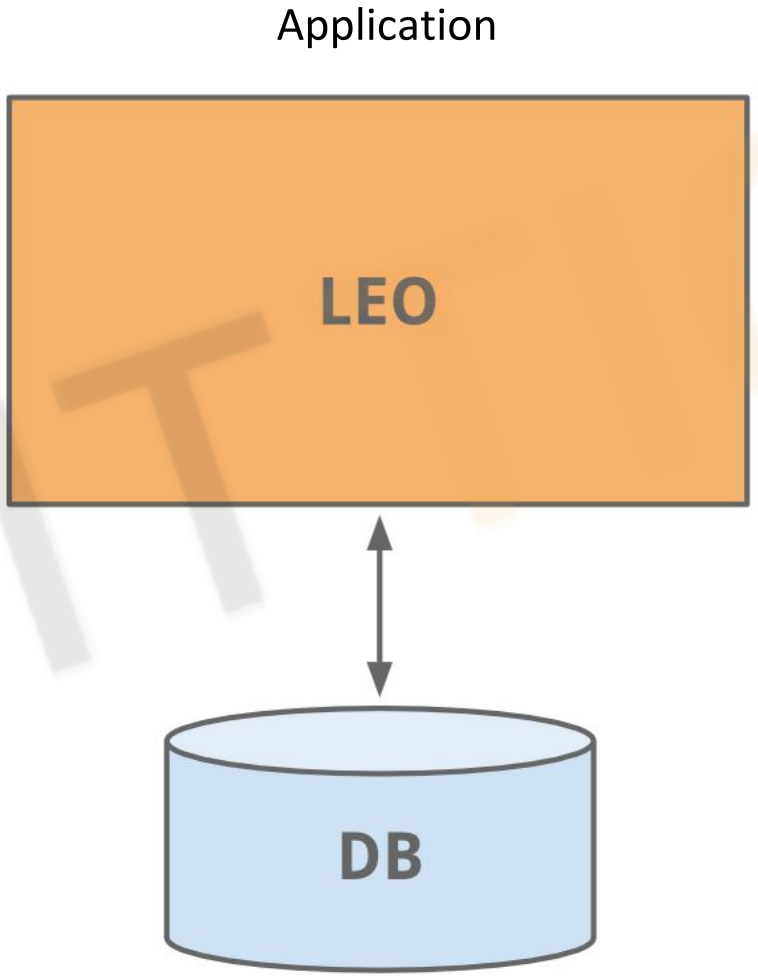
Design LinkedIn

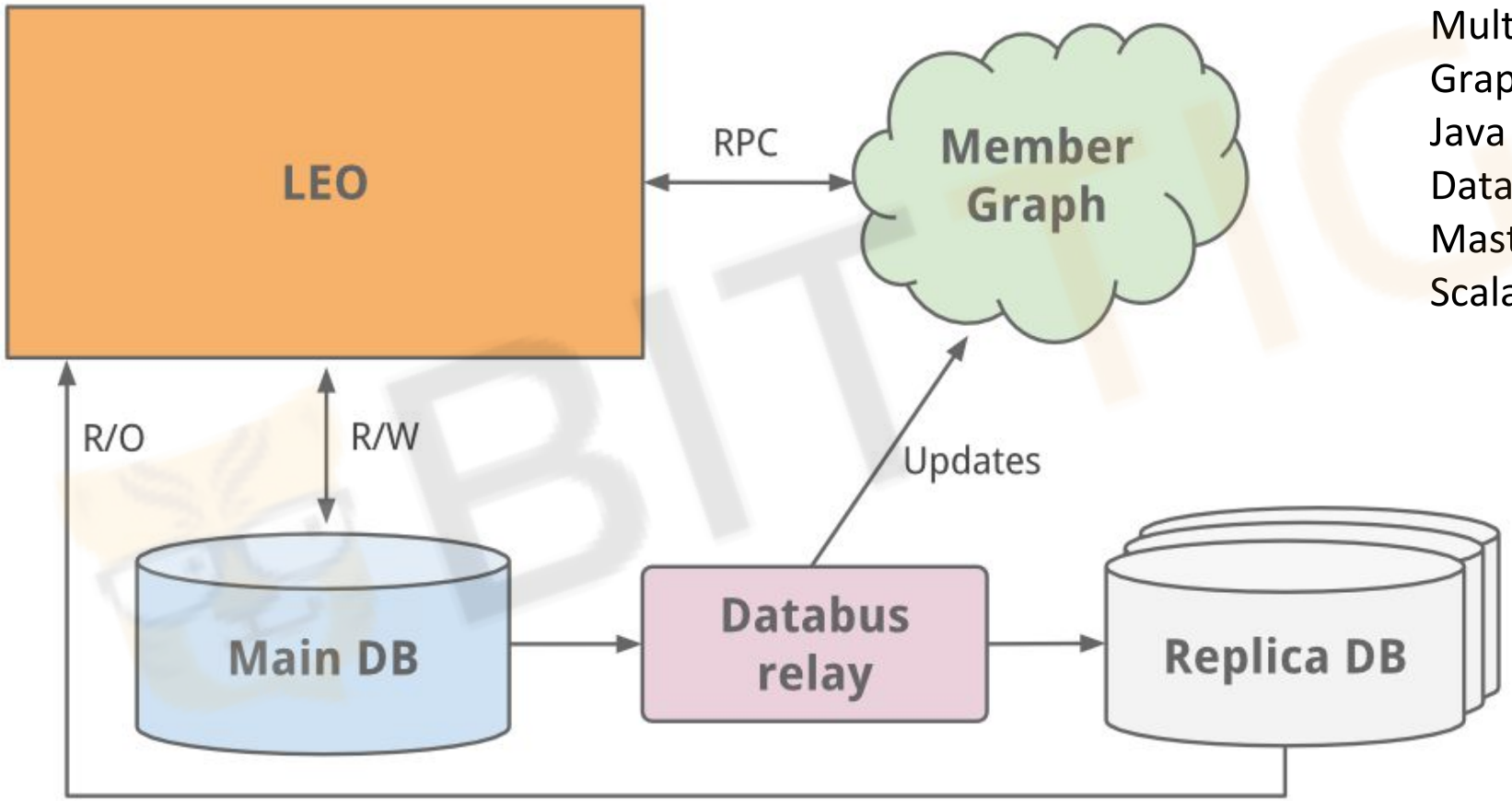
LinkedIn

- Profile
- Search
- Share
- Recommendation
- News feed
- Ads
- Messages

The early days

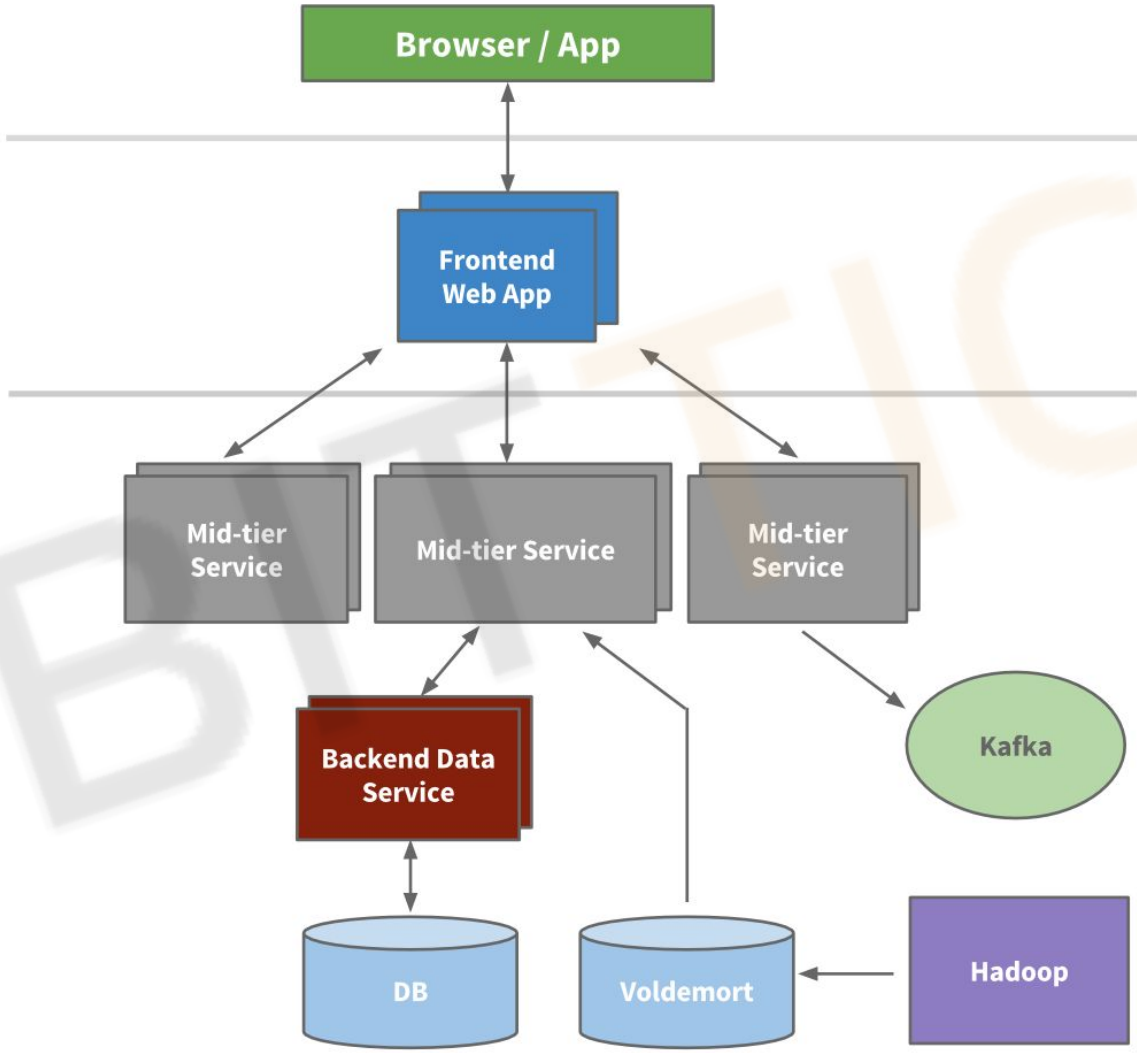


Distributed systems

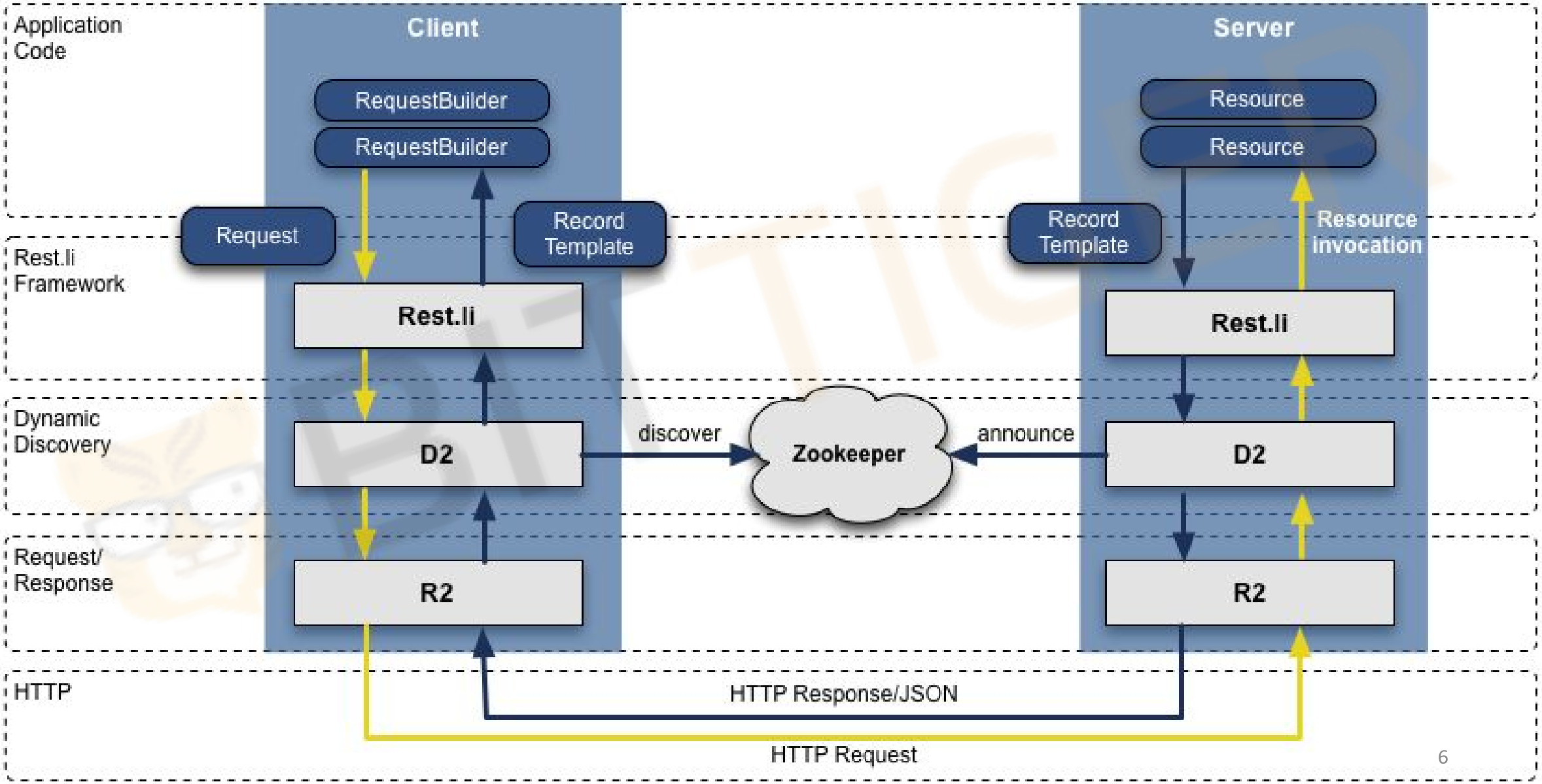


Multiple LEO
Graph queries
Java RPC
Database replication
Master/Slave Architecture
Scalable read

