NOTE: THE CANONICAL VERSION OF THIS API NOW LIVES AT
https://github.com/bazelbuild/remote-apis/tree/master/build/bazel/remote/asset/v1

Design Proposal: Remote Asset API

EXTERNAL DOCUMENT

Authors: Eric Burnett (ericburnett@google.com), Sander Striker (s.striker@striker.nl)
Updated: 2019/12/11
Status: Approved

Background

The 'Remote Execution' API is a standard API covering file caching and remote action execution, targeted primarily at build tools to enable them to do more work off-host. It is implemented by a growing number of clients and servers.

That API is generally reusable, but does not cover all use-cases build tools wish to offload. One notable use-case not served today is for asset "Downloads", with proposals out from John Millikin (jmillikin@stripe.com) for dependency archive fetching/caching; and from @rahul-nitkkr for adding a remote file system to Bazel.

This proposal attempts to unify the above two into a single API, as well as satisfy my (Eric Burnett's) goals for reusability and future-proofing; it should be evaluated against these goals and the general question of whether consolidation is valuable enough vs the cost of generalizing.

Prior art in this area includes BuildStream which has a concept of a SourceCache that is used for all sources. GetSource could be replaced with the Fetch API here, and the Push API would be used for out-of-band source updates. A fuller fetcher service proposal was also considered earlier, but not pursued at that time.

Overview

This proposal introduces a new API for "`Fetch`ing" assets, to act as a framework for a class of operations including both of the proposals above. This API operates on a few general principles:

- The `FetchBlob` and `FetchDirectory` calls serve to *translate requests* and *make available* referenced assets. It ensures that referenced data is available in the ContentAddressableStore, returning *metadata* to callers.
- Fetches may operate on floating, non-deterministic references, e.g. web URLs or repo refs.
- Responses are required to be fully-specified and deterministic.
- Any given server supporting Fetch may understand or allow only a subset of request types; return only a subset of possible response types.

This proposal also introduces a complementary new API for "`Push`ing" assets, for cases not well suited to having a service that can pull directly from origin.

- The `PushBlob` and `PushDirectory` calls serve to explicitly create associations between references and assets. For servers that do not support implicit fetching of referenced assets, these associations can be used to respond to future client Fetch requests.

For example, this API is suitable for:

- Requesting files be fetched from the web and made available in the CAS.
    - Fetch(url) -> file Digest OR Directory tree
- Mapping from a floating repo reference ("repo X at master") to a concrete and internally-consistent description of the contents
    - Fetch(repo, ref, max_staleness) -> tarball Digest OR Directory tree
- Storing named dependencies for reuse.
    - Push(key, value); Fetch(key) -> file Digest OR Directory tree

This API is *not* suitable for:

- Directly fetching contents, a la `wget`
    - Fetch(url) -/> bytes
- Querying under-specified data

    - Fetch(repo, ref, max_staleness) -/> list of repo-relative files

Content identifiers

The Fetch API provides a mapping from a URI and Qualifiers to Digests.

Multiple URIs may be used to refer to the same content. For example, the same tarball may exist at multiple mirrors and thus be retrievable from multiple URLs. When URLs are used, these should refer to actual content as Fetch API service implementations may choose to fetch the content directly from the origin. For example, the HEAD of a git repository's active branch can be referred to as:

- uri: https://github.com/bazelbuild/remote-apis.git

URNs may be used to strongly identify content, for instance by using the uuid namespace identifier: urn:uuid:f81d4fae-7dec-11d0-a765-00a0c91e6bf6. This is most applicable to named content that is `Push`'d, where the URN serves as an agreed-upon key, but carries no other inherent meaning.

Service implementations may choose to support only URLs, only URNs for Push'd content, only other URIs for which the server and client agree upon semantics of, or any mixture of the above.

Clients may supply one or more URIs in requests, servers can match any one of the supplied URIs. All URIs are expected to indicate the same content, and all URIs share the same Qualifiers.

