draft-verdt-netmod-yang-solutions-00

Netmod WG

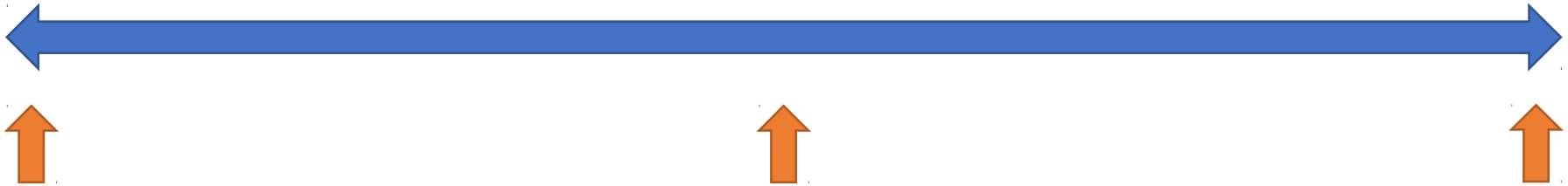
November 8, 2018

Netmod YANG Version Design Team
Rob Wilton (presenting)

Agenda

- Interesting questions for solutions to consider
- 5 potential solutions and comparison
- Modified semver in more detail
- Other work in solution space (only if time permitting):
  - Data node lifecycle (proposed changes to status)
  - Revision dates
  - Import by version
  - Client backwards compatibility (e.g. version selection)

