Health Check Support in Nerdctl – Proposal

Problem Statement

Health checks are a critical mechanism for monitoring the health of containerized applications. They go beyond checking if a container process is running by validating that the application itself is functioning correctly—whether it's responding to HTTP requests, processing jobs, or performing other expected tasks. This proactive monitoring helps detect failures early and enables automated actions like restarting unhealthy containers or alerting operators, ultimately improving system resilience and reliability. Currently, finch/nerdctl doesn't support configuring health checks via dockerfile, compose or cli.
(Open discussion requesting health check feature). To address this, we need to implement health checks with the following requirements:

- Users should be able to configure and modify container health checks using the CLI, Dockerfile, and Compose, with full support for all configuration options available in Docker.
- Users should be able to view health status and health check logs of a running container by listing or inspecting container.

Proposed Solution

NERDCTL + SYSTEMD TIMERS (RECOMMENDED)

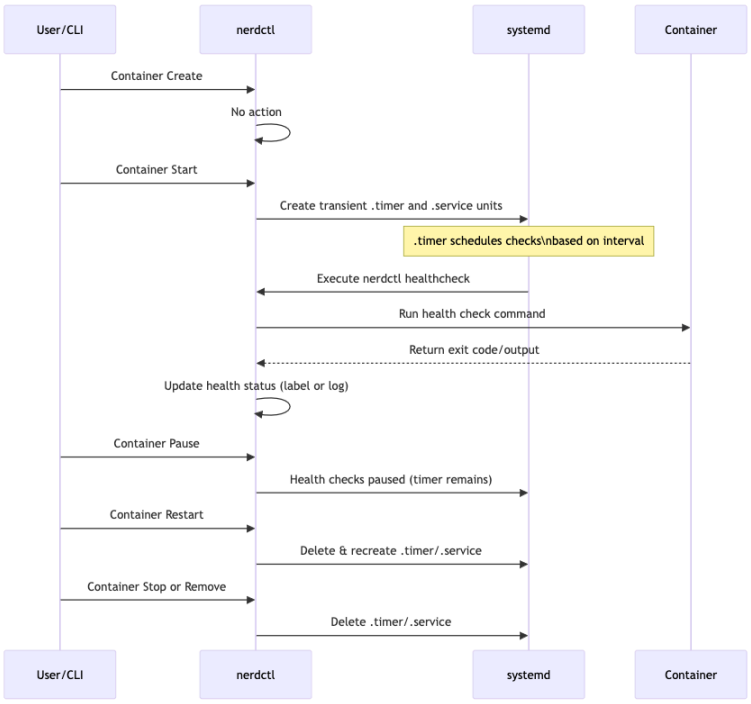
This approach is similar to Podman's model, using systemd timers and services to execute health checks periodically, independent of the daemon. We'll introduce a new `healthcheck` command in nerdctl. This command would take a container ID/Name as input, load the corresponding container, and check whether the container's labels/config include health check settings. If a health check is configured, the `healthcheck` command would execute the test command inside the container using `container exec`, applying the configured timeout. Additionally, users can manually run this command to perform a one-time health check and verify if the application inside the container is running.

```
nerdctl healthcheck run <containerID or name>
```

Scheduling health checks using systemd

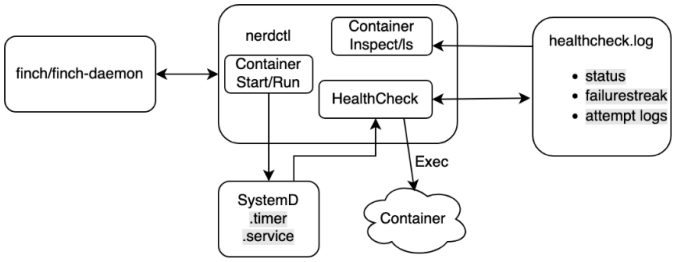
A systemd timer is a unit that schedules the execution of a service unit, similar to a cron job but fully managed by systemd. To continuously run health checks in nerdctl, we leverage systemd timers. For each running container with health checks configured, we create transient systemd `.timer` and `.service` units, typically named using the container's ID. The `.timer` defines when to run the health check (intervals) and the linked `.service` defines what to run — in this case, the `nerdctl healthcheck` command for the specific container. We'll need to make changes in nerdctl to handle container life cycle events as follows:

- **Container Create** → No action
- **Container Start** → Create transient systemd timer & service for health checks.
- **Container Pause** → Systemd timer remains, but health checks stop temporarily.
- **Container Restart** → Delete & recreate the systemd timer & service.
- **Container Stop/Remove** → Delete the systemd timer & service associated with the container.