Qualifiers

Qualifiers are a complement to the URI(s), optionally used to identify subresources like branches or subdirectories, or additional constraints relevant to the fetch. For example, a particular subdirectory of a particular branch of a git repository may be expressed as:

- uri: https://github.com/bazelbuild/remote-apis.git

qualifier:

- key: directory
  value: build/bazel/remote/execution/v2
- key: scm.branch
  master

Qualifiers are optionally used to identify unstable content at a URL. For example, a git repository at a certain ref of e.g. e77c4eb could be referenced as:

- uri: https://github.com/bazelbuild/remote-apis.git
- uri: https://git-mirror.example.com/bazelbuild/remote-apis.git

qualifier:

- key: git.ref

value: e77c4eb

Another example would be a tarball, where we qualify the content by a Subresource Integrity checksum.

- uri: https://github.com/bazelbuild/bazel/archive/1.2.0.tar.gz
qualifier:
  - key: checksum.sri
    value: sha384-6NmGPKYyMNRmjZe6r+7kpAy06QYOEbHg31zhc51IxeeJ8qsqww9w/99Vd6n9W3xj

In all cases, the server *MUST* return contents that match the given qualifiers. So for example, it would be a spec violation to accept a Fetch request with Subresource Integrity checksum provided and then to return content that does not match said checksum. Note that it is not required that clients make fully-specified requests (see Overview), only that servers respond with *a* matching response.

A few additional clarifications:

- When submitting fetch requests qualifier keys are expected to be unique. Semantics of repeated keys are unspecified at this time.
- Qualifiers may be present in any order.
- No attempt is made to distinguish between “Standard” and “Nonstandard” qualifiers. In this we follow the guidance of https://tools.ietf.org/html/rfc6648. It is anticipated that a standard set of keys and associated value semantics will be developed as this proposal lands, and added to the documentation as appropriate.
- In cases where the semantics of the request are not immediately clear from the URL and/or qualifiers - e.g. dictated by URL scheme - it is *recommended* to use an additional qualifier to remove the ambiguity. The key ‘resource_type’ is recommended for this purpose.
- For Push’d assets with qualifiers, the push originator *MUST* ensure the qualifiers match the content they’re pushing.
- Transitively, for Fetch’s of content that were previously Push’d, servers may support either exact-match or subset-match of Qualifiers. This allows servers to return content including Qualifiers that the server itself doesn’t understand and cannot validate.

