Tales From Inside the Crater

Jesper Louis Andersen
jesper.louis.andersen@gmail.com
ShopGun

October 2, 2017

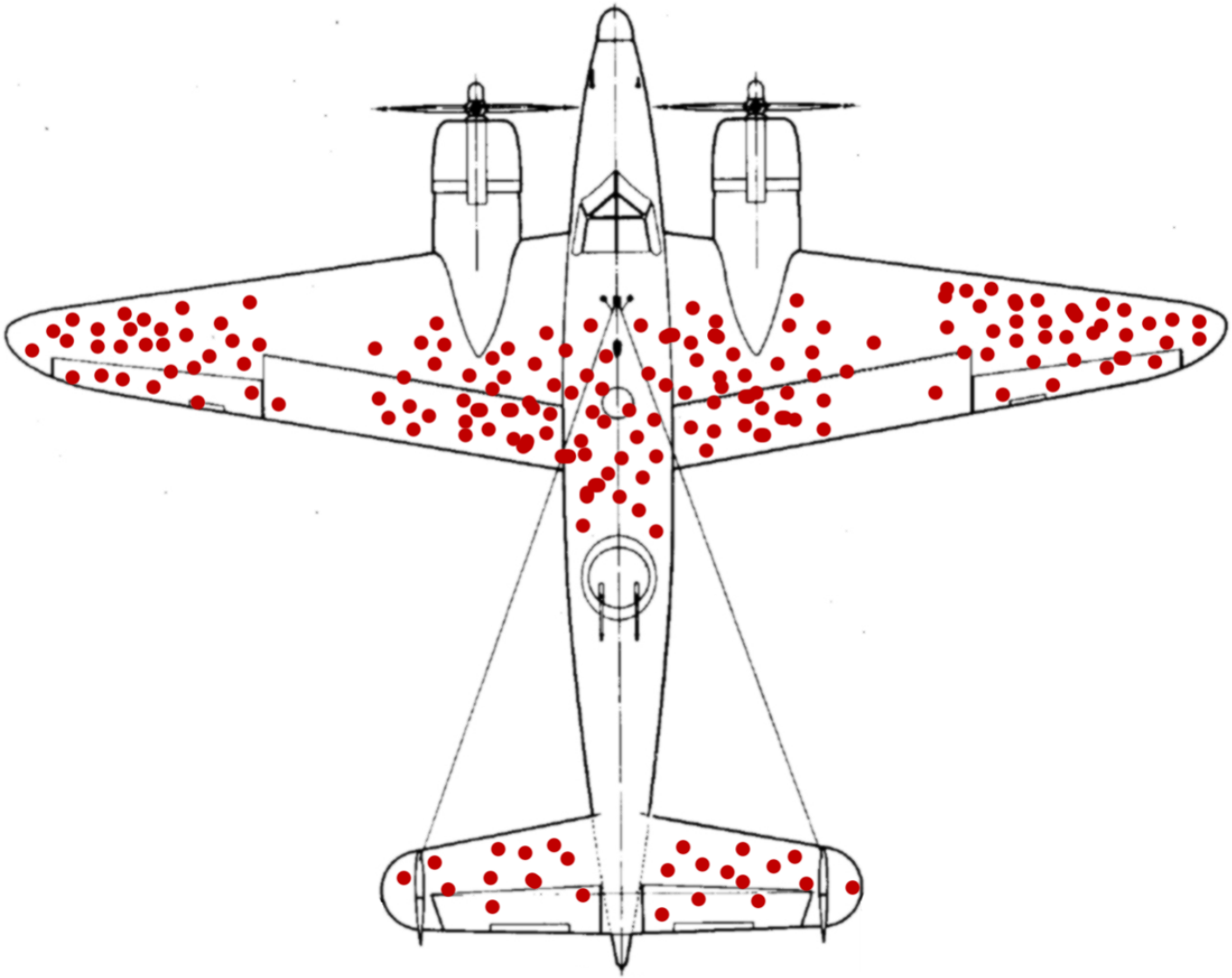
Motivation

Setup

- Question is “Does strategy X work in software”
- Typical examples for X: “Microservices”, “Agile”, “Serverless”, “Static typing”, “Write *everything* in Haskell”
- Want *random* samples in our data set, selected uniformly among projects

