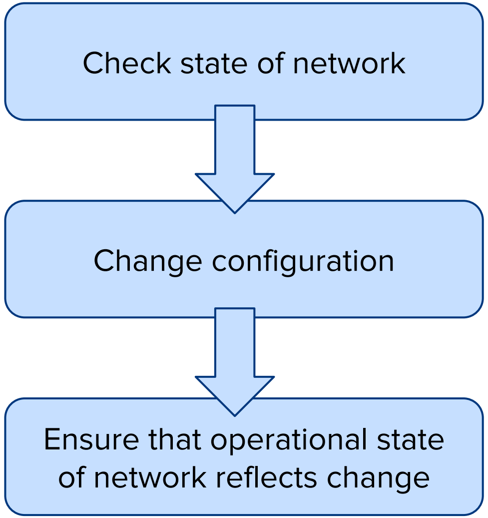
# OpenConfig OpState

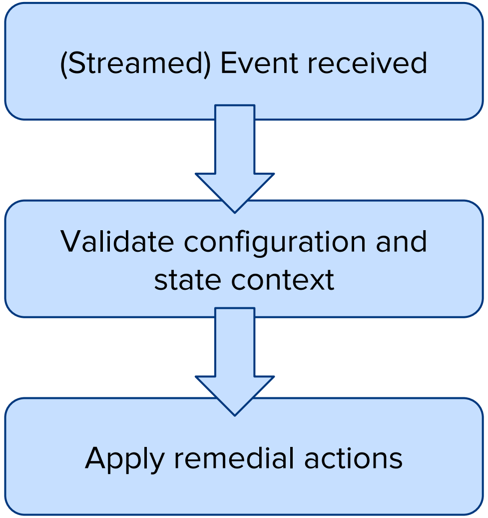
Rob Shakir, Jive Communications (rjs@jive.com)
Anees Shaikh, Google (aashaikh@google.com)
Marcus Hines, Google (hines@google.com)

# Some history...

- November 2014 - need a solution for storing both operational state and configuration parameters in YANG.

- Two types of event flows that need simple correlation between state and configuration data:



# Overview

```
       +---------+
       |         |    transition intended
       |intended |    to applied
       | config  +---------+
       |         |         |
       +---------+         |
           ^               |         config: true
+----------|------------------------------------+
           |               |         config: false
           |               |
           |               |
           |       +-----------------------------+
           |       |       | operational state   |
           |       |  +----v----+ +-----------+  |
           |       |  |         | |           |  |
           +       |  | applied | |  derived   |  |   operational:true
         same +------>| config  | |   state    |<-------+
         leaves    |  |         | |           |  |
                   |  |         | |           |  |
                   |  +---------+ +-----------+  |
                   +-----------------------------+
```

# Solution proposed...

- `container`
  - `config`
    - `<configuration-parameters>` ***(Intended config)***
  - `state`
    - `<reflection-of-configuration-parameters>` ***(Applied config)***
    - `<counters-statistics-protocol-parameters>` ***(Derived state)***

- Operational experiences with this approach
  - Refactored a number of models to reflect this approach (BGP, MPLS) -- a bit more effort
  - Removes the need for mapping dictionaries for config -> state
    - ‘router bgp X, neighbor Y remote-as Z’ -> OID x.y.z....a.b.c....0
    - ‘router bgp X, neighbor Y remote-as Z’ -> ‘show ip bgp neigh Y | i Remote AS’
  - Expressible in YANG today - extensions required are for query efficiency rather than core operation.
  - Code to consume these models and perform inter-relation written in multiple operators.

# Support for opstate-requirements (I).

1. Interact with intended and applied configuration
   a. Possible to get only applied by filtering on `operational: false` elements of `state`.
   b. “Mirrored” config leaves in `state` represent applied configuration as read-only.
   c. Leaves in `config` correspond 1:1 with leaves in `state`.
   d. A server only updates the `state` leaf when the value in the `config` leaf has been applied.

2. Applied config and derived state can be retrieved by retrieving the `state` paths.

3. b) Simple to determine differences without additional operation to validate - one can simply get both `config` and `state` containers and ‘diff’ the values. Does not need server changes.

4. 
   a. Derived state retrieved by filtering on `operational: true` in state.
   b. Applied config retrieved by filtering on `operational: false` (default) in state.
   c. Applied config and derived state can be retrieved by retrieving the `state` path.

# Support for opstate-requirements (II).

5. Derived state can be retrieved by filtering by retrieving `operational: true` nodes.

6. 
    a. Intended config can be simply mapped to applied state by relating the `config` and `state` leaves
    b. Intended config (`config`) leaves can be related to the `state` container in the same path.
    c. Structure means that simply parsing the model allows mapping (no mapping table or other annotations required)

7. N/A.

# Plan going forward...

- Continue to write code implementing OpenConfig models - which implement the opstate solution described.
- Use this learning to iterate/determine the next steps.