Optimistic Policies: Trying

- Isolate state into versions
  - E.g. by grouping into a helper class
- Isolate state changes to atomic commit method that swaps in new state
- On method entry
  - Save/record current state
  - Apply action to new state
- Only commit if
  - Action succeeds and current state version is unchanged
- If can’t commit: fail or retry
  - Failures are clean (no side effects)
  - Retry policy is variation of a busy-wait
- Only applicable if actions fully reversible
  - No I/O or thread construction unless safely cancellable
  - All internally called methods must be undoable

Optimistic Techniques

- Variations for recording versions of mutable data:
  - Immutable helper classes
  - Version numbers
  - Transaction Ids
  - Time-stamps
- May be more efficient than guarded waits when:
  - Conflicts are rare and when running on multiple CPUs
- Retries can livelock unless proven wait-free
  - Analog of deadlock in guarded waits
  - Should arrange to fail after a certain time or number of attempts
Optimistic Bounded Counter

```java
public class OptimisticBoundedCounter {
    private final long MIN, MAX;
    private Long count; // MIN <= count <= MAX

    public OptimisticBoundedCounter(long min, long max) {
        MIN = min; MAX = max;
        count = new Long(MIN);
    }
    public long value() { return count().longValue(); }
    public synchronized Long count() { return count; }
    private synchronized boolean commit(Long oldc, Long newc) {
        boolean success = (count == oldc);
        if (success) count = newc;
        return success;
    }
    public void inc() throws InterruptedException{
        for (;;){ // retry-based
            if (Thread.interrupted())
                throw new InterruptedException();
            Long c = count(); // record current state
            long v = c.longValue();
            if (v < MAX && commit(c, new Long(v+1))
                break;
            Thread.yield(); // a good idea in spin loops
        }
    }
    public void dec() { /* symmetrical */}
}
```

Specifying Policies

- Some policies are per-type
  - Optimistic approaches require all methods to conform
- Some policies can be specified per-call
  - Balking vs. Guarding vs. Guarding with time-out
- Options for specifying per-call policy:
  - Extra parameters
    ```java
    void put(Object x, long timeout )
    void put(Object x, boolean balk )
    ```
  - Different name for balking or guarding
    ```java
    boolean tryPut( Object x ) // balking
    void put( Object x ) // guarding
    ```
  - May need different exception signatures
Thread Creation Patterns

- Three general sets of patterns for introducing concurrency:
  - Autonomous loops
    - Establishing independent cyclic behaviour
  - Oneway messages
    - Sending messages without waiting for reply or termination
    - Improves availability of sender object
  - Interactive messages (not covered—see CPJ)
    - Requests that later result in reply or callback messages
    - Allows client to proceed concurrently for a while
  - Most design ideas and semantics stem from active object models

Autonomous Loops

- Simple non-reactive active objects contain a run loop of form:

  ```java
  public void run() {
    while (!Thread.interrupted())
      doSomething();
  }
  ```

- Normally established with a constructor containing:

  ```java
  new Thread(this).start();
  ```

  - Or by a specific start method
  - Perhaps also setting priority and daemon status
- Normally also support other methods called from other threads
  - Requires standard safety measures
- Common Applications
  - Animations, Simulations, Message buffer Consumers, Polling daemons that periodically sense state of world
- This is the basic approach of our web server so far
Oneway Messages

- Conceptually oneway messages are sent with
  - No need for replies
  - No concern about failure (exceptions)
  - No dependence on termination of called method
  - No dependence on order that messages are received
  But may sometimes want to cancel messages or resulting activities
- Once oneway message has been sent, host is ready to accept the next message

Oneway Message Styles

<table>
<thead>
<tr>
<th>Events</th>
<th>Mouse clicks, etc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notifications</td>
<td>Status change alerts, etc</td>
</tr>
<tr>
<td>Postings</td>
<td>Mail messages, stock quotes, etc</td>
</tr>
<tr>
<td>Activations</td>
<td>Applet creation, etc</td>
</tr>
<tr>
<td>Commands</td>
<td>Print requests, repaint requests, etc</td>
</tr>
<tr>
<td>Relays</td>
<td>Chain of responsibility designs, etc</td>
</tr>
</tbody>
</table>

- Some semantic choices
  - **Asynchronous**: Entire message send is independent
    - By far, most common style in reactive applications
  - **Synchronous**: Caller must wait until message is accepted
    - Basis for rendezvous protocols
  - **Multicast**: Message is sent to group of recipients
    - The group might not even have any members
Messages in Java

