Antifragile Software Development – Part 2: Efficiency and Redundancy

9 May 2014 | **Business Innovation, Software Engineering** | **Roman Bertolami**
Reading time: 5 minutes

In his seminal book "Antifragile – Things that gain from disorder" N.N. Taleb makes some bold statements derived from a risk based view of the world. In this series we want to look at some of his claims from a software development perspective.

While I discussed the Problem with Size in my first post, the second part now covers efficiency, optimisation and redundancy.

The trade-off between redundancy and efficiency is one of the key aspects of the book. According to Mr. Taleb:

> "A central problem of the world today is the misunderstanding of nonlinear response by those involved in creating 'efficiencies' and 'optimization' of systems."

He repeatedly argues to introduce more redundancy to reduce fragility:

> "Layers of redundancy are the central risk management property of natural systems."

> "Redundancy is ambiguous because it seems like a waste if nothing unusual happens. Except that something unusual happens — usually."

The trade-off between redundancy and efficiency is present in software engineering, too. Especially in two fields. The first field is process and people and the second field is the design and architecture of the systems we build.

Process and People

Bus factor
It might be highly efficient when every developer in a team is developing its own part of the system. She will know her part extremely well and will be able to add new features very quickly. However, with such a setup, the bus factor is very small, just one. Consequently, this means that if she quits for whatever reason, the project will be slowed down dramatically or even stopped. There is no one else who knows her part of the system.

Adding redundancy and thus increasing the bus factor can be done in several ways, e.g. by implementing a collective code ownership, code reviews or pair programming. Each of these

strategies has some cost but considering the mitigation of risks it can really pay off.

Horizontal Teams vs Vertical Teams
A similar issue arises when the project is large enough to be developed by multiple teams. How these teams are split across the system architecture is crucial. Having a vertical splitting, e.g. a database team, a backend team and a frontend team might be very efficient at first sight. However, these teams are highly dependent on each other and thus fragile. It is typically better to remove these dependencies by building cross functional feature teams that are able to develop features across all layers of the architecture.

System Architecture

On the architecture level there are several techniques to build redundancy into our systems to reduce the risk of outages. All these techniques come at the cost of efficiency, either performance, compute units, or storage space. Sometimes we are forced to optimise our system, e.g. by implementing caching, which on the other hand increases the risk of failures.

Horizontal scaling, stateless systems
A crucial point to add redundancy is to keep shared state as small as possible and not to store it in the service layer. Thus, we build stateless web servers that can be easily scaled horizontally. If one of the server crashes, another server can continue to handle the requests of the client.

Data Redundancy
However, hardly any system is completely stateless. Typically, there is a database that is responsible for keeping the current state of the system. To avoid the single point of failure we build in data redundancy by storing the data among multiple database servers (e.g. based on a master-slave model). This typically comes at a cost of write performance. On a lower level we implement data redundancy by using RAIDs. On purpose we buy more disks than necessary to be able to recover data when a disk crashes.

Caching
Caching is a typical optimisation done in software systems to increase performance by distributing state. However, caching can lead to serious bugs when the underlying data changes. Those kinds of bugs are typically very hard to find. One way to mitigate the risk is to be able to turn off all caching in a system and assuring that the bug does not occur anymore when no caching is done.

Avoid vendor lock-ins

> “The notion of efficiency becomes quite meaningless on its own. If a gambler has a risk of terminal blowup (losing back everything), the ‘potential returns’ of his strategy are totally inconsequential.”

A software system dependent of a specific vendor always bears the risk of “terminal blowup”. For example if the vendor decides to change legal conditions, discontinues the maintenance of the software or hardware we are dependent on, or if the vendor itself is blown up.

A way to mitigate the risk is to use clean boundaries as stated in “Clean Code” by Robert C. Martin: “We should avoid letting too much of your code know about the third party particulars. It is better to depend on something *you* control than on something you don’t control, lest it end up controlling you.”

Antifragile Software Development – Part 1: The Problem with Size

Antifragile Software Development – Part 3: Interventionism and Iatrogenics