Arvados - Story #8997

Keep: rethink role of "signature tokens"

04/15/2016 08:27 AM - Peter Grandi

Status:	Closed	**Start date:**	04/15/2016
Priority:	Normal	**Due date:**	
Assigned To:		**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			

Description

During last night around 4AM I woke up and suddenly I understood (or I think I did) the role of "signature tokens" described in:

https://dev.arvados.org/projects/arvados/wiki/Keep_server#Permission

especially in relationship to block lifetimes as per issues:

https://dev.arvados.org/issues/8993
https://dev.arvados.org/issues/8878
https://dev.arvados.org/issues/8867

So my current understanding is...

In Keep block liveness is reachability from a server-side manifest or a client-side signature token. Similar to UNIX directory entries+inodes for manifests or file descriptors for files, where "authorization" in the form of having a file descriptor implies liveness.

But permissions tokens are client-side and persistent capabilities, even if time-limited, unlike file descriptors that are server-side and disappear on reboot (which is a raw form of garbage collection) capabilities.

Since the Data Manager cannot trace signature tokens, which may be anywhere, only manifests, it must make worst-case assumptions on them, both as to their existence and expiry times, which implies that the block signature TTL must be monotonically increasing, which is hard to ensure.

One could have Keep record server-side which signature tokens have been issued (block and lifetime), and have the same signature token cover multiple blocks too, but then they become essentially temporary collections ("partial manifests" IIRC).

Also it is pointless for Keep to issue read permissions tokens to 'arv-put' when it uploads a block, as it does not need to read them.

All that 'arv-put' needs to know is that when it registers a manifest all block hashes mentioned in it are live if the registration succeeded.

So what should actually happen is that the API server on registering a manifest verifies it has all the blocks for the hashes in the manifest, and otherwise returns a list of the blocks it does not have (and signature tokens should just be about permissions, not necessarily imply liveness, because they should not have the same dual role file descriptors have).

That's because it then becomes a compare-and-swap server-side sequence of atomic transactions. In a distributed setup the best that can be hoped for is eventual or even potential convergence.

The verification can be based on first checking whether the hashes in the new manifest are already present in other manifests (and "locking" them for the duration), and then asking all the keep servers to check, and a non-persistent TTL guarantee for the result of that check may
happen at that point.

Everything else is an optimization, for example:

- Having 'arv-put' effectively do that check during the upload, e.g. by registering temporary partial collection manifests, issuing at the end the final full manifest and after that deleting the temporary partial ones.
- Maybe every Keepstore server keeping a persistent hint-list of blocks it has, and perhaps the API server keeping a persistent hint-list of recently known to be live blocks and on which servers.

PS Computery-sciency stuff that may be related: Dijkstra parallel garbage collector with "white", "black", or "grey" (being uploaded) states. Also P Bishop's distributed parallel garbage collector MIT TR-178 (and successors).

History

#1 - 04/15/2016 01:43 PM - Peter Grandi

A different approach could be halfway: instead of having the API server check whether all the hashes in a manifest to be registered are "live" on some Keepstore, the registration request could come with a list of not-expired "signature tokens" obtained by the uploader client from the Keepstores. These could have a pretty short lifetime and in any case the uploader client could reobtain them.

BTW I am basing here my example on the case where the upload source is persistent storage that can be reread. The case where a 25TB upload over 3 weeks and comes from a non-repeatable stream and the relevant collection is registered solely at the end, and the 25TB previously uploaded are guaranteed to be still available for a registration transaction at the end of the 3 weeks, effectively using Keep unregistered blocks as a giant transaction buffer, seems to me not very deserving of support, and another good argument for temporary collections ("partial manifests").

#2 - 04/18/2016 09:16 AM - Peter Grandi

To belabor this point with different wording...

Using the "signature token" to imply lifetime of blocks is analogous to UNIX file descriptors, in that the lifetime of a signature token means that the relevant block is "open" and thus should not be garbage collected even if there is no reference to that block from a "manifest".

But this is in general very hard or futile to attempt to achieve in distributed system, and the illustration is that even NFS does not offer that guarantee. Therefore there can be "stale filehandles":

«A filehandle becomes stale whenever the file or directory referenced by the handle is removed by another host, while your client still holds an active reference to the object. A typical example occurs when the current directory of a process, running on your client, is removed on the server (either by a process running on the server or on another client).»

http://support.esilibrary.com/index.php?pg=kb.page&id=38
https://access.redhat.com/solutions/2674
https://oss.oracle.com/~cel/linux-2.6/estale-strategy.html

#3 - 04/20/2016 03:35 PM - Tom Clegg

Peter Grandi wrote:

> it is pointless for Keep to issue read permissions tokens to 'arv-put' when it uploads a block, as it does not need to read them.

