Public: etcd livez and readyz probes

https://tinyurl.com/livez-readyz-design-doc

Visibility: Public
Status: Implementing
Authors: chaochn47, siyuanfoundation
Issue: https://github.com/etcd-io/etcd/issues/16007
Tracking spreadsheet: Public: etcd livez/readyz PR tracking
Last updated: Sep 22, 2023

Github handle	Role	Review Status
serathius	Approver	
ahrtr	Approver	lgtm
logicalhan (api-machinery)	Approver	lgtm
lavacat	Reviewer	lgtm
wenjiaswe	Reviewer	lgtm
<add your name>		

Background

The current etcd implementation has a single /health probe that is used to determine both the liveness and readiness of a node.

What does it do?

- check if local node has leader (detects network partition)
- check if local node has any alarm activated
- check if local node is capable of serving a linearizable read request within hard-coded timeout (5s + 2 * election-timeout) (default 7s)

Problem

Current etcd health probe is not Kubernetes API compliant. It does not differentiate whether etcd needs to restart or stop taking traffic. etcd liveness and readiness probe configured with kubeadm using the same health probe is insufficient.

Proposal

Add two separate probes

1. **Liveness**: the liveness probe would check that the local individual node is up and running, or else restart the node.
2. **Readiness**: the readiness probe would check that the cluster is ready to serve traffic.

The existing health probe stays unchanged except bug fixes.

Tenet

Adding the above two probes should be backward compatible.

Definition for livez and readyz

	/livez	/readyz
Definition	**Refer to k8s, properly reflect the fact whether the process is alive or not hence if it needs a restart.** Being alive means all the internal processes and resources are running properly, regardless of external dependencies of the peers or clients.	**Refer to k8s, properly reflect the fact that process is ready to serve traffic** Being ready means the process is able to serve as a good access point to perform strongly consistent KV and watch operations on the underlying distributed key value store. It is an indicator suggesting the client should not send KV and watch requests to this server, but it does not mean the server is actively blocking the requests. Readiness is not an indicator of performance. Slow response is not covered by readiness. Readiness does not cover admin functions. Administrators should connect directly to members to do

		maintenance.
Expected behavior	1. Return true if defrag is active 2. Catch deadlock in the raft loop 3. Catch deadlock in writing to and reading from db	1. Return false if defrag is active 2. Return false if corruption alarm is activated 3. Return false if etcd does not have leader 4. Validate linearizable read can be processed
Examples of no failures	Data corruption: restarting the server alone would not make it better.	Out of quota: the server is still able to take read/delete requests, and still able forward write requests to the leader to write successfully in the cluster. Linearizable read timeout: the timeout could be due to multiple different reasons such as slow follower, readiness does not cover performance issues.
Consumer	Supervisor running close to the process (like Kubelet in Kubernetes) that can restart the process.	Supervisor running close to the process (like Kubelet in Kubernetes) that sets the ready status. L7 Loadbalancer running as gateway Client-side loadbalancer (like grpc client, proposal)
Expected execution	Every 5 seconds, after 3 failures the process is restarted.	Every 1 second, after X failure traffic is no longer sent to the process.

API Design

There will be 2 main http endpoints installed on `listen-client-http-urls` if the user opts in and falls back to default on `listen-client-urls`.

1. **/livez**
2. **/readyz**

The API would return OK if **all** of the checks within that check group listed in “Desired behavior” are OK.

The API also supports excluding specific checks from that health check group with query parameters. For example

