

RePoSE

Restful Proxy Service Engine

Jorge L. Williams
Grant Herbon



Team: Cloud Integration

Mission: Accelerate other cloud product lines by providing them with common tools and services. To ensure consistency between cloud products.

Common Challenges

- Multiple Service Teams
 - Different Developers = Different Sets of Skills
 - Different Programming Languages: Python, Ruby, Java
- All APIs must support common functionality in exactly the same way -- implementing some of these tasks is not trivial
 - Versioning
 - Rate Limiting
 - Content Negotiation (JSON? XML?)
 - Authentication / Authorization
 - Caching
 - Logging
- All APIs must interact with:
 - Auth Service / IDM System
 - Billing
 - These interactions may be complex

Given this...

- How do we ensure consistency?
 - Our APIs should look as though they are part of a suite.
 - How do we make sure something like Rate Limiting works the exact same way between services written by different devs, in different programming languages?
 - Consistency from an operations perspective is also a concern: One team uses squid, another varnish...different skill sets required to support caching.

Given this...

- How do we avoid duplicating effort?
 - Rate Limiting, Versioning, Content Negotiation, Format Conversion, Caching these are:
 - Difficult to implement and QA
 - Take time to get right
 - Why waste time implementing and QA-ing the same functionality more than once?

Given this...

- How do we simplify interactions with the service framework?
 - Interactions with the service framework can be complicated. How can we allow service teams to easily integrate with our enterprise?
- Also, how do we abstract away these interactions...
 - OpenStack services may be deployed outside Rackspace.
 - How do we build services that can be deployed on other enterprises where we can have
 - Different authentication mechanisms
 - Different integration points
 - Different definition of a tenant
 - Different chargeback schemes etc

The problem becomes more difficult as we develop new products...



...

A Solution...

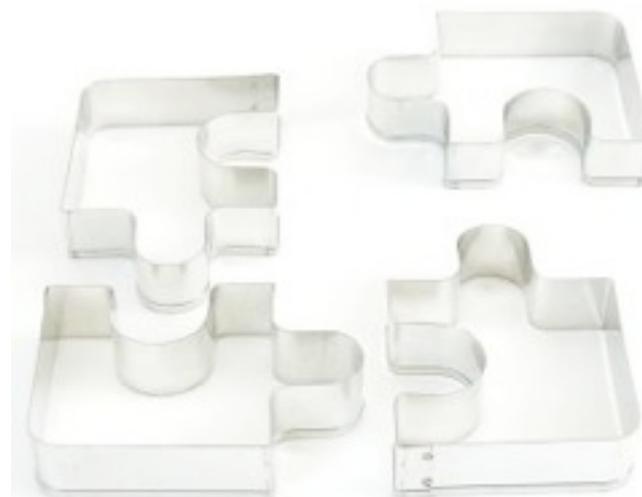
- We allow code reuse of API related code and services
- In other words, we write....
 - Rate Limiting,
 - Versioning,
 - Caching,
 - Logging,
 - etc...



- ...once and allow service teams to use the implementations directly.
- Take a cookie cutter approach to putting APIs together...
 - Have Service teams focus on things that are different, the X in XaaS.
 - Get the hard API stuff for free
 - Code and services are made available and implemented as components.

Code Reuse is nice...

- ...but how?
 - Different development environments
 - Operating Systems
 - Programming Languages
 - Tools/Capabilities
 - Etc
- This is actually a very common problem...

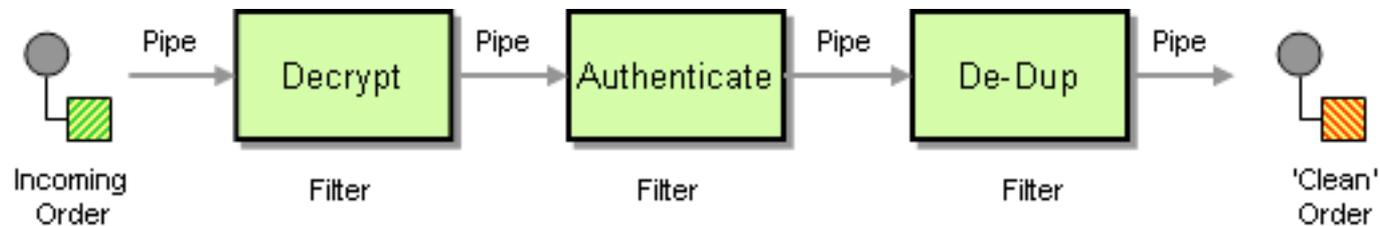


Integration

- It is actually an integration problem...
 - Services integrating with the Service framework
 - Clients integrating with Services
 - Services integrating with other Services
 - Repetitive functionality, ideal candidates for code reusability but services may be written in different languages etc.
- Lots of Research on this Topic:
 - Enterprise Integration Patterns
 - by Gregor Hohpe and Bobby Woolf
 - <http://www.enterpriseintegrationpatterns.com/toc.html>
 - Common Integration Patterns exist for solving this very problem...

