Explication détaillée étape par étape de la production des graphes de throughput VS latency

Calcul du throughput du protocole de consensus (ligne 161 de implementation_bloc/benchmark/logs.py)

```
def _end_to_end_throughput(self):
        if not self.commits:
            return 0, 0, 0
        start, end = min(self.start), max(self.commits.values())
        duration = end - start
        bytes = sum(self.sizes.values())
        bps = bytes / duration
        tps = bps / self.size[0]
        return tps, bps, duration
```

Calcul de la latency du protocole de consensus (ligne 171 de implementation_bloc/benchmark/logs.py)

```
    def _end_to_end_latency(self):
        latency = []
        for sent, received in zip(self.sent_samples,
self.received_samples):
            for tx_id, batch_id in received.items():
                if batch_id in self.commits:
                    assert tx_id in sent  # We receive txs that we
sent.
                    start = sent[tx_id]
                    end = self.commits[batch_id]
                    latency += [end-start]
        return mean(latency) if latency else 0
```

Ci-dessus, le code permettant d'obtenir les throughput et latency, qui rejoignent les explications présentées dans le fichier Jolteon-explication-graphe-throughput-latency.

Ensuite, la fonction result (ligne 182 du fichier implementation_bloc/benchmark/logs.py) permet d'afficher le résultat de benchmark dans les fichiers
implementation_bloc/benchmark/data/2-chain/results/bench-*.txt
implementation_bloc/benchmark/data/3-chain/results/bench-*.txt

NB : le format des fichiers de benchmark est : **bench-FAULTS-NODES-INPUT_RATE-TX_SIZE.txt**
Donc par exemple si le fichier s'appelle bench-0-20-140000-512.txt alors il y a 0 noeud byzantins, 20 noeuds, 140000 tr

```
def _tps(self, data):
        values = findall(r' TPS: (\d+) \+/- (\d+)', data)
        values = [(int(x), int(y)) for x, y in values]
        return list(zip(*values))
```

```
    def _latency(self, data, scale=1):
        values = findall(r' Latency: (\d+) \+/- (\d+)', data)
```

```
        values = [(float(x)/scale, float(y)/scale) for x, y in
values]
        return list(zip(*values))

    def _variable(self, data):
        return [int(x) for x in findall(r'Variable value:
X=(\d+)', data)]

    def _tps2bps(self, x):
        data = self.results[0]
        size = int(search(r'Transaction size: (\d+)',
data).group(1))
        return x * size / 10**6

    def _bps2tps(self, x):
        data = self.results[0]
        size = int(search(r'Transaction size: (\d+)',
data).group(1))
        return x * 10**6 / size
```

Ensuite, les lignes suivantes (lignes 33 à 54 dans le fichier plot.py) utilisent la librairie python re (regex, pour du pattern matching avec des expressions régulières) pour trouver dans les fichiers bench-*.txt chacune des informations.
Par exemple, dans la fonction _latency, la fonction findall de la librairie re permet de trouver toutes les occurences de la forme TPS : [digit(s)] +/- [digit(s)]

Le fichier implementation_bloc/fabfile.py permet de gérer l'ensemble du cycle de vie des benchmarks, depuis la création et la gestion des instances, l'exécution des benchmarks en local ou à distance, jusqu'à la génération de graphiques de performance et le traitement des logs.

Autres fichiers et leurs fonctions :

- aggregate.py : Lorsqu'un objet `LogAggregator` est créé, il lit tous les fichiers de log dans un répertoire spécifié par `PathMaker.results_path()`. Il parse ces logs en utilisant les classes `Setup` et `Result`, et agrège les résultats en calculant les moyennes et les écarts types. La méthode `print` génère des rapports pour la latence, le TPS et la robustesse. Les résultats sont organisés, triés et écrits dans des fichiers dans un format lisible.

- instance.py : contient une classe `InstanceManager` qui gère des instances AWS EC2 pour un environnement de benchmark, en utilisant la bibliothèque `boto3` pour interagir avec AWS.

- local.py : contient une classe `LocalBench` qui gère l'exécution de benchmarks locaux pour un système distribué. Cette classe utilise plusieurs modules personnalisés pour configurer, exécuter et analyser les benchmarks

- utils.py : dirige le benchmarking du protocole en local, en gérant les paramètres, la configuration des nœuds, l'exécution des clients et des nœuds, la collecte des résultats, et la gestion des erreurs.

Description pas-à-pas de la création des fichiers

Quand on lance fabfile.py, le programme essaie de faire (fabfile.py, ligne 135) :

```
Ploter.plot(plot_params)
```

Cela appelle la @classmethod plot de la classe Ploter (on voit bien qu'il n'y a pas de création d'instance de la classe Plot)

La fonction plot, située à la ligne 135 du fichier plot.py lance tout d'abord un appel à la classe PlotParameters du fichier config.py (ligne 148) via :

```
params = PlotParameters(params_dict)
```

Le fichier s'occupe ensuite de créer les fichiers avec différentes mesures caractérisant l'efficacité du protocole de consensus et crée les graphiques à l'aide des appels suivants (ligne 162 du fichier plot.py) :

```
cls.plot_robustness(robustness_files)
cls.plot_latency(latency_files)
cls.plot_tps(tps_files)
```

Il s'appuie aussi pour réaliser sur le fichier utils.py