分散合意アルゴリズムにおけるリーダーを含む全ノード参加型の選挙によるビザンチン故障耐性とクラスター内の合意

岡田 大輝[1]　小山 智之[1]　串田 高幸[1]

概要：分散合意アルゴリズムでの Byzantine fault tolerance の達成は，各ノードとの密な通信によって実現される．PBFT(Practical Byzantine Fault Tolerance) アルゴリズムは実用的ではあるが，合意形成時の通信コストは $O(N^2)$ であり，ノードが増加するごとにその通信コストは大幅に増加する．そこで，選挙形式の合意形成を行うアルゴリズムを提案する．このアルゴリズムは，クラスター内の各ノードを leader もしくは follower の 2 つの役割に分ける. そして，leader を含めた全ノード参加型の多数決による選挙を行う. この合意選挙は,leader からマルチキャストで送信されたデータを受信した follower がランダムのタイムアウトをトリガーに, 自身が leader から受信したデータを他ノードに送信することにより開始される. 他の follower からの選挙開始と共にデータを受信した follower は自身が leader から受信したデータと一致するか比較を行う.leader はクライアントから受け取ったデータと一致するかを比較を行う. 一致した場合はその選挙に応答する. その応答が leader を含む過半数のノードから送信されたことを確認できた場合, 選挙は成功であり, クラスター内の定足数ノードに複製されたことになる. その後,leader がコミットの許可を出す. この選挙形式の合意形成手法を採用することにより,PBFT のような各ノード間の密な通信を行わずに各ノードの受信状況を把握できる. 結果として, 合意形成に至るまでの通信回数を大幅に削減できる. このアルゴリズムを Kubernetes 環境でコンテナに実装し，合意形成にかかる通信コスト (通信回数) を評価する．本提案アルゴリズムの合意形成に要する通信回数の平均値と PBFT の通信回数を算出して比較した結果,PBFT の約 76 %以下のコストに削減することが期待できた.

1. はじめに

背景

今日の情報化社会を担うシステムの多くは，複数のコンピュータで連携や分担させることにより処理を分散しつつ，あたかも一つのシステムのように動作させる分散システムの形態をとっている. この形態を採用する事によりスケーラビリティや耐故障性において恩恵を受けることが出来る.

分散システムの構築では，システム全体でのデータの一貫性の確保や各マシンに同じ命令を実行するために，どのようにデータや命令を各マシン間で一つに合意し複製するかという点を設計，実装する必要がある．そして，一般的に分散合意アルゴリズムを実装することによりこの問題を解決する．分散合意アルゴリズムは，システムが一貫したグループとして機能するようサポートする．これにより，クラスターを構成している一部のノードに障害が発生してもシステムが動作できる耐障害性を獲得出来るようになり，大規模なシステムの構築において重要な役割を果たしている [1].

1.1 分散システム

分散システムの定義については, 今日では多くの説明がなされているが, タネンバウムがその著書で完結にまとめている. いわく, 分散システムとは, ユーザに対して, 単一で一貫性のあるシステムとして動作する独立したコンピュータの集合である, と記している. 本稿でも, 分散システムの定義を採用し, この定義に則ったシステム形態を分散システムとして理解する. この定義で分散システムの重要な側面として挙げられることの一つは, 独立したコンピュータの集合であるという点である. 個々が単一のコンピュータとして完結しているマシンを協調動作させているという事だ. 分散システム向けにハードウェアが構成されていたり, 他のマシンと協同しないと動作できないといった単一で一つのコンピュータとして動作できないマシンの集合でなければ分散システムと定義されないというわけではない. さらに付け加えるならば, 構成するコンピュータの種類には何も制限がないという事もいえる. 高性能なコンピュータ

[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

や小さなマシンの集合であったり, ハードウェアや性能が異なるコンピュータで構成することも出来る.

1.2 透過性

本項では, 分散システムの分野に存在する概念の一つである透過性 (透明性) について触れる. 透過性とは, 分散システムに存在するあらゆる側面をユーザから隠ぺいする性質の事を指す. 以下に, 主な透過性の種類を示す.

- **アクセス**
  データ表現の差異やリソースへのアクセスがどのように行われているかを隠ぺいする
- **ロケーション (位置)**
  その分散システムが保有しているリソースがどこに存在しているか, そのオブジェクトの所在を隠ぺいする
- **リロケーション (移動)**
  システムムノ使用中にリソースが、別の場所に移動できることを隠ぺいする
- **ミグレーション (再配置)**
  リソースが別の場所に移動する可能性があることを隠ぺいする
- **レプリケーション (複製)**
  リソースが複製されるていることを隠ぺいする
- **並行**
  システムのリソースが複数の独立したユーザによって共有される可能性があることを隠ぺいする
- **障害**
  リソースの障害と回復を隠ぺいする

アクセス透過性とは, データ表現の相違やシステムが保有, 活用しているリソースのアクセス方法の違いをユーザに認識させない事である.1.1 項で, 分散システムの構成においては, 異なるコンピュータの集合によって構成されうる事を示した. そのマシン間の違いは, ハードウェアの種類や性能の違い,OS の違いであり, それらの差異はデータの保持方式の違いやリソースへのアクセス方法の違いを生むことになる. 単一のシステムとして動作するにはこの違いをユーザから隠さなければならない.

ロケーション (位置) 透過性は, システムのリソースが物理的にどこに存在するかを不明にすることである. 例えばフォールトトレランスのために, リソースを複数の場所に存在するデータセンターに物理的に分散することがある. これにより, システムの所在地で災害が発生しても被害のない場所に存在するデータセンターにリソースがあることによってシステムの完全停止を防ぐことが出来る. この時にシステムを使用しているユーザにこのリソースが物理的にどこに存在するのかを意識させないようにする事が位置透過性である. これは同時に, リソースがどこに存在してもユーザへ同じように機能を提供できる必要があるということと不可分である.

リロケーション (移動) 透過性は, システムのリソースが, アクセスに影響なくマシン内やマシン間で移動することを隠すことである. 例えば,web ページのファイル (index.html) の位置の変更があったとしても, 変更前と変更後で全く同じようにアクセスできる場合, 移動透過性があるといえる. また, その web ページの閲覧中に, 閲覧しているページのファイルの場所に変更があった場合でもユーザに変化なく閲覧, アクセスが出来る状態を提供できているならば, ミグレーション (再配置) 透過性があるともいえる.

レプリケーション (複製) 透過性は, リソースが複製されてシステム内で複数同じオブジェクトを所持していることをユーザから隠すことである. この透過性によってユーザからは単一のリソースに見せる. 複製は, フォールトトレランスにおいては基本的な手法の一つである. その複数のコピーがある状況において, 常にユーザが単一でオリジナルのデータを使用しているようにすることが重要である.

並行透過性とは, 同システムを利用している複数のユーザをそれぞれのユーザから意識させないようにすることである. これは, システムを利用しているユーザが同時に複数人存在していても, 自分一人でシステムの利用やリソースのアクセスが出来るようにする必要がある. 互いに影響を与え合わないアクセスや操作であれば困難な場面になる事は少ない. しかし, 競合する処理を複数人で行う場合, この透過性は複雑な問題になりえる.

障害透過性は, 分散システム内部で発生した障害やその回復をユーザにわからないようにする性質である. 独立したコンピュータの集合である以上, 構成しているマシンが個々に故障を起こす. 一部の故障は直ちに分散システムを停止させることはない場合が多いが, のちに一貫性の保持において重大な問題を引き起こす可能性がある. それらの故障やそこからの回復がなされた前後において, ユーザにそれを感じさせず, 変わらずにシステムを利用できるようにすることが, 障害透過性の達成において考えなければいけないことである.

1.3 State Machine Replication

State Machine Replication(以下,SMR) は 1980 年代に提案されたフォールトトレランスに関するアプローチだ [2]. State Mahicne(以下ステートマシン) とは, 有限個の状態と他の状態へ移る遷移, その状態での動作でコンピュータを抽象化したモデルである．そのステートマシンがとれる状態のうちどの瞬間においても必ず何かの状態を取っている．そして複数の状態を同時にとる事もない．ステートマシンは有限オートマトンとも呼ばれている．ステートマシンの性質の一つとして決定論的であることが挙げられる．つまり, 同じステートマシンが, 同状態であるときに, 同じ入力を同じ順序で処理した場合, 出力される処理結果も同一であるという事である.

