Redesigning Hera

Sébastien Kalbusch

February 21, 2021

"Simplicity is the unavoidable price we must pay for reliability."
-C.A.R. Hoare

1 Disclaimer

Hera was an attempt at sensor fusion and was used to build a demonstrator. This does not mean that the application has not reached its goal of genericity nor that it does not work. The authors, Julien Bastin and Guillaume Neirinckx have done a great job, but the application is only at its first iteration and can be improved on.

2 Problems with the current design

- **Synchronization:** If a node is too slow to perform a measurement or if the network is too slow, an other node will be allowed to perform a measurement and the late reply will still be accepted. As a consequence, measurements can become unsynchronized because every node is allowed to make his measurement when it wants.

  Moreover, during the synchronization a worker is expected to finish its measure in a certain time. Currently this value is fixed, but it should be a parameter because different measures require different timings. Ideally, an implementation that does not require prior knowledge on the system would be better because finding this parameter might be difficult for the user.

  About prior knowledge on the system, it would be nice that synchronized measurement could also be dynamic because currently it is required to "declare" them in the application environment.

- **Worker pool:** When a worker crashes the pool server will remove it from the pool and put a queued worker in the pool, but the supervisor will restart the dead worker. The problem is that the server is suppose to limit the pool size and because of this bug there is a leakage.

- **Workers restarting:** The current restarting strategy is "permanent" meaning that a measurement (or calculation or filtering) will always be restarted. In case of crash this is problematic because we cannot allow the warm up phase while the system is running (in the case of a sonar, a person could be in the way). That being said, it would be interesting to start the measurement with the warm up value a.k.a. calibration so that it could be reused in case of crash.

- **Hera is limited to Grisp:** Currently the application is made to work with Grisp because of an OS type check. This must be removed because ideally we want Hera to work on any device.

- **The code is complicated:** It took me three days to read the code and I really feel like it is cumbersome. The application is not doing that much things it should not be so large.

3 Simplifying the code

Having a simple code is important because other people may work with it and they will be more efficient if the code is clear. Moreover, a simple design is less likely to fail.

- **The supervision tree:** It represent a large part of the code and it is kind of a maze around the "pools" (lots of back and forth required to understand how it works).

- **The pool system:** The purpose of this system is to allow dynamic insertion of workers. This could be an application on its own and could be abstracted or simplified. Moreover, this system is also suppose to limit the number of workers per pool. Yet, one can wonder if this feature even makes sense in the context of sensor fusion because there is already hardware limitations (ex: we cannot have 1000 measurements at 50 [Hz] each on the same sonar). Still, dynamically starting workers would be very useful in a real life scenario therefore a compromise should be taken there (dynamically starting workers for a minimal complexity).

- **Restart measurement:** This feature is conflicting with the pool system. Indeed, when the measurements are done, the process goes in hibernation and thus remains in the pool. Because of this, once the maximal number of worker will be reached, new workers will never be run. Additionally, I don't really see the benefit

of such feature because we can simply start a new measurement (the same but as a different process). The only requirement is to find the appropriate sequence number when the measurement starts.

- **Filtering:** The idea of a filter is very simple, make a measurement and decide whether or not it should be taken into account. In Hera, the approach is quite different really. A filter is a server with its own pool system and supervision tree and the filter is in charge of sending the measure if it is valid. To me, this is just over engineering. A filter is a simple function (that can be part of a behaviour) and if there is really a need for multi threading then it could be made as an after though.
- **Calculations:** A calculation is very similar to a measurement and these two features could be merged.

4 Hera new design

4.1 Supervision tree

The supervision tree can be found at Fig.1. The top level supervisor `hera_sup` supervises three processes: `hera_data` (section 4.2), `hera_com` (section 4.3) and `hera_measure_sup`. These processes are considered vital for the application and are not expected to fail often. Moreover they are considered independent. Therefore `hera_sup` uses a `one_for_one` strategy and tolerates six fails per hour.

The second supervisor `hera_measure_sup` supervises `hera_measure` processes (section 4.4). Theses processes are considered independent from each other and should be dynamically added. Because a measure process may interact with real world components such as sensors, but also because some might require synchronization (see section 5) we expect them to fail more often. Therefore `hera_measure_sup` uses a `simple_one_for_one` strategy and tolerates ten fails per minute. This supervisor is not design to supervise a very large number of processes because there is a natural limit to the number of `hera_measure` that can run simultaneously (network congestion, computational limitations, ...). However, the design remains open by allowing the user to supervise its `hera_measure` processes himself instead of using our supervisor. There will be no undesirable effect. The only purpose of this supervisor is to ease the implementation effort of the user.

