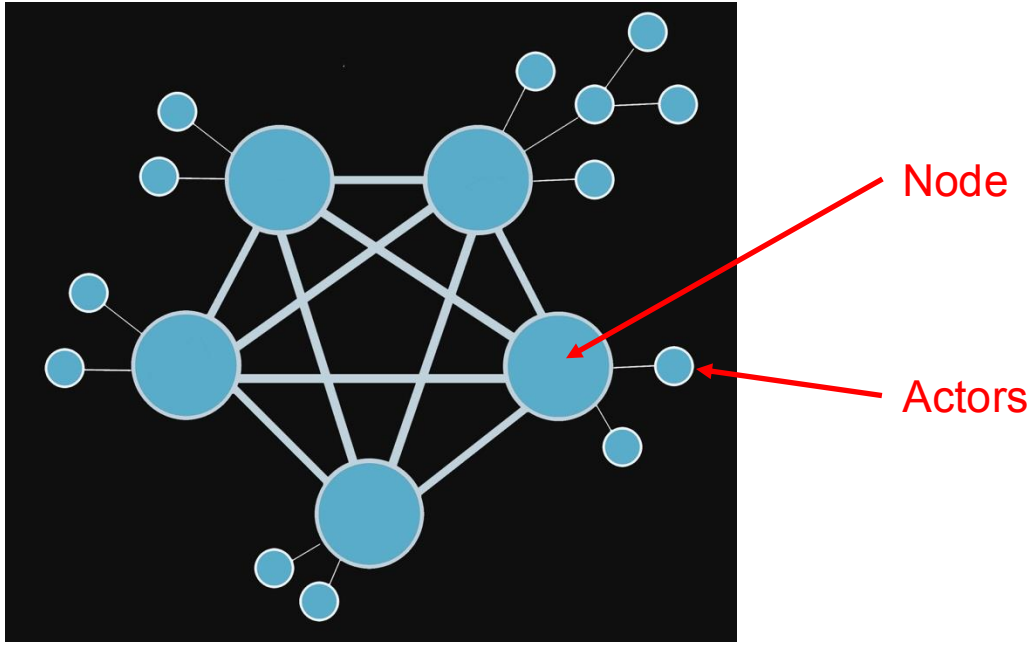
CAP Theorem

- Choose consistency over availability
  - the system will return an error or a time-out if particular information cannot be guaranteed to be up to date due to network partitioning.
- Choosing availability over consistency
  - the system will always process the query and try to return the most recent available version of the information, even if it cannot guarantee it is up to date due to network partitioning.
- In the absence of network failure – that is, when the distributed system is running normally – both availability and consistency can be satisfied.

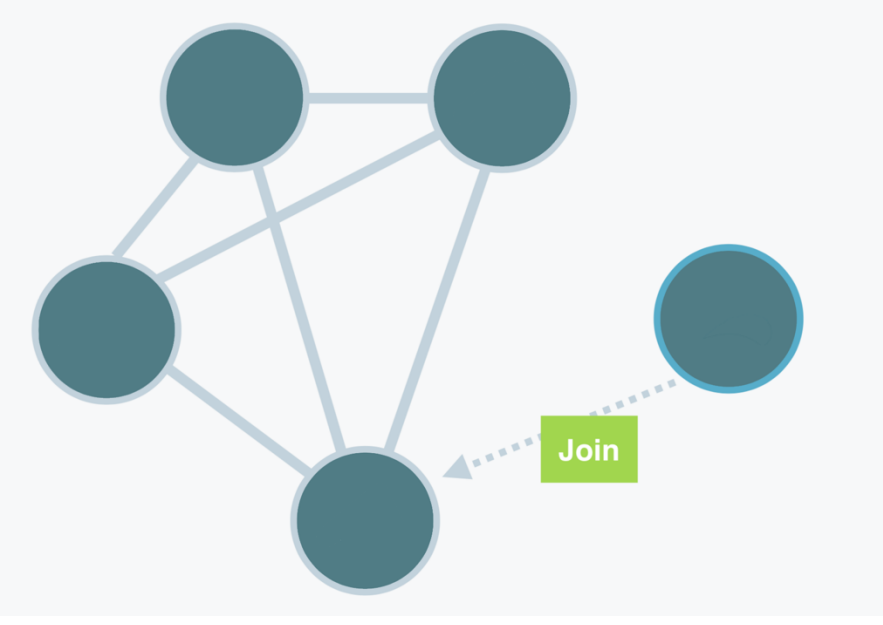
ACID vs BASE

- ACID: **A**tomicity, **C**onsistency, **I**solation, **D**urability
- Base: **B**asically **A**vailable, **S**oft state, **E**ventual consistency
- Database systems designed with traditional ACID guarantees in mind such as RDBMS choose consistency over availability
- Systems designed around the BASE philosophy, common in the NoSQL movement for example, choose availability over consistency

