**Solid Design Principal:**

**1.) Single Responsibility Principle**

**// single responsibility principle - bad example**

interface IEmail {

public void setSender(String sender);

public void setReceiver(String receiver);

public void setContent(String content);

}

class Email implements IEmail {

public void setSender(String sender) {// set sender; }

public void setReceiver(String receiver) {// set receiver; }

public void setContent(String content) {// set content; }

}

**// single responsibility principle - good example**

interface IEmail {

public void setSender(String sender);

public void setReceiver(String receiver);

public void setContent(IContent content);

}

interface Content {

public String getAsString(); // used for serialization

}

class Email implements IEmail {

public void setSender(String sender) {// set sender; }

public void setReceiver(String receiver) {// set receiver; }

public void setContent(IContent content) {// set content; }

}

**2.) Open Closed Principle:**

**// Open-Close Principle - Bad example**

class GraphicEditor {

public void drawShape(Shape s) {

if (s.m\_type==1)

drawRectangle(s);

else if (s.m\_type==2)

drawCircle(s);

}

public void drawCircle(Circle r) {....}

public void drawRectangle(Rectangle r) {....}

}

class Shape { int m\_type; }

class Rectangle extends Shape { Rectangle() {

super.m\_type=1;}

}

class Circle extends Shape

{Circle() { super.m\_type=2; } }

**// Open-Close Principle - Good example**

class GraphicEditor {

public void drawShape(Shape s) {

s.draw(); } }

class Shape {

abstract void draw(); }

class Rectangle extends Shape {

public void draw() {

// draw the rectangle

}

}

**3.) Liskov’s Substitution Principle:**

Likov's Substitution Principle states that if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.

**//Violation of Likov's Substitution Principle**

class Rectangle

{

protected int m\_width; protected int m\_height;

public void setWidth(int width) { m\_width = width; }

public void setHeight(int height) { m\_height = height; }

public int getWidth() { return m\_width;}

public int getHeight() { return m\_height; }

public int getArea(){ return m\_width \* m\_height; }

}

class Square extends Rectangle {

public void setWidth(int width){

m\_width = width;

m\_height = width;}

public void setHeight(int height){

m\_width = height;

m\_height = height;

}}

class LspTest{

private static Rectangle getNewRectangle() { return new Square();}

public static void main (String args[]){

Rectangle r = LspTest.getNewRectangle();

r.setWidth(5); r.setHeight(10);

System.out.println(r.getArea());

}

}

**4.) Interface Segregation Principle:**

**// interface segregation principle - bad example**

interface IWorker {

public void work(); public void eat();

}

class Worker implements IWorker{

public void work() {

// ....working

}

public void eat() {

// ...... eating in launch break

}

}

class SuperWorker implements IWorker{

public void work() {

//.... working much more

}

public void eat() {

//.... eating in launch break

}

}

class Manager {

IWorker worker;

public void setWorker(IWorker w) {

worker=w;

}

public void manage() {

worker.work();

}

}

**// interface segregation principle - good example**

interface IWorker extends Feedable, Workable {}

interface IWorkable { public void work();}

interface IFeedable { public void eat(); }

class Worker implements IWorkable, IFeedable

{ public void work() { // ....working }

public void eat() {

//.... eating in launch break }

}

class Robot implements IWorkable{

public void work() { // ....working }

}

class SuperWorker implements IWorkable, IFeedable{

public void work() {

//.... working much more

}

public void eat() {

//.... eating in launch break

}

}

class Manager {

Workable worker;

public void setWorker(Workable w) {

worker=w; }

public void manage() {

worker.work();

}

}

**5.) Dependency Inversion Principle:**

**// Dependency Inversion Principle - Bad example**

class Worker {

public void work() { // ....working }

}

class Manager {

Worker worker;

public void setWorker(Worker w) {

worker = w;

}

public void manage() {

worker.work();

}}

class SuperWorker {

public void work() { }

}

**// Dependency Inversion Principle - Good example**

interface IWorker { public void work(); }

class Worker implements IWorker{

public void work() {

// ....working }

}

class SuperWorker implements IWorker{

public void work() {}

}

class Manager {

IWorker worker;

public void setWorker(IWorker w) {

worker = w;

}

public void manage() {

worker.work();}

}

**HLD**

**1.) System Design URL Shortener:**

**Functional Requirements:**

* Given a long URL give short URL
* When user clicks on short URL, redirect to original URL with minimum latency
* User Feed Display
* Comment and like a post

