Evolving Health Checks: Adding Primitives for Checking Arbitrary Task States

Epic link: MESOS-6906

Motivation

When humans read data from a sensor, they may *interpret* these data and *act* on it. For example, if they check air temperature, they usually *interpret* temperature readings and say whether it's cold or warm outside; they may also *act* on the interpretation and decide to apply sunscreen or put on an extra jacket.

Similar reasoning can be applied to checking task's state in Mesos:

1. Perform a check;
2. Optionally *interpret* the result and, for example, declare the task either healthy or unhealthy;
3. Optionally *act* on the interpretation by killing unhealthy task.

Currently supported Mesos health checks do all of the above, **1+2+3**: run the check, declare the task healthy or not, and kill it after `consecutive_failures` have occurred. Though efficient and scalable, this strategy is inflexible for the needs of frameworks which may want to run an arbitrary check without Mesos interpreting result in any way, for example, to transmit internal task's state transitions.

MESOS-6833 aims to make killing of unhealthy tasks optional, i.e., Mesos will not necessarily *act* on health check failures. This will allow to support **1+2**.

Local interpretation **2** of a check's result is useful not only for local decision making **3**, but also for giving third-party tools visibility into the task's state. For example, a load balancer may forward traffic only to healthy tasks, without knowing what "healthy" means for this particular task. However it is not always desirable or possible to locally interpret the result of a check.

This document introduces a concept of a general task **check**, whose result is not interpreted by Mesos. This allows to support **1**.

User Stories

As a framework developer, I would like to track task's internal state transitions reliably without Mesos taking action on them.

As a framework developer, I should be able to check multiple tasks in a task group separately.

As a framework developer, I need a clear and reliable (in terms of delivery) signal (e.g. a task status update) when check's result changes.

As a framework developer, I need a check's most recent result to be included into the task reconciliation status update. Moreover, check status updates should not preclude health status from being included into reconciliation messages and vice versa.

Proposed Changes

To address the aforementioned user stories in a holistic and extendable way, *in addition* to existing health checks, we suggest to introduce a new type of a check in `TaskInfo` described by a `CheckInfo` protobuf message. This check is run by an executor and its result is reliably, i.e., via a task status update, forwarded to the framework's scheduler. An executor should (all built-in executors must) neither interpret nor act on the check's result. A task status update with the check's *status* should (for all built-in executors must) be sent if and only if the check's *state* changes (status includes state and possibly other fields in the future).

If both `HealthCheck` and `CheckInfo` are set in `TaskInfo`, information about both of them must be transmitted in each task status update. The executor must fill in the most recent check's result. This is to avoid shadowing of a preceding status update and hence to ensure both health and check statuses are available on reconciliation. The case when no result information is available yet should be distinguishable from the case when the corresponding check has not been defined.

If a check times out, empty result is reported. This is indistinguishable from the check in the delayed period.

To help frameworks and external tooling identify task status updates that are triggered by check status changes, introduce a new `Reason` in `TaskStatus`, e.g., `REASON_TASK_CHECK_STATUS_UPDATED`.

TaskInfo

```
message TaskInfo {
  <...>

  // A health check for the task. Implemented for executor-less
  // command-based tasks. For tasks that specify an executor, it is
  // the executor's responsibility to implement the health checking.
  optional HealthCheck health_check = 8;


  // A general check for the task. Implemented for all built-in executors.
  // For tasks that specify an executor, it is the executor's responsibility
  // to implement checking support. Executors should (all built-in executors
  // will) neither interpret nor act on the check's result.
  //
  // NOTE: Check support in built-in executors is experimental.
  //
  // TODO(alexr): Consider supporting multiple checks per task.
  optional CheckInfo check = 13;

 <...>
}
```

