Ethereum Performance Metrics

Version:1.0.0

ETHTerakoya Scaling Working group

Index

1　イントロダクション

1.1　本資料作成の背景

近年、自社のみ、あるいは単一の業界に留まらず、様々な利害関係者が一部のエンティティのコントロール下に置かれることなく必要なデータを共有可能とする同一のネットワークに参加し、結果として新たなユーザ体験を提供しえるサービス構築を実現可能とするブロックチェーン(または分散台帳)技術の導入検討が各社・各業界で進んでいる。

一方でブロックチェーンの導入時にはいくつかの課題が存在する。
それらの課題のうちの一つは本資料の目的にも繋がるが、導入時における性能評価における課題である。

他のシステム導入と同様に、ブロックチェーンの導入の検討にあたっても、自社サービスに必要な性能指標を満たすかの確認が必要になるが、ブロックチェーン自体も Bitcoin や Ethereum などのパブリック型、あるいは、主に企業ユースを前提として設計された Hyperledger Fabric や Corda などのようなコンソーシアム型(プライベート型)などの複数の基盤・ネットワークが存在しており、各社・各業界で実証実験や本格導入に向けた基盤選定が異なっていることもあり、現状、性能テスト条件や結果として提示する数値項目に関する業界共通の基準が存在しない。
その結果、各サービスやプロダクトの性能公表値について、各社や各業界にて、各々の性能指標を設定し、測定を行っているという実情がある。

本来は下記のプロセスを経ることで、適切な性能評価・比較が可能となると考えられるが、現状では明示されることが少なく、ビジネス要件に合致しているかの判断が正確に行えない状況である。

【性能評価に求められる適切なプロセス】
1　トランザクションの種類やシステム負荷などの**性能テスト条件を明示**した上で
2　スループットや遅延時間などの**性能テスト結果を提示**すること

1.2　本資料の目的

本資料では上記の背景に記載した課題に対する解決策として、まず Ethereum を主とする性能評価の測定指標を策定することを目指す。本資料において定義され、評価された内容は、全てのブロックチェーンにおいても適用可能とはならないものの、性能評価に関する基本的な枠組みを提示しており、他のブロックチェーンにも応用可能であると考えられる。

また本資料では、性能テスト実施時に必要な観点を性能テスト条件、システム負荷条件、テスト結果項目として提示し、用語および定義の共通化を図ることも意図している。
同時に実際に性能テスト実施した結果を記載する為のフォーマットも提示することで、今後、ブロックチェーンを活用したサービス・プロダクトを提供しようとする際に、検討段階におけるブロックチェーン選定や実証実験、あるいは実開発フェーズ以降における開発・テストの一環として実施する性能評価の共通化・効率化に資することを期待する。

1.3　本資料の対象読者

本資料で想定する対象読者は以下の通り。

- ブロックチェーンをビジネスに導入したい企業・担当者
- 開発後に、性能評価を行う必要のある開発者
- 自社でブロックチェーンを活用したサービス・プロダクトを開発・提供するベンダー

1.4　本資料の内容

ブロックチェーンの性能を測定・比較する際に必要となる下記項目は以下の通り。

- 測定条件・結果確認に関する主要な観点
- 上記観点に関する基礎知識
- 測定の実施手法

1.5　本資料の活用方法

ブロックチェーンの性能テスト・比較の際に参照し、測定に用いるべき観点や基礎知識を理解すること。
また測定の実施手法を理解し、適切な性能テスト条件の設定および結果の測定を行うこと。

2 用語と定義

本資料で使用される用語とその定義を以下に記載する。なお、本章で示す用語と定義は、後段の「3 性能テスト結果のレポート・フォーマット」で用いるものについて網羅するよう記載している。

