Frisbee
Automated Testing Over Kubernetes

Fotis Nikolaidis
Systems Research Engineer
ICS-FORTH, Greece
A story about a lab ...
A story about a lab …

- The fun …
  - Writing proper tests require the same level of engineering effort as the system they test! Much less attractive to write ….

- The misery …
  - As opposed to features, the value of tests is seldom visible in the short term
  - Test automation becomes essential as the projects grow.

Systems Software
- Storage systems and I/O
- Datacenter Resource Management and Scheduling
- Datacenter Accelerators
- Low-overhead, RDMA-based Communication Protocols
- Frameworks and Platforms for High-Performance Data Processing
- Cloud-native computing
Systems Testing Challenges

- Multiple interacting components
- Multiple programming environments
- Multiple libraries
- Multiple applications
- Multiple architectures
- Multiple clusters
- Functionality and performance are both important
Key Requirements

- Allow researchers to focus on their systems, minimizing distractions from testing issues.
- Offers a fully automated and disposable testing environments with the tools that researchers may need.
- Help researchers under the performance and behavior of their systems, under various operating conditions.
- Automatically validates the system for transition into erroneous states or SLA violations.
- Integration with CI/CD pipelines to test early, and test often!
The testing landscape

- 😞 Binary Processes on Physical Infrastructure
- 😞 Binary Processes on VMs
- 😞 Containers on Docker
- 😞 Containers on Docker-Compose
- 😞 Containers on Kubernetes
- 😞 Containers on Kubernetes with CI/CD pipelines
- 😎 Frisbee: A Kubernetes-native testing platform
The testing landscape

- Binary Processes on Physical Infrastructure
  - Manual scripts to deploy software and synchronize test-steps.
  - Portability of the test
  - High maintenance cost
  - Low execution overheads
The testing landscape

- Binary Processes on VMs
  - Manual scripts to deploy software and synchronize test-steps.
  - Portability of the test
  - High maintenance cost
  - High execution overheads
The testing landscape

- Containers on Docker
  - Manual scripts to deploy software and synchronize test-steps.
  - Portability of the test
  - Low maintenance cost
  - Low execution overhead
The testing landscape

- Containers on Docker-Compose
  - Rich and expressive DSL for writing multi-stage scenarios
  - Portability of the test
  - Low maintenance cost
  - Low execution overhead
  - Bound to a single node

```yaml
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
    redis:
      image: redis
```

```
$ docker-compose up
Creating network "composetest_default" with the default driver
Creating composetest_web_1 ...
Creating composetest_redis_1 ...
Creating composetest_web_1
Creating composetest_redis_1 ... done
```
The testing landscape

- Containers on Kubernetes
  - DSL for deployment and configuration, not for multi-stage experiments
  - Portability of the test
  - Low maintenance cost
  - Low execution overhead
  - Multi-node testing environment
The testing landscape

- Containers on Kubernetes with CI/CD pipelines
  - Rich and expressive DSL for writing multi-stage scenarios
  - Portability of the test
  - Low maintenance cost
  - Low execution overhead
  - Multi-node testing environment
  - Still requires human effort to analyze the results
The Frisbee Testbed

Frisbee is a Kubernetes extension that provides:

- ✓ Rich and expressive DSL for writing complex testing scenarios
- ✓ Portability of the test
- ✓ Low maintenance cost
- ✓ Low execution overhead
- ✓ Multi-node testing environment
- ✓ Programmatically assertable conditions
A “Hello, Network!” Frisbee test

spec:
actions:

1. # Create an iperf server
   action: Service
   name: server
   service:
     templateRef: iperf.server

2. # Create a cluster of iperf clients
   action: Cluster
   name: clients
   depends: { running: [ server ] }
   assert:
     state: '{{.state.NumOfFailures()}} >= 1'
     metrics: 'avg() of query(metric, 5m, now) is below(1000)'
   cluster:
     templateRef: iperf.client
     instances: 30
     inputs:
       - { server: .service.server.one, seconds: "600" }
       - { server: .service.server.one, seconds: "30" }
     schedule:
       cron: "@every 1m"

Scenario

1. Instantiate a Service
2. Based on the given template
3. Then instantiate a cluster of services
4. When certain conditions are met.
5. Abort on failures or SLA violations
6. Within the cluster, create 30 services.
7. Iterating over the given inputs.
8. Inputs may use addressing macros
9. Schedule 1 service every 1 minute.
A “Hello, Network!” Frisbee test

spec:
actions:
  # Create an iperf server
  action: Service
  name: server
  service:
    templateRef: iperf.server

  # Create a cluster of iperf clients
  action: Cluster
  name: clients
  depends: { running: [ server ] }
  assert:
    state: '{$.state.NumOfFailures() >= 1}'
    metrics: 'avg() of query(metric, 5m, now) is below(1000)'
  cluster:
    templateRef: iperf.client
    instances: 30
  inputs:
    - { server: .service.server.one, seconds: "600" }
    - { server: .service.server.one, seconds: "30" }
  schedule:
    cron: "@every 1m"

---

**Scenario**

1. Instantiate a Service
2. Join Server
3. Join Client-0
4. Join Client-1
5. Exit
6. Join
7. Iterating over the given inputs.
8. Inputs may use addressing macros
9. Schedule 1 service every 1 minute.
A “Bye Bye, Network!” Frisbee test

spec:
  actions:
    # Create an iperf server
    - action: Service
      name: server
    
    # Create a cluster of iperf clients
    - action: Cluster
      name: clients
      depends: 
        - running: [server]
      assert:
        state: '{{.state.NumOfFailures()}} >= 1'
      metrics:
        'avg() of query(metric, 5m, now) is below(1000)'
      cluster:
        templateRef: iperf.client
      instances: 30
      inputs:
        - server: .service.server.one, seconds: "600"
        - server: .service.server.one, seconds: "30"
    