- **Direct method invocations**
  - Rely on standard call/return mechanics
- **Command strings**
  - Recipient parses then dispatches to underlying method
  - Widely used in client/server systems including HTTP
- **EventObjects** and service codes
  - Recipient dispatches
  - Widely used in GUIs, including AWT
- **Request** objects, asking to perform encoded operation
  - Used in distributed object systems — RMI and CORBA
- **Class** objects (normally via `.class` files)
  - Recipient creates instance of class
  - Used in Java Applet framework
- **Runnable** commands
  - Basis for thread instantiation, mobile code systems

Design Goals for Oneway Messages

- **Safety**
  - Local state changes should be atomic (normally, locked)
    - Typical need for locking leads to main differences vs single-threaded Event systems
  - Safe guarding and failure policies, when applicable
- **Availability**
  - Minimize delay until host can accept another message
- **Flow**
  - The activity should progress with minimal contention
- **Performance**
  - Minimize overhead and resource usage
  - Introducing threads is not always the best solution
    - Consider just issuing open calls
Threads-Per-Message Web Server

- Return to one-shot version of `startServer` but pass each accepted connection to a new thread for processing:

```java
// WebServer14.java
Thread serverThread;

public synchronized void startServer() throws ...
{ ...
  if (serverThread != null)
    throw new IllegalStateException("Already started");
  serverThread = new Thread(new ConnectionHandler());
  serverThread.start();
}

private class ConnectionHandler implements Runnable {
  public void run() {
    // ...
    try {
      while (!Thread.interrupted()) {
        RequestHandler r =
          new RequestHandler(server.accept());
        new Thread(r, "worker-thread").start();
      }
    } catch (InterruptedException ex) { /* ignore */ }
    catch (IOException ex) { /* report */ }
  }
}
```
Thread-Per-Message Web Server (cont...)

```java
private class RequestHandler implements Runnable {
    private final Socket sock;
    public RequestHandler(Socket sock) {
        this.sock = sock;
    }
    public void run() {
        try {
            processRequest(sock);
        } catch (Throwable t) { /* report */ }
    }
}
```

- Shutdown process simplified: only one thread uses ServerSocket

```java
public void shutdownServer() throws InterruptedException, IOException {
    synchronized (this) {
        if (shutdownInitiated)
            throw new IllegalStateException("Shutdown performed");
        if (serverThread == null)
            throw new IllegalStateException("Server not started");
        shutdownInitiated = true;
        // only one thread will ever execute this section of code
        serverThread.interrupt();
        serverThread.join();
        server.close();
    }
}
```

Thread-Per-Object via Worker Threads

- Establish a producer-consumer chain
  - Producer
    - Reactive method just places message in a channel
      - Channel might be a buffer, queue, stream, etc
      - Message might be a Runnable command, event, etc
  - Consumer
    - Host contains an autonomous loop thread of form:
      ```java
      while (!Thread.interrupted()) {
          m = channel.take();
          process(m);
      }
      ```
  - Common variants
    - Pools
      - Use more than one worker thread
    - Listeners
      - Separate producer and consumer in different objects
Web Server Using Worker Thread

```java
public interface Channel { // buffer, queue, stream etc
    Object take() throws InterruptedException;
    void put(Object obj) throws InterruptedException;
    int size();
}

// WebServer15.java
private Channel channel = new BoundedBuffer(); // synchronized

private class ConnectionHandler implements Runnable {
    public void run() {
        RequestHandler r = null;
        try {
            while (!Thread.interrupted()) {
                r = new RequestHandler(server.accept());
                channel.put(r);
            }
        } // ... interrupt and exception handling - more complex
    }
}

private class ChannelConsumer extends Thread {
    // Exception handling elided for simplicity
    // Also for simplicity, assumes channel has only one consumer
    public void run() {
        boolean stopProcessing = Thread.interrupted();
        while (!stopProcessing) {
            ((Runnable) channel.take()).run();
            if (!stopProcessing)
                stopProcessing = Thread.interrupted();
        }
    }
}
```

Channel Options

- **Unbounded queues**
  - Can exhaust resources if clients faster than handlers
- **Bounded buffers**
  - Can cause clients to block when full
- **Synchronous channels**
  - Force client to wait for handler to complete previous task
- **Leaky bounded buffers**
  - For example, drop oldest if full
- **Priority queues**
  - Run more important tasks first
- **Streams or sockets**
  - Enable persistence, remote execution
- **Non-blocking channels**
  - Must take evasive action if put or take fail or time out