

# 分散システムにおけるノードのグルーピングとグループ内マスターノードによる分断耐性の強化

新宮 隆太<sup>1,a)</sup> 串田 高幸<sup>1</sup>

概要：分散システムの奇数台ノード運用において障害で偶数台になった際に 1:1 にノードが別れてしまい一貫性が失われるという問題点が存在している。そこで、複数台あるノードを奇数になるようにグルーピングを行い、各グループ内に 1 つ Primary ノードを定め、グループ内でネットワーク分断が発生した際には Primary ノード同士が疎通の取れるグループ数で生存するかの判定を行うことにより偶数台ノードの際に障害発生しても一貫性を高めることができる手法を提案する。

## 1. はじめに

### 1.1 背景

近年の高度に情報化された世の中では身の回りの様々なものがネットワークでつながっており、生活に必要不可欠な社会インフラの 1 つとなっている。そのような社会の中でシステムは障害を起こしてもできるだけ正常に動作を続けられるような堅牢性を求められており、多くの堅牢性を求める大規模システムは分散システムとして実装されている [1]。

分散システムには 2000 年に Eric A. Brewer 氏が提唱した CAP 定理というものがある [2]。CAP 定理は分散システムノード間データ複製において一貫性 (Consistency)、可用性 (Availability)、分断耐性 (Partition-tolerance) を 3 つ同時には保証できないというものである。この内一貫性と分断耐性を重要視する (一般的には CP システムと呼ばれる) etcd, Apache Cassandra を始めとする分散システムでは奇数台のノードで運用されることが前提になっている。これはネットワーク分断で 1:1 にノードが別れてしまいそのままどちらのノード群もネットワーク分断されたまま稼働して一貫性を失ってしまうことを阻止するためである [3]。

### 1.2 課題

一貫性を重要視する分散システムで用いられることの多いアルゴリズムに Raft がある [4]。Raft は一貫性のあるレプリケーションを行うことができることや長期間の停止、

大幅に遅延したノードが復帰したりノード故障によるリーダー交代が発生しても合意したデータが破壊されるシナリオが存在しないというメリットがあるが、1 つのリーダーが多くの機能を有しているため、その他のノード (フォロワー) の数が多いと通信が多くなりリーダーに大きな負荷がかかるという点や過半数の同意に基づくアルゴリズムのためノード数は奇数構成が推奨されておりネットワーク分断の際に 1:1 に分かれると一貫性が失われるリスクが存在している [5][6]。

例えば、分散してユーザーの預金を保存している金融システムがあるとする。図 1 のようにノードが 5 つの金融システムの場合、あるユーザーの残高は 100 円だったとしたときにそこに 100 円を追加で入金すると 5 つのノードすべてが残高 200 円という情報を保持する。ここで 3:2 のような形でネットワーク分断が発生すると 3:2 の 2 の方のノード群は全体のノードの過半数を下回っているため動作を停止する。これによって稼働しているノードはすべて残高 200 円という情報を保持することができる。しかしながら図 2 のようにノード数が 4 つの状態でもネットワーク分断が発生し、ノードが 1:1 になる。つまり 2 つのノードの塊が 2 つという状況になると両方のノードが過半数を下回っているためどちらのノード群もネットワークが分断されたまま稼働し続けてしまい片方では残高 200 円、もう片方では残高が 100 円というように不整合になってしまう。これは過半数以上稼働しているノード群を稼働し続けさせて過半数を下回っているノード群は動作を停止させるという仕組みに起因している。