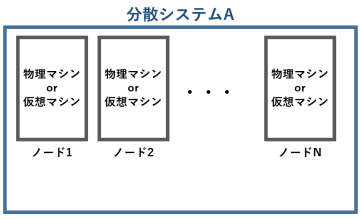

図 1　一般的な分散システムの構成

前述したような性質を持つステートマシンを複製する事により, フォールトトレランスの問題を解決しようとする概念が,SMR である [3]．複数の同ステートマシンを構成することにより, 一部に故障が発生しても, 正常に動作している複製されたステートマシンによって, 入力を処理することが出来る．そして, ステートマシンは決定論的である．そのため, 代わりに入力を処理したステートマシンの出力結果は, 本来その処理を行うはずだった故障したステートマシンが処理した時の出力結果と同一である．以上の事から,SMR は今日の分散システムのフォールトトレランス設計の主要な手法となっている．

そこで問題になるのが、どのように同一のステートマシンを複製し保持するかという点である．SMR は, あくまでコンピュータの動作や設計を抽象化したモデルである．構成する各マシンの性能やそれを取り巻く環境を考慮してはいない．分散システムの構成として, 同一性能のマシンで設計, 実装する事はそれほど困難なことではない．しかし, 外部環境も考慮すると, この問題は複雑になる．その最たるものがネットワークである．各マシンのネットワークの閑居や性能が同一であること, 同一であり続ける事に関して開発者が保証する事は困難な事である．実際の環境において, ネットワークは遅延や途絶, 分断を起こす可能性を持っており, 常に一定であることは期待できない．ステートマシンが決定論的であるという性質は, 前述した通り同じ入力と順序である事を条件の一つとしている．つまり, 何らかのネットワーク異常により各ステートマシンの入力に差異が生まれた場合 (例として, ネットワークの途絶により入力が受信できない, 遅延を原因とした入力順序の変化の発生),SMR の体制は瓦解する事になる．瓦解すれば, その分散システムが単一のシステムとしての一貫性や可用性を著しく損なう事になる．よって,SMR の概念において、どのようにステートマシンの複製と保持を実現するかは重要な課題である．そして, 分散合意アルゴリズムが解決すべき問題は, この SMR の問題と置き換えることも出来る．

1.4　分散合意アルゴリズム

分散合意アルゴリズムとは，分散アルゴリズムの一つに分類されるアルゴリズムであり, 合意プロトコルとも呼ばれる．これは，分散している各ノードで共通の決定に合意するためのアルゴリズムであり，主にレプリケーションの部分を担っている．

前節で述べた通り, 分散システムは異なる複数の物理マシン (時には仮想マシン) を協同させて 1 つのシステムとして動作させるという性質を有している．つまり, 一般的に, 図 1 に示すような構成に抽象化することが出来る. 図の青枠は, 単一の分散システム (ここでは,「A」と呼称) を示している．青枠内の黒枠は分散システム A を構成しているマシン (以下ノード) を示している．図中の各ノードは単一の分散システム A を動作させるために協同している．しかし, 各ノードはそれぞれが単一のマシンであり, 各ノードは, 物理的もしくは論理的に独立している．また, 各ノードは分散システムを動作させるために, 同じ動作を担当することもあれば, 違う役割を担う事がある．

異なるノードに同じ役割を与え, あたかも単一のマシンで動作しているように協同させる場合に考慮しなければいけない問題の一つは, レプリケーションについてである．つまり, 異なるノードに如何にして同じデータ, 同じプロセスを与え, コミットさせるかという点である．

1.5　故障モデル

分散システムは, 単一のマシンのように動いているが, その構築はとても困難である．1 つに, 構成しているノードは独立したマシンであるため, 各ノードがそれぞれのタイミング, 原因で故障を起こす．とあるノードは正常に動作しているのにも関わらず, あるノードは故障しており, 動作を停止している状況が起こりうるのだ．分散システム内において起こる故障は図 2 で示すように, その複雑さによって階層化されている．

図 2 内の矩形の枠と内部の名称は, 各故障モデルを表している．そして左に示している上方向の矢印はその故障モデルの複雑さを表している．この図の場合, 上に重なっている故障モデルほど, 複雑であることを示している．

Fail-Stop は, 一般に故障と聞いて思い浮かべるマシンの状況に最も近しい故障モデルである．つまり, マシンは故障により完全にその動作を停止している状態であり, 他ノードからの通信への応答も不可能ある．そのため, 故障の判別は比較的容易である．しかし, 分散システムを構成するノードは 1.2 節で述べた通り, 物理的に分散している. つま

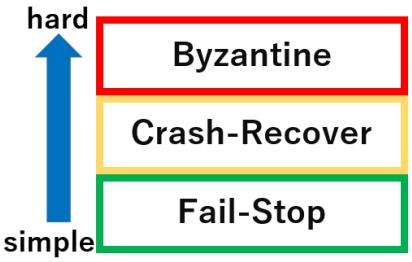

図 2　分散システムで定義されている故障モデル

り, 単一の分散システムを構成しているノードがその距離を問わず, 地理的に分散している可能性が考えられる．それは, 場合によっては国を跨いでいる可能性も今日では考えなくてはならない．つまり, ノード間のネットワークの性能は各ノード間で異なるという事態が発生する (あるノード間では高速で安定した通信が可能であり, 別のノード間では, 低速で不安定な通信の可能性がある)．故障の判別に ping による疎通確認を使った場合, それが, ネットワークの性能にその原因があるものなのか, マシンの故障による完全な停止によるものなのかまで判断する必要がある場合, 問題はより複雑になる可能性がある．

Crash-Recover は, 先に述べた Fail-Stop 故障モデルよりも複雑なモデルとして定義されている．この故障モデルの特徴は, 故障していたと思われていたノードが任意のタイミングで回復し, 処理に参加してくる可能性を考慮する点である．その原因は, 単純な故障の可能性もあるが, 先に述べたようなネットワークの性能に起因するものである可能性も考えられる．一見, 自動で復旧してくれるので, システムとしては良い現象のように見える．しかし, マシン間のレプリケーションや合意という視点で見ると大きな問題となる．動作が停止し通信が出来ない故障状態に陥ったノードは, その期間内に発生した分散システム内の変更を共有できていないのである. そのノードが復帰して処理に再参加する. 同じデータや命令を同じ順序で実行することにより単一のシステムとして動く必要がある中で, 周りのノードとは全く違う状態のノードが突然参加してくるのである. これは, 分散システム内の一貫性や整合性に問題をもたらす可能性がある. 実装によっては, 一つの故障ノードがその他の大多数のノードに影響を及ぼし, 分散システム内の状態を予期せぬタイミングで, 予期せぬ状態に変えてしまうことも起こりうる.

Byzantine 故障モデルは, 定義されているモデルの中で最も複雑かつ解決が困難な故障モデルである. 詳しくは次節で述べるが, この故障モデルの特徴は一般的な故障や自然な回復からの再参加以外に, 任意のタイミングで任意の動作をする可能性を想定する事である. つまり, 本来の想定している動作以外の動きをするノードが出現するのである. 原因としては,1 つに悪意ある第三者による分散システム内への介入が考えられる. 昨今の情報化社会においては, 情報セキュリティの課題は尽きない. 分散システムも例外ではない. 複数のマシン間で相互に通信しながら協同している性質上, システム内の至る所にサイバー攻撃を受ける可能性を含んでいる. さらに, 外部からの攻撃ではなくシステムを利用している内部 (例えば元従業員や業務を請け負っている業者) からの攻撃も報告されている [4].2 つ目に実装のミスといったヒューマンエラーも考えられる. テスト段階で正常な動作が確認できたとしても, それが起こりうる全ての状況において正しい動作を保証するものではない.3 つ目に不具合やバグといったものに起因するものだ. これは, マシン内のシステムのみならず, ノード間のネットワークで発生する可能性もある.

この故障モデルの定義において重要なことの一つに, その故障モデルより下の故障モデルに対応できないときは, その故障モデルに対応することは出来ないという事である. つまり,Fail-Stop 故障モデルの環境下で正常に動作できないのであれば,Crash-Recovery 故障モデルで動作することも出来ない. 同じように Crash-Recovery 故障モデルを想定する環境下で正常動作できないのであれば,Byzantine 故障モデル環境下で正常に動作することは不可能である. 言い換えれば, その故障モデルで正常に動作することが出来るならば, それ以下の故障モデルでも正常に動作できるという事である. このことは, 各故障モデルの関係性を考えれば明らかな事である.

まず,Fail-Stop 故障モデルと Crash-Recovery 故障モデルの関係を考える. 各故障モデルの概要については前に述べたとおりである. ここから,Crash-Recovery 故障は,Fail-Stop 故障に「自然に回復し, システムに再参加する」という状況を付け加えたものととらえることが出来る. つまり,Fail-Stop 故障に想定する状況を純粋にプラスしたようなものだと言い換えることが出来る. このように考えると, そもそもの Fail-Stop 故障の対応が出来ないと,Crash-Recovery 故障に対応できないという事が理解できる.

そしてこれは,Crash-Recovery 故障モデルと,Byzantine 故障モデルとの間にも, 同様の関係性が見て取れる.Byzantine 故障を想定した場合, ノードが起こしうる動作は列挙しきることが困難なほど存在する. つまり, 任意のタイミングで Crash-Recovery 故障を起こすことが出来るという事である. さらに, その故障は修正が行われない限り上限なく行われるかの可能性も考えれる (つまり, ひたすら故障と回復を短いサイクルで繰り返し続けるかもしれない). 以上の事から考えると,Crash-Recovery 故障に対応できないのであれば,Byzantine 故障に対応することが出来ないという事は明白である.