Tracking Health Check Configuration and Results

We use **container labels** to store the **health check configuration**, such as the command to run, interval, timeout, retries, and start period. Labels integrate seamlessly with the container lifecycle, ensuring that health data is removed when a container is deleted. They provide a persistent and lightweight way to associate configuration data with containers, and can be easily accessed and updated using containerd APIs.
For tracking **health status/logs**, we follow an approach similar to Podman by creating a `<cid>-healthcheck.log` JSON file stored alongside the container's runtime directory. After each health check execution, we read the existing `healthcheck.log`(create if it doesn't already exist) and update the health status and failure streak, and write the updated state back to disk. Since the log is stored on disk, we're not limited by label size constraints and can retain a larger history—though we default to storing the last 5 entries. When a container restarts, the health check state resets and starts fresh. Because systemd is independent of the daemon, timers remain unaffected by client/daemon restarts. The on-disk log ensures health status continuity across restarts and is cleaned up automatically when the container is removed or during a system-wide prune.



This design allows nerdctl to track and report health status over time without needing an always-running daemon, with systemd managing the scheduling and local JSON log files maintaining the health history tied to each container's state. Systemd is not available in all environments, such as minimal Linux distributions (e.g., Alpine) and Windows, requiring alternative scheduling methods. In these cases, health checks will not run automatically unless explicitly configured. To enable

automation, users can utilize cron jobs on Linux or Task Scheduler on Windows (Although WSL 2 now supports systemd) to periodically invoke `nerdctl healthcheck` command.

Currently, nerdctl does not implement the logic required to parse and persist health check configurations defined in Dockerfiles, Compose files, or CLI flags. In Docker, health check instructions from the Dockerfile are stored as part of the image manifest, and Compose overrides are applied at container runtime. To support similar functionality, we will need to extend nerdctl to parse health check configuration from all supported sources and ensure it is either stored in the image metadata during build or applied at runtime when using Compose or CLI. In cases where multiple sources provide health check configuration, we should adopt Docker's behavior by prioritizing Compose-defined settings over those in the Dockerfile. The exact mechanics of parsing, merging, and persisting health config in nerdctl will require further exploration as part of the implementation.

Pros:

- Implementing health checks in nerdctl allows us to extend this feature to both finch and finch-daemon.
- Systemd handles scheduling without requiring a persistent daemon, reducing memory and CPU overhead when no containers require health checks.
- Logs are persisted across daemon restarts, ensuring continuity of health monitoring.
- Podman has proven this approach to work.

Cons:

- Relies on systemd, limiting portability to non-systemd environments

HEALTH CHECK PLUGIN IN CONTAINERD (ALTERNATIVE)

Containerd's modular plugin architecture allows it to be extended and customized without modifying its core code. Many of its core functionalities, such as snapshotting and content storage, are implemented as plugins, making it highly extensible. At startup, containerd loads built-in plugins and can also integrate external extensions through proxy processes or shared libraries. While containerd includes plugins for monitoring resource usage and publishing state changes, it does not have a built-in mechanism for performing application-level health checks.

Hence, a custom plugin for container health checks needs to be developed, adhering to ContainerD's existing plugin interfaces. Ideally, this would be an internal plugin registered under a new `health` type, or possibly integrated under the existing `monitor` type. External plugins currently support only specific service types (like snapshotter, content store etc) making them unsuitable for implementing generalized health check logic. The health check plugin will periodically run user-defined commands inside running containers by leveraging containerd's Task API, which allows executing additional processes (`exec`) in the container context. This enables direct, low-level execution of health check probes, such as `curl` or script-based checks, without modifying the main container process. Health check results can be stored within container metadata in labels, to ensure easy accessibility.

However, embedding application-level health checks directly within ContainerD might not be ideal. ContainerD's primary responsibility is managing container lifecycles and providing low-level telemetry. Implementing detailed, application-specific health checks within ContainerD could complicate its core responsibilities and might duplicate capabilities better handled by higher-level systems.

Pros:

- Leverages containerd's plugin system.
- Potential reuse across other containerd-based projects.

Cons:

- Requires additional deployment and maintenance of the plugin.
- Requires buy-in from containerd maintainers to upstream these changes, which may add delays to adoption.
- containerd will need to be aware of docker specific image schema to support image health configs.