Orchestrating Mayhem
Functional
Chaos Engineering

Szymon Mentel
szymon.mentel@erlang-solutions.com
@szymonmetel
github.com/mentels
1. Chaos: WHAT WHY HOW
WHAT is Chaos: complex systems
**WHAT** is Chaos: complex systems

**Interactions** compounded with **real-world** events may lead to unpredictable outcomes
Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system’s capability to withstand turbulent conditions in production.

~ http://principlesofchaos.org/
Chaos Engineering: experimenting
What is the rationale for Chaos Engineering?

- Trust
- Being Proactive
- Cost-effectiveness
HOW: Steady State
HOW: 4 steps

1. Define
2. Hypothesize
3. Introduce variables
4. Try to disprove hypothesis

Steady State

Steady State will continue

Learnt Behaviour

Real-World Event

Control Group

Experimental Group
HOW: Chaos Engineering at Netflix

How API handles failure of Rating service?
2. Applying Chaos Engineering Principles
Applying ChE: Injection Points

- Message Archive Database
- MongooseIM (XMPP)
- Erlang Distribution
- Authentication Service
- Native protocol
- Native protocol
- HTTP

Client

Streaming Service
Applying ChE: Comparing

Control Load Test

Scenario

Experimental Load Test with Hypothesis

Load Generator → MongooseIM

Load Generator → MongooseIM

Fault Injection

Compare
Hypothesis 1: Database Failure

Failure to write to the database won’t disrupt the service
Hypothesis 1: Database Failure

# setup
kill_pool = fn f ->
    workers = Supervisor.which_children(PoolSup)
    Enum.each(workers,
        fn pid -> Process.exit(pid, :chaos) end)
    Process.sleep(10)
    f.(f)
end
# run
killer = spawn(fn -> kill_pool.(kill_pool) end)
# stop
Process.exit(killer, :kill)
Hypothesis 1: Database Failure

Chaos Started

Request Response Time

Chaos Stopped
Hypothesis 2: Slow Network

Delay on the connection to the Authentication Service won’t prevent users from logging in
Hypothesis 2: Slow Network

# setup
delay = 100
delayed_auth = fn user, pass ->
    Process.sleep(delay)
    AuthService.authenticate(user, pass)
end
:ok = :meck.new(AuthService, [:passthrough])

# run
:ok = :meck.expect(AuthService, :authenticate,
    fn user, pass -> delayed_auth(user, pass) end)

# stop
:ok = :meck.unload(AuthService)
Hypothesis 2: Slow Network

The affected node

Not affected nodes

Session Count
Hypothesis 3: Network Glitch

Network glitch on the connection to Kafka won’t cause any data loss
Hypothesis 3: Network Glitch

# setup

cmd = ["INPUT -m state --state NEW,ESTABLISHED,RELATED -p tcp --dport 9092 -s #{mim_addr} -j DROP"]
enable_cmd = "-A" <> cmd
disable_cmd = "-D" <> cmd

# run

{_, 0} = System.cmd("iptables", enable_cmd)

# stop

{_, 0} = System.cmd("iptables", disable_cmd)
Hypothesis 3: Network Glitch

Kafka Messages

Network Recovery

- Produced
- Retries
- Consumed

Application Retries

Kafka Driver Queue

M M M M
Fault Injection: no recompilation

Direct or Remote

- RPC
  
  ```erlang
  :rpc.call(:aff_node@localhost, PoolSup, :which_children, [])
  ```

- Remote processes
  
  ```erlang
  Node.spawn(:aff_node, fn -> ... end)
  ```
3. MongooseIM Chaos Engineering Automation
Automation of Chaos Engineering

Chaos Experiment

- Control Load Test
- Load Generator
- MongooseIM
- Load Generator
- MongooseIM
- Fault Injection

Chaos Automation (Typhoon)

Experimental Load Test with Hypothesis

Compare
Typhoon: Elixir Application

```
apps
├── typhoon
│   └── lib
│       └── typhoon
│           └── chaos_manager
│           └── fault_injection
│                     └── infrastructure
└── typhoon_web
    └── lib
        └── typhoon_web
```
**Typhoon: Elixir Application**

Infrastructure

```elixir
%TestSetup{}
%TestTopology{}
```

Fault Injection

Fault Protocol

```elixir
%MyFault{}
```

Chaos Manager

```elixir
%Experiment{
  :control_test_id,
  :experimental_test_id,
  :setup,
  :faults,
  :faults_offsets,
}
```
Typhoon: Elixir Application

Load Test Config: ...
Faults: ...

TyphoonWeb

1. %Setup{} (Control Test)
2. %Setup{} (Exp. Test)

Chaos Manager

%Experiment{}

%Topology{}
[faults]

Fault Injection

Infrastructure

Load Testing Infrastructure

Steady State

Compare

Learnt Behaviour
Typhoon: Fault Injection

defprotocol FaultInjection.Fault do
  @doc "Applies the fault to the load test run by `test_id`"
  @spec apply(struct(), test_id()) :: :ok | {:error, term()}
  def apply(fault, test_id)
end

defimpl FaultInjection.Fault, for: MyFault do
  def apply(fault, test_id), do: send self, %{fault: MyFault}
end
Typhoon: Fault Injection

defmodule FaultInjection.Fault.MyFault do
  embedded_schema do
    field(:param1, :integer)
    field(:param2, :string)
  end

  def changeset(struct, attrs) do
    struct
    |> cast(attrs, [:param1, :param2])
    |> validate_required([:param1, :param2])
  end

  defimpl FaultInjection.Fault do
    def apply(fault, test_id), do: send self, %{fault: MyFault}
  end
end
CHAOS ENGINEERING
is for everyone - go and explore it

APPLY
it to your system using the most basic techniques available

AUTOMATE
if it works for you add it to your continuous integration pipeline
THANK YOU!

szymon.mentel@erlang-solutions.com
@szymonmetel
github.com/mentels
medium.com/@mentels