What I’ve learned doing chaos at Netflix

Lorin Hochstein (@lhochstein)
Chaos engineering should not be endorsed by the ICSE community. Accepting a workshop pretty much endorses the topic.

-- Reviewer, rejected ICSE’16 chaos workshop proposal
Some context about Netflix
We care about availability
Whoops, something went wrong...

Netflix Streaming Error
We’re having trouble playing this title right now. Please try again later or select a different title.
SPS: Stream starts Per Second

Number of people who hit the “play” button and successfully started
99.95%
Microservice architecture
Many points of failure!
A play in three acts

- Act I: Chaos at Netflix when I got there
- Act II: Chaos as experimentation
- Act III: Lessons learned
Act I: Chaos at Netflix when I got there
Only Chaos Monkey was in use
Chaos Monkey randomly terminates instances in production
Chaos Monkey had already exposed single-instance termination weaknesses
Latency monkey was too dangerous
FIT: Failure Injection Testing
Inject failure or latency at "injection points" in code
Example injection point: remote procedure call
Failures are scoped, not random
Example: Is the bookmarks service critical?
| api | ----> | bookmarks |
Fail calls from the "api" service to the "bookmarks" service for account "123456"

| api | --×--> | bookmarks |
Many service failures look like errors or latency
Great for testing with a single device
Some problems only appear when many calls fail
503 Service Unavailable
FIT supported large-scale failure injection
Example: Inject failure for 10% of customer traffic
How much should you inject?
Too much: unnecessary customer pain
Too little: can't tell if there's a vulnerability
Did this have impact?
Act II: Chaos as experimentation
Chaos Automation Platform
Want: clear signal if failure injection having negative impact...
...on customers...
...and on services
Big idea: stickiness
Failure injection sessions are sticky to users
Failure injection sessions are sticky to clusters
We can do controlled experiments!
Upstream

98%

API

API - baseline

API - canary

fail

Bookmarks
CPU utilization

- **Baseline**
  - Max: 57.500
  - Min: 1.000
  - Avg: 45.981
  - Last: 3.000
  - Tot: 2.321k
  - Cnt: 54.000

- **Canary**
  - Max: 67.000
  - Min: 2.000
  - Avg: 46.509
  - Last: 2.000
  - Tot: 2.321k
  - Cnt: 54.000

- **Experiment**
  - Max: 1.000
  - Min: 0.000
  - Avg: 810.182m
  - Last: 0.000
  - Tot: 45.000
  - Cnt: 55.000

... 1 of 1 lines matched filter ...

Frame: 55k, End: 2018-11-27T14:42:88.000Z(us/Pacific), Step: In
Fetch: 493ms (L: 16.0, B: 0, 4: 0, G: 960.0, 448.0, 228 0.0)
How do we do this safely?
STOP CHAOS
Automatic stop
(<5 minutes)
Business hours only
Limit number of simultaneous runs
How do we scale this?
First attempt: self-serve
Actively engage with multiple teams
Didn't see uptake after engagements 😞
Second attempt: automatically generate experiments
Problem: need to understand services to design experiments
What other services do they communicate with?
Tracing!
Which RPCs do we believe are safe to fail?
Heuristics!
Is there a fallback?
Does the fallback ever get invoked?
How much latency should we inject?
NIWS Client Latency and Timeouts

- **Average Latency**
  - Max: 8.329
  - Min: 3.647
  - Avg: 3.119
  - Last: 3.122
  - Tot: 2.575k
  - Cnt: 503,060

- **95th Percentile Latency**
  - Max: 45.888
  - Min: 5.637
  - Avg: 7.806
  - Last: 7.317
  - Tot: 3.526k
  - Cnt: 503,060

- **99th Percentile Latency**
  - Max: 58.698
  - Min: 14.730
  - Avg: 29.339
  - Last: 18.798
  - Tot: 19.321k
  - Cnt: 503,060