<sup>1</sup> 東京工科大学コンピュータサイエンス学部  
〒192-0982 東京都八王子市片倉町 1404-1

a) C0117155

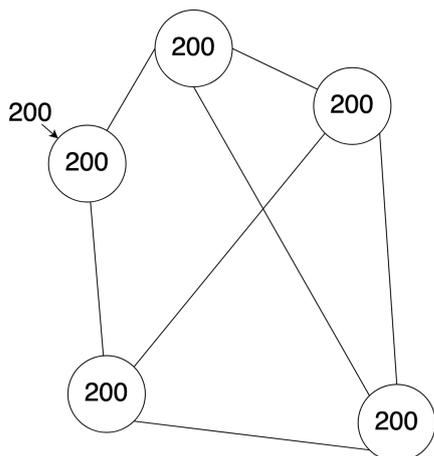


図 1 ノード 5 つの金融システムの例

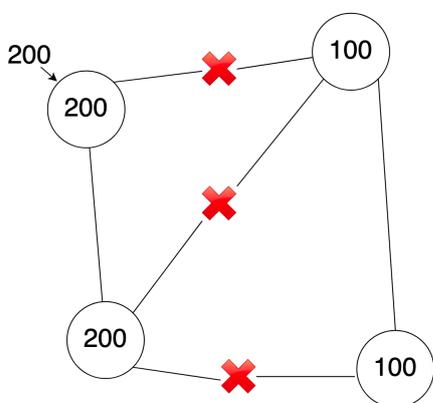


図 2 ノード 4 つの金融システムの例

本稿ではネットワーク分断によってノードが 1:1 に分断されてもノード数以外の指標を用いて片方のノード群を停止することで一貫性を維持しつつ運用し続けることのできる手法について提案を行う。

## 2. 関連研究

分散システムの障害耐性についての研究はアルゴリズムの提案が多い。今回、それらの中的相关性の高い研究を挙げていく。

ユーザーに低遅延で障害耐性のあるサービスを提供するためのミドルウェアを提案している研究では、提案の中で複数ノードをグループにまとめてそのグループ内に Primary ノード 1 台と複数台 Backup ノードを設置しており、それを複数設置することにより障害耐性を高めている。しかしながら普段は Primary ノード間でしか処理を行わないため 1:1 でのネットワーク分断があった際には Raft を代表とする既存手法と変わらず一貫性を失う問題点がある。また、Primary ノード以外に Backup ノードが複数事前にスタンバイすることで処理に参加できないノードを用意するリソースが必要になり効率的でないという問題点も存在して

いる [7].

ノード障害、ネットワーク障害、ネットワーク分断時のフォールトトレラントな分散ストリーム処理に対するレプリケーションベースの提案を行っている研究では、閾値を設けてその閾値時間以内に処理を完了することを保証した上で障害発生時でも閾値以内はノードを稼働し続ける。その閾値をユーザーに委ねることで可用性と一貫性の重みを変更できるようにしている。しかしながらこの提案では強く一貫性を求めるシステムの場合には一時的にでも不整合になってしまう点が問題となる [8].

## 3. 提案

本稿ではネットワーク分断によってノードが 1:1 に分断されてもノードをグループに分けてそこで稼働する Primary ノードの生存数を指標として片方のノード群を停止することにより、奇数でしか稼働できないという制約を無くして一貫性を維持しつつ運用し続けることのできる手法について提案する。

複数台あるノードを奇数個のグループになるようにグルーピングを行う。その後各グループ内に 1 つ Primary ノードを定め、グループ内でネットワーク分断が発生した際には生存している Primary ノードの個数で生存するかの判定を行うことにより偶数台ノードの際に障害発生しても片方のノード群は動作を停止するため一貫性を高めることができる。図 3 ではノードが 10 個稼働しておりそれを 3 のグループに分けている。この時ネットワーク分断が発生しノードが 5 台と 5 台、つまり 1:1 に別れた際に本稿の提案手法を用いることで Primary ノード同士が疎通の取れるグループ数が 1:2 に別れているため左側のノード群だけが停止することが可能となっている。

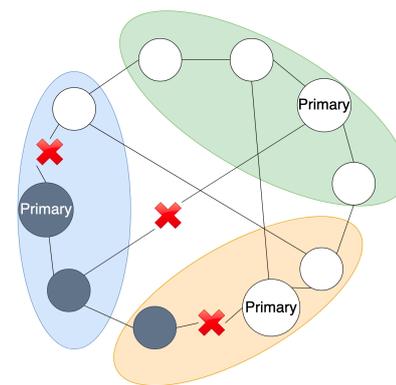


図 3 グルーピングをしたノード群がネットワーク分断を起こした例

本稿で提案する手法の大まかな流れとしてはまずノードネットワークの形成、その後ノードグルーピング、Primary ノード選出、障害検出の順番で処理を行う。また、障害発生時はノードネットワークの復旧の処理も行う。それでは本稿で提案する手法の詳細な処理を順に説明する。