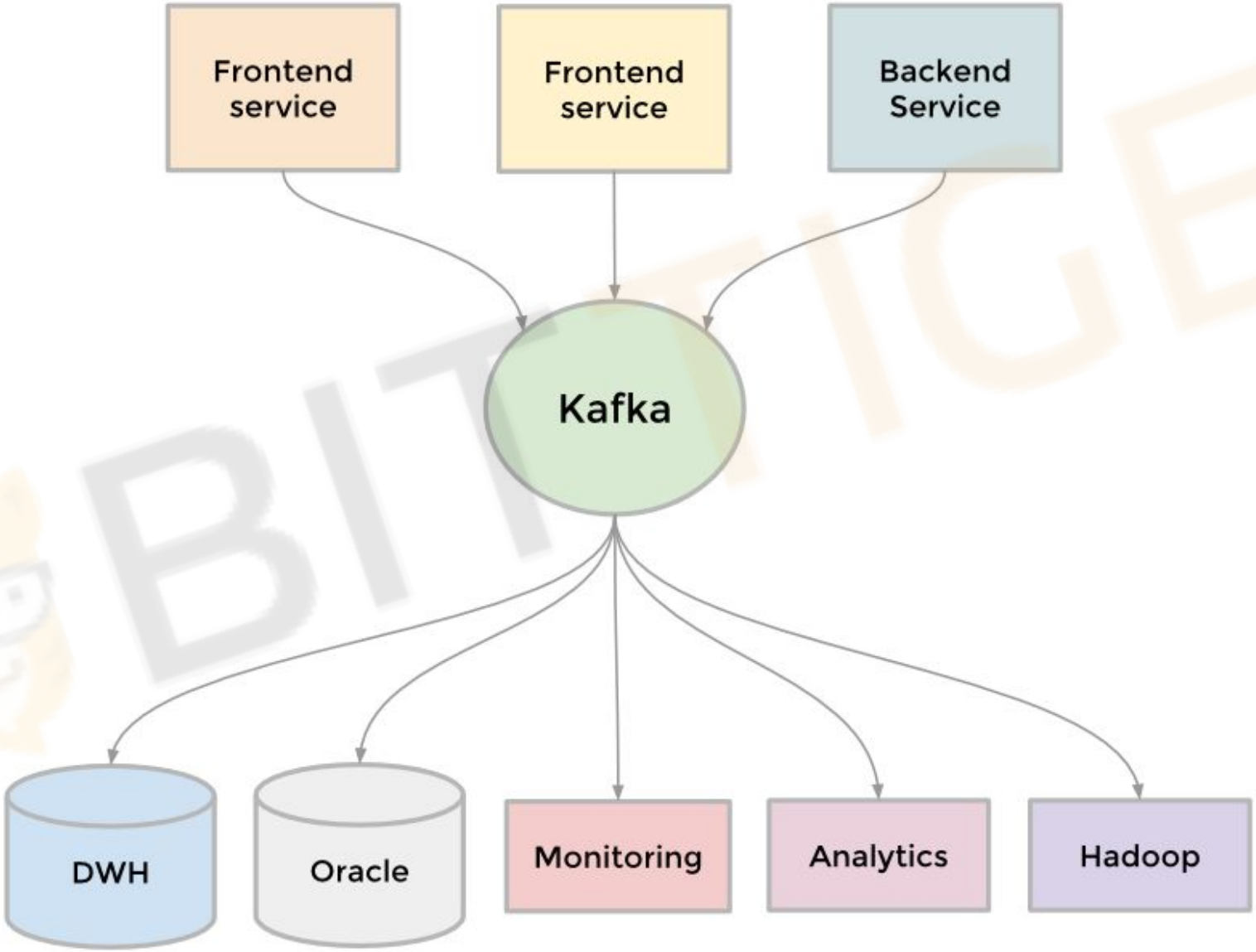
Service oriented architecture



Standardize REST API



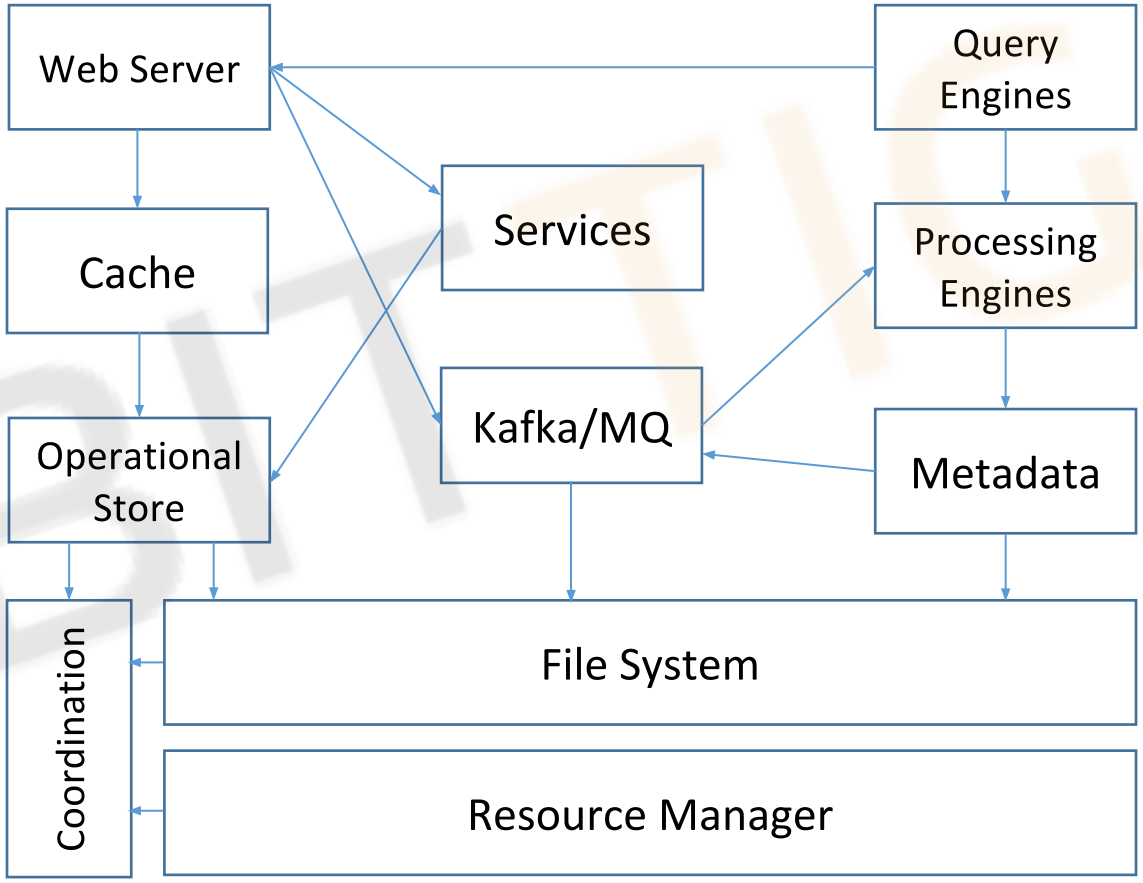
Data pipeline



Scaling LinkedIn

- Distributed systems
  - member graph
- Separation of concern
  - service oriented architecture
- Optimization of common cases
  - read/write separation
- Asynchronous processing
  - Kafka
- Standard
