COMPSCI 630 Systems **Spring 2018**

Lecture 17

Lecturer: Emery Berger *Scribe: Qi Pan*

17.1 Summary of talks and logistics

We quickly talked about the three presentations we saw as a class. They were:

- Karthik Pattabiraman , UBC - Why do Modern Web Applications Fail and What Can We Do About it? (mostly on why JS fails)
- Adam O'Neill, Georgetown University - Secure Outsourced Databases: Past, Present, and Future (a crypto talk about partially revealed encryption)
- Heather Miller, Northeastern - Bringing Distributed Data Systems to End Users (about spark and practical issues which she addressed such as slow serialization)

Then the next assignment was mentioned to be in Go and related to distributed systems. Also it was noted that the details from the crypto talk (at most stuff from RSA paper) would not be in the final exam as it was beyond the scope of the course but the materials from the others presentations could be.

17.2 Single node fault tolerance

We started talking about fault tolerance in the traditional sense (as opposed to Karthik's presentation which is applied to JS). The traditional sense of fault tolerance refers to the a systems ability to still run when one or more of it's components fails. We listed and discussed many physical examples such as:

- power outage
- physical hardware failure
- acts of god (lightning strikes e.g. Emery's computer getting fried in Florida)
- hardware defects

Besides physical failure, there can also be software errors such as kernel panics, BSOD, and server segmentation faults. The handeling of these failures is called a fail-stop → "when something bad happens, STOP!".

17.3 Checkpointing

Next we discussed some ways practical applications do fault tolerance. A classic/standard technique is to **checkpoint periodically**, meaning to save a copy of whatever work the user is doing every once in a while

to serve as backup (roll back to in case of crash). Prominent examples of this were Microsoft Word, Chrome tabs, and the EMAC.

We then focused on the specific fault of **kernel panics**, which is when the system basically shuts down because a fatal error has caused there to be a high risk of data loss if the system keeps running. We talked about writing a "snapshot" of a virtual machines memory out to disc and how this could be a problem if the snapshot is large (e.g. 64GB in our class example). In our class example, lets say it takes 5 minutes to write out the 64GB into disc, if concurrent things are running during this time, the snapshot could turn out to be inconsistent as things are editing the memory as it is being written out to disc.

A solution to this would be to only grant read access while the memory is being written out, but this can obviously be inconvenient to the user if checkpointing is frequent. Another solution to increase the speed of the checkpointing is to only write the differences in memory between checkpoints and to keep track of the memory pages (fixed block of virtual memory) being used. If they are time stamped, we can follow the diffs to get to where the memory was, and this procedure **can even be paralleled for different sections of memory pages.**

17.4 Availability

The availability of a system is simply the % of time the system is available. Often in computer science the availability of a system is described in terms of "9's", e.g. five 9's = 99.999% availability. We briefly touched on avoiding a single point of failure (making sure the system does not get taken down by one part failing). We then discussed availability in the context of services with multiple nodes and how hard it is to make sure no machine is failing at all. For example if p is the probability any of the machines will fail, and there are n machines, then:

$$P(\textit{no machine is failing}) = (1-p)^n \tag{17.1}$$

Meaning that for a fixed p, as the number of machines increases $P(\textit{no machine is failing})$ decreases exponentially making it very hard to keep availability up when scaling by the number of machines. The only way to cope with this is through **replication of machines** which increases the cost by the number of additional replicas you want per machine.

17.5 CAP theorem

Regarding the tradeoff between fault tolerance and cost as in the multiple node replica case, the class talked about the CAP theorem. CAP stands for Consistency (are the multiple nodes in sync?), Availability (time service is available), and Partition tolerance (tolerate disconnects in the network). In summary the theorem basically proves that it's possible to have **two of these three** in a system. We also talked about the fact that even though the theorem was formally proven, it is fairly obvious logically and pragmatically. We talked about how most large modern systems like Facebook have eventual consistency which means syncing is not necessarily done in real time but eventually.

Another thing that can help consistency is the presence of **immutables**, which simply means what variables/things are declared cannot be changed. This obviously enforces consistency if you copy them to other machines as they cannot be changed by design.

17.6 RAID

Next we talked about RAID (Redundant Array of Inexpensive Discs) which is essentially a physical stack of multiple drives into one unit. This allows for data redundancy (fault tolerance) but also different configurations can balance this fault tolerance with speedup (parallelism). Some techniques applied to the array of discs include:

- data striping - storing consecutive memory on different discs of the array, this allows for speedup since reading and writing is parallel
- mirroring - replicating memory on separate drives, redundancy makes memory more fault tolerant
- error-correcting code (ECC) - can detect single bit errors and correct them (I think it was mentioned this can use checksums which detect small changes to code)

Also mentioned was that in practice there can be many errors with the manufacturing of these discs. Also, even thought the discs are "independent" (later the I in RAID was renamed to independent) because they are physically connected, there is still some hardware fault dependency among them. A common modern thing to do is to have a stack of SSD's which are less sensitive to physical failure since there are no moving parts.

Sidenote: while we were on the topic of RAID, some of Patterson's other papers were mentioned such as RISC, NOW, and ROC.

17.7 Syncrony

Lastly we touched on how distributed systems can stay in sync with each other and issues with timestamps. It is simply difficult to ensure all machines in the distributed system will have the exact same clock. However, this can be mitigated with Lamport clocks or Vector Clocks. Details were not tailed about other than the fact that Lamport clocks maintain the property:

$$\text{if } A \text{ before } B \rightarrow Lamp(A) \leq Lamp(B) \tag{17.2}$$