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一貫性の強化

新宮 隆太[1,a)]　串田 高幸[1]

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 Leader 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 Leader 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の提案と実装を行う．また, 本稿提案手法を用いてノード数 10 でグループ数を 3, 4, 5 個にした際の各グルーピング速度の検証とそれぞれノード数 10, ノード数 20 のときにおいてネットワーク分断の復旧時のクラスタの再構成処理時間の検証を行った．その結果, グルーピング速度の検証では, 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は見られず, クラスタの再構成処理時間の検証ではノード数が多いほど再構成に時間がかかり, ノードが 5 台増えるごとに平均 2.52 秒増加することがわかった．

1. はじめに

背景

今日, 情報システムは金融, 医療を始めとする多くの場所で用いられており, 重要なインフラとしての役割を担っている．そのような中で, 情報システムは障害が発生してもできるだけ正常に動作し続けられる堅牢性を求められている．そこで, 堅牢性を持つ分散システムは大きな注目を浴びている [1].

その分散システムには 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．CP システムのような一貫性を重要視する際に用いられることの多い, Raft を始めとする過半数の同意に基づくコンセンサスアルゴリズムを用いる分散システムではノードが 1:1 に分かれて過半数を取れなくなることがないように奇数台ノードでの運用が前提となっている [3].

課題

一貫性を重視する分散システムで用いられることの多いコンセンサスアルゴリズムでは過半数の同意に基づく挙動を行う．そのためクラスタはノード奇数台で構成することが推奨されている．これは偶数台で構成されたクラスタの際にネットワーク分断でノードが 1:1 に分かれてしまう事があるためである．通常, 奇数台で構成されたクラスタの時にネットワーク分断が発生すると必ず片方のノード群が総ノード数のうち過半数以上のノードと通信可能な状態となる．過半数以上のノードと通信ができなかったノード群は, 自律的に動作を停止することで一貫性を担保している．

しかしながら, 偶数台構成されたの際にノードがネットワーク分断によって 1:1 に分かれるとどちらのノード群も過半数を取れなくなり一貫性を優先して動作を停止するか, 可用性を優先して一貫性を捨てるかのどちらかとなってしまい, 分散システムとしての役割を全うすることができなくなる [4][5].

例えば, ユーザーの預金を管理する分散システムがあるとする．図 1 のように偶数台ノードで構成されたクラスタで運用されていた場合にネットワーク分断が発生したとする．この場合, 3:2 にノードが分断されているため右側の 2 台のノード群が動作を停止する．そこでユーザーが残高 30 万円の口座から 20 万円引き出す．その後別の場所から残高照会を行っても正しく残高が表示される．しかしながら, 図 2 のように奇数台ノードで構成されたクラスタで運

[1] 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

用された場合にノードが 1:1 に分かれるようなネットワーク分断が発生したとする．す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う可能性がある．図 2 では先程の例と同じようにユーザーが残高 30 万円の口座から 20 万円を引き出す．するとネットワーク分断された先のノード群, 図 2 の右側のノード群には 20 万円が出金されたという情報が伝達されないまま稼働し続けてしまう．このままユーザーが別の ATM から残高照会をかけると出金される前の残高である 30 万円という情報が表示されてしまうこととなる．これが一貫性を失っているという状態である．