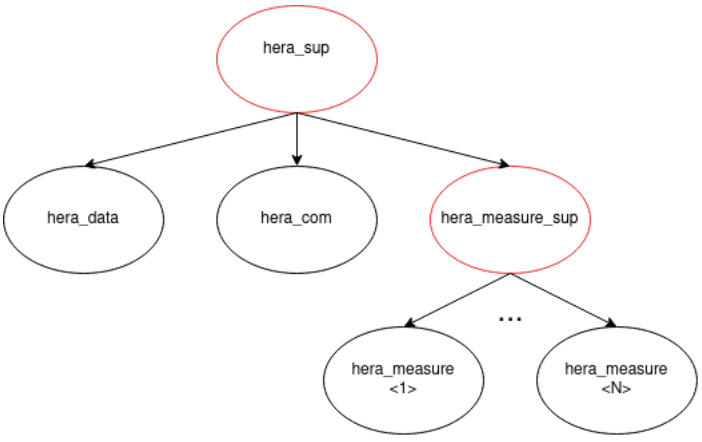

Figure 1: Hera supervision tree

4.2 Hera data

This module is a `gen_server` used as a data storage process for measurement. It stores only the most recent data identified by a name and the node who sent it. The age of a data is based on a sequence number and the data is marked by a timestamp when it is stored. This way, the user knows when the data was received. If he also wishes to know when the measurement was performed he must send a timestamp along with its measure. By setting the

`log_data` environment variable to `true`, it is also possible to log every measurement in a unique csv file for each data identifier (name and node).

4.3 Hera com

This module is a communication process for sharing data across the network with other nodes. It uses a multi-cast udp group for fast but unreliable data sharing.

4.4 Hera measure

This module provides a generic measurement process. It allows the user to simply provide a few parameters and an initial state (like a calibration) in the `init(Args)` callback as well as a `measure(State)` callback to perform a measure. The expected result is either a list of numbers or the atom `undefined` followed by the new state. The undefined atom allows the user to implement a filtering function or an averaging stored in the callback state. In that case, the sequence number will not advance and the call will not be counted as an iteration. If the measure must be synchronized the process wait to receive an authorization from the synchronization process (see section 5) before calling the `measure` callback and will manage the subscription itself.

4.5 Interaction between processes

Fig.2 illustrate how the different processes interacts. The top scenario shows how a measure is shared to all the nodes. The bottom left shows how data can be retrieved. And the bottom right shows how the user can start a measure.

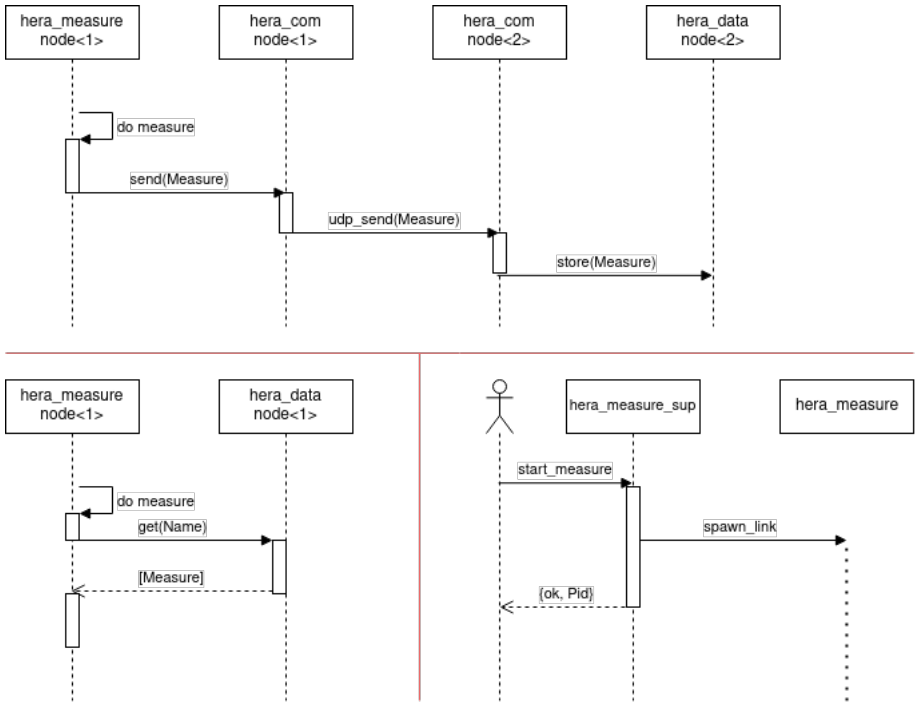

Figure 2: Measure sharing

5 Hera synchronization new design

There are 3 possible properties: perfect sync, perfect timing, partitioning. Perfect sync means that there will never be an unsynchronized measurement (=consistency). Perfect timing means that we control the timing between two authorisations (=availability). Partitioning means that we tolerate a partition in the network. If this is unclear we can reduce the synchronization task as a simple token passing application.