  - Rest.li

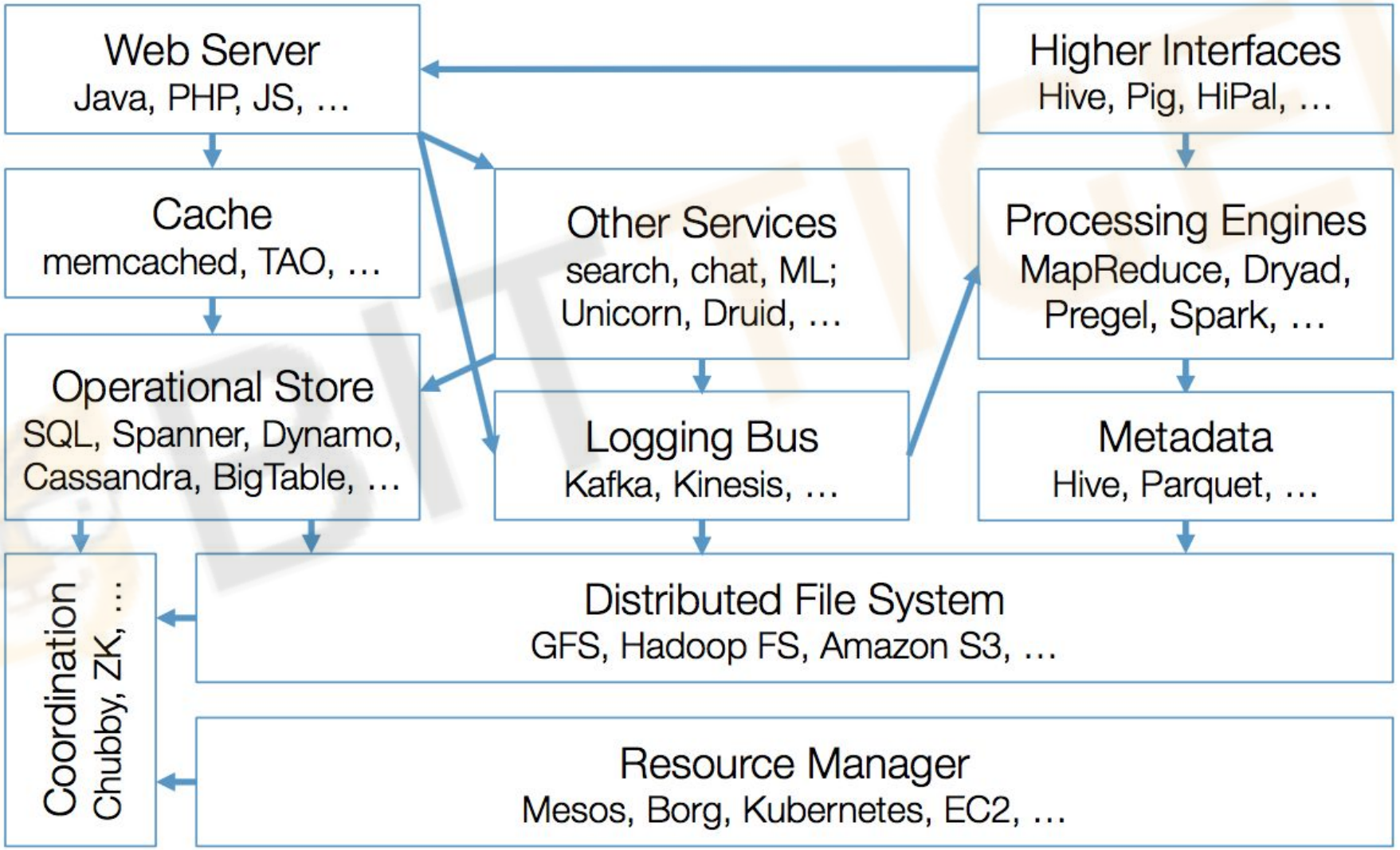
Scalable architecture



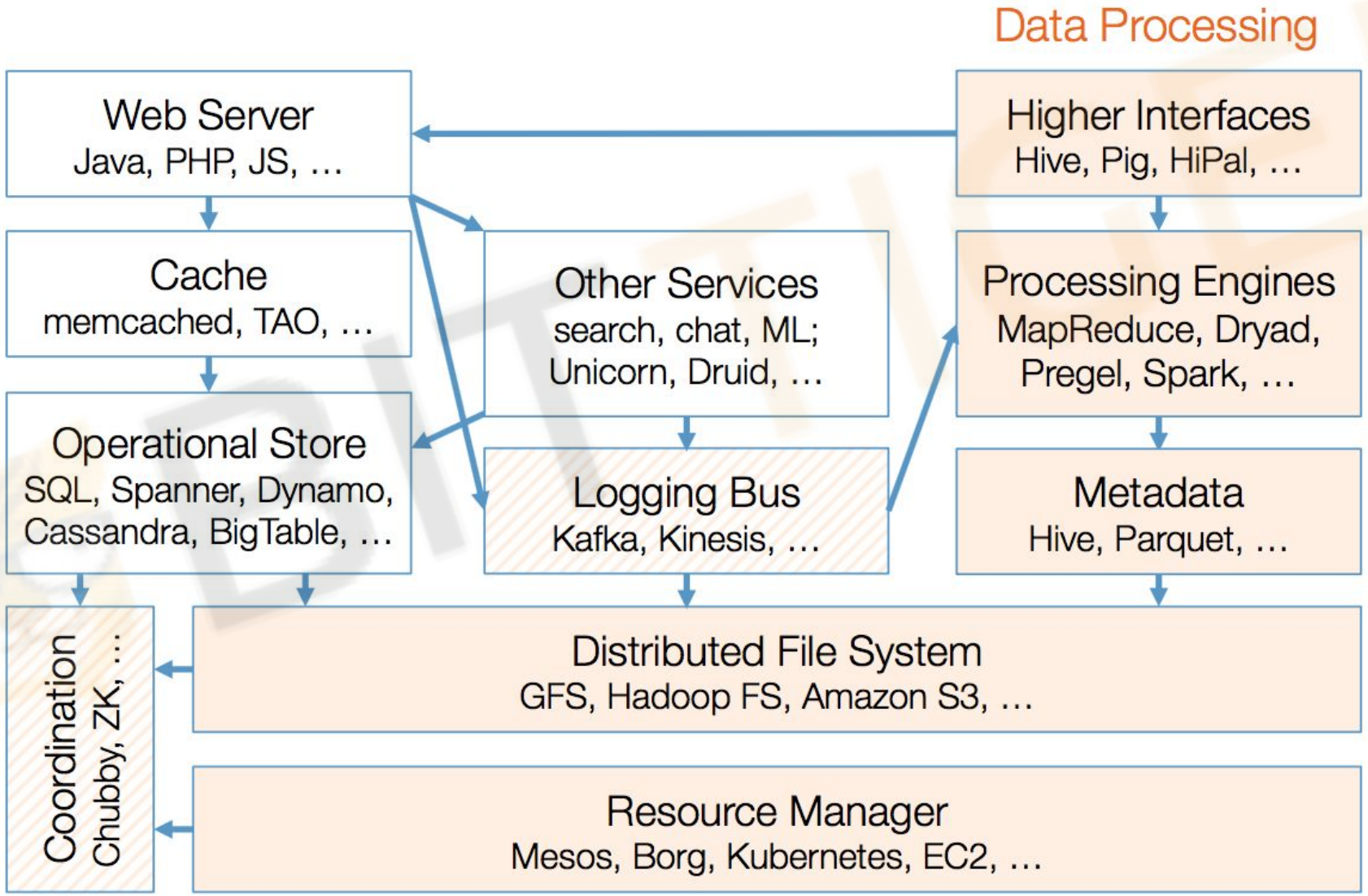
LinkedIn software stack

- Operational Store
  - Espresso
  - Vodemort
- Services
  - Search: Galene
  - Graph: member graph service
  - SOA: Rest.li
- Processing engines
  - Offline: Hadoop, Spark
  - Online: Samza
- Metadata
  - Schema Registry
  - Hive metastore

