medium.com

Design a distributed web crawler - KK XX - Medium

KK XX

8-10 minutes

The Problem

statement 1 (source from internet) :

> Download all urls from 1000 hosts. Imagine all the urls are graph. Requirement: Each host has bad internet connection among each other, Has to download url exactly once.

statement 2 (source from internet) :

> Use 10k IOT machine (limited cpu, bandwidth) to crawl wikipedia. 1. don’t repeat same URL; 2. Minimize traffic. 3. save webpage locally (no storage problem).

Single machine

On a single machine, this problem is deduced to a simple graph search problem, which can be solved with a simple BFS algorithm :

```
seed = { ... }  # initial url set
seen = {}
```

```
for url in seed :
  break if goes too deep
  page = download(url)
  urls = extract_url(page)
  for url in urls :
   if url not in seen :
     seen.add(url)
     seed.add(url)
```

Multiple machines

Let's assume that the machines are within the same data center. When it comes to multiple machines, here are some problems we need to consider :

1. **how to distribute work ?** Basically each machine should be responsible for a subset of urls, we need to divide the entire url set into subsets, such that :

- Machines might be heterogeneous. that means some machines might be faster than other machines. Ideally the partition algorithm should be able to assign partitions according to a machine's capacity.
- Maximize network usage. If we assign urls belonging to the same domain to the same machine, we may save time rebuilding TCP connections; also, in real world that the servers most likely have some rate limit mechanism, which means we need to control the speed for sending requests to the same server. Sending urls belonging to the same domain on the same machine makes it much easier to control such speed. This may not be a problem if we are asked to download Wiki pages (since all of those urls

belong to the same domain).

- Tolerable to membership change. In a real system when network partition occurs, or a hardware fails, some machines might not be available; on the other hand, we can always add more machines to even the workload. The partition algorithm should try to minimize the work when membership change happens.

  In practice we can first map each url to a non-negative 64-bit integer, and divide the whole range into Q sections (Q >> M, the number of machines). Then we randomly assign each section to machines based on machine's capacity. When a new machine joins, it "steals" some partitions from existing machines. The "stealth" is a good chance of balancing the workload of the cluster— for example, it can simply steal partitions from the machines with the highest workload.

  Note that a common term mentioned in literature or distributed system blogs is "consistent hashing". However, it's very easy to get misused without tuning it carefully. One can refer to [1][2] for how to tune consistent hashing correctly.

  Also note that the even distribution of urls may not result in even workload on different machines, because the size of the page, the geo-location of the servers, and the workload of the servers may all be different. To achieve a real even workload, we need to rebalance workload when necessary. Technically the process of "rebalance" can be the same as "stealth" (when membership change happens), and it's a manual process in a lot real-world systems.

  2. **how to coordinate ?** Each machine may download urls that need to be handled by other machines. When it happens, how to

forward such urls to other machines ?

The answer to this question would affect the architecture of the system. In a master-slave system where a single master exists for all the coordination work, while all the slave nodes can be simply stateless, all urls should be forwarded to the master, and the master will forward urls to machines. There's no communication between slave nodes at all; On the other hand, in a peer to peer system, there is no single master, all nodes have to know where to forward a url.

It would be a long list to compare the pros and cons of these two architectures. However, the problem gives a good hint : minimize the traffic between nodes. This indicates that a peer-to-peer architecture would be a better one since its overall traffic is only half of a master-slave architecture.

3. **how to make sure each URL is downloaded exactly once ?**

Since we already decide to forward url based on hash partition, that means the same url will always be handled by the same machine (given no machine failure). In a single machine we can use either a hash table or a bloom filter (if the number of urls to download is huge) to achieve this.

So far we got a rough design that works. There are a few follow-up domains to further discuss (you can briefly discuss with interviewers during interview). In this article I will focus on a common follow-up problem : fault tolerance. i.e., what happens if a node fails ?

4. **how to make it fault tolerant ?**

A node can fail at any time, and recover later (ex., restart,

temporary network partition). When such failure happens, we need to find other nodes to continue the work of the failed node; also for those nodes who are forwarding urls to the failed node, they should forward to a different one.

