CMPSCI 691ST Systems **Fall 2009**

Lecture 10

Lecturer: Emery Berger *Scribe: Vijayaraghavan Apparsundaram*

10.1 Introductory Comments

A few people have said that the papers are different from the traditional ones because they're partly about systems and partly about software engineering. These are not traditional papers, in the sense that they don't (or even attempt to) prove what they say. The end-to-end approach paper has some weaknesses because the author doesn't quantify what the benefits of checks at different layers is, but he is mostly accurate. Consider for example, reliable file copying let the p, the probability of a packet being copied correctly be 0.99(which is very high and optimistic), and let the time for each packet transfer be f. for a single packet time taken to copy is about f/0.99. if your file has 1000 packets.Then p(file) = pˆ1000 = 0.991000 = 4.3E-5. so you have to try pˆ-1000 times to get the file correctly copied, and then the time required becomes pˆ-1000*(1000*f) which is a fairly large number. So the naive approach will not work. TCP/IP implements reliable file copying by having checks in intermediate stages. What he's saying is not that you should make a dumb pipe. He is saying that you can't put everything in the pipe, and at some point you have to handle things at the application level.

10.2 Proving the properties of a system mathematically

Djisktra wanted to prove the correctness of a system mathematically. This is OK if correctness was the only concern, but it isn't. There are concerns which just can't be proved. For ex, you can't prove that the communication between China and San Francisco is reliable. One thing you're worried about is the average case performance, not the worst case. To improve system performance, it makes more sense to make the common case fast, and the worst case work (common case optimisation). But average case analysis is much tougher than the worst case, because it is very difficult to decide what the average case input is. ("What's the average file or program?") Whenever you can though, you should definitely prove the properties of a system mathematically, which brings us to the next topic.

10.3 Queueing Theory

A mathematical model basically abstracts away certain details from a system, allowing us to prove its properties. One example of a model, is queueing network model. Suppose you want to model a process which has a series of queues. Ex, Web servers, theres a http server which accepts requests, passes it on to a database server and so on, each of which has a queue. Each server picks a process from its queue and processes it, and possibly passes it on to another server. A QNM can help you analyse such a system. You can run simulations on the model, and for ex, you can estimate the latency for requests given a certain kind of traffic distribution. Emery mentions Flux, one of his older projects, which is a language for programming servers, and which also produces a QNM optionally.

10.4 UML is bad

Someone mentions UML. Emery says the original idea of state charts, which sort of inspired UML, was brilliant, because it let you model, and reason about a system visually and elegantly. It was a visual language to analyse problems. UML on the other hand, is just bad. It is bad because, it is just like documentation. Getting it right the first time is extremely hard, and then to maintain it while the system changes is nearly impossible. And since you can't test it, it's wrong. Passing mention of "data debugging", which is that programs are for the most part correct. On the other hand,there is very little assurance that inputs to programs are right.

10.5 Lottery Scheduling

Instead of traditional CPU scheduling, this guy Carl Waldspurger came up with an alternative: What if you want to schedule it in a way that you give a different share of CPU time to different processes. Like A gets 80Waldspurger also came up with something called stride scheduling. Someone asks if lotterie aren't expensive because you have to generate random nos. Emery says its not expensive because it is amortized over the entire cost of scheduling.He says amortization is an important performance technique.A related technique is piggybacking, where you want to do something that would be expensive on its own, but if you combine it with something else, it is only incrementally more work. Ex, some assertions are expensive, so you can piggyback them on garbage collection.

10.6 Traditional Databases vs Distributed Databases

Traditional databases have ACID(atomicity,consistency,isolation,durability) which guarantee that database transactions are reliable. The NoSQL movement is a revolt against the traditional RDBMS model, because the ACID restrictions make RDBMSes unsuitable for a large distributed environment. Alexandre mentions the CAP theorem: You have consistency,partition survival and availability, pick any 2. Explanation: You have a distributed database with two servers and the link between them goes down. Now each of them gets a conflicting request. If you choose availability and service both requests your system is now inconsistent,or you can choose consistency and turn one of them down. Amazon's solution is eventual consistency. This is bad because no one knows when everyone will have the same view of the system. You can write code, and you won't know how it will behave. The programmer shouldn't have to worry about stuff like consistency. You need something like a memory fence to do this.