Distributed Systems Lab 08

Health checks

During the previous lab, we discussed data serialization. This lab presents another technique commonly used in distributed systems, health checks, which are used to monitor the status of a system. We will use files contained in this package, which you should download and extract locally.

Learning section

Failures and failure detection

Processes of real-world distributed systems fail for various reasons. Therefore, the systems have to be designed to continue their operations despite individual **failures**. Monitoring, which processes of a system are alive (and which are not) seems to be a fairly straightforward problem. However, there are many fundamental issues that should be considered.

Usually, to implement **failure detection**, each process of a system provides a simple interface (API/GUI), which can be queried to obtain basic information about its current status. Such a query is commonly named a **health check**. When a process replies to a health check, it can be concluded that the process is alive. However, not receiving a reply for some *time* can indicate any of the following problems:

- the process is not alive,
- the process needs more time to reply *at the moment*, but is actually working,
- there is some network failure,
- the process intentionally does not report its status (e.g., a Byzantine failure occurs in the arbitrary-fault model).

The time necessary for the reply to arrive may vary widely in real-world systems, so it is impossible to be fully sure about current states of other processes. One way of dealing with this inherent uncertainty is to treat the failure detection as a *suggestion* that some process might not be working correctly.

Usually, when a process is suspected to have failed, processes communicating with it stop sending requests to it and continue providing reduced functionality or announce themselves not to operate correctly too. Health checks provide also a convenient way of describing the state of a system, as information that the system is not in a healthy state is far more useful that no information at all.

Practical aspects of health checks

A particular format of a response to a health check depends primarily on the type of entity that receives them. For instance, for humans, it is arguably the most preferred to have an HTTP service that provides easily readable information as plain text or HTML websites. For a software system, arguably the most convenient way is to use a stateless communication protocol (e.g., UDP) and to exchange possibly small messages to reduce the transmission overhead (i.e., it is advisable not to

exceed the Maximum Transmission Unit, MTU). UDP is not a fully reliable protocol, but since the entire issue is subject to the inherent unreliability, it does not undermine the results of health checks.

In complex systems, it might be beneficial to report not only information on whether a process is alive, but also to collect more detailed data like, for instance, CPU load, usage of RAM, IO operations reported by the operating system, statuses of operations that the system is currently executing, and so on. The process collection of such metrics is often called a **telemetry**, and they can be then aggregated at many levels to provide an overview of the system at large. Graphs, for instance, are useful to visualize such aggregates. This way it is possible to easily learn whether a large-scale failure has occurred or not.

To get more information about a problem encountered within a system, one can use logs. Aggregating logs to one location (but only those at the highest levels of importance) can easily show that one of the processes is not operating correctly (such processes usually generate a lot of log entries).

For all of these to work reliably, the software must be designed in a way guaranteeing that problems in some parts of the system do not disrupt running health checks, gathering logs, or other core features. To this end, one can employ the event-driven shared-nothing architecture with modules. Moreover, one can introduce a hierarchy of modules, modeling a complex system as a tree of modules, with parent modules monitoring the states of their children modules and responding to their failures (e.g., restarting a failed child module).

Small Assignment

Your task is to implement a distributed system in which all processes are monitoring each other over UDP. The implementation shall follow the *EventualFailureDetector* algorithm presented during the lectures.

The operation of every process of the system (`FailureDetectorModule`) is divided into intervals. At the beginning of each interval, the process sends `HeartbeatRequest` to all other processes. Then, until the end of the interval, it collects their responses (`HeartbeatResponse` messages) over UDP. At the end of the interval, the process starts suspecting the processes that haven't responded during the interval of having failed.

The processes considered alive by a given process can be queried by sending the `AliveRequest` message. The `AliveInfo` response shall contain identifiers of processes, which, according to the queried process, were alive during the previous time interval (i.e., are not suspected to have failed).

The initial length of the interval is given by `delta`. At the end of every interval in which a process finds out that a process it previously suspected is actually alive, the process should increase the length of the interval by `delta`.

You shall complete the implementation of the system provided in the template. You shall implement:

- `FailureDetectorModule::new()`, which creates and initializes the module,
- handler for the `DetectorOperationUdp` messages, which serve health checks and querying the status,
- handler for the `Timeout` message, which should trigger the timeout (next interval) event of the failure detector.

To serialize requests and responses to raw bytes you shall use the `bincode` crate. To run the system, you should use the module system you implemented as the first Large Assignment.

Additional Learning

If you are interested in modern, industry-standard solutions to telemetry and log aggregation (sometimes jointly described as “observability”), you may want to read about the architectures of the following projects:

- Prometheus. OpenTSDB for telemetry (databases),
- Grafana Loki for logs aggregation,
- Jaeger for tracing microservices (such as most of our modules at the labs).

All of the aforementioned projects may be an input for visualization with Grafana.

Authors: F. Plata, K. Iwanicki, M. Banaszek, W. Ciszewski, M. Matraszek