**Non-Functional Requirements:**

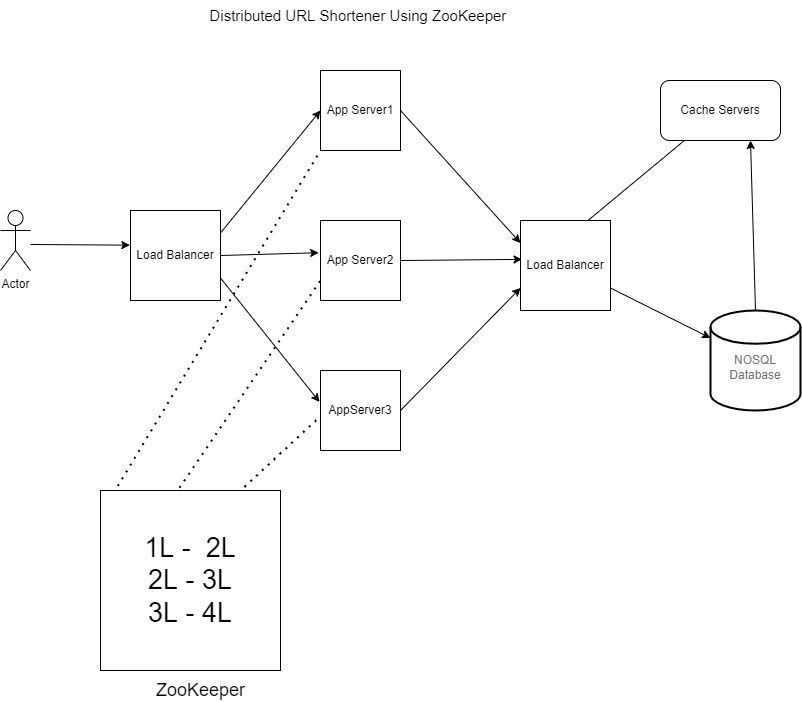
* High Available (Consistency not on priority)
* Scalable

**APIS:**

* createTinyUrl(Long url)
* getLongUrl(Short Url)

**Size of Database Required:**

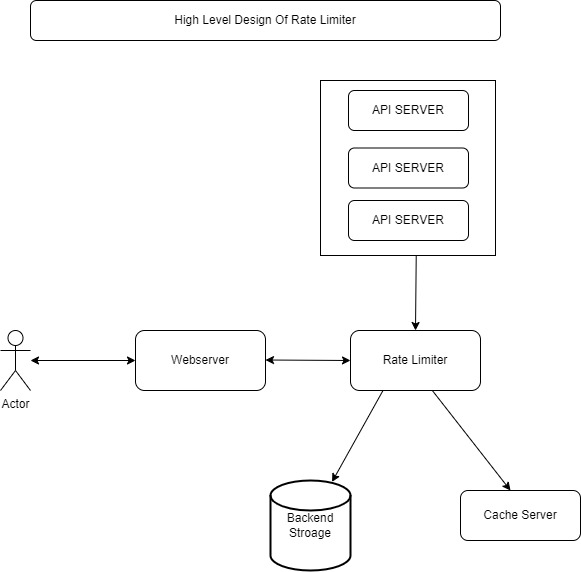
* Each URL mapping size – 2kb
* Number of users per month 10 M – 10M\*2KB = 20GB per month – 1TB 5 years



**2.) Designing an API Rate Limiter:**

**How to implement rate limiter: Most common algorithms are:**

* Token Bucket – 5 ball per min
* Leaky Bucket – Put tasks in queue if queue is full return too many requests return after sometime – (Effective and easy to implement)
* Fixed Window – similar to token bucket – time range is fixed for ex – 5:00 to 05:59 - 5 request – Problems (Clustering around boundary, race condition in case of distributed systems)
* Sliding Window: Better the fixed window : it will check 5 request is previous window if it’s there then it will not accept any new request;



**3.) System Design Instagram:**

**Functional Requirements:**

* Upload Photo/Create Post
* View Photo/Post
* Search Users
* User Feed Display
* Comment and like a post