Akka Cluster

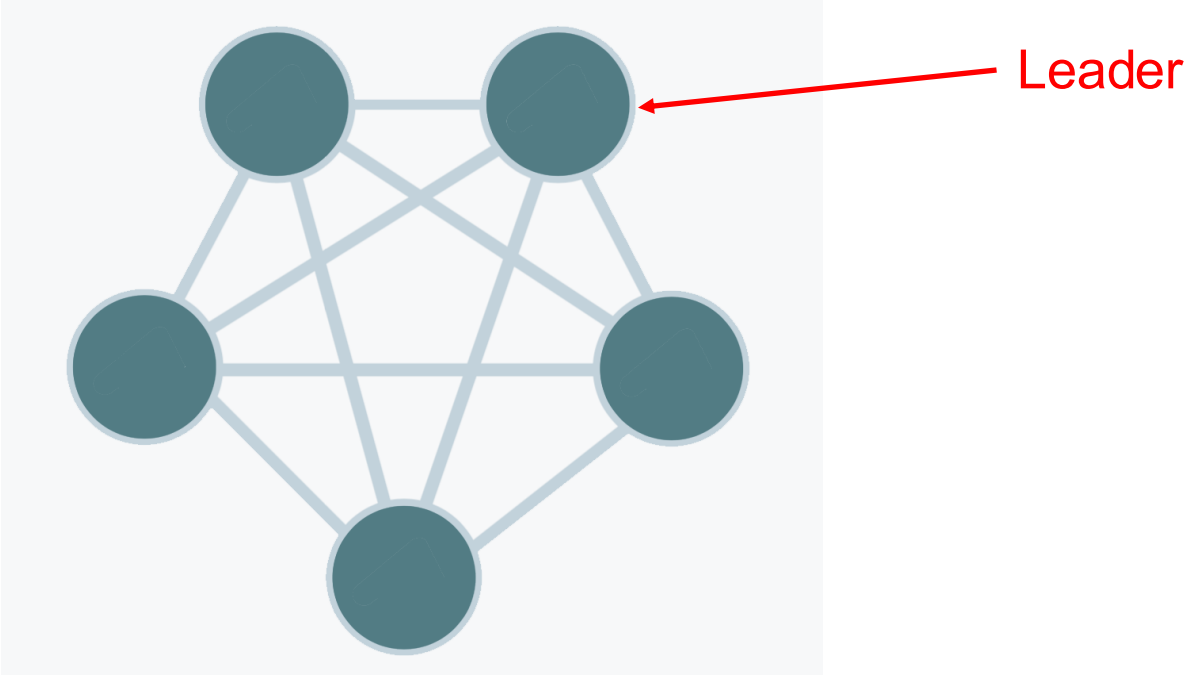


Joining



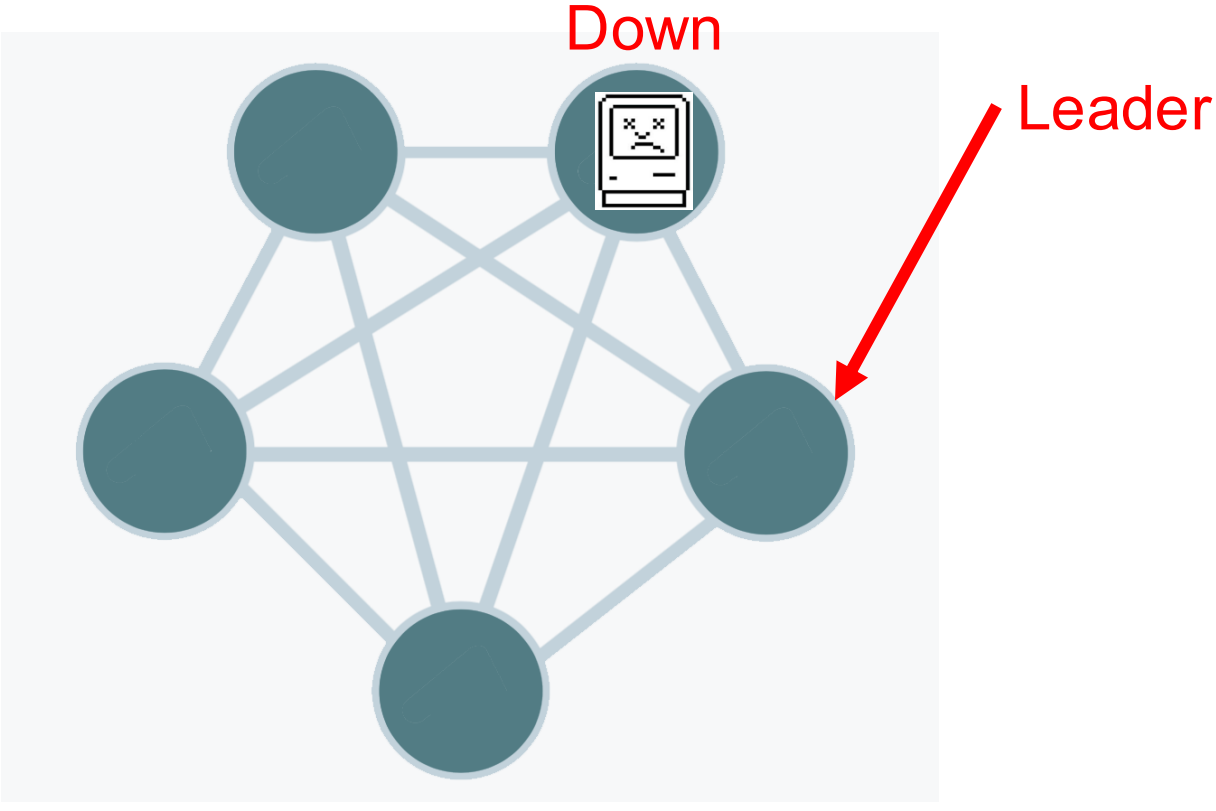
When a new node is started it sends a message to all seed nodes and then sends join command to the one that answers first.

