433 Final Exam, Fall 2006

December 20th, 2006

Questions

1. In the paper "A Note on Distributed Computing", the authors claim that some frameworks for distributed computing such as Remote Procedure Call (RPC) and CORBA offer an attractive vision that is not a good foundation on which to build reliable distributed systems.

   What is the vision that they warn against, and name and describe at least two of the differences (they mention 4) between local and distributed computing (you will lose points for listing differences not in their list of 4).

   Answer: They warn against:

   > There is an overall vision of distributed object-oriented computing in which, from the programmer's point of view, there is no essential distinction between objects that share an address space and objects that are on two machines with different architectures located on different continents....
   >
   > ... While the underlying mechanisms used to make a method call may differ depending on the location of the object, those mechanisms are hidden from the programmer who writes exactly the same code for either type of call, and the system takes care of delivery.

   The differences they site are:

   - Latency
   - Memory access
   - Partial Failure
   - Concurrency

2. Write a class with two methods, t1() and t2(), that if called by two different threads, might deadlock. Add whatever fields or other methods you need. Your methods shouldn't (and don't need to) call wait() or await(). This question isn't asking to write some useful abstraction, but rather to provide a very concrete description of what deadlock is. Also provide a short (English) guideline for how to avoid deadlock (it doesn't matter if the guideline isn't universally applicable; just describe what steps suffice in the majority of cases to avoid deadlock).

   Answer:

```
class DeadLock {
  Object a = new Object();
  Object b = new Object();
  public void t1() {
    synchronized(a) {
      synchronized(b) {}}
  }
  public void t2() {
    synchronized(b) {
      synchronized(a) {}}
  }
}
```

To avoid deadlock, it suffices to never hold a lock on more than one object at a time. In rare cases where you do need to hold a lock on more than one object at a time, make sure there is a consistent order in which the locks are obtained (e.g., if you ever lock both o1 and o2, make sure you get the lock on o1 before you try to get the lock on o2).

3. This question concerns the difference between `Set<? extends Foo>`, `Set<Foo>` and `Set<? super Foo>`.

   (a) Describe some operations that will compile against some of them, but not against other, that distinguish them (e.g., you can do *Blah* to a *X* or a *Y*, but doing it to a *Z* won't compile. Give enough examples to distinguish all three.

   (b) Consider the two methods below:

   ```
   class Bar {
     HashSet<Foo> foos;
     Set<? extends Foo> getFoosVersionOne() { return foos; }
     Set<? super Foo> getFoosVersionTwo() { return foos; }
     ... other methods
     }
   ```

   Assume your project is trying to decide which which return type the getFoos() method should have, based on what behaviors they want to encourage/discourage/permit/forbid in the caller of the getFoo() method. Summarize, briefly, how they should decide based on the behaviors they want to e/d/p/f.

   (c) Does the choice above actually forbid certain behaviors, or merely discourage them? Explain.

   Answer:

   (a) (Not the only examples) You can iterate through the Foo's in a `Set<? extends Foo>` or a `Set<Foo>`, but not a `Set<? super Foo>`. You can add a Foo to a `Set<? super Foo>` or a `Set<Foo>`, but not a `Set<? extends Foo>`.

   (b) getFoosVersionOne discourages callers from adding anything to the returned set. getFoosVersionTwo discourages callers from iterating through the Foo's in the returned set, but allows them to add Foo's to the returned set.

   (c) It only discourages. Casts can be used to get rid of the generic types.

4. Short Answer:

   (a) What design pattern is used to construct instances of the InetAddress class, and why?

   Answer: Static factory, since depending on the name provided and the DNS resolution, you might get back either a Inet4Address or a Inet6Address.

   (b) What design pattern is principally used in the Java input/output stream classes (e.g., java.io.DataInputStream)?

   Answer: Decorator pattern.

   (c) In distributed method invocation, such as our project 3, is a thread typically associated with a stub, a skeleton, or both?

   Answer: Skeleton, since it has to listen for input coming from other machines.

5. Consider TwoPlaceQueue a blocking queue implementation that can store up to two elements. Write a multithreaded test case, using the metronome timer and testing framework discussed in class and used in project 2, that tests the scenario where an attempt to put a third item into the queue blocks because the queue is full. Your test case should only put a total of three items into the queue, remove all three items from the queue, and make sure the right items come out in the right order and that the queue is empty after all three items have been removed, and that the methods block when they should and don't block when they shouldn't.

For your case, you should only worry about testing the methods shown below (take and put are the blocking methods). Note: unlike the BlockingQueue interface, in this class the put() and take() methods don't throw InterruptedException, and you don't have to worry about interrupts.

```
public class TwoPlaceBlockingQueue<E> implements BlockingQueue<E> {

  public E take() { ... }

  public void put(E e) { ... }

  public boolean isEmpty() { ... }

  ... other methods you shouldn't be testing
  }
```

Your MultithreadedTestCase for the scenario described:

Answer:

```
static class BlocksWhenFull extends MultithreadedTestCase {

   TwoPlaceBlockingQueue<Integer> queue = new TwoPlaceBlockingQueue<Integer>();

    public void thread0() {
      queue.put(0);
      queue.put(1);
      assertEquals(0, getTick());
      queue.put(2);
      assertEquals(1, getTick());
    }

    public void thread1() {
      waitForTick(1);
      assertEquals(0, queue.take());
      assertEquals(1, getTick());
      waitForTick(2);
      assertEquals(1, queue.take());
      assertEquals(2, queue.take());
      assertTrue( queue.isEmpty());
      assertEquals(2, getTick());
    }
  }
```

6. Garbage collection: Assume you had to select just one garbage collection algorithm for a Java virtual machine: mark-and-sweep or a copying collector. What characteristic of your application would be most relevant to choosing the garbage collector? How would you choose (e.g., if my application required *Blarg*, I'd choose a *Snaff* garbage collector)?

   Answer: A copying collector is very efficient if the application produces a lot of garbage (i.e., allocates a lot of memory that isn't still being used at the time of the next GC). However, a straightforward copying GC needs to have at least twice as much memory available as the amount of live heap data at any particular time. So if the application needs a large amount of live data, compared to the memory available, and/or if the application doesn't produce a lot of garbage, then a mark-and-sweep collector would be better.