Distributed systems – Theory
10. Fault tolerance

Faults

- A system is said to fail when it does not meet its specification.
- Gravity:
  - a supermarket's distributed ordering system, a failure may result in some store running out of canned beans.
  - in a distributed air traffic control system, a failure may be catastrophic.
- Types:
  - Component faults
  - Distr. system failures

Component faults

- can fail due to a fault in some component, such as a processor, memory, device, cable, or software.
- A *fault* is a malfunction, possibly caused by a
  - design error,
  - a manufacturing error,
  - a programming error,
  - physical damage,
  - deterioration in the course of time,
  - harsh environmental conditions (it snowed on the computer),
  - unexpected inputs,
  - operator error,
  - rodents eating part of it,
  - and many other causes.
- Not all faults lead (immediately) to system failures, but some do.

Component faults - classification

- *Transient faults* occur once and then disappear.
  - If the operation is repeated, the fault goes away.
  - A bird flying through the beam of a microwave transmitter may cause lost bits on some network
  - If the transmission times out and is retried, it will probably work the second time.
- An *intermittent fault* occurs, then vanishes of its own accord, then reappears, and so on.
  - A loose contact on a connector will often cause an intermittent fault.
  - Intermittent faults cause a great deal of aggravation because they are difficult to diagnose.
  - Typically, whenever the fault doctor shows up, the system works perfectly.
- A *permanent fault* is one that continues to exist until the faulty component is repaired.
  - Burnt-out chips, software bugs, and disk head crashes often cause permanent faults.

Goal of designing and building fault-tolerant systs

- Ensure that the system as a whole continues to function correctly, even in the presence of faults.
- Traditional work in the area: statistical analysis of electronic component faults.
- Very briefly:
  - if some component has a probability *p* of malfunctioning in a given second of time, the mean time to failure = l/*p*
  - For example, if the probably of a crash is 10^{-6} per second, the mean time to failure is 10^{6} sec or about 1 1.6 days.

System failures

- Critical DS: syst must survive component (in particular, processor!) faults, rather than just making these unlikely.
- Processor faults or crashes should be understood to mean equally well process faults or crashes due to software bugs.
- Combinations of processor faults with communication line faults can be considered,
  - since standard protocols can recover from line errors in predictable ways,

    we examine only processor faults!

Types of processor faults

- *Fail-silent faults:*
  - a faulty processor just stops and does not respond to subsequent input or produce further output, except perhaps to announce that it is no longer functioning.
  - also called *fail-stop* faults.
- *Byzantine faults:*
  - faulty processor continues to run, issuing wrong answers to questions, & possibly working together maliciously with other faulty processors giving impression that to work correctly when they are not.
  - undetected software bugs often exhibit Byzantine faults.
  - dealing with Byzantine faults is going to be much more difficult than dealing with fail-silent ones.
  - the term "Byzantine" refers to the Byzantine Empire time (330-1453) and place (the Balkans) in which endless conspiracies, intrigue, and untruthfulness were alleged to be common in ruling circles.

"Synchronous" vs. "Asynchronous" Systs

- Suppose:
  - A system in which if one processor sends a message to another, it is guaranteed to get a reply within a time *T* known in advance.
  - Failure to get a reply means that the receiving system has crashed.
  - The time *T* includes sufficient time to deal with lost messages (by sending them up to n times).
- A system …
  - … that has the property of always responding to a message within a known finite bound if it is working is said to be *synchronous*.
  - … not having this property is said to be *asynchronous*.
- This terminology is unfortunately since it conflicts with traditional uses of the terms; it is widely used among workers in fault tolerance.
- Asynch. systs are going to be harder to deal with than synch. ones.
  - If a processor can send a message and know that the absence of a reply within T sec means that the intended recipient has failed -> can take corrective action.
  - No upper limit to how long the response might take -> determining whether there has been a failure is going to be a problem.