2.1 性能テスト条件として使用される用語

性能テストを実施する際の主な条件としての用語および定義を以下に示す。

2.1.1 Blockchain Name

ブロックチェーンのクライアント名。

例:
- Ethereum / geth
- Hyperledger Fabric
- Hyperledger Besu
- Corda

など

2.1.2 Consensus Algorithm(合意形成モデル)

ブロックチェーンにおけるネットワーク全体の合意を行うための方式。ブロックチェーンにおける合意形成とは、一般的には、合意形成の主体となるノードがネットワークに送信されたトランザクションを任意のデータサイズのブロックに取り込み、ネットワーク全体にブロードキャストした後、各ノードによって検証され新たなブロックとしてブロックチェーンに取り込まれるまでのネットワーク全体で行われる一連のプロセスと解されている。

典型的な例を次頁に記載する (太字はプロダクトとしてのデフォルト設定)。

系列	製品	合意形成モデル
Ethereum (含む、EEA)	Go-Ethereum	**Proof of Work (Ethash)**
		Proof of Authority (Clique)
	Open Ethereum	**Proof of Work (Ethash)**
		Proof of Authority (Aura)
	Hyperledger BESU Consensys Quorum)	Proof of Work (Ethash)
		Proof of Authority (Clique)
		Proof of Authority (IBFT 2.0)
	GoQuorum Consensys Quorum)	Proof of Authority (Clique)
		Proof of Authority (IBFT)
		Raft
Hyperledger Fabric		Endorsement Ordering ＋ Solo (v2.x 以降非推奨)
		Endorsement Ordering ＋ Kafka (v2.x 以降非推奨)
		Endorsement Ordering ＋ **Raft**
Corda		系全体の合意形成なし

合意形成のモデルによって性能が変化するのは当然であるが、同一の合意形成モデルを採用していても、パラメータの指定やノード配置によっても性能は変化する。

2.1.3 Transaction Method

ブロックチェーンのデータをある値から別の値に変更する状態遷移法。例えば、ネットワーク上にデプロイしたスマートコントラクトの処理コードなどが挙げられる。

2.1.4 Network Size (Node 数)

合意形成に参加しているマイニングノードやバリデータノードの数。ネットワーク上には、左記ノード以外にも、各々役割を有した複数のクライアントノードが存在するが、本項目では、あくまでも直接的にブロック生成に関与するノードのみを対象とする。
テスト対象とするコンセンサスアルゴリズムにより、最低限必要なバリデータノード数がある場合には、その数を確保する必要がある。

2.1.5 ブロック生成時間

ブロックを生成する時間間隔。

2.2 システム負荷条件

性能テスト実施時に、ネットワークやテスト対象ノードに対してかける負荷条件について以下に示す。

2.2.1 負荷生成サーバ

負荷テストツールを稼働させるサーバ。

2.2.2 負荷生成クライアント

実際のサービス・プロダクトにおけるユーザの代替となり、自動化されたテストスクリプトなどを通じて、トランザクションを発行するクライアントノードあるいは代用プログラムなど。(例: 200 clients)

2.2.3 負荷総リクエスト数

負荷生成クライアントにより発行された 1 秒間あたりのネットワーク全体の総トランザクション数。(例: 73,555 tps)

2.2.4　負荷継続時間

負荷総リクエスト数を継続させる時間。実質的に性能テスト時間となる。(例: 60 秒間)

2.2.5　目標 TPS

当設定値の間隔で負荷総リクエスト数まで送信を行う。（例：50TPS）

2.2.6　成功とみなすまでの承認数

トランザクションの生成を成功とみなすための承認数。

2.2.7　負荷ツール単体での限界リクエスト数

テストに用いる負荷ツールで生成できるリクエスト数の限界値。
利用するツールにより、負荷ツール単体での限界リクエスト数が異なるため、テストの参考情報として提示することが望ましい。
また、関連するパラメータが基盤側にあるケースもあるため、基盤側の確認も必要となる。