A client must provide permission tokens to the API server when creating collections. Otherwise, it would be trivial to circumvent permissions entirely:

1. Choose a block you want to retrieve, which you currently have no permission to read.
2. Create a new collection that references that block.
3. Retrieve the collection. The API server will provide an up-to-date permission token.
4. Use the permission token to read the block.

This is why keepstore needs to provide permission tokens to arv-put.

The meaning of a permission token is "the client has proved that it has permission to read this data block." A client can prove this by [a] sending the data to keepstore, e.g., from local disk, or [b] asking the API server to confirm in its database that it has read access to a collection that references the block.

The information that "client C is allowed to read data D" is useful to us if and only if data D is available in keep, i.e., it hasn't been garbage-collected. IOW, it's not an accident that the permission TTL is the same as the garbage collection TTL for newly written data.

#4 - 04/20/2016 03:45 PM - Tom Clegg

There are surely other approaches that could be used to build a working system, but that's how interesting that is depends on whether/how the current approach is broken. So for starters -- is the current approach broken?

We have identified one shortcoming in the current implementation, i.e., you can't *decrease* the signature TTL on a live system without breaking things. ("Breaking" means potentially losing data from in-flight operations, until we merge #8936, at which point "breaking" will mean failing in-flight operations.)

This situation doesn't seem (to me) either too bad, or too hard to improve on within the current token system. But I might be missing some other shortcomings that are harder to fix...?

#5 - 04/26/2016 03:09 PM - Peter Grandi

For me the question here is not just whether something is broken now, but also whether it is error prone or fragile or generates surprising behaviour, and we are talking about exceptionally critical infrastructure. "What could possibly go wrong?" :-). Hopefully the just discovered issue with arv-put and non-monotonically-increasing TTLs is and will remain the only issue.

Then I worry about users writing programs that access Keep directly, and being less attentive about the precise semantics of signature tokens, and the implications for long-running jobs. They get in trouble enough with NFS sync/cache issues or k5start :-).

But I think that it is reasonable to have a different opinion as to the risks.

#6 - 04/28/2016 08:53 PM - Brett Smith

Peter,

You're on the right track, but some of what you wrote apparently misunderstands the role of access tokens. So let me start from the beginning, and get down to brass tacks, to try to better explain how the system is intended to work, and then see if there's an issue to address.

Peter Grandi wrote:

> In Keep block liveness is reachability from a server-side manifest or a client-side signature token.

It is neither. The only sure way to determine whether or not the block is readable is to try to actually read it, with a GET or HEAD request. In general, Data Manager is allowed to delete blocks that meet either of these criteria, and today's implementation can do that. To demonstrate this:

1. Create a new collection with replication_desired=0.
2. Shortly before the cluster's configured access token TTL passes, re-get the collection from the API server. You'll have a fresh set of access tokens, good for the TTL's duration.
3. After the collection is slightly older than the TTL, run Data Manager. It will delete any blocks that are unique to the collection, even though the collection object still exists, and the API server recently issued new access tokens for them.

When clients have a block locator+access token, the TTL in there is intentionally a property of the access token, and not the block itself. The TTL not intended to imply anything about the accessibility of the block in the general case. Keep declines to delete blocks that have been PUT more recently than the TTL, because that means there could be a client in progress that is about to create a collection that refers to the block, and that would change Data Manager's own analysis of whether or not the block should be deleted.

Aside from that special case, it absolutely can happen that a client has a valid access token for a block that has since been deleted. It can even happen that the block is already gone when the client receives the access token. Clients are expected to handle this case gracefully. Right now the Python SDK at least distinguishes "block not found" errors as a special case of Keep read errors, so you have that hint. Our higher-level tools could do more with that information (probably as part of a general effort to improve error reporting). We could also let the user know more proactively when they're reading data that might have been deleted by Data Manager.

> Since the Data Manager cannot trace signature tokens, which may be anywhere, only manifests, it must make worst-case assumptions on them, both as to their existence and expiry times, which implies that the block signature TTL must be monotonically increasing, which is hard to ensure.
>
> One could have Keep record server-side which signature tokens have been issued (block and lifetime), and have the same signature token cover multiple blocks too, but then they become essentially temporary collections ("partial manifests" IIRC).