Solution space

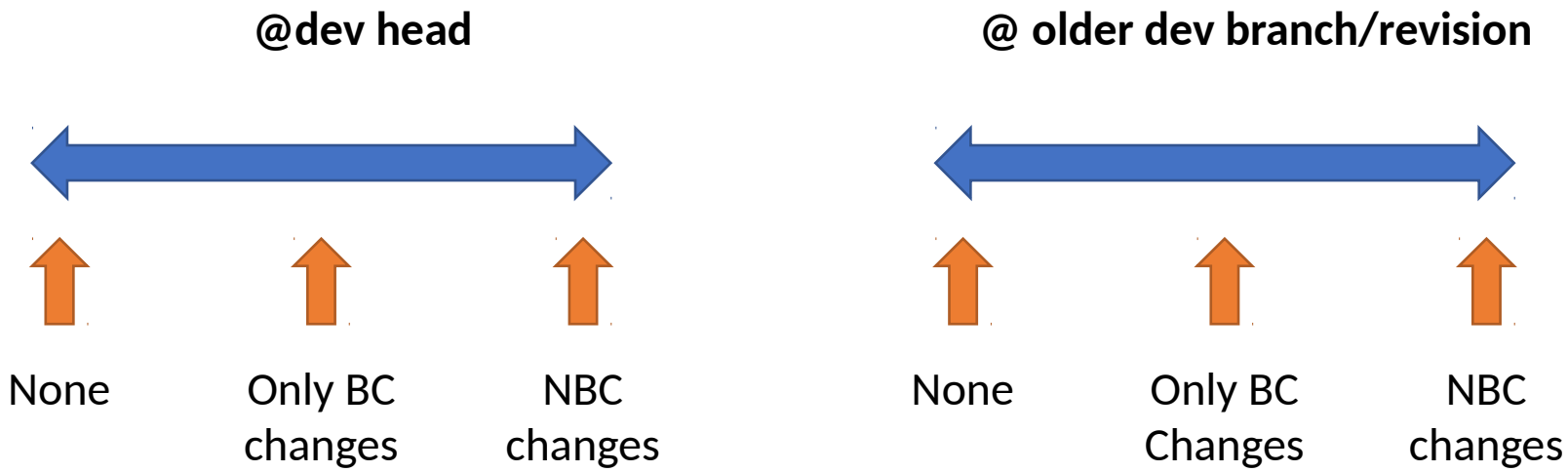


Easy for clients
Hard for servers/authors

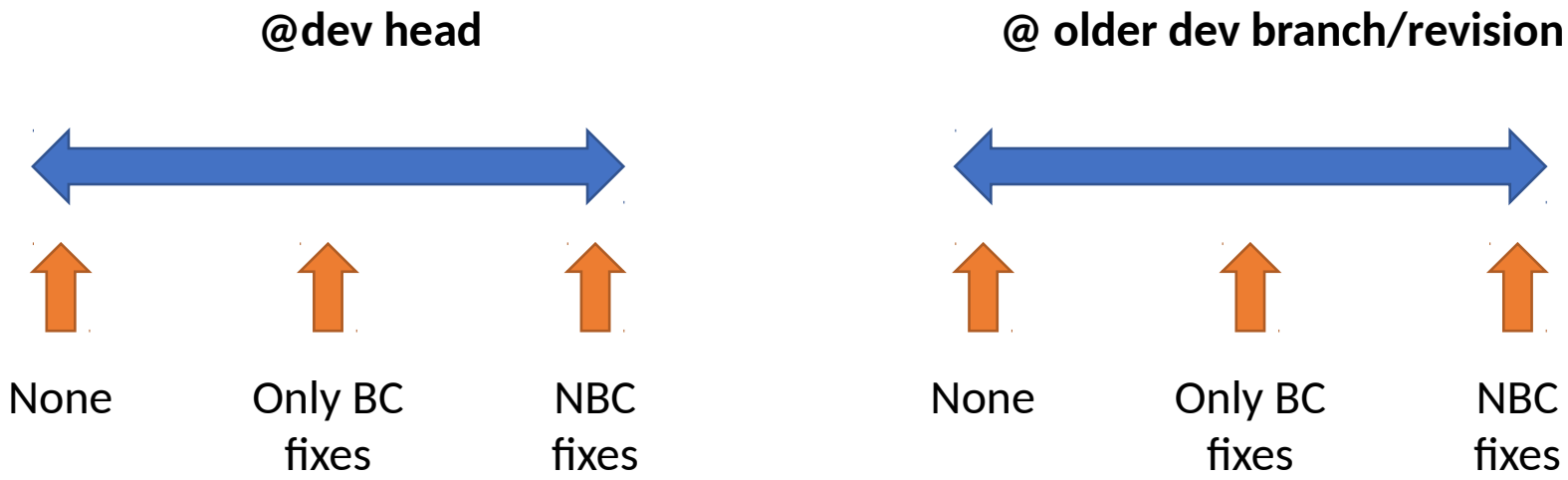
We are trying to find a solution somewhere in the middle

Hard for clients
Easy for servers/authors

Q. What level of change in YANG modules should be allowed? For **new development:**



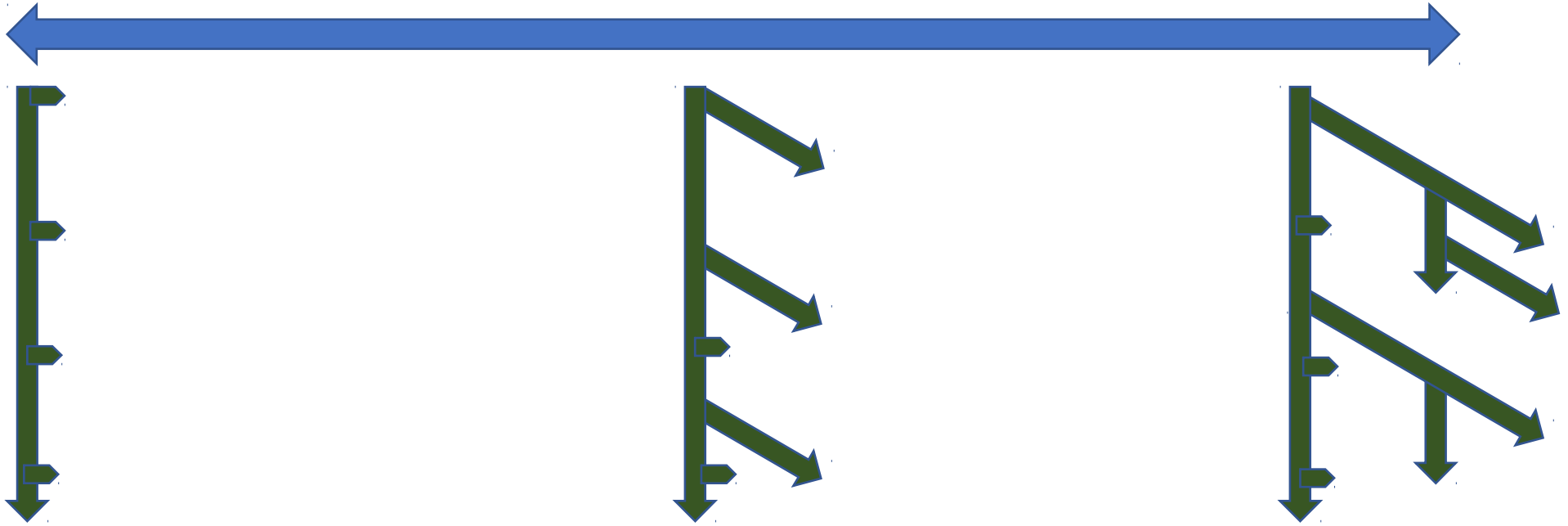
Q. What level of change in YANG modules should be allowed? For **bug fixes:**



