Making Strongly-typed NETCONF Usable

Ryan Goulding
Colin Dixon

The goals of YANG

- Strongly typed
- Machine consumable
- (Automatically) Catch errors early

- **tl;dr** Standard libraries should do lots of the work (serialization, deserialization, type checking, error handling, etc.) for you
  - If your library likes it, the device should like it too
  - You shouldn't have to write parsing and error checking code

- Like most automation, this is great until it doesn't work…

Trade-offs

Strongly Typed

- Free error-checking
- Free parsing
- High likelihood of success if the data validates

Error-tolerant

- Mount NETCONF devices as-is
- Least restrictions on what you can do
- Resolve bugs/issues at any layer

This talk is about the trade-offs we have made in OpenDaylight with 3+ years of experience.

- Including 2+ years in product with tier-1 service providers
- Lessons are not OpenDaylight-specific

Outline

- Common Pitfalls
- OpenDaylight as a NETCONF compatibility layer
- Demo 1: Show discount of broken models in OpenDaylight
- Demo 2: Show schema cache in OpenDaylight
- Conclusions
- Questions

Outline

- **Common Pitfalls**
- OpenDaylight as a NETCONF compatibility layer
- Demo 1: Show discount of broken models in OpenDaylight
- Demo 2: Show schema cache in OpenDaylight
- Conclusions
- Questions

Common Pitfalls

- Ambiguities in YANG
- Small syntax mistakes
- Protocol errors

Common Pitfalls

This is not specific to the OpenDaylight implementation!

```
# ncclient example in python
# This code was shamelessly copied and pasted from the ncclient github https://github.com/ncclient/ncclient

from ncclient import manager

with manager.connect(host=host, port=830, username=user, hostkey_verify=False, device_params={'name':'junos'}) as m:
    c = m.get_config(source='running').data_xml
    with open("%s.xml" % host, 'w') as f:
        f.write(c)
```

Common Pitfalls

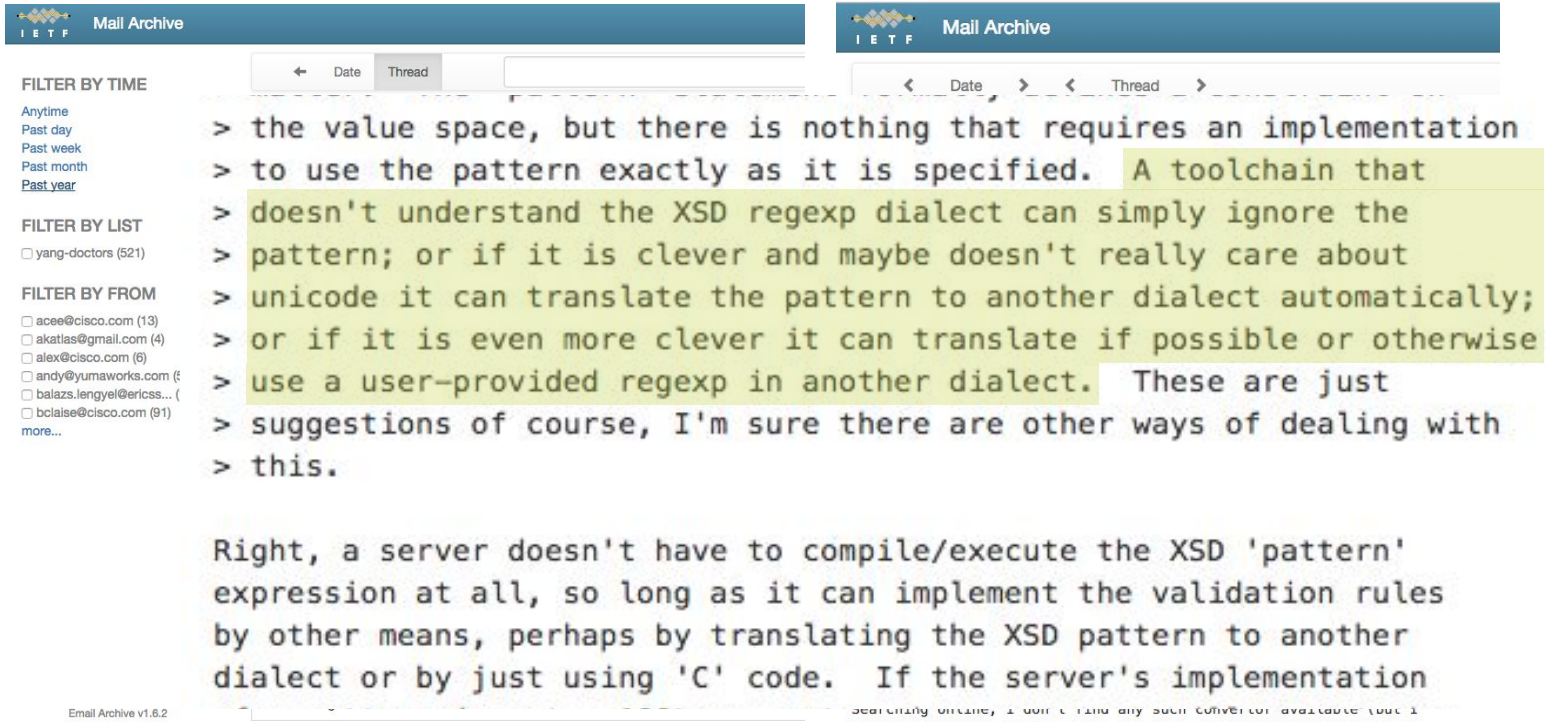
- **Ambiguities in YANG**
- Small syntax mistakes
- Protocol errors