Note that the API server also issues new access tokens, whenever it returns a collection's manifest to a client. (This is how you get access to data in Keep after the access tokens from the initial upload have expired: get the collection from the API server, which provides you with a fresh set of access tokens.) In order to make the worst-case assumption you describe, Data Manager would need to know the last time any user requested a collection's manifest from the API server--and the API server doesn't provide this information, because we don't want Data Manager to be so limited.

> Also it is pointless for Keep to issue read permissions tokens to 'arv-put' when it uploads a block, as it does not need to read them.

When a non-admin user creates a new collection, or updates a collection's manifest, the API server validates all of the access tokens in the manifest. If any block locators are not accompanied by a valid access token, the API server rejects the request. Otherwise, a user could get access to data in Keep that they shouldn't have, by creating a collection whose manifest refers to blocks of data that they want to get. Per the above, they would get valid access tokens when they retrieved that new collection from the API server.

Keep returns a block locator+access token on PUT so the client can put that in a manifest, and prove to the API server that it has permission to read the data.

To summarize: you're right that the access tokens, including their TTL, are only about authorization, and not about the accessibility of the underlying block. I hope this helps explain how Keep and the API server coordinate to grant and cross-check access tokens to enforce that authorization, and that we're not overloading the TTL with any concept of block accessibility. If you have further questions from here, please don't hesitate to ask.

#7 - 05/19/2016 09:30 AM - Peter Grandi

Sorry for the long delay, but I though I had understood your reply, but I was recently asked to explain when stuff is considered "deleted" in Keep, and I realized that I am still confused and rethinking the role of "signature tokens" :-).

«Create a new collection with replication_desired=0»

That is not a good example. That should not even be possible. It essentially says that the collection manifest is to be ignored at least for the purpose of reachability.

«it absolutely can happen that a client has a valid access token for a block that has since been deleted. It can even happen that the block is already gone when the client receives the access token.»

That is good to hear, as it simplifies the reachability rules. Because in the 'arv-put' case it looked as if reducing the TTL was a violation of the

reachability rules of Keep.

«Clients are expected to handle this case gracefully.»

My current understanding then of block reachability is that Keep/Data Manager can delete at any time the replicas of a block that exceed its target replication count given as the sum of all desired replications in all manifests in which is it mentioned, regardless of the global TTL value or the TTL value in a permission token.

Which is more or less the same as for files over NFS or most other distributed systems, as also per:

«We could also let the user know more proactively when they're reading data that might have been deleted by Data Manager.»

Then my current guess of blocks that have a desired replication count of 0 but have been in the "to be deleted" state for less than the global TTL is that the Data Manager won't actually erase them, and if a client requests them their content will be returned, but on a "it just happens to be there" basis.

But in that case I don't understand on which basis permission is granted to access them, as permissions IIRC are granted on the basis of access via a collection id.

So perhaps blocks that have a desired replication count of 0 are actually not accessible, except in the special case of blocks that have just been uploaded and which belong to a "temporary collection".

#8 - 10/13/2016 06:24 PM - Tom Clegg

(continuing with long delays)

Above, Brett described situations where a client has a valid signature for a block that has been deleted. I agree that (with the current code) there are multiple ways to achieve such situations. However, I consider these to be bugs. IMO it is both possible and desirable to eliminate all of those loopholes. Specifically:

- After handing out a signature token S for block B, keepstore refuses to delete B until after S expires. (This is implemented now.)
- After the API server hands out a signature token S for block B, datamanager (now keep-balance) refuses to delete B until S expires. (This is *not* fully implemented now.)

The API/keep-balance loopholes are discussed on the Expiring collections wiki, which is currently in an inconsistent state after the addition of "trashtime".

The main insight is that the API server can make the same guarantee as keepstore (i.e., signature TTL = guaranteed not to delete) using the following approach:

- Track (implicitly or explicitly) the earliest possible data-deletion time for each collection
- Don't hand out signatures that will live past that time
- Ensure the garbage collector (keep-balance) can still see the manifest until that time arrives, even if the user hits all of the "delete" buttons

The simplest solution seems to be to force "trashtime" (the interval in which a client can recover a collection from the trash) to be longer than the configured blob signature TTL, and to ensure keep-balance sees trashed collections.

#9 - 01/18/2020 01:14 AM - Peter Amstutz

- Status changed from New to Closed