WRITTEN 1

OPERATING/ DISTRIBUTIVE SYSTEMS

(Sindhuja Yerramalla, Syrrmlla, U00839259)

Question 1:

Note 1.3 is basically a discussion against distribution transparency. Distribution transparency is nothing but hiding the fact that processes or resources are dispersed across multiple computers. Processes and resources are not visible to end users, they are hidden. But several researchers are arguing that **distribution transparency further complicates the development of distributed systems**. For example, note 1.3 took one of the access transparency techniques, in which the procedure calls are extended to remote servers. But Waldo proved that sometimes the procedure calls will change when those are executed over faulty communication link. So instead of hiding completely it is better to use **less transparency.** Less transparency is like using web when fetching pages instead of message-style communication and getting results from remote machines. And another best way is, if we have guarantee about partial failures even after successful execution at remote servers then it is best to perform local executions leading to **copy-before-use principle.** Copy-before-use is nothing, but the data can be accessed only after transferring that to the machine which needs that data to finish the process.

Question 2:

Note 2.4 is an argument about placing user interface and application in client side or server side. If the user-interface and application is placed on client side those type are called as **Fat clients.** So, in fat clients the application software will be handled by different end users, hence maintaining and developing end-user friendly software is a hard job and can have many difficulties like some times end-user may not have proper resources or a compatible OS platform etc. But that doesn't mean that thin clients are good over fat clients. Fat clients are not good only from system-management perspective, but they are good in many applications like multimedia where the processing is done mostly on client side. And now a days by using Web browsing technology, placing and managing the client-side software is done by just uploading some scripts. This story doesn't mean that if we move away from fat clients, we no longer need distributed systems. Even in thin clients, if the application and data base both are at server side we need help of distributed systems to **replace single server with multiple servers running on different machines** (like cloud computing where the server side process is being executed in data centers.)

Question 3:

Note 2.6 is about Flooding versus random walks. In random walk, a request is sent to a randomly chose neighbor or node whereas in flooding the request of data item is sent to all neighboring nodes at a time. The argument says that since we are dealing with replicated data, **random walk is effective for very small replication factors** and different replication distributions than compared to flooding. If we consider N nodes and r Randomly chosen nodes and the process of selecting a random node will be repeated till the item is found. If P[k] is the probability of item is found after k attempts and S be the average search size, then we have r/N value in computation of S. so r/N is low as 0.1% then S would be

approximately 1000 nodes. On the other hand if R(k) is the probability of item found after k flooding steps, the R(k) will be a good estimate as long as we have few flooding steps.

$R(k) = d(d-1)k\text{-}1$ where d is number of randomly selected neighbors

we can expect a fraction of r/N to have the requested data item, meaning that when $r/N \cdot R(k) \geq 1$, we will most likely have found a node that has the data item. The disadvantage of using **random walks is that it takes much longer to get an answer.**

Question 4:

Note 3.2 is about Light weight processes, A user-level thread package is also provided by the system, giving programs access to the standard functions for creating and terminating threads. The thread package's complete user space implementation is a crucial issue. In other words, all thread operations happen without the kernel getting involved. Multithreaded applications are constructed by creating threads, and subsequently assigning each thread to an LWP. The thread package can be shared by multiple LWPs, This means that each LWP can be running its own (user-level) thread. The thread table, which is used to keep track of the current set of threads, is thus shared by the LWPs and the scheduler.

When an LWP finds a runnable thread, it switches context to that thread. If a thread needs to block on a mutex or condition variable, it does the necessary administration and eventually calls the scheduling routine. The beauty of all this is that the LWP executing the thread need not be informed. When a thread does a blocking system call, execution changes from user mode to kernel mode, but still continues in the context of the current LWP. At the point where an LWP can no longer continue, the operating system may decide to switch context to another LWP, which also implies a context switch back to user mode.

They can be easily used in multiprocessing environments by executing different LWPs on different CPUs. The only drawback of lightweight processes is that **we still need to create and destroy LWPs**, which is just as expensive as with kernel-level threads.

An alternative to lightweight processes (LWP) is to make use of **scheduler activations**. When a thread blocks on a system call, the kernel does an upcall to the thread package to select the next runnable thread. The advantage of this approach is that it saves management of LWPs by the kernel. Many-to-many threading models are often preferred over single-threaded models because of their simplicity and low performance gain.

Question 5:

Note 4.3 is basically explaining a remote procedure call which is implemented for append operation and a python code is also included. In code **the "class Client" (client stub)** has implementation of append. dbList is list of objects that would be in database environment. When client is called with data and dbList as **parameters**, the is **transformed to a tuple** (containing APPEND, data, dbList) and then **sends a request to server class and waits for response**, after

getting the response it will pass the result it got from the server to the program from which it is called initially. On the other hand, in **“class Server**”, When a message arrives, the server waits and **examines which operation needs to be called**. If it got a request to call append, it would then **execute a local call** to its append implementation with the relevant parameters, which are also contained in the request tuple. After that, the **result is sent to the client.**