The Evolution of Network Configuration Management

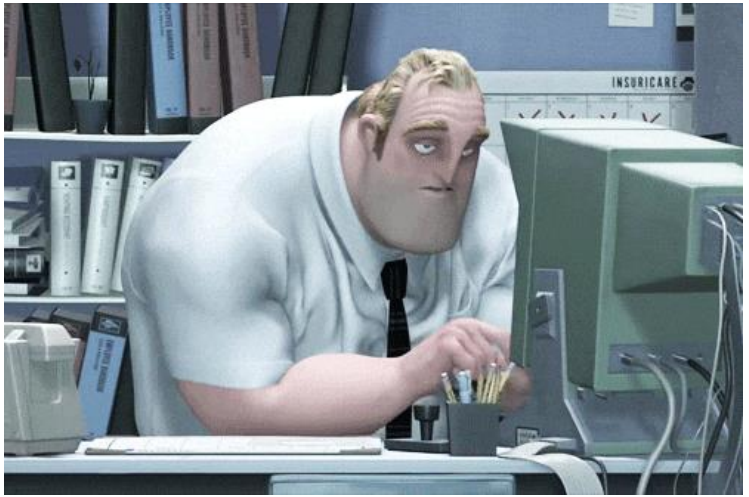
Tim Raphael

Regional Product Manager – IP Automation

In the beginning…

binaries, buffers and text files

- Configuration typed directly into command buffers via serial
- Limited filesystems – config stored in NVRAM
- *nix boxes using CPU processing stored config in text files, needed restarting / reloading to apply changes.



RFC789*

July 1981

Vulnerabilities of Network Control Protocols: An Example

Eric C. Rosen

Bolt Beranek and Newman Inc.

This paper has appeared in the January 1981 edition of the SIGSOFT Software Engineering Notes, and will soon appear in the SIGCOMM Computer Communications Review. It is being circulated as an RFC because it is thought that it may be of interest to a wider audience, particularly to the internet community. It is a case study of a particular kind of problem that can arise in large distributed systems, and of the approach used in the ARPANET to deal with one such problem.

On October 27, 1980, there was an unusual occurrence on the ARPANET. For a period of several hours, the network appeared to be unusable, due to what was later diagnosed as a high priority software process running out of control. Network-wide

Management Protocols

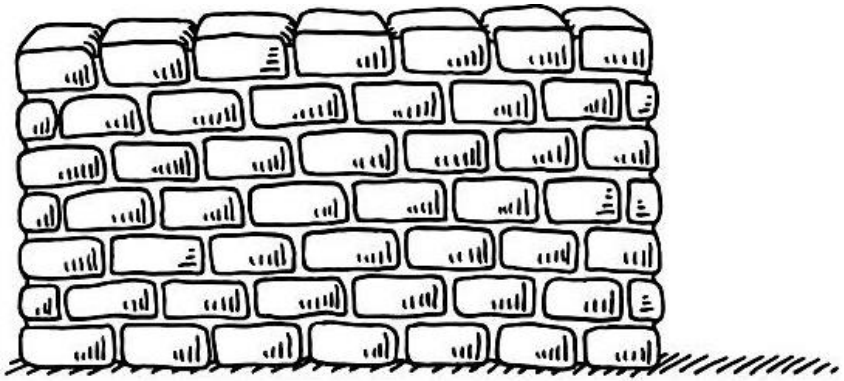
management protocols

Simple Network Management Protocol started out with a few earlier variants:

- SGMP – Simple Gateway Monitoring Protocol (RFC1028, 1987)
- HEMP - High-level Entity Management Protocol (RFC1021)
- CMIP – Common Management Information Protocol (ISO)

Same basic idea

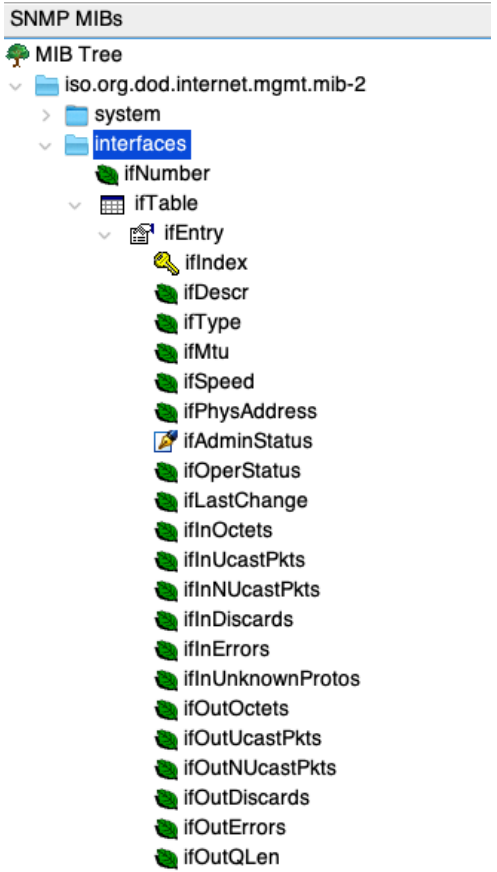
- Get – query information
- Set – update information
- Notify/Trap – device-originated, proactive query response



management protocols

Shared Management Information Base that structures how information should be queried.

