Aalto University
School of Electrical
Engineering

Service evolution

21.3.2019

Santeri Paavolainen

So far …

- **We've discussed primarily**
  - Systems with static interfaces or where interface changes are not an issue
  - Co-developed systems / internal customers that can be updated at the same time
- **What if not a valid assumption? Where not? How to handle?**

Why service evolution?

- **Services change over time**
  - Internal implementation should not be visible
  - ... but abstractions are leaky
  - Sometimes for performance, security or other reasons internal changes must be reflected externally
- **Needs and requirements change over time**
  - Many changes are actually desirable

- **Obsolescence**
  - Maintaining "old" systems becomes a cost
    - *Hard-to-find hardware components, insecure software, difficult upgrades, skill retention, "undesirable" maintenance jobs → increased maintenance costs (and risks)*
  - Most severe service interface change: removal

➔ **Externally visible changes unavoidable**

When a problem?

- Coupling
- Hidden assumptions
- Undocumented behavior
- Customers
- Marketplace

Coupling

- **Interface definition "id of 6 alphanumeric characters"**
  - Customers: "`create table … (remote_id varchar(6) not null)`"
- **Update to "id of 12 alphanumeric characters"**

- **Interface definitions create coupling between interface and implementation**
  - This is actually what interfaces are for! You want to create coupling <u>only</u> based on an interface
- **Problematic when interface changes**
- **<u>Explicit change</u>**

Hidden assumptions

- **Interface: “id of alphanumeric characters”**
- **Customer sees only ids of <u>six</u> characters**
  - “`create table … (remote_id varchar(6) not null)`”
- **Update to “id of 12 alphanumeric characters”**
  - Note that specification did not set an explicit bound
  - Technically the new spec is a <u>subset</u> of the old id space

- **Assumption is hidden from the interface provider**
  - Still causes problems with customers
  - Originally problem was ambiguity in the interface definition
- **<u>Explicit change</u>, but probably assumed not to cause problems**

