Testing NFR

江少傑 Ruckus Networks

Introduction: what is NFR?

- Terminology:
  - Performance
  - Throughput
  - Capacity
- Fallacies and pitfalls: both extremes
- NFR (or cross-functional requirement? System characteristics?)
  - Availability
  - Capacity
  - Security

Managing NFRs

- Crosscutting nature
- Easily overlooked in the beginning
- Choose architecture: analysis
- ATAM (Architectural Tradeoff Analysis Method)
  - Security ⟵⟶ Usability
  - Flexibility ⟵⟶ Performance
  - arch, schedule, test and cost

Analyzing NFRs

- Just like functional requirements: acceptance criteria
- Reasonable level of details:
  - “as fast as possible?” Elaborate so we can estimate and prioritize
  - “take less than 2 seconds to respond?” under what circumstances?
  - usability instead of performance requiments.

Programming for Capacity

- Knuth: “Premature optimization is the root of all evil”
  - 97% should be forgotten, focus on critical 3% and prioritize
  - don’t guess, measure
- Overengineering
  - Simplicity is performant (ex. 7 hand-crafted queues)
  - Comm across process and network is costly
- Procrastination: we could fix capacity issues later

Capacity strategies

1. Architecture (proc, net, IO)
2. Patterns/antipatterns: Nygard’s “Release It!”
3. Clarity and simplicity over esoterica, unless there is proof.
4. Data structures and algorithms.
5. Threading: blocked threads leading to cascading failures
6. Automate capacity tests
7. Profiling tool to fix problems identified by tests
8. Real-world capacity measures: number and patterns.

Measuring Capacity

- Scalability
- Longevity
- Throughput
- Load

- First 2: relative, second 2: absolute
- Reflecting reality: alternative path parallelism
- Scenario based

Capacity test success/fail definition

- Measurements: graph, trending and incorporated into pipeline
- Thresholding
  - Too high: regular, intermittent failures
  - Too low: let culprits slip away
  - Less virtualization: overhead
  - Ratcheting to protect against capacity-damaging changes
- Reality: interleaved parallelism and composite scenario interactions

Capacity Testing Environment

- Replica of production
  - Entirety
  - Core
  - Ex: cluster of ipods
- Canarying
- Linear scaling factor? Naïve…
- Buy all the equip money can buy to identify issues earlier
- Single leg: proportioned to simulate contention

Automating Capacity Testing

- It’s a promethean enterprise in its own right
  - Integrate simple/light ones into commit stage

1. Real-world scenario
2. Predefined threshold
3. Reasonably short
4. Robust against change
5. Composable into large-scale complex
6. Repeatable and could run sequentially or in parallel

Automating Capacity Testing

- Adapt from existing acceptance tests
  - Realism
  - Pathology
- Record/replay through:
  - UI
  - Service or public API
  - Lower-level API

Capacity Testing via UI

- Most popular choice among commercial products
- Reality: ratio of client to servers (thick clients)
- High maintenance cost
- Tests against API are less fragile
- Separate UI test: injection or against backend stub
- Summary: avoid UI

Recording Interactions against pub API

- web service
- message queues
- Event-driven comm
- SOA

Using Recorded Interaction Templates

- Instrument acceptance tests and record them to disk
- As little tagging as possible: limit coupling between test and test data
- Add success criteria for the template
- To feed open source perf test tools: Jmeter, locust, or gatling
- Caveat: test code should be the most performant, not prod
- Not down to tweaking clocks and machine code yet

JMeter vs locust vs gatling

- Use echo server to benchmark them: we want test code/framework to be robust and reliable
- Simple java echo server setup
- Jmeter: 8000
- Locust: 12xxx
- Gatling: 22xxx
- Gatling.io wins with reliability and performance as well as versatility

Capacity Test Stubs to Develop Tests

- Writing test is harder than writing the code it tests for
- No-op stub (like echo server previously)
- Make sure tests withstand testing scenarios:
  - Asserting test itself is valid before we trust its results.
  - Reported alongside capacity test results.

Adding Capacity Tests to CD Pipeline

- Minimum capacity threshold: good enough
- Guarding the trunk against capacity degrading changes like other tests
- YAGNI: You Ain’t ‘Gonna Need It (Knuth)
- Beware of runtime optimizing compilers: warm-up
- Hot-spot perf smoke tests as part of commit stage

Adding Capacity Tests to CD Pipeline

- Rule out long-running, self-sustaining ones
  - Complication
  - Long time to run
  - Demo, manual testing… etc run in parallel with capacity testing
  - Don’t run capacity tests as frequently as acceptance tests
- Capacity test as a wholly separate stage (appended to CD pipeline)
- Safeguarding RC (no pass no deploy)
- Appended to acceptance test stage

Additional Benefits of Capacity Test System

- Diagnostics in prod-like system
  - Repro complex prod defects
  - Mem leaks
  - Longevity testing
  - Impact of GC
  - Tuning GC
  - Tuning 3rd-party app config, OS, app server, db
  - Simulating pathology
  - Eval solutions to complex problems
  - Measuring scalability over different HW config
  - Load-testing comm with external systems
  - Rehearsing rollback
  - Chaos engineering (graceful degradation)
  - Real-world capacity benchmarks: scaling factors
  - Etc…

Summary

- Don’t go extremes
- NFRs force tech ppl to provide more input for analysis: focus on biz
- Like building bridge: guard against our own tendency to see tech solutions first
- Capture NFRs just like functional ones and select appropriate arch.
- Automate capacity tests as part of CD pipeline (commit to capacity)