Design

```
service Fetcher {
  // These methods all do some variant of resolving or fetching
  // referenced assets, making them available to the caller and other consumers in
  // the [ContentAddressableStorage].
  //
  // Servers *MAY* fetch content that they do not already have cached, for any URLs
```

```
  // they support.
  // Servers *SHOULD* ensure that referenced files are present in the CAS at the
  // time of the response, and (if supported) that they will remain available for a
  // reasonable period of time. In the event that a client receives a reference
  // to content that is no longer present, it *MAY* re-issue the request with
  // [FetchRequest.oldest_content_accepted] set to a more recent timestamp than
  // the original attempt, to induce a re-fetch from origin.
  // Servers *MAY* cache fetched content and reuse it for subsequent requests,
  // subject to FetchMetadata.oldest_content_accepted.
  // Servers *MAY* support the complementary [Pusher] API and allow content to
  // be directly inserted for use in future fetch responses.
  // Servers *MUST* ensure Fetch'd content matches all the specified
  // [FetchRequest.qualifiers], except in the case of previously Push'd resources,
  // for which the server *MAY* trust the pushing client to have set the
  // qualifiers correctly, without validation.
  // Servers not implementing the complementary [Pusher] API *MUST* reject
  // requests containing qualifiers it does not support.
  //
  // Errors handling the requested assets will be returned as gRPC Status errors
  // here; errors outside the server's control will be returned inline in the
  // FetchResponse.status field (see comment there for details).
  // The possible RPC errors include:
  // * `INVALID_ARGUMENT`: One or more arguments were invalid, such as a qualifier
  //     that is not supported by the server.

  // * `RESOURCE_EXHAUSTED`: There is insufficient quota of some resource to
        perform the requested operation. The client may retry after a delay.
  // * `UNAVAILABLE`: Due to a transient condition the operation could not be
  //     completed. The client should retry.
  // * `INTERNAL`: An internal error occurred while performing the operation. The
  //     client should retry.
  // * `DEADLINE_EXCEEDED`: The fetch could not be completed within the given
  //     RPC deadline. The client should retry for at least as long as the value
  //     provided in [request.timeout].
  //
  // In the case of unsupported qualifiers, the server *SHOULD* additionally
  // send a [BadRequest][google.rpc.BadRequest] error detail where, for each
  // unsupported qualifier, there is a `FieldViolation` with a `field` of
  // `qualifiers.key` and a `description` of `"{qualifier}" not supported`
  // indicating the name of the unsupported qualifier.
  rpc FetchBlob(FetchBlobRequest) returns (FetchBlobResponse)
  rpc FetchDirectory(FetchDirectoryRequest) returns (FetchDirectoryResponse)
}


// Qualifiers are used to disambiguate or sub-select content that shares a URI.
// This may include specifying a particular commit or branch, in the case of URIs
// referencing a repository; they could also be used to specify a particular
// subdirectory of a repository or tarball. Qualifiers may also be used to ensure
// content matches what the client expects, even when there is no ambiguity to be
// had - for example, a qualifier specifying a checksum value.
message Qualifier {
  // The "key" of the qualifier, for example "resource_type" or "scm.branch". No
```

```
  // separation is made between ‘standard’ and ‘nonstandard’ keys, in accordance
  // with https://tools.ietf.org/html/rfc6648, however implementers *SHOULD* take
  // care to avoid ambiguity.
  string key = 1;
  // The “value” of the qualifier. Semantics will be dictated by the key.
  string value = 2;
}

message FetchBlobRequest {
  string instance_name = 1;

  // The timeout for the underlying fetch, if content needs to be retrieved from
  // origin. This value is allowed to exceed the RPC deadline, in which case the
  // server *SHOULD* keep the fetch going after the RPC completes, to be made
  // available for future Fetch calls.
  // If this timeout is exceeded on an attempt to retrieve content from origin
  // the client will receive DEADLINE_EXCEEDED in [FetchResponse.status].
  google.protobuf.Duration timeout = 2;
  // The oldest content the client is willing to accept, as measured from the
  // time it was Push’d or when the underlying retrieval from origin was started.
  // Upon retries of Fetch requests that cannot be completed within a single RPC,
  // clients *SHOULD* provide the same value for subsequent requests as the