CheckInfo

```
/**
* Describes a general non-interpreting non-killing check for a task or
* executor (or any arbitrary process/command). A type is picked by
* specifying one of the optional fields. Specifying more than one type
* is an error.
*
* NOTE: This API is unstable and the related feature is experimental.
*/
message CheckInfo {
  enum Type {
    UNKNOWN = 0;
    COMMAND = 1;
    HTTP = 2;
    TCP = 3;

    // TODO(alexr): Consider supporting TCP checks and custom user checks.
    // The latter should probably be paired with a `data` field and
    // complemented by a `data` response in `CheckStatusInfo`.
  }

  // Describes a command check. If applicable, enters mount and/or network
  // namespaces of the task.
  message Command {
    required CommandInfo command = 1;
  }

  // Describes an HTTP check. Sends a GET request to
  // http://<host>:port/path. Note that <host> is not configurable and is
  // resolved automatically to 127.0.0.1.
  message Http {
    // Port to send the HTTP request.
    required uint32 port = 1;

    // HTTP request path.
    optional string path = 2;

    // TODO(alexr): Add support for HTTP method. While adding POST
    // and PUT is simple, supporting payload is more involved.

    // TODO(alexr): Add support for custom HTTP headers.

    // TODO(alexr): Consider adding an optional message to describe TLS
    // options and thus enabling https. Such message might contain certificate
    // validation, TLS version.
  }

  // Describes a TCP check, i.e. based on establishing a TCP connection to
  // the specified port. Note that <host> is not configurable and is resolved
  // automatically to 127.0.0.1.
  message Tcp {
    required uint32 port = 1;
  }

  // The type of the check.
  optional Type type = 1;

  // Command check.
  optional Command command = 2;

  // HTTP check.
  optional Http http = 3;

  // TCP check.
  optional Tcp tcp = 7;

  // Amount of time to wait to start checking the task after it
  // transitions to `TASK_RUNNING` or `TASK_STARTING` if the latter
```

```
  // is used by the executor.
  optional double delay_seconds = 4 [default = 15.0];

  // Interval between check attempts, i.e., amount of time to wait after
  // the previous check finished or timed out to start the next check.
  optional double interval_seconds = 5 [default = 10.0];

  // Amount of time to wait for the check to complete. Zero means infinite
  // timeout.
  //
  // After this timeout, the check attempt is aborted and no result is
  // reported. Note that this may be considered a state change and hence
  // may trigger a check status change delivery to the corresponding
  // scheduler. See `CheckStatusInfo` for more details.
  optional double timeout_seconds = 6 [default = 20.0];
}
```

TaskStatus

```
message TaskStatus {
  <...>

  // Describes whether the task has been determined to be healthy (true) or
  // unhealthy (false) according to the `health_check` field in `TaskInfo`.
  optional bool healthy = 8;

  // Contains check status for the check specified in the corresponding
  // `TaskInfo`. If no check has been specified, this field must be
  // absent, otherwise it must be present even if the check status is
  // not available yet. If the status update is triggered for a different
  // reason than `REASON_TASK_CHECK_STATUS_UPDATED`, this field will contain
  // the last known value.
  //
  // NOTE: A check-related task status update is triggered if and only if
  // the value or presence of any field in `CheckStatusInfo` changes.
  //
  // NOTE: Check support in built-in executors is experimental.
  optional CheckStatusInfo check_status = 15;

  <...>
}
```

CheckStatusInfo

```
/**
* Describes the status of a check. Type and the corresponding field, i.e.,
* `command` or `http` must be set. If the result of the check is not available
* (e.g., the check timed out), these fields must contain empty messages, i.e.,
* `exit_code` or `status_code` will be unset.
*
* NOTE: This API is unstable and the related feature is experimental.
```

```
*/
message CheckStatusInfo {
  message Command {
    // Exit code of a command check. It is the result of calling
    // `WEXITSTATUS()` on `waitpid()` termination information on
    // Posix and calling `GetExitCodeProcess()` on Windows.
    optional int32 exit_code = 1;
  }

  message Http {
    // HTTP status code of an HTTP check.
    optional uint32 status_code = 1;
  }

  message Tcp {
    // Whether a TCP connection succeeded.
    optional bool succeeded = 1;
  }

  // TODO(alexr): Consider adding a `data` field, which can contain, e.g.,
  // truncated stdout/stderr output for command checks or HTTP response body
  // for HTTP checks. Alternatively, it can be an even shorter `message` field
  // containing the last line of stdout or Reason-Phrase of the status line of
  // the HTTP response.

  // The type of the check this status corresponds to.
  optional CheckInfo.Type type = 1;

  // Status of a command check.
  optional Command command = 2;

  // Status of an HTTP check.
  optional Http http = 3;

  // Status of a TCP check.
  optional Tcp tcp = 4;

  // TODO(alexr): Consider introducing a "last changed at" timestamp, since
  // task status update's timestamp may not correspond to the last check's
  // state, e.g., for reconciliation.

  // TODO(alexr): Consider introducing a `reason` enum here to explicitly
  // distinguish between completed, delayed, and timed out checks.
}
```

Limitations

- Due to the short-polling nature of a check, some task state transitions may be missed. For example, if the task transitions are `Init [111] → Join [418] → Ready [200]`, the observed HTTP status codes in check statuses may be `111 → 200`.
- Due to the short-polling nature of a check and to the task status updates delivery overhead, oscillating check's state may lead to performance and scalability issues.
- Only HTTP status or command exit code are available.