1.6 分散合意アルゴリズムと透過性

本項では, 分散合意アルゴリズムを 1.2 項で触れた透過性の達成の視点からその役割を論考する.

まず一つに, 分散合意アルゴリズムは, その動作によってレプリケーション (複製) 透過性をサポートする. この透過性がユーザから隠すのは, オブジェクトのコピーが存在するという事実である. その事実を隠すために重要な事として挙げられるのは, オリジナルのオブジェクトをいかにして正確に複製するかということである. 正確な複製の達成は,State Machine Replication で定義されたマシンとオブジェクトの状況ををいかに実システムに実現できるか, とも言い換えることが出来る. 不正確な複製オブジェクトの

存在はユーザにオブジェクトが単一のものであると認識させる事において障害になる可能性がある. 仮に, このような複製オブジェクトにアクセスできてしまうのであれば, それはユーザはより一層複製の存在を強く認識することになる.

もう一つには, 障害の透過性の実現に重要な役割を果たす. 分散システムの障害を隠すための一般的なアプローチは, どのノードでも同じ入力に対して同じ出力結果を返すように構成する事である. 各ノードが同じステートマシンとして抽象化が可能であれば, 決定論的であるという性質を有していることになるので, 一部ノードの障害を認識させない一貫した正常なシステムのような動作をユーザに提供できる.1.3 節でも言及したように, このステートマシンの正確な複製の達成を分散合意アルゴリズムはサポートする. 分散システムの故障は, 反対の性質を持つ集中システムよりも複雑である. 分散システムの多くはネットワークを介する事によって協調動作を実現している. 必然的に, マシンそのものの故障だけでなく, ネットワークの障害もシステムの直接的な故障につながる. 単純なコピーの作成だけでは, ステートマシンの複製はいずれ破綻することになる. 各ノード間で厳密な手順によって合意形成を行い, 正確な複製の実現, ひいては障害の隠蔽とフォールトトレランスを達成しなくてはならない.

1.7 分散合意アルゴリズムの必要性

前節や前項で述べた通り, 分散システムの形態をとることによって, 多大な恩恵を得る事が出来る. しかし, それと同時にシステムの内部はより複雑になる. 複数ノードによる正確な協同動作の達成は困難であり, 分散させることによる固有の問題や故障への対応が, システム開発の大きな障害となる.

分散合意アルゴリズムは, 分散システムを構成しているクラスター内でのデータや命令のレプリケーション, システム内の一貫性や整合性の問題を解決手法の一つである. 各ノード間の通信, データの受け渡し方やその手順をアルゴリズムとして方式化する. アルゴリズム化する事によって, 具体的な実装の設計図となり, その分散システムで達成すべきレプリケーション, データの一貫性と整合性の問題の解決をサポートすることが出来る. 現在に至るまでに提案されている各合意アルゴリズムの特性により, 正常な動作に必要な条件には差異が見られる. 例として最低限必要なノード数や, クラスターメンバーシップの変更を考慮するか否かの項目. 前項で示した故障モデルの内, どの故障モデルまでを想定しているか. そしてどのような環境において最高の効率で動作することが出来るのか. 開発するシステムによって最適な選択をする必要がある.

そして，このアルゴリズムはフォールトトレランスの問題を解決するためにも使用される．つまり，データの一貫性と整合性が守られるようにされていると同時に，アルゴリズムそのものが分散システムや分散処理において想定される様々な故障や障害に対して耐性を持つように設計されている必要がある.

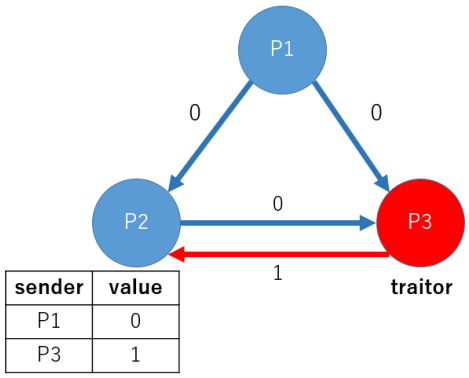

図 3 Byzantine 将軍問題時の提案状況と P2 が把握する値

1.8 Byzantine 将軍問題

Byzantine 将軍問題とは，東ローマ帝国の将軍達の間で起きた作戦遂行に関する合意の話を元にした問題である．そして，Leslie Lamport らによって分散コンピューティング上の合意問題として定義された [5].

図 3 は，Byzantine 将軍問題時の提案と P2 が把握する値を示している．まず，P1 が値 0 を提案する．P1 はその提案した値 0 を P2 と P3 にそれぞれ送信される．P2 と P3 は受け取った値を別のノードへ転送する．この時，P3 が合意を妨げる動作をする裏切り者 (traitor) だったと仮定する．その P3 によって P2 には，提案された値 0 とは違う値 1 が送られる．この場合，P2 は P1 から提案された値 0 と P3 から提案された値 1 の二つの値を持っている事になる．これにより，P2 からは 0 と 1 のどちらが正しい値かを判断することが出来ない．よって，この 3 つのノード間で 1 つの値に合意することが出来ない状況になっている．このような，一定数の裏切り者 (traitor) と呼ばれる存在がいる場合に正常な合意形成がおこなえるかを問う問題なのである.

そしてこの問題は，分散システムのクラスター内でのデータの一貫性や耐故障性をサポートする分散合意アルゴリズムでも影響を及ぼす．故障ノードは，異なる値の送信，容認されていない任意のタイミングで任意のメッセージ送信といったアルゴリズムの設計では想定されない誤作動を起こして合意形成を妨げてくる可能性がある．分散合意アルゴリズムに BFT を実装する場合，本節で述べたような作為的な障害に晒された場合でもシステムが正常に動作できるように分散合意アルゴリズムも設計されなくてはならない.

この Byzantine 将軍問題に帰着する障害は，Byzantine 故障や Byzantine 障害と呼ばれている．また，この障害に

対する耐性を Byzantine fault tolerance(以下，BFT) と呼ぶ．これは，分散システム内に一定数の Byzantine 故障ノードが発生した場合でも，システムが正常に動作する性質を指している．

Byzantine 故障への対応は, システムの開発をより困難なものにする可能性が高い. しかし,Kotla らは Zyzzyva を提案した論文で, 単純な故障ではない複雑な故障の報告件数の増加について触れている [6]. さらに同論文の中で, 今日の研究により,BFT 実装のコストが低下してきているとしており, その実装がより魅力的なものとなっていることも述べている [6].

1.9 PBFT(Practical Byzantine Fault Tolerance)

本節では, 分散合意アルゴリズムの中でも, 本研究において最も関係の深い,Practical Byzantine Fault Tolerance(以下 PBFT) について、概説する.PBFT は 1999 年に Miguel Castro と Barbara Liskov の二人によって発案された分散合意アルゴリズムである [7]. このアルゴリズムの功績の一つは, ビザンチン故障耐性をもった分散合意アルゴリズムにおいて実用的であるレベルまで至った初めてのアルゴリズムであるという点がある.

この研究は, 後の分散合意の分野に少なくない影響を及ぼしている. 事実, このアルゴリズムを元に,PBFT を修正する形で発表されたアルゴリズムが存在する [8][9]. 近年注目が高まりつつある分野の一つにブロックチェーン技術がある. ブロックチェーンはサトシ・ナカモトが発表したビットコインに関する論文にその構想が見て取れる [10]. ブロックチェーンも分散システムの形態の一つといえる. そして, ブロックチェーン技術を達成する上で欠かせないのが, コンセンサスの問題の解決手段だ．今日では, ブロックチェーンでの使用を前提とした合意アルゴリズムの研究もある [11]．その中での, 解決手法の一つに PBFT が挙げられている. ビットコインの論文の中では, コンセンサスの手法として,Proof of Work(PoW) が示されている [10]. この手法より PBFT が優れているところは, スループットが高速であることが 1 つとして挙げられている.

PBFT の合意手法は, システムに参加しているノード間で数回にわたり密に通信を行う. 送られてきたトランザクションを互いに確認しあうことによって, ビザンチン故障が想定される環境下においても, 安全に合意を実現できるようにしたものである. ノードにはクライアントとの通信やシステム内の処理の主導を担うリーダーノードとその他のレプリカノードという 2 つの役割の内の一つが与えられる.PBFT において, 正常に合意するために必要なノード数は, 故障を許容するノードを f とした場合,3f+1 となる. つまり, 最小構成は 4 ノードである. しかし, 偶数ノードによる動作は, 多数決を元にした合意形成において問題を起こす可能性がある (ある合意形成において, 同数: 同数で分かれるかもしれない). そのため, 奇数ノードでの動作が好ましい傾向がある. そのため, 実用を考えた場合は最小構成は 5 ノードが妥当だと考える.