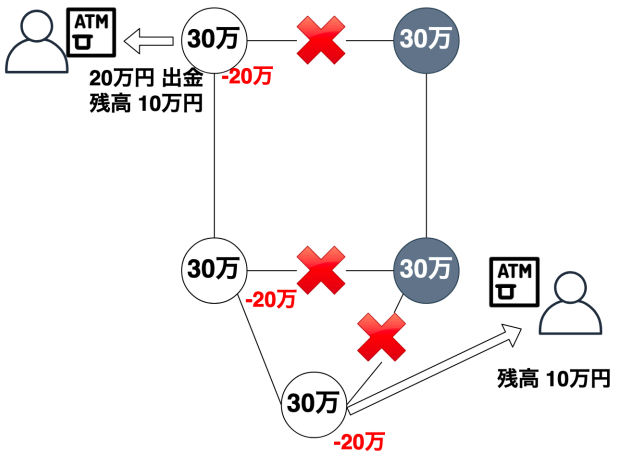

図 1　ノード 5 つの金融システムの例

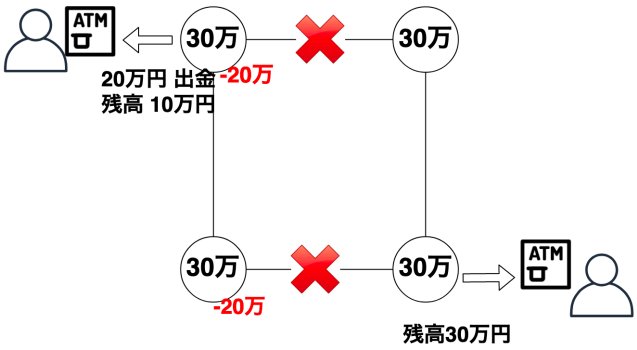

図 2　ノード 4 つの金融システムの例

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．

各章の概要

第 2 章では本稿と関連性のある研究についてとりあげ, 本稿との類似性や違いの説明を行う．第 3 章では本稿の提案であるノードのグルーピングを用いたアルゴリズムについての説明を行う．第 4 章では本稿提案のアルゴリズムの実装についての説明を行う．第 5 章では 4 章で実装したアプリケーションを用いてグルーピング速度とネットワーク分断復旧時のクラスタの再構成処理時間の検証と検証結果について説明を行う．第 6 章では 5 章の検証結果を元に本稿提案手法についての議論を行う．第 7 章では, 本稿提案の総括を行う．

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, それらの中の関連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 [6]．また, Primary ノード以外に Backup ノードが複数事前にスタンバイして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い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

本稿では偶数台で構成されたクラスタにおいてネットワーク分断によってノードが 1:1 に分断された際, ノードを奇数個のグループに分け各グループで選出される Leader ノードと疎通の取れるノードの台数を指標として片方のノード群を停止することにより, 既存手法に存在した奇数台構成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．

表 1 は本稿で用いる Leader ノードと Temporary Leader ノード, Member ノードの定義である．各ノードはクラスタに参加しているノードの IP アドレス, ノードの uid, ノードが稼働開始した時間 (以下 boot_time), そのノードが Leader ノードであるかどうかのフラグが記載された情報 (以下 node_list) を保持している．

表 1 Leader ノード・Temporary Leader ノード・Member ノードの定義

Leader ノード	グループの中から 1 つ選出されるノードで, 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
Temporary Leader ノード	Leader ノードに障害が発生した際に次の新しい Leader ノードが選出されるまでの間, 一時的に Leader ノードの役割を持つノード
Member ノード	Leader ノードでも Temporary Leader ノードでもない場合はこのノードとなる

本稿提案の手法では, まず, 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. その後各グループに属する Member ノードの中からそれぞれ Leader ノードを選出する. グループ内でネットワーク分断が発生した際には疎通の取れる Leader 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が発生してノードが 1:1 に分断されても片方のノード群は動作を停止するため一貫性を維持しつつシステムを稼働し続けることができる.

図 3 ではノードが 10 個稼働しておりそれを 3 つ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と, Leader 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停止することが可能となっている.

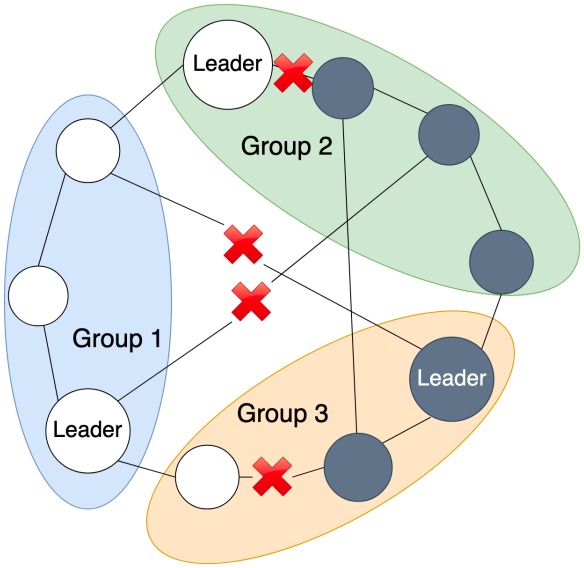

図 3 3 つのグループで構成されたクラスタでネットワーク分断が起きた際の例

本稿では提案をクラスタの形成, node_list 差分情報の配信, ノードグルーピング, Leader ノード選出, ハートビート, 障害の検出, 障害からの復旧の大きく 7 つに分けて説明を行う.

