Robust and Scalable Concurrent Programming: Lessons from the Trenches

Sangjin Lee, Mahesh Somani, & Debasish Saha

eBay Inc.
Agenda

- Why Does It Matter?
- Patterns: the Good, the Bad, and the $#&%@!
- Lessons Learned
Why Does It Matter?

Concurrency is becoming only more important

- Multi-core systems are becoming the norm of enterprise servers
- Your enterprise software needs to run under high concurrency to take advantage of hardware
  - There are other ways to scale (virtualization or running multiple processes), but it is not always doable
Why Does It Matter?

> Writing safe and concurrent code is hard
  
  • Thread safety is hard for an average Java developer
    
    • Many still don’t fully understand the Java Memory Model
    • Thread safety is often an afterthought
    • Shared data is still the primary mechanism of handling concurrency in Java
      
      • Other approaches: Actor-based *share-nothing* concurrency (Scala), STM, etc.
  
  • Thread safety concerns are not easily isolated
Why Does It Matter?

> Writing safe and concurrent code is hard
  • Writing safe and scalable code is even harder
    • It requires understanding of your software under concurrency
      • Will this lock become hot under our production traffic usage?
      • You should tune *only if* it is shown to be a real hot spot
    • It requires knowledge of what scales and what doesn’t
      • Synchronized HashMap vs. ConcurrentHashMap?
      • When does ReadWriteLock make sense?
      • Atomic variables?
Why Does It Matter?

> Yet, the penalty for incorrect or non-scalable code is severe

- They are often the most difficult bugs: hard to reproduce, and happen only under load (a.k.a. your production peak time)
- The code works just fine under light load, but it blows up in your face as the traffic peaks
Patterns

- Lazy Initialization
Patterns
[1] Lazy Initialization

> What’s wrong with this picture?

```java
public class Singleton {
    private static Singleton instance;

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    private Singleton() {}  

    public class Singleton {
        private static Singleton instance;

        public static Singleton getInstance() {
            if (instance == null) {
                instance = new Singleton();
            }
            return instance;
        }

        private Singleton() {}
    }
```
Patterns

[1] Lazy Initialization

> Not thread safe of course! Let’s fix it.

```java
public class Singleton {
    private static Singleton instance;

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    private Singleton() {}
}
```
Patterns

[1] Lazy Initialization

- Fix #1: synchronize!

```java
public class Singleton {
    private static Singleton instance;

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    private Singleton() {}  
}
```
Patterns
[1] Lazy Initialization

> It is now correct (thread safe) but ...
> We have just introduced a potential lock contention
> It can pose a serious scalability issue if it is in the critical path

- “The principal threat to scalability in concurrent applications is the exclusive resource lock.” – Brian Goetz, Java Concurrency in Practice
Patterns

[1] Lazy Initialization

> Impact of a lock contention
Patterns
[1] Lazy Initialization

Fix #2 (bad): synchronize only when we allocate

```java
public class Singleton {
    private static Singleton instance;

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }

    private Singleton() {}
}
```
Patterns

[1] Lazy Initialization

- There is a name for this pattern 😊
- Optimization gone awry: it appears correct and efficient, but it is **not thread safe**
Fix #3: $#$&%@!

```java
public class Singleton {
    private static Singleton instance;
    private static ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

    public static Singleton getInstance() {
        lock.readLock().lock();
        try {
            if (instance == null) {
                lock.readLock().unlock();
                lock.writeLock().lock();
                try {
                    if (instance == null) {
                        instance = new Singleton();
                    }
                } finally {
                    lock.readLock().lock();
                    lock.writeLock().unlock();
                }
            }
        } finally {
            lock.readLock().unlock();
        }
        return instance;
    }

    private Singleton() {}
}
```
Patterns

[1] Lazy Initialization

> None of the fixes quite works: they are either unsafe, or don’t scale if in the critical path

> Then what is a better fix?
Patterns

[1] Lazy Initialization

> Eager Initialization!