Hidden assumptions

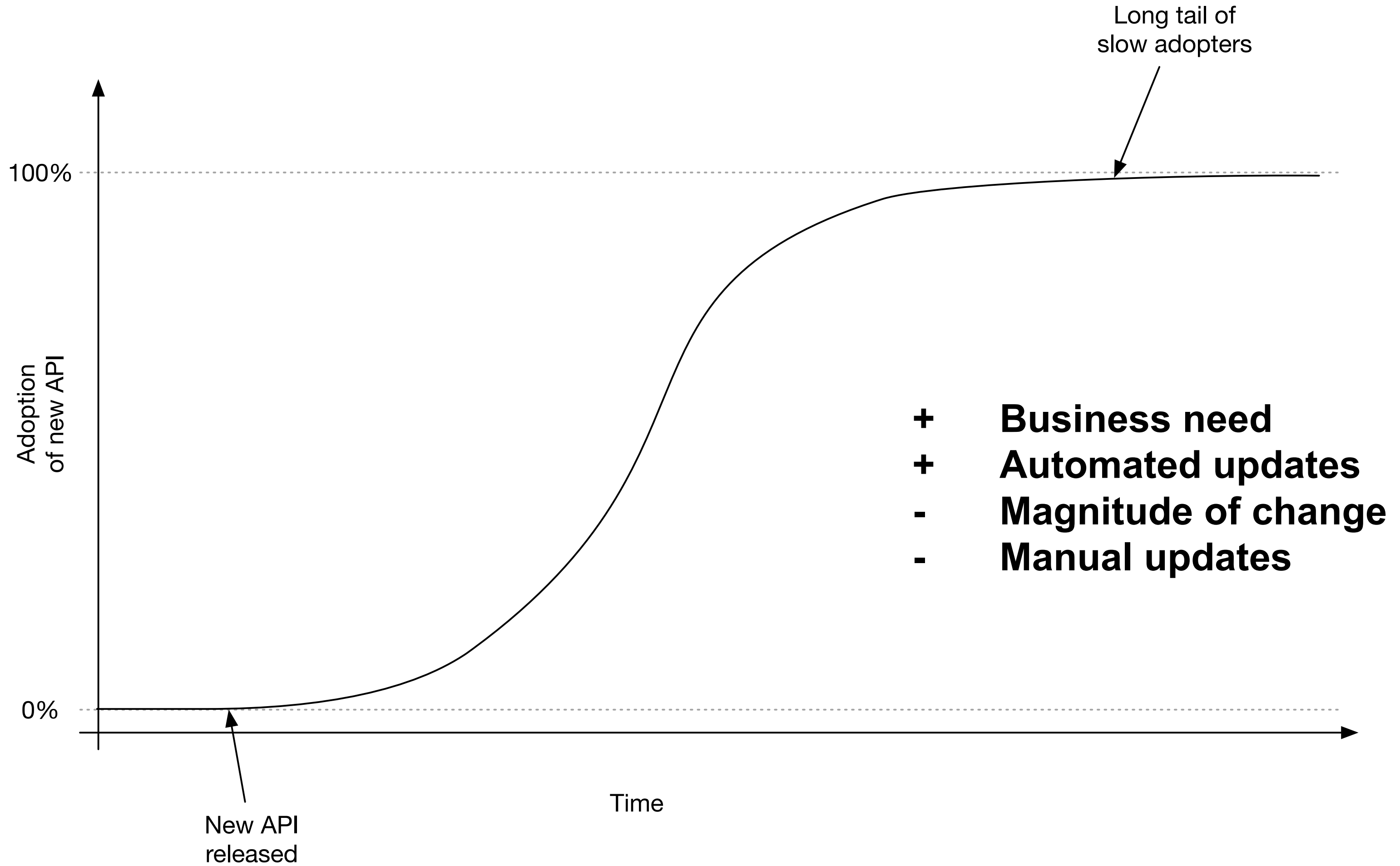
- **Interface: “replies with POST to CB URL of DATA(seq) and END(seq) messages, messages are ordered by their sequence number”**
  - Implementation synchronous and <u>always</u> does first DATA(n) and only after first POST completes, then END(n+1)
- **Changed to asynchronous implementation**
  - Now can do:
DATA(n)→END(n+1),
END(n+1)→DATA(n) or even parallel
DATA(n)|END(n+1)
- **Would this cause problems? In what situations?**

Undocumented behavior

- **(I just saw a cool YouTube video going deep into this, that's why)**
- **Commodore 64's PLA chip had a timing issue (C64 1982)**
  - This was exploited by FastLoad (and others)
  - Became known widely and used by many games etc.
- **Results**
  1. Commodore changed from externally sourced PLA chip to in-house design → had to religiously re-create glitches from original
  2. People creating modern PLA replacements (chips from step 1. fail) have to re-create glitching behavior (not trivial)
- **(What's the difference between hidden assumption and undocumented behavior?)**

Customers (+ users)

- **Both internal (f.ex. mobile app team) and external (3rd parties)**
- **Interface changes can be made atomic; integrations to them change asynchronously**
  - Development lead times (design, coding, testing, etc.)
  - Inertia to changes (organizational, security, financial, …)
  - Contractual commitments (supporting specific version)
  - Deployments may take time and depend on end-users (consider firmware upgrades to TVs, for example)
- **Result: delays in adoption, potentially with very long tail**


Long tail of
slow adopters
100%
Adoption
of new API
+ Business need
+ Automated updates
- Magnitude of change
- Manual updates
0%
New API
released
Time

Marketplace (aka business context)

- **Who holds the power? Customer or the vendor?**
  - If customer, things usually go on their pace
- **Is there competition on the market?**
  - Major changes in APIs may lead to customer re-evaluating their changes (e.g. a large expenditure anyway for an upgrade)

Fixes?