3.1 クラスタの形成

新しくノードをクラスタに追加するにはすでに稼働している任意のノードへクラスタ追加依頼を送信することで可能となる. クラスタ追加依頼にはノード起動時に自動生成された一意な識別子 (以下 node_id), 自ノードの IP アドレス, boot_time, 追加依頼を生成したタイムスタンプが記載されている. クラスタ追加依頼を受け取ったノードは自身の持っている node_list にクラスタ追加依頼に記載された情報を追記して返送する.

3.2 node_list 差分情報の配信

各ノードは自分の持っている node_list に変更があるとその変更において発生した差分の情報を他のノードへ配信する. また, 配信する際, 永遠に同じ差分情報をクラスタ内で配信し続けることを防ぐために送信する際の基本ポリシーを用意する.

自分が Leader ノードのときは Member ノードから送信された差分情報は自分の Member ノードには送信せずその他の Leader ノードへのみ送信を行い, 他グループの Leader ノードから送信された差分情報は自分の Member ノードへのみ送信を行う. 逆に, 自分が Member ノードの時は Leader ノードから送信された差分情報はどこにも送信を行わない. ただし, 他ノードから差分情報受け取って発生した変更ではなく, 新しくクラスタ追加依頼を直接受け取った時, 後述する障害検出において自ノードが障害を最初に検知した時においてはこの限りではなく, もし自分が Member ノードであるならば自グループの Leader へ, 自分が Leader ノードであるならば他グループの Leader ノードと自グループの Member ノードへ node_list の更新差分の送信を行う.

例えば, 図 4 のようにクラスタ追加依頼を受け取ったノードは node_list に新しいノード情報が追加される. するとクラスタ追加依頼を受け取ったノードはこれを Leader ノードへ送信する. Leader ノードは別のグループの Leader ノードと自グループの Member ノードへ差分情報を送信する. これによりすべてのノードに新しいノードがクラスタに参加したという差分情報を配信することができる.

Leader ノードが障害で停止しクラスタの再構築をする際には新しい Leader が再選出されるまでの間, 停止した Leader ノードの代わりに後述する Temporary Leader ノードが選出され, それを介して node_list の差分の配布を行う.

3.3 ノードグルーピング

グループは node_list の差分の配信を行う際や Leader

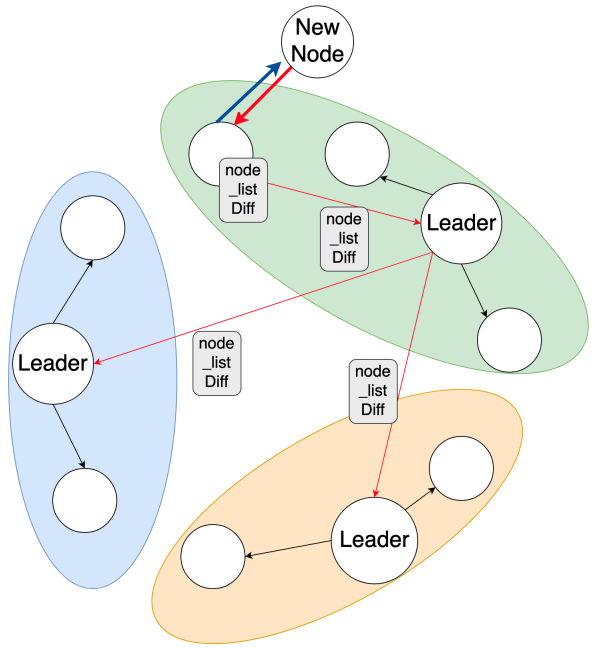

図 4 新しくノードがクラスタに参加する際の node_list の差分配信の例

ノードを選出する際に用いる．グループに属するノードの数に制限はないが，クラスタ全体で奇数個でなければならない．

ノードグルーピングは各ノードがそれぞれ自分の持っている node_list を元に boot_time を参照して行う．

図 5 のようにグルーピングは，boot_time をソートして最近起動したノードと最も昔に起動したノード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．最近起動したノード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が集まったグループを作らないようにグルーピングを行う．