2.3　テスト結果項目

本テストでは、「4 Appendix」に記載した手順に従って、Hyperledger Caliper を用いた性能テストを実施することを目的としている。具体的なテスト項目における定義を以下に示す。(ブロックチェーンの性能テストを実施する際の Hyperledger Caliper を用いたネットワーク構築方法に関しては、参考文献 1) も参照のこと。)

2.3.1　Read Latency(読み取り遅延時間)

読み取り要求を送信し、その応答を受信するまでの合計時間。以下の式によって算出される。

$$Read\ Latency\ =\ Response\ received\ time\ -\ submission\ time$$

2.3.2 Read Throughput(読み取りスループット)

スループットとは、単位時間あたりで処理できるデータ量のことで、本資料では、Read Throughput とは 1 秒あたりの読み取り数（reads per second ; rps)。定義された期間内に完了した読み取り操作の数をカウントする為の具体的な指標であり、以下の式によって算出される。

$$Read\ Throughput\ =\ Total\ read\ operations\ /\ total\ time\ in\ seconds$$

Read Throughput はほとんどのシステムにおいては通常、CLI や API 等を介してブロックチェーンに接続された外部 DB サーバを用いて、ネットワークに直接負荷や影響を与えないクライアント処理として読み取りと問い合わせの効率化を実現している為、ブロックチェーンを測定する中心的なパフォーマンスとしては使用されない。

2.3.3 Transaction Latency(トランザクション遅延時間)

ネットワーク全体がトランザクションを検証するのにかかる時間。ブロードキャスト時間と合意形成プロセスが費やす割り当て時間をカバーし、次頁の式によって算出される。

$$Transaction\ Latency\ =\ Confirmation\ time\ *\ network\ threshold\ -\ submission\ time$$

ネットワークの一部がトランザクションをコミットするのにかかる時間の量を表す(定義詳細などは、参考文献 2) を参照のこと)。多くの場合、ネットワークの閾値は無く、実際には $Confirmation\ time\ -\ submittion\ time$ によって計算される。

2.3.4 Transaction Throughput(トランザクションスループット)

定義された期間にブロックチェーンによって有効なトランザクションがコミットされる割合。1 秒あたりのコミット数（transaction per second, tps）で表し、以下の式によって算出される。

$$Transaction\ Throughput\ =\ Total\ valid\ transactions\ /total\ time\ in\ seconds$$

トランザクションのスループットは単一のノードでの測定ではなく、ネットワーク全体の全てのノードの平均を測定する。

2.3.5 CPU 負荷

CPU に対する負荷。(例: vmstat の結果)
CPU が性能評価時に推奨される基準を満たしている前提で、性能テストによって CPU 負荷が高まりすぎノードの処理に影響をあたえる場合は、ブロックチェーン基盤の仕様上の問題なのか、クライアントアプリケーションの仕様の問題なのか、等を見極める必要がある。

2.3.6 ディスク容量負荷

ディスク容量に対する負荷。(例: iostat の結果)

3　性能テスト結果レポートのフォーマット

性能テスト条件とテスト結果に関する以下の項目を、レポート内に記載することを推奨する。

3.1　テスト条件

種別	項目	定義	値(例)
ブロックチェーンに関する情報	Blockchain Client Name	ブロックチェーンのクライアント名	Ethereum / geth, Hyperledger Fabric など
	Consensus Algorithm	ブロックチェーンにおけるネットワーク全体の合意を行うための方式。	Proof of Work (Ethash), Proof of Authority (Clique) など
	Transaction Method	ブロックチェーンのデータをある値から別の値に変更する状態遷移法。 性能測定の対象となる処理の内容。	性能評価に使用したスマートコントラクトのコード など
	Network Size(Node 数)	合意形成に参加しているバリデータノードの数。	8 台
	ブロック生成時間	ブロックを生成する時間間隔。	10 秒
ハードウェア情報	負荷生成サーバ: クラウドの場合		
	使用したクラウドサービス	クラウドサービス名	AWS, GCP, Azure など
	インスタンスタイプ	クラウドサービスのインスタンス名	2.xlarge, t3.medium など
	RAM 容量	クラウドサービスのインスタンスの RAM 容量	16GB