この PBFT は現在においても, 実用性が認められているアルゴリズムである. しかし, 議論すべき点もいくつか存在する. 本研究で扱う点おいては 1.9 節において述べるが, その他に挙げられる点の一つに, 構成するノードの変更には対応していないという点がある. つまり, 動作途中のメンバーシップの変更は考慮しておらず, 完全にプライベートな状況下での動作を想定しているという事である.

ブロックチェーンの分野においては, 主要な形態として 3 種類ある．それが, パフリック型とプライベート型, そしてコンソーシアム型である．パブリック型は, ネットワーク上の不特定多数のユーザが自由に参加できる．3 つの中で一番開かれた形態である．これにより, 管理者の必要性を排し取引の透明性という利点があるがコンセンサスに時間を要するというデメリットも存在する．プライベート型は特定の管理者と定められたユーザのみが参加できる形態である．ユーザの自由な参加は出来ないが公職に処理が出来るという利点がある．しかし, 管理者の機能不全により, システム全体が正常に動作できなくなる可能性があるという弱点も存在する．3 つ目のコンソーシアム型は, パブリック型とプライベート型を混合させた形態である．プライベート型よりはある程度開かれた形態であり (パブリック型ほどの自由な参加は出来ない), パブリック型よりも高速な処理が可能だ．PBFT が威力を発揮できるのはこのうちのプライベート型とコンソーシアム型の 2 つである.

この二つの型の共通点の 1 つは, 管理者が存在するという事である．管理者の数自体には差異があるが, その役割を担うノードが必ず 1 つ以上存在する.PBFT の基本構造においても処理を主導するノードを擁立しているため, 相性が良い事が考えられる．2 つ目に処理に参加するメンバーが固定されているという点である．あらかじめ定められたノード数でシステムが稼働していくため, クラスターメンバーシップの変更に対するアルゴリズム上での対応が必要ない．元より, 定められたノード間での安全な合意の実現を主目的とした PBFT と想定される環境が酷似しており, 十分な性能での動作が期待できるのである.

では, この PBFT を一般的な分散システムにおいての合意形成やレプリケーションの点においての手法としての活用を想定する. そこで問題になるのが, 前述したクラスターメンバーシップの変更に対応できていないという点である．分散システムの利点としてその拡張性を挙げることが出来るだろう. つまり, 必要に応じて, ノード数を増やして, 分散の度合を増加させて大量の処理に対応できるようにする．また, 過剰な場合はノード数を減らすことによって, 維持に関するコストを抑えることも可能だ. マシンそのものの性能を上げるのではなく, 構成するマシンの数を増加さ

せる手法は, 分散システムが実行できる良い手法の一つに挙げることが出来る. 近年ではマシンの低コスト化により, この選択はコストの面においても優れている [6]. そして, メンバーシップの変更というのはコストという以外の点だけではく, システムの運営や保守, 故障への対応という要素とも不可分である. システムに機能変更やアップデートを施す際は, 同時にそれに最適な構成になるように現在の構成に変更を加える事になる. 障害への対応として, 故障したマシンを取り除き, 正常に動作する新しいマシンを追加する方法をとる事もある. コンピュータは物理的な機械である以上, 物理的な故障 (ハードウェアの異常) の発生は常に考えられる. これらの事実から, 分散システム内部の構成が変化しないと考える事が非現実的だという事は明らかである. 前述した問題点だけで考えても,PBFT を論文で発表されたそのままにシステムに採用するという試みは, 成功に至ることは不可能に近いと考えることは容易である.

以上で PBFT について概説した. この分散合意アルゴリズムは,Byzantine 将軍問題という合意形成問題の提起の際に定義された Byzantine 障害への耐性を持つとして 1999 年に発表されたアルゴリズムだ. そしてこれが, 実用的な段階まで至った初めてのアルゴリズムである. その合意手法は,3 フェーズに分けることができる. そして, 構成する全ノード間で密な通信を実施することによって安全な合意形成を目指す. このアルゴリズムが活用されている分野の一つがブロックチェーンの分野である. 特に, 処理に参加する存在の変更を考慮しない (受け付けない) プライベート型とコンソーシアム型での合意形成では, 魅力的な選択になる. しかし, 問題点も確かに存在する. その一つとして本節ではクラスターメンバーシップの変更に対応できないという事を挙げた. 分散システムを構成する要素の変更は分散システムの利点を享受するうえでは当然の事象である. そのメリットを打ち消してしまうこのアルゴリズムの性質は, 一般的な分散システムへの活用を考える場合は大きな問題点になりえることを論じた.

課題

Lamport らが定義した Byzantine 将軍問題は分散システムでも起こりうる問題である [5]．オリジナルの問題で定義されている裏切り者は，システム上でも故障ノードとして分散システムの信頼性やデータの一貫性，整合性を侵害する．そのために，正常の動作とは異なる様々な動作を行う可能性がある．そのような任意とも言える障害の発生が想定される環境でシステムの信頼性を守る事は難しい問題である．

PBFT(Practical Byzantine Fault Tolerance) アルゴリズムは，既存の研究は実用性に欠けると述べ，実用的な Byzantine fault tolerance アルゴリズムとして提案された [7]．PBFT では各ノード間が全ノードに対して通信を行って合意形成を行う物である．これは，実用的なレベルに達した初めてのアルゴリズムとして，今日の合意アルゴリズムの BFT 実装の基本となっている．しかし，全ノードが他ノードに対して密に，通信を行う必要があるため通信コストが高く，合意形成を行うグループの大規模化が難しい．PBFT が合意形成にかかる通信コストは N をノード数とした場合，$O(N^2)$ となる．これは，ノードが増加するごとにその中で合意を行うための通信量が大幅に増加していくことを意味している．

各章の概要

本研究では，BFT においての高通信コストによる大規模化が困難である点を課題に設定する．2 章では，分散合意アルゴリズムの関連研究を提示する．3 章では，本研究で示すアルゴリズムの具体的な説明を行う．4 章では，実装の環境や方法について記述し，それで行った実験の評価を示す．5 章では提案するアルゴリズムに対して議論されるべき点と，合意アルゴリズムにおいて考慮，解決すべき点の提示と考察を行う．6 章では本研究のサマリーを行う．

2. 関連研究

BFT プロトコルに関する研究については, 低コスト化を目的としたものがいくつか存在する.

1 つに, クォーラムベースの BFT プロトコルがある [12]. クォーラムとは, 日本語で「定足数」を意味する単語である. 分散システムの分野においては,「多数決」や「過半数」という意味合いで使用される．つまり, クォーラムベースとは, システムに参加しているノードによる多数決を行い, そのうちの過半数以上のノードに投票を土台に合意形成を行う手法である．そして, この研究で示されている手法は, 各ノードが client と直接通信を行う手法である. これにより通信の低コスト化を実現し, かつ優れたパフォーマンスを維持した状態で BFT を達成できるものであるとした. しかし, この手法は client に複製処理に関わる権限を持たせることになる．これは,client を交えてしまうがために固有の新たな問題を引き起こす可能性がある (ユーザはクラスターに侵入して改ざんする必要はなく, 自身の client 側のデータを改ざんして, 合意形成を妨害できる). 複製と合意の処理はクラスター内のノードで完結されており,client はマシンへの命令と合意後の結果の取得のみができる状態であることが好ましいと考える.

また, クォーラムベースのアプローチとレプリカベースのアプローチを組み合わせ, これら 2 つの問題を克服するハイブリッド型の BFT レプリケーションプロトコルを提案した研究がある [13]. これは, 競合が少ない状況であれば,3f+1 ノード (f は故障ノード) で高効率で動作することが出来る. しかし, 競合が多い状況では,BFT の方がスループットが優れているという結論を出した. 本研究のアルゴ

リズムは BFT の改良と位置付けることができ, 競合が多い状況においても安定した合意形成が実現できる.

別の研究では, 合意形成の通信を 2 ステップにする事により, 一般的な BFT プロトコルよりも低い通信コストを実現し処理の高速化する手法が提案されている [14]. これにより,1 回の要求ごとに全体で 3 ステップで合意を実現できるものである. この手法は高速な合意を実現しているが, 必要なノード数は 5f+1(f は故障ノード) に増加している. 本研究のアルゴリズムは, 最小 4 ノード (3f+1) でありシステム設計者にノード数の決定においてこの研究よりも多くの選択肢を提供できる.

研究 [15] は, 同じく BFT の達成を目標としたものである. この研究の興味深い所は, マスターノードの選択 (リーダー選出) において, そのノードの requtation values(評判値) という値を導入し, それを使用して投票を行うという点である. これにより,Byzantine 故障を起こしているノードがマスターノードに選出される可能性を低下させている. 6 章の議論でも詳しく触れるが, リーダーを擁立するプロトコルである場合は, リーダーが Byzantine 故障ノードである可能性が発生する. Byzantine 故障ノードがリーダーとして選出された場合, クライアントの入力を正しくシステムに反映する事は非常に困難になる.