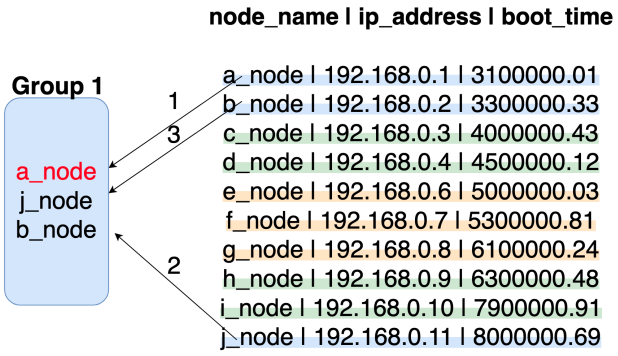

図 5 node_list の boot_time を参照してグルーピングを行う例

boot_time をグルーピングの指標に用いる理由としては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．これは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ような動作の不安定なノードをいち早く検出する事ができる．

また，各ノードでそれぞれグルーピングの処理を行うためどのノードから見ても同じ指標である boot_time を参照することでノードごとにグルーピングの結果が変わってしまうことを防止することができる．

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 node_list 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 20 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 Leader 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(
           node_list)
3      grouped_node_list = list()
4      node_list_length = len(sorted_node_list)
5
6      for i in range(group_num):
7          flag = True
8          for j in range(int(node_list_length /
               group_num)):
9              if flag:
10                 flag = False
11                 sorted_node_list[0]['is_leader']
                       = False
12                 sorted_node_list[0]['group_id'] =
                        i + 1
13                 local_grouped_list.append(
                       sorted_node_list.pop(0))
14             else:
15                 flag = True
16                 sorted_node_list[-1]['is_leader']
                        = False
17                 sorted_node_list[-1]['group_id']
                       = i + 1
18                 local_grouped_list.append(
                       sorted_node_list.pop(-1))
19
20         local_grouped_list[-1]['is_leader'] =
               True
21         grouped_node_list.extend(
               local_grouped_list)
```

3.4 Leader ノード選出

Leader ノードは node_list の差分を配信，後述するネットワーク分断の判定，ネットワーク分断発生時の生存するノード群を決定の際に用いる．

Leader ノードはグルーピング終了後，各グループ内で最も boot_time が古いノードがそれぞれ選出される．これは，

グルーピングの際に boot_time を用いた理由と同じで, 再起動を繰り返す不安定なノードが Leader ノードに選出されないようにするためである.

また, Leader ノードが動作を停止した際に次の Leader ノードが選出されるまでの間, 一時的に Leader ノードの役割を担う Temporary Leader ノードは Leader ノードの次に boot_time が古いノードが選出される.

3.5 ハートビート

ハートビートは正しくクラスタ内のノードが稼働しているかを相互監視する際に用いる通信である. 自身が Leader ノードの場合はその他の Leader ノードと自グループの Member ノードへ, 自身が Member ノードの場合は自グループの Leader ノードと自グループの自分以外の Member ノードへハートビートを送信する.

ハートビートの中身は非常に単純でリクエストを識別するための uid とリクエストを生成したタイミングのタイムスタンプの 2 つだけである.

3.6 障害の検出

先述したハートビートを送信してその結果が帰ってこなかった場合, 何らかの障害が発生しているということがわかる.

その障害がノード単体の故障なのかそれともネットワーク分断なのかを判定するためにハートビートを送信したノードはクラスタ内に存在する全ての Leader ノードへハートビートを送信する. もしクラスタ内に存在する Leader ノードの過半数以上から応答があった場合はノード単体の故障である, もしくはネットワーク分断が発生しているが自身は過半数以上の Leader ノードと疎通が取れているため動作を停止する必要がないと判断する. この場合, 応答のなかったノードを自身の持っている node_list から削除し, その差分をクラスタの形成のときと同じように各ノードへ配信する.

逆に, クラスタ内に存在する Leader ノードの過半数以上の応答が無かった場合はネットワーク分断が発生し, さらに過半数以上の Leader ノードと疎通を取ることができなかったため自身の動作を停止しなければならないと判断する. この場合, 自身はクラスタから離脱する.

3.7 障害からの復旧

ネットワーク分断によってそれぞれ能動的に動作を停止したノード群はネットワーク分断が解消された際には再びクラスタに参加するために復旧処理を行う.

ノードがネットワーク分断によって自ノードが動作を停止しなければならないという判定を下すと, 実際には完全にノードの動作を停止するのではなく各ノードが持っている稼働ステータスを”停止”に変更する.

