Chapter 11

Upgrading Live Services

Professor Rieks

Taking the Service Down for Upgrading

- One way to upgrade a service is to take it down, push the new code out to all systems, and bring the service back up
- Very simple to implement, and it permits testing of the service before real users are given access to the newly upgraded service.
- Requires downtime (in most cases except when there is a cluster or multiple servers in play)
- **Rolling Upgrades:** In a rolling upgrade, individual machines or servers are removed from service, upgraded, and put back in service. This is repeated for each element being upgraded; the process rolls through all of them until it is complete.
- The customer sees continuous service because the individual outages are hidden by a local load balancer

Avoiding Code Pushes When Sleepy

- The best time to do a code push is during the day.
- You are wide awake and more co-workers are available if something goes wrong.
- Many organizations do code pushes very late at night. The typical excuse for a 3 AM upgrade is that the upgrade is risky and doing it late at night decreases exposure.
- Doing critical upgrades while half-asleep is a much bigger risk. Ideally, by now we’ve convinced you that a much better strategy for reducing risk is automated testing and small batches.
- Alternatively, you can have a team eight time zones east of your primary location that does code pushes. Those deployments will occur in the middle of the night for your customers but not for your team.

Canary Process

- The canary process is a special form of the rolling upgrade that is more appropriate when large numbers of elements need to be upgraded.

- If there are hundreds or thousands of servers or machines, the rolling upgrade process can take a long time. If each server takes 10 minutes, upgrading 1000 servers will take about a week. That would be unacceptable—yet upgrading all the servers at once is too risky.

- The canary process involves upgrading a very small number of servers, waiting to see if obvious problems develop, and then moving on to progressively larger groups of machines.

    - In the old days of coal mining, miners would bring caged canaries into the mines. These birds are far more sensitive than humans to harmful gases. If your canary started acting sick or fell from its perch, it was time to get out of the mine before you became incapacitated by the gases.

- The canary technique upgrades a single machine and then tests it for a while. Problems tend to appear in the first 5 or 10 minutes. If the canary lives, a group of machines are upgraded. There is another wait and more testing, and then a larger group is upgraded.

Canary Process

- A common canary process is to upgrade one server, then one server per minute until 1 percent of all servers are upgraded, and then one server per second until all are upgraded. Between each group there may be an extended pause. While this is happening, verification tests are run against all the upgraded servers. These tests are usually very simplistic, generally just verifying that the code is not crashing and live queries are succeeding.
- If trouble is found (i.e., if the canary dies), the process is stopped. At this point the servers that were upgraded can be rolled back. Alternatively, if there is enough capacity, they can be shut down until a new release becomes available.

CANARYING IS NOT A SUBSTITUTE FOR SYSTEM TESTING

Phased Roll-outs

- Partition users into groups that are upgraded one at a time. Each group, or phase, is identified by its tolerance for risk.
- Upgrade that group, if all goes well, rollout the update onto a larger group that is less tolerant of risk but, again, it's larger and gets you closer to your goal of 100% rollout.
- Fix problems quickly and move on
- One-some-many
  - Upgrade one user or group of users
  - Some more users
  - Many more

Proportional Shedding

- **Proportional shedding** is a deployment technique whereby the new service is built on new machines in parallel to the old service.
- The load balancer sends, or sheds, a small percentage of traffic to the new service. If this succeeds, a larger percentage is sent. This process continues until all traffic is going to the new service.
- Proportional shedding can be used to move traffic between two systems. The old cluster is not turned down until the entire process is complete. If problems are discovered, the load can be transferred back to the old cluster.
- The problem with this technique is that twice as much capacity is required during the transition. If the service fits on a single machine, having two machines running for the duration of the upgrade is reasonable.

Blue-Green Deployment

- **Blue-green** deployment is similar to proportional shedding but does not require twice as many resources.

- There are two environments on the same machine, one called “blue” and the other called “green.” Green is the live environment and blue is the environment that is dormant. Both exist on the same machine by a mechanism as simple as two different subdirectories, each of which is used as a different virtual host of the same web server. The blue environment consumes very little resources.

- When the new release is to go live, traffic is directed to the blue environment. When the process is finished, the names of the environments are swapped. This system permits rolling back to the previous environment to take place easily.

- This is a very simple way of providing zero-downtime deployments on applications that weren’t designed for it, as long as the applications support being installed in two different places on the same machine.

Rapid Development

- To enable rapid development, features are built up by a series of small changes.
- Merging small amounts of new code together with old code is less error prone.
- Given this fact, incomplete features are hidden by flags that are disabled until the feature is ready. The flags may be enabled earlier in some environments, and for some users, than in/for others.
  - For example, previewing a feature to product management might be done by enabling that flag in the demo environment for the Director.
- Once confidence exists in new code, BETA users can gain access to new features and provide feedback.
- **Differentiated Services:** Sometimes there is a need to enable different services for different users. A good flag system can enable paid customers to see different features than unpaid users see.
  - Many membership levels can be implemented by associating a set of flags with each one.

Live Schema Changes

- Sometimes a new software release expects a different database schema from the previous release.
- If the service could withstand downtime, you could bring down the service, upgrade the software and change the database schema, and then restart the service. However, downtime is almost always unacceptable in a web environment.
- In a typical web environment, there are many web server frontends, or replicas, that all talk to the same database server. The older software release may be unable to understand the new database schema and will malfunction or crash.
- The newer software release is unable to understand the old database schema and will also malfunction.
- You cannot change the database schema and then do the software upgrade: the older replicas will fail as soon as the database is modified. You cannot upgrade the replicas and then change the database schema because any upgraded replica will fail.
- The schema cannot be upgraded during the rolling upgrade: new replicas will fail, and then at the moment the database schema changes, all the failing replicas will start to work and the working replicas will start to fail.
- **What do you do?**

Live Schema Changes

- One way to deal with this scenario is to use **database view**.
- A **view** appears to be a table in a database but it's really the results of a query (select statement)
- Each view provides a different abstraction to the same database. A new view is coded for each new software version. This decouples software upgrades from schema changes. Now when the schema changes, each view's code must change to provide the same abstraction to the new schema. The change in schema and the upgrade of view code happen atomically, enabling smooth upgrades.
- An alternative is to change the database schema by making the change over a period of two software releases: one after adding any new fields to the database and another before removing any obsolete fields.

What Do You Do When It Doesn't Go Well?

- **Dynamic Roll Backs:** It is easier to disable an ill-behaved new feature by disabling a flag than by rolling back to an older release of the software.
    - If a new release has many new features, it would be a shame to have to roll back the entire release because of one bad feature. With flag flips, just the one feature can be disabled.
- **Bug Isolation:** Having each change associated with a flag helps isolate a bug. Imagine a memory leak that may be in one of 30 recently added features.
    - If they are all attached to toggles, a binary search can identify which feature is creating the problem. If the binary search fails to isolate the problem in a test environment, doing this bug detection in production via flags is considerably more sane than via many individual releases.
- **Revert to a VMWare Snapshot**: when it all goes BAD, you can always revert to a backup or a pre-upgrade VMWare snapshot. A snapshot is a point-in-time backup of the system.

Questions