Handling service evolution

- **Case-by-case basis**
  - Changes in customers, marketplace etc.
  - OTOH, usually same practices mostly apply

- **Things to do prior to changes**
  - SLAs, contracts, communication
- **Technical solutions**
  - Versioning, interface migration, service contracts
  - Backward-compatible changes
  - Adaptable protocols

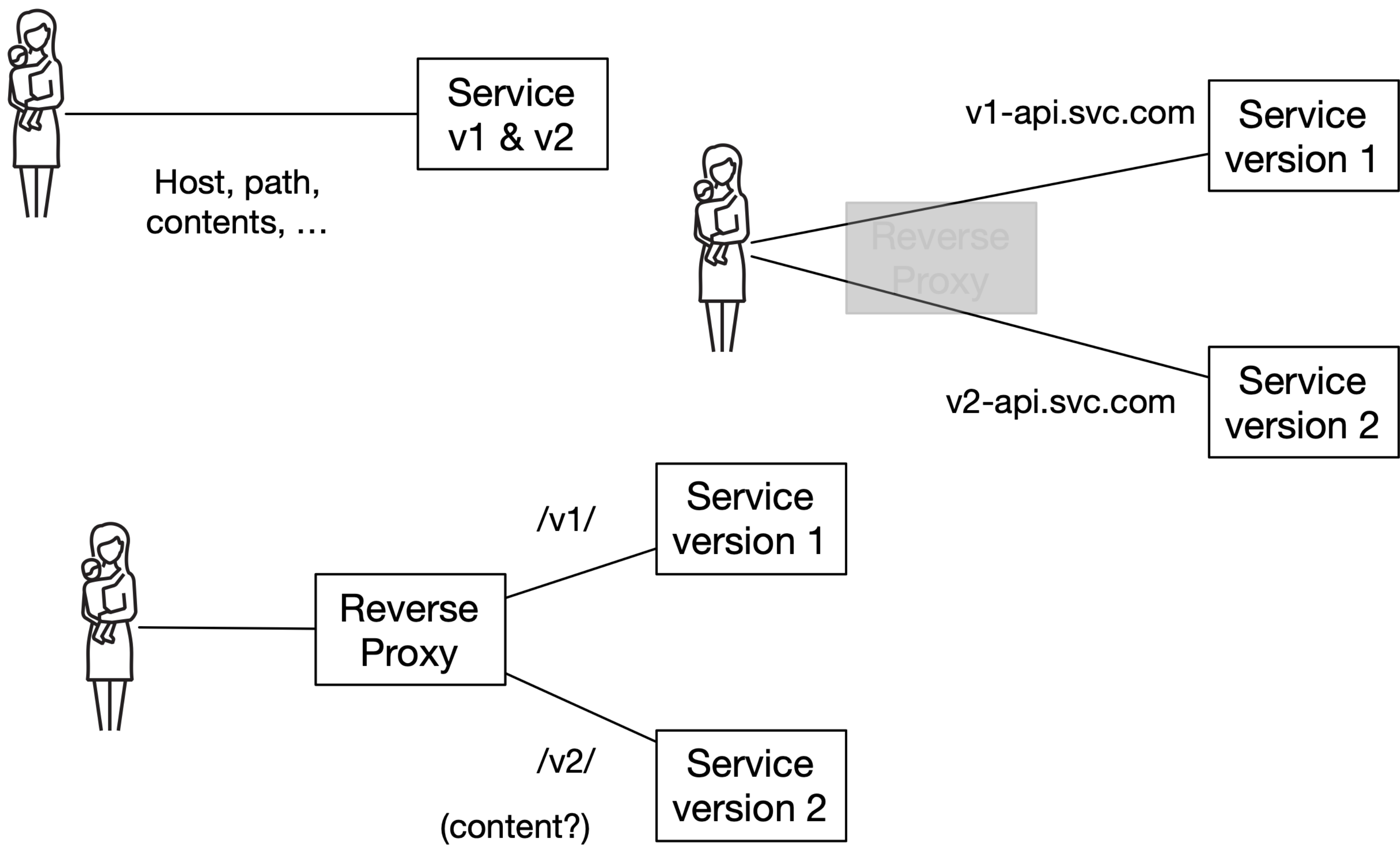
SLAs, contracts and communication

- **Think about interface (API) as a contract**
  - Unilateral decisions usually bad (you think you know your customers?)
- **Bake into SLAs how interface changes are handled:**
  - “patch, minor and major releases” – different support targets
  - What is the maximum time “old” versions are supported (if at all)
  - Is there advance warning given to customers of changes? (Do you commit to those?)
- **Communicate with your customers**
  - Discussion forums, beta testers, prior information on substantial (non-backward compatible) changes, etc.
- **Goal is to minimize surprises to customers and frustration**