稼働ステータスを”停止”に変更すると, 他ノードからの node_list の差分情報配信, クラスタ追加依頼, ハートビートをすべて受け取らない状態となる. これにより, クラスタに参加するノードとして必要な機能を失い, 他ノードと疎通を取る手段を失うため擬似的に動作を停止した状態となる. この状態で自身が持っている node_list に記載されている複数の Leader ノードへ一定間隔でハートビートの送信を行う. このハートビートの応答が正しく返ってきた時, ネットワーク分断が復旧したと判断する. ネットワーク分断が復旧したと判断されると自身の稼働ステータスを”停止”から”稼働”に変更し, ハートビートを送信したノードへクラスタ追加依頼を再度送信することでクラスタに復旧することができる.

4. 実装と実験環境

4.1 実装

本稿提案のアルゴリズムの実装と GUI でクラスタの状態を確認するための WEB アプリケーションの実装を行う.

本稿提案のアルゴリズムは, 図 6 のようなクラスタ形成, ノードのグルーピング, ハートビートの送信, Leader ノード選出, 障害の検出を行う NodeGroupingApplication と他のノードからの rpc の待ち受けや node_list のノード間共有を行う gRPC サーバーの実装を行った.

また, クラスタの状態を確認できる Web アプリケーションは, 図 7 のように各ノードがどのグループに属しているのか, どのノードが Leader ノードとして選出されたのかを確認できるすることができるようになっている. さらに, 擬似的なネットワーク分断の発生, 復旧の操作も行うことができる.

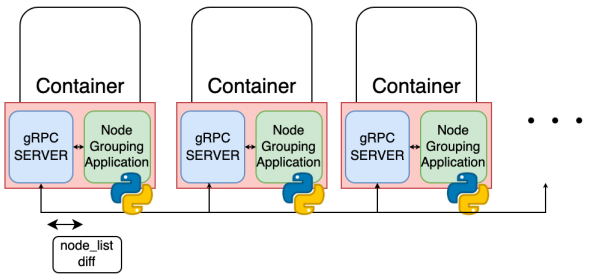

図 6 NodeGroupingApplication と gRPC サーバーの概要図

いずれのソースコードも GitHub のリポジトリ[*1]から確認することができる.

実装は主に Python3.8 で行い, ノード間通信では gRPC, ノード自身の 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, ノードの 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

[*1] https://github.com/homirun/node_grouping_v2

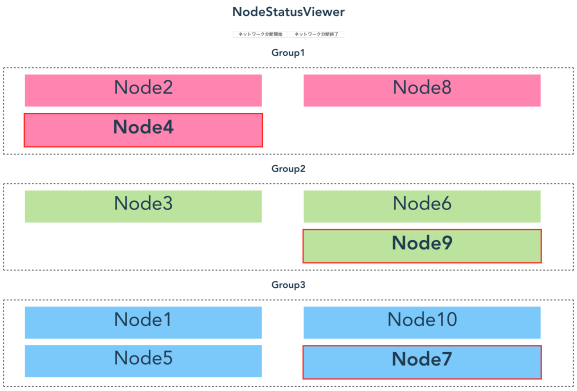

図 7　ノードの状態を確認できる Web アプリケーション

uptime, Web アプリケーションのバックエンドサーバとして Flask, フロントエンドには Vue.js, axios を用いた.

図 8 は NodeGroupingApplication と gRPC サーバーの処理の大まかな流れを示している.

NodeGroupingApplication を起動すると, まず最初にエントリポイントである main メソッドが呼び出される. 初期処理として init メソッドが呼び出され自ノードの IP アドレスと boot_time の取得, node_id の生成, server Process の起動を行い後述する server.py の serve メソッドを呼び出して gRPC サーバーを起動し rpc を待ち受ける. 初期処理が終了すると, create_node メソッドを呼び出し, init メソッドで生成した node_id, boot_time, IP アドレスをもちいて自ノードの情報だけが入っている node_list を生成する. node_list を生成すると throw_add_request メソッドを使ってすでにクラスタに参加しているノードへ先程生成した自ノードの情報のみが記載された node_list を含むクラスタ追加依頼を送信し, 返ってきた node_list に手元の node_list へ上書きする. これによりノードがクラスタに参加したことになる. クラスタに参加すると自分の持っている node_list の情報を元にして throw_heart_beat メソッドを呼び出してハートビートを送信する. これを定期的に繰り返すことで障害発生時に検知をすることができる.

ハートビートが失敗し続けて障害が発生したと判断されると request_heart_beat_for_leader メソッドを呼び出す. このメソッドはグループに関係なく全ての Leader ノードへハートビートを送信し, 全体の過半数以上の Leader ノードと疎通が取れるかを検証する. 過半数以上の Leader ノードと疎通が取れた場合はハートビートが失敗したノードを node_list から削除する. 全体の過半数以上の Leader ノードと疎通が取れない場合, _down_node メソッドを呼び出し, ノードの状態を表すフラグを False に変更して server Process を停止する.

また, 他のノードから throw_add_request を受信すると RequestServiceServicer クラスの add_request メソッドが発火, update_request を受信すると update_request メソッドが発火, throw_heart_beat を受信すると, heart_beat_request が発火, request_heart_beat_for_leader を受信すると request_heart_beat_request が発火する.

さらに, node_list が更新されるたびに適宜 node.py の grouping メソッドが呼び出され, grouping が実行される.

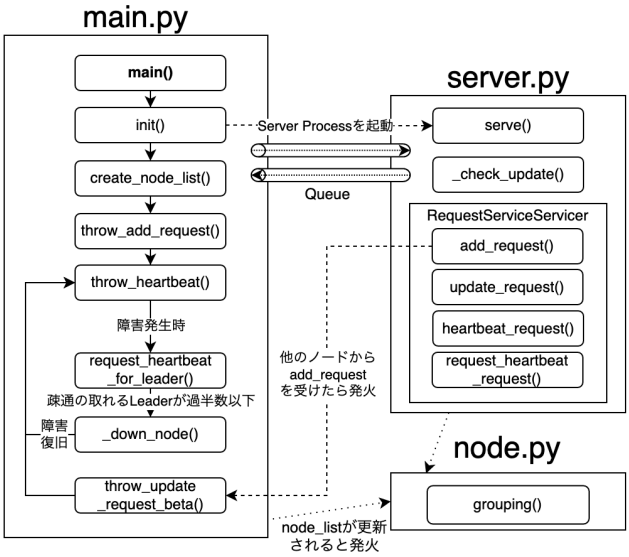

図 8　NodeGroupingApplication と gRPC サーバーの処理の流れ

4.1.1　クラスタへの参加

NodeGroupingApplication は起動するとまず最初に server プロセスを立ち上げる. その後 server プロセスから別ノードへソースコード 2 の proto ファイルの定義に従ってクラスタ追加依頼としてリクエストを識別するために uuid.uuid4 メソッドを用いて生成した一意な ID(request_id) と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(node_id), 送信元 IP アドレス (sender_ip), ノードが起動した時間 (boot_time), 追加依頼のリクエストが生成された際の時間 (time_stamp) を送信先の gRPC サーバーへ送信する.

ソースコード 2　クラスタ追加依頼のリクエストを定義した proto ファイル