Open questions:

- What is the definition of a bug fix?
- Can a bug fix include NBC changes?
- And does NBC mean strict RFC 7950 module update rules or something softer?
- Should we even differentiate bug fixes vs enhancements?

Q. Linear or branched module evolution?



Strict linear History
(e.g. RFC 7950 rules today)

Limited Branching
(e.g. perhaps for bug fixes, and/or minor enhancements)

Unlimited branching
(e.g. like git)

Solutions

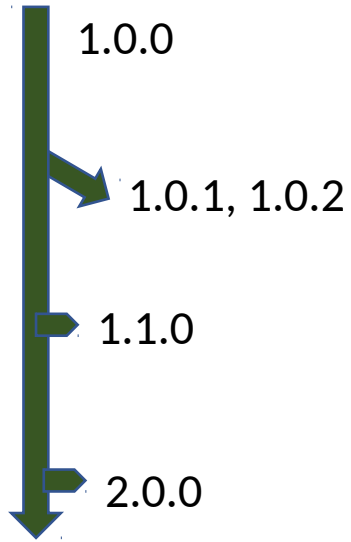
Five solutions considered in the draft:

1. Semver (as per semver.org, also used by OpenConfig)
2. Modified semver
3. Release semver
4. Schema comparison tool
5. RFC 7950 module update rules

1. Semver

Version number is MAJOR.MINOR.PATCH

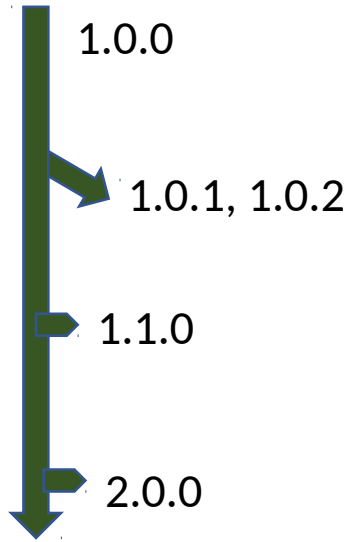
- Specified on semver.org
- Update version number based on change:
  - **NBC change** => Increment **major** (reset minor, patch)
  - **BC change** => Increment **minor** (reset patch)
  - **Editorial**/Implementation **change** => Increment **patch**
  - **0.x.y => pre-release, don't have to follow rules**
- Expectation is most development is broadly linear



1. Semver – pros/cons

Benefits:

1. Compare two versions => BC, NBC, editorial change
2. Widely known/used (e.g. used by OpenConfig)



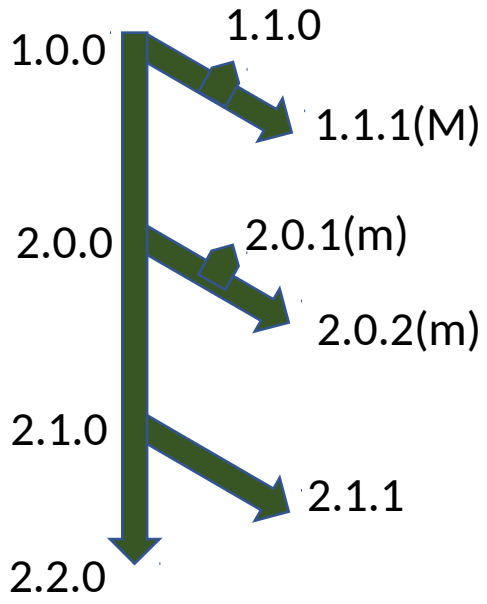
Disadvantages:

3. Very limited support for branching:
   - Doesn’t facilitate bug fixes to released versions
   - Doesn’t facilitate enhancements to released versions

2. Modified semver

Version number is MAJOR.MINOR.PATCH(m|M)

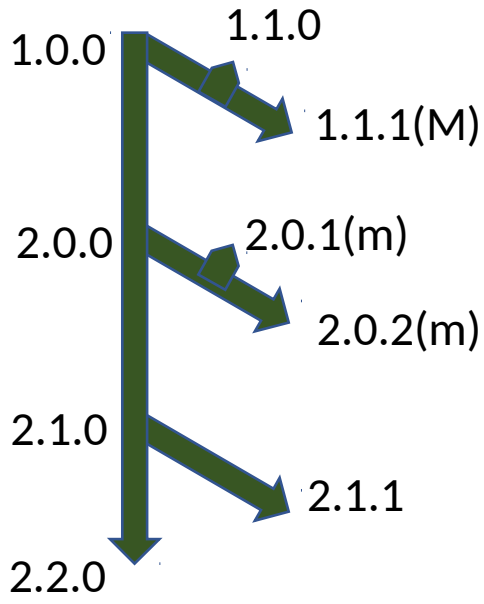
- Update version number based on change:
  - **NBC change** => Increment **major** or "**patch** + (M)"
  - **BC change** => Increment **minor** or "**patch** + (m)"
  - **Editorial**/Implementation **change** => Increment **patch**
  - **0.x.y => pre-release, don't have to follow rules**
  - **(m|M) is sticky** for a given "major.minor"
- Follow semver rules where possible (e.g. @dev head, and where possible for fixes/enhancements)
- But allows fixes/enhancements to shipped modules
- With great power comes great responsibility



2. Modified semver – pros/cons

Benefits:

1. Like semver but allows limited branching and backwards compatibility without too much extra complexity
2. Allows fixes (and enhancements) to released modules
3. Backwards compatible with semver scheme.



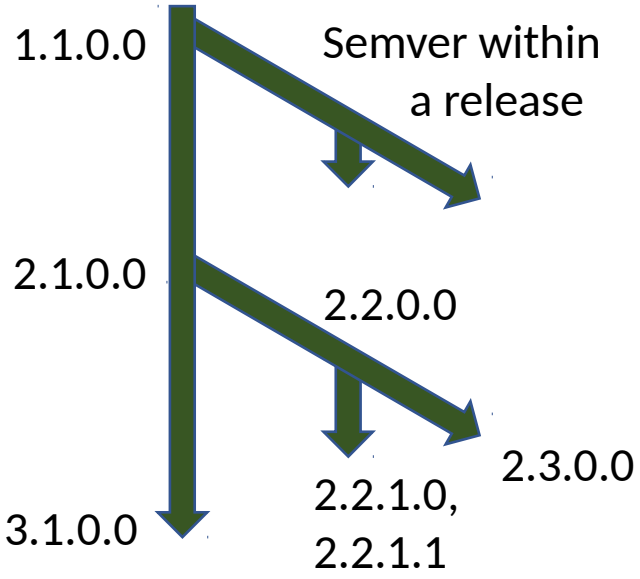
Disadvantages:

4. Cannot always semantically compare branches, e.g. 2.0.1(m) to 2.1.0 may be nbc.
5. Different to semver and slightly higher complexity

3. Release semver

Version number is RELEASE.MAJOR.MINOR.PATCH

- **Release** is updated for each major release
  - Major.minor.patch reset to 1.0.0
  - Module author defines major releases
- Within a release:
  - Normal semver rules are followed