  // original, to simplify combining the request with the previous attempt.
  google.protobuf.Timestamp oldest_content_accepted = 3;

  // The URI(s) of the content to fetch. These may be resources that the server can
  // directly fetch from origin, in which case multiple URIs *SHOULD* represent
  // the same content available at different locations (such as an origin and
  // secondary mirrors). These may also be URIs for content known to the server
  // through other mechanisms, e.g. pushed via the [Pusher] API.
  repeated string uris = 4;
  // Qualifiers sub-specifying the content to fetch - see comments on [Qualifier].
  // The same qualifiers apply to all URIs.
  repeated Qualifier qualifiers = 5;
}

Message FetchDirectoryRequest {
  string instance_name = 1;

  // The timeout for the underlying fetch, if content needs to be retrieved from
  // origin. This value is allowed to exceed the RPC deadline, in which case the
  // server *SHOULD* keep the fetch going after the RPC completes, to be made
  // available for future Fetch calls.
  // If this timeout is exceeded on an attempt to retrieve content from origin
  // the client will receive DEADLINE_EXCEEDED in [FetchResponse.status].
  google.protobuf.Duration timeout = 2;
  // The oldest content the client is willing to accept, as measured from the
  // time it was Push’d or when the underlying retrieval from origin was started.
  // Upon retries of Fetch requests that cannot be completed within a single RPC,
  // clients *SHOULD* provide the same value for subsequent requests as the
  // original, to simplify combining the request with the previous attempt.
  google.protobuf.Timestamp oldest_content_accepted = 3;
```

```
  // The URI(s) of the content to fetch. These may be resources that the server can
  // directly fetch from origin, in which case multiple URIs *SHOULD* represent
  // the same content available at different locations (such as an origin and
  // secondary mirrors). These may also be URIs for content known to the server
  // through other mechanisms, e.g. pushed via the [Pusher] API.
  repeated string uris = 4;
  // Qualifiers sub-specifying the content to fetch - see comments on [Qualifier].
  // The same qualifiers apply to all URIs.
  repeated Qualifier qualifiers = 5;
}


message FetchBlobResponse {
  // If the status has a code other than `OK`, it indicates that the operation was
  // unable to be completed for reasons outside the servers' control.
  // The possible fetch errors include:
  // * `DEADLINE_EXCEEDED`: The operation could not be completed within the
  //    specified timeout.
  // * `NOT_FOUND`: The requested asset was not found at the specified location.
  // * `PERMISSION_DENIED`: The request was rejected by a remote server, or
  //     requested an asset from a disallowed origin.
  // * `ABORTED`: The operation could not be completed, typically due to a failed
         consistency check.
  google.rpc.Status status = 1;

  // The uri from the request that resulted in a successful retrieval, or from
  // which the error indicated in `status` was obtained.
  string uri = 2;
  // Any qualifiers known to the server and of interest to clients.
  repeated Qualifier qualifiers = 3;

  // A minimum timestamp the content is expected to be available through.
  // Servers *MAY* omit this field, if not known with confidence.
  google.protobuf.Timestamp expires_at = 4;

  // The result of the fetch, if the status had code `OK`.
  // The digest of the file's contents, available for download through the CAS.
  Digest blob_digest = 5;
}

message FetchDirectoryResponse {
  // If the status has a code other than `OK`, it indicates that the operation was
  // unable to be completed for reasons outside the servers' control.
  // The possible fetch errors include:
  // * `DEADLINE_EXCEEDED`: The operation could not be completed within the
  //    specified timeout.
  // * `NOT_FOUND`: The requested asset was not found at the specified location.
  // * `PERMISSION_DENIED`: The request was rejected by a remote server, or
  //     requested an asset from a disallowed origin.
  // * `ABORTED`: The operation could not be completed, typically due to a failed
         consistency check.
  google.rpc.Status status = 1;
```

```
  // The uri from the request that resulted in a successful retrieval, or from
  // which the error indicated in `status` was obtained.
  string uri = 2;
  // Any qualifiers known to the server and of interest to clients.
  repeated Qualifier qualifiers = 3;

  // A minimum timestamp the content is expected to be available through.
  // Servers *MAY* omit this field, if not known with confidence.