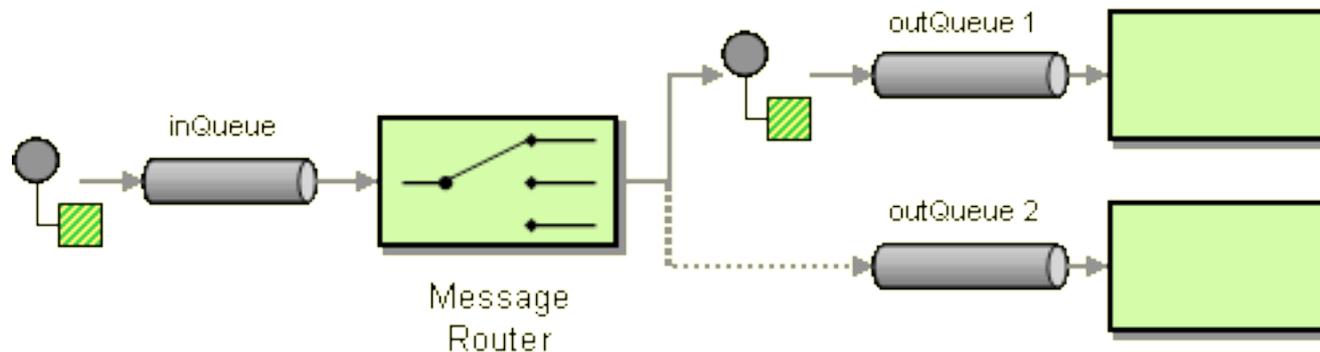
Common Patterns: Pipes and Filters

- How can we perform complex processing on a message while maintaining independence and flexibility?



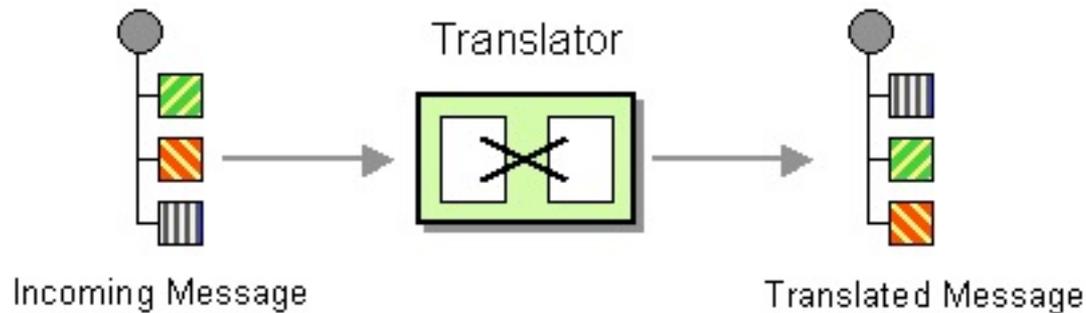
Common Patterns: Message Router Pattern

- How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?



