Crack the System Design Interview

http://www.puncsky.com/blog/2016/02/14/crack-the-system-design-interview/
Interviewers are looking for future teammates that they like to work with. The future teammates are expected to be, at least, capable of solving problems independently. There are so many solutions to any given problem, but not all of them are suited given the context. So the interviewee has to specify different choices and their tradeoffs. To summarize, **system design interview is a happy show-off of our knowledge on technologies and their tradeoffs. Ideally, keep talking what the interviewer expect throughout the interview, before they even have to ask.**

1.1 Step 1. Clarify Requirements and Specs

First things first, the ultimate goals should always be clear.

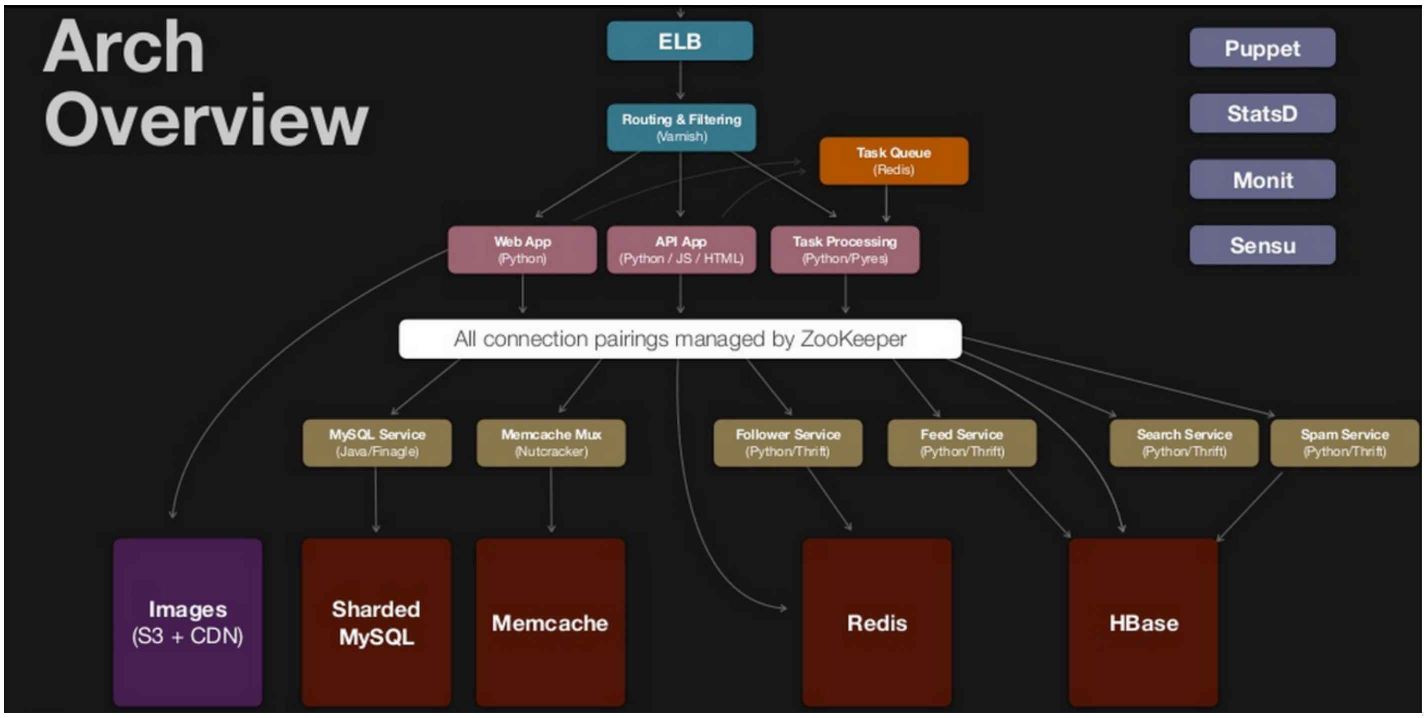
Pinterest is a highly scalable photo-sharing service:

- features: user profile, upload photos, news feed, etc.
- scaling out: horizontal scalability and micro services.