Yin らは 2018 年に HotStuff と名付けた BFT を実現するコンセンサスアルゴリズムを提案した [16]. このアルゴリズムの特徴は,PBFT の合意形成におけるノード間の通信の複雑性を緩和したところである. PBFT の通信方式はネットワークの形態におけるメッシュ型に相当する. これにより各ノードが全ノードと通信が可能になり, それにより BFT を獲得している. HotStuff では, リーダーノードを中心としたスター型の通信方式をとる. システムを構成しているレプリカノードは, リーダーノードとのみ通信することを許可しており, レプリカノード間での通信を合意形成において行う事はない. この方式を採用する事により, ノード間の通信が大幅に簡易化されて, 合意形成のコストを削減する事に成功している. しかし, これはリーダーの正当性に大きく依存する方式でもある. また, 動作の条件としてリーダーノードが正常であるという前提がある.PBFT よりもリーダーノードがプロトコル内で強い権力を得ると同時により厳密な選出が必要になる. 研究 [15] でも取り組んでいる通り, リーダーの Byzantine 故障を想定する事は実環境において重要な事である. 本研究のアルゴリズムはリーダーベースではあるが, 合意形成においてこの研究よりも弱い権力を働かせている. これにより, リーダーの合意形成への妨害の可能性を減らしている. さらに, 最大効率で合意形成に至れた場合, この研究と同等の低コストで動作を実現できる.

3. 提案

本研究の合意アルゴリズムは, 単一の leader を擁立して行うリーダーベースの合意アルゴリズムのである. 各ノードに leader もしくは follower の役割が与えられる. 合意形成は, leader から命令の複製を受け取った follower がその命令の提案を行い, leader を含めた全ノード参加型の合意選挙によって行われる.

client からの命令は, その分散システム構成しているマシンに対する操作であり, ソフトウェアの基本機能である CRUD の内の Create(作成), Update(更新), Delete(削除) の 3 つだけを扱う. システムに対する Read(読み取り) の命令はデータの変更ではなく単純な参照であるため, 合意アルゴリズムの動作は必要ない.

3.1 役割

本研究のアルゴリズムはリーダーベースの分散合意アルゴリズムである. 従って, 合意形成の際には, 必ず各ノードに以下の役割を付与する. そして, 全てのノードは役割を兼任する事はない (leader であり follower という事は起こらない).

- **leader**
  client との通信, 各 follower への命令の複製と適用の指示を行う.
- **follower**
  leader から受け取った複製された client からの要求の正当性チェックのために選挙を発生させる. 合意後, 要求を適用する.

leader は, クラスターを形成するノードから擁立される. 合意形成時には, 必ず 1 つのノードが leader の役割を担う. leader は前述の通り, アルゴリズムの動作を主導する. client からの命令を受け取るのは leader であり, 受け取った命令の複製, 処理結果を client に返すのも leader である. leader は leader 選挙によって選出が行われる. 一度選出された leader は特定の条件を満たすまで leader であり続ける. leader は, follower に対して以下の 3 種類の通信を行う事が出来る.

- **send**
  client からの命令を, follower へ複製して送信するための通信である. これには, 命令の複製を転送を示す send の文字列と leader 自身の識別子と client からの命令内容を送信する
- **vote**
  合意形成において follower から命令の提案がされた時, その提案が自身の持つ命令と一致した際に, 提案した follower に送る通信である. これには投票することを示す vote の文字列と自身の識別子を送信する. 違う

場合は投票を拒否する (返信を送らない)

- **apply**
  合意形成の終了後に命令をマシンへ適用することを指示するために行う通信である．これには，適用を指示する apply の文字列と自身の識別子を送信する．

follower は，クラスター内の leader 以外のノードがこの役割を担う．leader を通して送られた命令について，follower からの命令の提案を元に始まる選挙形式で合意を行う．そして，合意に成功した場合には client からの要求をマシンに適用する．follower は，他ノードに対して以下の 3 種類の通信を行う事が出来る．

- **proposal**
  leader から命令が複製された際に follower が合意選挙を開始するために行う通信である．これには，提案を示す proposal の文字列と自身の識別子，そして，提案する命令内容を送信する．
- **vote**
  合意形成において他 follower から命令の提案がされた時，その提案が自身の持つ命令と一致した際に，提案した follower に送る通信である．これには投票することを示す vote の文字列と自身の識別子を送信する．違う場合は投票を拒否する (返信を送らない)
- **win**
  proposal を行った follower が，3.2 節で示す勝利条件を満たした際に leader へ送る通信である．これには，勝利したことを示す win の文字列と自身の識別子，そして選挙に勝利した際の命令を送信する．

3.2 合意選挙の勝利条件

合意選挙は，クラスター内の全ノード間で，同じ命令を保持しているかを確認する選挙である．leader から命令の複製を受け取った follower は，提案する形で自分が受け取った命令を全ノードに送る．提案が送られたノードは自身の持つ命令と照合し，同じであれば投票する．そして，本節に示す勝利条件を満たす投票が得られた場合には，その命令は大多数に正しく複製された命令という事になる．勝利条件は以下の 2 つに設定する (N はクラスター内のノード数である)．2 つの勝利条件は両方同時に達成されなければならない．そのため，どちらか片方のみの達成はそのノードが敗北，もしくは選挙そのものの失敗である．

(1) **自身を含めた過半数のノードからの投票** $(N/2+1)$

(2) **(1) の過半数の投票の中に leader からの投票がある**

(1) は，選挙という方式をとるうえでは当然の勝利条件であると考える．過半数から投票を得られない命令はクラスター内に複製された命令とは違う内容の可能性が高い．少数派の命令が選挙に勝利できるようにすると，大多数へ複製が出来た正常な命令が上書きされてしまう事態が発生する．正常な leader から正常に命令を複製されているのであれば，一度の全 follower への send で送る命令は一種類なので，送られてきた正常な命令が多数派になる．そして，自分と同じ命令の提案でないと投票しないという投票条件を設ける事で，仮に少数派 (Byzantine 故障ノード) が最初に異なる命令を提案しても票を得ることが出来ず選挙に勝利できないようにすることが出来る．

(2) は，client からの命令は leader からのみクラスター内のノードへ流れてくるという事を考慮した勝利条件である．follower が client からの命令を把握する方法は leader からその複製を送信してもらうことのみである．follower は自身が持つ命令に正当性がある場合はその命令は leader と同一である．つまり，その合意選挙において follower が leader とは異なる命令を把握していた場合，それは，アルゴリズムの正常な動作で生み出される事のない命令である可能性が高い．

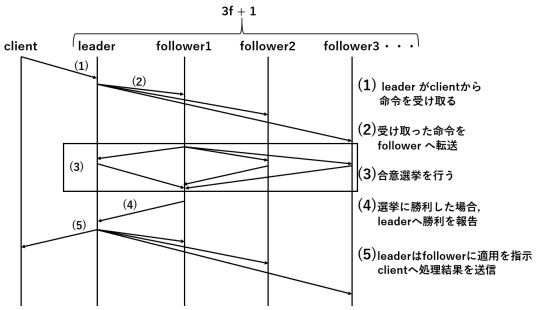

図 4 合意形成の通信フロー

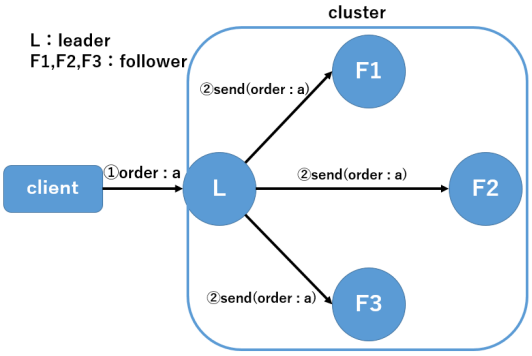

図 5 合意形成 (1)(2) の動き

3.3 合意形成

本研究のアルゴリズムは leader を含めた全ノード参加型の選挙によって合意を行う．概要として，leader が client からの命令を受け取り前述した形式での合意選挙を行う．選挙が正常に終了した場合，その命令を適用する．選挙の勝利者が現れなかった場合は選挙は失敗として，その命令を破棄し，次の命令の合意選挙を行う．本節では，この合意形成の詳細な処理の流れと制約を示す．図 4 は，順序立てを行い示した合意形成においての通信の流れである．

