

Tutorial 6
Notes on Solutions

Fault tolerance

1. A firewall is a mission-critical aspect of many organisations' computer infrastructure. It is proposed that a replicated firewall built of 3 computers be designed, to provide a fault-tolerant solution.
 - a. Discuss the extent to which each of the following apply to this example:
 - i. Triple modular redundancy.

TMR would be overkill. It would also require extra hardware for voting. Existing network protocols can detect lost packets or garbled data, so a voting protocol adds more than is really needed.

- ii. Single thread using non-blocking system calls.

This is not relevant to the question.

- iii. Availability, reliability, safety and maintainability.

Each of these is an issue to consider in relation to the actual requirements of the organization:

- *availability can be enhanced by simple replication; if the network on the internal side of the firewall could handle the extra traffic, duplicate packets, while a performance issue, would not result in errors in transactions. A more sophisticated solution would be to split traffic between the replicas, and have them regularly ping each other. If silent failures are the most likely situation (true in many cases), the worst-case scenario would be that some packets may have to be retransmitted.*
- *ironically, replication doesn't make a system more reliable: it is likely to break more often because there are more components (hence more chance of random errors, hardware failures, etc.). However this is not relevant to this situation because high availability is more important*
- *safety may or may not be an issue but in this scenario, it is more likely to be handled somewhere in the network protocols or application design*
- *maintainability is potentially enhanced with this design, because 1 or even 2 of the replicas can be taken out without stopping the system. In the scenario of load being split between the replicas, you could either warn the others when one was being taken down so they could take over its load, or simply rely on the fault detection mechanism to take over its share of the load.*

iv. Byzantine failures.

Given the complexity of filtering packets, it is more likely that the software would fail completely rather than introduce random errors but still transmit packets. Random errors in the data would be picked up by the network protocols, so it would be overkill to attempt to handle such errors.

In all cases, consider the likely nature of failures in this specific application, and include consideration of other levels of fault tolerance, e.g., in the network.

- b. Discuss advantages and disadvantages of a distributed solution over buying a purpose-design fault-tolerant computer to implement a highly-dependable firewall.
2. Explain why Byzantine failures require more replicated resources to deal with than silent failures.

Because erroneous results have to be voted down, you need a majority (50%+1) to detect Byzantine failures. A silent failure produces no result and hence is not in contention for being part of the solution: provided a timeout or other suitable technique can detect the failure, only one correct alternative source of solution is needed.

3. Work through the Byzantine generals problem as in the lecture slides, but add a second traitorous general (anyone will do, but let's pick on general 2).

- a. What happens?

In the final step, you should see that generals 1 and 3 (the non-traitorous ones) end up with no results on which a majority agree -- not too surprising, given that the ``good'' ones must outnumber the traitors.

- b. What would happen if the traitorous generals were more consistent in their fibbing?

It would theoretically be possible for them to gang up and outvote the ``good'' ones, but a clear majority would have to do this.

- c. Does this problem correspond with any real situation?

In a distributed system, the ``traitors'' represent faulty processes which produce incorrect results. A clear majority of processes would have to consistently produce the same incorrect result. If the fault was a design fault which some processes had in common (e.g.: two different releases of the software, one erroneous, the other correct, and the erroneous release consistently produced the same error).

4. Explain why network-related problems make it hard to implement transparent RPC.

An ordinary procedure (or method) call doesn't have problems like timeouts, or losing contact with either party (caller or callee. Consequently, it's hard to make RPC behave exactly like a local call -- at least the error handling has to be different.

5. Idempotence is a desirable property of network transactions in a distributed system.
 - a. Explain why requiring idempotence, in general, is not useful.

Many real-world transactions change state, e.g., update a bank balance, so they inherently are defined as unreasonable to do more than once.

- b. Explain workarounds of the problem of dealing with nonidempotent transactions.

There are various options, such as using sequence numbers so duplicates can be detected. However, these mechanisms require keeping state at the server (operation results) side long enough to send the reply without repeating transactions.

- c. Is handing in an assignment through an electronic submission system idempotent? Explain.

It depends whether your latest handin is a duplicate or a new version.

6. Explain why orphans are a difficult problem in the face of client crashes.

Orphans on their own are a problem in that it is not necessarily clear that a given process should die after its parent crashes. We are left with messy problems to fix up either way, e.g., an orphan which completes just before the crash, but before its results are recorded in the client. Grandorphans further complicate the problem.

7. Multicasting in general has problems scaling.
 - a. Are the problems harder for distributed systems than for general networking? Explain.

In distributed computing, we would like a model of delivery to the application layer, where we are sure the data is consumed, which is a tougher standard of delivery than in a purely networked world. We would also ideally like the application layer to be ignorant of the network underneath, which further complicates matters, since the application layer needs to in effect acknowledge receipt. All of these things add to the usual problems of scaling multicast (acknowledgement implosion, changing group membership, etc.)

- b. A group of distributed processes needs to stop every now and then to exchange information. The following mechanism is proposed:
 - processes involved in a cooperative computation are members of a multicast group, and periodically multicast state information to the rest of the group
 - when one process determines that the group has reached a checkpoint at which they should exchange more information, it sends a message to

a process not already in the group, requesting that it join the multicast group

- this new process joins the group, and waits for a message from all the other processes, indicating that they have sent their information to the other processes
- when it has seen information from all the other processes, it leaves the group

i. Discuss the role of *view changes* in this solution.

A view change can be used to implement a barrier. A barrier is a synchronization primitive which forces processes to wait until all other processes have reached the same point in the computation. In effect, the dummy process which enters and leaves the group has the same effect as a barrier.

ii. How robust is this solution? Consider ways it could go wrong, and possible remedies.

The mechanism is highly dependent on the "barrier" process (single point of failure). It also depends on correct implementation of virtual synchrony, which is a hard problem.

8. Can you think of any situation (ordinary networking or computing, or distributed computing) where you have encountered *forward recovery*? Explain your example.

One example is the use of error correction in RAID disks -- when an error is detected, it is corrected, rather than requiring a retry. Any other examples which illustrate the basic idea: rather than backtrack to a previous, correct state, try to move ahead to a new state by filling in missing information.