Leadership

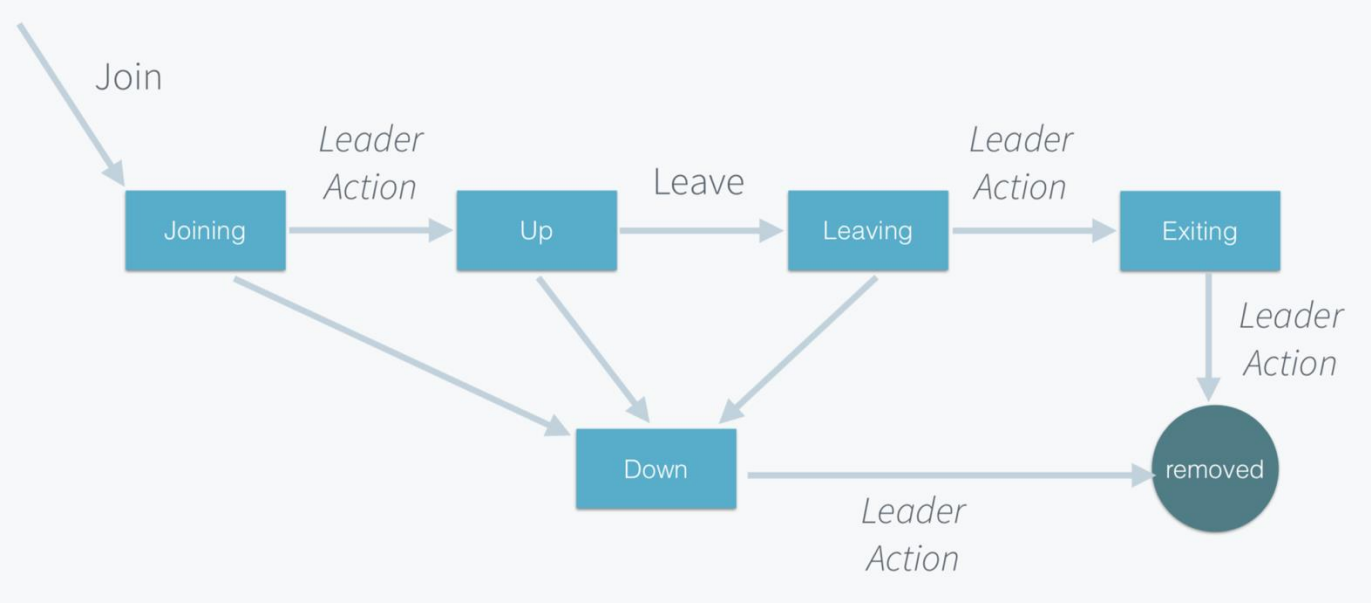


Leader: A single node in the cluster that acts as the leader.
Managing cluster convergence and membership state transitions.