- Early example of an information model
- Shared between device and management system / client.
- Attempts to standardise across vendors.
- Included a descriptor language for units, but still had variations between platforms.
- Relatively slow, CPU intensive and had limited ability for bulk changes - often requiring multiple RTTs.



Era of the CLI

command-line interfaces

- Given early devices were just computers with interfaces*, it's no surprise network operating systems just used UNIX fundamentals.
- Some vendors built their user interfaces on top of SNMP (and it's early variants) while others looked for better ways…
- Cisco originally relied on TFTP to pull configuration and load an on-device configuration buffer.
  - Until 1992, they released the first version of a configuration-specific CLI where each line was parsed before being applied.

```
[/]
A:admin@PE-6# configure {
    service {
        pw-template "PW2-template" {
            pw-template-id 2
            hash-label {
            }
            split-horizon-group {
                name "SHG"
            }
        }
        vpls "VPLS-2" {
            admin-state enable
            service-id 2
            customer "1"
            bgp 1 {
                pw-template-binding "PW2-V1" {
                }
            }
            bgp-ad {
                admin-state enable
                vpls-id "64496:2"
            }
            sap 1/1/4:2 {
            }
        }
    }
}
```

automating command-line interfaces

- Operators would copy and paste changes, prepared in text files into the device at the time of change.
- Some basic, manual templating was common but due to a lack of consistent service offerings, variations in interfaces etc. a lot of this was manually managed.
- TCL (Tool Command Language) was originally created in 1988 as a dynamic, string-typed language that casts everything into the mold of a command.
- Expect extension to TCL in 1990 to specifically handle programs that expose a text interface.
  - → Advent of Expect-driven configuration automation.

```
#!/usr/bin/env expect

# Common device prompts: '>', '#', or '$' (expand as needed)
set prompt_re {[#>\$]\s?$}

# Spawn SSH (adjust options for your environment)
spawn ssh -o StrictHostKeyChecking=accept-new -o UserKnownHostsFile=/dev/null \
          -o KbdInteractiveAuthentication=no \
          -l $user $host

# Handle first-connection, password, MFA/noise, and get to a prompt
expect {
    -re "yes/no" {
        send -- "yes\r"
        exp_continue
    }
    -re "(P|p)assword:" {
        send -- "$pass\r"
        exp_continue
    }
    -re $prompt_re {
        # At device prompt
    }
    timeout {
        puts "ERROR: Timeout waiting for SSH banner/login."
        exit 1
    }
    eof {
        puts "ERROR: SSH session closed unexpectedly."
        exit 1
    }
}
```

APIs

NETCONF (2003)

- Standardisation of ideas from other XML-RPC APIs.
- <RPC> invocations
- <RPC-REPLY> responses
- <NOTIFICATION> updates / events
- Linked by message-id attribute
- NETCONF over SSH
- NETCONF over TLS with Mutual X.509 Authentication
- JSON encoding added later
  - → improved parsing performance

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<edit-config>
    <target><candidate/></target>
    <config>
        <configure xmlns="urn:nokia.com:sros:ns:yang:sr:conf">
            <service>
                <epipe>
                    <service-name>CustDoc</service-name>
                    <customer>1</customer>
                    <description>Local epipe</description>
                </epipe>
            </service>
        </configure>
    </config>
</edit-config>
</rpc>
```

YANG (2010)

- Standardised, human-readable modelling language
- Model a structure of:
  - module → container(s) →list(s) → leaf(s)
  - Build logical structures for describing data
- Standardised models (e.g. OpenConfig)
  - Enables a standard way for configuration and state data to be exchanged between devices and management platforms.
- Protocol-independence means it can be encoded into XML, JSON or YAML (or a future format).
- From definitions, bindings can be generated → enabling easier integration with various languages and platforms.