- **99.5th Percentile Latency**
  - Max: 95.711
  - Min: 23.781
  - Avg: 39.247
  - Last: 32.656
  - Tot: 17.724k
  - Cnt: 503,060

- **Configured Timeout 1: 500 ms**
  - Max: 500.060
  - Min: 500.060
  - Avg: 500.060
  - Last: 500.060
  - Tot: 252.060k
  - Cnt: 1,004,060

Frame: 1v
End: 2638-09-21T23:06:07-07:00[US/Pacific], Step: 20m
Fetch: 913ms (L: 242.2k, 11.9k, 3.9k, 0: 14.5m, 5.9m, 2.5m)
We found vulnerabilities!
Still requires human effort to interpret results
Experimental design limited by our heuristics
Current state: hybrid approach
Busy season: right before the holidays
Act III: Lessons learned
Safety
It needs to be safe, or nobody will use it
Safe = limited impact
Simplicity is prerequisite for reliability

-- Edsger Dijkstra

No!
Safety adds complexity
WithinLimit \triangleq \text{Sum(running)} \leq \text{TrafficLimit}

TypeOK \triangleq \wedge \text{queue} \in \text{SUBSET Runs}
\wedge \text{owned} \in \text{SUBSET Runs}
\wedge \text{running} \in \text{SUBSET Runs}
\wedge \text{traffic} \in [\text{Runs} \rightarrow \text{Nat} \setminus \{0\}]
\wedge \text{candidate} \in [\text{ProcSet} \rightarrow \text{Runs} \cup \{\text{NoRun}\}]
\wedge \text{known} \in [\text{ProcSet} \rightarrow \text{SUBSET Runs}]
\wedge \text{pc} \in [\text{ProcSet} \rightarrow \{\text{"p1"}, \text{"p2"}, \text{"p3"}, \text{"Done"}\}]

Inv \triangleq \wedge \text{TypeOK}
\wedge \forall i \in \text{ProcSet} : \text{known}[i] \subseteq \text{owned}
\wedge \forall i \in \text{ProcSet} : \forall \text{candidate}[i] = \text{NoRun}
\wedge \forall \text{candidate}[i] \in \text{owned}
\wedge \forall \text{run} \in \text{running} : \exists i \in \text{ProcSet} : \forall \text{pc}[i] = \text{"Done"}
\wedge \forall \text{candidate}[i] = \text{run}
\wedge \forall i, j \in \text{ProcSet} : \forall \text{known}[i] \subseteq \text{known}[j]
\wedge \forall \text{known}[j] \subseteq \text{known}[i]
\wedge \text{WithinLimit}

\text{ASSUME } \text{NumWorkersInNat} \triangleq \text{NumWorkers} \in \text{Nat} \setminus \{0\}
\text{ASSUME } \text{TrafficLimitInNat} \triangleq \text{TrafficLimit} \in \text{Nat} \setminus \{0\}

\text{LEMMA } \text{SumPrime} \triangleq \forall S \in \text{SUBSET Runs} : (\text{Sum}(S))' = \text{Sum}(S')

\text{LEMMA } \text{EmptySwinsIsZero} \triangleq \text{Sum}() = 0

\text{LEMMA } \text{SumInNat} \triangleq \forall S \in \text{SUBSET Runs} : \text{Sum}(S) \in \text{Nat}

\text{THEOREM } \text{Spec} \Rightarrow \square \text{WithinLimit}
(1) \text{ USE } \text{DEF ProcSet, Inv}
(1)1. \text{Init} \Rightarrow \text{Inv}
(2) \text{SUFFICES } \text{ASSUME } \text{Init}
\text{ PROVE } \text{Inv}

