On Designing and Deploying Internet-Scale Services

Tom Rochette <tom.rochette@coreteks.org>

July 24, 2025 — daae079c

- 3 tenets
  - Expect failures
  - Keep things simple
  - Automate everything
- The entire service must be capable of surviving failure without human administrative interaction
- The best way to test the failure path is never to shut the service down normally. Just hard-fail it
- The acid test: is the operations team willing and able to bring down any server in the service at any time without draining the work load first?
  - If they are, then there is synchronous redundancy (no data loss), failure detection, and automatic take-over
- Large clusters of commodity servers are much less expensive than the small number of large servers they replace
- Server performance continues to increase much faster than I/O performance, making a small server a more balanced system for a given amount of disk
- Power consumption scales linearly with servers but cubically with clock frequency, making higher performance servers more expensive to operate
- A small server affects a smaller proportion of the overall service workload when failing over
- Two factors that make some services less expensive to develop and faster to evolve than most packaged products are
  - the software needs to only target a single internal deployment
  - previous versions don't have to be supported for a decade as is the case for enterprise-targeted products
- Basic design tenets
  - Design for failure
  - Implement redundancy and fault recovery
  - Depend upon a commodity hardware slice
  - Support single-version software
  - Enable multi-tenancy
- Each pod should be as close to 100% independent and without interpod correlated failures
- What isn't tested in production won't work, so periodically the operations team should a fire drill using these tools
- If the service-availability risk of a drill is excessively high, then insufficient investment has been made in the design, development, and testing of the tools
- Some form of throttling or admission control is common at the entry to the service, but there should also be admission control at all major components boundaries
- The general rule is to attempt to gracefully degrade rather than hard failing and to block entry to the service before giving uniform poor service to all users
- Partitions should be infinitely-adjustable and fine-grained, and not be bounded by any real world entity
  - We recommend using a look-up table at the mid-tier that maps fine-grained entities, typically users, to the system where their data is managed
- Expect to run in a mixed-version environment. The goal is to run single version software but multiple versions will be live during rollout and production testing
- Best practices in designing for automation include

  - Be restartable and redundant
  - Support geo-distribution
  - Automatic provisioning and installation
  - Configuration and code as a unit
  - Manage server roles or personalities rather than servers
  - Multi-system failures are common
  - Recover at the service level
  - Never rely on local storage for non-recoverable information
  - Keep deployment simple
  - Fail services regularly
- Dependency management
  - Expect latency
  - Isolate failures
  - Use shipping and proven components
  - Implement inter-service monitoring and alerting
  - Dependent services require the same design point
  - Decouple components
- Testing in production is a reality and needs to be part of the quality assurance approach used by all internet-scale services
- The following rules must be followed
  - The production system has to have sufficient redundancy that, in the event of catastrophic new service failure, state can be quickly recovered
  - Data corruption or state-related failures have to be extremely unlikely (functional testing must first be passing)
  - Errors must be detected and the engineering team (rather than operations) must be monitoring system health of the code in test
  - It must be possible to quickly roll back all changes and this roll back must be tested before going into production
- Big-bang deployments are very dangerous
- We favor deployment mid-day rather than at night
- Some best practices for release cycle and testing include
  - Ship often
  - Use production data to find problems
    * A few strategies
      · Measurable release criteria
      · Tune goals in real time
      · Always collect the actual numbers
      · Minimize false positives
      · Analyze trends
      · Make the system health highly visible
      · Monitor continuously
      · Invest in engineering
      · Support version roll-back
      · Maintain forward and backward compatibility
      · Single-server deployment
      · Stress test for load
      · Perform capacity and performance testing prior to new releases
      · Build and deploy shallowly and iteratively
      · Test with real data
      · Run system-level acceptance tests
      · Test and develop in full environments
- Best practices for hardware selection include
  - Use only standard SKUs
  - Purchase full racks

    - Write to hardware abstraction
    - Abstract the network and naming
- Make the development team responsible
- Soft delete only
- Track resource allocation
- Make one change at a time
- Make everything configurable
- To be effective, each alert has to represent a problem
- To get alerting levels correct, two metrics can help and are worth tracking
    - alerts-to-trouble ticket ratio (with a goal of near one)
    - number of systems health issues without corresponding alerts (with a goal of near zero)
- Best practices include
    - Instrument everything
    - Data is the most valuable asset
    - Have a customer view of service
    - Instrument for production testing
    - Latencies are the toughest problem
    - Have sufficient production data
        * The most important data we've relied upon includes
            · Use performance counters for all operations
            · Audit all operations
            · Track all fault tolerance mechanisms
            · Track operations against important entities
            · Asserts
            · Keep historical data
    - Configurable logging
    - Expose health information for monitoring
    - Make all reported errors actionable
    - Enable quick diagnosis of production problems
        * Give enough information to diagnose
        * Chain of evidence
        * Debugging in production
        * Record all significant actions
- Two best practices, a "big red switch" and admission control, need to be tailored to each service
- Support a "big red switch"
    - The ability to shed non-critical load in an emergency
- Control admission
- Meter admission