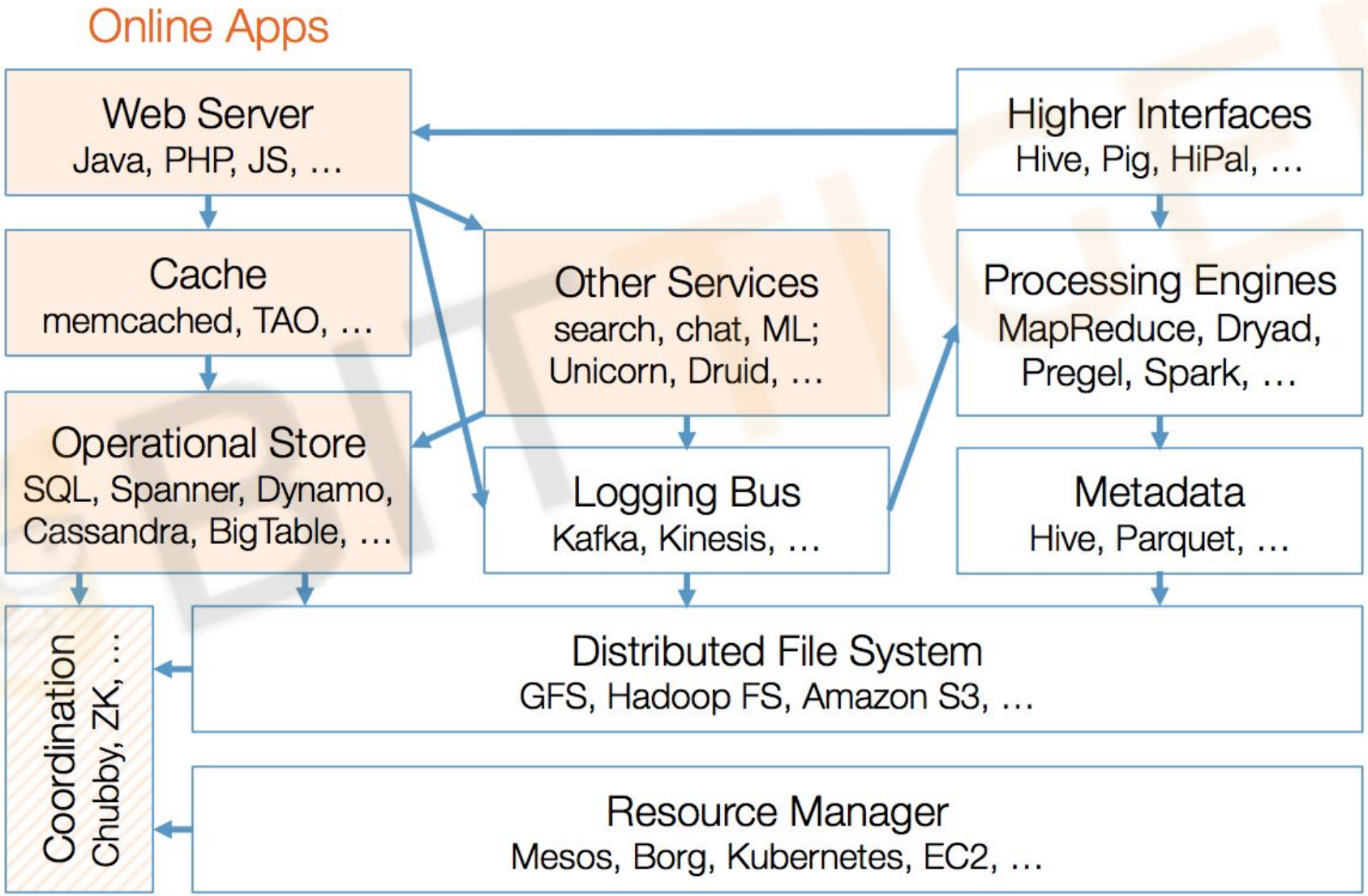
Scalable architecture



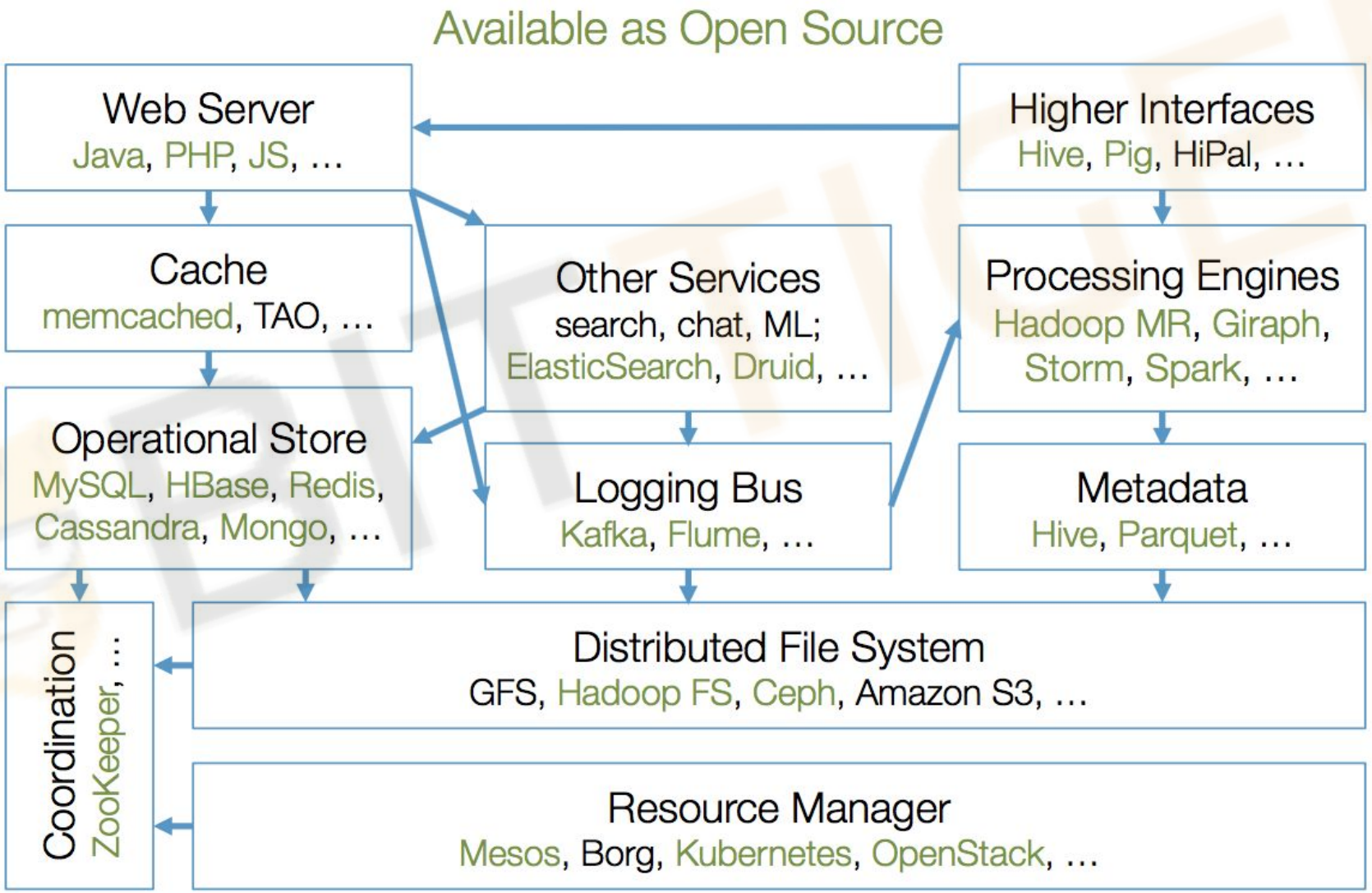
Scalable architecture



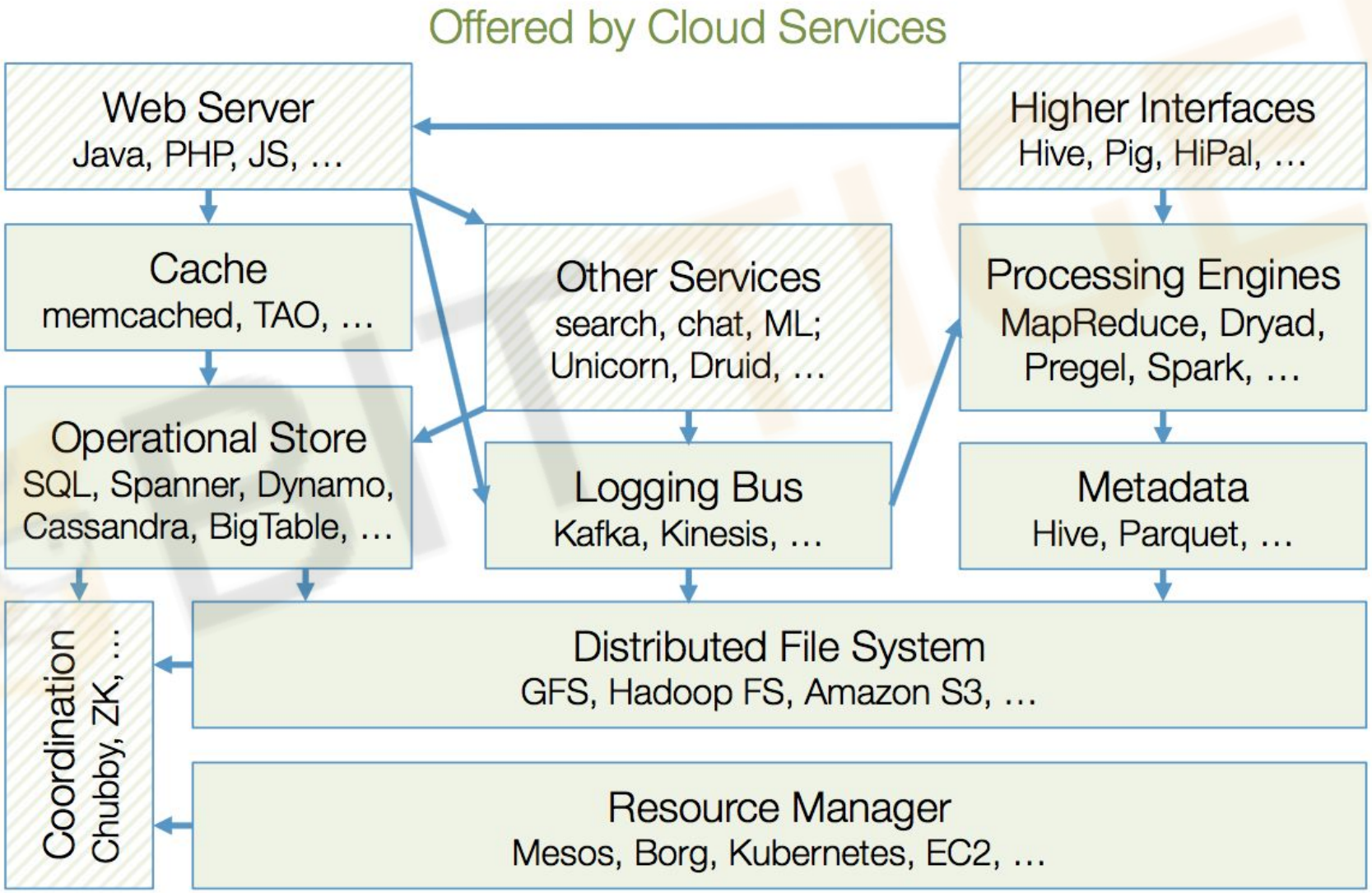
Scalable architecture



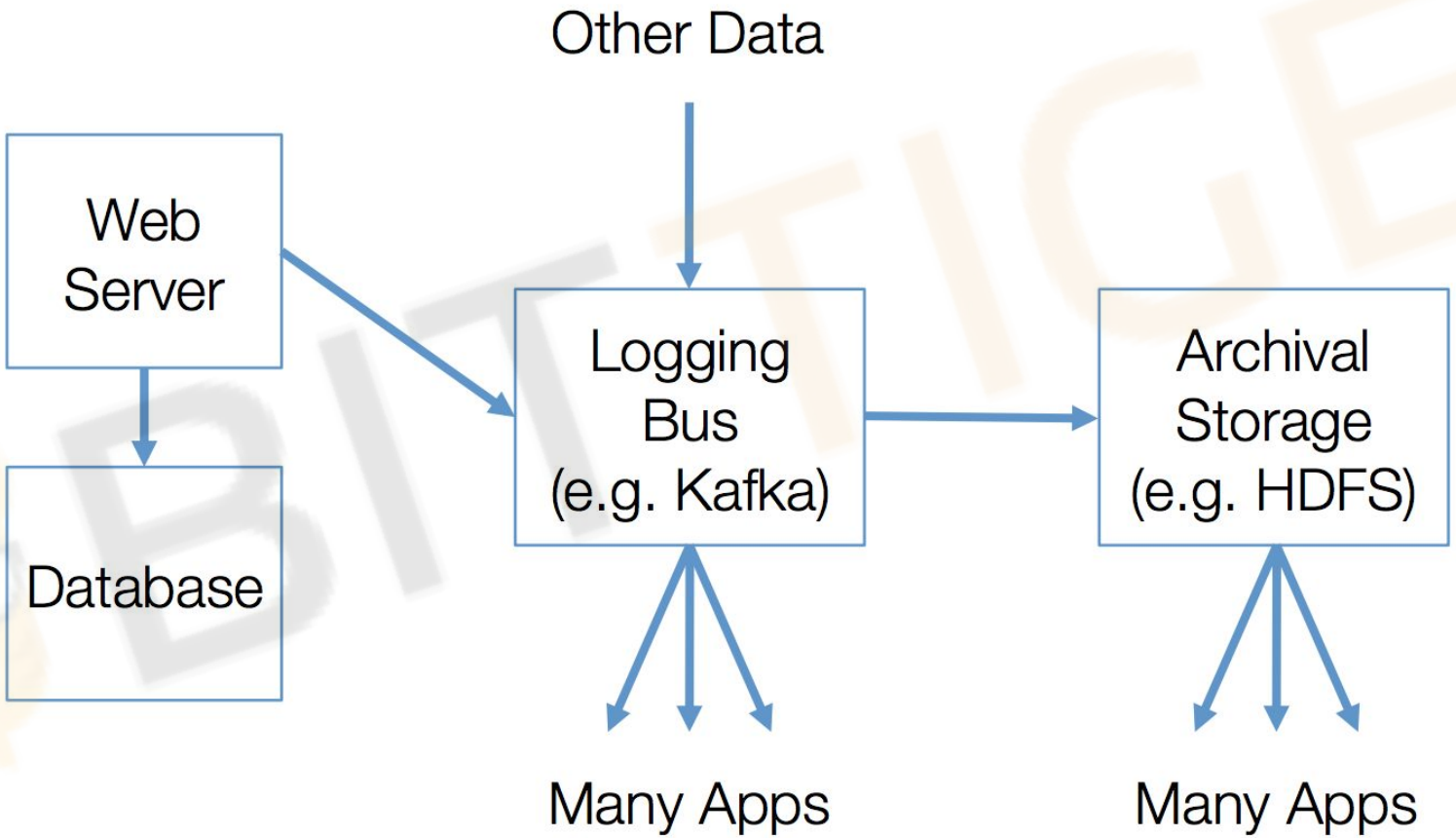
Scalable architecture



Software stack



Example architecture



Recommended reading: scaling facebook

- https://people.csail.mit.edu/matei/courses/2015/6.S897/readings/facebook-hpca2012.pdf
- Challenges of scaling facebook
  - Data is connected
  - Complex infrastructure
- Scaling facebook
  - Web: HipHop VM Services: Pull approach
  - Cache: Tao, Memcached
  - Storage: Haystack, f4

Delayed task scheduler

Delayed task scheduler

```
public interface Scheduler{
 void schedule(Task t, long delayMs);
}

public interface task{
 void run();
}
```

Delayed task scheduler: attempt 1

```
public class SchedulerImpl implements Scheduler{
 public void schedule(Task t, long delayMs){
  try{
   Thread.sleep(delayMs);
   t.run();
  }catch(InterruptedException e){
   //ignore
  }
 }
}
```

Delayed task scheduler: attempt 1

```
public class SchedulerImpl implements Scheduler{
 public void schedule(Task t, long delayMs){
  try{
   Thread.sleep(delayMs);
   t.run();
  }catch(InterruptedException e){
   //ignore
  }
 }
}
```

Sleep for delayMs, then execute the task