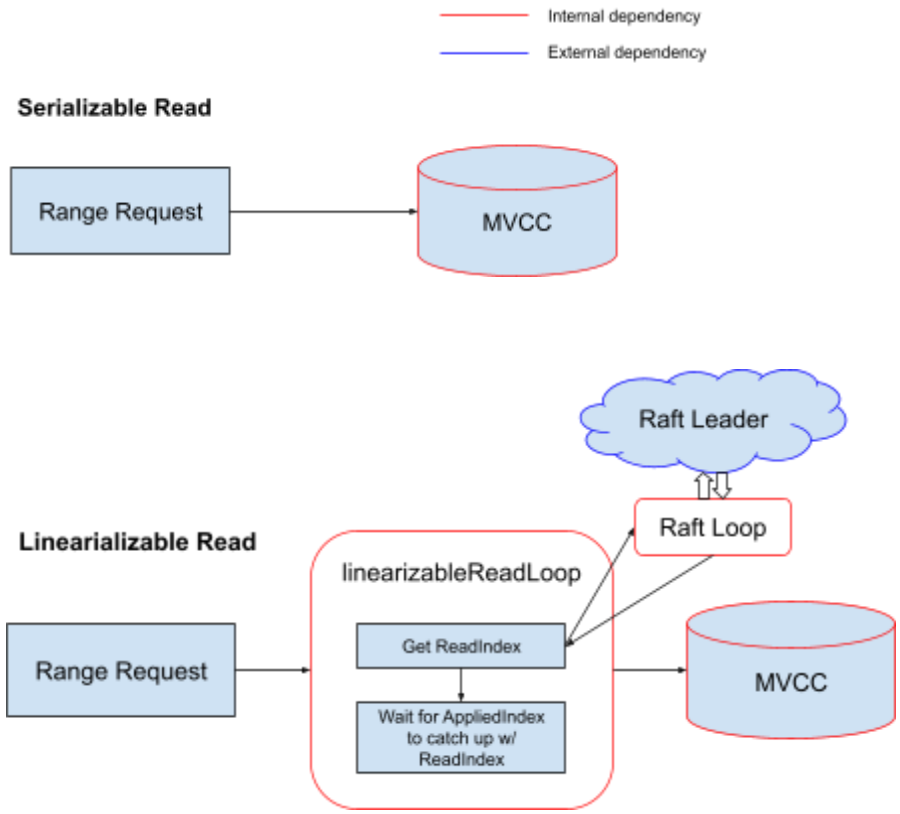
```
curl -k 'https://localhost:2379/readyz?exclude=defragmentation'
curl -k 'https://localhost:2379/livez?exclude=serializable_read'
```

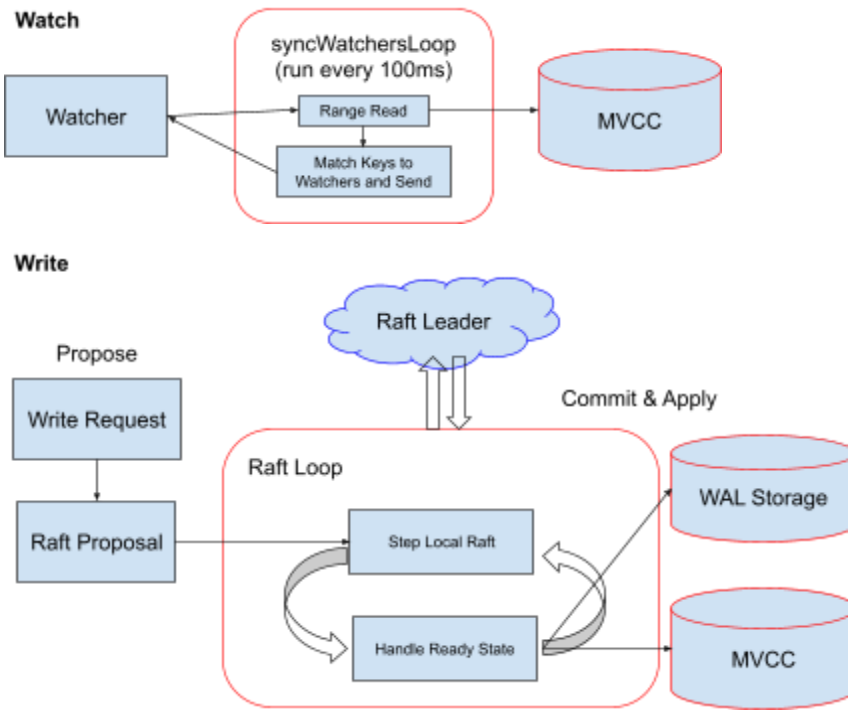
Each individual health check exposes an HTTP endpoint and can be checked individually. The schema for the individual health checks is /livez/<healthcheck-name> or /readyz/<healthcheck-name>.

```
curl -k 'https://localhost:2379/readyz/defragmentation'
```