1.2 Step 2. Sketch Out High Level Design

Do not dive into details before outlining the big picture. Otherwise, going off too far towards a wrong direction would make it harder to even provide a roughly correct solution. We will regret wasting time on irrelevant details when we do not have time to finish the task.

OK, let us sketch out the following diagram without concerning too much about the implementation detail of these components.


Arch Overview
ELB
Routing & Filtering (Varnish)
Task Queue (Redis)
Web App (Python)
API App (Python / JS / HTML)
Task Processing (Python/Pyres)
All connection pairings managed by ZooKeeper
MySQL Service (Java/Finagle)
Memcache Mux (Nutcracker)
Follower Service (Python/Thrift)
Feed Service (Python/Thrift)
Search Service (Python/Thrift)
Spam Service (Python/Thrift)
Images (S3 + CDN)
Sharded MySQL
Memcache
Redis
HBase
Puppet
StatsD
Monit
Sensu

1.3 Step 3. Discuss individual components and how they interact in detail

When we truly understand a system, we should be able to identify what each component is and explain how they interact with one another. Take these components in the above diagram and specify each one by one. This could lead to more general discussions, such as the three common topics in Section 2, and to more specific domains, like how to design the photo storage data layout…

1.3.1 Load Balancer

Generally speaking, load balancers fall into three categories:

- DNS Round Robin (rarely used): clients get a randomly-ordered list of IP addresses.
    - pros: easy to implement and free
    - cons: hard to control and not responsive, since DNS cache needs time to expire
- L3/L4 Load Balancer: traffic is routed by IP address and port. L3 is network layer (IP). L4 is session layer (TCP).
    - pros: better granularity, simple, responsive
- L7 Load Balancer: traffic is routed by what is inside the HTTP protocol. L7 is application layer (HTTP).

It is good enough to talk in this level of detail on this topic, but in case the interviewer wants more, we can suggest exact algorithms like round robin, weighted round robin, least loaded, least loaded with slow start, utilization limit, latency, cascade, etc.

1.3.2 Reverse Proxy

Reverse proxy, like varnish, centralizes internal services and provides unified interfaces to the public. For example, www.example.com/index and www.example.com/sports appear to come from the same domain, but in fact they are from different micro services behind the reverse proxy. Reverse proxy could also help with caching and load balancing.

1.3.3 (Frontend) Web Tier

This is where web pages are served, and usually combined with the service / backend tier in the very early stage of a web service.

Stateless

There are two major bottlenecks of the whole system – requests per second (rps) and bandwidth. We could improve the situation by using more efficient tech stack, like frameworks with async and non-blocking reactor pattern, and enhancing the hardware, like **scaling up (aka vertical scaling) or scaling out (aka horizontal scaling)**.

Internet companies prefer scaling out, since it is more cost-efficient with a huge number of commodity machines. This is also good for recruiting, because the target skillsets are equipped by. After all, people rarely play with super computers or mainframes at home.

Frontend web tier and service tier must be stateless in order to add or remove hosts conveniently, thus achieving horizontal scalability. As for feature switch or configs, there could be a database table / standalone service to keep those states.

Web Application and API

MVC(MVT) or MVVC pattern is the dominant pattern for this layer. Traditionally, view or template is rendered to HTML by the server at runtime. In the age of mobile computing, view can be as simple as serving the minimal package of data transporting to the mobile devices, which is called **web API**. People believe that the API can be shared by clients and browsers. And that is why **single page web**

applications are becoming more and more popular, especially with the assistance of frontend frameworks like react.js, angular.js, backbone.js, etc.

1.3.4 App Service Tier

The single responsibility principle advocates small and autonomous services that work together, so that each service can do one thing well and not block others. Small teams with small services can plan much more aggressively for the sake of hyper-growth.