```
1 message AddRequestDef {
2   string request_id = 1;
3   string id = 2;
4   string sender_ip = 3;
5   double boot_time = 4;
6   double time_stamp = 5;
7 }
```

追加依頼を受け取ったノードは main プロセス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. それをソースコード 3 のような形で gRPC サーバーを介して依頼送信元へ返す. レスポンスは AddResponseDef

の定義にあるように node_list と time_stamp を送信する．node_list の形式は 6 行目以降で定義されている．Node はノードの情報を示している．上から順に，リクエストを識別するために uuid.uuid4 メソッドを用いて生成した一意な ID(request_id) と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(id), IP アドレス (ip), ノードが起動した時間 (boot_time), 所属しているグループ ID(group_id), そのノードが現在 Leader ノードかどうか (is_leader) の 5 つの情報を示している．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 を更新してクラスタへの参加処理が終了する．

ソースコード 3 クラスタ追加依頼のレスポンスを定義した proto ファイル

```
1  message AddResponseDef {
2    string request_id = 1;
3    repeated Node node_list = 2;
4    double time_stamp = 3;
5  }
6
7  message Node{
8    string id = 1;
9    string ip = 2;
10   double boot_time = 3;
11   uint64 group_id = 4;
12   bool is_leader = 5;
13 }
```

4.1.2 ノードグルーピングと Leader 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 を第 1 キー,node_id を第 2 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に同じ一つの group_id を付与している．また Leader 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 Leader 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 main プロセスと server プロセスでそれぞれ別の node_list を持っており，そのプロセス間の node_list の更新を検知し正しく更新しなければならない．

実装では，どちらのプロセスもそれぞれ node_list の更新が発生すると Queue にソースコード 4 のような形式で追加される．diff_list は今回変更された node_list の差分，method はその差分が追加か削除かを示し，is_allow_propagation はこの差分の更新を他のノードへ送信していいのかを示している．

もう一方のプロセスが定期的に Queue からそのデータを取得し，更新することによってそれぞれのプロセスで持っている node_list の整合性を保っている．

ソースコード 4 Queue 間でやり取りされるデータ形式

