

Understanding Fault and Fault Tolerant Techniques

3 majors concerns of System design:

1. Reliability
2. Scalability
3. Maintability

Reliability:

Basic reliability is, system should work. System should continue to work even if some components go down.

Say we want to get mail from inbox

Mail ---> get (mail, inbox). We are sending a request to the system to return all the mails. The system responds mail sent. Whatever operation we want to perform, the system performs the exact operation

Whenever user makes mistake, lets say User writes only 'a' in the name field, which is not correct. So I will provide some kind of validation, for that name field, to stop this user. Our system should perform the way we wanted. Even if the user is trying to break the application, it is our responsibility to stop user.

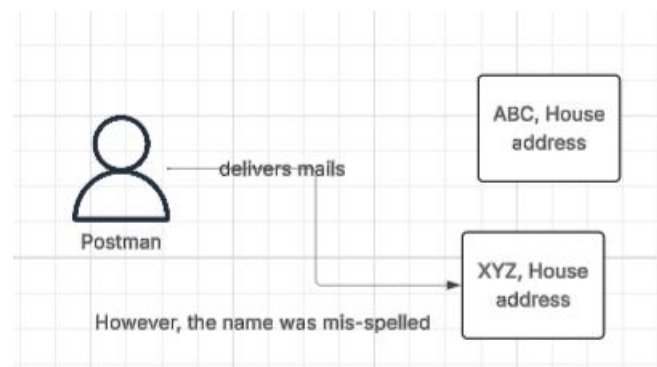
System should work

System should perform the expected operation

System should be smart enough to handle those user errors

Even if our system is reliable, still things could go wrong. When can things go wrong? And how do we categories? When something goes wrong in the system, it is categorized as 'ERROR' . Error is we want to perform something and it doesn't meet expected performance.

Error / Failure / Fault three things we use them interchangeably but they are not the same



Scenario 1: Instead of delivering to ABC's house the mail is delivered to XYZ's house. Is this a failure, error, or fault?

This is the error done by postman.

When we want to get emails from Inbox and instead if it provides from Sent folder then again it is an error. Same thing, if a page is unavailable and we are getting a '404' message then it is an error.

Eventually, any error or fault could lead to a failure of the system though. Say we have developed an application without any authentication then it is a huge error on the developer side, then any user can access any resources.

Scenario 2, you are expecting a post today, but the postman is on leave. This is not an error, it is fault of the postoffice that they rely on a single postman. In case, postman is sick, the postoffice should give the mails to some other postman still to deliver the mails. If we have only one webserver for our application and it goes down with no errors, it is again a fault. Fault is nobody's mistake/error but the

thing that is supposed to perform, didn't. basically the resource is down and it is a fault. Eventually fault will direct to failure only. Failure is the response of error or fault. Eventually, error and fault fail our entire system. Whenever a system stops working then it is a failure.

Prevention is better than cure

If don't prevent our systems from error or fault, the system will eventually crash or fail.

We as a System designer should capture all the possible faults and errors in advance.

Types of faults/errors:

What can go wrong in a system

1. Hardware fault --> hardware faults are random and non-deterministic by nature. Any server can fail and we cannot pre-identify server failures

- a) HDD fail
- b) Server down
- c) RAM/memory corruption
- d) Out of memory
- e) Storage is full
- f) Network cable damage
- g) No power/power supply shutdown
- h) Overheating
- i) Database corruption

2. Software fault --> How software can destroy application. It is not random, if it gives an error at a particular request, it will give errors at that particular request. This is fixable since it is our code issue. Software fault is more dangerous than Hardware fault because Software fault will happen again and it is guaranteed. Therefore, we got to identify, capture the bug and fix asap.

- a) Bad code
- b) Virus, malware
- c) Bad Exception handling
- d) Testing
- e) Configuration
- f) API handling
- g) Edge case handling
- h) Deployment issues (merging with a different code, branch)
- i) Backward compatibility (if we see an issue in V2, we should be able to rollback to V1)
- j) Performance issue

3. Human errors --> Humans make mistakes, sometimes silly errors sometimes huge. The most unreliable component of the system is apparently humans. Also, at the same time, it is the most powerful resource that can design the system and fix the system.

- a) Most unreliable
- b) Most powerful

Scalability:

System should work when the load increases

What's load?

- Number of requests per time
- Number of concurrent active users per time
- Memory usage at a particular time
- Incoming tickets or API calls
- Network calls and how heavy they are
- Transaction per second
- Queries per second for API
- Write per second for DB

These are all called Load parameters

Because of one or two, we have to scale our system

All these parameters combinedly tell us how our system is performing

When you increase the load parameters without increasing the system resources

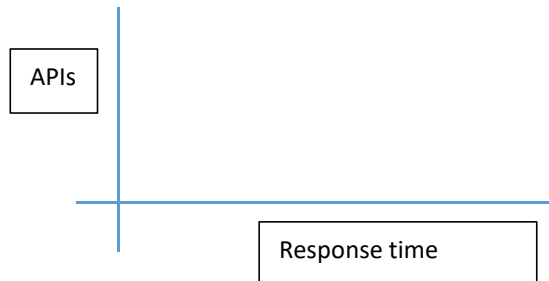
If you increase the load parameter, how your system is going to take it

How much system resource is required to handle that particular load

Two questions for performance:

1. If load increases, how is the system performance affected? (if API latency increases, then server is not able to process the request in the same time that means, system upgradation is required. Then how much upgradation is required?)
2. If load increases again, how many resources / instances do we need? How to scale?

Say we have multiple APIs



Now how do we calculate the response time?

One way is Average of response times but it doesn't give us the clear picture. Average is not a reliable metric.

Another metric is Percentile

First we sort response times in ascending order then we pick one response time. Lets say P90 is 110ms that means 90% of the response times are under 110. 90% of data points are under 110 that's p90. P50 is 4ms that means 50% of response times are under 4ms. We see that there is huge difference between P50 and P90. Normally we consider P95 or P99 and never P100. If we use P100, then all values are less than P100 and the largest response time or the last response time is 150ms. Mostly, it is of no use because only some APIs would take much greater time than normal time. P100 is usually cases that cannot be fixed. We can identify the APIs that are actually taking more than normal time.