Apache Traffic Server
Extensible Host Resolution

at ApacheCon NA 2014

Speaker

- Alan M. Carroll, Apache Member, PMC
  - Started working on Traffic Server in summer 2010.
  - Implemented
    - Transparency, IPv6, range acceleration, yada yada yada
  - Works for Network Geographics
    - Provides ATS and other development services

Outline

- Current state of things.
- Design and Implementation.
  - Not just an API upgrade.
- Example extensions / use cases.

Current status

HOSTDB / DNS

Resolution Services

- Resolve host names (“FQDN”) to IP addresses.
  - “Fully Qualified Domain Name”
- HostDB caches resolutions for performance.
  - Persistent across process restarts.
- DNS performs DNS queries (packet level)
- Partially handles some additional features
  - Round robin
  - Split DNS
- Handles IPv4, IPv6, SRV to some extent.

Host Resolution

- HTTP state machine creates query for FDQN.
- HostDB does look up and returns data if found.
- Otherwise the query is passed to the DNS and a network query is done.
- HTTP state machine gets raw data, handles server retries and round robin.

What's the point?

- High performance is not possible if doing a DNS query for every transaction.
- Need more control than available from standard OS host resolution calls.
  - Control latency / retries.
  - Non-blocking.
  - May require specialized DNS for Traffic Server.

What's the problem?

- Short answer – it's a mess.
- Resolution logic is spread between transaction state machine and HostDB.
  - Example: had to change SM for TS-1422
- Little modularity inside HostDB.
- Mishmash of data structures for IPv4, IPv6, SRV records segregated by hash fiddling.

More problems

- Very difficult to upgrade or change
  - Logic is hardwired, not accessible via any API
    - This means changes require expertise in ATS core.
  - Few configuration options.
- Fixed sized heap allocation
  - Configured at process start.
  - Must configure size and count (inode problem).
  - Bad things happen if size exceeded.

Doing Better

- Remove address resolution state from HTTP state machine.
- Plugin address provisioning.
  - Customized DNS querying.
  - Other address sources (file/database/YP/etc.).
- Filter and re-order addresses.
- Extensible data associated with addresses or FQDN.
- Remove requirement for core expertise.

Take a bow for the new Resolution

HOST RESOLUTION ARCHITECTURE

Style

- API
  - Minimal.
  - Orthogonal.
  - Consistent.
- No limits on extensions.
- Maintainable.
- Clean separation of framework from function.

Features

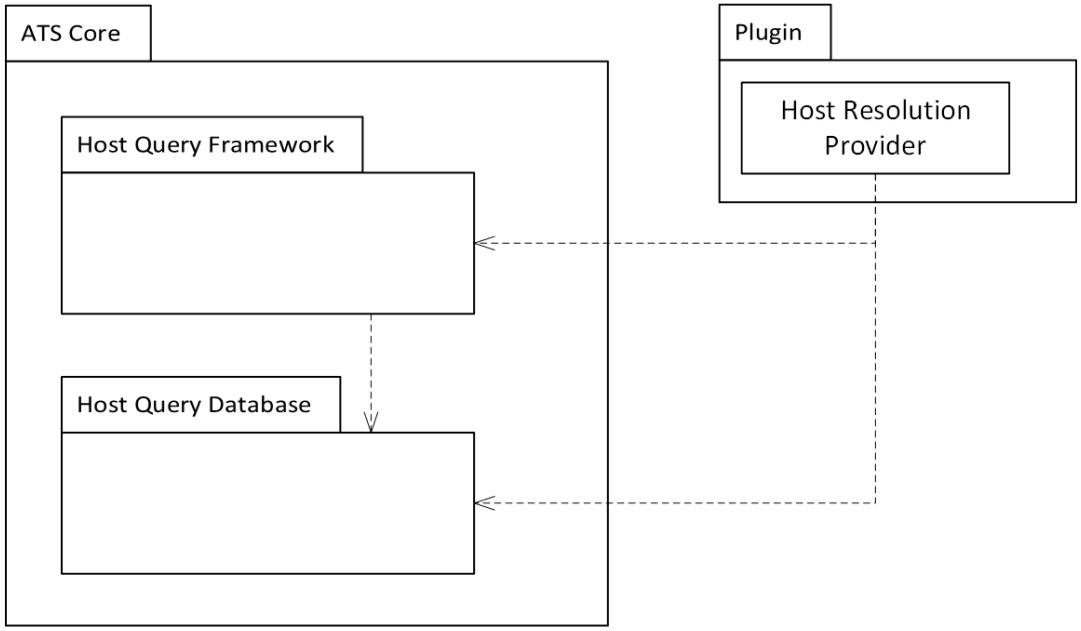
- Plugins for host resolution.
  - Access to external address data.
  - Filtering and control of resolved addresses.
- Simple interface for HTTP state machine
  - Generator / forward iterator style.
- Maintain current functionality, modularized.
- Minimize performance loss.
- Asynchronous.