Service Discovery

How do those services find each other? Zookeeper is a popular and centralized choice. Instances with name, address, port, etc. are registered into the path in ZooKeeper for each service. If one service does not know where to find another service, it can query Zookeeper for the location and memorize it until that location is unavailable.

Zookeeper is a CP system in terms of CAP theorem (See Section 2.3 for more discussion), which means it stays consistent in the case of failures, but the leader of the centralized consensus will be unavailable for registering new services.

In contrast to Zookeeper, Uber is doing interesting work in a decentralized way, named hyperbahn, based on Ringpop consisten hash ring. Read Amazon’ s Dynamo to understand AP and eventual consistency.

Micro Services

For the Pinterest case, these micro services could be user profile, follower, feed, search, spam, etc. Any of those topics could lead to an in-depth discussion. Useful links are listed in Section 3: Future Studies, to help us deal with them.

However, for a general interview question like “design Pinterest”, it is good enough to leave those services as black boxes.. If we want to show more passion, elaborate with some sample endpoints / APIs for those services would be great.

1.3.5 Data Tier

Although a relational database can do almost all the storage work, please remember **do not save a blob, like a photo, into a relational database, and choose the right database for the right service.** For example, read performance is important for follower service, therefore it makes sense to use a key-value cache. Feeds are generated as time passes by, so HBase / Cassandra’ s timestamp index is a great fit for this use case. Users have relationships with other users or objects, so a relational database is our choice by default in an user profile service.

Data and storage is a rather wide topic, and we will discuss it later in Section 2.2 Storage.

1.4 (Optional) Step 4. Back-of-the-envelope Calculation

The final step, estimating how many machines are required, is optional, because time is probably up after all the discussions above and three common topics below. In case we run into this topic, we’ d better prepare for it as well. It is a little tricky… we need come up with some variables and a function first, and then make some guesses for the values of those variables, and finally calculate the result.

The cost is a function of CPU, RAM, storage, bandwidth, number and size of the images uploaded each day.

- N users 10^{10}

- i images / (user * day) 10
- s size in bytes / image 10^6
- viewed v times / image 100
- d days
- h requests / sec 10^4 (bottleneck)
- b bandwidth (bottleneck)
- Server cost: $1000 / server
- Storage cost: $0.1 / GB
- Storage = Nisd

Remember the two bottlenecks we mentioned in section 1.3.3 Web Tier? – requests per second (rps) and bandwidth. So the final expression would be

Number of required servers = max(Niv/h, Nivs/b)

2 Three Common Topics

There are three common topics that could be applied to almost every system design question. They are extracted and summarized in this section.

2.1 Communication

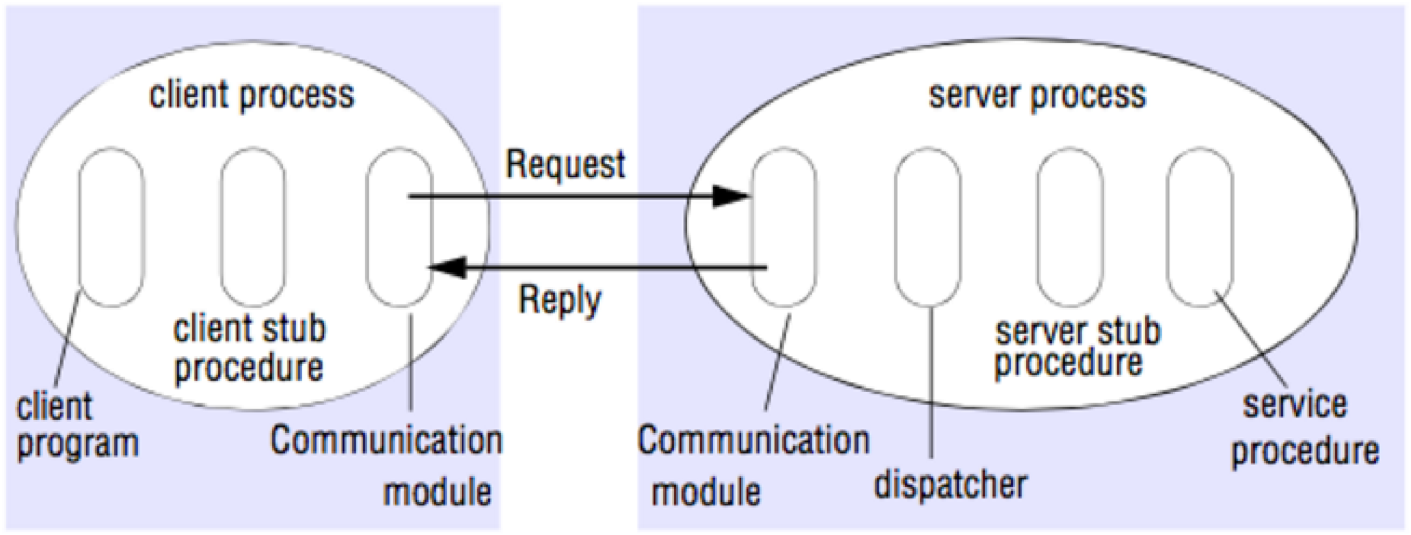
How do different components interact with each other? – communication protocols.