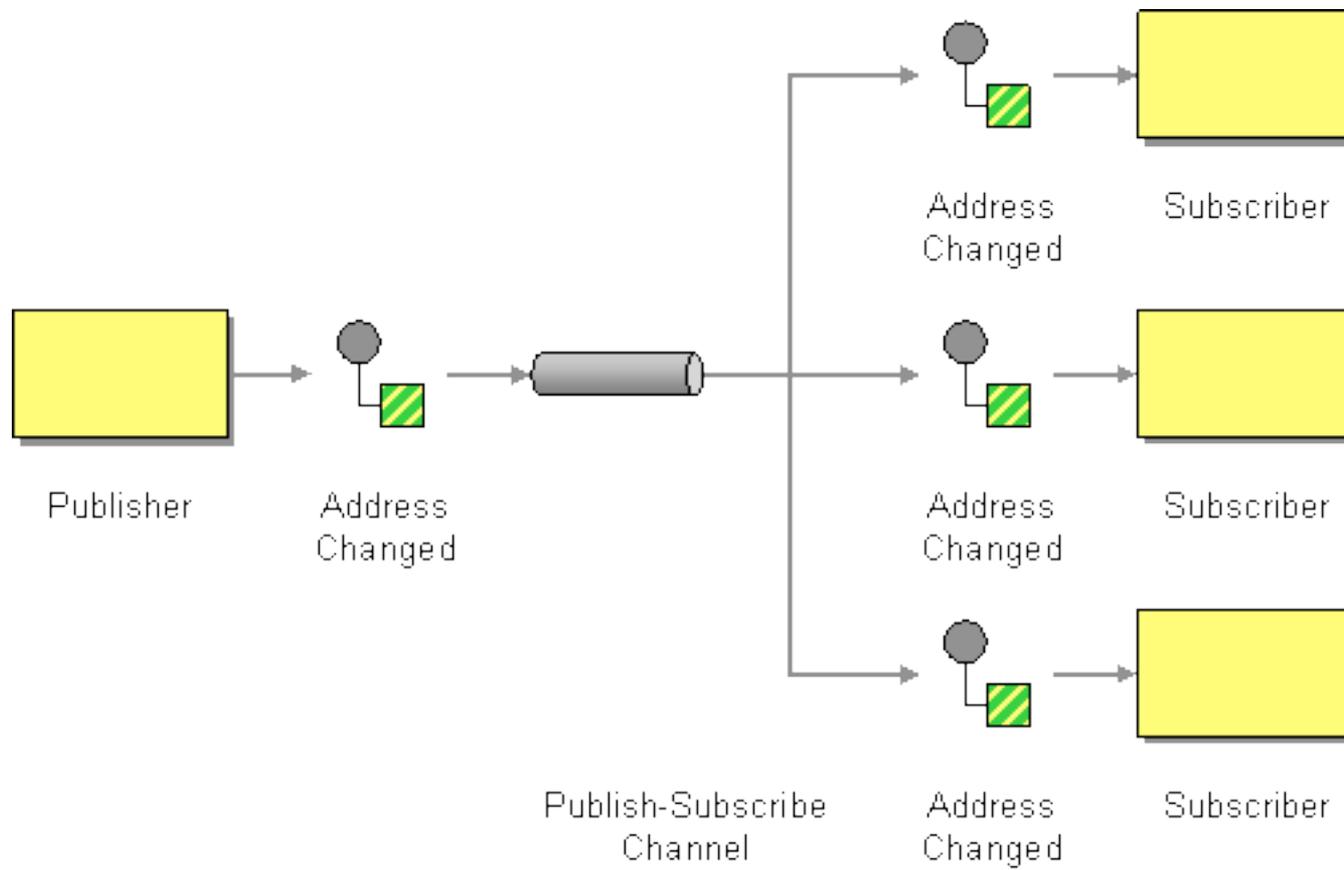
Common Patterns: Message Translator Pattern

- How can systems using different data formats communicate with each other using messaging?



Common Patterns: Publish-Subscribe Pattern

- How can the sender broadcast an event to all interested receivers?



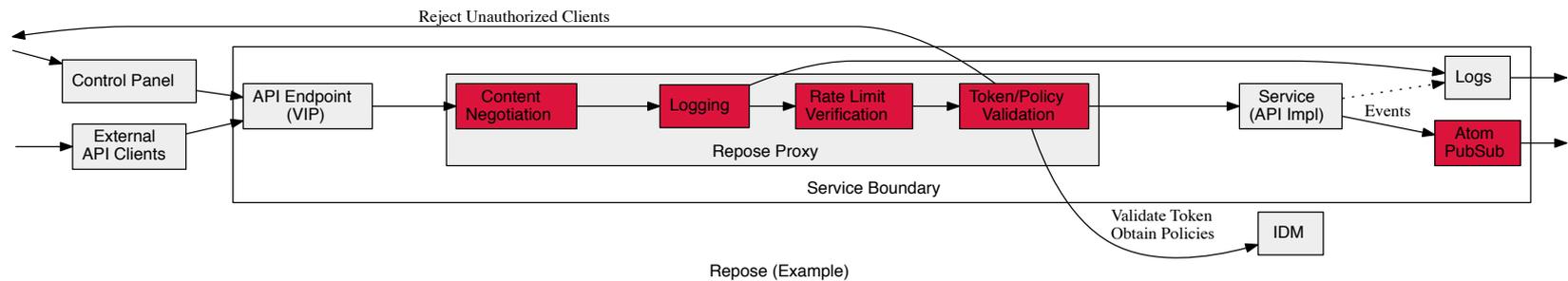
Common Patterns

- API functionality that we want to reuse can be described in terms of these integration patterns.
 - Versioning
 - Rate Limiting
 - Content Negotiation
 - Authentication / Authorization
 - Caching
 - Logging
- Lots of software implement these patterns, very efficiently
 - ESBs implement all of the patterns described in the text
 - But they are very SOAP centric :-)
 - Very inefficient for ReST services
 - There are REST related services that implement some of these patterns
 - apigee, mashery
 - ...but none open or free

What is Repose?