Delayed task scheduler : attempt 1

```
public class TaskImpl implements Task{
 private int id;
 public TaskImpl(int id){
  this.id = id;
 }
 public void run(){
  System.out.println(”Task ” + id + “ is running!”);
 }
}
```

Delayed task scheduler : attempt 1

```
public class SchedulerImpl implements Scheduler{
 public void schedule(Task t, long delayMs){…}
 public static void main(String[] args){
  Scheduler scheduler = new SchedulerImpl();
  Task t1 = new TaskImpl(1);
  Task t2 = new TaskImpl(2);
  scheduler.schedule(t1,10000);
  scheduler.schedule(t2,1);
 }
}
```

Delayed task scheduler: attempt 1

```
public class SchedulerImpl implements Scheduler{
 public void schedule(Task t, long delayMs){…}
 public static void main(String[] args){
  Scheduler scheduler = new SchedulerImpl();
  Task t1 = new TaskImpl(1);
  Task t2 = new TaskImpl(2);
  scheduler.schedule(t1,10000);
  scheduler.schedule(t2,1);
 }
}
```

Main thread in Timedwaiting state for 10000ms

Delayed task scheduler : attempt 1

```
public class SchedulerImpl implements Scheduler{
 public void schedule(Task t, long delayMs){…}
 public static void main(String[] args){
  Scheduler scheduler = new SchedulerImpl();
  Task t1 = new TaskImpl(1);
  Task t2 = new TaskImpl(2);
  scheduler.schedule(t1,10000);
  scheduler.schedule(t2,1);
 }
}
```

