Designing and Deploying Internet-Scale Services

James Hamilton

2008.12.02
Architect, Data Center Futures
e: JamesRH@microsoft.com
w: research.microsoft.com/~jamesrh
w: perspectives.mvdirona.com
Background & Biases

• 15 years in database engine development
  – Lead architect on IBM DB2
  – Architect on SQL Server
    • Led variety of core engine teams including SQL client, SQL compiler, optimizer, XML, full text search, execution engine, protocols, etc.

• Led the Exchange Hosted Services Team
  – Email anti-spam, anti-virus, and archiving for 2.2m seats with $27m revenue
  – ~700 servers in 10 data centers world-wide

• Architect on Windows Live Platform Services

• Currently Data Center Futures Architect

• Automation & redundancy is only way to:
  – Reduce costs
  – Improve rate of innovation
  – Reduce operational failures and downtime
Agenda

• Motivation & Overview
• Recovery-Oriented Computing
• Overall Application Design
• Operational Issues
• Summary

Contributors: Search, Mail, Exchange Hosted Services, Live Collaboration Server, Contacts & Storage, Spaces, Xbox Live, Rackable Systems, Messenger, WinLive Operations, & MS.com Ops
Motivation

- System-to-admin ratio indicator of admin costs
  - Tracking total ops costs often gamed
    - Outsourcing halves ops costs without addressing real issues
  - Inefficient properties: <10:1
  - Enterprise: 150:1
  - Best services: over 2,000:1
- 80% of ops issues from design and development
  - Poorly written applications are difficult to automate
- Focus on reducing ops costs during design & development
What Does Operations do?

- **51% is deployment & incident management (known resolution)**

- **Teams:** Messenger, Contacts and Storage & business unit IT services

Source: Deepak Patil, Global Foundation Services (8/14/2006)

http://perspectives.mvdirona.com
ROC Design Pattern

• Recover-oriented computing (ROC)
  – Assume software & hardware will fail frequently & unpredictably
• Heavily instrument applications to detect failures

- **Bohr bug**: Repeatable functional software issue (functional bugs); should be rare in production
- **Heisenbug**: Software issue that only occurs in unusual cross-request timing issues or the pattern of long sequences of independent operations; some found only in production

- Machine out of rotation and power down
- Set LCD/LED to "needs service"
Overall Application Design

• Development and testing with full service
  – Single-box deployment
  – Quick service health check
• Pod or cluster independence
  – Zero trust of underlying components
• Implement & test ops tools and utilities
• Simplicity throughout
• Partition & version everything
Design for Auto-Mgmt & Provisioning

- Never rely on local, non-replicated persistent state
- Support for geo-distribution
- Auto-provisioning & auto-installation mandatory
  - Explicitly install everything & then verify
  - Manage "service role" rather than servers
- Multi-system failures are common
  - Limit automation range of action
- Force fail all services and components regularly
  - Don't worry about clean shutdown
    - Often won't get it & need this path tested
Release Cycle & Testing

• Ship frequently:
  – Small releases ship more smoothly
  – Increases pace of innovation
  – Long stabilization periods not required in services

• Use production data to find problems (traffic capture)
  – Measurable release criteria
  – Release criteria includes quality and throughput data

• Track all recovered errors to protect against automation-supported service entropy

• Test all error paths in integration & in production

• Test in production via incremental deployment & roll-back
  – Never deploy without tested roll-back
  – Continue testing after release
Design for Incremental Release

- Incrementally release with schema changes?
  - Old code must run against new schema, or
  - Two-phase process (avoid if possible)
    • Update code to support both, commit changes, and then upgrade schema
- Incrementally release with user experience (UX) changes?
  - Separate UX from infrastructure
  - Ensure old UX works with new infrastructure
  - Deploy infrastructure incrementally
  - On success, bring a small beta population onto new UX
  - On continued success, announce new UX and set a date to roll out
- Client-side code?
  - Ensure old & new clients both run with new infrastructure
Graceful Degradation & Admission Control

• No amount of "head room" is sufficient
  – Even at 25-50% H/W utilization, spikes will exceed 100%
• Prevent overload through admission control
• Graceful degradation prior to admission control
  – Find less resource-intensive modes to provide (possibly) degraded services
• Related concept: Metered rate-of-service admission
  – Service login typically more expensive than steady state
  – Allow a single or small number of users in when restarting a service after failure
Auditing, Monitoring & Alerting

- Produce perf data, health data & throughput data
- All config changes need to be tracked via audit log
- Alerting goals:
  - No customer events without an alert (detect problems)
  - Alert to event ratio nearing 1 (don’t false alarm)
- Alerting is an art ... need to tune alerting frequently
  - Can’t embed in code (too hard to change)
  - Code produces events, events tracked centrally, alerts produced via queries over event DB
- Testing in production requires very reliable monitoring
  - Combination of detection & capability to roll back allows nimbleness
- Tracked events for all interesting issues
  - Latencies are toughest issues to detect
Dependency Management

• Expect latency & failures in dependent services
  – Run on cached data or offer degraded services
  – Test failure & latency frequently in production
• Don’t depend upon features not yet shipped
  – It takes time to work out reliability & scaling issues
• Select dependent components & services thoughtfully
  – On-server components need consistent quality goals
  – Dependent services should be large granule (“worth” sharing)
• Isolate services & decouple components
  – Contain faults within services
  – Assume different upgrade rates
  – Rather than auth on each connect, use session key and refresh every N hours (avoids login storms)
Customer & Press Communications Plan

- Systems fail & you will experience latency
- Communicate through multiple channels
  - Opt-in RSS, web, IM, email, etc.
  - If app has client, report details through client
- Set ETA expectations & inform
- Some events will bring press attention
- There is a natural tendency to hide systems issues
- Prepare for serious scenarios in advance
  - Data loss, data corruption, security breach, privacy violation
- Prepare communications skeleton plan in advance
  - Who gets called, communicates with the press, & how data is gathered
  - Silence typically interpreted as hiding something or lack of control
Summary

• Reduced operations costs & improved reliability through automation
• Full automation dependent upon partitioning & redundancy
• Each human administrative interaction is an opportunity for error
• Design for failure in all components & test frequently
• Rollback & deep monitoring allows safe production testing
More Information

• Designing & Deploying Internet-Scale Services paper:

• Autopilot: Automatic Data Center Operation

• Recovery-Oriented Computing
  – http://roc.cs.berkeley.edu/
  – http://www.cs.berkeley.edu/~pattrsn/talks/HPCAkeynote.ppt
  – http://www.sciam.com/article.cfm?articleID=000DAA41-3B4E-1EB7-BDC0809EC588EEDF

• These slides:
  – Will be posted to http://research.microsoft.com/~jamesrh later in the week

• Email:
  – JamesRH@microsoft.com

• External Blog:
  – http://perspectives.mvdirona.com