図 4 の (1) で，leader は client から命令を受け取る．なお，client からの命令を受け取りは leader のみの特権であ

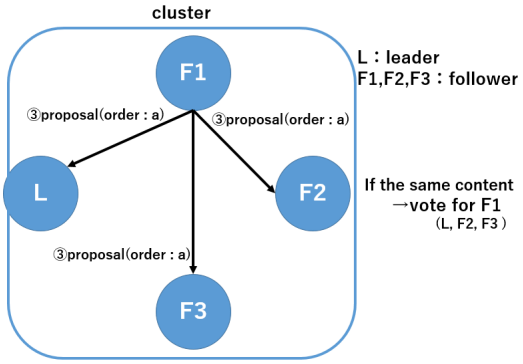

図 6　合意形成 (3) の処理

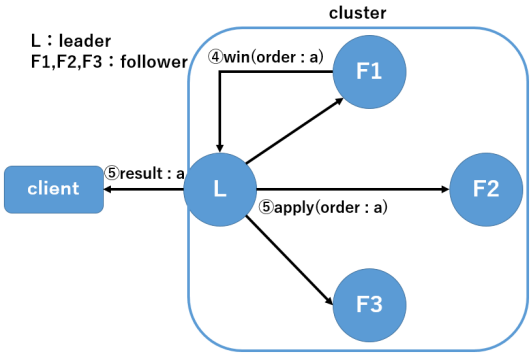

図 7　合意形成 (4)(5) の処理

る．なので，follower に client からの通信があった場合は，leader へリダイレクトするような仕組みを導入して，必ず leader を通して各 follower へ命令が送られるようにする必要がある．この段階は，図 5 中の①order:a である．図中では client からの命令である order:a を leader(L) が受け取っている．

図 4 の (2) では，leader は全ての follower へ client から送られた命令を転送する．この際，follower が leader へ通信を行えるように，leader ノードの識別子も通信に含める必要がある．3.5 章では，この場面において leader と follower 間でメッセージ異常が発生した場合を想定する．この段階は，図 5 中の②send(order:a) である．図中では leader(L) は client からの受け取った order:a をクラスター内の follower(F1，F2，F3) に send(order:a) として転送している．

図 4 の (3) では，leader から通信を受け取った命令について合意選挙を開始する．これは，命令を受け取った follower から開始される全ノード参加型の選挙である．follower は，受け取った命令内容を送り主である leader を含めたクラスター内の全ノードに自身に送られた命令の内容を提案する．そして，自分が受信した命令の正当性についての投票を促す．この際，投票を促すノードも自身に投票する．他 follower から投票を促された follower は，自身が leader から受け取った命令内容と同じ内容であれば投票する (vote)．もし，内容に相違があった場合は，投票を拒否する (proposal に対して返信しない)．各ノードは，1 回の合意選挙で投票できるノード数は 1 である．leader は提案を行った follower の命令内容と自身が client から受け取った命令内容が同一であった場合にその follower へ投票する．そうでない場合は投票を拒否する．この段階は，図 6 中の③proposal(order:a) である．図中では，follower ノードの 1 つである F1 が leader から受け取った order:a を提案して各ノードに投票を促している．他ノード (L，F2，F3) は，自身が持つ命令と同一であった場合 F1 の提案に投票する．

この提案・投票の場面においては，複数 follower が同時に提案を全ノードに送信し，分割投票が発生する場合がある．分割投票が発生した場合，正常に命令が複製されたにもかかわらず，選挙が失敗してしまう可能性がある．これを防ぐためには，Raft のランダムアプローチを適用して，各 follower が提案をするタイミングをランダムにすることにより，複数ノードによる同時提案を防ぐという手法がある [1].

図 4 の (4) では，(3) で行われた合意選挙において勝利条件を満たした follower が出現した．合意選挙に勝利した follower は図 7leader へ勝利条件を満たしたことを勝利した際の命令内容と自身の識別子を通信する．3.2 章で示した，leader からの投票を含む過半数ノードからの投票を得るという合意選挙の勝利条件により，選挙に勝利した follower とその時の leader は同一の命令を持っているようになっており，leader と違う命令を持った follower は選挙に勝利できないようになっている．もし，合意選挙において，勝利者が現れなかった場合，この選挙は失敗したと扱う．合意に失敗した命令は破棄され，次の命令の合意選挙に移行する．この段階は，図 7 中の④win(order:a) である．図中では F1 が order:a を提案し，無事選挙に勝利した．勝利した F1 は order:a で選挙に勝利したことを leader(L) に報告する．

ここで起こりうる問題として，Byzantine 故障ノードの動作により，一つは正常に勝利した follower，もう一つは勝利したことを偽装する follower の 2 つ以上の follower から勝利報告が届くことである．しかし，前述した通り，各ノードの投票権は 1 回のみであり，leader もこのルールに従う．よって，勝利の偽装は leader からの投票を得ていなければならず，leader から投票を得られるのならそれは正しく命令を複製している正常なノードである．leader からの投票を偽装する可能性も考えられるが，leader が自分が投票したノードの識別子を保持していれば勝利の偽装は避けられる．

図 4 の (5) の段階に入った時点で，命令の複製と合意選挙については正常に終了したと言える．leader は合意選挙に勝利した follower が出現したことを把握した後，再度全ての follower へ合意選挙に勝利した命令内容と適用を指示を許可するメッセージを送り，follower はそのメッセージ

を受信した際に，送られてきた命令の適応を開始する．この段階は，図 7 中の⑤apply(order:a) である．図中では，F1 が order:a で選挙に勝利したことを把握した leader(L) は，全 follower(F1,F2,F3) へ選挙に勝利した命令内容である order:a を適用するよう指示している．

ここで送られた命令内容が，これがこの合意選挙の最終結果となる．ここで共に送られてきた命令を follower はマシンへ適応できる．これにより，仮に合意選挙中に違った命令が保持されていた場合でも正常な follower はここで送られてきた命令を適用する事により．他ノードと同じマシンの適用状態になる．

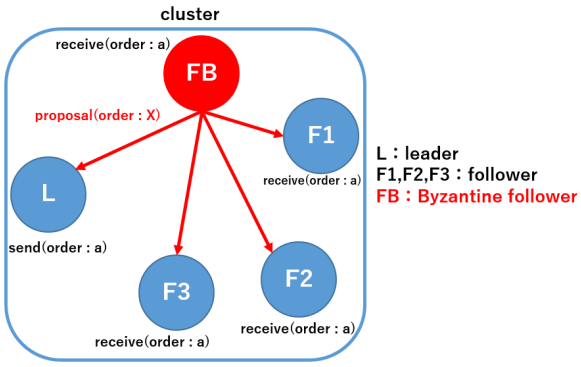

図 8 follower に Byzantine 故障ノードが出現した場合

3.4 全体ノード数と故障ノードの許容数

本研究のアルゴリズムは最小 4 ノードで動作することを想定している．クラスター内のノード数 (N) の決定は，どのくらいの個数の故障ノード (f) を許容するかを考慮して決定されるべきである．研究 [5] は，正常なプロセス 2f+1 個存在しなければならず，さらに，Byzantine 故障以外の不作為な障害を起こすノードの発生も想定したノード数にしなければならない事を示した．これらの事から，クラスターのノード数を BFT を考慮して決定する場合は数式 1 の計算式になる事が示されている [17][18].

$$N = 3f + 1 \tag{1}$$

以上の事から，BFT を考慮して決定するノード数の最小値は 4 であり，最低限 4 ノード以上でクラスターを構成しないと BFT を達成できないという事になる．本研究のアルゴリズムもこの研究に準拠し，最小 4 ノードで動作するとした．

3.5 Byzantine 故障の発生

本節では，実際に 3 分の 1 以下のノードが Byzantine 故障ノードであった場合に本研究のアルゴリズムがどのような状態になるかを示す．

最初に，図 8 のような状況を考える．これは，follower の一つが Byzantine 故障ノードとして出現している状況である．この Byzantine 故障ノードは，合意を妨げるために，正しい leader から送られた命令とは全く違う命令を提案して投票するように他 follower に要求するような動作をすることもある．そして，提案タイミングにランダム化アプローチを採用していた場合は，Byzantine 故障ノードの提案がこの合意選挙において一番最初の提案になることもある．しかし，3.3 節で述べた投票条件により，これは誰からも投票される事無く，この提案は失敗に終わる．そしてこれ以降の合意選挙においても提案に失敗し続けるため，結果的に全体の合意形成に影響を及ぼすことはない．

Byzantine 故障ノードが発生する状況での合意形成において，一番発生してはいけないのは，Byzantine 故障ノードが leader になってしまうことである．leader に Byzantine 故障ノードが選出された場合，その状況で正しい合意形成を行う事は非常に困難である．client と直接通信する権利や follower に命令を複製する権利は leader のみが持つ．そのため，Byzantine 故障ノードに強権を与える事になってしまうからである．leader となった Byzantine 故障ノードの動作や命令は，follower には不正なものとして捉える事は難しい．follower の基本動作は leader からの命令に従うのみだからである．

4. 実装

実装を図 9 に図示する．本研究のアルゴリズムを Consensus Module というソフトウェアとして実装する．Consensus Module は図 9 に示している 3 つのコンポーネントで構成されている．

controller は，各 ConsensusModule を制御する役割を担っている．各 Module とシステムを利用する client は controller を通して通信を行う．そして，controller が自身の現在の状態 (leader か follower) や識別子 (ID) を管理する．さらに，現在の leader を把握するために leader の識別子を保有する．これは，leader としてハートビートを送ってきたノードの識別子である．そして，controller は自身の役割と他ノードからの通信に含まれる通信の目的を示す文字列を参照して，follower function もしくは leader function を呼び出して処理を行う．そしてその結果を他ノード，もしくは client へ送信する．

leader function は，自身の役割が leader の際に controller から呼び出されるコンポーネントである．leader function には，以下の leader が持つべき機能と行うべき動作が定義されている．