Use of Redundancy as general approach

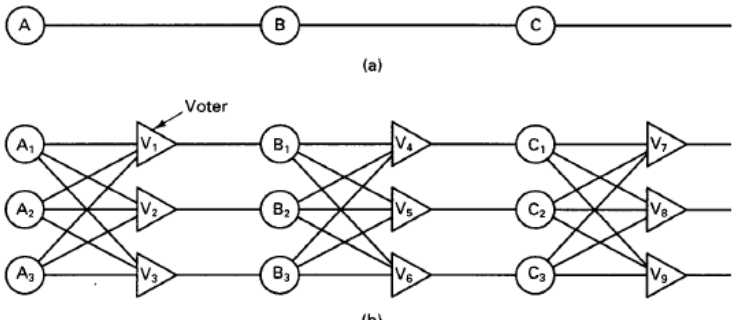
1. information redundancy
   - extra bits are added to allow recovery from garbled bits.
   - E.g. a Hamming code can be added to transmitted data to recover from noise on the transmission line.
2. time redundancy,
   - an action is performed, and then, if need be, it is performed again.
   - example: using the atomic transactions – if a transaction aborts, it can be redone with no harm.
   - especially helpful when the faults are transient or intermittent.
3. physical redundancy

Physical redundancy

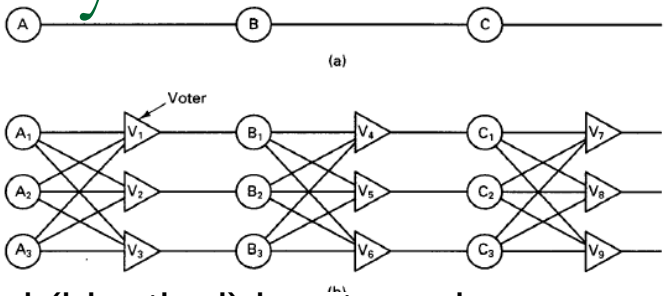
- extra equipment is added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components.
- example: extra processors can be added to the system so that if a few of them crash, the system can still function correctly.
- two ways to organize these extra processors:
  - active replication and
  - primary backup.
  - example: case of a server.
    - When active replication is used, all the processors are used all the time as servers (in parallel) in order to hide faults completely.
    - The primary backup scheme just uses one processor as a server, replacing it with a backup if it fails.

Fault Tolerance Using Active Replication

- used in
  - biology (mammals have two eyes, two ears, two lungs, etc.),
  - aircraft (747s have four engines but can fly on three), and
  - sports (multiple referees in case one misses an event).
- Some authors refer to active replication as the state machine approach.
- used for fault tolerance in electronic circuits for years.
  - the circuit in (a):
    - signals pass through devices A, B and C, in sequence.
    - If one of them is faulty, the final result will probably be wrong.
  - the circuit in (b),
    - each device is replicated three times.
    - following each stage in the circuit is a triplicated voter.
    - each voter is a circuit that has three inputs and one output.
    - If two or three of the inputs are the same, the output is equal to that input.
    - If all three inputs are different, the output is undefined.
    - this kind of design is known as TMR (Triple Modular Redundancy).



Triple modular redundancy



- Suppose element A2 fails:
  - Each of the voters, V1, V2 and V3 gets two good (identical) inputs and one rogue input, and each of them outputs the correct value to the second stage.
  - In essence, the effect of A2 failing is completed masked, so that the inputs to B1, B2 and B3 are exactly the same as they would have been had no fault occurred.
- Now consider what happens if B3 and C1, are also faulty, in addition to A2.
  - These effects are also masked, so the three final outputs are still correct.
- At first it may not be obvious why three voters are needed at each stage.
  - After all, one voter could also detect and pass though the majority view.
    - However, a voter is also a component and can also be faulty.
  - Suppose, for example, that V1 malfunctions.
    - The input to B1 will be wrong, but B2 & B3 will produce the same output and V4, V5 & V6 will all produce the correct result into stage three.
  - A fault in V1 is effectively no different than a fault in B1 produces incorrect output, but in both cases it is voted down later.
- TMR can be applied recursively,
  - for example, to make a chip highly reliable by using TMR inside it,

