High Availability (HA): Failure/Recovery Analysis

Revised: Jan 5, 2006

Introduction

Given:

- Two registrars, R1 and R2
- Phones P1, P2, ...
- Three network connections:
    - P -> R1
    - P -> R2
    - R1 <-> R2

Consider how failure and recovery plays out with the HA design.
Phones are configured to register with the domain. Use DNS SRV to randomly distribute registrations between the two registrars. Configure both registrars at the same priority and with weights equal to 50.

R1 fails and recovers

R1 fails

- Redirect requests that would have gone to R1 fail over to R2 via DNS SRV
- Before R1 failed, R2 received almost all of R1's registration records via registrar synchronization, allowing R2 to act as a redirector for phones that registered with R1 as well as R2
- A few registrations may be lost because R1 crashed before it was able to push them to R2
- Those phones are effectively unregistered until
    1. Their registrations expire, at which time they reregister (actually, if the phones are smart, they reregister in less than 1/2 of their registration interval, although registration-randomization will mess that up), or
    2. R1 comes back up, if it is able to recover its registry fully
- Next time R2 tries to push an update to R1, that will fail, so R2 will declare R1 unreachable, periodically trying again to see if R1 has become reachable

R1 recovers

- R1 performs startup processing
    - If R1 is able to recover its registry, then
        - Pull from R2 any new updates for which R1 was the primary registrar (updates that R1 has lost)
        - Pull from R2 any new updates for which R2 was the primary registrar (updates that R1 has not yet received, or lost when it went down)
    - Otherwise
        - Pull from R2 all updates for which R1 was the primary registrar
        - Pull from R2 all updates for which R2 was the primary registrar
    - R1 and R2 declare each other to be reachable for replication
- R1 starts accepting registration and redirect requests. The system is now back to redundant operation
- The scenario "R2 fails and recovers" is identical, so we don't need to consider it separately

Phone P loses its network connection to R1

- No problem: DNS failover allows it to register with R2

- Same if P can't connect to R2, it then registers with R1
- In practice this is more complicated since phones connect to registrars through proxies, but the effect is the same
- Do phones handle this right? Need to test that the phone can reach the proxy, and the proxy can reach the registrar

Phone P fails

- When it comes back up, it reregisters

R1 loses its connection to R2

- R1 and R2 operate as registrars, in parallel
- Each registrar declares the other registrar to be unreachable
- This is a really bad scenario because each registrar is only half-informed about what's going on and will send bad answers. Consider some kind of system alarm to alert the admin about this situation.
- If either registrar goes down, the other registrar won't be able to step in as a backup for existing registrations, since it was not able to replicate registrations from the now-unreachable registrar
- This situation only gets bad as registrations expire. If R1's connection to R2 is resumed fairly quickly, before any registrations expire, there will be no interruption of service.This is an argument that when we are randomizing registrations, that the minimum time we choose should be at least twice as long as the longest transient network outage we expect. That would ensure that no registrations would expire during a transient network outage.

R2 misses one or more updates from R1

- XML-RPC is synchronous and reliably delivered, so the only way R2 can miss an update is if the connection goes down, or if R2 goes down
- In either case, R1 and R2 declare each other unreachable, and don't start synchronizing again until they are able to communicate again, at which point they reset their mutual synchronization state

R2 receives an out-of-order update from R1

- Highly unlikely since updates are sent reliably and in order, and message loss forces resetting synchronization state
- Accept out-of-order updates iff they contain more current callid/cseq data that the local DB
- Don't need to compare the update number with PeerReceivedDbUpdate, callid/cseq is what matters
- But log out-of-order updates at warning level because they might reveal a bug or some other problem

Both R1 and R2 fail

Case 1:

R1 goes down before R2 and comes back up after R2. Here's a crude time axis depiction, where "X" indicates being down:

```
R1:     XXXXX
R2:       XXX
```

When R1 and R2 are both down, registration/redirection are down. When R2 comes back up, it tries to reach R1 and fails, so it goes live using its registration DB, which may be missing some info. When R1 comes back up, R1 and R2 are able to synchronize and resume redundant operation.

Case 2:

R1 goes down before R2 and comes back up before R2:

```
R1:     XXX
R2:       XXX
```

When R1 goes down, synchronization stops but R2 carries the torch. R1 comes back up and resynchronizes with R2. Then R2 goes down and R1 acts as the single registrar.