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
Roadmap

- Asynchronous invocations
- Simple Relays
  - Direct invocations
  - Thread-based messages
  - Command-based messages
- Tail calls
- Early replies
- Futures
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();
}
```
Consider the following issues:

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

- Asynchronous invocations
- **Simple Relays**
  - Direct invocations
  - Thread-based messages
  - Command-based messages
- Tail calls
- Early replies
- Futures
Simple Relays — three variants

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
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!
    }
}
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);
    }
}
```
Roadmap

> Asynchronous invocations
> Simple Relays
  — Direct invocations
  — Thread-based messages
  — Command-based messages
> Tail calls
> Early replies
> Futures
Variation: 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();
    }
    ...
}
```
Thread-based messages ...

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*
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
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
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) { // unsynchronized
        doUpdate(d); // partially synchronized
        sendNotification(); // unsynchronized
    }
    protected synchronized void doUpdate(double d) { // synchronized
        state = d;
    }
    protected void sendNotification() { // unsynchronized
        observer.update(this, state);
    }
}
```

NB: The host is immediately available to accept new requests.
Tail calls with new threads

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
Early reply allows a host to perform useful activities after returning a result to the client:

*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
    } ...
}
Roadmap

> Asynchronous invocations
> Simple Relays
  — Direct invocations
  — Thread-based messages
  — Command-based messages
> Tail calls
> Early replies
> Futures
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;
    }
}
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();
```
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?
Liveness and Asynchrony

License

http://creativecommons.org/licenses/by-sa/2.5/

You are free:
• to copy, distribute, display, and perform the work
• to make derivative works
• to make commercial use of the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

• For any reuse or distribution, you must make clear to others the license terms of this work.
• Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.