Ambiguities in YANG

- Import without revision-date:
  - “When the optional "revision-date" substatement is present, any typedef, grouping, extension, feature, and identity referenced by definitions in the local module are taken from the specified revision of the imported module. It is an error if the specified revision of the imported module does not exist. If no "revision-date" substatement is present, it is undefined from which revision of the module they are taken.

    Multiple revisions of the same module MUST NOT be imported.” [1]
- In other words, it is implementation specific.
- What about the case when a specific version is needed for a particular southbound device?

Ambiguities in YANG



Common Pitfalls

- Ambiguities in YANG
- **Small syntax mistakes**
- Protocol errors

Small Syntax Mistakes

- Protocol messages do not contain appropriate namespaces.
  - Poor implementation, but it actually does happen quite often.
  - “There is no reason to punish our users for their vendors’ bad behaviors.”
- A leaf with type “int” has a range from 1...5, but the device is reporting back a value out of range.
  - This is not good implementation, but everyone makes mistakes.
  - Yes, it should be fixed by the vendor, but it shouldn’t totally stop a user from becoming useful with OpenDaylight. Allowance of some mistakes should be an optional configuration.
  - What do you value more: strict model enforcement or getting something working?
- Correct syntax is important, but some leniency with proper messaging unlocks the ability to use non compliant devices.
- Can also happen in YANG models themselves (see demo)

Common Pitfalls

- Ambiguities in YANG
- Small syntax mistakes
- **Protocol errors**

Protocol Errors

- Common scenario: the device does not report back the correct available capabilities.
  - Yes, this does actually happen.

Outline

- Common Pitfalls
- **OpenDaylight as a NETCONF compatibility layer**
- Demo 1: Show discount of broken models in OpenDaylight
- Demo 2: Show schema cache in OpenDaylight
- Conclusions
- Questions

OpenDaylight Mechanisms

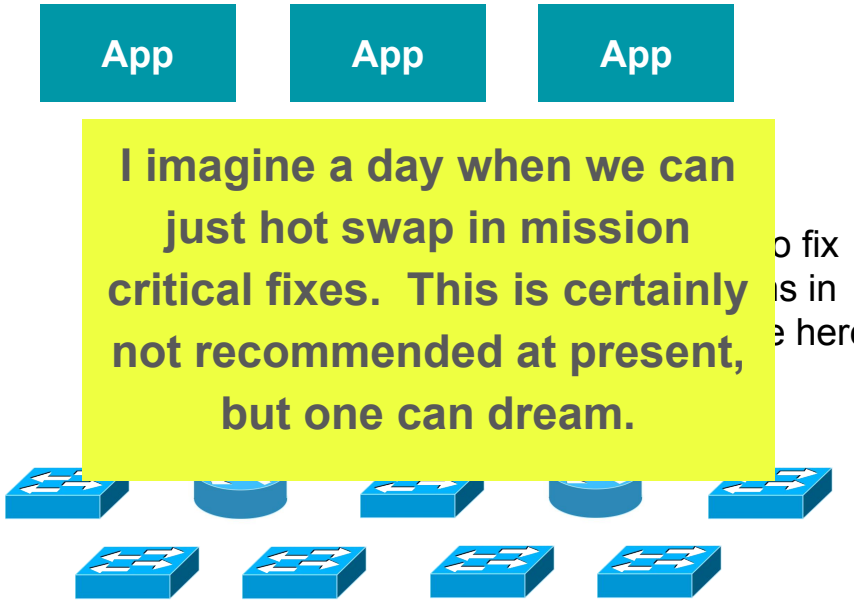
- Side-loading of YANG models
  - Normally, models are loaded using RFC 6022 to fetch YANG modes in-band
  - OpenDaylight looks for matching YANG files it its *schema cache* first