- ハートビート
- client との通信機能
- follower への通信機能 (send, vote, apply)

ハートビートは，leader が全 follower に定期的に送る通信である．これは，leader が正常に稼働している事を follower に伝えるための通信であり，follower はこれにつ

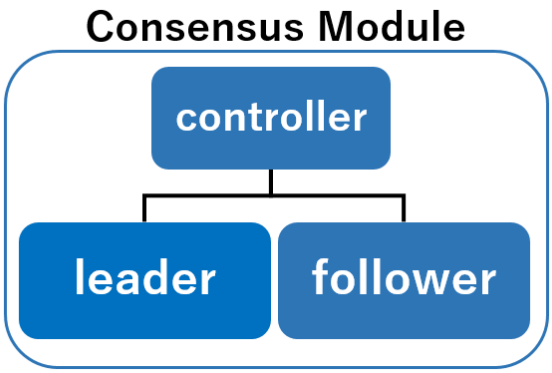

図 9　実装構成図

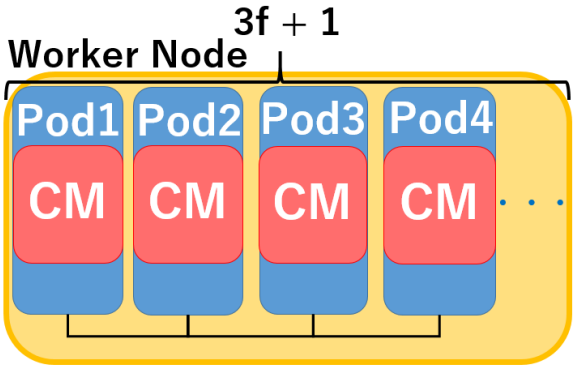

図 10　実験環境構成図

いては合意を行わず，返信も送らない．このハートビートが定期的に送られることにより，follower は現在の leader が少なくとも稼働しているのかを知ることが出来る．このハートビートが送られないという事は，現在の leader が故障して動作が出来なくなっている可能性が高い．client との通信機能は，client からの要求の受信と命令適用後の結果を client に返す機能である．client との通信は leader のみが行えるものであり，この機能は leader function のみがもつ．follower との通信機能は 3.1 節で leader が follower に対して実行できる 3 種類 (send, vote, apply) の通信である．

follower function は，自身の役割が follower の際に呼び出されるコンポーネントである．follower function には，3.1 節で示した合意選挙時に他ノードに行う通信 (proposal, vote，win) が定義されている．

5.　実験と分析

5.1　実験環境

実験は，図 10 で示した環境で行う．Kubernetes 環境を構築し，Master Node と Worker Node をそれぞれ 1 ノード用意する．それぞれのノードは VM であり，表 1 に示すスペックで作成されている．各 Pod は単一コンテナで構成されており，1Pod が 1 ノードとして動作する．Pod に

表 1　VM のスペック

	内容
OS	Ubuntu18.04
CPU	2 コア
メモリ	2048MB
ストレージ	20GB

は Consensus Module(図 10 中では CM と表記) が 1 つずつ配置されている．そして，Pod 間で通信を行って合意形成を行う．実験では，開始時点で 1 つのノードを leader として擁立させる．leader ノードは client から命令を貰う代わりに，そのノード内にクラスター内に適応する命令が記述されているテキストファイル (Order.txt) を保有している．Order.txt から 1 行ずつ命令を読み出して follower へ命令を転送する．実験では，以下の 3 種類の命令を扱う．

- **create: filename**
  filename で指定された名前でテキストファイルを生成する．
- **append: [filename, value]**
  filename のテキストファイルに value を書き込む．
- **delete: filename**
  filename のテキストファイルを削除する．

follower は，3.3 節で示した合意選挙を発生させる．そして，合意選挙にて勝利者が出現した際には勝利した命令を自身のノードで実行する．

follower の中に Byzantine 故障ノードを意図的に出現させる．個数は，3.4 節で示したノード数を参考に決定する．Byzantine 故障ノードとなった follower は leader が持つ Order.txt とは別に命令が記述されたテキストファイルを持つ (ByzantineOrder.txt). 書かれている命令の種類は Order.txt と同じ 3 種類だが，内容が異なるものである．Byzantine 故障ノードは leader から send で命令が送られた際，その命令ではなく ByzantineOrder.txt で記述された命令をクラスター内に提案するように動作させる．

5.2　評価方法

評価では，Byzantine 故障ノードの出現した状況でも大多数に命令の複製と適用が出来るかという BFT の点と，合意形成を行う際の通信コスト (通信回数) の点で評価を行う．

まず、BFT の点を評価するために，各 follower に自分が適用した命令を記録させるようにする．これは，Order.txt と同じ形式になるように記録する．その各 follower が持つ記録ファイルと leader が持つ Order.txt とを比較し，どの程度差分が発生するかを評価する．差分がなければ，その follower は leader からの命令を全て複製、適用できたことになる．

通信コストの点は，1 命令において通信を行った回数をカウントするように各ノードの機能に一部変更を加える．

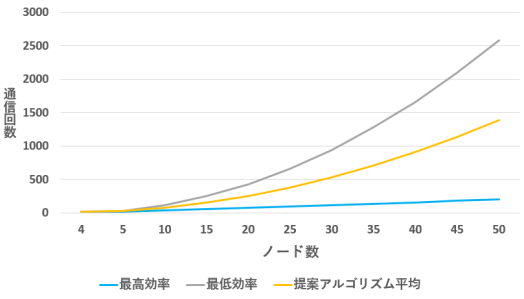

図 11 本提案のアルゴリズムの通信回数の推移

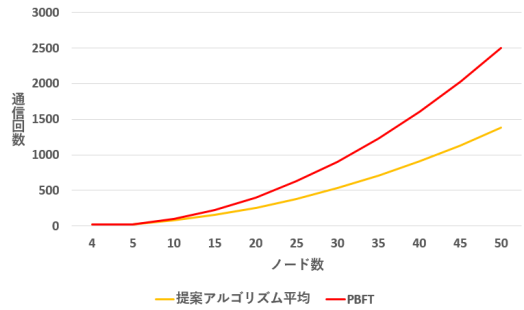

図 12 PBFT と提案アルゴリズム平均の通信回数比較

そして leader の send から最後の leader の apply までを 1 セットとして各セットの通信回数を集計し，その平均を算出する．算出された値が 1.9 節で示した BFT の概ねの通信コストである $O(N^2)$ とどの程度の差が生じるかで評価する．

5.3 評価結果

本節では, 本提案のアルゴリズムと PBFT アルゴリズムの比較を行う. なお, 本稿で示す結果は実測値ではなく, 手計算で求めたものであることに留意すべきである. まず, 本提案のアルゴリズムの通信回数を図 11 に示す.

図 11 は, 本提案アルゴリズムの合意形成に要する通信コストの最大値と最小値, そこから算出した平均を示している. 最高効率とは最小値を表しており, 合意選挙が初回の 1 回で成功し, 合意に至れた際の通信回数である. また, 最低効率は最大値を示している. これは, 合意選挙が最大回数行われた後に合意に至れた際の通信回数である. 平均は, 最大効率と最低効率の 2 値の平均である.

ノード数が 1 桁台から 10 個前後である場合, 通信回数にそれほどの違いが見られない. しかし, ノード数の増加とともにそれぞれの通信回数に大きく差異が現れている. 最高効率の場合, ノードの増加に伴う通信回数の推移は非常に緩やかで線形的になっている. 最小 4 ノード時に 14 回だった通信回数は,50 台の時には 198 回に変化した. 増加の幅は約 1,414 %であった. 反対に最低効率の場合, ノード数 10 以降から大幅な通信回数の増大が見られる. 最小 4 ノード時には 21 回ではあるが,50 台になった時点で 2,574 回と大幅な増加が見られた. 増加の幅は 12,257 %であり, パーセンテージを比較すると最低効率は最高効率の約 9 倍の増加幅であった. 最大回数行う合意選挙は,PBFT のメッシュ状の通信に近い状態になり, 本提案のアルゴリズムが最低効率で動作した場合,PBFT と同等の効率に近い. しかし, これは 1 ラウンドごとの合意選挙の回数に上限を設けることにより, 回数の増大を制限することが出来る.

次に, 図 12 に PBFT の通信コストと本提案アルゴリズムの平均通信コストの比較を示す. 図 11 同様, 黄色のグラフが提案のアルゴリズムの平均を表しており, 赤のグラフが PBFT の通信コストを示している.