Versioning

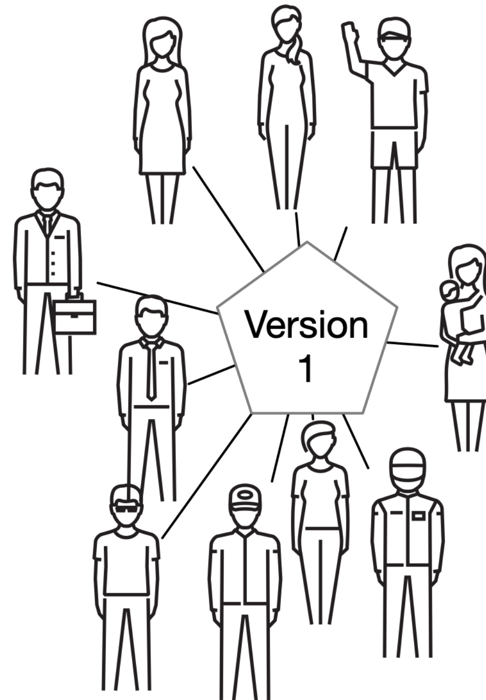
- **Explicitly specifying what interface version used**
- **Typical versioning schemes**
  - Simple monotonic integer: v1, v2, v3 …
  - Dotted (A.B or A.B.C): v1.10, v.1.10.12 (always consider as integers! v1.1 = v1.01, having v.1 being implicitly two-digit v.10 is a disaster)
- **Part of request (maybe also reply)**

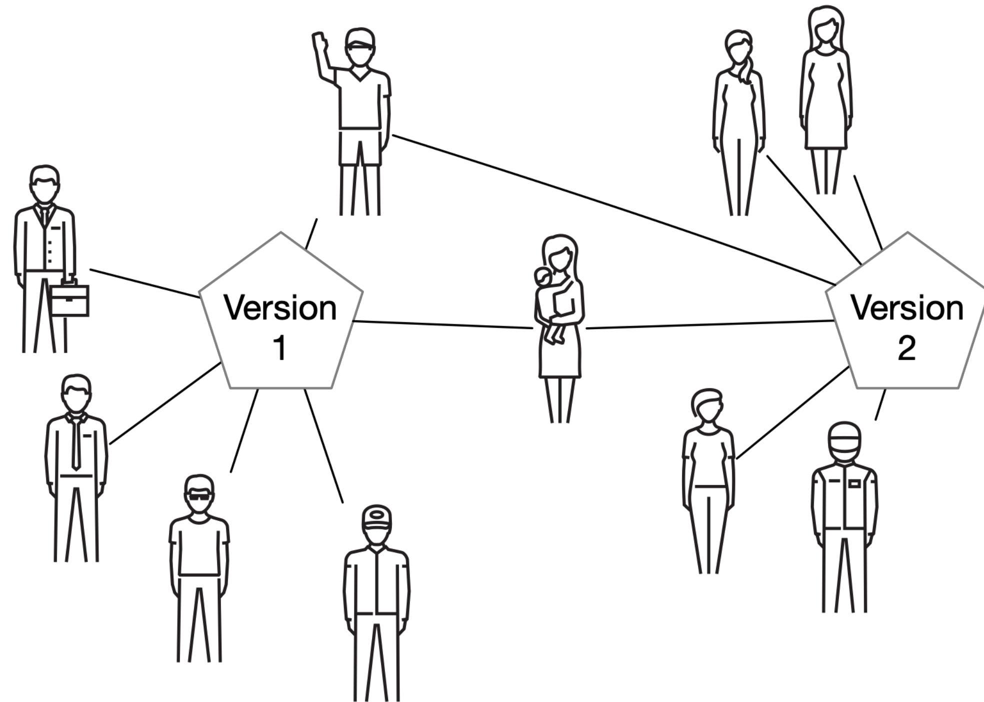
- **Host:**
  - `v1.api.service.com`, `api-v2.service.com`
- **URL:**
  - `/v1/resource`, even `/latest/resource` for adventurous
- **Query string:**
  - `/resource?version=2`
- **Accept header:**
  - `Accept: application/mytype+v1`
- **Custom header:**
  - `X-Interface-Version: 1.0`
- **Request body (JSON, XML):**
  - `{"version": "1.0", …}`