  google.protobuf.Timestamp expires_at = 4;

  // The result of the fetch, if the status had code `OK`.
  // the root digest of a directory tree, suitable for fetching via
  // [ContentAddressableStorage.GetTree].
  Digest root_directory_digest = 5;
}

service Pusher {
  // The Pusher service is complementary to the Fetcher, and allows for
  // associating contents of URLs to be returned in future Fetcher API calls.
  //
  // These APIs associate the identifying information of a resource, as indicated
  // by URI and optionally Qualifiers, with content available in the CAS. For
  // example, associating a repository url and a commit id with a Directory Digest.
  // Servers *SHOULD* only allow trusted clients to associate content, and *MAY*
  // only allow certain URIs to be pushed.
  // Clients *MUST* ensure associated content is available in CAS prior to pushing.
  // Clients *MUST* ensure the Qualifiers listed correctly match the contents, and
  // Servers *MAY* trust these values without validation.
  // Fetch servers *MAY* require exact match of all qualifiers when returning
  // content previously pushed, or allow fetching content with only a subset of
  // the qualifiers specified on Push.
  // Clients can specify expiration information that the server *SHOULD*
  // respect. Subsequent requests can be used to alter the expiration time.
  //
  // A minimal compliant Fetcher implementation may support only Push'd content and
  // return `NOT_FOUND` for any resource that was not pushed first. Alternatively,
  // a compliant implementation may choose to not support Push and only return
  // resources that can be Fetch'd from origin.
  //
  // Errors will be returned as gRPC Status errors.
  // The possible RPC errors include:
  // * `INVALID_ARGUMENT`: One or more arguments to the RPC were invalid.
  // * `RESOURCE_EXHAUSTED`: There is insufficient quota of some resource to
  //     perform the requested operation. The client may retry after a delay.
  // * `UNAVAILABLE`: Due to a transient condition the operation could not be
  //     completed. The client should retry.
  // * `INTERNAL`: An internal error occurred while performing the operation. The
  //     client should retry.
  rpc PushBlob(PushBlobRequest) returns (PushBlobResponse)
  rpc PushDirectory(PushDirectoryRequest) returns (PushDirectoryResponse)
}

message PushBlobRequest {
```

```
  string instance_name = 1;

  // The URI(s) of the content to associate. If multiple URIs are specified, the
  // pushed content will be available to fetch by specifying any of them.
  repeated string uris = 2;
  // Qualifiers sub-specifying the content that is being pushed - see comments on
  // [Qualifier]. The same qualifiers apply to all URIs.
  repeated Qualifier qualifiers = 3;
  // A time after which this content should stop being returned via Fetch.
  // Servers *MAY* expire content early, e.g. due to storage pressure.
  google.protobuf.Timestamp expire_at = 4;

  // The blob to associate.
  Digest blob_digest = 5;

  // Referenced blobs or directories that need to not expire before expiration
  // of this association, in addition to `blob_digest` itself.
  // These fields are hints - clients *MAY* omit them, and servers *SHOULD* respect
  // them, at the risk of increased incidents of Fetch responses indirectly
  // referencing unavailable blobs.
  repeated Digest references_blobs = 6;
  repeated Digest references_directories = 7;
}

message PushBlobResponse {

}

message PushDirectoryRequest {
  string instance_name = 1;

  // The URI(s) of the content to associate. If multiple URIs are specified, the
  // pushed content will be available to fetch by specifying any of them.
  repeated string uris = 2;
  // Qualifiers sub-specifying the content that is being pushed - see comments on
  // [Qualifier]. The same qualifiers apply to all URIs.
  repeated Qualifier qualifiers = 3;
  // A time after which this content should stop being returned via Fetch.
  // Servers *MAY* expire content early, e.g. due to storage pressure.
  google.protobuf.Timestamp expire_at = 4;

  // Directory to associate
  Digest root_directory_digest = 5;

  // Referenced blobs or directories that need to not expire before expiration
  // of this association, in addition to `root_directory_digest` itself.
  // These fields are hints - clients *MAY* omit them, and servers *SHOULD* respect
  // them, at the risk of increased incidents of Fetch responses indirectly
  // referencing unavailable blobs.
  repeated Digest references_blobs = 6;
  repeated Digest references_directories = 7;

}
```

```
message PushDirectoryResponse {

}
```

