# Chan and Golab: Fault Tolerance

## Introduction

Things fuck up. That's the unavoidable truth of life. Travel plans, software, relationships, you name it. If there's a way for it to go right, there's twelve ways for it to go wrong. But the question is - what do we do when it happens?

Unfortunately these notes won't undo your argument with your girlfriend, but they will teach you about software failures and how to handle them.

## Definitions and Background

Fault tolerance is quite closely related to the concept of dependability, which consists of the following four things:

Availability - how often is it operating correctly at any given point in time?
- ex. 95% available

Reliability - how long does it work for until a failure occurs?
- ex. usually runs for three weeks without crashing

Safety - what happens if this component fails?
- ex. The display on your phone failing shouldn't mean that it causes the CPU to burst into flames

Maintainability - how easy is it to fix?
- ex. If the SSD in your computer dies, you can just swap it with a new one

But what exactly is the difference between an error, a fault, and a failure?

A fault is the source of an error. An error is a state in the overall system that might lead to a failure. A failure is when a system cannot do what it is supposed to do.

So as an examde, a person using a phone walks underground is a fault, as it makes the reception bad. This may lead to a corrupted/lost packet, which is an error. Enough dropped packets means that our person can't view his friend's Snapchat, no matter how funny it might be, which is a failure.

So fault tolerance, the topic of this lecture series, is about what to do when we encounter sources of error

Faults themselves are split into three distinct categories, based on how much they affect the system.

Transient faults are one-time things that go away once the event has completed, like birds flying in front of a microwave receiver.

Intermittent faults occur, then disappear, then occur again later on. One such example is a race condition - where sometimes it'll do the right thing and other times it won't, and it'll keep happening until the root of the problem is resolved.

Permanent faults continue to exist until the faulty component is replaced - like a dead heard drive.

## Failure Variants

Failures are also split into five categories.

Crash failure – the server stops, but right before it stops it works fine
- "the server crashed"

Omission failure – the server stops responding to messages
- receive omission
  - server stops receiving messages
- send omission
  - server stops sending responses

Timing failure – server responds to requests but out of the time interval it's supposed to (generally too late)

Response failure – the server responds to messages with incorrect results
- value failure
  - exact response is the wrong value or type
- state transition failure
  - server has followed the incorrect control flow path

Byzantine failure – server produce arbitrary responses at arbitrary times

The last thing is outside the scope of these notes.

## Masking Failures Using Redundancy

The first technique we'll look at is including additional redundant components. By consulting all copies of the component and using a voting system, failures of other components can be hidden.

Triple modular redundancy is one such example of hardware redundancy. Say we have a simple pathway that looks like this:

$$\begin{tikzcd}
\text{inputs} \arrow[r] & \textcircled{+} \arrow[r] & \textcircled{\times} \arrow[r] & \text{outputs}
\end{tikzcd}$$

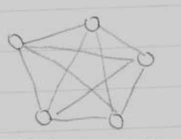
This is a simple adder and multiplier in series. Obviously if any of these components fail, the output will be incorrect or non-existent.

$$\begin{tikzcd}
 & \boxed{+1} \arrow[dr] & & \boxed{\times 1} \arrow[dr] & & \\
\text{inputs} \arrow[ur] \arrow[r] \arrow[dr] & \boxed{+2} \arrow[r] & \boxed{\text{Majority}} \arrow[ur] \arrow[r] \arrow[dr] & \boxed{\times 2} \arrow[r] & \boxed{\text{Majority}} \arrow[r] & \text{out} \\
 & \boxed{+3} \arrow[ur] & & \boxed{\times 3} \arrow[ur] & &
\end{tikzcd}$$

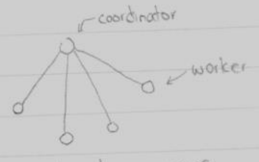
This configuration can handle the failure of one component per step by passing outputs of each component to a majority gate. However, this particular scheme more than triples the hardware required

This sort of redundancy can be applied in a similar way to software.

We can arrange identical processes in one of two formats:



Flat grouping



Hierarchical grouping

So the question is, how do these groups figure out the supposedly correct value? This is known as the "consensus problem".