Design Elements

- Framework – Traffic Server core.
- Host Query Database – Traffic Server core.
- Host Resolution Provider – plugin.
- Host Resolver – rooted tree of providers.

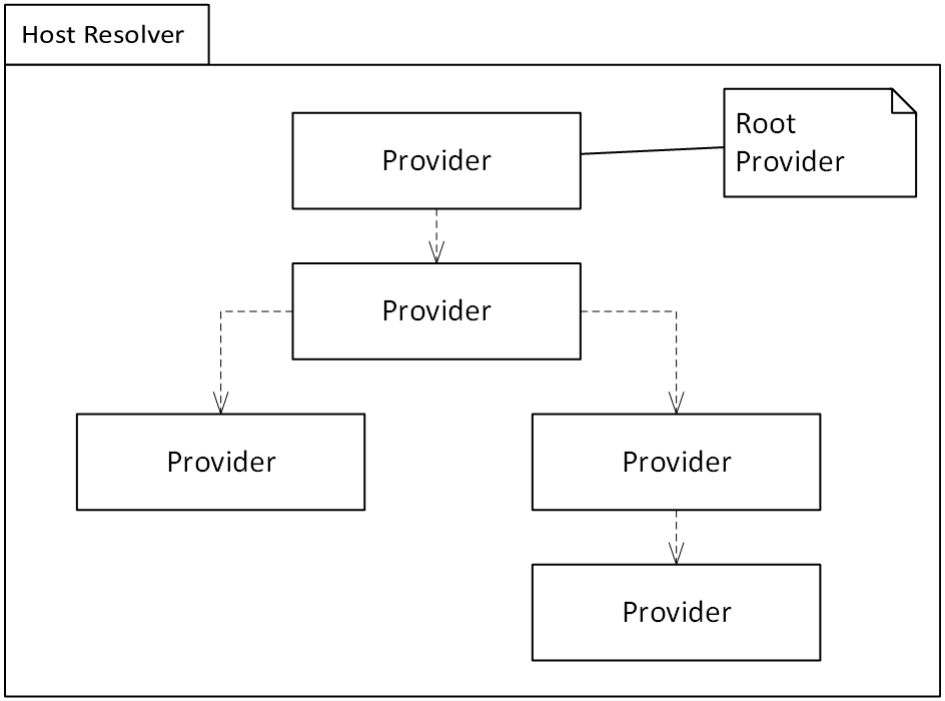
Package Layout

The ATS core has the framework and the HQD. Providers are contained in user created plugins.



Host Resolver

A host resolver is a tree of providers with a root. The root serves as the external interface of the resolver.