Statistical dangers I

- Reality: Survivor bias



Statistical dangers II

- Selection bias: The selection process of the projects to study are not random, nor fair.
- Reporting bias: Projects have details which go under-reported or ignored because it doesn't *sell* the cause, or because people didn't think the data important

Main Hypothesis

- Most strategies in software have *small* effect size
- Most strategies are “phase shifts:” they trade off certain advantages for other advantages

100% success rate implies data tampering

System design

- Everything in this talk are things I've experienced
- Fools your intuition
- …or was told by people I trust
- A certain amount of osmosis is expected

Microservices

Definition

One possible definition:

- The system is split into modular *processes*
- Processes are *isolated* (software, VM, hardware)
- Processes communicate by *message passing*
- Communication is *not* reliable

Goal: emergent behavior among processes!

Erlang I

One possible definition:

- The system is split into modular *processes*
- Processes are *isolated* (software, VM, hardware)
- Processes communicate by *message passing*
- Communication is *not* reliable

Goal: emergent behavior among processes!

Erlang II

- Erlang systems have a 30 year head start on microservices
- Most of the ideas from the Reactive manifesto also overlaps
- Microservices are independently invented at many companies around 2013
- “Microservices” term from 2014 (Google Trends)

Does your microservice use 1/1.000.000 of your machine capacity?

Distributed

- A system based on microservices is a *distributed* system
- Distributed systems trade complexity for either greater resilience or greater efficiency (redundancy / sharding)
- There are many more failure modes in a distributed system
- Of 1000 nodes, at least 5 are down at any point in time
- Consistency (Serializability / Linearizability, ACID) is often extremely hard to achieve
- Most dist-sys get consistency wrong (see e.g., Kyle Kingsbury: Jepsen)
- Maintenance is often way harder (immaturity, scale, …)

Epistemic Logic corresponds to distributed systems

- In propositional logic facts are *globally* true
- Epistemic logic has Agents
- Agents *knows* facts
- “I know that Dan knows if it will rain”
- Correspondance: services with state (facts)

Sharing facts is harder in epistemic logic

Key observation:
~~Distributed systems~~
Epistemic logic requires radically different methods

You earn $25 per hour, 24/7. Your monolith (vertical) scalability?

Economic wall

- Amazon AWS X1E system: $26 per hour (on demand)
- 4 Terabytes of memory
- 128 CPU cores
- 25 gigabit internet
- I.e., a vast majority of companies will scale on a single machine instance just fine
- Trend: this is getting easier over time. Machines are getting more capacity
- Run your payments on the monolith: easy consistency

Capacity $\propto$ Earnings

Protocols

- In a message passing system, the protocol reign supreme
- What happens *inside* the service is not interesting
- What its *protocol* looks like matter
- Replacing a service is modular, loosely coupled
- Replacing a protocol is not

Often hard to define!

Protocol design learned the hard way

- Make a *global* protocol for the company up front
- Local payload inside this protocol
- Haphazard introduction of microservices means microprotocols
- TTL in messages (delivery count, milliseconds) (avoids poison scenarios)
  - Deduct queue sojourn and forwarding time as well
  - Jobs which cannot finish in time are thrown out
- Keep 2 bits: (user-facing bit, important bit). Prioritization under overload (Google SRE handbook)
- Unique request-Ids on everything for tracing

Protocol design learned the hard way II

- JSON has no Type-Zoo and is a lowest common denominator
- JSON requires a parser to look at every byte (i.e., slow)
- HTTP forces your hand into REST very often
- HTTP/2 is required for out-of-order processing
- It is not clear REST is a good fit for microservices
- Communication is a large part of the program

RPC is dangerous

RPC

- RPC forces a certain pattern in software
- Synchronous behavior slows down work
- Message flow is limited (often no cycles!)
- Tight coupling (monolith-of-microservices)
- James E. White in RFC 707, 708 (1976!)
  - Asynchronous
  - Binary protocol
  - Bidirectional messaging
  - HTTP/2 only solves this halfway (see: PUSH)
  - Plan9's 9p protocol does it way better