- A platform for solving common integration problems, like a traditional ESB, except:
 - The interface is always REST
 - The protocol is always HTTP
 - Must operate at cloud scale
- Another way to think of it is that Repose is a programmable HTTP proxy
 - AppServer is to an Origin HTTP Server as Repose is to a Reverse HTTP Proxy
- Developers can develop components that live within Repose that handle common integration issues.

HTTP Proxy Approach



The Repose Proxy

- Organizes the API components via configuration
- Provides services to components
 - L1/L2 Cache
 - Secondary Storage
 - Config Update Notifications
 - Etc..

Rate Limiting Example

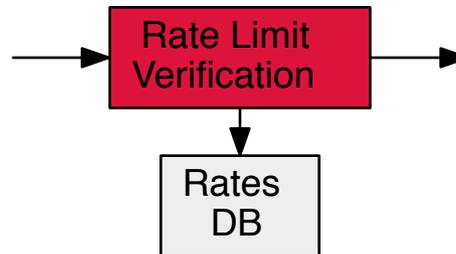
Example: Rate Limiting

- Keep track of requests reject after some limit has been reached
- Rate Limiting seems simple but it presents some challenges



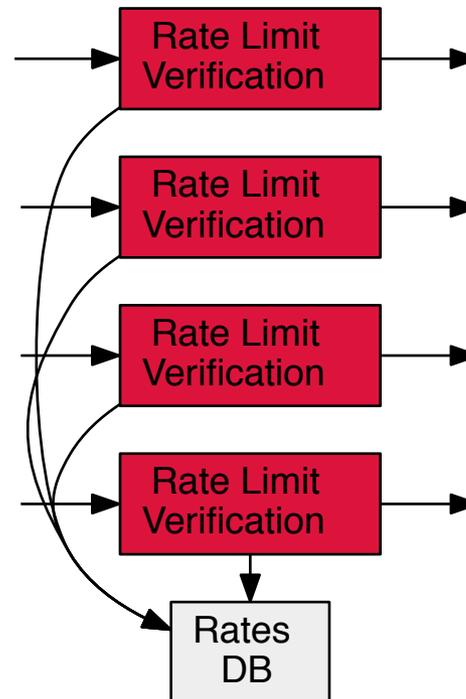
Example: Rate Limiting

- You may want to rate limit per customer.
- That requires some state: How many requests per second so far?
- We could store the current count in a database...



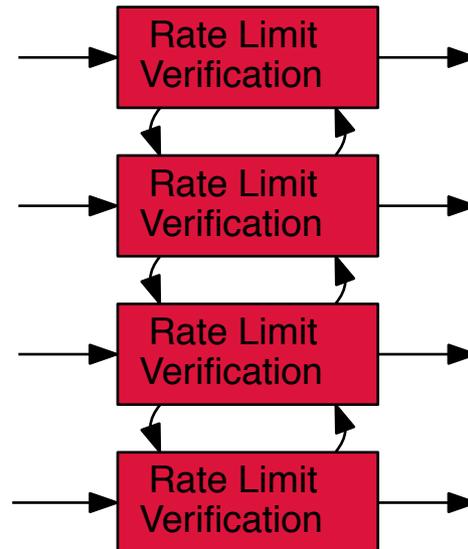
Example: Rate Limiting

- As we scale we begin to introduce a single point of failure.
- May be worth it...depending on how rigid your rate limits are
- May have to scale RatesDB



