Answers to Homework #3, CS 239, Section 1, Spring 2007

1.

A. VoIP traffic normally uses a single size of packet. Increased load in a system supporting VoIP should be represented as an increase in the number of packets, not an increase in the size of each packet. Depending on details of the LAN, the results of testing this way could be very misleading.

B. File server behavior depends heavily on caching. As described, the workload is likely not to have realistic caching behavior. First, there will be no time-based locality of reference in which files are chosen to be accessed, which is almost always present in real file server behavior. Second, the type of request will not properly match the actual order that would be observed in real file behavior. Depending on precisely how the file system is implemented, for example, performing a directory traversal could bring some short files into memory along with the directory, in which case they would already be in the cache if they were subsequently opened and read. The access generator should use at least a Markov model, and maybe a more detailed order-based model.

C. As in the file system case, it seems likely that the logging requests will have some history. However, it's not clear that would be a vital element to be captured in the workload for the performance of the system, since the data is being written, not read. If the log server stored separate logs for different types of activities, it is possible that performance effects related to the disk behavior might be altered by not producing a realistic stream of logging requests.

More important is that the other processes are started at random intervals. That's not how the real system works at all. Backups, for example, are performed at night, presumably because an office environment's machines are largely idle at night, so the backups will not compete seriously with the logging activity. Similarly, the log compression utility runs during idle periods in the real system, not at random intervals. Unless the log server is idle more often than it is busy, the compression utility will often be started during busy periods, falsely indicating load that isn't present in the real system.

D. Average isn't the whole story. Variation could be pretty important here. As an extreme example, perhaps half the time the network achieves a loss rate of 2% and half the time a loss rate of 4%. That means that, half the time, games are unplayable. The company won't stay in business long, in that case. Knowing the average loss rate is not enough here. Really, even knowing the average plus any of the common measures of deviation isn't enough here. If I were running the company, the number I want to know about is the percentage of time that the network experiences a loss rate over 3%.

A secondary issue is that there isn't a lot of detail here about testing conditions. We can, perhaps, presume that the analyst tested a workload representative of lots of kids playing video games, but what about other experimental conditions? In particular, was anyone else in the room while he was testing the system? Large numbers of people in a relatively small space can cause degradation of the wireless signal, perhaps leading to a higher loss rate. Other experimental conditions might also be important. For instance, did he test during the periods of the day and week when we expect high usage? It's possible that there are other wireless networks in the same vicinity. If he tested during a time when they were largely inactive, he might see different results than if he tested during a period when they generated a lot of competing traffic.

2. The issue of measuring file hoarding is actually surprisingly complex. I don't expect anyone to include all of the points below, and will give full points for answers that are working in the right directions.

A. The key metric is some kind of indication of the number of failed file accesses experienced due to the hoard not holding the requested file. In a hoarding system, applications often crash if a required file has not been hoarded. Each hoard miss can thus be regarded as a likely failure of an application. However, actual experience showed that not all hoard misses are fatal. Some are no worse than a trivial inconvenience, perhaps even one that the application is prepared for. In other cases, there might be no discernable record of a failure to open a file, yet the user experienced a hoard miss. Consider, for example, a case when a user lists the contents of a directory, expecting to find and work on a particular document. If he doesn't see the document in the directory because it wasn't hoarded, he won't even try to open it, yet he has experienced a failure of the system.

Other metrics are possible. For example, one could measure time-to-first-miss, which is defined as how long the disconnected system was able to actively perform work before trying to access a file that wasn't hoarded.

Depending on experimental methodology, another possible metric is miss-free hoard size. This is only appropriate when the methodology allows replaying of the file system accesses, as it might with a trace or certain kinds of a generator. One could determine just how big a disk full of hoarded stuff one needed to get through the disconnection period without any hoard misses. This would allow comparison of hoarding algorithms and give a characterization of the efficiency of the hoarding algorithm.