Task t2 will be delayed 10001ms

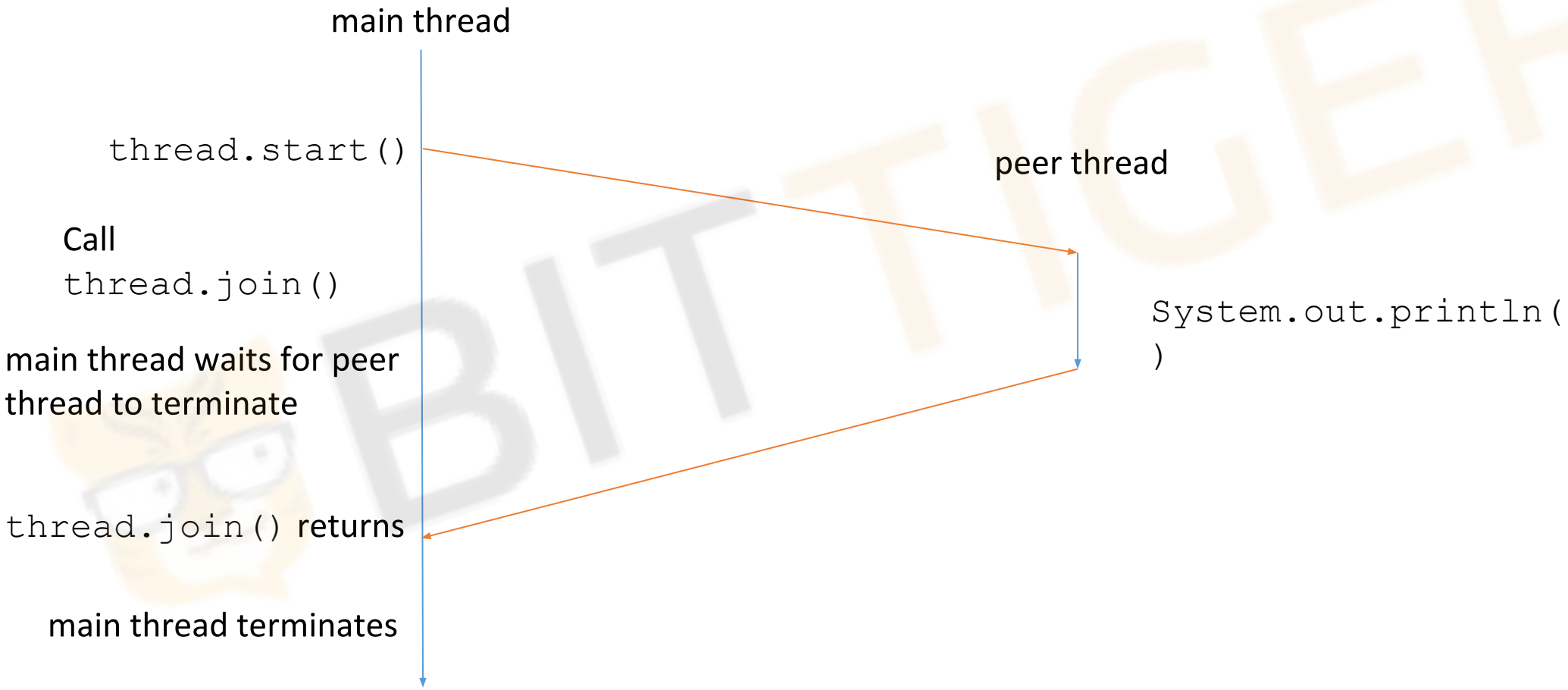
Delayed task scheduler : attempt 1

- Main thread is in Timedwaiting state for delayMs for each call of **`schedule().`**
- Only one thread, very low CPU utilization.
- Also, this is not working as later call
- How about sleeping in other threads?

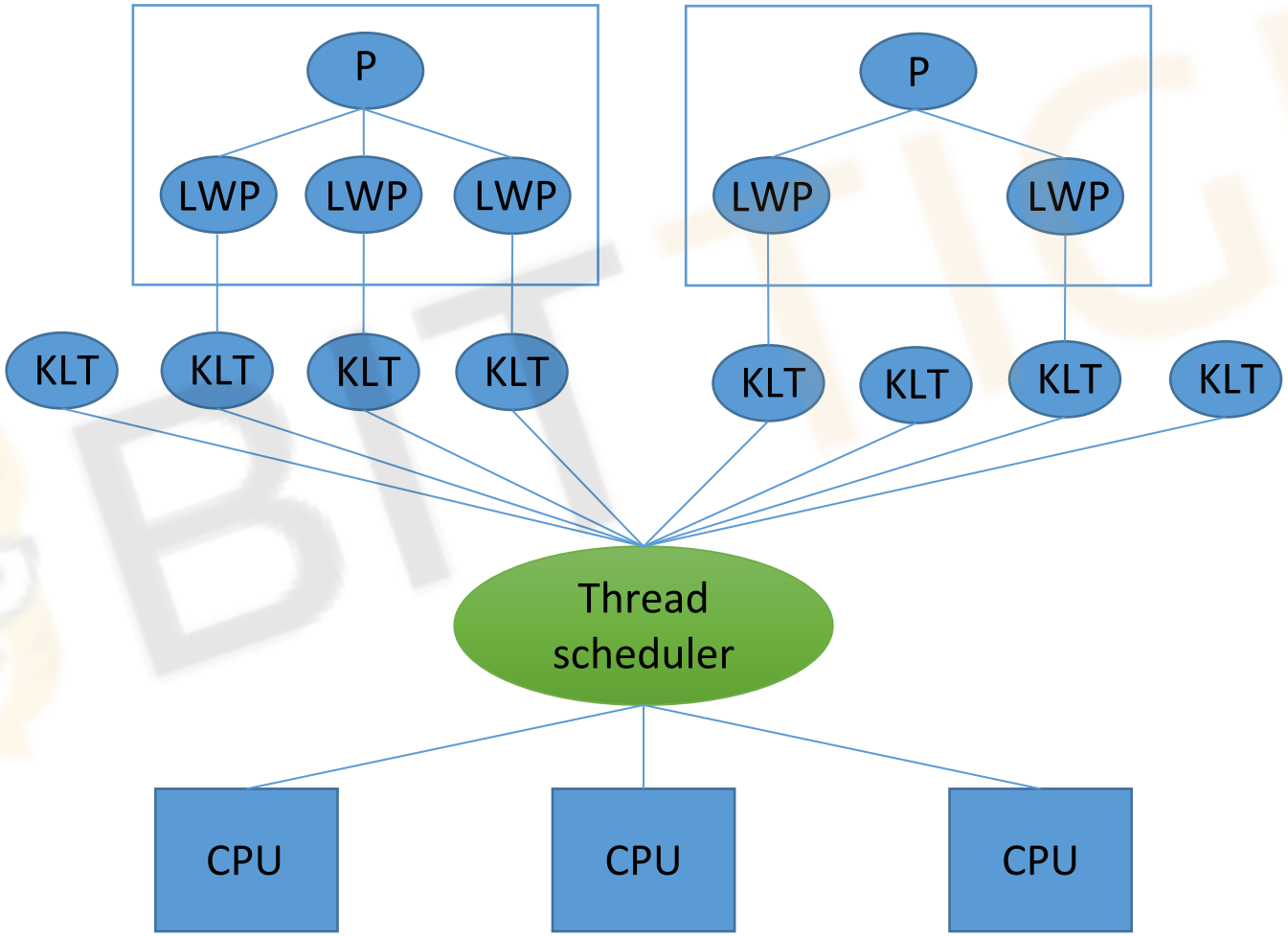
Java Threads

- Java Threads
  - Thread class
  - Runnable interface
- Thread creation
  - Override Thread class
  - Implement Runnable interface
- Hello World

Execution of threaded “Hello world”