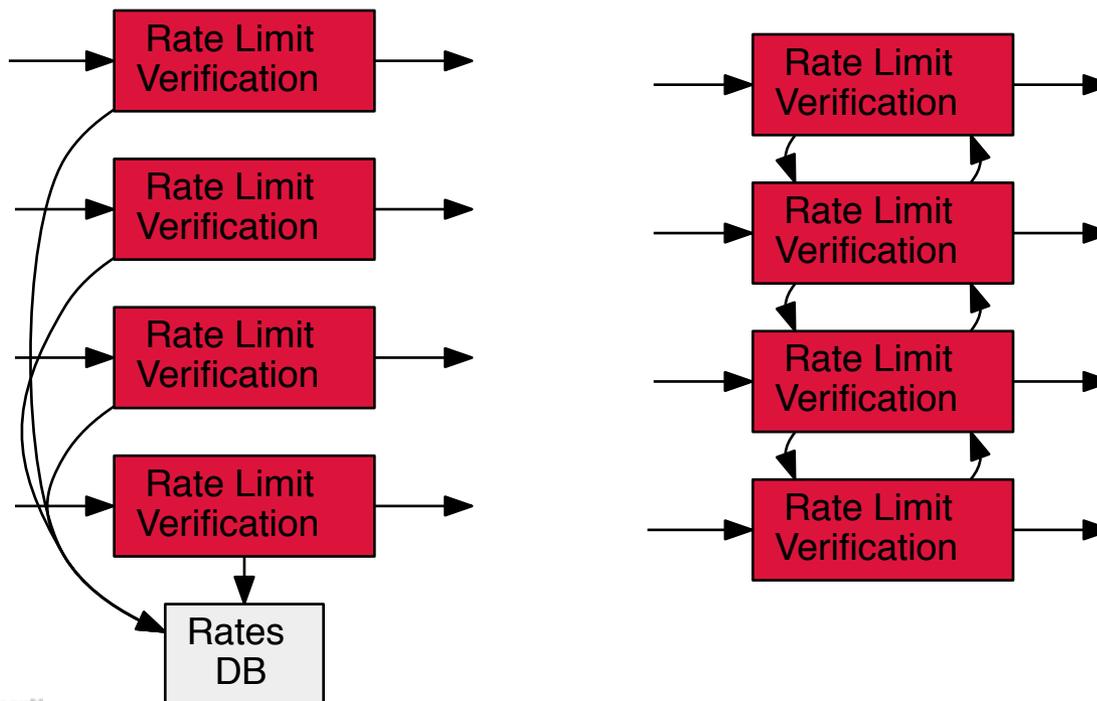
Example: Rate Limiting

- Another approach is to have the rate limits negotiated between nodes
- A DHT may be used: more scalable and fault tolerant
- But you have to deal with eventual consistency?



Example: Rate Limiting

- What's the right approach?
- Depends...
- Repose provides an L2 cache access to components. It's pluggable, so you can achieve both of these architectures...
- Built in simple fault tolerant O(1) DHT is the default...but it's swappable
- You can use either approach without changing the Rate Limit code!