```
module: srl_nokia-acl
  +--rw acl
     +--rw acl-filter* [name type]
        |  +--rw name                       srl_nokia-comm:name
        |  +--rw type                       enumeration
        |  +--rw description?               srl_nokia-comm:description
        |  +--rw subinterface-specific?     enumeration {not srl_nokia-feat:fpcx}?
        |  +--rw statistics-per-entry?      boolean {not srl_nokia-feat:fpcx}?
        |  +--ro last-clear?                srl_nokia-comm:date-and-time-delta
        |  +--rw entry* [sequence-id]
        |     +--rw sequence-id      uint32
        |     +--rw description?     srl_nokia-comm:description
        |     +--ro last-clear?      srl_nokia-comm:date-and-time-delta
        |     +--rw match
        |        +--rw l2 {not srl_nokia-feat:fpcx}?
        |        |  +--rw destination-mac
        |        |  |  +--rw address?   srl_nokia-comm:mac-address
        |        |  |  +--rw mask?      srl_nokia-comm:mac-address
        |        |  +--rw ethertype?          srl_nokia-pkt-match-types:ethertype
        |        |  +--rw source-mac
        |        |  |  +--rw address?   srl_nokia-comm:mac-address
        |        |  |  +--rw mask?      srl_nokia-comm:mac-address
        |        |  +--rw vlan
        |        |     +--rw outermost-vlan-id
        |        |        +--rw (vlan-matching-type)?
        |        |           +--:(range-with-one-value)
        |        |           |  +--rw operator?   srl_nokia-pkt-match-types:operator
        |        |           |  +--rw value?      srl_nokia-pkt-match-types:vlan-id-type
        |        |           +--:(range-with-two-values)
        |        |           |  +--rw range
        |        |           |     +--rw start?   srl_nokia-pkt-match-types:vlan-id-type
        |        |           |     +--rw end?     srl_nokia-pkt-match-types:vlan-id-type
        |        |           +--:(special-value-none-to-match-untagged)
        |        |              +--rw none?       empty
```

```
myAcl = YangModels.ACL()

myAcl.acl-filter["customFilter"].entries[0] = {
    name: "BGP_INPUT",
    type: "IPv4"
    description: "Allow BGP"
    ...
}
```

gRPC

gRPC

- Open sourced in 2015 – contributed to CNCF.
- Uses Protocol Buffers (protobufs) as the description language for request and reply payload formats- defining the data structures used.
- Proto files (*.proto) gRPC Services come in multiple formats: Unirary, Server-streaming, Client-streaming, Bidirectional-streaming.
- Uses HTTP/2 with optional TLS.
- gRPC interfaces for common network operations:
  - gNMI – configuration and state streaming.
  - gNOI – operational commands for common actions.
  - gRIBI – forwarding table manipulation.
  - gNSI – security-focused commands.

```
package tutorial;

message Person {
  string name = 1;
  int32 id = 2;
  string email = 3;

  enum PhoneType {
    PHONE_TYPE_UNSPECIFIED = 0;
    PHONE_TYPE_MOBILE = 1;
    PHONE_TYPE_HOME = 2;
    PHONE_TYPE_WORK = 3;
  }

  message PhoneNumber {
    string number = 1;
    PhoneType type = 2 [default = PHONE_TYPE_HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

OpenConfig

- Group of technical contributors from major operators and vendors **defining interfaces and tooling for managing networks**.
- **YANG Models**: operationally complete, vendor-neutral YANG models that provide consistent and cohesive descriptions for configuration and state.

gNMI

- Configuration + state manipulation via various vendor and OC YANG models.
- Methods:
  - → Capbility discovery
  - → Get a snapshot of state information (configuration and status)
  - → Modify state information (configuration)
  - → Subscribe to telemetry updates
    - Once
    - Stream – interval or ON_CHANGE
    - Poll – interval

gNOI

- RPCs for doing common actions against devices
- File Service: Get / Put / Stat / Remove
- System Service: Ping / Traceroute / Reboot / Time / SwitchControlProcessor
- OS Service: Install / Activate / Verify
- Factory Reset Service: Start
- Healthz Service: Get / Check / List / Acknowledge RPCs for chassis components
- Packet Link Qual Service: Perform RPCs to validate the link between two devices

gRIBI

- RPCs for forwarding table manipulation:
- FIB Manipulations:
  - → GET – retrieve the state of a set of FIB entries.
  - → Modify – add/modify/remove FIB entries.
  - → Flush – remove all gRIBI-related FIB entries.