This is how the consensus problem works:

1) Each process has a propose(val) and decide() method
2) Each process proposes a value by calling propose(val). It's assumed that val is a deterministic result of the state of the process.
3) Each process then learns the decided value by calling decide().

To guarantee some portions of the problem, the following is assumed:

1) Two calls by working processes (correct ones) should never produce two different values
2) If a correct process calls decide() and gets the value v back, one process must have called propose(v)
3) propose(val) and decide() terminate eventually as long as the process is correct.

In the flat grouping, each process will propose() a value to every other process. Decide() would then be made based on all values proposed at each individual process.

In the hierarchical formation, worker processes propose values to the master process, at which point the master's decide() becomes the single source of truth.

These are not solutions, however - for a given system, there is only a possibility of solving the consensus problem. The possibility is dependent on a few factors of the system's distribution.

Synchronous vs. Async Processes - how long does it take to go to the next step? Can one process be at step 3 while another is at step 6? Is the bound on this delay known?

Communication delay - is there a bound on the time it takes to deliver a message? Is the bound known?

Message delivery order - does the order of messages matter?

Unicast vs. multicast messaging - can a process send a message to more than one process at a time?

| | Unordered | | Ordered | | |
|---|---|---|---|---|---|
| | Unicast | Multicast | Unicast | Multicast | |
| Synchronous | X | X | X | X | Bounded comm. delay |
| | | | X | X | Unbounded |
| Async | | | | X | Bounded |
| | | | | X | Unbounded |

Above is an informal summary of possibility (X) of solving the consensus problem.

## RPC Failures and Failure Tolerance

Since RPCs are the prime way for distributed systems to communicate, it's important to understand their potential failures.

1) Client cannot locate the server
2) Client request is lost
3) Server crashes after receiving request
4) Server reply is lost
5) Client crashes after sending a request

We'll look at number 3 first. In this case, we don't know if the server crashed during execution and before sending, or after execution and while sending.

We can sanely approach this a few ways.

1) Reissue the request. Execution must be idempotent as we might process the request several times.

2) Give up, and report a failure.

3) Figure out whether the server stub was executed, and only re-issue request if it failed.

1 and 2 are examples of "at-least-once" semantics while 3 is an "exactly once". 3 is nice in that there's no repetition, but it's quite a difficult scheme to implement. The other two can be done using acknowledgements.

Let's say that we have a non-idempotent server stub that increments an integer by 1. We want to acknowledge the request, but we can do it before the execution (for speed) or after execution (for more correctness).

As such, we have three events in this scenario:

ACK : acknowledge
E : execute server stub
C : crash

Which lead to the following possibilities:

| | |
|---|---|
| ACK → E → C : | crash after acknowledgement and execution |
| ACK → C : | crash after acknowledgement but before increment |
| E → ACK → C : | crash after integer is incremented and acknowledgement sent |
| E → C : | integer incremented but crash before acknowledge message was sent |
| C : | crash before server could do anything |

Now we can see the result of using various reissue strategies in addition to the difference of acknowledgement timings. (with one re-issue).

| | ACK → E | | |
|---|---|---|---|
| Reissue On | ACK → E → C | ACK → C | C |
| Always | 2 | 1 | 1 |
| Only on ACK | 2 | 1 | 0 |
| Only w/o ACK | 1 | 0 | 1 |
| Never | 1 | 0 | 0 |

Let's walk through the first entry to see how it works.

1) Client sends message
2) Server sends ACK
3) Server increments (i=1)
4) Client re-issues request
5) Server crashes (and restarts)
6) Server receives message and ACKs
7) Server increments (i=2)

Step 4 and beyond don't happen in any particular order.
Let's look at what happens if we go E → ACK

| Reissue On | E → ACK → C | E → C | C |
|---|---|---|---|
| Always | 2 | 2 | 1 |
| Only on ACK | 2 | 1 | 0 |
| Only w/o ACK | 1 | 2 | 1 |
| Never | 1 | 1 | 0 |

These tables go to show that idempotency is a pretty crucial element to have if we want to re-issue requests.