Appendix – Notes taken from the Udemy course “Pragmatic System Design” by Alexey Soshin

Product Goals (Functional Requirements)

- What the product does
- Product's main features
- Use cases
- Examples:
    - External interfaces (API)
    - Authentication
    - Payments
    - Analytics
    - Recommendations
    - Search
    - Cart
    - Feed
    - User Data
    - Types of Users (Driver/Rider in Uber)

Technical Goals (Non-Functional Requirements)

- How the product works
- User expectation
- Get realistic metrics
    - Peak active users
    - Total number of users
    - Transaction/User/Day
- Mandatory/Desirable goals:
    - Performance
    - Reliable
    - Scalable
    - Consistent
    - Latency
    - Volume
    - Secure
    - Throughput (Calculated request/s)

Protocols

- TCP (Transmission Control Protocol)
    - Reliable
        - Receiver either confirms packets were received or times out and sender sends again
    - Ordered
        - Packets are numbered and makes sure it's received in order
    - Error-checked
        - Checksum
    - Slower than some other protocols
    - Types
        - WebSockets
            - Messaging service

  - Duplex protocol
  - More efficient than polling with HTTP
  - Connection only established once
  - Real time message delivery to the client
  - No defined protocols like HTTP
  - Load Balancers may have troubles as some are meant for short connections
- HTTP
  - Methods
    - GET
      - Gets entity
    - POST
      - Create new entity
      - Sometimes used instead of GET since GET has limit on length
    - PUT
      - Update entity
    - PATCH
      - Partial update (rarely used)
    - DELETE
      - Deleting entity
  - Types
    - REST
      - Use HTTP methods properly
      - May pass in query parameters like pagination
      - Use PUT as Boolean changer, use POST for else
    - gRPC (g Remote Procedure Call)
      - Doesn’t support browsers
      - App and server may use different call parameters
      - Parameters converted via stubs
      - Server reconverts call via stubs to how it understands it and executes the function
      - Sends back stub and client reconverts return using stub
      - Put RPC and language into the generator and you get your stub
        - Does not have any business logic
    - GraphQL (Graph Query Language)
      - Takes care of REST's over fetching and under fetching problem
        - May ask the server for user's name on REST and get back name, birthday, age, etc.
        - May ask the server for user's friends on REST and get back the list of IDs of friends, where we need to send REST request for each friend ID
      - Request and responses are in JSON
      - Let's you define the fields to return
      - Let's you define which nested entities to return
      - Great for reporting systems or mobile apps

            - Results are less cacheable

- UDP (User Datagram Protocol)
    - Less reliable
    - Unordered
    - Fast
    - Used for something that updates frequently (Constant stream of data)
    - Example of usage
        - Monitoring metrics
        - Video Streaming (Twitch)
        - Gaming

- Choosing between the protocols
    - External API

REST	Yes
WebSockets	Yes
gRPC	No
GraphQL	Yes
UDP	Yes
Long Polling	Yes

- Bi-Directional

REST	No
WebSockets	Yes
gRPC	No
GraphQL	No
UDP	Yes
Long Polling	Somewhat

- High Throughput

REST	No
WebSockets	Yes
gRPC	Yes
GraphQL	No
UDP	Yes
Long Polling	No

- Web Browser Support

REST	Yes
WebSockets	Yes
gRPC	No
GraphQL	Yes

UDP	No
Long Polling	Yes

Load Balancer

- Either a physical machine or software
- Very reliable
- Distributes loads by:
    - Round Robin
        - Easy to implement
        - Even number of connections
        - Load may not be distributed evenly
    - Least connection
    - Resource based
    - Weighted variants of the above
    - Random
- Types
    - Layer 4
        - Transport layer
        - Access to TCP or UDP, IP, Port
    - Layer 7
        - Application layer
        - Has everything layer 4 has plus
        - HTTP headers, cookies, payload