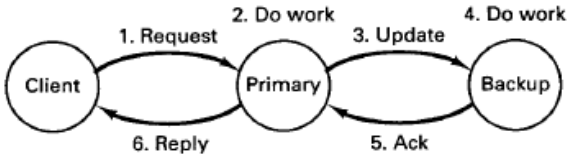
How much replication is needed?

- The answer depends on the amount of fault tolerance desired
- A system is said to be *k* fault tolerant if it can survive faults in *k* components and still meet its specifications.
- If the components, say processors, fail silently,
  - then having *k* + 1 of them is enough to provide *k* fault tolerance.
  - If *k* of them simply stop, then the answer from the other one can be used.
- If the processors exhibit Byzantine failures, continuing to run when sick and sending out erroneous or random replies,
  - a minimum of 2*k* + 1 processors are needed to achieve *k* fault tolerance.
    - In the worst case, the *k* failing processors could accidentally (or even intentionally) generate the same reply.
    - However, the remaining *k* + 1 will also produce the same answer, so the client or voter can just believe the majority.

Fault Tolerance Using Primary Backup

- The essential idea of the primary-backup method is that at any one instant, one server is the primary and does all the work.
- If the primary fails, the backup takes over.
- Ideally, the should take place in a clean way and be noticed only by the client operating system, not by the application programs.
- This scheme is widely used in the world.
  - Examples: are government (the Vice President), aviation (co-pilots), automobiles (spare tires), and diesel-powered electrical generators in hospital operating rooms.
- Primary-backup fault tolerance has two major advantages over active replication.
  - it is simpler during normal operation since messages go to just one server (the primary) and not to a whole group.
  - in practice it requires fewer machines, because at any instant one primary and one backup is needed
- Downside
  - works poorly in the presence of Byzantine failures in which the primary erroneously claims to be working perfectly.
  - recovery from a primary failure can be complex and time consuming.

A simple primary-backup protocol on a write operation



- The client sends a message to the primary, which does the work and then sends an update message to the backup.
- When the backup gets the message, it does the work and then sends an acknowledgement back to the primary.
- When the acknowledgement arrives, the primary sends the reply to the client.
- Effect of a primary crash at various moments during an RPC?
  - If the primary crashes before doing the work (step 2), no harm is done.
    - The client will time out and retry.
    - If it tries often enough, it will eventually get the backup & work will be done exactly once.
  - If the primary crashes after doing the work but before sending the update, when the backup takes over and the request comes in again,
    - the work will be done a second time – if the work has side effects, this could be a problem.
  - If the primary crashes after step 4 but before step 6,
    - the work may end up being done three times, once by the primary, once by the backup as a result of step 3, and once after the backup becomes the primary.
  - If requests carry identifiers -> possible to ensure that the work is done only twice
    - getting it done exactly once is difficult to impossible.

When to cut over from the primary to the backup?

- the backup could send: "Are you alive?" messages periodically to the primary.
- If the primary fails to respond within a certain time, the backup would take over.
- If the primary has not crashed, but is merely slow ,
  - In an asynch. syst. There is no way to distinguish between a slow primary and one that has gone down.
  - The best solution is a hardware mechanism in which the backup can forcibly stop or reboot the primary.
  - All primary-backup schemes require agreement, which is tricky to achieve, whereas active replication does not always require an agreement protocol (e.g. TMR).
  - Another solution is to use a dual-ported disk shared between the primary and secondary.
    - when the primary gets a request, it writes the request to disk before doing any work and also writes the results to disk.