### 3.1 ノードネットワークの形成

ノードネットワーク形成にはすでにノードネットワークに参加しているノード内で稼働している API サーバーにノードネットワーク追加依頼を自分の IP アドレスとノードが稼働開始した時間 (以下 `boot_time`) とともに送信する。受け取った側はノードネットワークに参加しているノードの IP と `boot_time` が記載された `node_list` に追加依頼されたノードの情報を追記して依頼したノードに返す。その後受け取った側のノードは図 4 のように Primary ノードを介してすべてのノードに更新された `node_list` を再配布する。また、Primary ノードが障害で停止しノードネットワークの再構築をする際には停止した Primary ノードの代わりに後述する Temporary Primary ノードが選出され、それを介して `node_list` の再配布を行う。

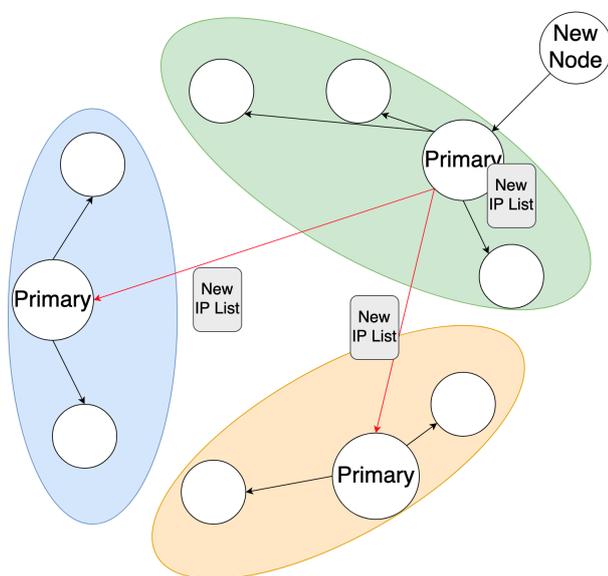


図 4 `node_list` 再配布の例

### 3.2 ノードグルーピング

ノードのグルーピングは最低一台のノードが属しているグループのグループ数が全体で奇数となるようにして、各ノードの稼働時間を参照しグルーピングを行う。ノードの稼働時間を参照する理由としては 2 点ある。まず 1 つ目に再起動を繰り返すノードを検知することができるという点である。稼働時間が極端に短いノードが連続で復帰動作を繰り返すことで動作の不安定なノードをいち早く検出することができる。2 つ目の理由として、グルーピングの段階では Primary が存在しないのでグルーピングを行うと各ノードによってグルーピングの結果が変わってしまうためどのノードから見ても同じ指標を用いてグルーピングしなければならない。そのためどのノードから見ても同一の指標として用いることのできるノードの稼働時間を参照することにした。各ノードの稼働時間は `node_list` で管理している `boot_time` を用いる。一般的な `up_time` を用いずに

`boot_time` を用いる理由として `up_time` はそのシステムが稼働し始めてからの時間を指しているためはノードリストに記載したタイミングによって変わってしまうため単一の指標にならないが `boot_time` は起動したときの時間を指しているためそれを単一の指標として用いることができるためである。

`boot_time` をソートして最新なものと同古のものを順番に取り出して規定グループ数になるようにグルーピングを行う。最近起動したものだけを集めたグループが存在すると動作の不安定なノードが複数混じってしまいそのグループ全体の動作が不安定になってしまう可能性があるため最近起動したノードだけグルーピングを行わないようにする。

ソースコード 1 はノードグルーピング処理の擬似コードである。2 行目でノードリストを `boot_time` をキーとして昇順ソートして `sorted_node_list` へ代入している。6 行目から 19 行目までの `for` 文では生成するグループ数が代入されている `group_num` の数まで `sorted_node_list` の一番最初と一番最後を交互に取り出す、つまり稼働時間が長いノードと短いノードを取り出してグループを振り分ける。最後に 17 行目で 1 つのグループの中で一番稼働時間が長いノードを Primary ノードとして設定する。

