# Common HLD terms

# Approaching

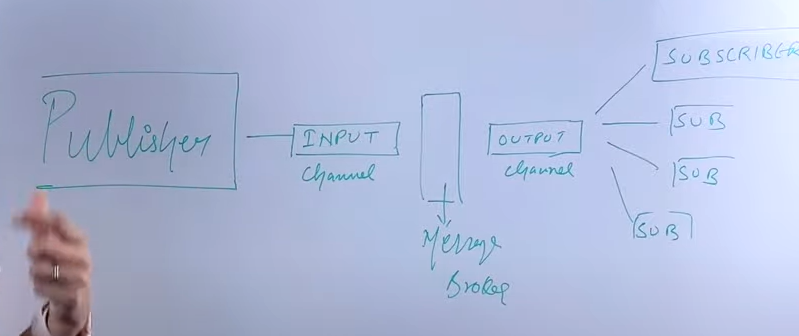
1. Functional requirement
2. Non-functional requirement
3. Use case diagram/para
4. ER/ stick diagram with various actors.
5. HLD
6. LLD

# Common concept

* + HA: High availability
  + Scalability: clustering, horizontal and vertical scaling
  + Database: SQL, NoSQL(unstructured). Based on structure and ACID prop
  + Sharding: horizontal slicing of DB
  + Load balancer: various logic can be used.
  + Distributed and hyper converged
  + Cache prefetch
  + Logging and data for analytics

SYSTEM DESIGN: <https://lnkd.in/dKPi59vK>

* System design: user + requirement + system components
* Components
  + Data: Text, image etc.
  + Database: SQL, NOSQL
  + Application: Java, c++
  + Cache: memecache, redis
  + Message q: Kafka, rabbit mQ
  + Infra: cloud
  + Communication: API, RPC, messages
* Client server architecture
  + Thick(ms word) and thin(netflix) client: based on how much logic sits on client
  + 2 tier: client + server
  + 3 tier: client + (server)logic + DB
  + N tier: inserting cache, proxy, load balancer
* Proxy: between client and server, can be used for cache, DDos, load balance etc.
  + Forward: client->proxy->server
  + Reverse: client<-proxy<-server
* Data and data flow: Data format at various stage of system
  + Client ( video), appl(JSON), bakend (DB) , NW(packets) etc
* Types of DB:
  + Relational DB: scema and ACID (atomic, consistent, Isolation, Durable), vertical scaling
  + Non Reln: Horizontal scaling ex. Key-Value, Document(json Style) DB, Column DB(similar to reln but no ACID), Search DBs, etc.
* Application and services:
  + Monolith vs Micro service
* APIs:
  + Private, public API, Web API, SDK and Lib API
  + RPC, SOAP, REST
* Cache:
  + LRU cache(Doubly linked list for LRU + hash(key -> node add))
  + FIFO, LIFO
  + A picture containing graphical user interface

    Description automatically generated
* REST API
  + GET, POST(create), PUT(update),DELETE
* Message Q
  + Sync(call) and Async(msg) communication
  + Ordered/unordered Q
  + Consumption
    - One to one: Producer consumer
    - One to many: publisher subscriber
  + Producer consumer (lock/semaphore etc.)
    - Priority (heap sort) etc.
* Publisher-subscriber: order placing on onlie platform
  + 
  + Diagram