Agreement in Faulty Systems

- In many DSs there is a need to have processes agree on something.
  - Examples are: electing a coordinator, deciding whether to commit a transaction or not, dividing up tasks among workers, synchronization, and so on.
- Goal of distributed agreement algs: to have all the faulty processors reach consensus on some issue, and do that within a finite number of steps.
- Different cases are possible depending on system parameters, including:

1. Are messages delivered reliably all the time?
2. Can processes crash, and if so, fail-silent or Byzantine?
3. Is the system synchronous or asynchronous?

The easy case

- Perfect processors + Communication lines that can lose messages
- A famous problem, known as the ***two-army problem***,
  - illustrates difficulty of getting even two perfect processors to reach agreement about 1 bit of information

  - The red army, with 5000 troops, is encamped in a valley.
  - Two blue armies, each 3000 strong, are encamped on the surrounding hillsides overlooking the valley.
  - If the two blue armies can coordinate their attacks on the red army, they will be victorious.
  - However, if either one attacks by itself it will be slaughtered.
  - The goal of the blue armies is to reach agreement about attacking.
  - The catch is that they can only communicate using an unreliable channel: sending a messenger who is subject to capture by the red army.

Example for the two army problem

- Commander of blue army 1, Gen. Alexander, sends a message to the commander of blue army 2, Gen.Bonaparte: "I have a plan-let's attack at dawn tomorrow."
- The messenger gets through and Bonaparte sends him back with a note saying: "Splendid idea, Alex. See you at dawn tomorrow."
- The messenger gets back to his base safely, delivers his messages, and Alexander tells his troops to prepare for battle at dawn.
- However, later that day, Alexander realizes that Bonaparte does not know if the messenger got back safely and not knowing this, may not dare to attack.
- Consequently, Alexander tells the messenger to go tell Bonaparte that his (Bonaparte's) message arrived and that the battle is set.
- Once again the messenger gets through and delivers the acknowledgement.
- But now Bonaparte worries that Alexander does not know if the acknowledgement got through. He reasons that if Bonaparte thinks that the messenger was captured, he will not be sure about his (Alexander's) plans, and may not risk the attack, so he sends the messenger back again.

!!! *Even if the messenger makes it through every time, it is easy to show that Alexander and Bonaparte will never reach agreement, no matter how many acknowledgements they send* !!!

Analysis of two army example

- Assume that there is some protocol that terminates in a finite no. steps.
- Remove any extra steps at the end to get the minimum protocol that works.
- Some message is now the last one and it is essential to the agreement (because this is the minimum protocol).
- If this message fails to arrive, the war is off.
- The sender of the last message does not know if the last message arrived.
- If it did not, the protocol did not complete and the other general will not attack.
- Thus the sender of the last message cannot know if the war is scheduled or not, and hence cannot safely commit his troops.
- Since the receiver of the last message knows the sender cannot be sure, he will not risk certain death either, and there is no agreement.

!!!! *Even with nonfaulty processors (generals), agreement between even two processes is not possible in the face of unreliable communication.* !!!