    # After a while, inject a network partition
    - action: Chaos
      name: partition
      depends: 
        - running: [server]
      chaos:
        templateRef: chaos.network.partition
      inputs:
        - server: .service.server.one, duration: "2m"

1. Abstract failures as Chaos Jobs
2. Enable execution-driven fault injection
Performance

Frisbee

Real-time Clients (Total Ops)

Total Operations (by Client)

Total Throughput

Max Latency

FOSDEM / D.Testing 2022
Scalability

Frisbee
Elasticity

Total Operations (by Client)

Total Throughput

Max Latency

Frisbee
Availability

Frisbee

![Graphs showing availability and performance metrics for Frisbee.](image)
Metrics-Driven assertions check whether the system operates within expected limits.

**Prometheus / Grafana**

**Real-Time Dashboards**

**Kubernetes API**

**Service**

**Pod**

- App Container
- Telemetry Agent
- Chaos Agent

**Webhook**

**Alerts**

metrics: 

\[
\text{avg()} \text{ of query(metric, 5m, now)} \text{ is below(1000)}
\]
State-Driven assertions check whether the system transits into erroneous states.
Spec:

```
actions:

# Create an iperf server
- action: Service
  name: server
  service:
    templateRef: iperf.server

# Create a cluster of iperf clients
- action: Cluster
  name: clients
  depends: [ running: [ server ] ]
  assert:
    state: '{{.state.NumOfFailures()}} >= 1'
    metrics: 'avg() of query(metric, 5m, now) is below(1000)'
  cluster:
    templateRef: iperf.client
    instances: 30
    inputs:
      - { server: .service.server.one, seconds: "600" }
      - { server: .service.server.one, seconds: "30" }
    schedule:
      cron: "@every 1m"

# After a while, inject a network partition
- action: Chaos
  name: partition
  depends: [ running: [ server ] ]
  chaos:
    templateRef: chaos.network.partition
    inputs:
      - server: .service.server.one
      - duration: "2m"
```

Scenario:
```
netfailure.yaml
```

Template:
```
spec:
  inputs:
    parameters:
      server: localhost
      seconds: "60"
    service:
      decorators:
        Telemetry:
          - platform.telemetry.container
          - iperfmon.client
        containers:
          - name: app
            image: someimage
            Command:...
            # ... blah blah

  server={{"{{.Inputs.Parameters.server}}"}}
  seconds={{"{{.Inputs.Parameters.seconds}}"}}

iperf3 -c $server -t $seconds
```

Exposed:
```
server={"{{.Inputs.Parameters.server}}"}
seconds={"{{.Inputs.Parameters.seconds}}"}
iperf3 -c $server -t $seconds
```

Hidden:
```
server={"{{.Inputs.Parameters.server}}"}
seconds={"{{.Inputs.Parameters.seconds}}"}
iperf3 -c $server -t $seconds
```
**A Frisbee Template**

```
spec:
  inputs:
  parameters:
    server: localhost
    seconds: "60"
  service:
  decorators:
    telemetry: [platform.telemetry.container, iperfmon.client]
  requirements:
    persistentVolumeClaim:
      name: datastore
      spec: ...
  containers:
    - name: app
      image: someimage
      volumeMounts:
        - name: datastore
          mountPath: /store
      Command: ...
      # ... blah blah

server={{"{{.Inputs.Parameters.server}}"}}
seconds={{"{{.Inputs.Parameters.seconds}}"}}

iperf3 -c ${server} -t ${seconds}
```

---

**Functionality**

1. Declare parameters
2. Re-use other templates
3. Deployment requirements
4. Automation are for free
5. Use inputs to manipulate the container
Project Structure

Frisbee

Library of testing components

Components packaged via HELM

Visualizations & Alerts
Usage
Test Factory
How to use Frisbee

**TL;DR**

1. Make sure that `kubectl` and `Helm` are installed on your system.
2. Update Helm repo.

   ```
   >> helm repo add frisbee https://carv.ics-forth.github.io/frisbee/charts
   ```
3. Install Helm Packages.

   ```
   # Install the platform
   >> helm upgrade --install --wait my-frisbee frisbee/platform
   # Install the package for monitoring YCSB output
   >> helm upgrade --install --wait my-ycsb frisbee/ycsb
   # Install TiKV store
   >> helm upgrade --install --wait my-tikv frisbee/tikv
   ```
4. Create/Destroy the test plan.

   ```
   # Create
   # Destroy
   ```

**Quick Tutorial**

- Charts are self-descriptive
- With dependencies
- And usage examples
Summary

- Frisbee: A platform for Kubernetes-native Testing
  - ✓ Multi-node testing environment
  - ✓ Similar environment for dev, test, and production
  - ✓ Controllers run within Kubernetes cluster. Batteries-includes
  - ✓ Experiments written in YAML -> Write once / Run anywhere.
  - ✓ System Spinup -> Testing Actions -> System Validation
Looking for Collaborators

**Frisbee**

### Devops
- Testing workflows
- Systems for testing

### Developers
- Kubernetes Controllers
- Testing Resources

### Researchers
- Many ideas floating around

https://github.com/CARV-ICS-FORTH/frisbee
fnikol@ics.forth.gr
THANKS

Do you have any questions?

fnikol@ics.forth.gr

FORTH, Crete, Greece

Acknowledgement:
This work is supported by the European Commission within the scope of:

ETHER (H2020-MSCA-IF-2019)
Grant Agreement ID: 894204