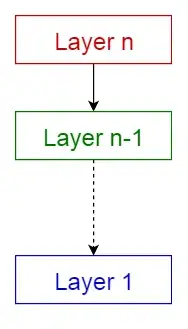
    Description automatically generated with medium confidence
  + Async, decouple, Load balancing, deferred processing, Data streaming.
* Performance matrix:
  + Throughput, bandwidth, latency, Response time
  + Components: app, DB, Network, cache, disk, OS, Message Q, services, instance
  + Memory/CPU
* Fault/Failure: Fault is point of failure.
  + Fault tolerance: ex. HA etc.
* Scaling:
  + Vertical and horizontal
* DB replication:
  + Sync (wait for all ack), async (no ack), semi sync (wait for 1 replica ack), partial (for few table)
* CAP: (consistency (returns consistent data), availability (returns data), partitioning (tolerate link breakage between HA))
  + Only two can achieved fully: CAP theorem.
* Database Sharding: divide mostly horizontal.
  + Vertical partition: diff colon diff machine
  + Horizontal: Sharding, parts of table
  + Physical (in diff DB) vs logical (in same DB)
  + Advantage: less query time, geo based, No single pt of failure
  + Logarithmic (static) vs dynamic sharding
  + Key based sharding: hash function to decide shard. Any static column can be chosen as shard key. Even combination of col can be used.
  + Range based: Any static column can be chosen as shard key. Even combination of col can be used.
  + Dir based: a db/dir to hold key ->destination shard mapping.
* Hashing: convert data to hash, mod it to save in map
  + Collision: chaining and closed hashing (linear probe, quadratic probe, double hash)
* System design interview:
  + Functional and non-functional requirement
  + Capacity estimation
  + Mistakes to avoid not clarifying requirements, not talking about capacity estimation, Not discussing the trade-off, Not finding the faults with system, Say I don’t know.
* Proxy server:
  + Server before actual server
  + Does task like clubbing req, encryption, decryption , DDos , various checks etc.

# Architectural pattern

https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013

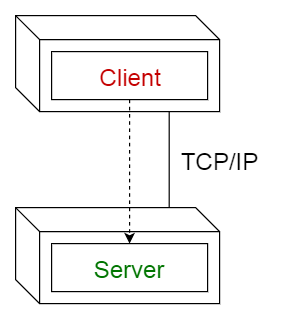
## Layered

* 1. Stacked in layers
  2. Ex: raid->FS->backupsoftware



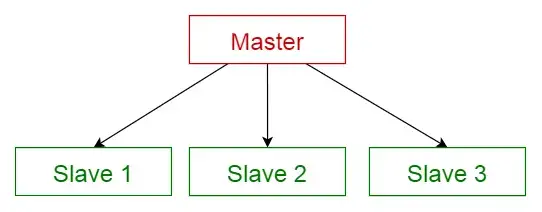
## Client server

* 1. One server , multiple client
  2. Commserver and FS client, web client /server



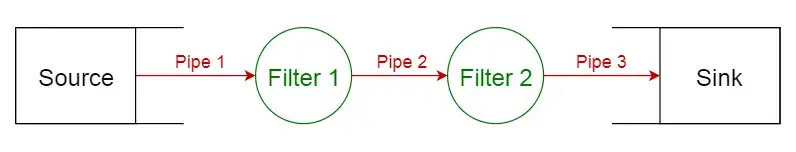
## Master slave

* 1. Master distributes the work and compute the final result
  2. Multi Node backup coordinator and controller



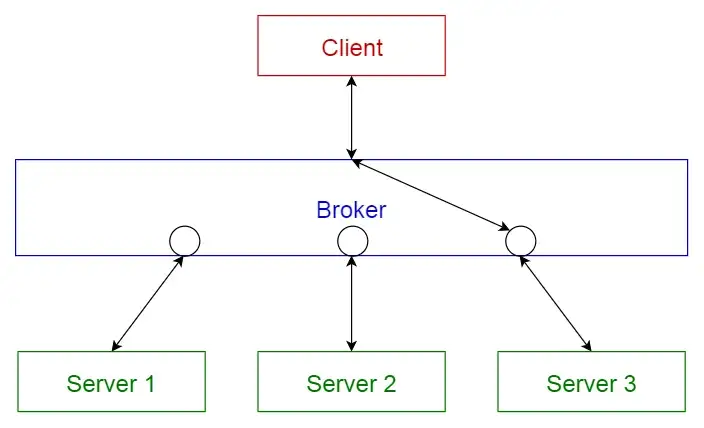
## Pipe filter

* 1. Pipe the data to filter/processor,which processes and pass to next pipe then next filter and so on
  2. Pipe is for buffer/sync
  3. Data analytics examples



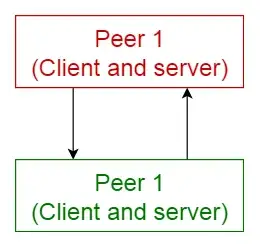
## Broker

* 1. A broker in between , publisher publish msg, broker knows subscriber, broker passes msg to all subscriber
  2. Publisher subscriber