Here is a simple comparison of those protocols.

- UDP and TCP are both transport layer protocols. TCP is reliable and connection-based. UDP is connectionless and unreliable.
- HTTP is in the application layer and normally TCP based, since HTTP assumes a reliable transport.
- RPC, an application layer protocol, is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network), without the programmer explicitly coding the details for this remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. In an Object-Oriented Programming context, RPC is also called remote invocation or remote method invocation (RMI).

Further discussions

Since RPC is super useful, some interviewers may ask how RPC works. The following picture is a brief answer.



*Stub procedure: a local procedure that marshals the procedure identifier and the arguments into a request message, and then to send via its communication module to the server. When the reply message arrives, it unmarshals the results.

We do not have to implement our own RPC protocols. There are off-the-shelf frameworks.

- Google Protobuf: an open source RPC with only APIs but no RPC implementations. Smaller serialized data and slightly faster. Better documentations and cleaner APIs.
- Facebook Thrift: supports more languages, richer data structures: list, set, map, etc. that Protobuf does not support) Incomplete documentation and hard to find good examples.
  - User case: Hbase/Cassandra/Hypertable/Scrib/..
- Apache Avro: Avro is heavily used in the hadoop ecosystem and based on dynamic schemas in Json. It features dynamic typing, untagged data, and no manually-assigned field IDs.

Generally speaking, RPC is internally used by many tech companies for performance issues, but it is rather hard to debug and not flexible. So for public APIs, we tend to use HTTP APIs, and are usually following the RESTful style.

- REST (Representational state transfer of resources)
  - Best practice of HTTP API to interact with resources.
  - URL only decides the location. Headers (Accept and Content-Type, etc.) decide the representation. HTTP methods(GET/POST/PUT/DELETE) decide the state transfer.
  - minimize the coupling between client and server (a huge number of HTTP infras on various clients, data-marshalling).
  - stateless and scaling out.
  - service partitioning feasible.
  - used for public API.

2.2 Storage

2.2.1 Relational Database

Relational database is the default choice for most use cases, by reason of ACID (atomicity, consistency, isolation, and durability). One tricky thing is **consistency – it means that any transaction will bring database from one valid state to another, (different from the consistency in CAP**, which will be discussed in Section 2.3).

Schema Design and 3rd Normal Form (3NF)

To reduce redundancy and improve consistency, people follow 3NF when designing database schemas:

- 1NF: tabular, each row-column intersection contains only one value
- 2NF: only the primary key determines all the attributes
- 3NF: only the candidate keys determine all the attributes (and non-prime attributes do not depend on each other)

Db Proxy

What if we want to eliminate single point of failure? What if the dataset is too large for one single machine to hold? For MySQL, the answer is to use a DB proxy to distribute data, either by clustering or by sharding.

Clustering is a decentralized solution. Everything is automatic. Data is distributed, moved, rebalanced automatically. Nodes gossip with each other, (though it may cause group isolation).