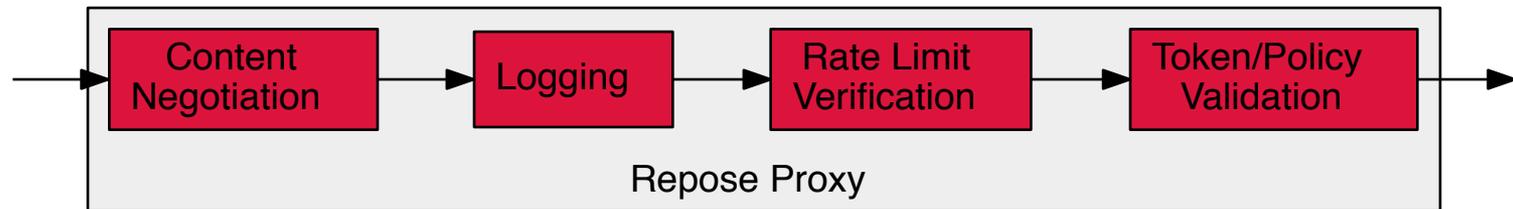
Another Example

experience fanatical support™
WWW.RACKSPACE.COM



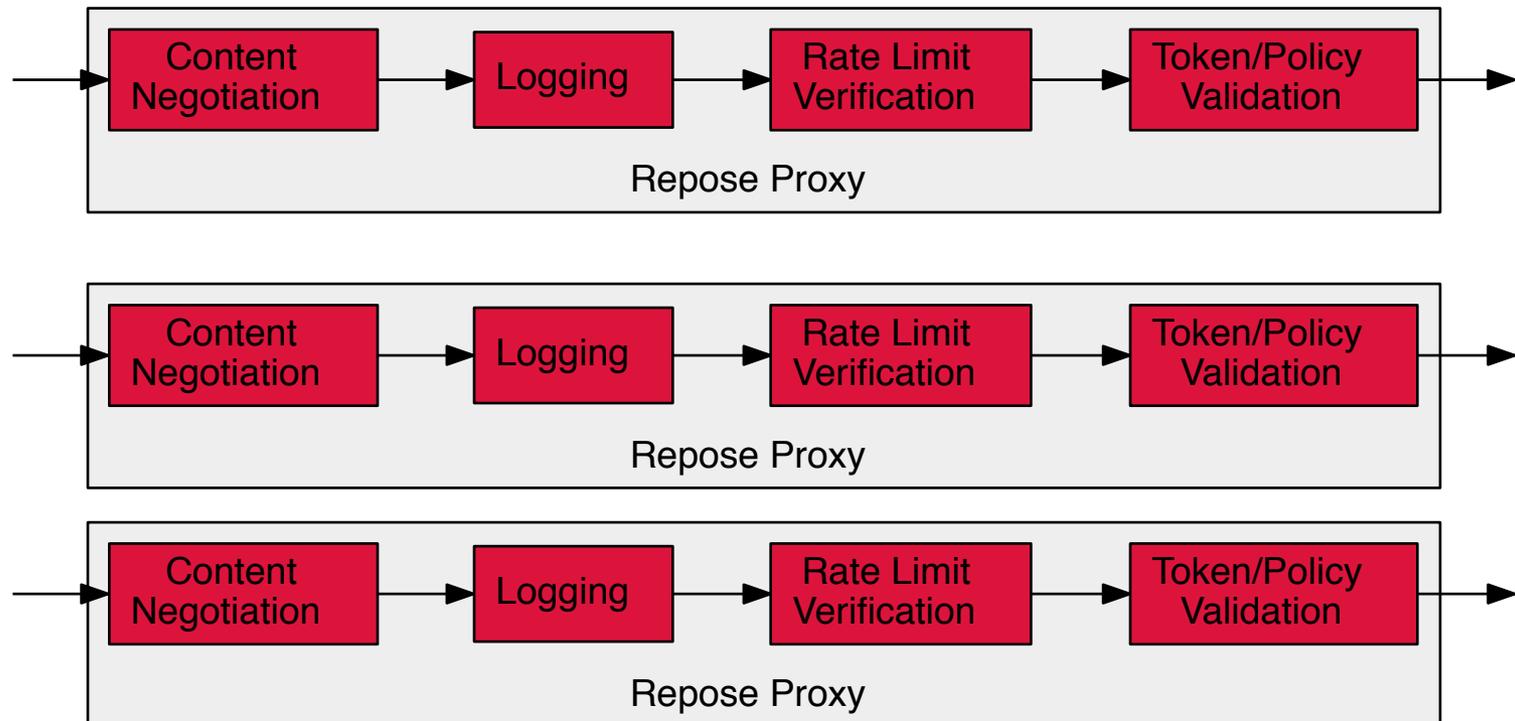
Example

- Say validation of a token requires verifying a digital signature which is computationally expensive.