Host Query Database

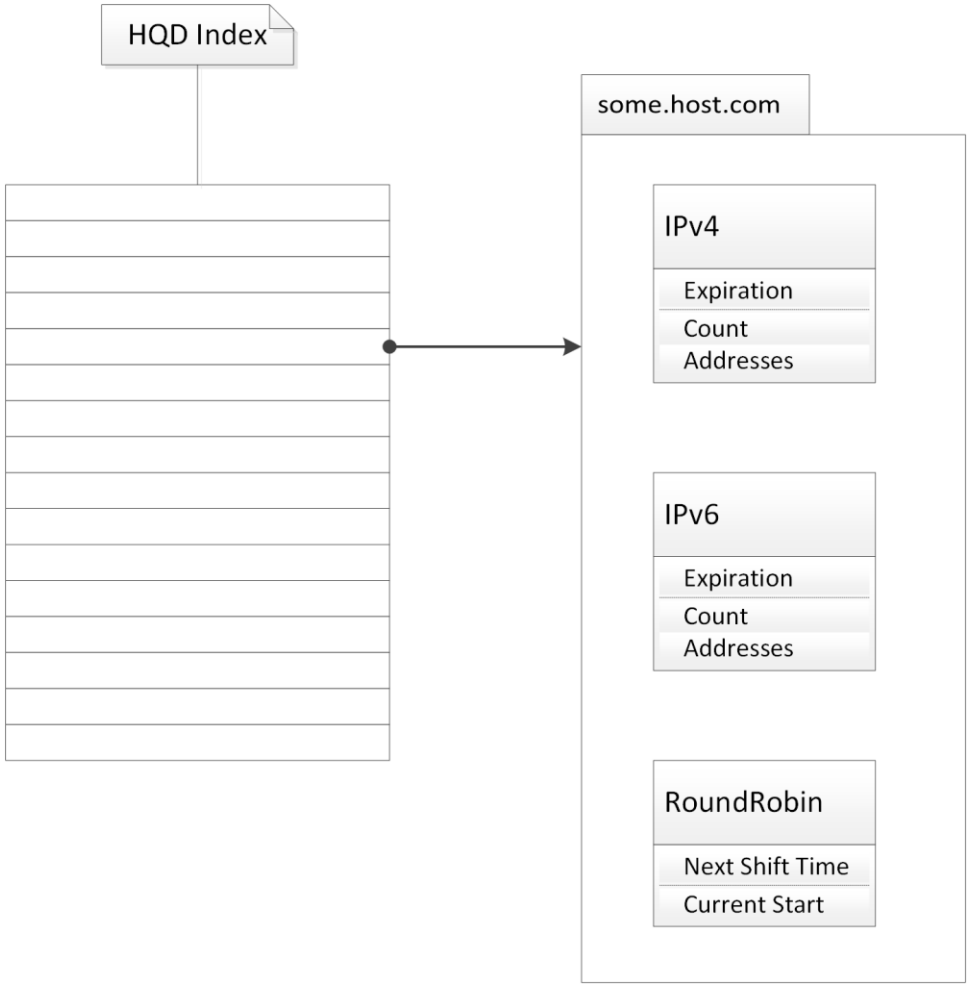
- Indexed by FQDN.
- Record is set of key/value pairs plus fixed metadata.
- Plugins must register keys.
- Values are plugin formatted, opaque to framework.
- Expiration / reaping by framework.

Host Query Database 2

- A process persistent store
  - Each provider can store data in a record.
  - This is similar to what is done now but more elegantly, explicitly, and extensibly.
- A communication channel.
  - Values are cooperative data – no access controls.
  - Providers are expected to share data via the HQD.
    - Core plugins will document value format

HQD Record

Simple hash table mapping from FQDN to a set of key / value pairs.