RPC II

- Example: `gRPC`
  - No mention in early doc of: fault, error, exception, throw, raise, …
  - Both TCP, HTTP/2 and gRPC implements flow control, yet build on each other in a stack
  - Flow-control is likely to fight, making performance bad

RPC in monoliths are function calls

Idempotent Ratchets

- Requests are *obligations* which can be restated
- Requests are idempotent and can be retried any number of times
- If we succeed, the ratchet turns one step and persists the new state safely.
- We can always restart from a safe ratchet state.
- Observation: Most stable systems I've built includes an idempotent ratchet somewhere.

Conway's law
Organization $\simeq$ System

Serverless

A Misnomer

- The *true* serverless architectures are peer-to-peer systems
- The client is *also* the server in P2P
- Examples: BitTorrent, Distributed Hash Tables (some Blockchains)
- What people call *serverless* really means “other people’s servers”
- Alternative name: outsourcing

Other people's servers

- You don't control the reliability of the service
  - The product can also become obsolete
- You don't control the quality of the service
  - Companies will min-max this: lousiest service which still keeps you as a paying customer
- If you don't run metrics on *their* service, you don't know their service level
- The client becomes fat
- Major reason for adoption: cost cutting, pay-as-you-go
- You cannot outsource:
  - Your core
  - Your reliability (your users point toward you anyway)

APIs & Hosted Functions

APIs

The new way to lock down customers:

- Do not implement an open protocol, force them to use your API
- Make the price for switches as expensive and time consuming as possible
- Even better, embed a large chunk of your own code in their system with an SDK
- Own their database, and force them to manage state between you and them
- Make their apps call to several different services, so they don't have a common factoring point

Hosted Functions

- Can produce *massive* cost cuts in organizations
- Reinvention of the PHP execution model: one apache process per incoming request
- Long-running jumps requires calling another hosted function in a recursive continuation pattern
- Stateless:
  - Easy to get correct
  - Inherently slow because every processing requires data loads
  - Solved with “caching” which means you keep the state in the cache
  - The cache is often outside the process, which implies latency
  - Better solved with lots of small stateful processes in most cases

Questions?

(Also happy to talk about: GraphQL, QuickCheck, Formal Methods, Type systems, Functional Programming, Concurrency models, …find me afterwards)

Overflow slides

Reactive Manifesto

Responsive

- Systems must cope with overload situations by dropping work
- Low latency by cheating: running *really really* fast
- Cooperative multithreading is dangerous
- Sensitive system: small fluctuations topples the system easily

Resilient

- Robustness: system survives unknown data
- Resilience: system gracefully degrades when failures happen
- Resilience is *far* harder than robustness (order of magnitude)
- Must cope with systems being unavailable
- Holistic behavior: cannot ignore client

Elastic

- Many systems won't need this!
- 1.5 kilobyte per customer, 1.000.000 customers
- $\Longrightarrow$ around 1.5 Gigabyte of memory
- AWS m3.medium instance: 3.5 gigabyte of memory
- Large OLTP databases fit in memory on modern machines (e.g., VoltDB)
- If it is in memory, miserable algorithms are fast

Message Driven

From experience:

- Common setup: processing stations with queues in between
- Means processing station temporarily impersonates message
- Internal vs External queues

Dualize!

- The *message* is a process (you'll get millions of them :)
- The *message* calls other subsystems
- Queuing happens in the system boundary
- Shaping happens in the system boundary

Serverless in Erlang

```
F = fun() -> some_expensive_closure end,
Initiator = self(),
Uniq = make_ref(),
Pid = spawn_link(fun() ->
             Res = F(),
             Initiator ! {result, Uniq, Res}
           end),
MRef = monitor(process, Pid),
...
receive
    {reply, Uniq, Res} ->
       demonitor(MRef, [flush]),
       ...;
    {'DOWN', MRef, process, _, Failure} ->
       ...
after Timeout ->
       ...
end
```