ソースコード 1 ノードグルーピング処理の擬似コード

```
1 def grouping(node_list: list, group_num: int):
2     sorted_node_list = boot_time_asc_sort(
3         node_list)
4     grouped_node_list = list()
5     node_list_length = len(sorted_node_list)
6
7     for i in range(group_num):
8         flag = True
9         for j in range(int(node_list_length /
10             group_num)):
11             if flag:
12                 flag = False
13                 sorted_node_list[0]['is_primary']
14                     = False
15                 sorted_node_list[0]['group_id'] =
16                     i + 1
17                 local_grouped_list.append(
18                     sorted_node_list.pop(0))
19             else:
20                 flag = True
21                 sorted_node_list[-1]['is_primary']
22                     = False
23                 sorted_node_list[-1]['group_id']
24                     = i + 1
25                 local_grouped_list.append(
26                     sorted_node_list.pop(-1))
27
28     local_grouped_list[-1]['is_primary'] =
29         True
30     grouped_node_list.extend(
31         local_grouped_list)
```

### 3.3 Primary ノード選出

Primary ノードは `node_list` の配布時の仲介とネットワーク分断発生時の生存するノード群を決定を行う際に用いる。その Primary ノードはグループ内で `boot_time` の一番古いノードを Primary ノードとする。これもノードグルーピングと同じように動作の不安定なノードをできるだけ避けるためにより長く稼働しているノードを Primary として選出する。また、Primary ノードが障害で停止した際には Temporary Primary ノードを選出する。Temporary Primary ノードは Primary ノードの次に `boot_time` が古いものを選出する。

### 3.4 障害検出

グループ内ノード間で疎通が取れなくなった際にはまずノードの故障が原因なのかネットワーク分断が原因なのかの判別を行う。もしノードの故障が原因と判定した場合にはそのノードを `node_list` から削除して再グルーピングを行い、ネットワーク分断や複数ノードが動作を停止した場合には相互で疎通の取れる Primary ノードが属しているグループの個数が少ない方のノード群が自発的に動作を停止する。

図 5 の例はノードが 4 つでグループ数が 3 の状態で Group3 の Primary ノードが動作を停止している。この例を用いて動作を停止しているノードとネットワーク分断の判別の方法について説明する。

- (1) Group3 の Primary ノードが他の同一グループのノードから疎通が取れなくなる。
- (2) Group3 の一般ノードが他グループすべての Primary ノードに Group3 の Primary ノードへの疎通確認依頼を送信する。
- (3) 他グループの Primary ノードが疎通確認を行う。
- (4) 疎通確認の結果が帰ってくる。

(4) の結果がすべて疎通確認できなかったという結果だった場合はノードの故障、一部が疎通できた場合やそもそも疎通確認を依頼した Primary ノードから結果が返ってこなかった場合はネットワーク分断であると判定を行う。また、疎通確認依頼を送る直前に他のノードから別のノードの疎通確認依頼が送られてきた場合もネットワーク分断と判定する。

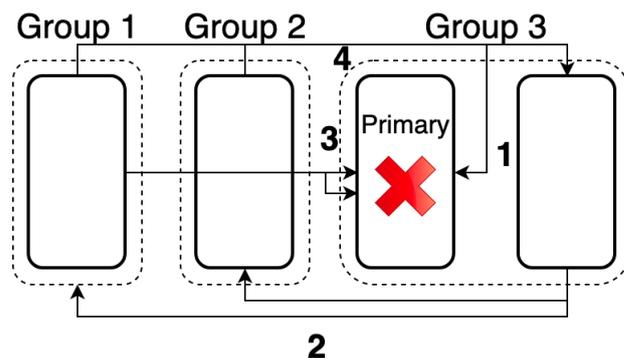


図 5 ノード故障とネットワーク分断の判別の例

### 3.5 ノードネットワークの復旧

故障で動作を停止したノードが復旧した場合やネットワーク分断によって動作を停止したノード群がネットワーク分断が解消された場合には再びノードネットワークに参加するために復旧動作を行う。

#### 3.5.1 故障で動作を停止したノードが復旧する場合

動作を再開したノードが障害発生前に保持していた `node_list` が存在している場合はその中のいずれかのノードにノードネットワーク追加依頼を送る。もし保持していた `node_list` の中に一台も現在稼働しているノードがない場合は、手動で現在稼働しているノードの IP を設定してノードネットワーク追加依頼を送る。

#### 3.5.2 ネットワーク分断によって動作を停止したノード群が復旧する場合

ネットワーク分断でノードの動作を停止した場合にはすべての機能を停止することはせずに定期的に疎通確認を行いネットワーク分断が解消されているかの確認を行う。もしネットワーク分断が解消されていた場合は動作を停止したノード群すべてを復旧する。また、ネットワーク分断中に新たなノードが稼働していた場合は再グルーピングを行い、新たなノードが稼働していなかった場合は動作停止前のグルーピング状態に戻る。