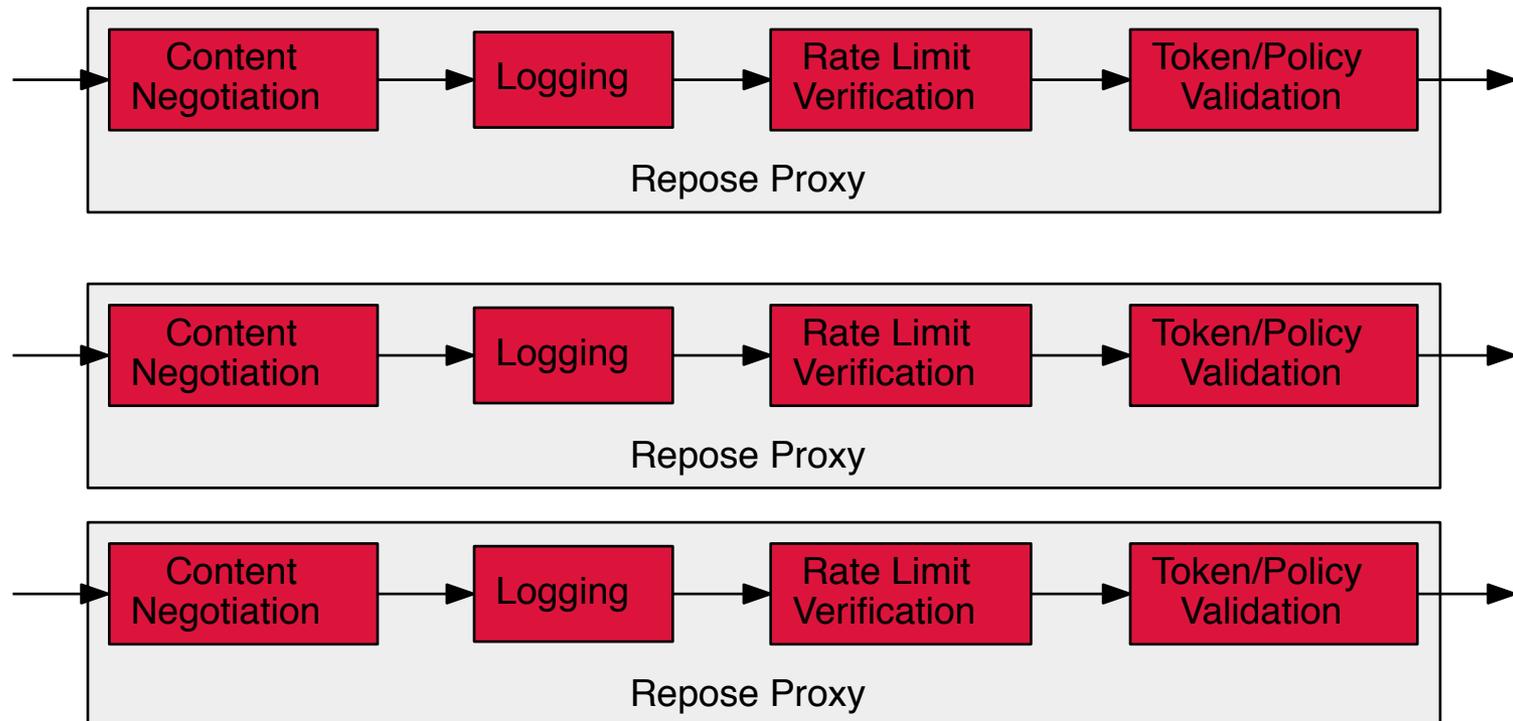
Example

- We could horizontally scale the entire configuration...