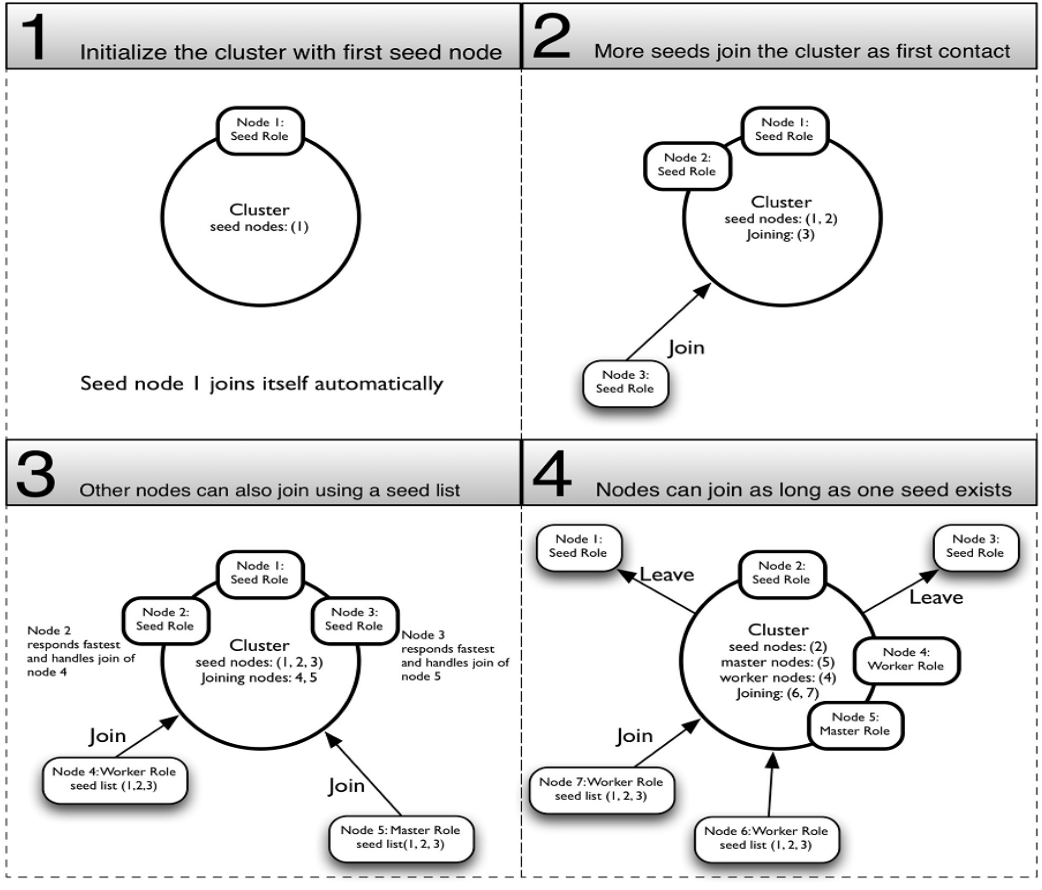
Leadership



Cluster Member Lifecycle



Joining the Cluster: Seed Node

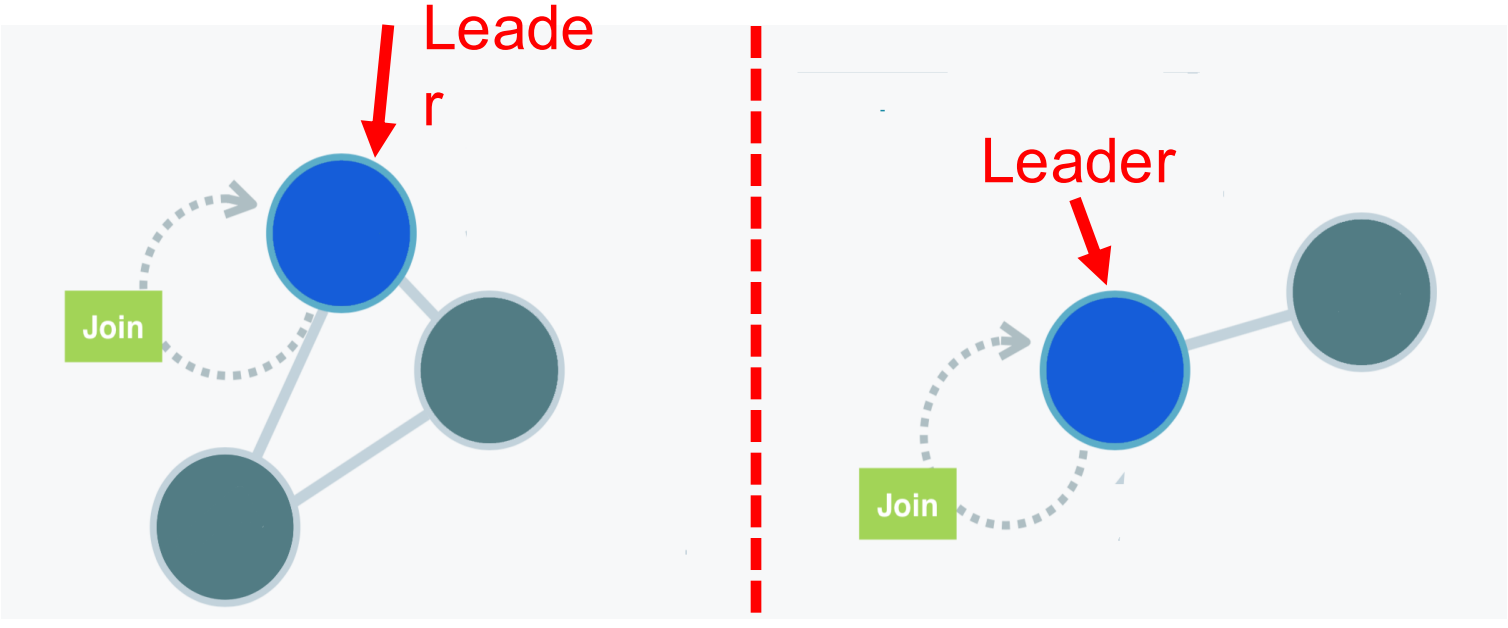


Seed Node Config

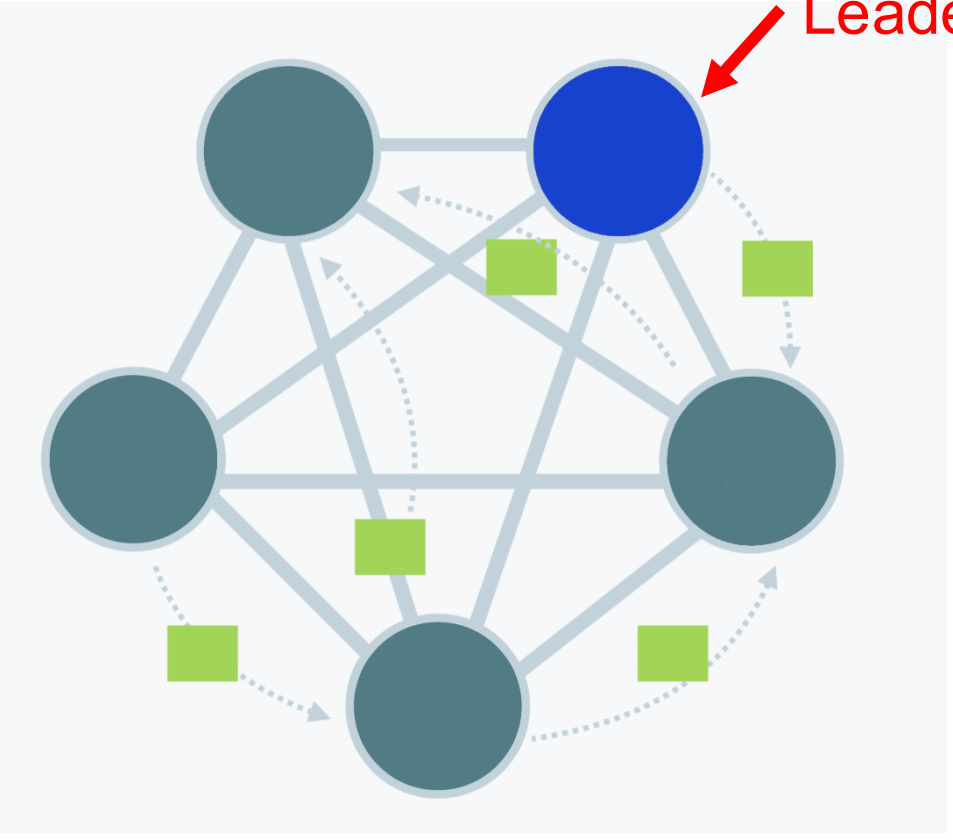
You define the seed nodes in the configuration file (application.conf):

```
akka.cluster.seed-nodes = [
        "akka.tcp://ClusterSystem@host1:2552",
        "akka.tcp://ClusterSystem@host2:2552"]
```