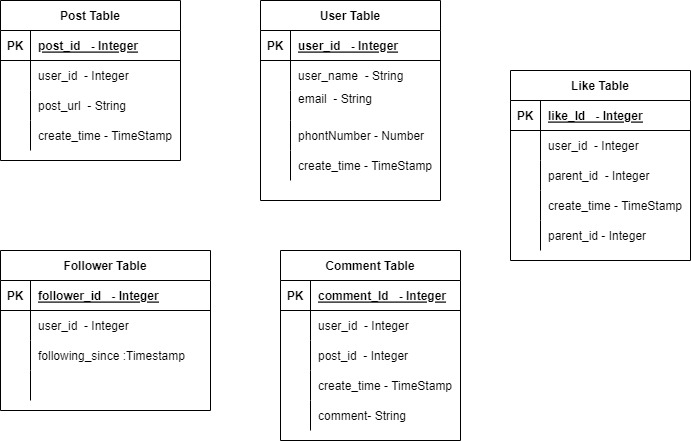
**Non-Functional Requirements:**

* High Available (Consistency not on priority)
* Reliable System hence partition tolerant - photo should not be lost
* Low latency while user feed service
* Read Heavy

Capacity Estimation:

* 100M users - 1M active users - Each Image or post size 1MB
* 1M\*1MB - 975GB perday
* Per year - 975\*365 = 350TB
* sharding will be on post\_id

**Database Schema:**



User Service: getUserDetails, postUserDetails, updateUserDetails

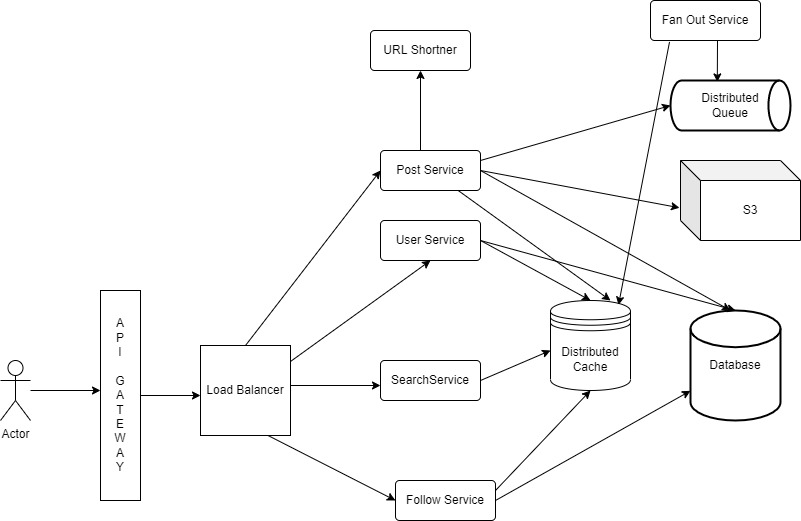
Follow Service: followUser, getListOfUserFollowedByAUser

Post Service: uploadPost, getPost, getUserFeed

Comment And Like Service: addCommentToThePost, getCommentForThePost, addLikeToThePost, getNumberOfLikesForThePost, getListOfUsersWhoLikeThePost

**Promise Based Cache:**

* Cache miss happen for same object for multiple request
* Rather than all request going to db and fetching result and setting in DB
* 1st request creates a promise object in cache
* Subsequent request will cache miss will check if a promise is present in a cache
* If promise is present, all subsequent request will wait for the promise to get completed