• No reason to initialize lazily: allocations are cheap
• Lazy initialization was popular when allocations used to be expensive
• Today we do it simply by habit for no good reason
• Static initialization takes care of thread safety, and is also lazy
• For singletons, you gain nothing by using an explicit lazy initialization
Patterns

[1] Lazy Initialization

> **Good fix #1**: eager initialization

```java
public class Singleton {
    private static final Singleton instance = new Singleton();

    public static Singleton getInstance() {
        return instance;
    }

    private Singleton() {} // Private constructor
}
```
Patterns

[1] Lazy Initialization

> Then when *does* lazy initialization make sense?

- (For a non-singleton) If instantiation is truly expensive, *and* it has a chance of not being needed
- If you want to retry if it fails the first time
- If you want to unload and reload the object
- To reduce confusion if there was an exception during eager initialization
  - ExceptionInInitializerError for the first call
  - NoClassDefFoundError for subsequent calls
Patterns
[1] Lazy Initialization

> What are the right lazy initialization patterns?
[1] Lazy Initialization

> **Good fix #2**: holder pattern (for non-singletons)

```java
public class Foo {
    public static Bar getBar() {
        return BarHolder.getInstance(); // lazily initialized
    }

    private static class BarHolder {
        private static final Bar instance = new Bar();

        static Bar getInstance() {
            return instance;
        }
    }
}
```
Patterns

[1] Lazy Initialization

> **Good fix #3: corrected** double-checked locking (only on Java 5 or later)

```java
public class Singleton {
    private static volatile Singleton instance;

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }

    private Singleton() {}
}
```
Patterns

- Lazy Initialization
- Map & Compound Operation
What’s wrong with this picture?

```java
public class KeyedCounter {
    private Map<String,Integer> map = new HashMap<String,Integer>();

    public void increment(String key) {
        Integer old = map.get(key);
        int value = (old == null) ? 1 : old.intValue()+1;
        map.put(key, value);
    }

    public Integer getCount(String key) {
        return map.get(key);
    }
}
```
It may be blazingly fast but ... it’s not thread safe!

```java
public class KeyedCounter {
    private Map<String,Integer> map = new HashMap<String,Integer>();

    public void increment(String key) {
        Integer old = map.get(key);
        int value = (old == null) ? 1 : old.intValue()+1;
        map.put(key, value);
    }

    public Integer getCount(String key) {
        return map.get(key);
    }
}
```
Patterns

> “How bad can it get?” Really bad.
  - ConcurrentModificationException may be your best outcome
  - Worse, you might end up in an infinite loop
  - Worse yet, it may simply give you a wrong result without any errors; data corruption, etc.

> Don’t play with fire: *never* modify an unsynchronized collection concurrently
Fix #1 (bad): synchronize the map

```java
public class KeyedCounter {
    private Map<String, Integer> map =
        Collections.synchronizedMap(new HashMap<String, Integer>());

    public void increment(String key) {
        Integer old = map.get(key);
        int value = (old == null) ? 1 : old.intValue() + 1;
        map.put(key, value);
    }

    public Integer getCount(String key) {
        return map.get(key);
    }
}
```
Patterns

> **Fix #1 (bad):** synchronize the map
  
  - It is still not thread safe!
  - Even though individual operations (get() and put()) are thread safe, the compound operation (KeyedCounter.increment()) is *not*
Fix #2: synchronize the operations

```java
public class KeyedCounter {
    private Map<String,Integer> map = new HashMap<String,Integer>();

    public synchronized void increment(String key) {
        Integer old = map.get(key);
        int value = (old == null) ? 1 : old.intValue()+1;
        map.put(key, value);
    }

    public synchronized Integer getCount(String key) {
        return map.get(key);
    }
}
```
Fix #2: synchronize the operations
  - It is now correct
  - But does it scale?
  - As increment() and getCount() get called from multiple threads, they may become a source of lock contention
Patterns


> **ConcurrentHashMap** comes to the rescue!

