Event-driven Infrastructure

Mike Place
mp@saltstack.com
https://mikeplace.io

SaltStack

Part 0: A Thank You

Principles > Products

Part I: The Future is a Distributed System With a Convergence Problems

How do we reason about complexity?

One way we reason about it is through patterns

Active humans building proactive processes create reactive systems

We’re very good at building systems that say:
If x happens, then do y

But we’re bad at building systems that say:

If NOT x, then y

Part II: New Ideas are Hard to Come By

Three Pillars of Distributed Computing

- The ability for the system to be aware of the existence and capability of its member nodes.
- The ability to co-ordinate tasks between those nodes.
- Inter-process communication which can connect nodes in the system to each other

The fundamental unit of distributed computing is the *event*

What are the properties of a message bus?

- Data model
- Command set
- Transport

Message Buses for Ops

- Monitoring
- Configuration management
- ChatOps
- Auto-scale
- Serverless
- Provisioning

Message Buses for Dev

- Almost anything that connects various layers of an application stack has a message bus of some kind
- Sometimes these are streaming
- Sometimes they’re just set up and torn down on demand.

Part IV:

Question:

What possibilities emerge when these siloed streams of events are shared with each other?

What does (most) automation look like right now?

- Packaged workflows
  - We take a series of steps (or a collection of structured data) and we list them out and then we go and do those steps, with, hopefully a little bit of abstraction and error control.
  - Much of the time, these workflows are initiated by lazy humans.
  - Despite our best-efforts, these can still be very brittle because one thing we're not very good at is understanding the state of a system before automation runs on it. We make a lot of assumptions, even today.

This doesn’t feel much like programming.

(Whether you think this is awesome says a lot about you.)

(And about us.)

Part V: Event Driven Programming

Event-driven Programming

- A programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads.
- Event-driven programming is the dominant paradigm used in graphical user interfaces and other applications that are centered on performing certain actions in response to user input.

Examples of Event Driven Programming

- JavaScript
  - node.js
- Most GUI applications on the desktop
- Pretty much any user interface on iOS

3 Principles of Event Driven Programming

- A set of functions which handle events.
- A mechanism for binding the registered functions to events.
- A loop which constantly polls for new events and calls the matching event handler(s) when a registered event is received.

Traditional Criticisms of Event-Driven Programming

- One one hand, there is imperative programming
  - Writing procedures to performs steps in a particular order.
- On the other hand, we have declarative programming
  - Describing the intended state of a system without explicitly describing the steps to achieve that state.

Criticisms continued

- Highly asynchronous code can be difficult to troubleshoot
- It takes a mindshift to think about imperative and declarative approaches melded into one. This can create some confusion.
- It can be challenging to translate procedural workflows into something event-driven.

Event-driven Programming Advantages

- It’s easy to find natural dividing lines for unit testing infrastructure.
- It’s highly composeable.
- It allows for a very simple and understandable model for both sides of the DevOps bridge.
- Both purely procedural and purely imperative approaches get brittle as they grow in length and complexity.
- It’s one good way to model systems that need to be both asynchronous and reactive.

High-speed event bus

+

Event-driven programming

==

Event-driven infrastructure

The e
Mess
-

Reactive Manifesto

The e
Mess
-

Principles of Event-Driven Automation

- Events originate from applications and from systems and are transmitted across a bus.
- Mechanisms exist to sort these events and apply rules to them.
- As rules are matched, registered actions occur — promises are met.

Disadvantages of an Event-Driven Approach

- Possible tight coupling between the event schema and the consumers of the schema.
- Message loss
- Reasoning about “blocking” operations might becoming more difficult.
- Testing

Advantages of Event-Driven Approach

- Distributed, scalable and loosely coupled between systems.
- A “DevOps” automation backplane for system
- Does more than just configure/provision systems at their birth. Allows for more complete lifecycle management.
- Provides an immediate, common programmable layer on top of existing automation/deployment systems.

Part VI: Fine then. How do we build one?

(But first, a manifesto)

Flow

- An event is emitted on the event bus
- It flows to a manager
- The manager checks to see if the event matches a registered handler
- If so, the a series of rules are checked
- For each set of rules which are matched, an action is undertaken.

