L09c. Content Delivery Networks

Introduction:

- A Content Distribution Network is a globally distributed overlay network of proxy web servers deployed in multiple data centers to serve cached content to end-users efficiently providing high-availability and improved performance.
- Advantages of CDNs:
  - Offload of website traffic from content provider results in:
    1. Improved performance of websites.
    2. Possible cost savings for the content provider.
  - Improved latency: Time taken for a client to receive information from the server.
  - Improved security: CDNs large distributed server infrastructure can provide protection against DoS attacks.

Distributed Hash Tables – DHT:

- CDNs are implemented using DHT. CDNs exploit the DHT technology to store content on the internet so that the content can be discovered and disseminated to the users.
- Example (storing a video on the internet):
  - We want to store the video such that users in various geographic locations can access a local cached copy of the video from the CDN. We will store the video in multiple locations using DHT.
  - We will use a <key, value> pair as the metadata associated with the video:
    1. The key will be the hash of the file contents.
    2. The value will be the node ID where the video is stored.
  - Now we need to find a node to store this metadata on, since we don't use a central server (to facilitate scaling).
  - The <key, value> pair will be stored on the node whose ID is similar (or mathematically close) to the key (e.g. if the key is 149, the node ID will be 149 or 150).
  - Now if someone is looking for that video, he will look for the unique signature associated with the video (the key, 149) and he will know from the DHT structure that he needs to look on node 149 or 150. Once he gets the <key, value> pair he knows on which node the content itself is stored (the value).
- How the content hash is generated?
  - The content hash of the file is generated using the SHA-1 algorithm, which guarantees that the generated hash is unique for different content.
  - There are two names spaces: <Key-space, Node-space>. The generic objective is:

    Store the metadata <ContentHash, ContentNodeId> at nodeId = ContentHash

    Example: Store the metadata <149, 80> at nodeId close to 149 = 150
  - In reality, running SHA-1 algorithm on the content gives us a 160-bit Hash.

- The APIs for manipulating this DHT would be:
  1. putkey(ContentHash, ContentNodeId): The ContentHash is the key/hash of the content that you want to store as a cache in CDN, and the ContentNodeId is the node ID (proxy) of where the Content is stored.
  2. ContentNodeId = getkey(ContentHash): The ContentHash is the key/hash of the content that you want to get from the CDN cache repository.

CDN as an Overlay Network:

- As we saw in the previous example, if I'm looking for a file on the network, I'll end up with a *value* that is not an IP address. We need at the user-level a way of mapping these virtual addresses (the value) to physical addresses.
- CDN is an Overlay Network (a virtual network on top of a physical network) that serves as a routing table to allow content to be shared and distributed with a set of users.
- This routing table is constructed from the relationships between nodes (and friends of nodes).
- If I'm willing to send some content to a node that I don't know anything about (except for the nodeID), I can send the content to a friend node (a node that I know its address) and this friend node might know the address of the target node, or might send it to another friend know and so on till the content reach the target node.
- Similar to IP Network, which is an overlay on top of LAN (IP address to MAC address), CDN is an overlay on top of TCP/IP (Node ID to IP address).
- DHT is an implementation vehicle for a CDN to populate the user-level CDN routing table.

The Traditional Approach:

- When you want to place a <key, value>, you pick a node *n*, where *n* is very close to the key.

  Store the metadata <key, value> at nodeId ~ key
- Similarly, to retrieve the Key, CDN goes to nodeId = key or a nearby node.
- If I'm looking for a node whose ID is not in my routing table, I go to any node in my table whose ID is close to the one I'm looking for, hoping that the target ID is known to the friend node I'm going to.
- This is called the Greedy approach because CDN tries to get to the desired destination as quickly as possible with the minimum number of hops at the virtual network level.
- Disadvantages of the Traditional Approach:
  - Metadata server overload: If a piece of content became popular, we'll have a lot of *putkey* operations for the same key, which will all go to the same metadata server causing it to overload.

