7. Liveness and Asynchrony

Prof. O. Nierstrasz
Roadmap

- Asynchronous invocations
- Simple Relays
  - Direct invocations
  - Thread-based messages
  - Command-based messages
- Tail calls
- Early replies
- Futures
- JUC (java.util.concurrent)
Roadmap

> Asynchronous invocations
> Simple Relays
  — Direct invocations
  — Thread-based messages
  — Command-based messages
> Tail calls
> Early replies
> Futures
> JUC (java.util.concurrent)
Pattern: Asynchronous Invocations

**Intent:** Avoid waiting for a request to be serviced by *decoupling sending from receiving.*

**Applicability**

> When a host object can distribute services amongst multiple helper objects.

> When an object does not immediately need the result of an invocation to continue doing useful work.

> When invocations that are logically asynchronous, regardless of whether they are coded using threads.

> During refactoring, when classes and methods are split in order to increase concurrency and reduce liveness problems.
Asynchronous Invocations — template

Asynchronous invocation typically looks like this:

```java
abstract class AbstractHost implements Host {
    public void service() {
        pre();                   // code to run before invocation
        invokeHelper();         // the invocation
        during();               // code to run in parallel
        post();                 // code to run after completion
    }

    // A host provides a service
    public interface Host {
        public void service();
    }
}
```

// A host provides a service
### Asynchronous Invocations — design steps

**Consider the following issues:**

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does the Host need results back from the Helper?</td>
<td>Not if, e.g., the Helper returns results directly to the Host’s caller!</td>
</tr>
<tr>
<td>Can the Host process new requests while the Helper is running?</td>
<td>Might depend on the kind of request ...</td>
</tr>
<tr>
<td>Can the Host do something while the Helper is running?</td>
<td>i.e., in the during() code</td>
</tr>
<tr>
<td>Does the Host need to synchronize pre-invocation processing?</td>
<td>i.e., if service() is guarded or if pre() updates the Host’s state</td>
</tr>
<tr>
<td>Does the Host need to synchronize post-invocation processing?</td>
<td>i.e., if post() updates the Host’s state</td>
</tr>
<tr>
<td>Does post-invocation processing only depend on the Helper’s result?</td>
<td>... or does the host have to wait for other conditions?</td>
</tr>
<tr>
<td>Is the same Helper always used?</td>
<td>Is a new one generated to help with each new service request?</td>
</tr>
</tbody>
</table>

© Oscar Nierstrasz
Roadmap

- Asynchronous invocations
- **Simple Relays**
  - Direct invocations
  - Thread-based messages
  - Command-based messages
- Tail calls
- Early replies
- Futures
- JUC (java.util.concurrent)
A relay method obtains all its functionality by delegating to the helper, without any pre(), during(), or post() actions.

> **Direct invocations:**
  — Invoke the Helper directly, but without synchronization

> **Thread-based messages:**
  — Create a new thread to invoke the Helper

> **Command-based messages:**
  — Pass the request to another object that will run it

Relays are commonly seen in Adaptors.
Roadmap

> Asynchronous invocations
> Simple Relays
  — Direct invocations
  — Thread-based messages
  — Command-based messages
> Tail calls
> Early replies
> Futures
> JUC (java.util.concurrent)
Variant: Direct invocations

```java
public class HostDirectRelay implements Host {
    // NB: Helper is also immutable, so unsynchronized
    protected final Helper helper = new CountingHelper();

    public void service() { // unsynchronized!
        invokeHelper(); // stateless method
    }

    protected void invokeHelper() {
        helper.help(); // unsynchronized!
    }
}
```

Asynchrony is achieved by avoiding synchronization.

The Host is free to accept other requests, while the Host’s caller must wait for the reply.
Direct invocations ...

If `helper` is mutable, it can be protected with an accessor:

```java
public class HostDirectRelaySyncHelper implements Host {
    protected Helper helper;
    public void service() { invokeHelper(); }
    protected void invokeHelper() {
        helper().help(); // partially unsynchronized!
    }
    protected synchronized Helper helper() {
        return helper;
    }
    public synchronized void setHelper(String name) {
        helper = new NamedHelper(name);
    }
}
```
Asynchronous invocations

Simple Relays
- Direct invocations
- Thread-based messages
- Command-based messages

Tail calls
Early replies
Futures
JUC (java.util.concurrent)
Variant: Thread-based messages

The invocation can be performed within a new thread:

```java
public class HostWithHelperThread implements Host {
    ...
    protected void invokeHelper() {
        new Thread() {
            public void run() {
                helper.help();
            }
        }.start();
    }
    ...
}
```
The cost of evaluating `Helper.help()` should outweigh the overhead of creating a thread!