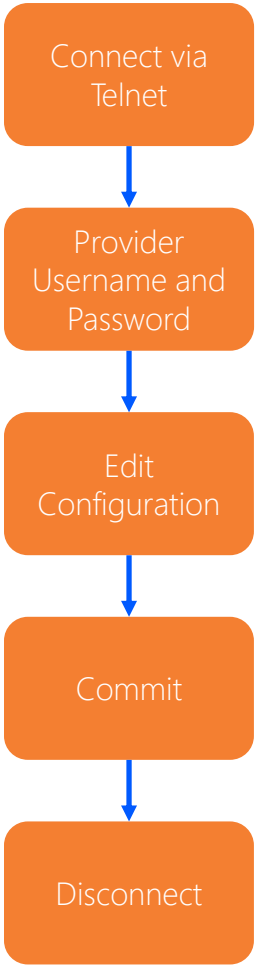
gNSI

- RPCs for managing security-related items of devices.
- **Authz Service:** Get / Probe / Rotate authorisation policies
- **Certz Service:** Get / Delete / AddProfile / GenerateCSR / Rotate certificates

So where is this all going?

imperative

- We started with very imperative ideas: “Make this change by setting abc to xyz”
- **Imperative** focuses on the **how** you get to the result.
- Modelled on the **user experience** of doing configuration and operational tasks **manually**.
- This is where Expect extensions to TCL came from.



Make me a sandwich…

declarative

"Upgrade this node to v1.2.3 at 23:45 tomorrow."

- Improved processing power on devices has allowed us to develop better APIs using modern technologies like gRPC.
- This has enabled a more declarative approach where we tell a device what we want, not how to get there.
- This is reflected in configuration management platforms like Puppet, Salt and Terraform.
- The developer defines the desired end state of the system and the internals of the platform "make it so".

declarative

"Upgrade this node to v1.2.3 at 23:45 tomorrow."

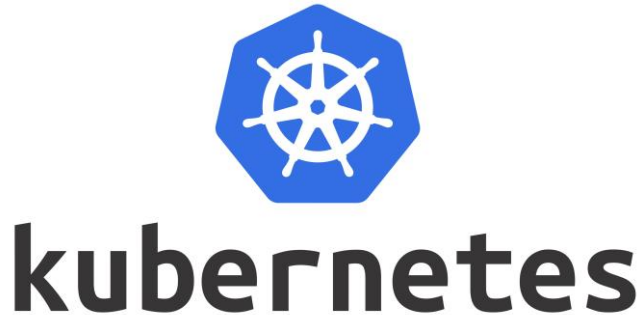
- Network architectures and technologies have increased in complexity – what used to be a simple config can now be thousands of lines.
- Treating the network like a programmable plane enables us to encapsulate complexity with **abstraction**.
- We can build constructs that represent complex network services and configurations… we call them… **intents**

Is this new?

… not really.

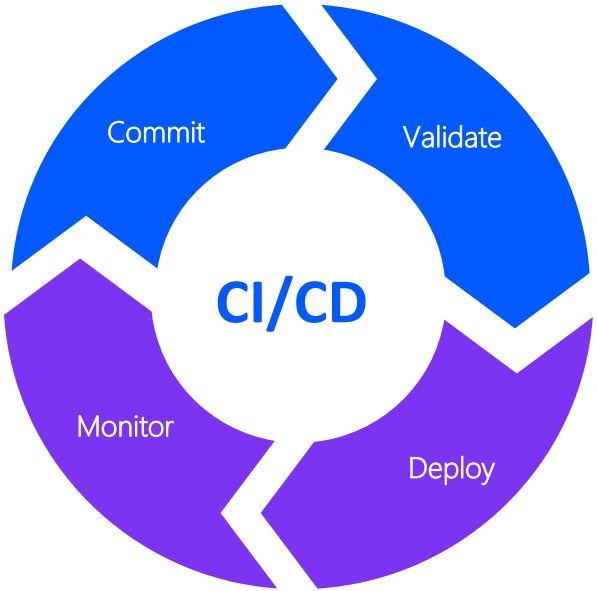
declarative configuration for networks

- K8s Operator Pattern
  - Common pattern used in infrastructure
  - Allows the creation of custom resources “owned” by a controller. The controller ensures that resources exist and are correct.
  - Why not define CRDs for network abstractions?
- Eventual Consistency is bad for networks.
  - → Need for transactional, atomic operations
  - → Need to know the state at any given moment



