Scalability

What, when and how much

Classic Tale

- “Interest really took off, we got more and more customers and we couldn’t keep up. Eventually our customers lost faith, went to a competitor and the company folded.”

Not Just Websites

- Enterprise systems go through similar cycles
  - Huge initial investment
    - First version released
      - Promise of improvement
  - Ultimately canned

Over and over

Tradition

- Scalability: maintaining an acceptable level of performance as number of concurrent requests increases.

Truth

- Scalability: Managing implications of architectural choices as success increases.

Success

- More customers (aka load)
  - Classic scale
- More features
  - Does your codebase scale?

Success

- More third-party integrations
  - Do your support systems scale?
- More machines
  - Operational scale

Not doing enough

- Costs money, maybe an entire business

Doing too much

- Costs money, maybe an entire business

Do you feel lucky?

Lifecycle

Painful economics

Lifecycle

- “Build it and they will come”
- Until they do, do the minimum to get them (save money!)

Lifecycle

- Ride the tornado of success
  - Run like mad to keep up!

Trend

- Repeated cycles
  - Onset of problem
  - Root cause identification
  - Make changes
  - Deploy

What really matters...

Time to fix

- Critical for customer retention (continued sponsorship)
- Dependent upon:
  - How quickly we spot the problem
  - How quickly we understand the problem
  - How quickly we can solve the problem

Introducing the toolkit

Contents so far...

- Mechanisms for identifying onset of problems
- Mechanisms for identifying root cause

Solutions

Come in many forms

Solutions

- Two elements
  - Immediate: Stabilising the system
  - Follow-up: Proper fix

Solution Example

- Too many customers (classic scalability)
  - Immediate: Admission control
  - Follow-up: Fix the bottleneck
    - Machines, architecture, algorithm

Solution Example

- Can’t make code changes reliably
  - Immediate: Prioritise and limit changes
  - Follow-up: Partition the codebase, automate tests and build


STACK-ON

More tools...

- Range of firefighting options
- Range of solution options

Complications

Life is never simple

Complications

- Some problems can’t ultimately be solved
  - Scaling ACID
    - Elements of stock exchanges

Complications

- Some solutions can be disruptive
  - Application of CAP to a synchronous, transactional system
  - Significant impact on APIs
  - Compromises previous design assumptions

Complications

- System not amenable to change
  - Architectural fixes can be relatively straightforward but costly to make

Trying to avoid...


STACK-ON

More tools...

- Means for keeping systems “liquid”

Recap

Complete toolkit contents

Toolkit contents

- Mechanisms for identifying onset of problems
- Mechanisms for identifying root cause
- Range of firefighting options
- Range of solution options
- Means for keeping systems “liquid”

Musings

Building the toolkit

Beware frameworks

- Enforce structure on architecture and code
  - Controlling
    - Real libraries are controlled
- Frameworks that define system architecture are dangerous
  - e.g. J2EE favours ACID
    - What if we choose a different CAP tradeoff?

Conspiracy of one

- Assuming there is only ever one of anything hurts
  - One database
  - One message queue
  - One cache
- Partitioning is a key weapon in (performance) scaling
  - e.g. Consistent hashing
- Location finding abstractions in code from day one?

Conspiracy of one

Cleanliness is everything?

- Cohesive design leads to:
  - Clear roles
  - Loose coupling
- Monolithic architectures typically have ill-defined roles and tight coupling
  - Hard to change
- Databases
  - Natural tendency towards tight coupling?

Cost of change

- Some don't ever have to change:
  - 37 Signals still have simple RoR 3-tier
- Some paint themselves into an expensive corner
  - Amazon got a bank loan to fix their problems
- Fixing a scalability problem is costly
  - Estimated cost of fix as indicator of priority?

Metrics

- Define an architecture
  - Determine a set of performance limits
    - Instrument the code
      - Track % of performance limits
      - Track stress indicators
    - Means for spotting issues early

Metrics - Part II

- Care is required in selection of metrics
  - Customer doesn't care about technical metrics
    - e.g. 2ms response time from service X
  - Cares about overall picture
    - 2 second rule
    - Availability of their information
    - The ability to place an order

CAP Theorem

- Consistency, availability, partitioning
  - Pick any two (kinda)

Transactions

- Transactions typically ACID
  - Use often related to programmer certainty - “something happened right now”
    - Avoids need for concurrent/asynchronous thinking
- ACID not always required
  - Trading systems can tolerate inconsistency temporarily for some things

Asynchronous

- Best for load management and failure handling
- Many programmers can’t think asynchronous
  - Tendency to synchronous style even with message queues or http
- Disruptive change
- Asynchronous from day one?

Failure becomes inevitable

- Get big enough and many machines fail in a day
  - Dean: 10000 machines means failure every day
- Should we ever assume hardware is reliable in software?
- Handling failure in software is hard
- Can’t hide it in frameworks
  - No such thing as magic

Upgrades

- Early on we can shut service down - lockstep upgrade
  - Some (e.g. banks) always have downtime windows
- Incremental (hot) upgrades required as we grow
  - Cost of downtime too high
- Changing packaging and deployment is painful
  - Package is process?
  - Puppet vs Autopilot/Red Dog

Local Decisions

- Classic means for ensuring a distributed system continues in face of failure
- Conveniently reduces network roundtrips
  - Encourages caching and exploitation of staleness

Decentralisation

- P2P is under-exploited in systems
  - Gossip potentially a good solution for scalable, reliable messaging
  - Auto-configuration saves on operational costs
  - Powerful monitoring capability - e.g. Astrolabe

Controlling load

- When a system begins to melt
  - Limiting active users is useful - admission control
    - System remains accessible to some
  - Throttling and rejection - limits impact of excessive load

Let’s get it on...

Toolkit contents

- Mechanisms for identifying onset of problems
- Mechanisms for identifying root cause
- Range of firefighting options
- Range of solution options
- Means for keeping systems “liquid”
- Representation: Patterns, philosophy, libraries/ frameworks