CDN

- Cache for static assets
- Decreases latency
- Increases complexity
- These assets shouldn't change too often
- Often stores:
    - Images
    - HTML
    - CSS
    - JavaScript
- Types
    - Push
        - Pushed to all the CDNs when uploaded to the server
        - Good when you don't have much static content
        - Slow and expensive for large amount
    - Pull
        - Lazy
        - Slow for first user
        - Lots of static content

Cache

- Types of strategies
    - Cache aside

        - Most common
        - App has access to both cache and storage
        - If in cache, use case, if not, go to the storage
        - Good since it caches only what's needed
        - Bad since cache misses are expensive
        - Stale data
    - Read through
        - App always interacts with cache
        - If not in cache, cache will fetch from data, save, then return to app
        - Cache misses are expensive
        - Stale data
    - Write through
        - Updates cache whenever app updates data, then cache updates the storage
        - Most up-to-date data
        - Writes are expensive
        - May write data to cache that no one needs
    - Write behind
        - Similar to write through but cache waits until timeout
        - Writes are cheap
        - Reduces load on storage
        - Poor reliability
        - Lack of consistency as storage is updated long after write
- Eviction Policies
    - LRU
        - Very efficient
        - If lots of new keys are requested at once, popular keys may be evicted
    - LFU
        - More cost
        - Key counter
- Redis
    - In-Memory
    - Key-Value store
    - Limited by RAM (500GB to 1TB)
    - Supports 100k+ requests per second per node
    - No native support for JSON
    - Time to Live (TTL) support
    - Stores data to disk but can lose recent data

Queues

- Types
    - Message queue
        - Pusher and Consumer
        - For payment, order service may send payment data to queue and queue chooses a payment service
        - Retry to another payment service if one payment service is busy
        - Delivery exactly once
        - Messages can arrive out of order
    - Pub/Sub (Kafka)

- Notify other services on what happened (payment success/fail)
- Delivery at least once
- Messages are always in order

- Kafka
    - Pub/Sub
    - Higher throughput (100k+ events per second)
    - Poor latency
    - If there are more consumers than partitions, some consumers won't receive any events
    - Each consumer reads data at own pace (Slow consumer don't affect queue performance)

Concurrency

- Processes
    - Interprocess communication
        - File
            - Multiple processes can access file at the same time
        - Signal
            - Send signal to process to do something
        - Socket
            - Client process connects to 8080, Server process listens to 8080
        - Pipe
            - Output of one process is the input of another process which uses that to output
- Thread
    - Light weight than process
    - If CPU and OS can handle more threads, it can run multiple threads at the same time
    - Making new thread is a bit slow
    - Threads can share the same resources
        - Use locks to prevent two threads writing to one resource at the same time
    - Thread pool

Database

- ACID
    - Atomicity - If there's a failure in the middle of the code, revert the state to original
    - Consistency - Database obeys the constraints and cannot have an impossible state
    - Isolation - Until one transaction is committed, other users cannot see the update
    - Durability - Data is safe after commit
- Sharding (Horizontal Scaling)
    - Shard to make Db smaller and manageable
    - Examples:
        - Gio-sharding (Tenant)
            - Easy to add new region/shard
            - Uneven distribution
        - Hash based sharding
            - Even distribution
            - Adding new shards is difficult
            - Weaker consistency (no foreign keys)
            - Some weakness fixed by shard locator but increases complexity

            - Use hash range to improve adding shard (resharding)
- Partitioning (Horizontal Scaling)
    - Break one large table to multiple tables by:
        - List of values (Placed, In-progress, Completed orders)
            - Smaller tables = Faster queries
            - Uneven data distribution
            - Must move data between tables
        - Range of dates
            - Smaller tables = Faster queries
            - Great for deleting old data
            - Uneven data
        - Hash of key
            - Even distribution of keys
            - Difficult to query range of data
- NoSQL
    - CAP Theorem
        - Consistency - All nodes see the same data, may not be able to write
        - Availability - All nodes are able to write, may not see the same data
        - Partition Tolerance - Not an option, it's a must