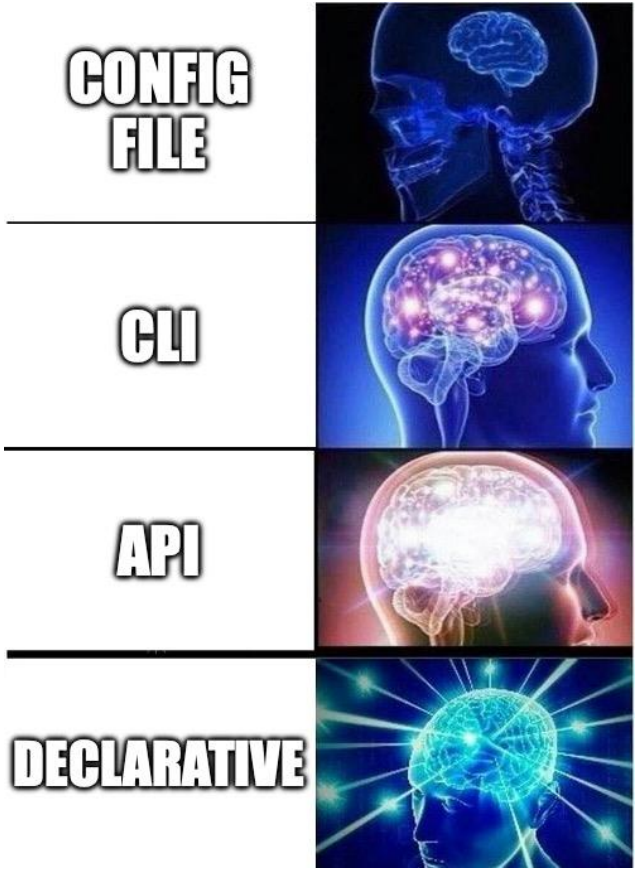
benefits

- Infrastructure as Code
  - Desired state is always known and stored in a way that is easy to re-create in the network.
  - Version-controlled changes with git-revert rollback.
  - A huge list of existing tooling designed for IaC processes.
- State information linked to resources
  - A network resource could bubble the state linked to that resource.
  - E.g. interface counter → service counter → Pretty Dashboards with little processing.



what does this mean for operators?

- We're not CLI jockeys anymore – we become declarative programmers.
  - This isn't too different to what we're used to doing – writing config to "make it so", we're now just using higher-level means.
- Changes are more reliable
  - Because of the state enforcement, any variation is known ahead of the change and can be skipped or mitigated.
- Operations at scale, efficiently
  - There's a good reason the hyper-scalers are using this sort of approach with magical, custom controllers – they can achieve big things with few operations due to the predictability.



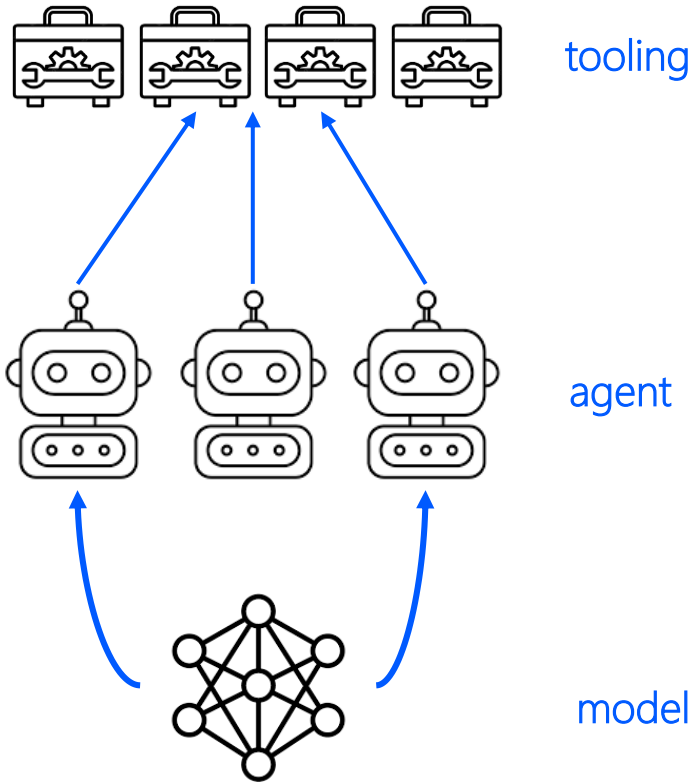
But what about…

AI

pragmatic use of AI agents

- Tooling for real-world use
  - Declarative interfaces define a clear promise of operation.
  - Build tooling for use-cases that suit the capability of LLMs:
    - Summarisation
    - Root cause analysis
    - Code/configuration generation

- Example use-cases:
  - Flood of alarms → "What just happened?" → Pull state from the network to correlate events.
  - Best-practise driven configuration suggestion
  - Security audit based on organisational policy



where to find out more

https://github.com/orgs/nokia

https://yang.srlinux.dev
https://github.com/nokia/7x50_YangModels

https://openconfig.net


NOKIA