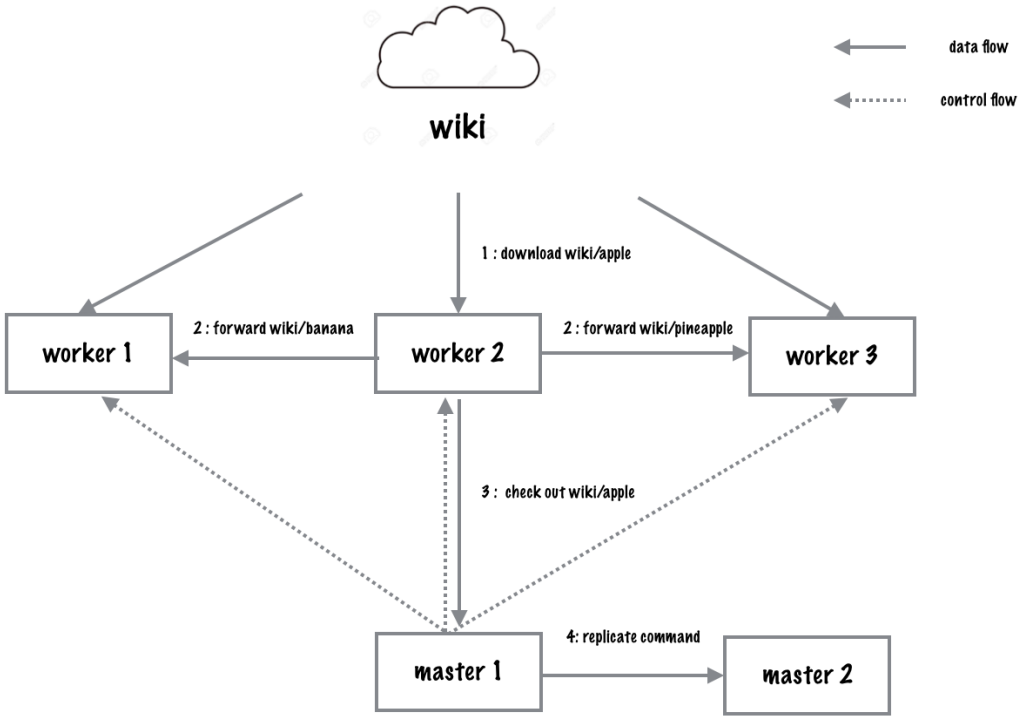
The second case is relatively easy to solve. We can borrow the idea of “consumer group” from [3], i.e., two or more nodes can form a group, ex., node 1, 2, 3 forms a group. Normally node 4 is sending urls to node 1, when it finds that node 1 is unreachable, it may chooses another node from the group, say node 2, and always forward new urls to 2 going forward (even though node 1 recovers later). This preserves the “exactly once downloading” property.

The first case is more difficult : how can we find a node to continue node 1’s work when it fails ? and what should we do when it recovers ? Unfortunately, to detect a node failure, and to find a new node to replace the failed node while preserving the “exactly once downloading” property, we need a master node. A master node plays two roles, first, it sends the heartbeat periodically to all worker nodes, so it can detect a node failure; second, each worker node should check out its current progress to some external system (or to master node), so its work can be continued when it fails. Keeping a master node simplifies the architecture.

Note that “exactly once downloading”, as described in the problem cannot be achieved accurately with the architecture described above. In practice we cannot distinguish between “downloaded the page successfully, but failed to check out the progress” and “failed to download the page”. It’s better to avoid using the term “exactly once” here.

Put it together

The description above yields a simple architecture below :



0. start up stage : deploy metadata to all workers. meta data include seed urls, and decisions about how to divide urls to partitions, assign partitions to each worker, and which workers form a group.

1. worker 2 downloads the page *wiki/apple*, and extract *wiki/banana* and *wiki/pineapple.*
2. based on the meta data, worker 2 decides to forward *wiki/banana* to worker 1, and forward *wiki/pineapple* to worker 3. If either worker 2 or worker 3 does not respond, worker 2 marks them as “fail” locally, and choose another group member to forward.
3. after downloading *wiki/apple*, worker 1 checks this progress out to master 1.
4. master 1 replicates the data to master 2, in case itself fails.

5. master 1 sends heartbeat to all workers regularly to detect node failure. Once detected, it assigns the work left on the failed node to another node, and update meta data on all workers.

Implementation

Coming soon…

All details should make sense when it's presented in code.

References

[1] Dynamo paper : https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[2] Consistent hashing with bounded loads : https://ai.googleblog.com/2017/04/consistent-hashing-with-bounded-loads.html

[3] Kafka design : https://kafka.apache.org/documentation/#design

[4] Web crawling survey : http://infolab.stanford.edu/~olston/publications/crawling_survey.pdf