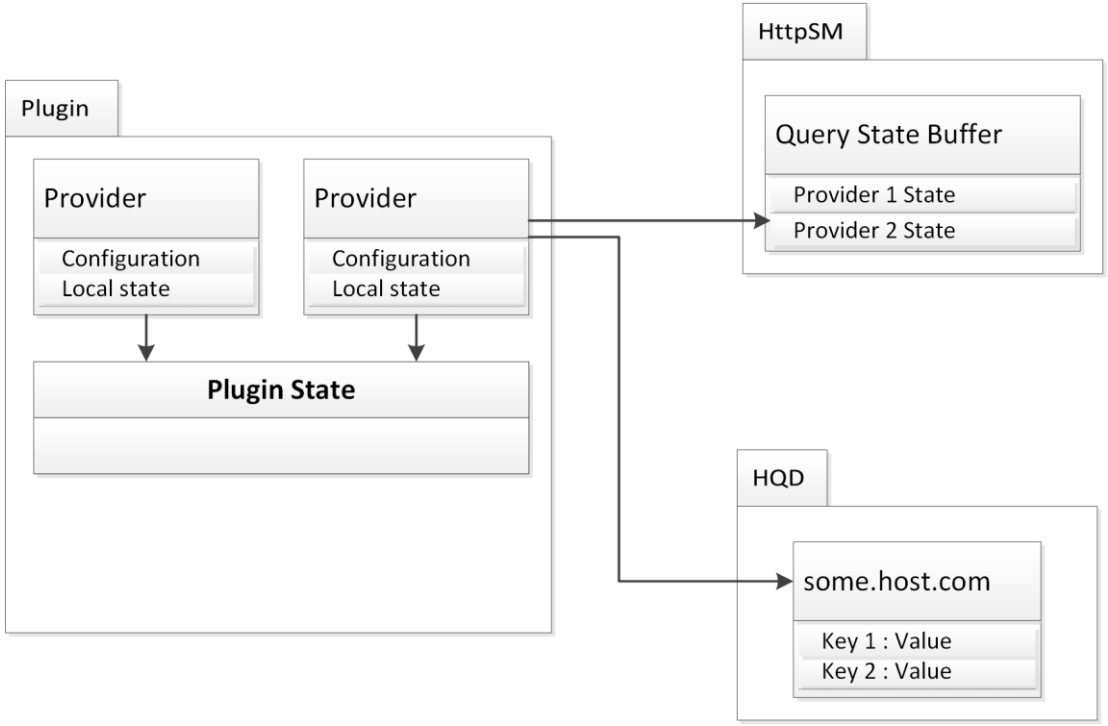


Provider

- Interface between framework and plugin.
  - Framework calls the plugin via provider instance.
- All providers must have the address iterator interface.
- Provider state.
  - Plugin globals.
  - Provider instance.
  - HQD record.
  - Query in HTTP state machine.

State data diagram

Providers can access data in the plugin (dynamic library) and any internally defined local state. Data for the FQDN of the query is available and a fixed amount of space in the HTTP state machine (transaction) instance.



Provider Instance State

- Support multiple instances of the same underlying provider with distinct configuration.
- Expected to be relatively inexpensive to have multiple instances of a provider.

Provider Query state

- Need state for each query.
- Want to avoid allocation, even class allocator.
- Most providers should need a fixed amount of query state.
- Provider must export amount of query state required.

Framework Query State

- Therefore – we can provide a static buffer in the state machine instance for the query.
  - Buffer size build time configurable.
  - Obvious error message if exceeded.
  - Dynamic storage must be allocated.
- Framework handles doling out memory to each provider.
- Framework will call provider for cleanup.

Resolver

- Resolver is a rooted tree of providers.
- HTTP State machine talks to root provider.
- Root provider can pass query to descendents.
- Each provider has complete control of whether any descendent provider is used.
- Providers share data via the HQD.

Query Actions

- State machine initiates query by invoking framework.
- Framework locates the HQD record and passes it and the query to root provider.
- Root provider can provide addresses out of current data or make requests to descendents.
- Framework also sends
  - IP family preference.
  - ATS HTTP Transaction handle.

Query Results

- Root provider streams addresses to HTTP state machine.
- HTTP state machine gets addresses as needed.
  - Provider can defer decisions until asked.
- TBD: does state machine provide feedback to provider on failed addresses?