The moving pieces

- Event bus transport
- Telemetry
- Actors
- Reactors

Building Event Buses

Concerns for the bus

- MUST handle
  - Security
  - Reliability
  - Serialization
- MAY handle
  - An easy set of interfaces for send/receive, with libraries for languages shared by Dev and Ops
  - Multiple architecture patterns
  - Message filtering
  - Message routing

Message bus topology

- Pub/sub
  - 1-M
  - Most implementations are brokered, which means they are loosely coupled. (i.e. the sender does not know or care) about the status of the recipient.
  - There are un-brokered implementations such as Data Distribution Service (over IP multicast) which exist, but they are (for the most part) not widely deployed.
- Push/pull
  - Is client/server but typically is a 1-1 relationship instead of 1-M
- Fan-out
  - Useful for creating a queue of workers through

Off the shelf messaging

- ZeroMQ
- RabbitMQ
- Celery
- ActiveMQ
- JBoss messaging
- Apache Qpid
- StormMQ
- Apache Kafka
- Lambda
- Redis
- SaltStack

Telemetry

- The ability for applications to emit events onto the bus.
- Should be light and easy enough that it’s simple to port into any language.
- Should be lightweight messaging. Not the place for pushing enormous amounts of data around.

```
#/usr/bin/env python
import zmq
import os
import socket
import time
import core.framer

# Create a context
ctx = zmq.Context()

# Our tag
tag = '/client/load/silver'
while True:
    # Frame it up. (Uses msgpack)
    event = framer.pack(tag, {'cur_load': os.getloadavg()})
    socket = ctx.socket(zmq.PUSH)
    socket.connect('tcp://localhost:2001')
    socket.send(event)
    socket.close()
    time.sleep(1)
```

Building Reactors

Decision Engines

- A decision engine registers events which occur on the bus to actions which need to be performed.
- Can be as simple or as complex as one wants.
- The DevOps idea, here, is though to create a shared abstraction for these rules.

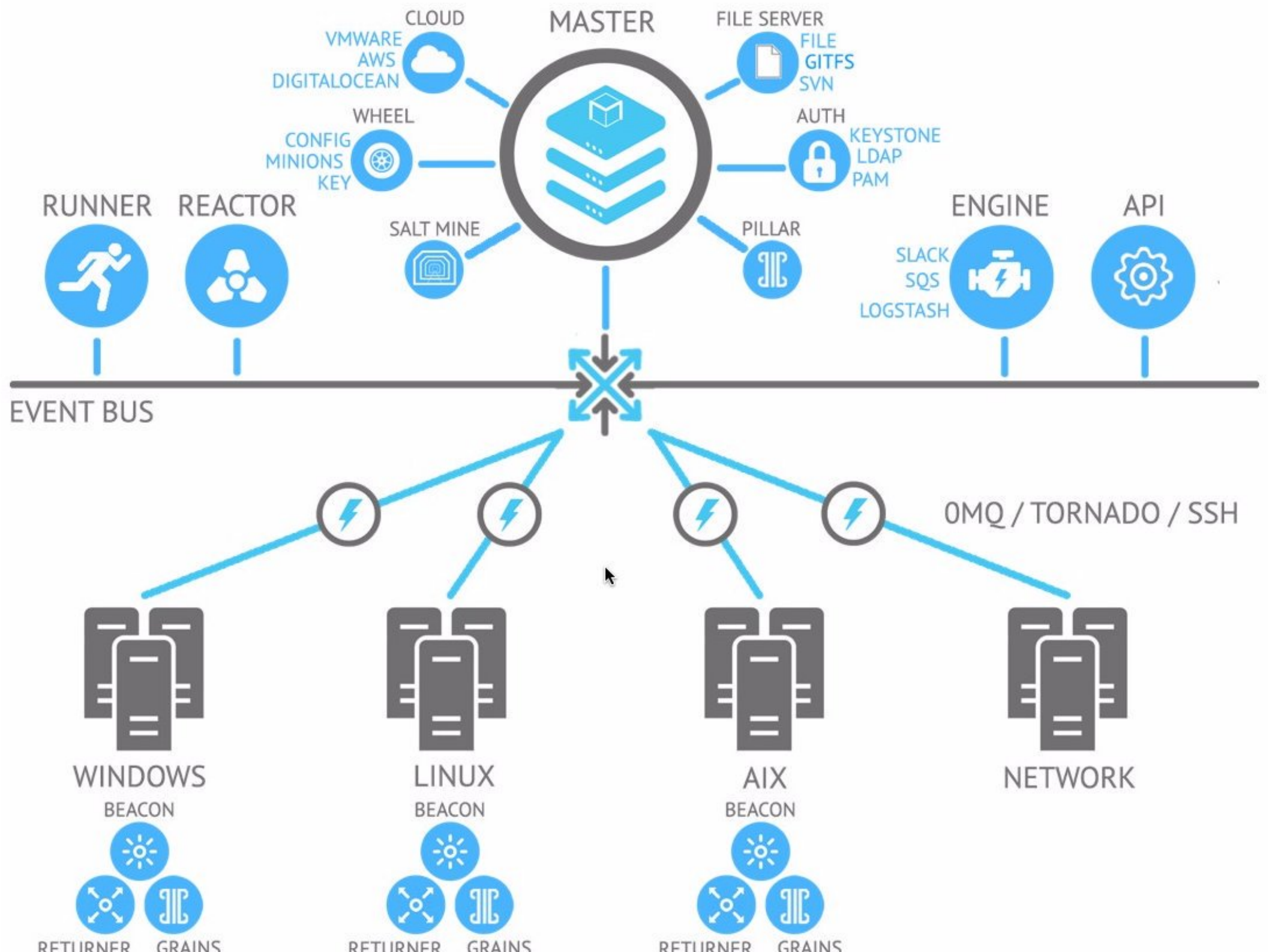
```
# This configuration maps simple event tags to actions
#


# If we receive a high load alert, print an alert and reconfig a LB


/client/load/*:
    reactions:
        - reactions.printer.event_printer
    register:
        - register.aggregate.register
    rules:
        - rules.simple.gt:
            register: register.aggregate.avg
            threshold: 20
            period: 10
            reactions:
                - reactions.eventer.fire_event:
                    tag: '/reconfig/lb'
                    data:
                        some_reconfig_data: 'dummy_data'
```