```
1 {
2     'diff_list': {'id': 'xxxxx-xxxxx-xxxxx', '
          ip': '192.168.124.1', 'boot_time'
          :34443275.32},
3     'method': 'add',
4     'is_allow_propagation': False,
5 }
```

4.1.4 ハートビートと障害の検出

ハートビートは Queue の中身の確認を 3 回するごとに 1 回ハートビートを送信する．ハートビートは 500ms のタイムアウトで失敗と判定する．ハートビートを 3 回連続で失敗すると障害が発生したと判断する．

障害が発生したと判定された場合，全ての Leader ノードへハートビートを送信し，ネットワーク分断かどうかの判定を行う．

4.2 実験環境

macOS 10.15.6 がインストールされた MacBook Pro(Late2019, Intel i7, RAM:16GB) で Docker Desktop for Mac 2.4.0.0 を用いて図 9 のように前の章で説明した NodeGroupingApplication と API サーバーが実行可能な Docker コンテナを複数立てた．

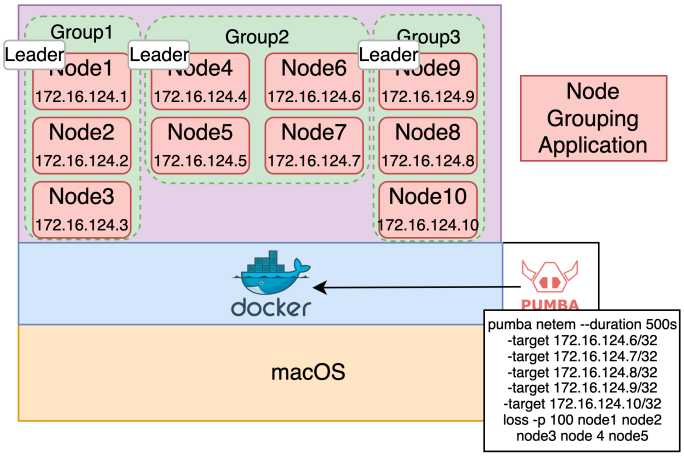

図 9 実験環境の構成図

5. 評価と分析

5.1 グルーピング処理時間とグループ数の関連性

ノード数が 10 台でグループ数がそれぞれ 3, 4, 5 個のときのグルーピング処理時間を Python の time メソッドを

用いて試行回数 10 回で計測した．この実験は事前の計算では処理のステップ数的にグループ数が多ければ多いほど処理時間が遅くなる傾向になることが見込まれた．

グループ数 3 のときの平均グルーピング処理時間が 1.7 ミリ秒, グループ数 4 のときは 2.7 ミリ秒, グループ数 5 のときは 2.5 ミリ秒であった．また, グループ数 3 のときの標準偏差が 1.8, グループ数 5 のときの標準偏差が 2.3 なのに対してグループ数 4 のときの標準偏差が 2.7 であった．さらに, 図 10 の箱ひげ図からも分かるようにグループ数 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, システムが計測できる時間精度上誤差の範囲であるといえ, 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．処理時間にばらつきがあるのは実験マシンのリソースが不足していた若しくは OS のスケジューリングによって引き起こされていると考えられる．これらはより高性能なマシン上で 10 回, 100 回など複数回グルーピングを行い計測し, それらを合算したものを比較することや, 検証を行うノード数を増やすことで傾向を見ることができると考えられる．

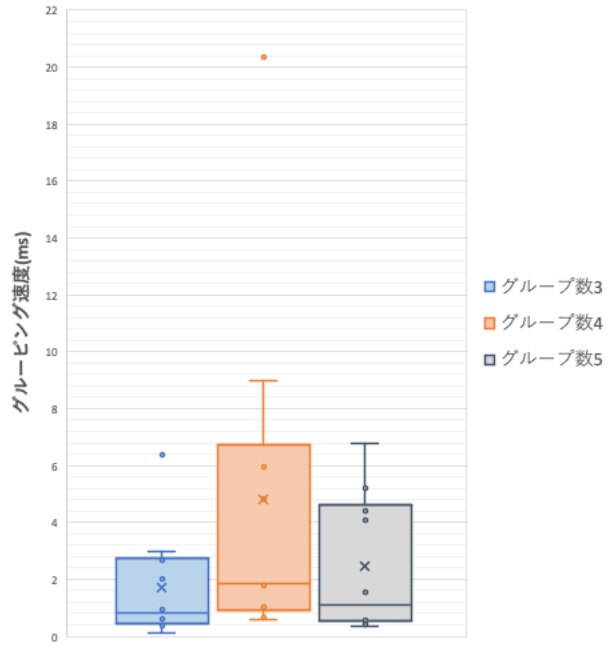

図 10　グルーピング速度の箱ひげ図

5.2　ネットワーク分断発生からクラスタ再形成までの処理時間

グループ数が 3 でノード数が 10 台と 20 台の時の 1:1 に分かれるようにネットワーク分断が発生し, 分断が解消された際のクラスタが元の形に戻るまでの時間を Python の time メソッドを用いて試行回数 10 回で計測した．この実験は復旧するノード数が多いほど処理時間がかかる傾向になることが見込まれた．

表 2 は障害復旧時のクラスタの再構成処理が終了するまでの時間を示している．ノード 10 台の時の平均復旧処理時間は 1.15 秒, ノード 20 台の時の平均復旧処理時間は 3.67 秒であった．ノード 10 台のとき復旧するノード数が 5 台, ノード 20 台のとき復旧するノード数が 10 台であるため復旧するノードが 5 台増えるごとに平均 2.52 秒処理時間が増加する事がわかり, 事前の見込み通りであることが確認できた．