\text{OBIVIOUS}
(2)1. \text{TypeOK}
\text{ BY } \text{DEF TypeOK}
(2)2. \forall i \in \text{ProcSet} : \text{known}[i] \subseteq \text{owned}
\text{OBIVIOUS}
(2)3. \forall i \in \text{ProcSet} : \forall \text{candidate}[i] = \text{NoRun}
\text{ BY } \text{DEF TypeOK}
\forall \text{candidate}[i] \in \text{owned}
You better have damn good tests around your failure injection logic...
...especially if it's a shared library in every app!
18:18:00,094 ERROR FitContextImpl:195 - Fit Error checking or injecting failure
java.lang.NullPointerException
    at com.netflix.fit.InjectionPointImpl.wildcardMatch(InjectionPointImpl.java:133)
    at com.netflix.fit.scenario.FitScenarioImpl.shouldImpact(FitScenarioImpl.java:45)
    at com.netflix.fit.FitContextImpl.shouldInjectFailure(FitContextImpl.java:130)
    at com.netflix.fit.FitContextImpl.checkAndInjectFailure(FitContextImpl.java:191)
    at com.netflix.fit.FitContext.checkAndInjectFailure(FitContext.java:40)
    at com.netflix.server.base.fit.FitHandler.handle(FitHandler.java:34)
    at com.netflix.server.base.NFFilter.safeDoFilter(NFFilter.java:574)
    at com.netflix.server.base.NFFilter.access$200(NFFilter.java:234)
    at com.netflix.server.base.NFFilter$3.call(NFFilter.java:482)
    at com.netflix.server.base.NFFilter$3.call(NFFilter.java:479)
    at com.netflix.lang.BindingContexts.callWithNewContext(BindingContexts.java:182)
    at com.netflix.server.base.NFFilter.doFilter(NFFilter.java:479)
    at com.google.inject.servlet.FilterChainInvocation.doFilter(FilterChainInvocation.java:82)
    at com.google.inject.servlet.ManagedFilterPipeline.dispatch(ManagedFilterPipeline.java:120)
    at com.google.inject.servlet.GuiceFilter.doFilter(GuiceFilter.java:135)
    at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:240)
    at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:207)
    at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:212)
    at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:106)
    at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:502)
    at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:141)
ChAP isn't a "black box"
Experimental design is a skill
Work isn't done when automated experiment reveals a weakness
Confirm it's a genuine problem
Communicate effectively back to service owners
Lots of tuning required
Length of experiment
Amount of traffic impacted
Auto-stop thresholds
Error counts are noisy
Leverage your internal tooling ecosystem
ChAP is really an orchestration tool

- Fault injection
- Sticky routing
- Continuous deployment
- Tracing
- Telemetry
- Automated canary analysis
The more heterogeneous your ecosystem, the harder life will be
Java -> Node.js
REST -> gRPC
VMs -> containers
Unexpected benefits
Info for experiment generation was useful to service owners
NIWS Client Latency and Timeouts

**Average Latency**
Max : 8.329  Min : 3.647  Avg : 3.119  Last : 5.122  Tot : 2.575k  Cnt : 503.000

**95th Percentile Latency**
Max : 45.888  Min : 5.637  Avg : 7.806  Last : 7.517  Tot : 3.525k  Cnt : 503.000

**99th Percentile Latency**
Max : 58.698  Min : 14.736  Avg : 29.539  Last : 18.798  Tot : 10.311k  Cnt : 503.000

**99.5th Percentile Latency**
Max : 95.711  Min : 23.781  Avg : 39.247  Last : 32.656  Tot : 17.725k  Cnt : 503.000

**Configured Timeout 1: 500 ms**
Max : 500.000  Min : 500.000  Avg : 500.000  Last : 500.000  Tot : 252.066k  Cnt : 504.000

Frame: 1v, End: 2618-09-02T23:00-07:00[US/Pacific], Step: 20m  
Fetch: 915ms (L: 542.2k, 11.8k, 5.9k, D: 14.5k, 5.9k, 2.5k)
Engineers created new use cases (sticky canary)
Image credits

- "Knobs", Ian Harding, CC-BY-NC-SA 2.0: https://flic.kr/p/d9sCZ