## 4. 実装と評価

### 4.1 実装

今回、本稿提案のアルゴリズムの簡易的な実装を行う。図 6 のようなノードネットワーク形成、ノードのグルーピングと Primary ノード選出を行う `NodeGroupingTestApplication` と他のノードからの通信を待ち受けや `node_list` のノード間共有のための API サーバーの実装を行った。ソースコードは GitHub のリポジトリ\*1から確認することができる。

\*1 [https://github.com/homirun/node\\_grouping](https://github.com/homirun/node_grouping)

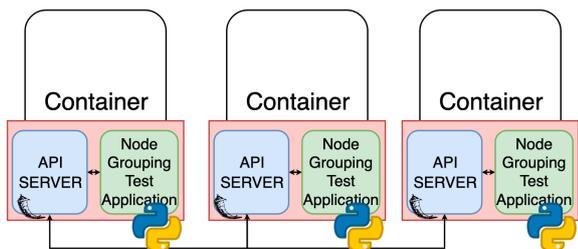


図 6 NodeGroupingTestApplication と API サーバーの構成

実装は主に Docker コンテナ上の Python3.8 で行い, API 部は Python の Web アプリケーション開発における軽量なフレームワークの Flask, HTTP 通信では高機能な HTTP ライブラリである Requests, node 自身の IP を取得するために実行環境のネットワークアダプタの情報を収集できるライブラリである netifaces, node の boot\_time を取得するために様々な環境でシステムの起動時間を取得することができるライブラリである uptime を用いた.

#### 4.1.1 ノードネットワークへの参加

図 7 は NodeGroupingTestApplication が起動してからノードネットワークに参加するまでのノード間の通信とスレッド間のやり取りについてを示している.

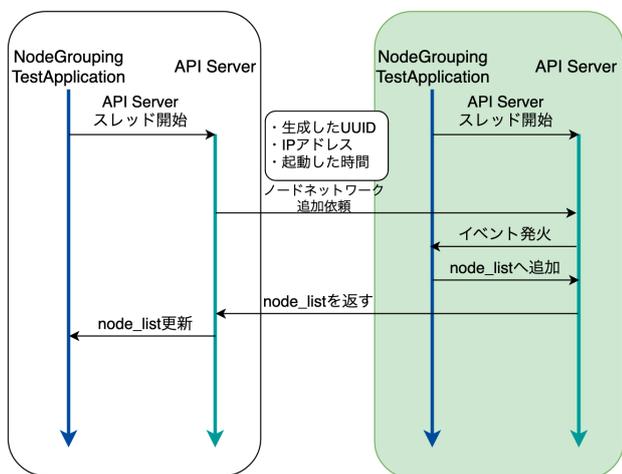


図 7 ノードネットワーク参加までのスレッドとノード間の処理

NodeGroupingTestApplication は起動するとまず最初に API Server スレッドを立ち上げる. その後 API Server スレッドから別ノードへノードネットワーク追加依頼として生成した一意な ID と IP アドレス, ノードが起動した時間を JSON 形式でソースコード 2 のような形で送信先の API サーバーへ送信する.

ソースコード 2 ノードネットワーク追加依頼のリクエスト

```

1 {
2   'type': 'add_request',
3   'id': '4a679292-3ab5-482c-a898-a62d60708a05',
4   'sender_ip': '192.168.48.4',
5   'boot_time': '1595239978.0',
6   'time_stamp': '1595239980.0'
7 }

```

追加依頼を受け取ったノードは Main Thread のイベントが発火して内部の node\_list に依頼送信元のノードを追加する. その後それを JSON 形式でソースコード 3 のような形で API サーバーを介して依頼送信元へ返す. JSON 形式のレスポンスは node\_list キー以下の部分がそれぞれのノードの情報を示しており, 上から順にノードを識別するための uuid.uuid4 メソッドを用いて生成した一意な ID, IP アドレス, ノードが起動した時間 (unix time), 所属しているグループ ID, そのノードが現在 Primary ノードかどうかの 5 つの情報を示している. レスポンスが返ってくる段階ではまだ再グルーピングがなされていないため追加依頼送信元のノードの group\_id は None となっている. 最後に依頼送信元のノードは返ってきた node\_list を使って内部の node\_list を更新してノードネットワークへの参加の処理が終了する.

