Name: AndrewID:

15-440 Homework #1
Spring 2013 (Kesden)

1. Networks [12 points]

(a) Given a 100Mbps network with an optimum sliding-window size of 1024 bytes, what would be the optimal window size if the bitrate was increased to 1Gbps and the maximum allowable run of the cable was cut in half? Why?

(b) Consider a wired, switched Ethernet network. Now imagine that the network-connected hosts are heavily mobile within the network, but across the various network legs connected to the switch. Upon moving, what does a host need to do in order to re-establish connectivity? If the host doesn't do this, how long will it be, before it is reachable from hosts on other legs? Why?

(c) When transmitting at the link layer, we normally "frame" data. Why? For example, why do we not just stream bits?

(d) MAC and other link-layer addresses are generally not used for higher level routing, such as IP routing. Why? In answering, please explain what property is essential for higher level routing and why it would be inconvenient for it to exist in link layer MAC addresses.

Monolithic Concurrency Control [12 points]

2. Consider a proposed concurrency control primitive, the *fubar*, with the following operations:

`fubar.sleep(boolean predicate)` – atomically tests predicate and, if false, blocks until the same object is signaled
`fubar.wakeup()` – wakes up –all- fubars sleeping on exactly the same object

Can *fubars* be used to ensure mutual exclusion (without deadlocking, livelocking, or other silliness) among two or more competitors for a single critical resource? If yes, please provide an example showing their use to protect some `criticalsection()` in competitors containing the following logic:

```
while (forever) {
        dosomestuff();
        criticalsection();
        dosomemorestuff();
}
```

If not, please explain the fundamental problem with this approach.

3. Consider a proposed concurrency control primitive, the *rabuf*, with the following operations:

`rabuf.sleep`(mutex predicateMutex) – always sleeps, placing caller into a FIFO queue for a later signal(), and atomically releases the mutex
`rabuf.signal()` – wakes up the head of the FIFO queue of those waiting for the *rabuf*

Can *rabufs* be used to ensure mutual exclusion exclusion (without deadlocking, livelocking, or other silliness) among two or more competitors for a single critical resource? If yes, please provide an example showing their use to protect some `criticalsection()` in competitors containing the following logic:

```
while (forever) {
        dosomestuff();
        criticalsection();
        dosomemorestuff();
}
```

If not, please explain the fundamental problem with this approach.

4. Consider a dinner table containing *n* plates, 2*n* forks, and unlimited servings of food (Ain't catering great?) . In order to eat, one needs to have 1 fork, 1 helping of food, and 1 plate. Consider a meal where each participant is starving and there is no nearby parental unit to enforce a communally productive sharing policy.

(a) Assume that the mealtime logic is implemented –exactly—as below within each and every participant.

```
while (hungry) {
     breath();
     grab(fork, 1);
     grab(plate, 1);
     grab(food,1);
     chowdown();
     burp();
}
```

Is deadlock possible? If so, in terms of n, how many participants are required before it becomes possible? How do you know?

(b) Assume that the logic within each participant is implemented as below within each and every participant, but that each participant will reach for forks and plates in different orders, depending on which happens to be closer.

```
while (hungry) {
     breath();
     grab(fork, 1);
     grab(plate, 1);
     grab(food,1);
     chowdown();
     burp();
}
```

Is deadlock possible? If so, please (a) identify, in terms of n, the minimum number of participants required before deadlock becomes a risk, and (b) illustrate using the code above as an example, how semaphores can be introduced to eliminate the risk of deadlock while ensuring progress.

5. Middleware/RPC/RMI [20 points]

(a) Consider the implementation of an RPC system in a homogenous environment (same hardware, same OS, same language, same, same, same). Is it possible to implement a pass-by-reference (not necessarily pass-by-address) mechanism? If not, why not? If so, in what ways might it be best to relax the semantics of a typical local pass-by-reference situation? Why?

(b) Consider the implementation of an RPC system in a heterogeneous environment (different processor architecture, different OS, different programming language, different, different, different). How might the heterogeneity complicate the model? Please consider each of the following:

i) Simple primitives (think back to 213 for how they can differ from system to system)
ii) Complex data types, including structs and strings,
iii) Higher-order language and library data structures, such as linked lists, maps, etc.
iv) Programming paradigms (function pointers, jump table, functors, etc)

(c) Consider Java's RMI facility, which generates stubs at compile time. Could it, instead, generate the stubs at runtime? For example, could it disassemble a class file, or inspect an object's properties at runtime, rather than at compile time? If not, why not. If so, what would be the advantages and disadvantages of this model?

(d) Consider Java's RMI facility, which only plays nicely with classes that implement the *Serializable* or *Remote* interfaces. Would it be possible to implement an RMI facility in Java that worked for all classes? For example, by using a combination of the class file, as well as reflection and other Java mechanisms to decompose, serialize, and reconstitute instances by brute force? If so, please explain any necessary limitations. If not, please example why not.

6. Distributed Concurrency Control [12 points]

In class we discussed enforcing mutual exclusion, among other ways, via a central server, majority voting, and token ring.

(a) [4 points] Which of these systems requires the fewest messages under heavy contention? How many messages are required per request?

(b) [4 points] Which of these systems requires the most messages under heavy contention? Why?

(c) [4 points] Which of these systems is most robust to failure? Why?

7. Distributed Concurrency Control [12 points]

Consider the voting techniques that we discussed in class, including voting districts. These approaches allowed for race cases during votes, including those that allowed the potential for deadlock. None-the-less, deadlock was not possible in a system where all nodes operate according to these protocols.

(a) How is deadlock prevented or corrected? Explain.

(b) Are these protocols *fair*? Does the first node to make a request always win? Why or why not?

(c) Do these protocols implement a priority-based resource allocation, e.g. certain nodes will always win races? If so, how? If not, provide an example of how a higher priority node, e.g. higher node ID, can lose a race.

8. Coordinator Election [16 points]

(a) Consider a bully-based technique for electing a coordinator. Is it possible for failure to result in multiple coordinators being elected? If so how? If not, why not?

(b) Consider a voting-based technique for electing a coordinator. Is it possible for failure to result in multiple coordinators being elected? If so how? If not, why not?

(c) In either case, if you believe that it is possible for a naïve implementation of either or both protocols to elect multiple coordinators under the right set of circumstances, what simple fix can you apply to guarantee that at most one coordinator be elected?

(d) Under what circumstance might it be desirable to allow multiple coordinators to be elected? Please provide and explain an example.

9. Replication and Quorums [16 points]

(a) Consider the writing of an object to the number of hosts required by some write quorum. Is any concurrency control required to protect this object during the update? If so, why? What can happen without it? If not, what property of the quorum system protects it without the need for external controls?

(b) Consider the impact of a write-all/read-none protocol. What is the risk of counting what are believed to be dead servers upon a write? How can it be mitigated?

(c) Consider Coda's use of logical time stamps, known as *Coda Version Vectors (CVVs)*, to manage replicas. Given a set of CVVs, how does a client determine if any are concurrent? What situations might result in a client observing concurrent CVVs?

(d) Consider Coda's use of logical time stamps, known as *Coda Version Vectors (CVVs)*, to manage replicas. What circumstance is indicated by non-identical, but non-concurrent CVVs? Should a client discover this, how should it react?