Framework Author Instructions

There are several important details in the way built-in Mesos executors implement checks (NOTE: docker executor does not currently support checks).

Built-in executors leverage task status updates to deliver check updates to the scheduler. To minimize performance overhead, a check-related task status update is triggered if and only if the value or presence of any field in `CheckStatusInfo` changes. As `CheckStatusInfo` message matures, in the future we might deduplicate only on specific fields in `CheckStatusInfo` to make sure that as few as possible updates are sent.

When a built-in executor sends a task status update because the check status has changed, it sets `TaskStatus.reason` to `REASON_TASK_CHECK_STATUS_UPDATED`. While sending such an update, the executor avoids shadowing other data that might have been injected previously, e.g., health check status. The opposite is also true: the executor strives to preserve the last status of a check in a subsequent update not related to check, e.g., health check status update.

To support 3rdparty tooling that might not have access to the original `TaskInfo` specification, a `TaskStatus.check_status` generated by built-in executors adheres the following conventions:

- If the original `TaskInfo` did not specify a check, `TaskStatus.check_status` is not present.
- If the check has been specified, `TaskStatus.check_status.type` indicates the check's type.
- If the check result is not available for some reason (check has not run yet, check timed out), the corresponding result is empty, e.g., `TaskStatus.check_status.command` is present and empty.

Frameworks that use custom executors are highly advised to follow the same principles built-in executors use for consistency.

On the differences between checks and health checks

Conceptually, a health check is a check with an interpretation and a kill policy. A check and a health check differ in how they are specified and implemented:

- Built-in executors do not (and shall not) interpret the result of a check. If they do, it should be a health check.
- There is no concept of a check failure, hence grace period and consecutive failures options are only for health checks. Note that a check can still time out (a health check interprets timeouts as failures), in this case an empty result is sent to the scheduler.
- Health checks do not propagate the result of the underlying check to the scheduler, only its interpretation: healthy or unhealthy. Note that this may change in the future.
- Health check updates are deduplicated based on the interpretation and not the result of the underlying check, i.e., given HTTP 4** status codes are considered failures, if the first HTTP check returns 404 and the second 403, only one status update after the first failure is sent.

NOTE: Slight changes in protobuf message naming and structure are due to backward compatibility reasons, in the future `HealthCheck` message will depend on `CheckInfo`.

Future improvements

These changes do not seem crucial at the moment, but will either make the check abstraction more flexible or enable some optimizations:

- Data field in `CheckStatusInfo`.
- Delivery policy: all updates, only state changes (current hard-coded policy), changes to any field (when there are more).
- Specify whether to stop probing after certain condition is met or allow removing checks on running tasks.

More future improvements are captured in the following JIRA epics: MESOS-5916, MESOS-7352, MESOS-7353.

Health checks should be ideally expressed in terms of a check with some extra configuration. For example, this is how future health check specification may look like:

```
message HealthCheckInfo {
 // How to interpret the check result: which codes consider as
 // health check failure.
 message InterpretationPolicy {
   message Command {
     repeated uint32 exit_codes = 1;
   }

   message Http {
     repeated uint32 status_codes = 1;
   }

   // The type should be the same as `check.type`.
   CheckInfo.Type type = 1;
   Http http = 2;
   Command command = 3;
 }

 message KillPolicy {
   optional double grace_period_seconds = 6 [default = 10.0];
   optional uint32 consecutive_unhealthy = 5 [default = 3];
 }

 CheckInfo check = 1;
 InterpretationPolicy interpretation_policy = 2;
 KillPolicy kill_policy = 3;
}

// NOTE: Shall we deduplicate on `healthy` or on `check_status` changes
// in general (what we do for `CheckStatusInfo`)?
message HealthCheckStatusInfo {
 CheckStatusInfo check_status = 1;
 bool healthy = 2;
}
```