**Database Types:**

RDBMS

NOSQL DB – Document – Columnar – Key Value – Graph

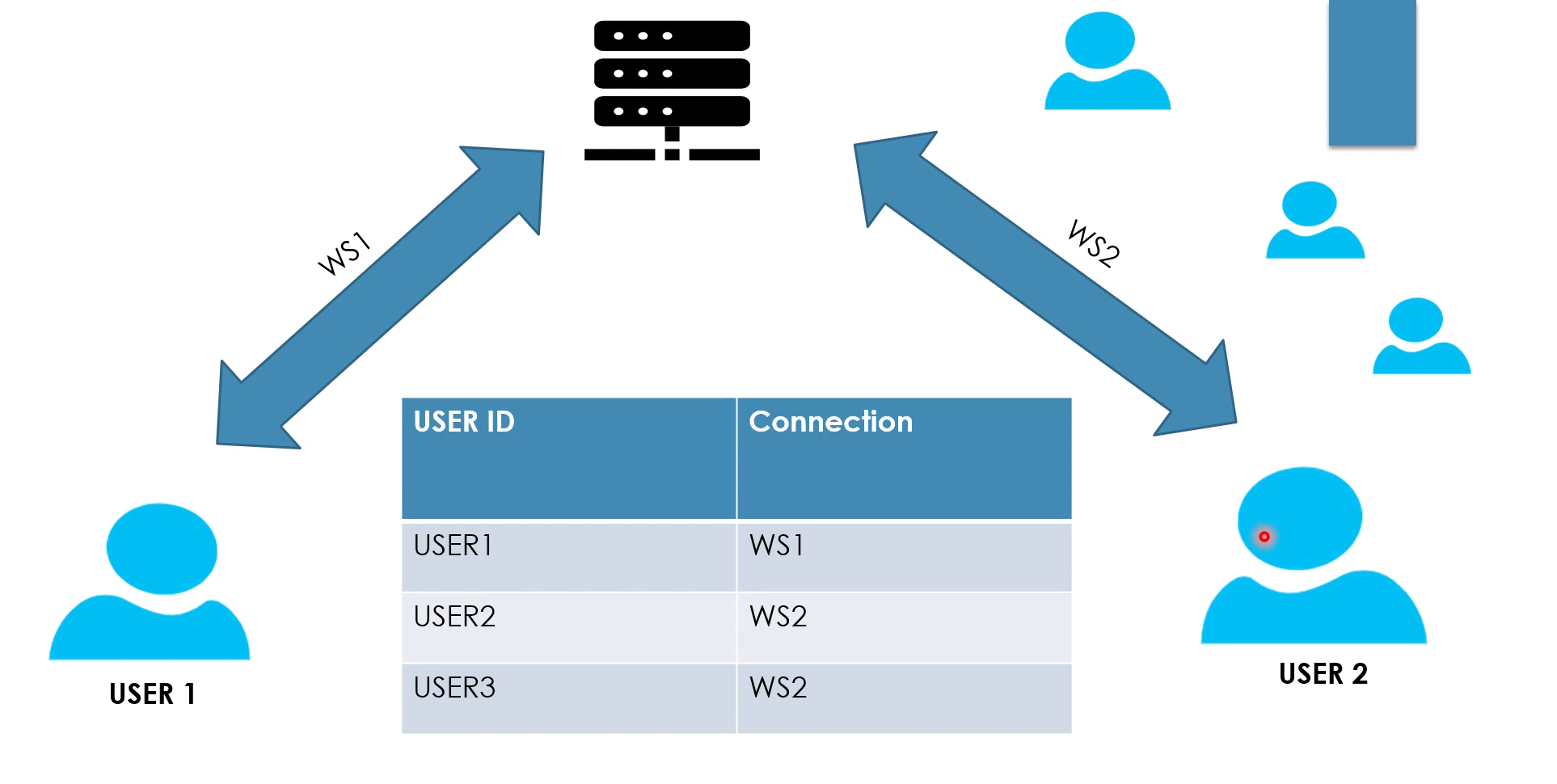
**4.) System Design Facebook Messenger/WhatsApp/Share Chat:**