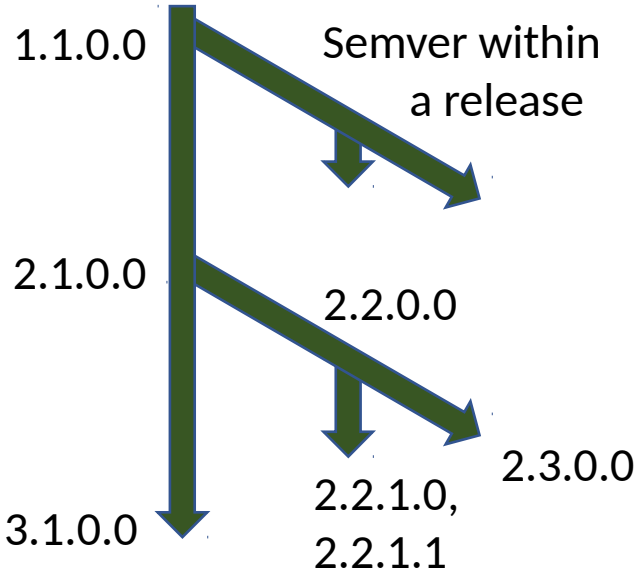
3. Release semver – pros/cons

Benefits:

1. Allows more branching, e.g. fixes/enhancements within a major release

Disadvantages:

2. Loses key semver comparison (e.g. 1.2.3.4 to 2.1.0.0 may have no changes)
3. Less familiar numbering scheme
4. What defines a major release?



4. Schema comparison tool

Idea: Use tooling to perform node by node comparison of full schema trees between two releases using rules like RFC 7950 module update rules.

Refinements:

1. Also take features and deviations into consideration
2. Restrict comparison to subset of modules that are used

Probably requires extension to tag schema nodes that cannot be automatically compared by tooling (e.g. description, pattern, xpath changes)

4. Schema comparison tool – pros/cons

Benefits:

1. Full solution - any two schema trees are comparable.
2. Can take more considerations into account, can give a more accurate answer.

Disadvantages:

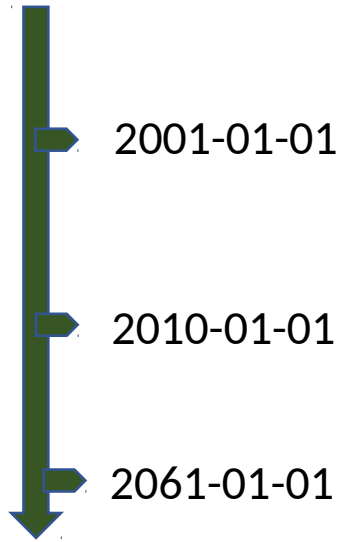
3. Have to run tool, can’t just look at a number.
4. Will require annotations in schema for cases where the tooling cannot figure it out, could be noisy depending on quality.
5. Requires a way to document instance data usage.

Probably not a solution on its own. Best used in conjunction with a versioning number scheme.

5. RFC 7950 module update rules

Specified in RFC 7950 chapter 11:

- All changes are backwards compatible
- Introduce new nodes (or modules) for nbc changes
- Deprecate, then obsolete, old nodes
- Strictly linear history, all updates to latest revision



Assumptions (not in RFC 7950):

- Deprecated nodes must be implemented
- If versioned by a new module, then old module is also supported (for a while)

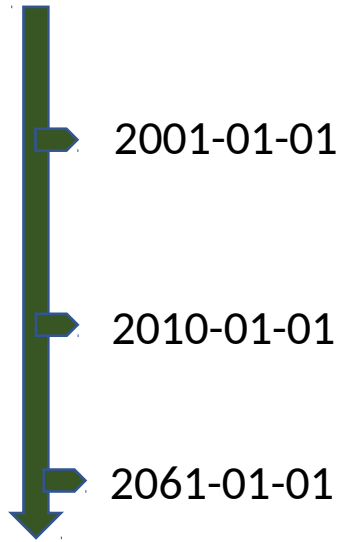
5. RFC 7950 module update rules– pros/cons

Benefits:

1. Already defined
2. Fully backwards compatible for clients

Disadvantages:

3. Not sufficient (or we wouldn't be here)
4. Doesn't allow bug fixes to older releases
5. New module breaks all existing paths
6. Two paths to one underlying property synchronization issues



Solution comparison table
(Not intended to be a beauty contest)

Solution	Branching	BC fixes	NBC fixes	BC dev	NBC dev	Semver
Semver: Maj.Min.Patch	Very limited	Only guaranteed at head	Only guaranteed at head	At head only	At head only	Yes
Modified Semver: Maj.Min.Patch(M|m)	Limited	Yes	Yes	Recommended at head. Allowed on branch	Recommended at head, allowed on branch	Yes, except changes on (m|M) versions.
Release Semver: Rel.Maj.Min.Patch	Yes, if release number updated	Yes, within a release	Yes, within a release	Yes, within a release	Yes, within a release	Within a release only
Schema compare tool	Supported	Anywhere	Anywhere	Anywhere	Anywhere	Yes, offline
RFC 6020/7950 module update rules	Linear	At head only	Limited/ Disallowed	At head only	Disallowed	Implied (always backwards compatible)

Modified semver – current front runner

- There is no perfect technical solution
- Modified semver seems to strike a pragmatic balance, hence DT front runner.

- Vendors should decouple versioning of modules (API) from the versioning software artifact (API implementation)
- The scheme allows some things that are undesirable, hence need to provide a set of sensible usage guidelines.
- Open issue: Is updating major and minor version number strict? Or can they be incremented without a major/minor version change?

Modified semver – example guidelines

1. Churn is painful for clients, try to ship high quality modules and keep them stable
2. Minimize nbc updates - particularly on released code
3. “m|M” should be used as a tool of last resort, use normal semver where possible
4. Version modules independently from releases
5. If a fix is made on an older branch then port it to all newer branches at the same time.
6. Deprecating nodes is a bc change, obsoleting is nbc.

Other work in solution space …

… only if time permitting:

- Data node lifecycle (proposed changes to status)
- Revision dates
- Import by version
- Client backwards compatibility (e.g. version selection)

Data node lifecycle - YANG status changes

1. We want the YANG versioning solution to refine:
   i. “status deprecated” to mean that a server MUST implement (or deviate)
   ii. “status obsolete” to mean that a server MUST NOT implement
   iii. Adding “status deprecated” to a module is a BC change
   iv. Adding “status obsolete” to a module is an NBC change
2. Allow “description” under “status” statement
3. Provide two global leaves in YANG library:
   i. “Implements deprecated nodes” - if set, the server always implements, otherwise behavior is unspecified
   ii. “No obsolete nodes“- if set, the server never implements, otherwise, behavior is unspecified

Revision dates

Currently assumed to chronologically ordered, but branching breaks this. Two choices:

1. Ditch revision dates =>
   Respin YANG library, hello messages, etc to use version number
2. Retain revision date =>
   - Lose chronological ordering guarantee
   - But retain uniqueness, i.e. (module name, rev-date) uniquely identifies a module

Design team is leaning towards 2.

Import by version

Not been discussed by the DT in any great detail yet, but:

- We want to extend YANG import to allow import by version

  Perhaps also want to recommend that import by revision is not used.

- Normally will be:
  - a set of compatible version compatible,
  - with some sort of wildcard support, or version X or later.
- If we go with modified semver, need to consider bc/nbc updates to patch number.

Client backwards compatibility

If we go with modified semver …

… also not been discussed by the DT in any great detail yet, but:

- This is a hard/expensive problem to solve
- Probably requires clients selecting a particular module set via YANG library and protocol additions and the server maps request data between older versions to latest.
- Mapping to latest will be hard, impossible in some cases.
- Could be done in server, controller, or client
- What version does the client get if it doesn’t choose (earliest, or latest)?

Next Steps

- Incorporate feedback from the WG
- Converge on a solution within the DT
- Hopefully have initial solution draft(s) ready for IETF 104
- We may split the solution into multiple drafts, e.g.
  - Core solution - module version number
  - Extension – version selection
  - Extension – schema tree comparison