  - Makes up for lack of RFC 6022 support
  - Can fix typos in YANG
  - Can make YANG match returned data (e.g., expand a range statement)
  - Can have multiple *schema caches* so that

- Suppress bad models
  - But mount the rest of the models

OpenDaylight as a NETCONF compatibility layer

- A NETCONF controller offers a fast, low-risk way to mediate discrepancies between apps and devices
  - Our users like simple patches (or config) changes that allow interoperation with particular devices (and software versions)
  - **Flips the pattern:** Instead of blocking on new firmware, can make the changes to temporarily alleviate miscommunication. *It is all OSS after all.*
- Much faster to get, test, and deploy changes to s/w controller than firmware
- Going forward, e.g., with ISSU for controllers, this will be even more powerful



Why not just be sloppy?

- Why not just return whatever?

- People are depending on us to do type checking so they don't have to.

```
for(int i=0; i<=modelData.getSize(); i++) <do stuff>;
```

vs.

```
int len;
if(modelData.containsKey("size")){
  String strLen = modelData.get("size);
  try {
    Integer.parseInt(strLen)
  } catch {
    <log error in non-standard way>
    <actually handle error>
  }
}
for(int i=0; i<=len; i++) <do stuff>;
```

- Or worse: code just blows up somewhere random because you violated an assumption they made.

Being sloppy doesn't fix problems, it just makes them somebody else's problem.

Outline

- Common Pitfalls
- OpenDaylight as a NETCONF compatibility layer
- **Demo 1: Show discount of broken models in OpenDaylight**
- Demo 2: Show schema cache in OpenDaylight
- Conclusions
- Questions

Outline

- Common Pitfalls
- OpenDaylight as a NETCONF compatibility layer
- Demo 1: Show discount of broken models in OpenDaylight
- **Demo 2: Show schema cache in OpenDaylight**
- Conclusions
- Questions

Outline

- Common Pitfalls
- OpenDaylight as a NETCONF compatibility layer
- Demo 1: Show discount of broken models in OpenDaylight
- Demo 2: Show schema cache in OpenDaylight
- **Conclusions**
- Questions

Future work

- Allow for partial parsing of modeled data
  - e.g., what if a leaf doesn’t match its pattern, but you never actually read that leaf?
  - This gets more complicated with malformed data because it potentially interferes with the parsing of the rest of the data

Conclusions

- There is a fundamental trade off between strong typing (thus reducing developer burden) and usability/interoperability
  - “Be liberal in what you accept, and conservative in what you send” —RFC 1122
  - What about when you are immediately “sending” what you “receive”?
- Being conservative (within reason) by default, but providing easy hooks to relax constraints (in limited areas) seems to be a compelling solution.
- This approach has dramatically increased OpenDaylight’s usability as NETCONF controller with real devices in production.

Outline

- Common Pitfalls
- OpenDaylight as a NETCONF compatibility layer
- Demo 1: Show discount of broken models in OpenDaylight
- Demo 2: Show schema cache in OpenDaylight
- Conclusions
- **Questions**

Backup Slides

Abstract

NETCONF's use of strongly-typed YANG to describe device configuration makes safe and robust device configuration possible, but also exposes complexities in operation. Most deployments choose to use a controller or management system to help with issues of inventory and credential management. Further, this controller often does sanity checks to ensure that operations are likely to be successful on devices and, if not, there is useful debugging information. Frustratingly, this exposes trade-offs between relying on strict enforcement of YANG to catching errors early and more relaxed behavior to enable compatibility with imperfect NETCONF implementations in the real world. We show how we navigated these trade-offs to provide a flexible, open-source NETCONF solution that is strongly-typed and enables simple configuration changes for compatibility with imperfect NETCONF implementations.