- If the Helper is a *daemon* (loops endlessly)
- If the Helper does *I/O*
- Possibly, if *multiple helper methods* are invoked

Typical application: web servers
Thread-per-message Gateways

The Host may construct a new Helper to service each request.

```java
public class FileIO {
    public void writeBytes(String file, byte[] data) {
        new Thread(new FileWriter(file, data)).start();
    }
    public void readBytes(...) { ... }
}

class FileWriter implements Runnable {
    private String nm_; // hold arguments
    private byte[] d_;
    public FileWriter(String name, byte[] data) { ... }
    public void run() { ... } // write to file ...
}
```
Roadmap

- Asynchronous invocations
- Simple Relays
  - Direct invocations
  - Thread-based messages
  - Command-based messages
- Tail calls
- Early replies
- Futures
- JUC (java.util.concurrent)
Variant: Command-based messages

The Host can also put a *Command object* in a queue for another object that will invoke the Helper:

```java
public class HostEventQueue implements Host {
    ...
    protected void invokeHelper() {
        EventQueue.invokeLater(new Runnable() {
            public void run() { helper.help(); }
        });
    }
}
```

Command-based forms are especially useful for:

> *scheduling* of helpers (i.e., by *pool* of threads)
> *undo and replay* capabilities
> transporting messages over *networks*
Roadmap

> Asynchronous invocations
> Simple Relays
  — Direct invocations
  — Thread-based messages
  — Command-based messages
> Tail calls
> Early replies
> Futures
> JUC (java.util.concurrent)
Tail calls

Applies when the helper method is the last statement of a service. Only pre() code is synchronized.

```java
public class TailCallSubject extends Observable {
    protected Observer observer = new Observer() { ... };
    protected double state;
    public void updateState(double d) {
        doUpdate(d);
        sendNotification();
    }
    protected synchronized void doUpdate(double d) {
        state = d;
    }
    protected void sendNotification() {
        observer.update(this, state);
    }
}
```

NB: The host is immediately available to accept new requests
Alternatively, the tail call may be made in a separate thread:

```java
public synchronized void updateState(double d) {
    state = d;
    new Thread() {
        public void run() {
            observer.update(TailCallSubject.this, state);
        }
    }.start();
}
```
Roadmap

- Asynchronous invocations
- Simple Relays
  - Direct invocations
  - Thread-based messages
  - Command-based messages
- Tail calls
- Early replies
- Futures
- JUC (java.util.concurrent)
Early reply allows a host to perform useful activities \textit{after returning a result} to the client:

\textit{Early reply is a built-in feature in some programming languages. It can be easily simulated when it is not a built-in feature.}
A one-slot buffer can be used to pick up the reply from a helper thread:

A one-slot buffer is a simple abstraction that can be used to implement many higher-level concurrency abstractions...
public class EarlyReplyDemo { ...

public Object service() { // unsynchronized
    final Slot reply = new Slot();
    final EarlyReplyDemo host = this;
    new Thread() { // Helper
        public void run() {
            synchronized (host) {
                reply.put(host.compute());
                host.cleanup(); // retain lock
            }
        }
    }.start();

    return reply.get(); // early reply
} ...
}
Asynchronous invocations

Simple Relays
  - Direct invocations
  - Thread-based messages
  - Command-based messages

Tail calls

Early replies

Futures

JUC (java.util.concurrent)
Futures allow a client to continue in parallel with a host until the future value is needed:
abstract class Future<Result, Argument> {
    private Result result; // initially null

    public Future(final Argument arg) {
        new Thread() {
            public void run() { setResult(computeResult(arg)); }
        }.start();
    }

    abstract protected Result computeResult(Argument arg);

    public synchronized void setResult(Result val) {
        result = val;
        notifyAll();
        return;
    }

    public synchronized Result result() {
        while (result == null) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        return result;
    }
}

Asynchrony
Using Futures in Java

Without special language support, the client must explicitly request a `result()` from the future object.

```java
Future<Integer, Integer> f = new Future<Integer, Integer>(n) {
    protected synchronized Integer computeResult(Integer n) {
        return fibonacci(n);
    }
    // slow, naive algorithm to force long compute times ;-)
    public int fibonacci(int n) {
        if (n<2) { return 1; }
        else { return fibonacci(n-1) + fibonacci(n-2); }
    }
};
int val = f.result();
```
Roadmap