Examples

Bazel `Remote Repository Cache` replacement

Requires:

- A server implementing `Fetcher.FetchBlob`.
- Bazel configuration allowing this to be used as a remote repository cache. (The linked proposal suggests `--remote_repository_cache=repocache.corp.example.com:443`; that would work equally well here.

Extension:

- A server implementing `Fetcher.FetchDirectory`.
- Clients to *optionally* leverage it rather than downloading+unpacking the tarball themselves.

Not currently supported:

- The Remote Repository Cache proposal allows for requests based purely on `canonical_id`, while we no longer have a `canonical_id` field in this proposal. Instead a URN or otherwise guaranteed unique URI could be used, or the canonical_id can be provided to supporting servers as a Qualifier.
- The Remote Repository Cache proposal allows for requests based purely on `integrity`, which we don't support. Instead a URN or otherwise guaranteed unique URI could be used, or the integrity checksum can be provided to supporting servers as a Qualifier.

Bazel `RemoteFileSystem` replacement

Requires:

- Standardizing qualifiers suitable for repositories
- A server implementing `Fetcher.FetchDirectory`.
- Bazel configuration allowing this to be used as a mounted filesystem. (The linked proposal suggests flags; I do not propose changes to those here.)

BuildStream `SourceCache` replacement

Requires:

- Standardizing qualifiers suitable for repositories
- A server implementing `Fetcher.FetchDirectory`.
- BuildStream expressing identify of raw sources in terms of (URI, contextual_id) tuples, and using `Fetcher.FetchDirectory` to obtain the sources, falling back to retrieving raw sources directly.
- BuildStream encoding source cache keys as URNs and using this instead of its current API. For example: urn:buildstream:source:41cf6794ba4200b839c53531555f0f3998df4cbb01a4d5cb0b94e3ca5e23947d
- A server implementing `Pusher.Push`.
- BuildStream, potentially a different instance, using `Pusher.Push` to populate the `SourceCache`.

BuildStream `ArtifactCache` replacement

Requires:

- A server supporting `Fetcher.FetchBlob` and `Pusher.PushBlob` with URNs
- BuildStream encoding artifact cache keys as URNs and using this instead of its current API. For example: urn:buildstream:artifact:c7c5c1d70c5dec4416ab6158afd0b223ef40c29b1dc1f97ed9428b94d4cadb1c

Open Discussion Topics

2. Should this API be async?

In theory fetch requests could take a while - minutes certainly. This API at present allows RPCs to be timed out and requests to be deduped together upon re-issue; this is different from how Execute allows explicit resumption of a past request. Would it be better to make Fetch async consistently with Execute?

3. Is anything needed in Capabilities?

Capabilities in V2 are relatively limited, and larger changes are to be considered for V3. I propose adding nothing to capabilities with the initial version of this API (in V2), and revisiting for later - possibly V3 - whether capabilities could be useful.

4. Does 'URL + Qualifiers' suffice, or should we have other top-level fields?

I (ericburnett) propose we stick with only URL as a privileged field, and use Qualifiers for everything else - even standard keys. There are a few reasons for this approach:

a. One example of prior art is the HTTP spec. At the top level, a HTTP request is composed of <method, resource, version, headers, body>. E.g.

```
GET / HTTP/1.1
Host: example.com
Foo: Bar

<Body goes here>
```

There are a few lessons for us here. The first is that standard strings and associated value semantics has worked out for HTTP across hundreds of different headers and use-cases. That's a fairly strong argument-by-example that it can be made to scalably work.

The second is in https://tools.ietf.org/html/rfc6648, which goes into detail on why segregating standardized and unstandardized parameters has "caused more problems than it solves" in scaled deployment, and defines a Best Practice *not* to do so. This is applicable both in terms of what to do with a set of string/value pairs (Qualifiers), if everything is defined in terms of them, as well as an argument against an approach like "*Standard keys become top-level fields of their own; non-standard keys go into Qualifiers*".

b. Assuming we don't try to segregate *all* 'standard' vs nonstandard keys, and don't try to make *every* key strongly typed (the original proposal included the use of Any protos to approach this, and was generally rejected) there remains a question still of whether any other fields should be pulled out from Qualifiers to the top level. In this case, I think the bar should be quite high - any such field should be broadly applicable, *and* ideally should *need* to be special. URL is a good example of both - we say every request requires a URL, and we allow multiple URLs to be specified.

I'm not currently aware of any other field that meets such a stringent bar. 'Integrity' for

example stands a good chance of being broadly useful, but certainly isn't required for *all* use-cases, and isn't really harmed by being included within Qualifiers instead of at the top level. Similarly, git hash will be likely very popular in the near term, but doesn't really *need* to be specialized.