Actors

- Actors are simply what is run as a result of an event matching a given rule.
- Possibilities
  - Call to external service
  - Configuration management call
  - Code running locally

Part VII: A Pillar of Salt

Configuration Management is a Service


CLOUD
VMWARE
AWS
DIGITALOCEAN
MASTER
FILE SERVER
FILE
GITFS
SVN
WHEEL
CONFIG
MINIONS
KEY
AUTH
KEYSTONE
LDAP
PAM
RUNNER
REACTOR
SALT MINE
PILLAR
ENGINE
SLACK
SQS
LOGSTASH
API
EVENT BUS
0MQ / TORNADO / SSH
WINDOWS
LINUX
AIX
NETWORK
BEACON
BEACON
BEACON
RETURNER
GRAINS
RETURNER
GRAINS
RETURNER
GRAINS

Engine

- An engine is just a long-running pure-Python process in Salt with access to the Salt functionality injected into its namespace.

Beacon

- A beacon is a small Python function that runs in a loop, looking for a state change and broadcasts that change to a Salt master.
- Operational changes ex.: File changes, network events, interface changes, container CRUDs, database inserts.

Reactor

- Events flow from machines under management (minions) to central servers (masters) via an event bus.
- On the master, a reactor listens to those events and executes appropriate jobs on matches.

A Networking Example

Translating message events w/ engines

```
<28>Jul 20 21:45:59 edge01.bjm01 mib2d[2424]: SNMP_TRAP_LINK_DOWN: ifIndex 502,
ifAdminStatus down(2), ifOperStatus down(2), ifName xe-0/0/0
```


```
napalm/syslog/iosxr/INTERFACE_DOWN/gw2.acy1 {
  "error": "INTERFACE_DOWN",
  "facility": 23,
  "host": "gw2.acy1",
  "ip": "5.6.7.8",
  "os": "iosxr",
  "severity": 3,
  "timestamp": 1499458574,
  "yang_message": {
      "interfaces": {
          "interface": {
              "TenGigE0/2/0/4": {
                  "state": {
                      "oper_status": "DOWN"
                  }
              }
          }
      }
  },
  "yang_model": "openconfig-interfaces"
}
```

Reactor

```
reactor:
  - 'napalm/syslog/*/INTERFACE_DOWN/*':
    - salt://reactor/if_down_shutdown.sls
    - salt://reactor/if_down_send_mail.sls
```

State files

```
shutdown_interface:
  local.net.load_template:
    - tgt: gw2.acy1
    - kwarg:
        template_name: salt://templates/shut_interface.jinja
        interface_name: TenGigE0/2/0/4
```

```
{%- if grains.os == 'iosxr' %}
interface {{ interface_name }}
  shutdown
{%- elif grains.os == 'junos' %}
deactivate interface {{ interface_name }};
{%- endif %}
```

State files (cont)

```
{%- set if_name = data.yang_message.interfaces.interface.keys()[0] %}

send_email:
  local.smtp.send_msg:
    - tgt: {{ data.host }}
    - arg:
        - mircea@example.com
        - Interface down notification email body.
    - kwarg:
        subject: Interface {{ if_name }} is down
```

https://github.com/cachedout/eventdriventalk

https://mirceaulinic.net/2017-10-19-event-driven-network-automation/

Review

- To build scalable systems, we need to adopt the lessons of distributed computing
- We can migrate from simple human-initiated workflows to reactive, programmable systems.
- Event buses are pretty good. Let’s build more of those.
- Build on top of tools (like Salt!) that can give us a full automation toolkit up and down the stack.

Questions?

Mike Place
@cachedout
mp@saltstack.com