Asynchronicity

- Provider can return “blocked”.
- Therefore queries forwarded to descendent providers can block.
- Provider can forward multiple blocked queries to different descendent providers.
  - This is how parallelism is done.
- Provider “resume” called for each completion on ancestor providers of blocking provider.

Asynchronicity 2

- Provider that blocks handles continuation and resumption.
- Must notify framework of resume.
- Framework calls standard “resume” method on ancestor providers.
- The “resume” method can return “blocked” to indicate the framework should wait for further continuation completions.

Locking

- Global lock for HQD for table lookup/modify.
- Per record lock for local access.
- Lock is held for query by framework.
- Lock is released if root provider is blocked.

Expiration

- Expiration time for record in fixed metadata.
- Providers can record additional timeouts.
- Providers control whether stale data is used.
- Framework cleans up expired records not in use.
  - Serious issue for forward proxies.

Open HQD

- HQD API is available to any plugin.
- Intended for external address control not directly involved with host resolution.
  - E.g. external updates to propagate to HQD.

Persistence

- Not in standard / default configuration.
- Can be useful for quicker restarts.
  - More useful for reverse proxies.
- Requires either
  - Persistence of HQD key registration values.
  - Key / ID table and conversion on load.

Core Providers

- DNS
- Host file
- Round robin
- Failover
- Split DNS

RESOLVER EXAMPLES

DNS

- Performs DNS queries to an external resolver.
- Stores results in HQD under ‘ipv4’ or ‘ipv6’
- Provides addresses in order of returned record.
- Each provider locked to a specific name server.

Host File

- Loads standard Unix host file to HQD.
- Handles external file update / synchronization.
  - This is its main function.
- Can act as pass through or an unaccessed leaf.

Round Robin

- Cycles HQD address data.
- Will forward to another provider for data.
- Shift by time or query count.

Failover

- Requires descendent DNS providers.
- Monitors external query success.
- Only uses one descendent at a time.
- Shifts among descendents based on availability.

Split DNS

- Requires descendent providers.
- Returns data if found.
- Otherwise selects descendent provider based on split rules.

Gatherer

- Requires DNS provider.
- Requests IPv4 and IPv6 addresses in parallel.
- May also be used to parallel request across multiple DNS servers for performance.

Simple Load Balancer

- Requires descendent providers for addresses.
- Permutes address data based on balancing rules.
  - Client IP address
  - Query counts
  - URL