	負荷生成サーバ: 実マシンの場合		
	CPU 種別	マシンの CPU 種別	Intel Core i9 3.5GHz
	GPU 種別 ※ PoW に GPU を 使う場合	マシンの GPU 種別	RTX 3090
	RAM 容量	マシンの RAM 容量	16GB
	ブロックチェーンの各ノード: クラウドの場合		
	使用したクラウドサービス	クラウドサービス名	AWS, GCP, Azure など
	インスタンスタイプ	クラウドサービスのインスタンス名	2.xlarge, t3.medium など
	RAM 容量	クラウドサービスのインスタンスの RAM 容量	16GB
	ブロックチェーンの各ノード: 実マシンの場合		
	CPU 種別	マシンの CPU 種別	Intel Core i9 3.5GHz
	GPU 種別 ※ PoW に GPU を 使う場合	マシンの GPU 種別	RTX 3090 など
	RAM 容量	マシンの RAM 容量	16GB
システム負荷条件	負荷生成クライアント数	システム負荷をかけたクライアントの数	200
	負荷総リクエスト数	クライアントから送信した総リクエスト数	73,555 tps

	負荷継続時間	クライアントからかけた負荷の継続時間	60 秒
	目標 TPS	当設定値の間隔で負荷総リクエスト数まで送信を行う	50TPS
	成功とみなすまでの承認数	トランザクションの生成を成功とみなすための承認数	2 承認
	負荷ツール単体での限界リクエスト数	負荷ツール単体でかけられるリクエスト数の最大値	73,555 tps

3.2　テスト結果

種別	項目	定義	値(例)
ブロックチェーンに関する情報	Read Latency	読み取り要求を送信し、その応答を受信するまでの合計時間	0.18 秒
	Read Throughput	1 秒あたりの読み取り数	813.1
	Transaction Latency	ネットワーク全体がトランザクションを検証するのにかかる時間	11.18 秒
	Transaction Throughput	定義された期間にブロックチェーンによって有効なトランザクションがコミットされる割合	27.4
システム負荷情報	CPU 負荷	CPU に対する負荷	Max: 56.7% Avg: 21.83%など
	ディスク容量負荷	ディスク容量に対する負荷	Min: 0.80 KB/s Max: 146.80 KB/s Avg: 49.47 KB/s

4 Appendix

4.1 Hyperledger Caliper でのテスト手順

ここでは実際に負荷検証ツールを用いて負荷テストを行う手順を紹介する。
負荷検証ツールは検証結果を一意にする為に、単一のツールを使い、同様の負荷テストを多くの環境に対して行う事が好ましいと考える。
ここでは負荷検証ツールとして、オープンソースで開発されている Hyperledger Caliper を使い負荷テストを行う。

【注記】
Hyperledger Caliper でテストを行う場合、汎用性においては以下の点で注意が必要となる。

- Hyperledger Caliper は web socket 経由のトランザクションのみを測定対象としているため、web socket に対応していないチェーンについては、他の測定手段を用いる必要がある。

- 数千 tps までの負荷に耐えることができる Layer2 等を測定対象とする場合、1 マシンでは十分な負荷をかけきれない場合がある。
  その際は複数マシンで負荷をかけるマルチワーカー対応が必要になる可能性がある。

- 処理プロセス上でボトルネックとなっている箇所まで把握したい場合、アプリケーションレベルでのテストはなく、チェーンの各処理にログを仕込む等の手法を検討する必要がある。

