About Juju

In order to manage the significant and ever-growing complexity in the Juju code-base, we strive to follow the single responsibility principle. Our code is organized into self-contained run-time components that can optionally provide services to other downstream components that depend on them.

For instance, an API connection component is responsible for establishing a connection to an API server and making that connection available to any component interested in performing API calls. Likewise, a lock managing component can be used to provide distributed locks to other components without the latter being privy to the internal implementation details of lock management.

As you may expect from any complex distributed system, each component follows its own life-cycle and must be able to recover from both transient and non-transient failures. In the API connection example above, if the connection to the API server is severed, the component must attempt to re-establish it. Furthermore, if the component crashes, it has to be restarted and its dependents able to respond to the restart.

The task

Your mission, should you choose to accept it, is to apply the above philosophy and write a simple data processing pipeline consisting of the following components:

- A data ingestion component that constantly requests payloads from a remote API and makes them available for processing.
- A dispatcher component that processes payloads and sends them up for processing by downstream components based on their type.
- Payload handling components, one for each possible type that receive and process data provided by the dispatcher.

Data ingestion component

The server can be acquired from https://github.com/juju/demoware. **(Please do *not* fork the repo to your personal GH to avoid others seeing that you are attempting the test. Just clone it locally and work locally.)** The data ingestion component is responsible for

authenticating to the remote API and performing calls to its /metrics endpoint. The response is a variable list of metrics responses. Note that each metric has its own payload format. To facilitate the processing of the metric list by the client, each metric is wrapped in an envelope which provides information about the metric type.

For this particular task, you can assume that the API generates the following payloads:

```
{
 "type": "load_avg",
 "payload": {
   "value": $system_load_value
 }
}

{
 "type": "cpu_usage",
 "payload": {
   "value": [$cpu0, ..., $cpuN]
 }
}

{
 "type": "last_kernel_upgrade",
 "payload": {
   "value": $timestamp
 }
}
```

Dispatcher

As mentioned above, the dispatcher processes the payloads emitted by the ingestion component and makes them available to interested downstream consumers.

Payload handler

Each payload handler instance should only consume payloads of a particular type and update its internal state accordingly. More specifically:

- For load payloads, keep track of the min and max load seen so far.
- For cpu_usage, keep track of the average usage per cpu seen so far
- For last_kernel_upgrade, keep track of the most recent timestamp.

Requirements

The following list of requirements must be satisfied for a valid solution:

- All components should be self-contained and able to recover from reasonable types of failure.
- The dispatcher should be started after the data ingestion component.
- Any payload handling components should be started after the dispatcher component.
- Think through the structures that you think will be useful when dealing with these payloads.
- We expect a reasonable amount of tests covering individual components, error handling and the end-to-end operation of the system. However, please don't spend too much time trying to maximize coverage!

Notes:

- You are free to use whatever Go libraries that you deem necessary.
- For simplicity, assume that all components are expected to run inside a single process.
- This is an intentionally open-ended task and there is no "expected" solution. We deal with such tasks all the time in Juju and we are more interested in seeing how you approach these kinds of problems.
- If you make any additional assumptions note them down and/or let us know.

Bonus points:

- Minimal use of locks.
- Handling of back-pressure: match the data ingestion component's API poll rate to the processing speed of downstream components.
- Designing the components and/or their interfaces so that they can be potentially distributed in the future.
- Per-component introspection endpoints so we can query/monitor their internal state.