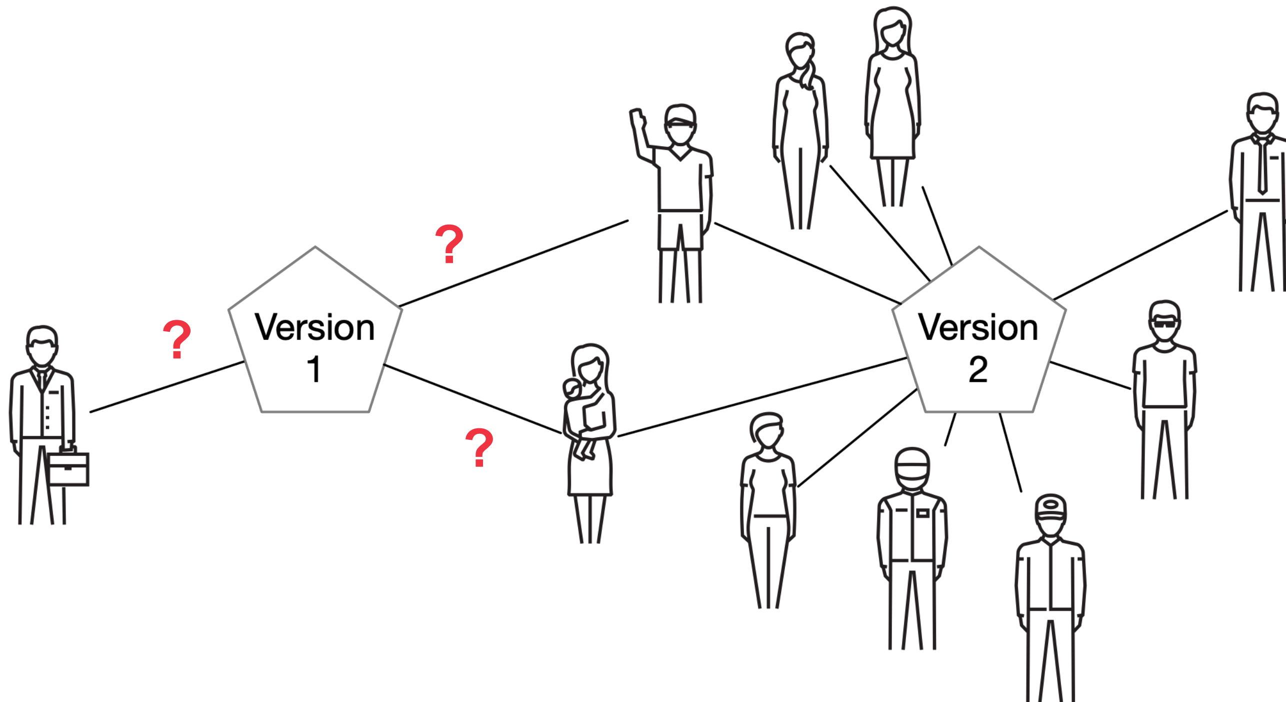

Service
v1 & v2
Host, path,
contents, …
v1-api.svc.com
Service
version 1
Reverse
Proxy
Service
version 2
v2-api.svc.com
/v1/
Service
version 1
Reverse
Proxy
/v2/
Service
version 2
(content?)