- In general it scales significantly better than a synchronized HashMap under concurrency
  - Full reader concurrency
  - Some writer concurrency with finer-grained locking

- The ConcurrentMap interface supports additional thread safe methods; e.g. putIfAbsent()
Patterns

> **Good fix**: ConcurrentHashMap with atomic variables

```
public class KeyedCounter {
    private final ConcurrentHashMap<String, AtomicInteger> map =
        new ConcurrentHashMap<String, AtomicInteger>();

    public void increment(String key) {
        AtomicInteger value = new AtomicInteger(0);
        AtomicInteger old = map.putIfAbsent(key, value);
        if (old != null) { value = old; }
        value.incrementAndGet(); // increment the value atomically
    }

    public Integer getCount(String key) {
        AtomicInteger value = map.get(key);
        return (value == null) ? null : value.get();
    }
}
```
Patterns

> Lazy Initialization
> Map & Compound Operation
> Other People’s Money Classes
  > Unsafe classes
Patterns
[3] Other People’s Classes #1

What’s wrong with this picture?

```java
public class DateFormatter {
    private static final DateFormat df = DateFormat.getInstance();

    public static String formatDate(Date d) {
        return df.format(d);
    }
}
```
Patterns

[3] Other People’s Classes #1

> **DateFormat is not thread safe!**

> **Some JDK classes that are not thread safe**
  * DateFormat: declared in javadoc as not thread safe
  * MessageDigest: no mention of thread safety
  * CharsetEncoder/CharsetDecoder: declared in javadoc as not thread safe

> **Lessons**
  * Do not assume thread safety of others’ classes
  * Even javadoc may not have a full disclosure
Patterns
[3] Other People’s Classes #1

> Fix #1: synchronize!
  
  - It is now safe, but has a potential of turning into a lock contention

```java
public class DateFormatter {
    private static final DateFormat df = DateFormat.getInstance;

    public static String formatDate(Date d) {
        synchronized (df) {
            return df.format(d);
        }
    }
}
```
Patterns
[3] Other People’s Classes #1

> **Good fix #2**: allocate it every time
  
  - It is completely safe (stack confinement via local variable)
  
  - Allocations are not as expensive as you might think

```java
public class DateFormatter {
    public static String formatDate(Date d) {
        DateFormat df = DateFormat.getInstance();
        return df.format(d);
    }
}
```
Patterns
[3] Other People’s Classes #1

> **Good fix #3**: use ThreadLocal
  
  - Use it if truly concerned about allocation cost

```java
public class DateFormatter {
    private static final ThreadLocal<DateFormat> tl =
        new ThreadLocal<DateFormat>() {
            @Override protected DateFormat initialValue() {
                return DateFormat.getInstance();
            }
        };

    public static String formatDate(Date d) {
        return tl.get().format(d);
    }
}
```
Patterns

> Lazy Initialization
> Map & Compound Operation
> **Other People’s Classes**
  - Unsafe classes
  - Hidden costs
What's wrong with this picture?

```java
public class ForceInit {
    public static void init(String className) {
        try {
            // Force initialization
            Class.forName(className);
        } catch (ClassNotFoundException e) {
            // May happen.  Do nothing.
        }
    }
}
```
Patterns
[3] Other People’s Classes #2

> Thread safe … but potential lock contention when missing class

```java
public class ForceInit {

    public static void init(String className) {
        try {
            // Force initialization
            Class.forName(className);
        } catch (ClassNotFoundException e) {
            // May happen. Do nothing.
        }
    }

}
```
Patterns

[3] Other People’s Classes #2

> Class.forName()
  • Positive results cached but not negative ones
  • When class not found, (expensive) search in classpath
  • Class loading deals with different locks: repeated invocation => lock contention and CPU hot spot!