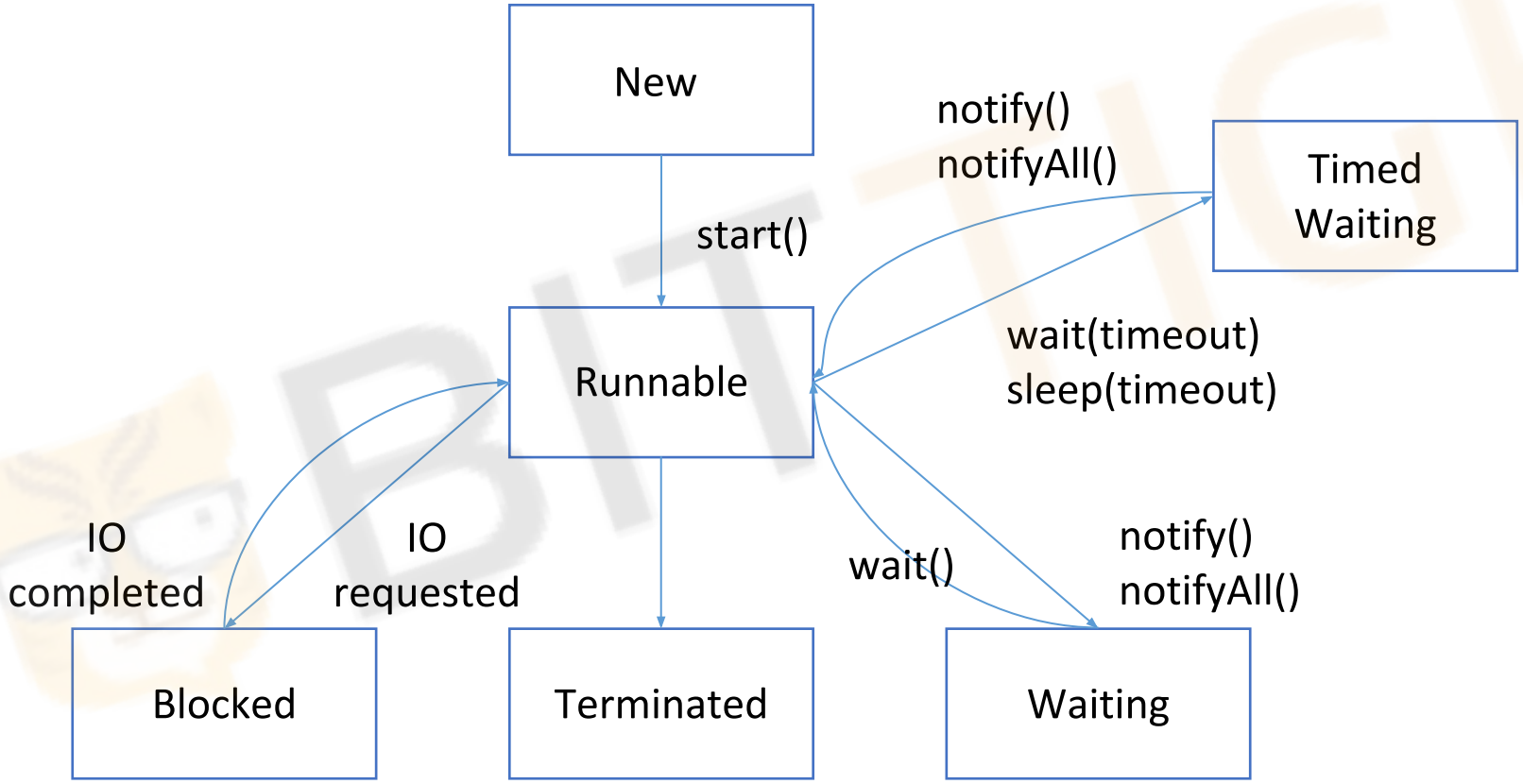
Threads implementation



Java thread states

- Java threads have 6 states, at any point of time, each thread can only be in one of the states
  - New
  - Runnable
  - Waiting
  - Timed waiting
  - Blocked
  - Terminated

Java thread lifecycle



Java thread lifecycle

```
Thread t = new Thread(); // New
t.start(); // Runnable
//Become running if scheduled by OS
Thread.sleep(1000); // Timed waiting
wait(); //Waiting
System.out.println(“Hello world”); //Blocked
```

Java threads summary

- Thread class and Runnable interface.
- Can use anonymous class to simplify thread creation.
- Java thread states: New, Runnable, Blocked, Waiting, Timed waiting, Terminated.
- A thread is in blocked state when performing IO. When blocked, the thread gives the CPU to other threads.
- In Linux, Java threads are mapped to kernel level threads (KLT) and scheduled by OS.

Synchronization

- Safety
  - Nothing bad happens ever
- Liveness
  - Something good happens eventually

Mutex

- Synchronized method

```
public class A {
   public synchronized void foo() {…}
   public synchronized void bar() {…}
   public void foobar() {…}
}
A a1 = new a1();
T1: a1.foo();
T2: a1.bar(); // T2 is blocked
T3: a1.foobar(); T3 is not blocked
```

Mutex

- Synchronized code block
  - `synchronized`(this) {..};
  - Object lock = new Object(); `synchronized`(lock) {…};
  - class A {..}; `synchronized`(A.class) {…};
- Only one thread can access the synchronized block, other threads will be blocked.

Mutex

- Synchronized method/block is reentrant to the same thread.
- Other threads are blocked.
- Java threads maps to kernel threads, so the kernel is in charge of thread state transition.
- user mode -> kernel mode, so synchronized is heavy weight lock.
- Changes in a synchronized method/block are made visible to other threads when exiting the synchronized block.

Mutex

- ```
  class A {
      public void foo();
    }
  ```
- `A a1 = new A();`
- `A a2 = new A();`
- `a1.foo(); // T1`
- `a2.foo(); // T2 blocked`
- We want to make sure that that only one thread can access foo() all all instances of A.
  ```

Reentrant lock

- `synchronized` is equivalent to reentrant lock

```
Lock lock = new ReentrantLock();
try {
   lock.lock();
   do something
} finally {
   lock.unlock();
}
```

Condition variable

- A condition variable is an explicit queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition).
- some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue.
- `wait()`
- `wait(long timeout)`
- `notify()`
- `notifyAll()`

Condition variable

- Conditional variable should be used inside a synchronized block.
- In Java, each object is associated with one and only one condition variable.
- This is a restriction, you may need to use explicit locking to get around this.

Condition variable

- Multiple condition variables can be associated with a reentrant lock.
- This is useful for implementing blocking queue.
- More about this in the next lecture.

Volatile variables

- Volatile semantics
  - Changes to volatile variables are visible to other threads immediately.
  - Volatile eliminate instruction reordering.
- Volatile use cases
  - Single writer.
  - The result doesn’t depend on the current value.

Atomic variables

- For an atomic operation, all threads either see the value of before assignment, or the value after assignment.
- Compare and swap (CAS)

Atomic variables

- ```
  public final int incrementAndGet() {
      for(;;) {
      int current = get();
      int next = current + 1;
      if (compareAndSet(current, next))
          return next;
      }
  }
  ```
- ABA problem
  - A variable is value A when when first read, and stays at A when assigning.
  - A -> B -> A
  ```

Delayed task scheduler : attempt 2

```
public class SchedulerImpl implements Scheduler{
 public void schedule(Task t, long delayMs){
  Thread t = new Thread(new Runnable(){
   public void run(){
    try{
     Thread.sleep(delayMs);
     t.run();
    }catch(InterruptedException e){
     //ignore
    }
   }
  };
  t.start();
 }
}
```

Delayed task scheduler : attempt 2

```
public class SchedulerImpl implements Scheduler{
 public void schedule(Task t, long delayMs){
  Thread t = new Thread(new Runnable(){
   public void run(){
    try{
     Thread.sleep(delayMs);
     t.run();
    }catch(InterruptedException e){
     //ignore
    }
   }
  };
  t.start();
 }
}
```

Create a new thread for each schedule call

Delayed task scheduler : attempt 2

- No blocking when calling schedule()
- What happens if we call schedule many times?
- A lot of thread creation overhead.
- Can be alleviated by using a thread pool, but still not ideal.

Delayed task scheduler: attempt 3

- Use a priority queue and a background thread.

Task

```
class Task implements Comparable<Task>{
 private long timeToRun;
 private int id;
 public void run(){
  System.out.println("Running task " + id);
 }
 public int compareTo(Task other){
  return (int) (timeToRun - other.getTimeToRun());
 }
 …
}
```

Task

```
class Task implements Comparable<Task>{
 private long timeToRun;
 private int id;
 public void run(){
  System.out.println("Running task " + id);
 }
 public int compareTo(Task other){
  return (int) (timeToRun - other.getTimeToRun());
 }
 …
}
```

The time to run the task

Task

```
class Task implements Comparable<Task>{
 private long timeToRun;
 private int id;
 public void run(){
  System.out.println("Running task " + id);
 }
 public int compareTo(Task other){
  return (int) (timeToRun - other.getTimeToRun());
 }
 …
}
```

Task id

Task

```
class Task implements Comparable<Task>{
 private long timeToRun;
 private int id;
 public void run(){
  System.out.println("Running task " + id);
 }
 public int compareTo(Task other){
  return (int) (timeToRun - other.getTimeToRun());
 }
 …
}
```

Dummy run method

Task

```
class Task implements Comparable<Task>{
 private long timeToRun;
 private int id;
 public void run(){
  System.out.println("Running task " + id);
 }
 public int compareTo(Task other){
  return (int) (timeToRun - other.getTimeToRun());
 }
 …
}
```

Order tasks by time to run

Scheduler: fields

```
public class Scheduler{
 private PriorityQueue<Task> tasks;
 private final Thread taskRunnerThread;
 private volatile boolean running;
 private final AtomicInteger taskId;
 public Scheduler(){…}
 public void schedule(Task task, long delayMs){…}
 public void stop(){…}
 private class TaskRunner implements Runnable{…}
}
```

Scheduler: fields

```
public class Scheduler{
 private PriorityQueue<Task> tasks;
 private final Thread taskRunnerThread;
 private volatile boolean running;
 private final AtomicInteger taskId;
 public Scheduler(){…}
 public void schedule(Task task, long delayMs){…}
 public void stop(){…}
 private class TaskRunner implements Runnable{…}
}
```

Priority queue to order tasks by time to run

Scheduler: fields

```
public class Scheduler{
 private PriorityQueue<Task> tasks;
 private final Thread taskRunnerThread;
 private volatile boolean running;
 private final AtomicInteger taskId;
 public Scheduler(){…}
 public void schedule(Task task, long delayMs){…}
 public void stop(){…}
 private class TaskRunner implements Runnable{…}
}
```

Background thread to run tasks

Scheduler: fields

```
public class Scheduler{
 private PriorityQueue<Task> tasks;
 private final Thread taskRunnerThread;
 private volatile boolean running;
 private final AtomicInteger taskId;
 public Scheduler(){…}
 public void schedule(Task task, long delayMs){…}
 public void stop(){…}
 private class TaskRunner implements Runnable{…}
}
```

State indicating the scheduler is running

Scheduler: fields

```
public class Scheduler{
 private PriorityQueue<Task> tasks;
 private final Thread taskRunnerThread;
 private volatile boolean running;
 private final AtomicInteger taskId;
 public Scheduler(){…}
 public void schedule(Task task, long delayMs){…}
 public void stop(){…}
 private class TaskRunner implements Runnable{…}
}
```