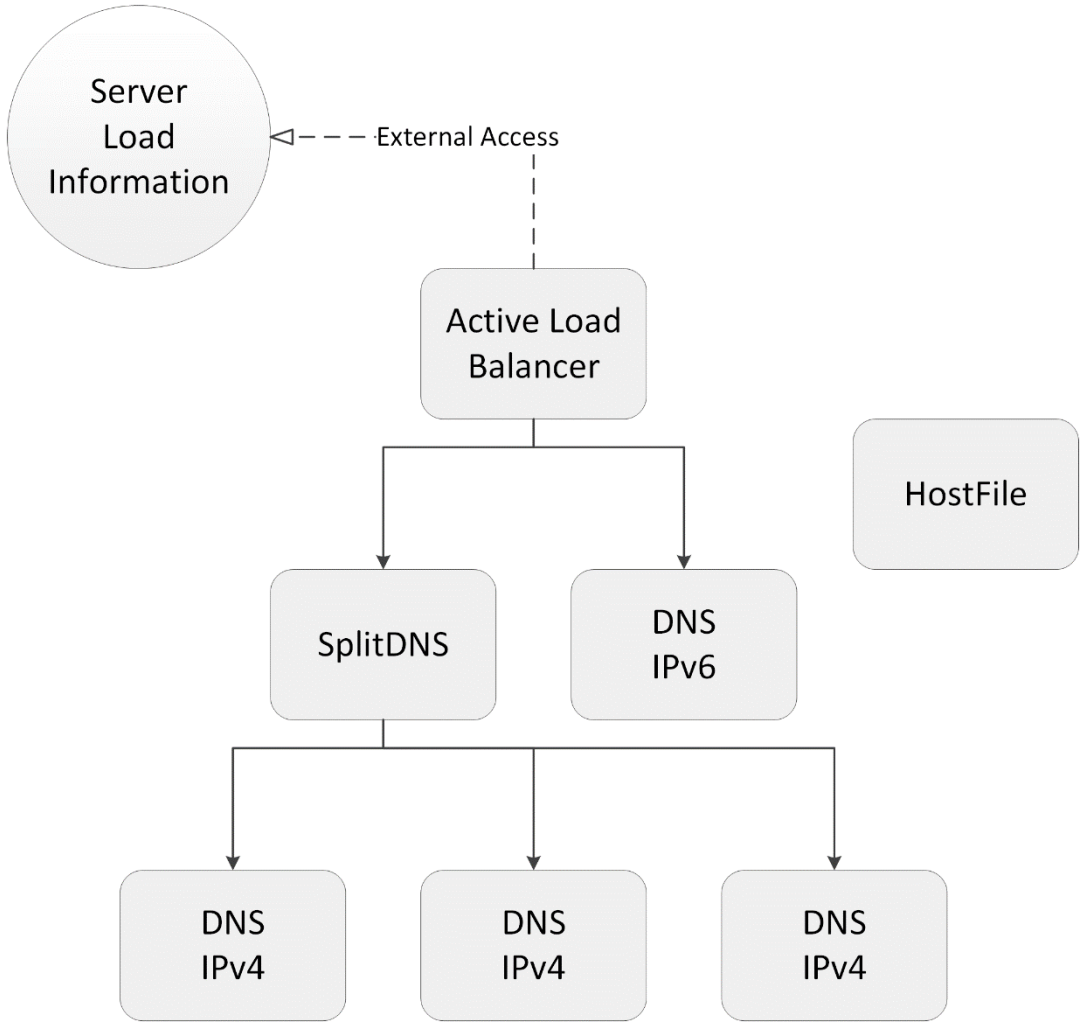
Active Load Balancer

- Requires descendent providers.
  - Or not? Derive from load data?
- Uses external data about current server loads.
  - Store in global state or HQD.
- Iterates addresses in inverse proportion to server loads.

Resolver Structure

HostFile is used to inject the /etc/hosts files but otherwise not directly used.

IPv4 is done in parallel for performance and redundancy.



Walkthrough

Root provider receives query. It uses the HQD handle to check for address information but does not find any. It forwards the query to an IPv6 provider which returns “blocked”. The ALB then forwards the query to the SplitDNS. It decides to forward the query to one of its IPv4 providers. This also returns “blocked” which SplitDNS returns and then the ALB returns.

When the DNS plugin receives a DNS reply it calls the framework to resume query processing. The framework in turns calls “resume” on the appropriate DNS provider and up the ancestors of that provider.

Other

- Fundamental point – all of these can be replaced by user without changes to core.
- Users can build tweaked versions based on core provider plugin code.
  - Much lower barrier to entry to work on plugin code vs. core code.

descendent information address providers

APPENDIX

Current Status

- Basically this slideware.
  - Overall design done
  - Waiting for time / funding to begin implementation.
- Thanks to Openwave for initial design funding.

Open Design Issues

- Feedback from server connect fails
  - Should the provider be told?
  - Does iteration suffice?
- Handling overlapping requests for same FQDN from different transactions.
- Fixed metadata for records
  - Expiration
  - Last modified?
  - Last accessed?

Resources

- ATS has online documentation, a wiki, mailing lists, bug tracker, and IRC channel. Access these via
  - http://trafficserver.apache.org
- Active community – become involved!
- NG Consulting services
  - http://network-geographics.com