4.1.1 用語

- caliper-cli: npm のパッケージ。caliper を操作するための CLI ツール
- caliper(npm, docker): 負荷を掛ける本体
- caliper-benchmarks: ベンチマーク用のサンプルアーティファクトが含まれている
- caliper manager: worker process に対して指示を与える機能。manager1 台で負荷を掛けることもできる
- caliper worker process: 負荷を掛ける機能

4.1.2 Caliper 初期設定

◆ Caliper のインストール

Nodejs が動く環境と Node パッケージマネージャ（npm）がインストールされている環境を想定とする。

例：

```
cd /home/vagrant/besu
npm install --only=prod @hyperledger/caliper-cli@0.4.0
```

◆ caliper-benchmark リポジトリのクローン

例：

```
git clone https://github.com/hyperledger/caliper-benchmarks.git
```

◆ caliper が使用するターゲットと SDK バージョンを指定

例：

```
cd /home/vagrant/besu
npx caliper bind --caliper-bind-sut besu:latest
```

4.1.3 ノードによる負荷

◆ caliper manager の起動

➢ besu の場合

例：

```
$ cd /home/vagrant/besu/caliper
$ npx caliper launch manager --caliper-bind-sut besu:latest --caliper-benchconfig
benchmarks/simple/config.yaml --caliper-
networkconfig ./networks/besu1node/networkconfig_simple.json --caliper-workspace .
```

➢ geth の場合

例：

```
$ cd /home/vagrant/besu/caliper
$ npx caliper launch manager --caliper-bind-sut ethereum:latest --caliper-benchconfig
benchmarks/simple/config.yaml --caliper-
networkconfig ./networks/geth1node/networkconfig.json --caliper-workspace .
```

起動オプションの詳細は以下の通り。

オプション	設定値	意味
--caliper-bind-sut	besu:latest	対象の環境とバージョン
--caliper-benchconfig	benchmarks/scenario/simple/config.yaml	測定するシナリオを定義したファイル
--caliper-networkconfig	./networks/networkconfig.json	測定環境を定義したファイル
--caliper-workspace	.	caliper が仕事するときの基点となるパス

設定値の詳細は下記を参照のこと。

https://hyperledger.github.io/caliper/v0.4.2/runtime-config/

4.1.4 複数ノードによる負荷

todo: https://qiita.com/jnmt/items/0fb65e2ea722f7b68921

4.1.5 処理の流れ

◆ caliper manager を起動

- docker-compose で環境構築
- 負荷測定開始
- 負荷測定終了
- docker-compose で環境を解体

◆ コマンドラインに結果が返る

◆ html でレポートが作成される

4.1.6 設定ファイルの解説

◆ フォルダ構成

```
./caliper
  ├ benchmarks    // ベンチマークを測定する負荷ツール
  ├ networks      // 測定対象の環境。docker-compose で作る。
  └ src           // 測定対象のスマートコントラクト. abi ファイルを格納
```

◆ networks

- networks/networkconfig.json
  - 公式ドキュメントは下記。
    https://hyperledger.github.io/caliper/v0.4.2/ethereum-config/
  - 参考のため、設定値の意味を記述したファイルを下記に用意した。
    caliper/networks/networkconfig.sample.json

- networks/data/genesis.json
  - besu の初期ブロック

- caliper/networks/keys/key
  - besu ノードの秘密鍵

◆ src

- 測定対象のスマートコントラクトの abi ファイルを格納
- 当 abi ファイルは networks/networkconfig.json の ethereum.contracts.simple.path で指定

◆ caliper/benchmarks/simple

- ベンチマーク測定用の負荷ツールの置き場。スマートコントラクトに合わせて自作する必要がある。
- caliper/benchmarks/simple/config.yaml
  - どのような負荷を掛けるかを定義したファイル
  - 目標 TPS、実行プログラムを決定
  - YAML 形式のフォーマット

- rateControl 部分は下記を参照
  https://hyperledger.github.io/caliper/v0.4.2/rate-controllers/