To decide which implementation we should go for, we must decide which properties we want. Of course, we cannot have more than two (CAP theorem). We cannot assume that there will never be a partition because the network can eventually fail. So we must choose between remaining available and insuring perfect synchronization. We chose consistency.

Fig.3 shows how the synchronization works. First, the measure subscribe to `hera_sub` which forwards the subscription to the dedicated process. Then, the `hera_sync` process will authorize the `hera_measure` in due time.

Fig.4 illustrate how the difference processes are supervised or monitored. The idea is to allow the system to dynamically synchronize measurements. And Fig.5 describes how the resilience is achieved thanks to monitors.

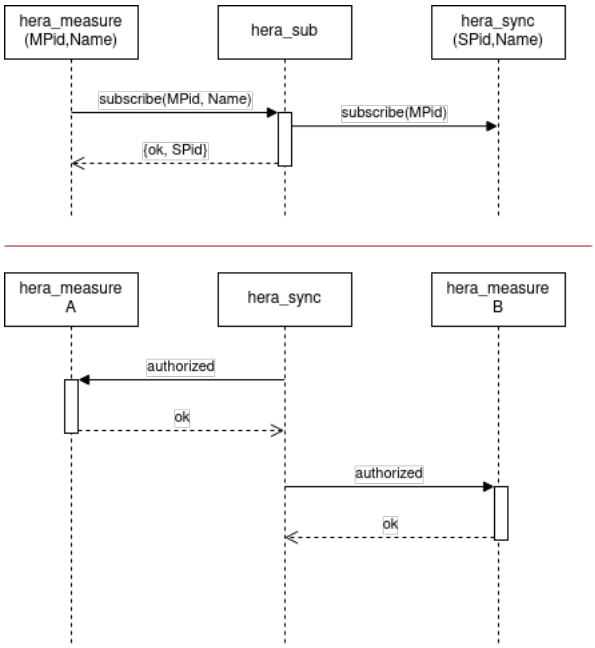

Figure 3: Synchronization principle

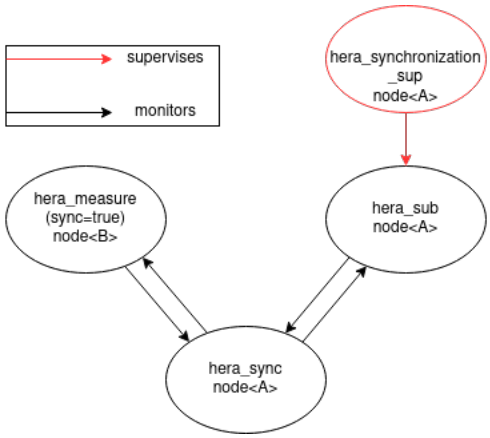

Figure 4: Synchronization monitoring

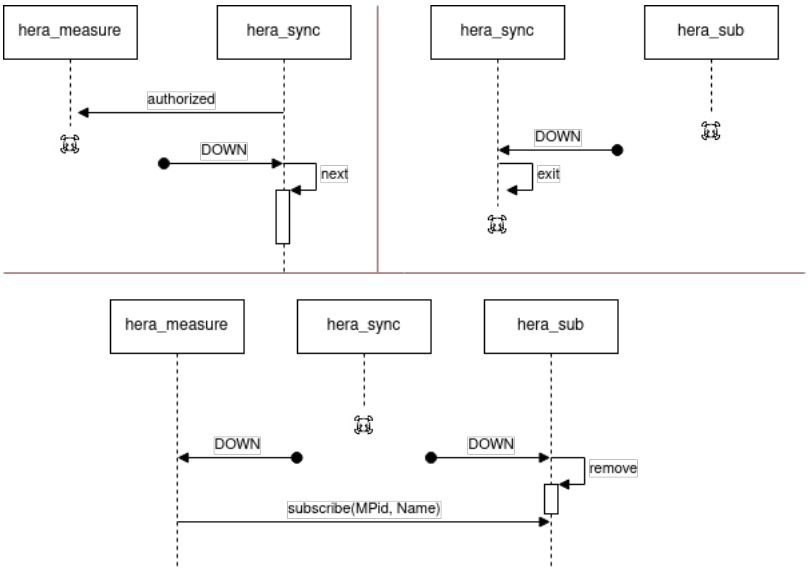

Figure 5: Synchronization resilience

6 Problems with the new design

How to cope with restarting of the application ? We loose calibration data, we do not know which measures were running before the app stopped. We loose the last knows sequence number so if we restart we will start with seq = 1, but other nodes who did not crash will ignore it.