- Network congestion: The greedy approach not only overloads the metadata server but also overloads the network causing network congestion in the path from the intermediate nodes to the destination node. The network congestion is in the form of a tree that is rooted at the destination metadata server and the nodes and the network near the root of this tree get congested. This is referred to as the **Tree-saturation problem**.
  This problem happens when:
  1. If we have multiple *putkey* operations with keys mathematically close to each other.
  2. If we have multiple *getkey* operations on the same key.
- Origin server overload: When we have a huge amount of download requests for a specific piece of content, the server storing this content will be overloaded. There're two solutions to this problem:
  1. Using a web proxy: This limits the amount of requests going out of the server. But this solution will not work if there's a Slashdot effect, that is if all the requests require live content, the web proxies will not suffice, since they carry cached content.
  2. CDNs: Content is mirrored from the origin server to selected geographical locations that are being updated periodically. User requests are dynamically routed to the geo-local mirror. The problem with this solution is that it's expensive. This is where Coral System comes in.

The Coral Approach:

- The non-greedy approach used by Coral CDN is implemented using a DHT called the Sloppy DHT. The Sloppy DHT has its *putkey* and *getkey* operations satisfied by nodes that have their node ID mathematically far apart from the key.
- The sloppy DHT implementation spreads the metadata overload so that no single node is overloaded or its nearby network is saturated.
- The exact node ID selection is done using a key-based routing algorithm, that calculates the XOR distance between the source node and the destination node. The bigger the XOR value, the larger is the distance between source and destination in the application namespace.
- The greedy approach wants to get from the source node to the destination node with the fewest number of hops using the user-level routing table that has information on directly reachable nodes. On the other hand, the Coral key-based routing approach slowly progresses on each hop by going approximately half the distance towards the destination node.
- The tradeoff disadvantage of the Coral approach is that it increases the latency in reaching the desired destination due to the increased number of hops with the tradeoff advantage that it results in common good of less load on the network and servers.

- How does the *putkey* operation avoid metadata server overload?
  - First, let's define 2 states for any node:
    1. Full: A node is considered to be Full if it has a maximum of L values per key, i.e. the node is willing to store for L values a particular key.
    2. Loaded: A node is considered to be Loaded if it has a maximum of B request rate per key, i.e. the node willing to entertain for B requests for a particular key.
  - The values of L and B are pre-configured for a Coral CDN instance.
  - As the Coral CDN's *putkey* operation slowly progresses to the destination node, it checks the state of each intermediate hop and if it is either Full or Loaded, then it infers that the remaining network path to the destination (origin server) is all clogged-up because of tree saturation. So it retracts and performs *putkey* at the previous hop.
  - In short, there are 2 phases of the *putkey* operation:
    1. Forward phase: Slowly progress by going approximately half the distance towards the final destination until an intermediate node is either Full or Loaded.
    2. Retract phase: If an intermediate node is either Full or Loaded, retract backwards to the previous hop, confirm that the state of the previous node is neither Full nor Loaded, then perform *putkey* on that node ID.
- How does the get*key* operation avoid metadata server overload?
  - In every hop, the Coral CDN's *getkey* operation slowly progresses by going approximately half the distance towards the final destination, in the hope that the *getkey* operation will find the key somewhere along the way, if the intermediate node is serving the metadata for a particular key.
  - If not, then the *getkey* operation will get the metadata for the particular key from the final destination since nobody has retrieved the key before. But the hope is that if the content is popular enough, then multiple proxies may have gotten the Key-Value pair and in turn when they got the content, they will have performed *putkey* of their own node ID as a potential node for the content. This will cause the metadata server to be an intermediate node along the path to the destination node if the content has been retrieved by somebody else earlier.
  - Thus, the Coral CDN avoids metadata server overload by distributing the *putkey* and *getkey* operations in a democratic manner so that the load for serving both as a metadata server as well as the Content server gets naturally distributed.