- caliper/benchmarks/simple/open.js, query.js, transfer.js

- 測定対象のプログラム毎にカスタマイズする必要がある
- 負荷を掛ける実行プログラムの本体
- createWorkloadModule createSimpleState submitTransaction の 3 つの関数から成り立つ
- createWorkloadModule
- Caliper によって呼び出される関数
- クラス名くらいを変える程度で問題ない
- createSimpleState
- caliper から渡される変数
- config.yaml の test.rounds.workload.arguments で定義した変数を使うことができるようになる
- submitTransaction
- caliper が BC に負荷を掛ける時に呼び出す関数
- 負荷を掛けるプログラム毎に SC の呼び出し関数が変わる
- this.createConnectorRequest('SC の関数名', 関数に渡す引数のオブジェクト)

- caliper/benchmarks/simple/utils/operation-base.js

- 編集不要
- 負荷を掛けるプログラムが継承する母体

- caliper/benchmarks/simple/utils/simple-state.js

- SC 毎に変更の必要あり
- SC の関数に渡す引数を調整するプログラム
- ランダムな値が設定された引数を作成

4.1.7 独自のコントラクトに負荷を掛ける方法

※以降は https://github.com/hkiridera/caliper-benchmarks をベースとして実行することを想定。

◆ スマートコントラクトを truffle でコンパイルする

◆ コンパイルされた結果のファイル(build/contracts/**.json)を参考にし、下記の構成で caliper/src ディレクトリに保存する

```
"name": "コントラクト名",
 "abi": [
   ABI 情報
 ],
 "bytecode": スマートコントラクトの実行コード,
 "gas": TX 発行時の Gas 量
```

◆ 性能測定をするインフラ構成のファイルを作成する

- caliper/networks/ethereum/geth1node を参考にし、同様に作成する

◆ 上記フォルダ内の networkconfig.json を編集する

- caliper.command.start: 開始時の処理。docker-compose.yml のパスを変える
- caliper.command.end: 終了時の処理。docker-compose.yml のパスを変える。不要ファイル削除ツールがあれば動かす
- ethereum.contracts."コントラクト名"にする
- ethereum.contracts."コントラクト名".path: caliper/src に作った json ファイルを指定する
- ethereum.contracts."コントラクト名".gas: 関数毎の gas 量を指定する

◆ ベンチマーク測定用実行プログラムを作成する

- caliper/benchmarks/simple をコピーし、 caliper/benchmarks/MyNFT とする

- caliper/benchmarks/MyNFT/config.yaml を編集する

· simpleArgs: 実行プログラムのプログラムに対して引数を渡す場合には、その引数と値を宣言する
· test.name: コントラクト名
· test.rounds: 測定対象の関数の分だけ用意する
· test.rounds.labels: 任意の名前. 測定対象の関数名と同じにするのが分かりやすい
· test.rounds.workload.module: 実行対象のプログラムのパスを指定する

- initializeWorkloadModule を編集する

· initializeWorkloadModule 内の this. は、caliper/benchmarks/MyNFT/config.yaml の simpleArgs に合わせること
· _createEthereumConnectorRequest の contract は 測定対象のコントラクト名を指定すること

- caliper/benchmarks/MyNFT/utils/simple-state.js を編集する

· コントラクトの関数で指定するパラメータを動的に変得る場合、ここで引数部分を調整する。
· caliper/benchmarks/MyNFT/config.yaml で指定した引数を使う場合は、 constructor 部分で変数に格納する

5　参考文献

1)Hang, Lei, and Do-Hyeun Kim. "Optimal Blockchain Network Construction Methodology Based on Analysis of Configurable
Components for Enhancing Hyperledger Fabric Performance." Blockchain: Research and Applications (2021): 100009.

2)Eyal, Ittay, et al. "Bitcoin-ng: A scalable blockchain protocol." 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16). 2016.

6 変更履歴

版数	改訂日	改訂内容
1.0.0	2021/10/29	初版公開