Problems with versioning

- Implies full support across versions
- Deployment and development complications


Version
1


Version 1
Version 2


?
Version
1
?
?
Version
2

Problems with versioning

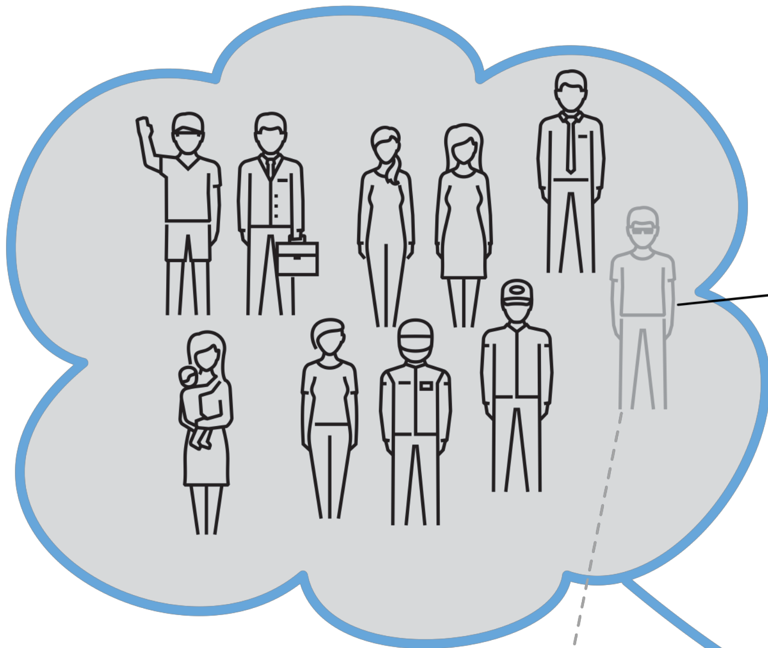
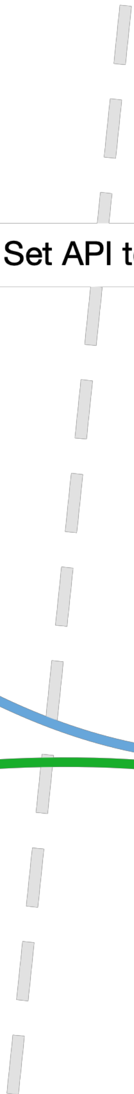
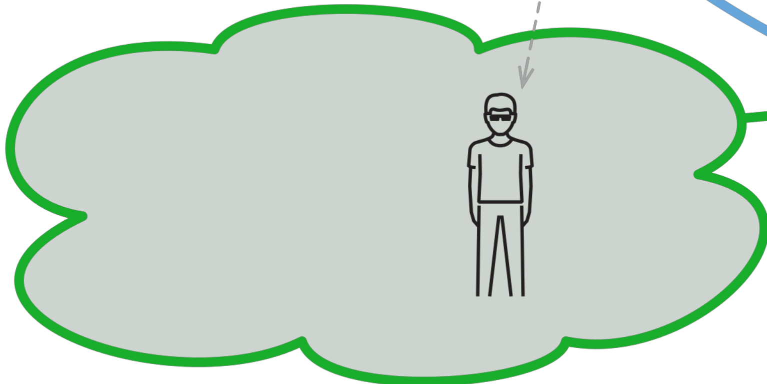
- **Implies full support across versions**
  - Customer may use v1 and v2 simultaneously!
  - Unless explicitly somehow managed (see later)
- **Deployment and development complications**
  - Version support built into the application?
  - Parallel deployment of multiple versions?
- **Stateless services easiest to version**