ソースコード 3 ノードネットワーク追加依頼のレスポンス

```

1 {
2   'status': 'add_request_response',
3   'node_list': [
4     {
5       'id': '58e4d948-b6b4-4125-9493-ed6b95eabb96',
6       'ip': '192.168.48.2',
7       'boot_time': '1595239954.0',
8       'group_id': 1,
9       'is_primary': True
10    },
11    {
12      'id': '4a679292-3ab5-482c-a898-a62d60708a05',
13      'ip': '192.168.48.4',
14      'boot_time': '1595239978.0',
15      'group_id': None,
16      'is_primary': False
17    }
18  ]
19 }

```

#### 4.1.2 ノードグルーピングと Primary ノードの選出

node\_list は dict 型 list として保持されている. ノードグルーピングの処理ではその node\_list を boot\_time をキーとしてソートし, それを slice を用いて list の一番上と一番下を交互に取り出し, ノードの起動時間の最長のものと

最短のものと同じ一つの `group_id` を付与している。また Primary ノードは同一 `group_id` が付与されているノードが格納されている `list` を `boot.time` をキーとしてソートし、一番下のノードを取り出す。取り出したノードは `is_primary` を True にすることで Primary ノードとして選出されたこととなる。

#### 4.1.3 node\_list の共有

`node.list` が更新されるとそれを他のノードに伝搬しなければならない。それを行うためにはまず `node.list` の更新を取得しなければならない。実装では、内部の `node.list` が更新されたかを直前までの `node.list` と比較することで確認を行っている。

`node.list` が更新されるとその更新は Primary ノードから送信されてきた更新なのかを確認する。もし Primary ノードからの更新だった場合は `node.list` 内に送信元 IP アドレスが記載されている `sender_ip` をキーとした dict が入っている。

更新がどのノードから送信されてきたかにより以下の4種のうち1つの挙動を行う。

- 自分が Primary ノードかつ Primary ノードから来た更新の場合は自グループのノードへ新しいノードリストを送信する。
- 自分が Primary ノードで一般ノードから来た更新の場合は他の Primary ノードへ新しいノードリストを送信する。
- 自分が一般ノードで Primary ノードから来た更新の場合はどこにも更新を送信しない。
- 自分が一般ノードかつ一般ノードから来た更新の場合は自グループの Primary ノードに新しいノードリストを送信する。

これらの動作を各ノードそれぞれが行うことによって同じノードを複数回経由することがないため効率的に `node.list` の共有を行うことができる。

#### 4.2 実験環境

macOS 10.15.4 がインストールされた MacBook Pro(Late2016, Intel i5, RAM:8GB) で Docker Desktop for Mac 2.3.0.2 を用いて前の章で説明した `NodeGroupingTestApplication` と API サーバーが実行可能な Docker コンテナを複数立てた。

#### 4.3 評価

ノード数が10台のときのノードリスト変更からグループピング処理が終了するまでの時間を Python の `time.time` メソッドを用いて計測した。表1はノード数10のときのグループ数ごとの1回のグループピング処理時間を表している。

	グループ数 3	グループ数 4	グループ数 5
1 回目	0.000958681 秒	0.00898242 秒	0.006782532 秒
2 回目	0.000364304 秒	0.00102973 秒	0.00041604 秒
3 回目	0.003001928 秒	0.00186348 秒	0.001536131 秒
4 回目	0.000119448 秒	0.005980015 秒	0.00406456 秒
5 回目	0.006360769 秒	0.020363808 秒	0.004426003 秒
6 回目	0.000958681 秒	0.001876116 秒	0.000371695 秒
7 回目	0.000364304 秒	0.000665426 秒	0.005215883 秒
8 回目	0.003001928 秒	0.001770496 秒	0.000584841 秒
9 回目	0.000119448 秒	0.000597239 秒	0.000583649 秒
10 回目	0.006360769 秒	0.004850388 秒	0.000700712 秒
平均	0.00172708 秒	0.004797912 秒	0.002468204 秒

表 1 ノード数 10 のときのグループ数ごとのグループピング処理時間の表

グループ数3のときの標準偏差が0.001815992, グループ数5のときの標準偏差が0.002285664 なのに対してグループ数4のときの標準偏差が0.005800368 であるためグループ数4のときは他のグループ数に比べてばらついており平均処理時間も最長となっていることがわかる。