	ノード 10 台	ノード 20 台
1 回目	1.47 秒	3.59 秒
2 回目	1.41 秒	4.01 秒
3 回目	1.37 秒	3.62 秒
4 回目	1.47 秒	3.61 秒
5 回目	0.89 秒	3.59 秒
6 回目	0.88 秒	3.14 秒
7 回目	1.37 秒	3.59 秒
8 回目	0.89 秒	3.94 秒
9 回目	0.86 秒	3.86 秒
10 回目	0.87 秒	3.66 秒

表 2　障害復旧時のクラスタ再構成処理時間

6.　議論

本稿では, 偶数台ノードで構成されるクラスタにおいても一貫性を保ちつつ運用しづつけることのできるアルゴリズムを提案した．しかしながら, 本稿の提案にはまだ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 3 個など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 4 個など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の計測やノード数を 10 ではなく 30 などより多く増やしての処理時間の計測を行い, 再検証を行う事を検討している．

次に通信量の多さである．ノードが 1:1 に分断された際に自分が動作を停止するべきかの判定を行うためのリクエストやクラスタに属するノード情報を各ノードが持っており, それを伝搬するためのリクエストが存在している．このため既存手法に比べて多く通信を行っている．この問題に対して, 通信のピギーバックを用いることで通信回数を抑えることを検討している．

7. 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 に分かれると一貫性を維持しつつ運用することができないという点をノードのグルーピングとその各グループ内に配置された Leader ノードを用いることで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. また, 本稿提案手法を用いてノード数 10 でグループ数を 3, 4, 5 個にした際の各グルーピング速度の検証を行った. その結果, 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は見られなかった. さらに, ノード数 10, ノード数 20 のときにおいてネットワーク分断の復旧時のクラスタの再構成処理時間の検証を行い, ノード数が増加するほど再構成処理時間が増加するということがわかった. 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, ノード間における通信の最適化を行う予定である.

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] Arora, V., Mittal, T., Agrawal, D., El Abbadi, A., Xue, X., Zhiyanan, Z. and Zhujianfeng, 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).

[5] 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).

[6] 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).

[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 Computing Machinery, p. 13–24 (online), available from ⟨https://doi.org/10.1145/1066157.1066160⟩ (2005).