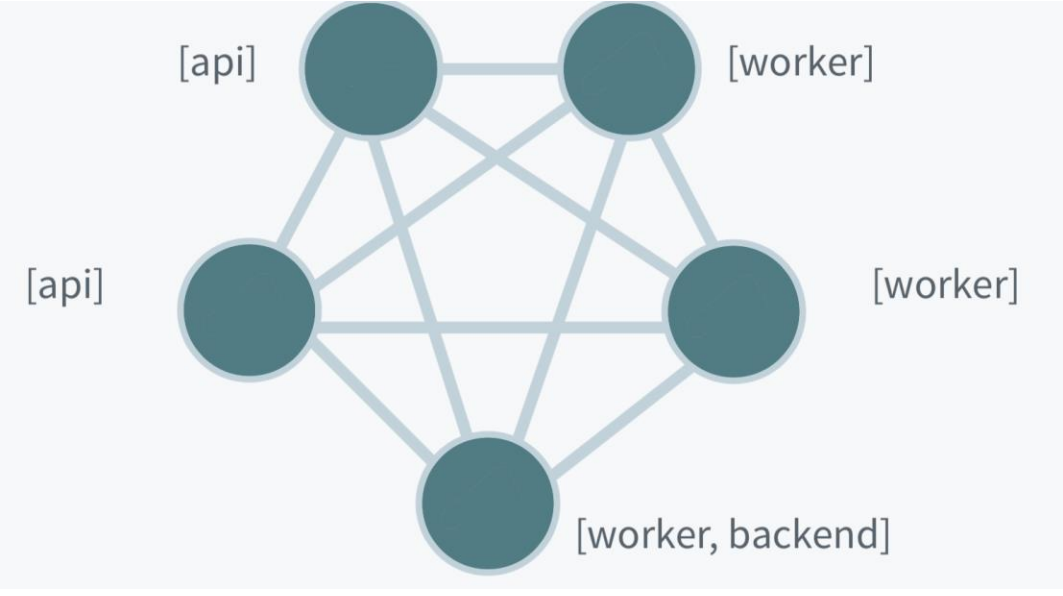
Network Partition



Gossip & Heartbeats



Node Roles

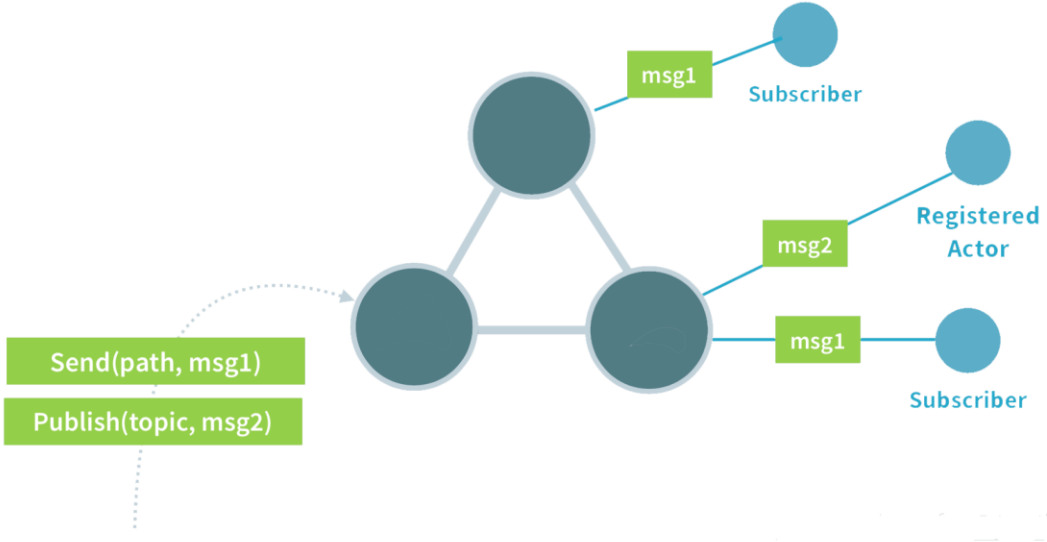


User API

- Node details
  - What roles am I in, what is my address
- Join, Leave, Down
  - Programmatic control over cluster membership
- Register listeners for cluster events
  - Every time the cluster state changes the listening actor will get a message

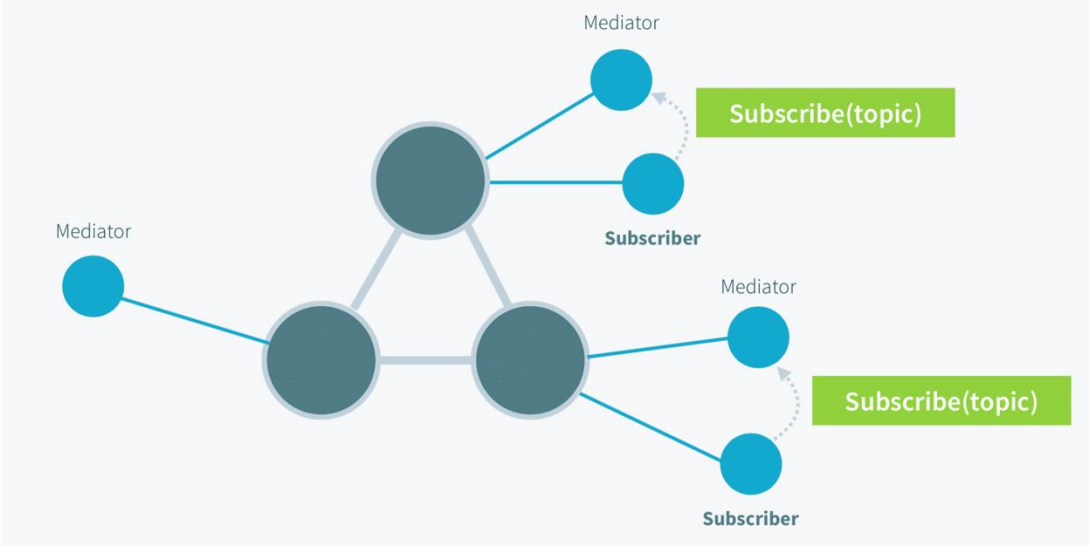
Distributed Publish Subscribe

- How do I send a message to an actor without knowing which node it is running on?
- How do I send messages to all actors in the cluster that have registered interest in a named topic?