Task id to assign to submitted tasks

Scheduler: constructor

```
public class Scheduler{
 public Scheduler(){
  tasks = new PriorityQueue<>();
  taskRunnerThread = new Thread(new TaskRunner());
  running = true;
  taskId = new AtomicInteger(0);
  taskRunnerThread.start();
 }
 public void schedule(Task task, long delayMs){…}
 public void stop(){…}
 private class TaskRunner implements Runnable{…}
}
```

Scheduler: constructor

```
public class Scheduler{
 public Scheduler(){
  tasks = new PriorityQueue<>();
  taskRunnerThread = new Thread(new TaskRunner());
  running = true;
  taskId = new AtomicInteger(0);
  taskRunnerThread.start();
 }
 public void schedule(Task task, long delayMs){…}
 public void stop(){…}
 private class TaskRunner implements Runnable{…}
}
```

Create the background task runner

Scheduler: constructor

```
public class Scheduler{
 public Scheduler(){
  tasks = new PriorityQueue<>();
  taskRunnerThread = new Thread(new TaskRunner());
  running = true;
  taskId = new AtomicInteger(0);
  taskRunnerThread.start();
 }
 public void schedule(Task task, long delayMs){…}
 public void stop(){…}
 private class TaskRunner implements Runnable{…}
}
```

Set running to true

Scheduler: constructor

```
public class Scheduler{
 public Scheduler(){
  tasks = new PriorityQueue<>();
  taskRunnerThread = new Thread(new TaskRunner());
  running = true;
  taskId = new AtomicInteger(0);
  taskRunnerThread.start();
 }
 public void schedule(Task task, long delayMs){…}
 public void stop(){…}
 private class TaskRunner implements Runnable{…}
}
```

Initialize task id

Scheduler: constructor

```
public class Scheduler{
 public Scheduler(){
  tasks = new PriorityQueue<>();
  taskRunnerThread = new Thread(new TaskRunner());
  running = true;
  taskId = new AtomicInteger(0);
  taskRunnerThread.start();
 }
 public void schedule(Task task, long delayMs){…}
 public void stop(){…}
 private class TaskRunner implements Runnable{…}
}
```

Start task runner thread

Scheduler: schedule()

```
public class Scheduler{
 public Scheduler(){…}
 public void schedule(Task task, long delayMs){
  long timeToRun = System.currentTimeMillis()+ delayMs;
  task.setTimeToRun(timeToRun);
  task.setId(taskId.incrementAndGet());
  synchronized(this){
   tasks.offer(task);
   this.notify();
  }
 }
 public void stop(){…}
 private class TaskRunner implements Runnable{…}
}
```

Scheduler: schedule()

```
public class Scheduler{
 public Scheduler(){…}
 public void schedule(Task task, long delayMs){
  long timeToRun = System.currentTimeMillis()+ delayMs;
  task.setTimeToRun(timeToRun);
  task.setId(taskId.incrementAndGet());
   synchronized(this){
    tasks.offer(task);
    this.notify();
   }
 }
 public void stop(){…}
 private class TaskRunner implements Runnable{…}
}
```

Set time to run and assign task id

Scheduler: schedule()

```
public class Scheduler{
 public Scheduler(){…}
 public void schedule(Task task, long delayMs){
  long timeToRun = System.currentTimeMillis()+ delayMs;
  task.setTimeToRun(timeToRun);
  task.setId(taskId.incrementAndGet());
  synchronized(this){
   tasks.offer(task);
   this.notify();
  }
 }
 public void stop(){…}
 private class TaskRunner implements Runnable{…}
}
```

Put the task in queue

Scheduler: stop()

```
public class Scheduler{
 public Scheduler(){…}
 public void schedule(Task task, long delayMs){…}
 public void stop(){
  synchronized(this){
   running = false;
   this.notify();
  }
  taskRunnerThread.join();
 }
 private class TaskRunner implements Runnable{…}
}
```

Scheduler: stop()

```
public class Scheduler{
 public Scheduler(){…}
 public void schedule(Task task, long delayMs){…}
 public void stop(){
  synchronized(this){
   running = false;
   this.notify();
  }
   taskRunnerThread.join();
 }
 private class TaskRunner implements Runnable{…}
}
```

Notify the task runner as it may be in wait()

Scheduler: stop()

```
public class Scheduler{
 public Scheduler(){…}
 public void schedule(Task task, long delayMs){…}
 public void stop(){
  synchronized(this){
   running = false;
   this.notify();
  }
  taskRunnerThread.join();
 }
 private class TaskRunner implements Runnable{…}
}
```

Wait for the task runner to terminate

Scheduler: TaskRunner

```
public class Scheduler{
 public Scheduler(){…}
 public void schedule(Task task, long delayMs){…}
 public void stop(){…}
 private class TaskRunner implements Runnable{…}
}
```

TaskRunner

```
private class TaskRunner implements Runnable{
 public void run(){
  while(running){
   synchornized(Scheduler.this){
    while(running&&tasks.isEmpty()){Scheduler.this.wait();}
    long now = System.currentTimeMillis();
    Task t = tasks.peek();
    if(t.getTimeToRun() < now){
     tasks.poll();
     t.run();
    }else{
     Scheduler.this.wait(t.getTimeToRun()-now);
    }
   }
  }
 }
}
```

TaskRunner

```
private class TaskRunner implements Runnable{
 public void run(){
  while(running){
   synchornized(Scheduler.this){
    while(running&&tasks.isEmpty()){Scheduler.this.wait();}
    long now = System.currentTimeMillis();
    Task t = tasks.peek();
    if(t.getTimeToRun() < now){
     tasks.poll();
     t.run();
    }else{
     Scheduler.this.wait(t.getTimeToRun()-now);
    }
   }
  }
 }
}
```

Inner class of Scheduler, implements Runnable

TaskRunner

```
private class TaskRunner implements Runnable{
 public void run(){
  while(running){
    synchornized(Scheduler.this){
     while(running&&tasks.isEmpty()){Scheduler.this.wait();}
     long now = System.currentTimeMillis();
     Task t = tasks.peek();
     if(running&&t.getTimeToRun() < now){
      tasks.poll();
      t.run();
     }else{
      Scheduler.this.wait(t.getTimeToRun()-now);
     }
    }
  }
 }
}
```

The task runner keeps running until stopped from main thread

TaskRunner

```
private class TaskRunner implements Runnable{
 public void run(){
  while(running){
   synchornized(Scheduler.this){
    while(running&&tasks.isEmpty()){Scheduler.this.wait();}
    long now = System.currentTimeMillis();
    Task t = tasks.peek();
    if(running&&t.getTimeToRun() < now){
     tasks.poll();
     t.run();
    }else{
     Scheduler.this.wait(t.getTimeToRun()-now);
    }
   }
  }
 }
}
```

Need to synchronize with main thread

TaskRunner

```
private class TaskRunner implements Runnable{
 public void run(){
  while(running){
   synchornized(Scheduler.this){
    while(running&&tasks.isEmpty()){Scheduler.this.wait();}
    long now = System.currentTimeMillis();
    Task t = tasks.peek();
    if(running&&t.getTimeToRun() < now){
     tasks.poll();
     t.run();
    }else{
     Scheduler.this.wait(t.getTimeToRun()-now);
    }
   }
  }
 }
}
```

Task runner is blocked when no tasks in queue

TaskRunner

```
private class TaskRunner implements Runnable{
 public void run(){
  while(running){
   synchornized(Scheduler.this){
    while(running&&tasks.isEmpty()){Scheduler.this.wait();}
    long now = System.currentTimeMillis();
    Task t = tasks.peek();
    if(running&&t.getTimeToRun() < now){
     tasks.poll();
     t.run();
    }else{
     Scheduler.this.wait(t.getTimeToRun()-now);
    }
   }
  }
 }
}
```

Check the first task in queue

TaskRunner

```
private class TaskRunner implements Runnable{
 public void run(){
  while(running){
   synchornized(Scheduler.this){
    while(running&&tasks.isEmpty()){Scheduler.this.wait();}
    long now = System.currentTimeMillis();
    Task t = tasks.peek();
    if(running&&t.getTimeToRun() < now){
     tasks.poll();
     t.run();
    }else{
     Scheduler.this.wait(t.getTimeToRun()-now);
    }
   }
  }
 }
}
```

Delay exhausted, execute task

TaskRunner

```
private class TaskRunner implements Runnable{
 public void run(){
  while(running){
   synchornized(Scheduler.this){
    while(running&&tasks.isEmpty()){Scheduler.this.wait();}
    long now = System.currentTimeMillis();
    Task t = tasks.peek();
    if(running&&t.getTimeToRun() < now){
     tasks.poll();
     t.run();
    }else{
     Scheduler.this.wait(t.getTimeToRun()-now);
    }
   }
  }
 }
}
```

No task executable, wait

The wait(timeout) method

```
q.wait(timeout);
```

- Releases lock on q
- Sleeps (gives up processor)
- Awakens (resumes running) by notify() and notifyAll()
- Awakens after timeout
- Reacquires lock & returns

Why wait(timeout)?

```
Scheduler scheduler = new Scheduler();
scheduler.schedule(new Task(),1000000);
scheduler.schedule(new Task(),1000);
```

Delayed task scheduler summary

- Single threaded implementation is not working because main the thread is blocked for each **`schedule()`** call.
- Creating a new thread for each **`schedule()`** call is not scalable with system resources.
- Use a priority queue and background task runner thread.
- sleep(timeout) vs wait(timeout)