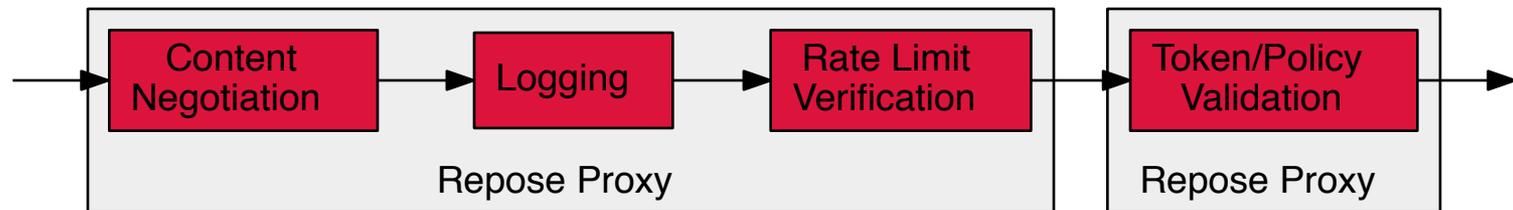
Example

- We can support this, but it adds overhead



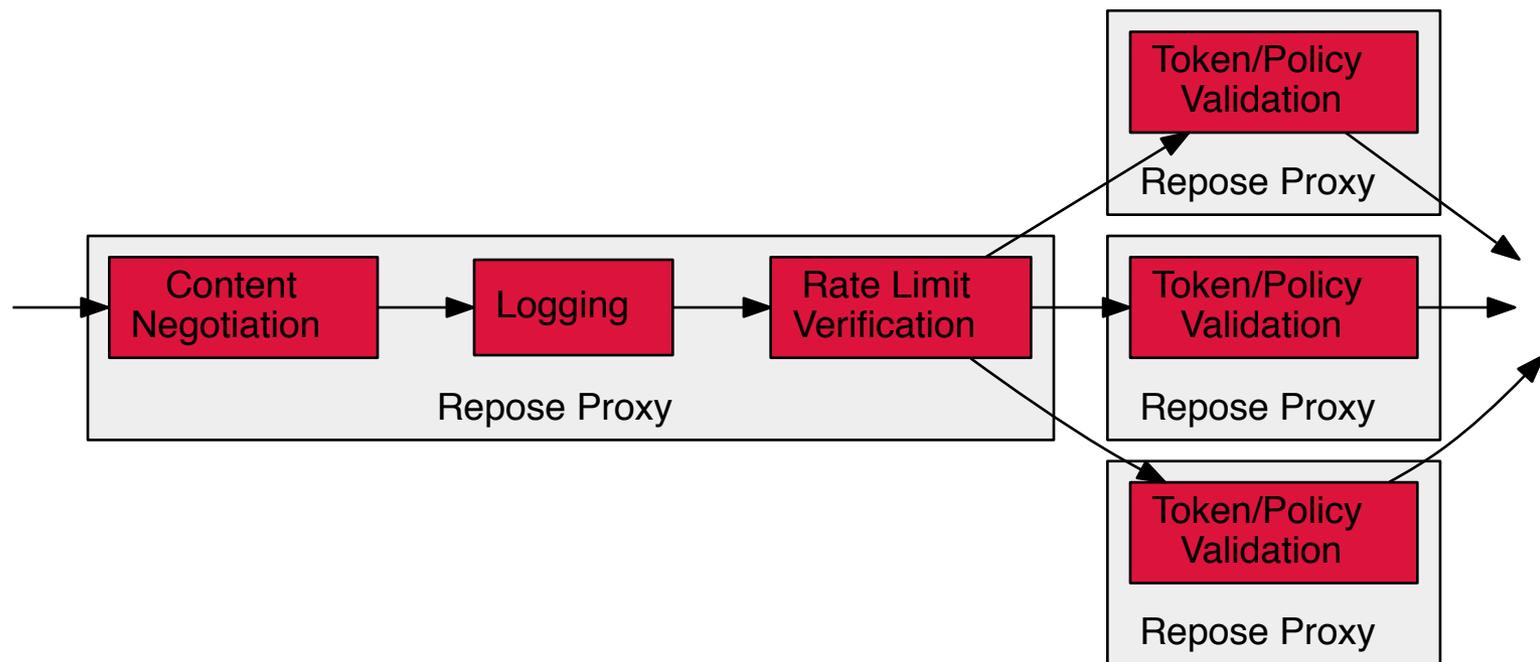
Example

- We can, on the other hand, move token validation to a separate proxy...



Example

- ...and horizontally scale that independently.
- A goal is to allow these type of configuration changes without downtime
 - Similar to the way Zeus and Varnish operate today



What's Available Today?

experience fanatical support™
WWW.RACKSPACE.COM



A Quick Note on API Components

- Most components are custom to OpenStack
 - Rate Limiting
 - Versioning
 - Authentication
 - ...
- These components are written using existing standards
 - Support for JEE6 servlet filters
 - Support for standard JEE artifacts



Geared towards
optimized for OS.

API Components

- A number of components have already been written and are ready for use
 - Normilization
 - If a user ends a request with .json update the request so that it eliminates the .json and sets the Accept header to application/json.
 - This allows APIs to support extensions “for fee”
 - Allows simplifying of Accept headers.
 - Error Code/Response Management
 - How do we ensure that the WADL isn’t violated when you send an error message.
 - This is tough problem.
 - Error code negotiation allows formatting of error messages so that they don’t violate the contract.
 - This involves some configuration
 - Logging
 - Fully customizable API logging a la Apache
 - Supports same basic configuration settings and formats
 - Can detect account number or really any regex from a URI and output it in the log.

API Components

• Versioning

- Given URIs for separate API versions. The component can route between them based on the URI (/v1/, /v2/) an Accept header (application/widgetv1+xml, application/widgetv2+json) or both.
- This allows the creation of perma-links for API resources
- Version information endpoints can automatically be populated so that users can keep up to date with latest versions.
- Additionally, a multiple choice response be given if a user sends a request that doesn't match any one particular version.
 - I know you want server 12345, but do you want it in API version 1? or 2?
- This component is not 100% complete, but we're close

• Rate limiting

- Support for dynamic rate limits based on groups.
- Customizable back off response via error code negotiation
- Multiple groups, group types etc.
- Shared L2 persistent cache of current rates for multiple deployments.

• Auth Support

- Front end for Keystone / Rackspace Auth 1.1
- Supports full validation and token caching
- Allows services to integrate with Auth service “for free”.

API Components

- AuthZ
 - Ensure that a request is allowed according to the EndPoint Catalog
- Root.WAR
 - Deploy the Repose Proxy within a Java APP Server, rather than as an external proxy

Repose Proxy

- An initial version of the Repose proxy
 - Provides config change updates
 - Pluggable caching layer
 - With a plugin for EH-Cache
 - Fully dynamic configuration without downtime of individual components.
- An external proxy container based on Jetty.

What we have planned

experience fanatical support™
WWW.RACKSPACE.COM



What we have planned

- Components:
 - Serialization Conversion
 - JSON/XML
 - Arbitrary conversions XSLT, E4X, etc.
 - Others...
- R&D a more efficient external container
 - Custom Netty / Mina
 - Zeus?

What we have planned

- API Components
 - Operation Filtering (Contract Scope)
 - Introspect a WADL, allow only the ones in the contract.
 - Can be used to will filter Admin calls from the public API.
 - Can be used to compute API coverage
 - Multiple Language Support
 - Define API components in languages other than Java
- More work on Repose Proxy
 - Efficient reconfiguration of changes between Nodes

What we have planned

- On GitHub soon...

Thanks!