しかしながら、グループ数4のときの処理時間を見ても平均は高いものの他のグループ数のときの処理時間とあまり変化はなく、グループ数が多ければ多いだけ処理が遅くなる傾向や逆に一つのグループに属するノード数が多ければ多いほど処理が遅くなる傾向が見られなかった。これは、同一ノード数の場合は単純なグループ数や1つのグループに属するノード数ではなく別の要因によって速度が変動していると考えられる。

#### 5. 議論

本稿では、偶数ノード運用であっても一貫性を保ち続けるアルゴリズムを提案した。しかしながら、課題となる点も残されている。

まず、グループの個数についてである。グループ数が少なく各グループのノード数が多い場合とグループ数が多く各グループのノード数が少ない場合での処理速度の差が確認できていない。今回の検証によって同一ノード数では単純なグループ数やグループに属するノード数ではなく別の要因によって速度が変動しているという結果となった。グループの個数についてはグループピング速度だけではなく、`node.list` の再配布等の処理も含めた処理時間を計測し優位性についての検証を検討している。

次にノード間ネットワーク形成についてである。現在ノードネットワークに参加するためには手動での IP アドレスの入力が必要である。これはノードが多くなるにつれて人間の手が介在する場面が増え、人的要因による障害を発生しかねない。そこでブロードキャスト通信を用いた半自動的ノードネットワーク形成を検討している。

## 6. おわりに

本稿では一貫性を重要視する分散システムにおいて多く使われている過半数の同意に基づくアルゴリズムの問題点である、ネットワーク分断の際に 1:1 に分かると一貫性が失われるという点をノードグルーピングとその各グループ内に配置された Primary ノードにより奇数でしか稼働できないという制約を無くしてネットワーク分断に対する障害耐性を向上させる手法を提案した。しかしながら、まだ確実な裏付けを取れていない要素も多い。それらの要素の検証を行っていききたい。

今後は適切なグループの個数とグループ内に配置するノード数の導出やハートビートの実装を行う予定である。

## 参考文献

- [1] Shestak, V., Smith, J., Siegel, H. J. and Maciejewski, A. A.: A Stochastic Approach to Measuring the Robustness of Resource Allocations in Distributed Systems, *2006 International Conference on Parallel Processing (ICPP'06)*, pp. 459–470 (2006).
- [2] Brewer, E. A.: Towards Robust Distributed Systems (Abstract), *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, Association for Computing Machinery, p. 7 (2000).
- [3] Howard, H., Schwarzkopf, M., Madhavapeddy, A. and Crowcroft, J.: Raft Refloated: Do We Have Consensus?, *SIGOPS Oper. Syst. Rev.*, Vol. 49, No. 1, p. 12–21 (online), available from (<https://doi.org/10.1145/2723872.2723876>) (2015).
- [4] Ongaro, D. and Ousterhout, J.: In Search of an Understandable Consensus Algorithm, *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, USENIX Association, pp. 305–319 (online), available from (<https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro>) (2014).
- [5] Arora, V., Mittal, T., Agrawal, D., El Abbadi, A., Xue, X., Zhiyanan, Z. and Zhu Jianfeng, Z.: Leader or Majority: Why Have One When You Can Have Both? Improving Read Scalability in Raft-like Consensus Protocols, *Proceedings of the 9th USENIX Conference on Hot Topics in Cloud Computing*, USA, USENIX Association, p. 14 (2017).
- [6] Zhang, Y., Ramadan, E., Mekky, H. and Zhang, Z.-L.: When Raft Meets SDN: How to Elect a Leader and Reach Consensus in an Unruly Network, *Proceedings of the First Asia-Pacific Workshop on Networking*, New York, NY, USA, Association for Computing Machinery, p. 1–7 (online), available from (<https://doi.org/10.1145/3106989.3106999>) (2017).
- [7] Zhao, W., Melliar-Smith, P. M. and Moser, L. E.: Fault Tolerance Middleware for Cloud Computing, *2010 IEEE 3rd International Conference on Cloud Computing*, pp. 67–74 (2010).
- [8] Balazinska, M., Balakrishnan, H., Madden, S. and Stonebraker, M.: Fault-Tolerance in the Borealis Distributed Stream Processing System, *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, Association for