- Asynchronous invocations
- Simple Relays
  - Direct invocations
  - Thread-based messages
  - Command-based messages
- Tail calls
- Early replies
- Futures
- JUC (java.util.concurrent)
Condition Objects

java.util.concurrent

Executors
  — Executor
  — ExecutorService
  — ScheduledExecutorService
  — Callable
  — Future
  — ScheduledFuture
  — Delayed
  — CompletionService
  — ThreadPoolExecutor
  — ScheduledThreadPoolExecutor
  — AbstractExecutorService
  — Executors
  — FutureTask
  — ExecutorCompletionService

Queues
  — BlockingQueue
  — ConcurrentHashMap
  — LinkedBlockingQueue
  — ArrayBlockingQueue
  — SynchronousQueue
  — PriorityBlockingQueue
  — DelayQueue

Concurrent Collections
  — ConcurrentHashMap
  — CopyOnWriteArray{List, Set}

Synchronizers
  — CountDownLatch
  — Semaphore
  — Exchanger
  — CyclicBarrier

Locks: java.util.concurrent.locks
  — Lock
  — Condition
  — ReadWriteLock
  — AbstractQueuedSynchronizer
  — LockSupport
  — ReentrantLock
  — ReentrantReadWriteLock

Atomics: java.util.concurrent.atomic
  — Atomic[Type]
  — Atomic[Type]Array
  — Atomic[Type]FieldUpdater
  — Atomic{Markable, Stampable}Reference
Key Functional Groups

- Executors, Thread pools and Futures
  - Execution frameworks for asynchronous tasking
- Concurrent Collections:
  - Queues, blocking queues, concurrent hash map, ...
  - Data structures designed for concurrent environments
- Locks and Conditions
  - More flexible synchronization control
  - Read/write locks
- Synchronizers: Semaphore, Latch, Barrier
  - Ready made tools for thread coordination
- Atomic variables
  - The key to writing lock-free algorithms
The Executor Framework

> Framework for asynchronous task execution
> Standardize asynchronous invocation
  — Framework to execute `Runnable` and `Callable` tasks
    - `Runnable: void run()`
    - `Callable<V>: V call() throws Exception`
> Separate submission from execution policy
  — Use `anExecutor.execute(aRunnable)`
  — Not new `Thread(aRunnable).start()`
> Cancellation and shutdown support
> Usually created via Executors factory class
  — Configures flexible `ThreadPoolExecutor`
  — Customize shutdown methods, before/after hooks, saturation policies, queuing
Executor

> Decouple submission policy from task execution

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

> Code which submits a task doesn't have to know in what thread the task will run

— Could run in the calling thread, in a thread pool, in a single background thread (or even in another JVM!)

— Executor implementation determines execution policy
  
  – Execution policy controls resource utilization, overload behavior, thread usage, logging, security, etc
  
  – Calling code need not know the execution policy
ExecutorService

> Adds lifecycle management
> ExecutorService supports both graceful and immediate shutdown

```java
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit);
    // ...
}
```

> Useful utility methods too

```java
<T> T invokeAny(Collection<Callable<T>> tasks)
  -Executes the given tasks returning the result of one that completed successfully (if any)

-Others involving Future objects
```
public FutureTask<Integer> service (final int n) {
    FutureTask<Integer> future =
        new FutureTask<Integer> (  
          new Callable<Integer>() {  
            public Integer call() {  
              return new Integer(fibonacci(n));
            }
          }) ;
    new Thread(future).start(); // or use an Executor
    return future;
}
Creating Executors

> Sample Executor implementations from Executors
> newSingleThreadExecutor
    — A pool of one, working from an unbounded queue
> newFixedThreadPool(int N)
    — A fixed pool of N, working from an unbounded queue
> newCachedThreadPool
    — A variable size pool that grows as needed and shrinks when idle
> newScheduledThreadPool
    — Pool for executing tasks after a given delay, or periodically
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
What you should know!

> What general form does an asynchronous invocation take?
> When should you consider using asynchronous invocations?
> In what sense can a direct invocation be “asynchronous”?
> Why (and how) would you use inner classes to implement asynchrony?
> What is “early reply”, and when would you use it?
> What are “futures”, and when would you use them?
> How can implement futures and early replies in Java?
Can you answer these questions?

> Why might you want to increase concurrency on a single-processor machine?
> Why are servers commonly structured as thread-per-message gateways?
> Which of the concurrency abstractions we have discussed till now can be implemented using one-slot-buffers as the only synchronized objects?
> When are futures better than early replies? Vice versa?
License

Attribution-ShareAlike 3.0 Unported

You are free:
  to Share — to copy, distribute and transmit the work
  to Remix — to adapt the work

Under the following conditions:
  Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
  Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Any of the above conditions can be waived if you get permission from the copyright holder.

Nothing in this license impairs or restricts the author's moral rights.

http://creativecommons.org/licenses/by-sa/3.0/