CSE221: Homework 2

November 1, 2007

Due: Thursday, November 8th, 2007 at the start of class (11:00am)

Answer the following questions. For questions asking for short answers, support your answers with material from the references papers, and/or with your own critical arguments, as appropriate. I am interested in your justifications as much as the answer itself. There may not necessarily be a “right” answer, although some answers may be easier to justify. Finally, do not use shorthand; write your answers using complete sentences.

1 The dangers of layering

1.1

Many of the papers we’ve seen structure their systems using consecutive layers. While this has methodology has many advantages, one of the major disadvantages has been the safety and progress issues inherent in the possibility of mutual recursion between layers. For example, when faced with the possibility of a "deadly embrace" caused by recursion between his virtual memory and his I/O facilities, Dijkstra re-architected his system to break up the I/O facilities and placed disk responsibilities within the virtual memory layer.

For the following papers, explain (a) how the system was layered (b) where the authors noted dangerous recursion, (c) how it was dangerous, and (d) what the authors either did, or described, to avoid the problem.

1. Upcalls
2. Pilot
3. Medusa

1.2

What is one other method, distinct from those used above, for avoiding recursion?
2 Parsimony of abstraction

2.1

Many of the systems we’ve seen so far attempt to virtualize a resource to provide transparency to the underlying implementation. When virtualizing a resource, a system designer provides an interface for that resource which hides details about the resource from the interface’s user.

Butler Lampson once gave a set of principles for system design; he considered his most important design principle to be that of keeping interfaces simple (but, of course, not too simple). He elaborates:

“An interface should capture the minimum essentials of an abstraction. Don’t generalize; generalizations are generally wrong. When an interface undertakes to do too much its implementation will probably be large, slow and complicated. An interface is a contract to deliver a certain amount of service; clients of the interface depend on the contract, which is usually documented in the interface specification. They also depend on incurring a reasonable cost (in time or other scarce resources) for using the interface; the definition of ”reasonable” is usually not documented anywhere. If there are six levels of abstraction, and each costs 50% more than is ”reasonable”, the service delivered at the top will miss by more than a factor of 10.”

For each of the following systems, explain: (a) What resource was virtualized (b) Did the interface (or interfaces) capture ”the minimum essentials of an abstraction”? More? Less? (b) Was the cost incurred in using the interfaces ”reasonable” by Lampson’s standards? That is, reasonable with regard to the cost of service delivered at the top of the abstraction? (d) What did the authors hope to achieve with their design?

1. L4
2. Sprite
3. V Kernel
4. Xen

3 The implications of sharing

3.1 Names

3.1.1

In distributed systems, care must be taken when naming shared resources. There are two classes of schemes for naming objects: hierarchical and flat.

Both Sprite and Grapevine used hierarchical naming schemes for files and users, respectively. LOCUS also used a hierarchical naming scheme for files, although with one crucial difference (that is arguably disadvantageous for LOCUS). What is this difference? What is one advantage it gave to Sprite and
Grapevine’s designers? How did LOCUS’ designers work around this disadvantage?

3.1.2
Recall that Sprite attempts to ease the bottleneck of name lookup by caching. What is the problem with doing caching for name lookup in Sprite? How did the authors address this problem without sacrificing significant performance? Estimate, to the best of your ability, the performance gain achieved using this scheme.

3.2 Content
3.2.1
Grapevine’s design was highly impacted by the relatively relaxed requirement of eventual consistency and the exposure of failures. LOCUS, on the other hand, advocates a very strict consistency and absolute transparency in most, if not all, cases. When does LOCUS allow for eventual consistency?

3.2.2
Consider a variant of LOCUS which allows for eventual consistency and consequently non-absolute transparency in the system in more general cases. Describe one way to change the LOCUS system to allow for a “multiple readers, multiple writers” policy. Estimate, to the best of your ability, the performance gain achieved by making your change.