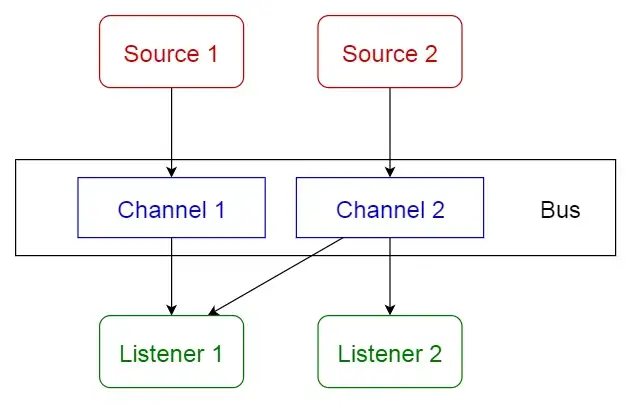
## Peer-to-peer

* 1. Peers working together to perform task,
  2. Startupgui and onetouchStarter.dll
  3. blockchain



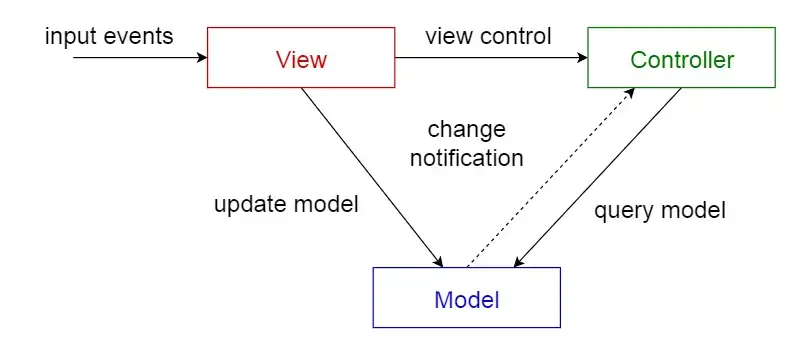
## Event bus

1. Source publishes event which listener acts upon



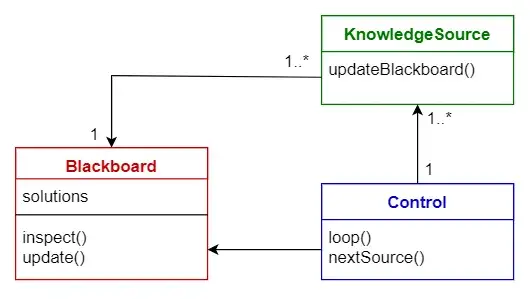
## Model view controller

1. Model has core functionality, view has view, controller is mediator
2. MFC, WPF framework



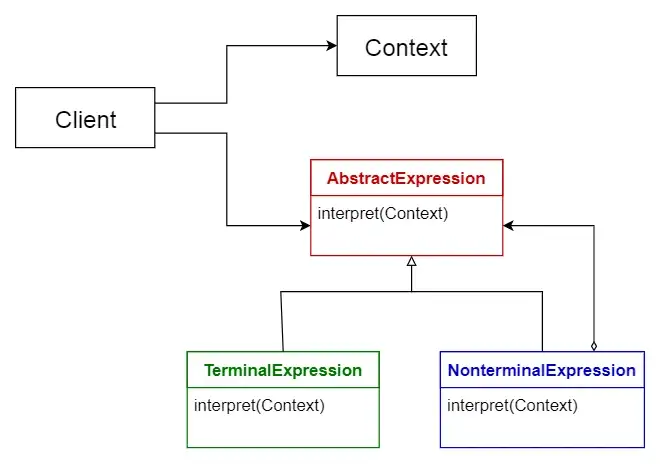
## Blackboard

1. Blackboard has objects from solution space, knowledge sources are specialised modules with their own implementation, control configure and executes them.
2. AIML models