Stateful service versioning

- **Support all versions by a <u>single service</u> (code)**
  - No “almost but not entirely” parallel deployments
  - Also potentially a nightmare to maintain
  - ```
    req.version match {
      case V1API => …
      case V2API => …
      case V3API|V4API => …
    }
    ```
- **Use multiple backends**
  - All backend versions use the same data storage
  - Must be updated together for any schema changes
  - Can isolate “compatibility” layer to old backend versions
- **Sometimes stateless translator is sufficient**
  - Format old request → new request and proxy
  - Format new response → old response
    ```

Migration

- **Idea: Run new and version in parallel but each customer in only one**
  - Migrate customers to the new version
  - Customers get to choose when to migrate


Version 1 customers
Set API to version 2
Customer Information
Query: What version?
Version 1
Reverse Proxy
Migrate customer data
Version 2


Version 2 customers

Migration

- **Idea: Run new and version in parallel but each customer in only one**
  - Migrate customers to the new version
  - Customers get to choose when to migrate
- **Pros**
  - Removes need of explicit versioning from interface
  - Version to be used becomes part of the customer configuration
  - Migration on customer's own pace
- **Cons**
  - Requires explicit customer information (+ no SLA on anonymous APIs)
  - Schema changes and data migration (Rollback? Lots of data to transfer?)
  - Gateway has to know which version to use (dynamic)

Adaptable protocols

- **Idea: Let the protocol itself be resilient and adaptable**
  - Implicitly by presence or absence of fields in data or messages
  - Explicitly through capability negotiation

Implicit adaptability in data

- **Assume two request formats (old and new)**
  - fullname or first_name + last_name fields
  - ```
    if (req.json.first_name &&
        req.json.last_name)
      full_name =
       req.json.first_name + " " +
       req.json.last_name
    else if (req.json.full_name)
      full_name = req.json.full_name
    else
      ???
    ```
- **Client-side adaptability really only relevant if multiple endpoints**
  - HTTP, …
- **Potentially could have schema (OpenAPI?) that informs the client whether a field must be recognized or whether is optional**
  - You will find these mechanisms in some protocols
  - Usually overkill (on problems you're likely to encounter)
- **Can become quickly very hairy**
  - Test coverage!
    ```

Capability negotiation

- **SMTP, IMAP, ISAKMP etc. examples**
  - Session-oriented protocols
  - * OK The Microsoft Exchange IMAP4 service is ready.
    1 capability
    * CAPABILITY IMAP4 IMAP4rev1 AUTH=PLAIN AUTH=NTLM AUTH=GSSAPI SASL-IR UIDPLUS MOVE ID UNSELECT CHILDREN IDLE NAMESPACE LITERAL+
    1 OK CAPABILITY completed.
    2 authenticate PLAIN
    ...
    2 OK AUTHENTICATE completed.
- **Client-server (IMAP, SMTP) or peer-to-peer (ISAKMP)**
- **Probably overkill for most interfaces**
  - Although potentially useful when you control the client library

Image: Monty Python’s Holy Grail (movie)

Holy grail of service evolution

- **No explicit versioning**
  - "It just works!"
- **Backward-compatible changes during transition**
  - "It just keeps on working!"
- **Customer versions are known accurately**
  - Interface contracts (per-customer versioning)
  - Alternatively you have a binding obsolescence policy

  → Able to know accurately when backward-compatibility can be safely removed
- **Of course, this is not trivial and may be impossible to achieve**

Examples

AWS resource id expansion

- **Original resource IDs 8 characters long**
  - Plus type prefix, e.g. “i-<8 chars>”, “sg-<8 chars>”
  - 16^8 = 32 bits = ~ 4 billion (10^9)
- **Currently resource ids 17 characters**

  $16^{17} =\sim 10^{20}$

- **<u>First mentions</u> in late 2015**
  - Announced to be starting in 2016
- **<u>Enabled</u> in Jan 2016**
  - Opt-in e.g. customer’s choice
- **<u>Deadline</u> in Dec 2016**
  - (roughly … a bit more nuanced)
- **About 11 months to adopt**
  - Could opt-in earlier
  - After deadline, would not receive shorter ids from APIs
  - Existing old ids continue to be valid
- **Provided APIs for querying id length and opting in!**

Twitter API cleanup

- **Removal of some API features in 2018**
  - "two legacy developer tools used by about 1% of third-party developers" (link)
- **Mixed reactions**

> There's a world in which Twitter embraced third-party developers fully, letting their energy and ideas infuse its struggling platform with new life. Most of Twitter's best ideas have come not from the company, but from its users. But that would introduce new costs and complexity into a company that is struggling to meet basic business objectives.