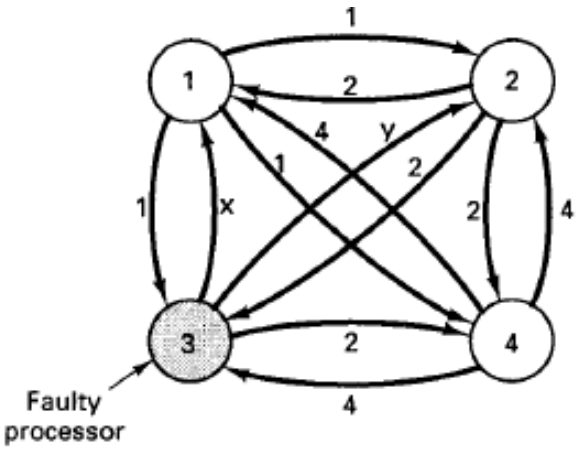
Second case

- Communication is perfect, but the processors are not.
- The classical problem here also occurs in a military setting and is called the ***Byzantine generals problem***.
  - the red army is still encamped in the valley, but *n* blue generals all head armies on the nearby hills.
  - communication is done by telephone and is perfect,
  - but *m* of the generals are traitors (faulty) and are actively trying to prevent the loyal generals from reaching agreement by feeding them incorrect and contradictory information (model malfunctioning procs.)
  - Question: the loyal generals can still reach agreement?
  - Each general is assumed to know how many troops he has.
  - Goal of the problem: for the generals to exchange troop strengths, so that at the end of the algorithm, each general has a vector of length *n* corresponding to all the armies.
  - If general *i* is loyal, then element *i* is his troop strength; otherwise, it is undefined.

Lamport's recursive algorithm (1982)

- Example for $n = 4$ and $m = 1$ => 4 steps

1. Every general sends a (reliable) mes. to all other gen. announcing his truth strength.
   - loyal generals tell the truth; traitors may tell every other general a different lie.
   - (a) we see that general 1 reports 1K troops, general 2 reports 2K troops, general 3 lies to everyone, giving *x, y,* and *z*, respectively, and general 4 reports 4K troops.
2. Results of the announcements of step 1 are collected together in vectors (b)
3. Step 3 consists of every general passing his vector from (b) to every other general.
   - general 3 lies through his teeth, inventing 12 new values,
   - the results of step 3 are shown in (c).
4. Step 4: each general examines the *i*th element of each of the newly received vectors.
   - If any value has a majority, that value is put into the result vector.
   - If no value has a majority, the corresponding element of the vector is marked UNKNOWN.
   - (c): generals 1, 2, and 4 all come to agreement on (1, 2, UNKNOWN, 4), the correct result.
   - The traitor was not able to gum up the works.

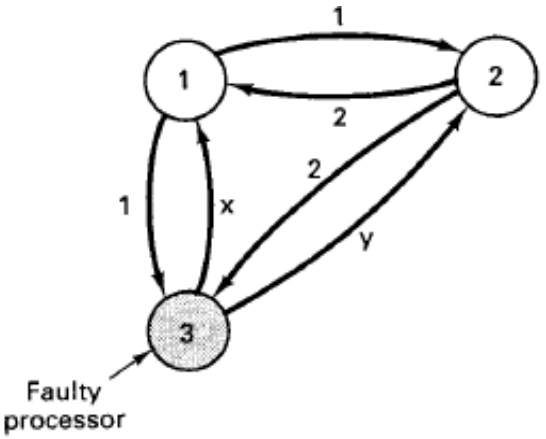


	1 Got	2 Got	3 Got
1 Got (1, 2, x, 4)			
2 Got (1, 2, y, 4)	(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
3 Got (1, 2, 3, 4)	(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
4 Got (1, 2, z, 4)	(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(b) (c)

Another example

- for $n = 3$ and $m = 1$, that is, only 2 loyal generals and 1 traitor,
- In (c): neither of the loyal generals sees a majority for element 1, element 2, or element 3, so all of them are marked UNKNOWN.
- The algorithm has failed to produce agreement.



(a)

1 Got (1, 2, x)
2 Got (1, 2, y)
3 Got (1, 2, 3)

(b)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

General case

- Lamport proved that in a system with *m* faulty processors, agreement can be achieved only if 2*m* + 1 correctly functioning processors are present, for a total of 3*m* + 1.
- *Agreement is possible only if more than two-thirds of the processors are working properly.*
- Worse: Fischer proved that in a DS

  with asynchronous processors + unbounded transmission delays,

  no agreement is possible if even one processor is faulty

  (even if that one processor fails silently).
  - The problem with asynchronous systems is that arbitrarily slow processors are indistinguishable from dead ones.
- Many other theoretical results are known about when agreement is possible and when it is not.