## Interpreter

* 1. Idea is to have class for each symbol of lang
  2. Language interpreter



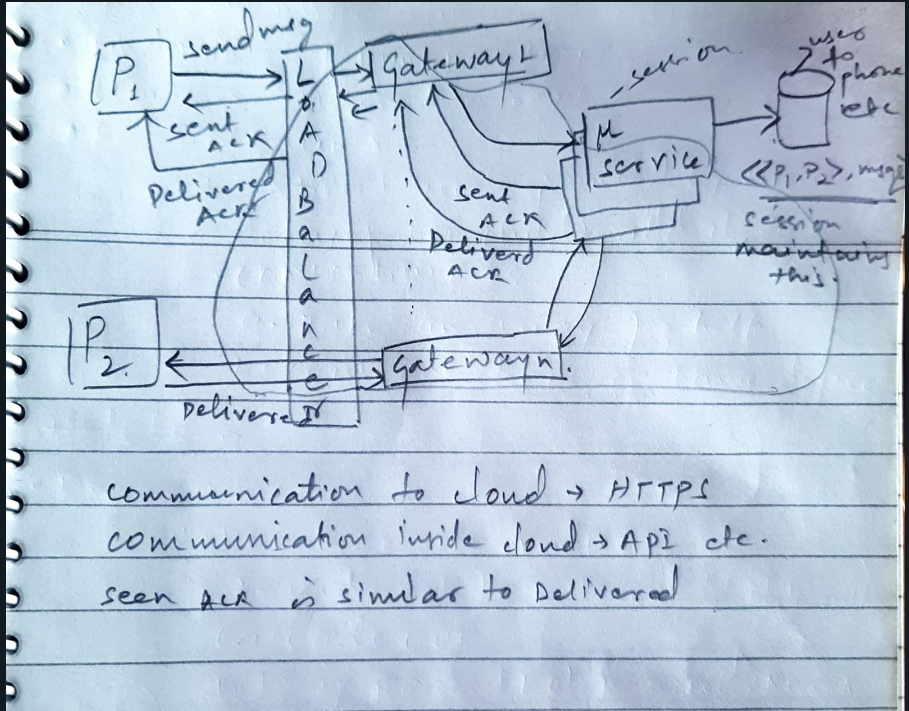
WhatsApp design: [Explaining WhatsApp System Design Basics to Papa!! Anyone can understand how messaging works!! - YouTube](https://www.youtube.com/watch?v=LsH-t75P544)

Functional requirement:

1. P1 send message to p2. Gets ack for send, delivered and seen.

Non-functional requirement:

1. Support million users :
   1. use DB sharding, load balancing, micro service for this.
   2. Make it robust by adding HA at various level.

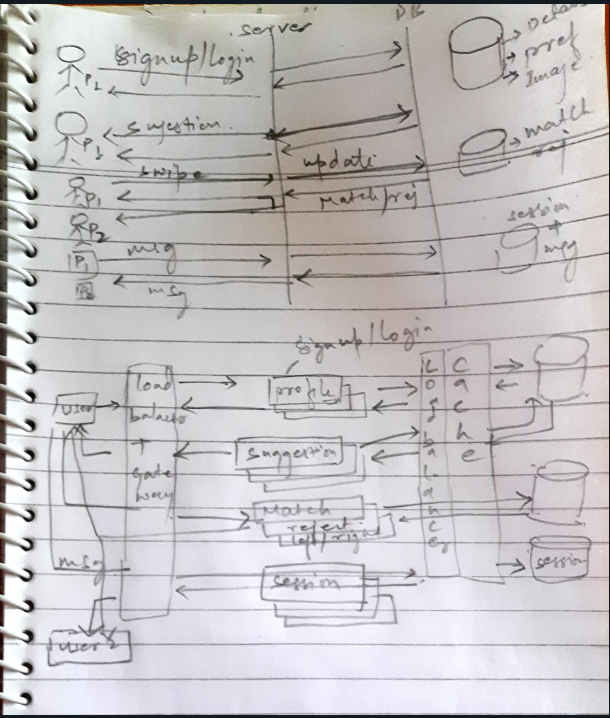


* Websocket used to send final message as HTTP is client to server protocol
* For group message, use group service, which uses queue to route to session

<https://lnkd.in/dQzUk9Ek> gaurav sen system design videos

Tinder

* Functional requirement
  + People signup/login: provide details, including preference.
  + Swipes
  + Match
  + messages
* Non-functional requirement:
  + Million users



* Use Cassandra/ distributed DBs for quick query/ sharding
* Images can be stored in File sys / blob

# Notes from SysDesign.pdf(bytebyteGo):

* Don’t save password hash in DB rather save hash(password + salt) and save salt too
* Check all the Diagrams

# Notes from SysDesign.pdf(Ravi tondon):

* Elements of distributed systems
  + Micro services
  + Load balancers
  + DB
  + Cache
  + File system: file, block and object
  + Network
  + Message Q