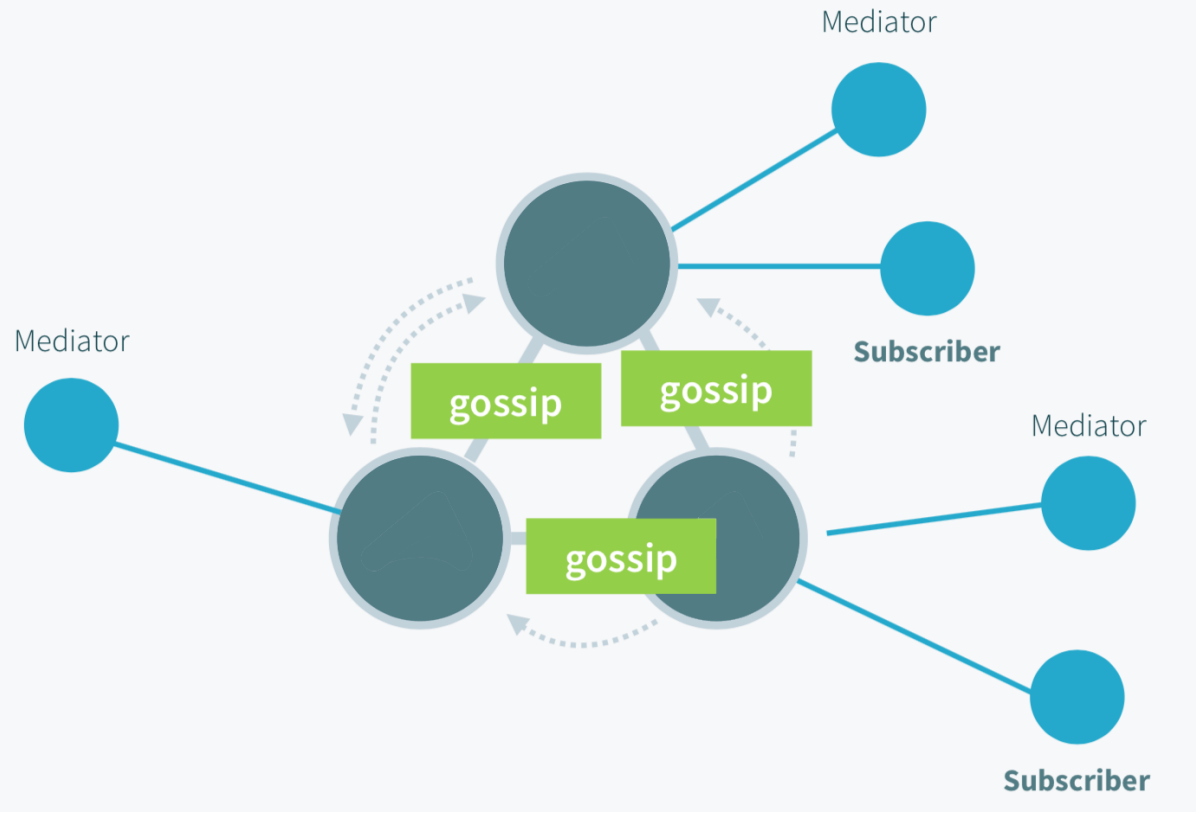


Distributed Publish Subscribe

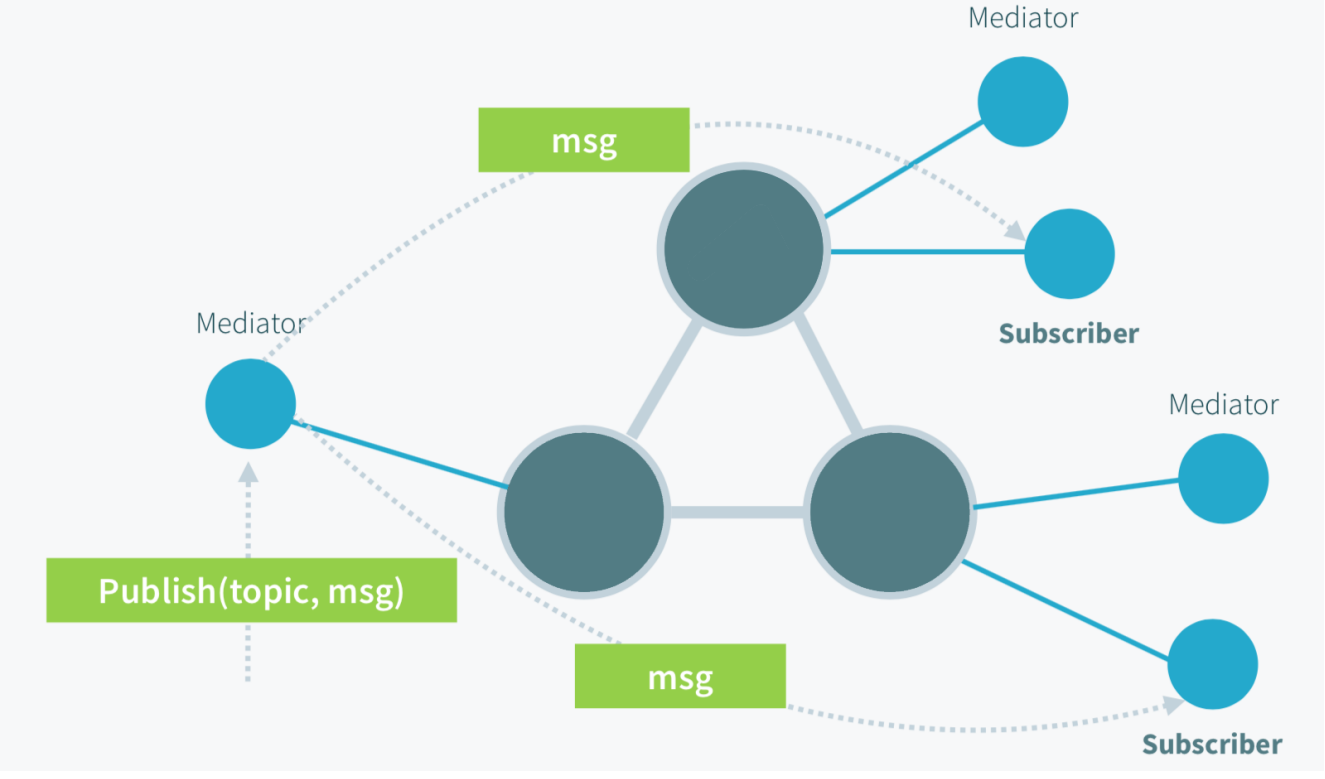
- Pub Sub Mediator actor (akka.cluster.pubsub.DistributedPubSubMediator) manages a registry of actor references and replicates the entries to peer actors among all cluster nodes or a group of nodes tagged with a specific role.