Sharding is a centralized solution. If we get rid of properties of clustering that we don’ t like, sharding is what we get. Data is distributed manually and does not move. Nodes are not aware of each other.

2.2.2 NoSQL

In a regular Internet service, the read write ratio is about 100:1 to 1000:1. However, when reading from a hard disk, a database join operation is time consuming, and 99% of the time is spent on disk seek. Not to mention a distributed join operation across networks.

To optimize the read performance, **denormalization** is introduced by adding redundant data or by grouping data. These four categories of NoSQL are here to help.

Key-value Store

The abstraction of a KV store is a giant hashtable/hashmap/dictionary.

The main reason we want to use a key-value cache is to reduce latency for accessing active data. Achieve an O(1) read/write performance on a fast and expensive media (like memory or SSD), instead of a traditional O(logn) read/write on a slow and cheap media (typically hard drive).

There are three major factors to consider when we design the cache.

1. Pattern: How to cache? is it read-through/write-through/write-around/write-back/cache-aside?
2. Placement: Where to place the cache? client side/distinct layer/server side?
3. Replacement: When to expire/replace the data? LRU/LFU/ARC?

Out-of-box choices: Redis/Memcache? Redis supports data persistence while memcache does not. Riak, Berkeley DB, HamsterDB, Amazon Dynamo, Project Voldemort, etc.

Document Store

The abstraction of a document store is like a KV store, but documents, like XML, JSON, BSON, and so on, are stored in the value part of the pair.

The main reason we want to use a document store is for flexibility and performance. Flexibility is obtained by schemaless document, and performance is improved by breaking 3NF. Startup’ s business requirements are changing from time to time. Flexible schema empowers them to move fast.

Out-of-box choices: MongoDB, CouchDB, Terrastore, OrientDB, RavenDB, etc.

Column-oriented Store

The abstraction of a column-oriented store is like a giant nested map: ColumnFamily<RowKey, Columns<Name, Value, Timestamp>>.

The main reason we want to use a column-oriented store is that it is distributed, highly-available, and optimized for write.

Out-of-box choices: Cassandra, HBase, Hypertable, Amazon SimpleDB, etc.

Graph Database

As the name indicates, this database’ s abstraction is a graph. It allows us to store entities and the relationships between them.

If we use a relational database to store the graph, adding/removing relationships may involve schema changes and data movement, which is not the case when using a graph database. On the other hand, when we create tables in a relational database for the graph, we model based on the traversal we want; if the traversal changes, the data will have to change.

Out-of-box choices: Neo4J, Infinitegraph, OrientDB, FlockDB, etc.

2.3 CAP Theorem

When we design a distributed system, **trading off among CAP (consistency, availability, and partition tolerance)** is almost the first thing we want to consider.

- Consistency: all nodes see the same data at the same time
- Availability: a guarantee that every request receives a response about whether it succeeded or failed
- Partition tolerance: system continues to operate despite arbitrary message loss or failure of part of the system

In a distributed context, the choice is between CP and AP. Unfortunately, CA is just a joke, because single point of failure is a red flag in the real distributed systems world.

To ensure consistency, there are some popular protocols to consider: 2PC, eventual consistency (vector clock + RWN), Paxos, In-Sync Replica, etc.

To ensure availability, we can add replicas for the data. As to components of the whole system, people usually do cold standby, warm standby, hot standby, and active-active to handle the failover.

- Tian’ s notes on big data from a programmer’ s perspective
- 100 open source big data architecture papers
- System design interview for IT companies
- The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, Volume 2
- NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence
- MongoDB Applied Design Patterns
- Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications

https://www.ibm.com/developerworks/community/blogs/RohitShetty/entry/high_availability_cold_warm_hot?lang=en

- **Active-Active (Load Balanced)**: In this method both the primary and secondary systems are active and processing requests in parallel. Data replication happens through software capabilities and would be bi-directional. This generally provides a recovery time that is instantaneous.