- **Rationale?**
  - Pretty vague about that "1%" – a single developer could easily account >50% of traffic
  - Business needs? Replacement API has $$$ tiers
- **3 month grace period**

IPv6 adoption

- **Original "interface" goof**
  - IPv4 (1981)
  - Only 2^{32} unique addresses
  - Has lead to widespread NAT use – bane of other protocols
  - Top-level RIRs exhausted pools of free address blocks in 2011

- **IPv6 as a replacement**
  - 2^{128} address space ($\sim 10^{38}$)
  - Draft standard in 1998
  - 21 years later adoption at 25%

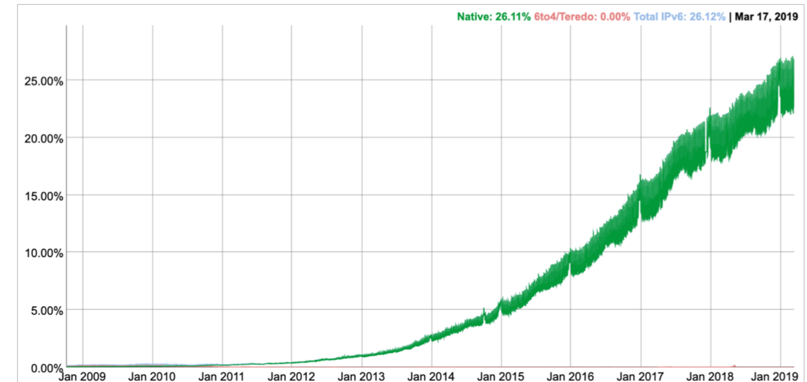


Image: Google IPv6 statistics

Future-proofing interfaces

Be conservative in what you send,
be liberal in what you accept

(Postel’s law)

Future-proofing interfaces

- **Adaptability vs. evolution**
  - Adapting the existing interface to work with new needs
  - Evolving a new version of the interface for new needs
- **Interface adaptability should not be a default!**
  - Only when changes are expected to occur, but details unknown
  - Extensibility in interfaces comes at cost: implementation, data transfer, complexity, …
  - Versioning is often way cheaper to do
- **When would be applicable?**

Extensibility in data formats

- **Extensibility in the transit data representation**
  - Unconstrained key-value maps easy to extend
  - Most marshallers map “objects” to maps
  - Avoid single-value lists, prefer maps
    `[1,2,3,4]` vs.
    `[{“value”:1},{“value”:2},…]`

- **Added fields should work with old marshallers**
  - `{“value”:1,”modified”:”…”}`
  -
```
type response1 struct {
  Value int `json: »value"`
}
type response2 struct {
  Value int `json: »value"`
  Modified string
`json: »modified"`
}
```

Extensibility in data reception

- **POST should try to work even if missing fields**
  - "Sane defaults"
- **PUTs should gracefully handle missing fields**
  - Difference between PUT /user
    `{"fullname": null}`,
    `{"fullname": ""}`, and
    `{}`
- **Same logic applies to other protocols (gRPC, Thrift)**
  - Though requires planning

Planning for evolution

What can you do in advance?

- **Define your commitment to past versions in SLA**
  - Different commitment for paying customers and free users
- **Decide on approach**
  - Versioning may be difficult to add later
  - Host vs. request vs. path vs. parameter? Implicit? Adaptability?
- **Future-proof ing interfaces**
  - POST / PUT logic
  - JSON: use maps
  - gRPC: see here
  - Consider even if explicit versioning is used!
- **Monitoring and metrics?**

What can you do in advance?

- **Understand your environment**
  - Internal and external customers, stakeholders, their needs
  - Business goals
- **Try to predict which dimensions will be critical**
  - Auditability, performance, security, data, scalability, …
- **These should guide understanding what initial choices to make**
  - They may be wrong, of course