1. PLUGIN ARCHITECTURE

In today rapidly evolving technological landscape, system requirements can change unexpectedly, making it impossible to anticipate every need in advance. Consequently, system software must be designed with flexibility and adaptability as fundamental principles. This necessity has led to the adoption of Plugin Architectures, which enable new features or modules to be integrated dynamically without interrupting normal operations or requiring system downtime. Such architectures not only enhance extensibility but also support continuous evolution, ensuring that complex systems can adapt efficiently to emerging technologies, user demands, and operational conditions.

A Plugin Architecture typically defines a base abstract class that outlines the essential methods required for all plugin implementations. Conceptually, this class provides the logical interface that every plugin must follow, ensuring consistency within the system. In practical terms, a plugin generally consists of three core components:

- The initialize() method, responsible for loading configurations and initializing necessary resources (plugin should have default parameter for best practice).
- The finalize() method, which handles cleanup tasks such as releasing resources, collecting execution statistics, or closing network connections.
- The run() method, which encapsulates the main operational logic of the plugin.

The following Python code demonstrates a simple implementation of a Plugin Architecture. In this example, a hierarchical file structure is assumed, as outlined below.

```
----main.py             # main program contains plugin manager
----base.py             # define plugin base class
--------plugin/         # all plugins lie in this directory
------------hello.py    # a specific plugin
```

```python
# file: base.py
from abc import ABC, abstractmethod

class BasePlugin(ABC):
    @abstractmethod
    def initialize(self):   # can also pass additional configuration through parameters
        pass

    @abstractmethod
    def run(self):          # can also pass additional configuration through parameters
        pass

    @abstractmethod
    def finalize(self):     # can also pass additional configuration through parameters
        pass
```

```python
# file: hello.py
from base import BasePlugin

class HelloPlugin(BasePlugin):
    def initialize(self):
        print("[HelloPlugin] initialized")

    def run(self):
        print("[HelloPlugin] running... Hello!")

    def finalize(self):
        print("[HelloPlugin] finalized")
```

To actually load and execute a specific plugin at runtime, Python provides the built-in **importlib** library, which enables dynamic importing of modules and classes. Using this approach, a new plugin code can be loaded by specifying its module name and class name, after which an instance of the class can be created and its methods invoked just like any other normal Python object.

```python
# file: main.py
import importlib

class PluginManager:
    def __init__(self, config):
        self.config = config
        self.plugins = []

    def load_plugins(self):
        for cls_path in self.config["plugins"]:
            plugin_cls = self._resolve_class(cls_path)
            if plugin_cls:
                plugin = plugin_cls()   # create an instance of this plugin
                plugin.initialize()     # initialize plugin (add additional config if need)
                self.plugins.append(plugin)

    def _resolve_class(self, cls_path):
        # load a plugin instance by resolving its module and class name
        module_name, class_name = cls_path.rsplit(".", 1)
        module = importlib.import_module(module_name)

        return getattr(module, class_name, None)


# config should be dynamically updated
CONFIG = { "plugins": ["plugin.hello.HelloPlugin"] }

def main():
    manager = PluginManager(CONFIG)
    manager.load_plugins()

    # main execution region
    # normally it should contain a while True loop
    for _ in range(3):
        for plugin in manager.plugins:
            plugin.run()

    # finalize after complete
    for plugin in manager.plugins:
        plugin.finalize()

if __name__ == "__main__":
    main()
```

Note: Other programming languages support this mechanism as well.

Note: The configuration that specifies which plugin should be executed should be managed and updated dynamically using etcd for example.

2. EXERCISES

Students are required to develop a monitoring tool that comprises a centralized server and multiple monitoring agents by integrating all labs. The overall system architecture and data flow of the monitoring tool are illustrated in Figure 1, where:

- Each monitor agent collects data from its localhost then sends to the centralized server, and also listens for commands from the server both through gRPC.
- Each monitor agent reads and updates its configuration in real time from etcd. The configuration format should follow the structure shown in Listing 2.
- Students are encouraged to implement various plugins. A typical example is that monitor agent can prevent data transmission when the new collected data is identical to the previously sent data, thereby reducing network bandwidth usage.
- The centralized server (gRPC server), after receiving monitoring data, forwards it to Kafka for downstream processing and vice versa. gRPC server takes role as a broker that transmits data and control signals between Kafka and monitor agents.
- The analysis application reads data from Kafka (produced by the gRPC server), then print it to stdout. The analysis application also sends commands to Kafka, which are then forwarded by the gRPC server to the respective agents for execution (dummy commands are enough for demonstration).

```
{
    "interval": 5,
    "metrics": ["cpu", "memory", "disk read", "disk write", "net in", "net out"],
    "plugins": ["<PLUGIN_MODULE>.<PLUGIN_CLASS>",]
}
```

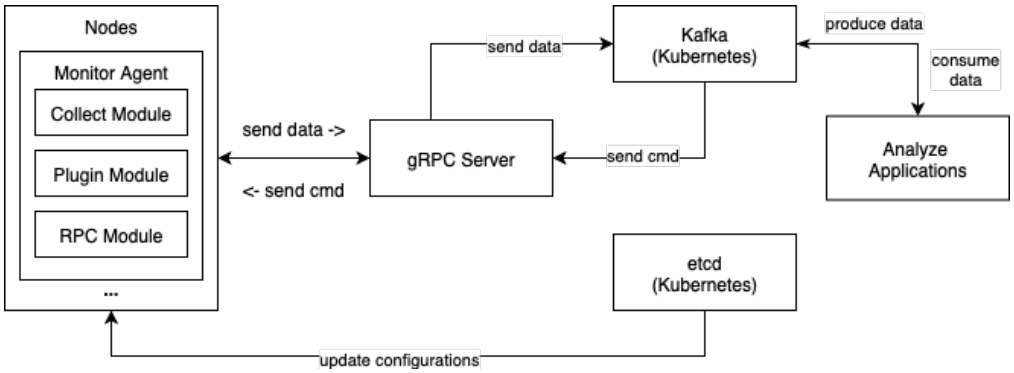

Figure 1: Overall architecture of the monitoring tool.

**** Submission:** Group work is allowed.

- Each group must demonstrate their application. Scores will be assigned based on the functionality and completeness of implemented features.
- Submit a single compressed file (one submission per group) containing all configuration files and source codes, named as **"<StudentID1>_<StudentID2>.zip"**.