Distributed Publish Subscribe



Distributed Publish Subscribe

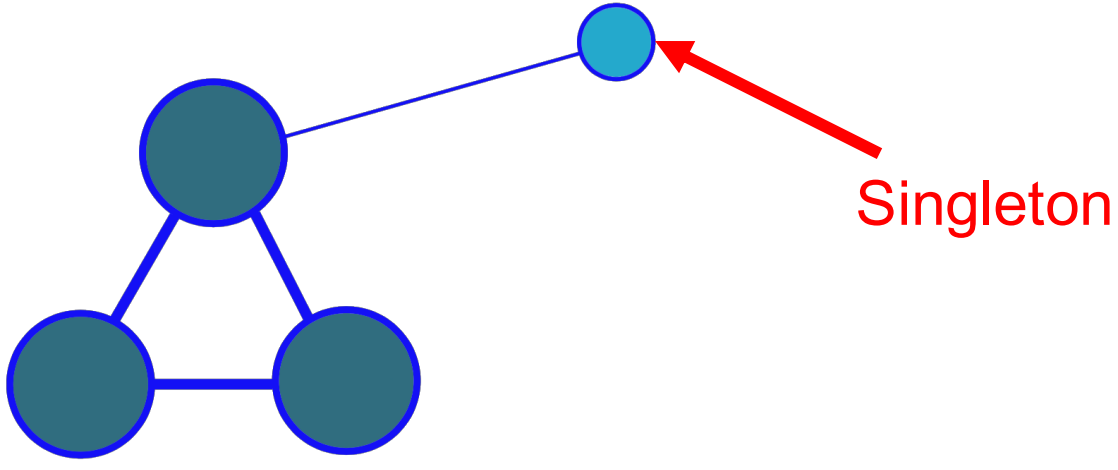


Cluster Singleton

- For some use cases it is convenient and sometimes also mandatory to ensure that you have exactly one actor of a certain type running somewhere in the cluster.
  - single point of responsibility for certain cluster-wide consistent decisions, or coordination of actions across the cluster system
  - single entry point to an external system
  - single master, many workers
  - centralized naming service, or routing logic

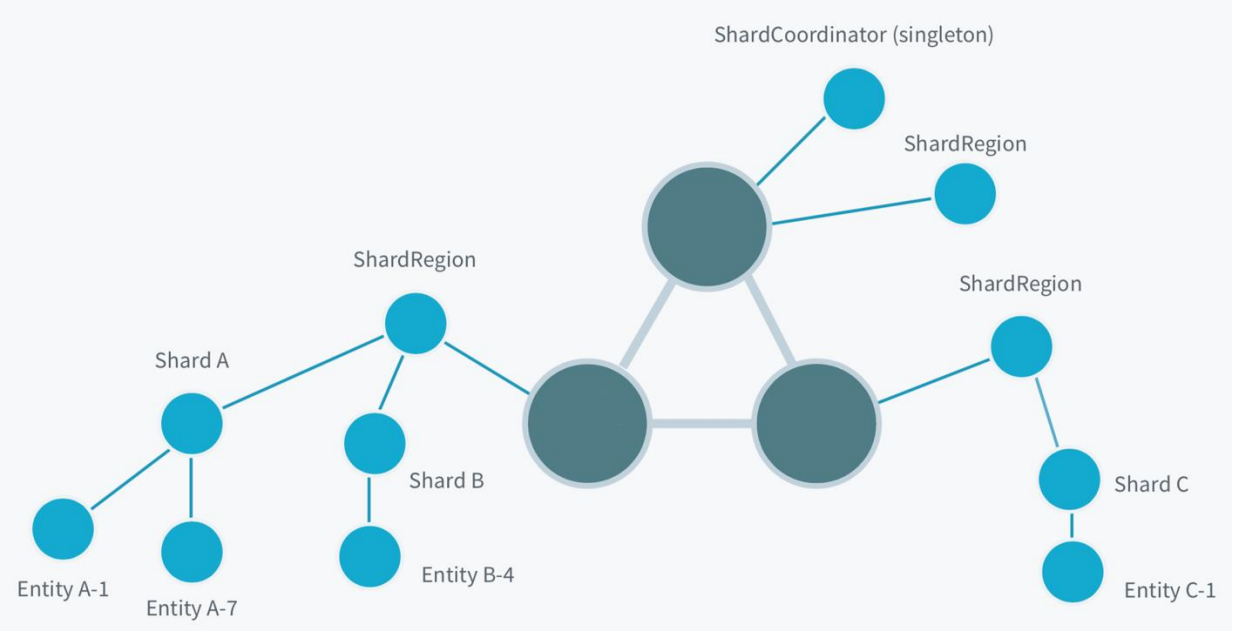
Cluster Singleton

- Single-point of bottleneck
- Failure: another singleton instance will eventually be started.



Cluster Singleton

Cluster Sharding



Distribute actors across several nodes in the cluster and interact with them using their logical identifier, but without having to care about their physical location in the cluster, which might also change over time.

Summary

- Akka Actors
- Akka Remote
- Akka Cluster
- **Akka Persistence**
- Akka Streams
  - An intuitive and safe way to do asynchronous, non-blocking backpressured stream processing.
- Akka HTTP
  - Modern, fast, asynchronous, streaming-first HTTP server and client.
- …