Failure Modes and Detection Methods

Below are the process execution paths for different critical functions of the server:





For each function, we are listing some potential failure modes for it below:

Function	**Potential Failure Modes**
Serializable Read	disk read failure, data corruption alarm, defrag
Linearizable Read	disk read failure, data corruption alarm, linearizable read loop deadlock, no raft leader, raft loop deadlock, no raft quorum, defrag
Watch	disk read failure, data corruption alarm, watch loop deadlock, defrag
Write	raft loop deadlock, stalled disk write, no raft quorum, defrag *(failure to write to stable or memory storage would result in FATAL or panic already)

In this initial iteration of the probes, we will only focus on the functions of Serializable Read and Linearizable Read. The probing of Watch and Write would merit their own dedicated discussions in the future.

Detection

The table below shows the checks we plan to implement to detect the aforementioned failure modes. This list just reflects the initial implementation, and is not supposed to be exhaustive. We will make the checks to be easily extensible for more checks to be added in the future.

Health Check Name	**Health Check Group**	**Related Failure Modes**	**Health Check Method***
data_corruption	/readyz	data corruption alarm	check for active alarm of AlarmType_CORRUPT.
read_index**	/readyz	no raft leader, raft loop deadlock	check if the server can get ReadIndex.
~~commit_index_progress~~	~~/readyz~~	~~raft quorum, raft loop deadlock, stalled disk write~~	~~In a separate go routine, every 0.5s, make a dummy proposal. In the health check, check if there is any progress of the commit index compared with the last health check.~~
serializable_read	/readyz /livez	mvcc read failure	check if a serializable range (limit 1) request returns error, precondition on: defrag is not active.
...			

*We expect to execute the readyz check every 1s, and livez check every 5s. Any health check within that group should be able to finish below that time scale under normal circumstances.

**Current health check checks if a linearizable read could finish. We prefer just checking the read index instead of doing a full linearizable read because we expect to execute the ready check every 1s, and a full linearizable read could timeout while the local server is trying to catch up with the applied entries, which is not covered under readiness.

Implementation Plan

The following steps would be required to implement this change:

1. Add two new probes to etcd: a liveness probe and a readiness probe.
2. Add http handlers that could detect the above failure modes.
3. Integration and E2E tests the changes with failure mode simulation to ensure that probers work as expected.

4. Back port the changes to supported versions (3.4 and 3.5).
5. Update the etcd documentation to reflect the changes.

Future Discussions/Improvements

There are several remaining topics that are worth more discussion or more work in the future to improve livez/readyz:

1. Should readyz check include checking writes?
2. Should readyz check cover performance issues?
3. What checks can we do to make sure watch is working properly?
4. In order to catch deadlock in raft loop in livez prober, is there a way to do this without involving external dependencies in multi-node scenario?

Reference

1. ahrtr's etcd livez and readyz
2. Public Design Doc: etcd livez and readyz probes
3. Monitoring disk stall in Go
    a. Design Doc: Terminate etcd on stalled storage writes
    b. Proof of Concept: https://github.com/etcd-io/etcd/pull/15440
4. Deadlocks in Go [The strategy is to prevent this from happening, to enable monitoring, you have to turn on profiling which may has a performance degradation]
    a. https://yourbasic.org/golang/detect-deadlock/
    b. https://www.craig-wood.com/nick/articles/deadlocks-in-go/
    c. Go deadlock is a practical tool for finding violations of total lock ordering and has been used to find deadlocks in Cockroach DB for example.
        i. It comes up with error messages that look rather like those from the race detector, but unlike the race detector it detects potential deadlocks before they happen so you can run this on your code which hasn't deadlocked yet.
        ii. Go deadlock only works for Mutex deadlocks, not channel deadlocks. This is probably 90% of the deadlocks so it is an excellent start.
        iii. When I want to use go deadlock with rclone I run this little script. It searches and replaces all the sync Mutex and sync RWMutex with their deadlock detector versions. Unfortunately I find you can't leave Go deadlock in production code as it has too much of a performance impact.
        iv. https://github.com/sasha-s/go-deadlock/issues/25
5. https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#configure-probes
6. https://kubernetes.io/docs/reference/using-api/health-checks/