> Lessons
  • Measure contention: positive and negative tests
  • Beware as javadoc may not have disclosure on locks or inner workings
Patterns

[3] Other People’s Classes #2

> **Fix**: trading memory for speed. Works when limited range of class names

```java
public class ForceInit {
    private static final ConcurrentHashMap<String, String> cache = new ConcurrentHashMap<String, String>();

    public static void init(String className) {
        try {
            if (cache.putIfAbsent(className, className) == null) {
                // Force initialization
                Class.forName(className);
            }
        } catch (ClassNotFoundException e) {
            // May happen. Do nothing.
        }
    }
}
```
Patterns

> Lazy Initialization
> Map & Compound Operation
> Other People’s Classes
  • Unsafe classes
  • Hidden costs
  • Hidden costs #2
What's wrong with this picture?

```java
public class XMLParser {
    public void parseXML(InputStream is, DefaultHandler handler) throws Exception {
        SAXParser parser = SAXParserFactory.newInstance().newSAXParser();
        parser.parse(is, handler);
    }
}
```
Patterns
[3] Other People’s Classes #3

- Depends … newSAXParser() could have a potentially significant lock contention

```java
public class XMLParser {
    public void parseXML(InputStream is, DefaultHandler handler) throws Exception {
        SAXParser parser = SAXParserFactory.newInstance().newSAXParser();
        parser.parse(is, handler);
    }
}
```
Patterns
[3] Other People’s Classes #3

> But why?
  • Implementation class determined in following order
    • System property → lib/jaxp.properties → META-INF/services/jaxp… in class path → Platform default
  • META-INF not cached! Possible lock contention

> Also applies to DOM & XML Reader

> JAXP implementations (e.g., Xerces) may have additional configurations read from classpath

> Lessons
  • Measure contention
  • Read documentation!
Patterns

[3] Other People’s Classes #3

> **Good fix #1**: Set up environment (system properties)

$ java -Djavax.xml.parsers...=IMPL_CLASS ... MAIN_CLASS [args]
Patterns

[3] Other People’s Classes #3

Good fix #2: ThreadLocal

- Beware reuse issues however

```java
public class XMLParser {
    private static final ThreadLocal<SAXParser> tlSax = new ThreadLocal<SAXParser>();

    private static SAXParser getParser() throws Exception {
        SAXParser parser = tlSax.get();
        if (parser == null) {
            parser = SAXParserFactory.newInstance().newSAXParser();
            tlSax.set(parser);
        }
        return parser;
    }

    public void parseXML(InputStream is, DefaultHandler handler) throws Exception {
        getParser().parse(is, handler);
    }
}
```
Patterns

> Lazy Initialization
> Map & Compound Operation
> Other People’s Classes
  • Unsafe classes
  • Hidden costs
  • Hidden costs #2
Lessons Learned: Best Practices

- Patterns & anti-patterns
  - Common problems and solutions
  - “Anti-patterns” are a good way of educating
  - Useful way to analyze solutions
  - Surprisingly repeated use and misuse
Lessons Learned: Best Practices

> Correctness first: thread safety is *non-negotiable*
  - Premature optimization for scalability could risk correctness
  - Concurrency issues are insidious, hard to reproduce and debug
  - Concurrency issues increase nonlinearly with load
Lessons: Not All Code is Created Equal

> Scalability next

- Scalability and lock contention
  - A few locks cause most of contention
  - TPS should increase linearly with utilization
    - Lock contention possible cause if nonlinear behavior
- Ways to determine contended locks
  - Lock analyzers, CPU profiling (oprofile, tprof)
  - Thread dumps a few seconds apart
Lessons: Best Practices

> Multi-pronged attack: *proactive*
  - Regular training and documentation
  - Static code analysis
  - Runtime code analysis
  - Ongoing performance monitoring and measurements

> Multi-pronged attack: *reactive*
  - Periodic tuning
  - Complimentary to programming discipline
  - Be disciplined: tune only the top hot spots
Lessons: Best Practices

> eBay’s Systems Lab

- Lab environment to put applications under microscope with different profiling tools
- On-going tests to systemically identify and fix bottlenecks
- Yielded substantial scalability improvements and server savings
Q & A

> Questions?