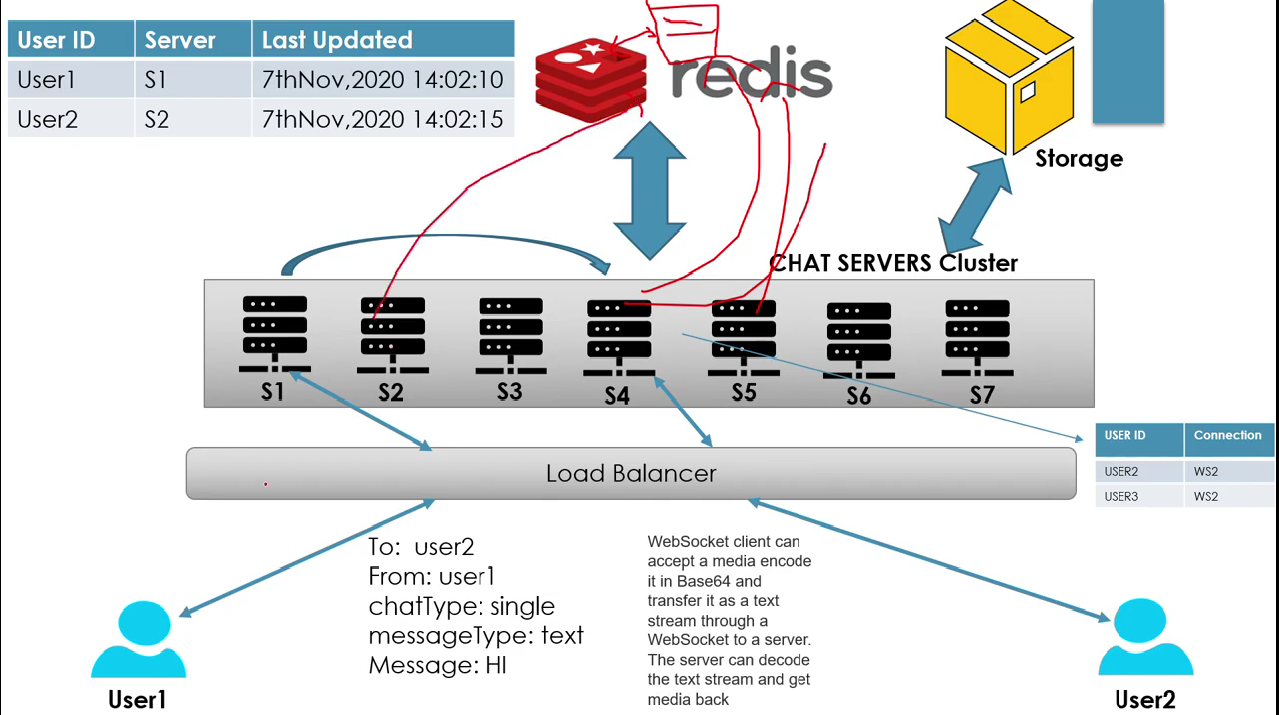
**Functional Requirements:**

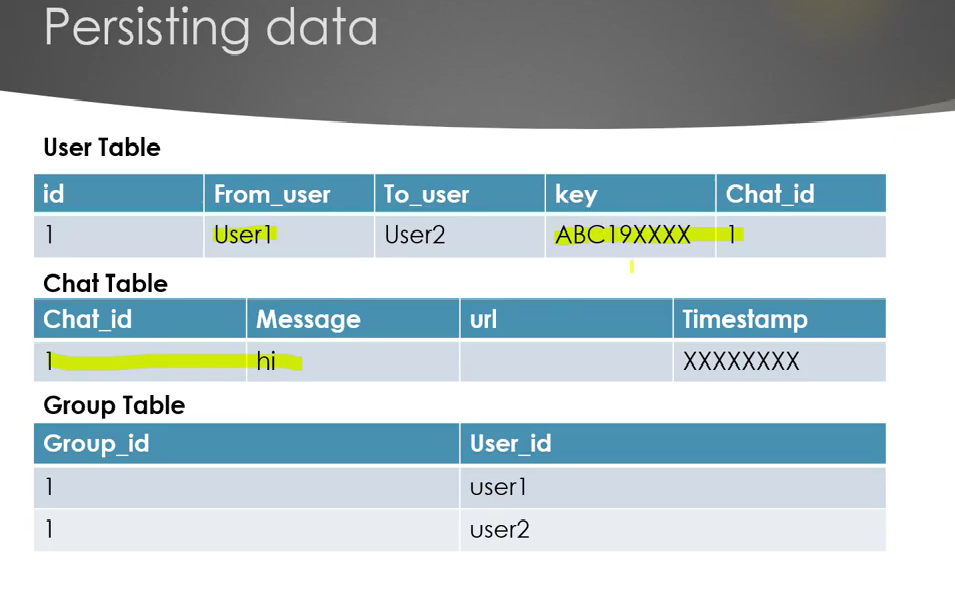
* One to One Message
* Sent/Delivered/Seen
* Last Online
* Media Message (Picture/Audio/Video)
* Persistent chat required or not?)

**Non-Functional Requirements:**

* Real Time Experience
* Consistent
* Scalable
* High Availability







**5.) System Design Cab Booking System Ola/Uber/Grab:**

**Functional Requirements:**

* Book Cab
* Show ETA and approx. price of the ride
* Track Location
* Of Course profile of rider and driver

**Non-Functional Requirements:**

* High Availability
* Scalable
* Low Latency – Response Time
* Consistent