B. If a live workload is not possible, a trace of real activity is next best. However, it must be run in such a way that relationships between processes and file accesses is preserved. If a process in the trace generates a single file access attempt that fails, all subsequent access attempts for that process should not be replayed or measured. One can argue about whether all subsequent new invocations of the process should or should not be cancelled, as well. If the hoarding miss was on a data file (e.g., the word processor could not find the document it was to edit), other invocations of the process might succeed. If the hoarding miss was on a required configuration file, however, not only will all future invocations fail, but sooner or later a real human user would stop trying to run that application, regardless of what the trace said. One might even argue that the first hoarding failure experienced in a trace invalidates the remainder of the trace, since, after all, failure of that application could entirely change the user's behavior for the rest of the trace. For instance, if a user hadn't been able to read a piece of email reminding him that a paper was due the next day (because of a hoarding miss), he might never have started editing the paper, a process that could otherwise proceed successfully if all files related to editing the paper were hoarded..

A generator could be created for this purpose, too. It would be best for such a generator to create full invocations of applications, not lower level file system accesses. That way, we can relatively cheaply determine if the applications succeed. Of course, if the applications require user input to run properly, the generator would have to generate that, as well, leading to greater complexities. The alternative, generating file system accesses, has its own drawbacks. The generator would need to understand application behavior well enough to generate a realistic set of file accesses to represent real processes.

C. This depends to a large extent on the testing methodology. If one is running traces, one needs to capture failed attempts to access a file. The operating system or its file system could be instrumented for this purpose, or the trace generator could simply observe the effects of each access attempt. The file system will tell it when the attempt failed. Similar methodology could be used by a generator.

For live user testing, in certain cases the user might not even notice a hoarding miss. Also, it's not uncommon for users to accidentally request access to non-existent files. (Particularly if they are using command line interfaces, but also by typing file names into prompts or windows.) It's not the hoarding system's fault if a file that never existing isn't there, and, in fact, pointing out the failure is correct system behavior. For this situation, perhaps the best instrumentation is an interface that allows users to register hoarding failure attempts.

3.

A. A sampling monitor could be used here, but an event driven monitor is perhaps better. The memory used by a process increases when it does some particular event, such as allocating and deallocating memory. Adding a little instrumentation at each occurrence of such events is probably reasonable.

B. Sampling is better. You probably can't afford to get in the router's way all the time by generating an event at each packet arrival. The exception would be if the router had built-in instrumentation that allowed it to gather the data.

C. A sampling monitor might do fine here. If you sample at a sufficiently high rate over a long enough period of time, you would expect the sampled values to be statistically similar to the values observed at any given moment. You could use an event driven monitor, as the handling of a web request is a heavyweight operation compared to recording a queue length. If you badly need to know the true maximum queue size observed, an event-driven monitor will be better, since the sampling monitor is only likely to capture that maximum if the system spends a lot of time with that many requests in its queue.

4. Assuming one has access to the source code of the mail reader (it being open source, you probably do), the best choice is to insert code into the mail reader to record the amount of time spent in its message handling and spam analysis components for each message. In addition, it's wise to measure the overall time from when the first message in the trace is offered to the mail reader to the time when it has gotten them all and finished any processing it is doing on them. While one might expect the spam analysis component to take most or all of the time related to handling a message, it's possible that there are other elements of the mail handler that also see a performance impact due to the detection of spam. For example, it might be rebuilding detection filters "in the background" between handling messages, which could have an overall effect on mail performance that differs depending on the mix of messages. By measuring the overall elapsed time to handle the entire workload, one can determine if there are such hidden costs spread throughout the program. One could insert many more probe points into the

code without seriously impacting performance to find hidden costs more directly, but doing so will require quite a bit of programming effort.

If you don't have access to the source code, or can't afford the time to understand and properly instrument the source, then unless you have human observers helping you time things (which would be tedious, at best), probably the best you can do is wrap the application in timing code to determine when it stops and starts.   This approach will probably require you to use the workload in a different way. Instead of feeding the entire corpus of good and bad messages to the mail reader in one group, you will want to divide it into pieces, with each piece designed to have a chosen proportion of good and bad mail. By measuring the different times taken for different mixes, you can deduce the costs of each. A good deal of care might be needed in choosing which messages go in the mix, since the program might be doing some learning procedures. If you repeat the delivery of a given spam message between two runs, its handling on the second run might be quite different than its handling on the first.