2 つのアルゴリズムもノード数 1 桁台から 10 台前後の範囲では, ほとんど差異は見られない. しかし, 最小 4 ノードの時,PBFT が 16 回に対して, 本提案アルゴリズムは 18 回であり, 最小構成である場合は PBFT の方がより低コストになる事が期待される. しかし,5 ノード時で本提案アルゴリズムと PBFT の通信回数は一致し, それ以上のノード数では本提案アルゴリズムの平均値の方が低コストになっている. その差は, ノードを増加するほど広がっていく.10 台の時点で PBFT の約 76 %のコストで合意形成を行える. そしてノード数が 50 台になると PBFT の約 55 %のコストで合意形成を行えることが期待できる. 結果として,PBFT のおよそ 76 %以下のコストで本提案アルゴリズムは合意形成を行える事がわかった.

6. 議論

リーダーベースのアルゴリズムにおいては，リーダー選挙の仕組みの導入は不可欠である，これは，本研究のアルゴリズムにおいても議論されるべき点である．もし BFT を考慮しないのであれば，Raft のように，選挙タイムアウトを設けて，立候補のタイミングがプロセス間で衝突することを防ぎ，リーダー選挙を leader になれるノードの条件付けを行った多数決で決定すれば良い [1]．BFT を考慮する場合，Byzantine 故障ノードが leader になろうと動作することも考えられる．その Byzantien 故障ノードは，現 leader が正常に動作している事も関係なく，自身の情報を更新して leader 選挙を開始する可能性がある．考えられる対策として，各ノード共通の選挙許可タイムアウトを導入し現 leader が動作中はこのタイムアウトが作動しないようにする (leader のハートビートが届くたびにタイムアウトをリセットする)．そして，この選挙許可タイムアウトがタイムアウトしない限り投票を拒否する．これにより，Byzantine 故障ノードが一人でに選挙を始めても，すぐにそのノードが leader になる事は無くなる．

しかし，Byzantine 故障ノードが leader にならないように完全に対策を施すことは困難である (leader 選出時は正常でも動作中に Byzantine 故障ノードと化す可能性もある)．なので，Byzantine 故障ノードが leader になってしまう可能性を予め想定し，leader に Byzantine 故障の疑いが

ある際には follower の側から新しい leader を選出するように動作を追加する方が現実的であると考える．例として，リーダーがフォロワーに送るハートビートに何かしらの情報を入れ，不整合であった場合に，すぐリーダー選挙を始める．合意選挙が失敗したり，制限時間を設けてそれ以内に選挙が終わらなかった場合には異常正常関わらず新しくリーダー選挙を始めるといった対策が挙げられる．また，研究 [19] では，合意形成の貢献に応じて変動するスコアを導入して，その数値に応じてノードの除外をするといった手法を提案している．leader をどのように制御するかはリーダーベースの合意アルゴリズムを設計する上での問題の 1 つである．いくつかの関連研究では，リーダーを擁立せずに Byzantine 状況下でも合意出来る可能性についての研究や leader を擁立しない合意アルゴリズムが提案されている [17][20][21].

また, 考えうる対策の一つに, 新しい役割を追加する事も考えられる. つまり, その権利のおいては leader 以下であり follower 以上である中間に位置するノードを登場させることである. その中間ノードは,leader と同時にクライアントからの要求を直接受信できる権限を与えるが, クライアントからの通信に対しては受信のみに制約を加える. そして,Byzantine 故障を起こした leader が, クライアントからの要求以外の内容について合意選挙を開始しようとした際に,commit する前のいずれかのタイミングで選挙を中止させるように動作させる. これにより,Byzantine 故障を起こした leader による想定外の合意選挙を防ぐことが出来る.

リーダー (もしくはプライマリー) を擁立し, 処理を主導させる手法は, 合意やレプリケーションの問題の多くを解決に導くことが出来る. しかしこれは, 強みと同時に弱点を生み出すことになる. 前述したとおり,Byzantine 故障を想定した場合, そのリーダーノードが単一障害点として分散システムの大きな弱点になる. リーダーの裏切りにはどのレプリカノードも気づくことは非常に困難である.

この問題は, リーダーノードのみがクライアントと直接通信を行うことが可能であり, リーダーノードのみが真のクライアントの要求を正確に把握できる構造であることが原因の 1 つであると考えれる. リーダーノード以外のレプリカノードは, クライアントとの直接通信は基本的に許可されておらず, いずれも, 他ノードをはさむことになる. いわば又聞きの状態である. その制約の中で, リーダーがクライアントの要求を正しくレプリカノードに複製しているか確認する手段を持たない. 正確には複製されて送られてきた要求が正しいかどうか確認するための比較対象をレプリカノードは持ちえないのである. その状況を改善するために, 前述した中間ノードを擁立する. これにより, クライアントから直接要求を受け取れるノードが 2 つ存在することになる. そして, その中間ノードが監査の役割を担うことにより, 健全な合意へ至れるようサポートすれば良いという考えである.

3.3 節で示した合意選挙は，勝利条件を満たした follower が出現するまで続けられることになる．選挙回数に制限を加えない場合最悪のケースとして follower の数だけ提案と全ノードへの通信が行われることになる．正しく命令が follower へ転送できたにも関わらず，合意できないケースも考える (ネットワークの遅延や単純な故障)．しかし，正しく転送出来ているのであれば，全 follower が提案するよりも早く合意選挙の勝利条件を満たした follower が出現すると考える方が現実的であり，合意できないほどの故障，または故障ノード数に侵されている場合は，どんな，合意アルゴリズムでさえ正常に動作することは不可能である．最後に提案した follower が勝利条件を満たす可能性もあるが，そこまで合意できなかった命令の正当性があるかは疑問である．よって，一回の合意選挙において実行できる提案回数に制限も設け，システムのボトルネックになる可能性を防ぐ必要がある．

7. おわりに

今日のシステムの多くは, 分散システムと呼ばれるシステム形態によって構築されている. 分散システムとは, 単一で一貫性のあるシステムとして動作をする独立したコンピュータの集合であると定義されている. 分散システムを構成している各マシンが単一のコンピュータであり, それは高性能なマシンや, 最小構成のコンピュータ, もしくはセンサーネットワークと用途に応じてその姿は変わる. その分散システムの合意において重要な役割を果たすのが分散合意アルゴリズムである.

分散合意アルゴリズムの分野においていくつかの故障モデルが定義されている. その中で最も複雑なものが Byzantine 故障モデルである. そして,Byzantine 故障モデルへの耐性 (Byzantine Fault Tolerance) の実装は, 近年ではより一層達成されるべき特性となっている. その BFT 達成の手法の中で PBFT は実用性に耐えるレベルに至った初めての分散合意アルゴリズムである. そのアルゴリズムは BFT 実装の土台として今日においても用いられている.

しかし, 各ノード間で密に複数回通信を行うことにより, その通信コスト, 計算量はノードの増加により指数関数的に増加する. そのため, より大きな分散システムへの拡張が困難という課題がある. そこで, 本研究では BFT が実装された比較的簡略化されたリーダーベースの分散合意アルゴリズムを提案した．本研究のアルゴリズムは，値や命令, トランザクションの合意の際に，leader を含めた全ノード参加型の合意選挙の手法をとっている．選挙の手法を採用することによって,PBFT での合意形成に比べ, 密に通信しあう必要をなくすことが狙いである. さらに, 合意選挙の勝利条件を厳密にすることにより, 存在しないトランザクションが合意されることを防ぐ. そして，本研究で提案したア

ルゴリズムを Kubernetes 環境で実装と実験を行う事を示した.

本項では, 実測値ではなく計算によって算出した通信回数を比較基準として,PBFT と本提案のアルゴリズムの合意形成に要する通信コストを比較した. まず最初に, 本提案アルゴリズムの通信回数の振れ幅を, 最高効率と最低効率の 2 値にして示した. この振れ幅が生まれる原因は,1 ラウンドにおいて合意選挙が複数回行われる可能性がある事によるものである. そしてその値から平均を算出して図 11 に示した. 計算結果として, ノード数が一桁台から約 10 個以下の場合, 大きな差は見られなかった. しかし, ノードの増加とともにその差は明確になっていった. 最終的な増加の幅の差は, ノード数 50 の時点で最低効率は, 最高効率の約 9 倍の増加幅であることが計算の結果判明した. そして, 計算によって判明した平均値と PBFT の通信コストの 2 値を比較した. 比較によって, 最小ノード (4 ノード) 時は, 本提案アルゴリズムの平均値よりも,PBFT の方が僅差で低コストであることが分かった。しかし,5 ノード時点でその通信回数は同等になり, それよりも多いノード上では, 本提案アルゴリズムの平均値が PBFT のコストを下回ることが分かった. そのコストの差は, ノード数の増加とともに, 大きくなっていった. 動作するノード数によって異なるが, 本提案アルゴリズムの平均値が PBFT よりも低コストになる状況に限って言及すれば,PBFT の約 76 %以下の通信回数で合意形成が出来ることが分かった. 本稿で示した評価結果の最大ノード数は 50 ノードであるが, その時点で本提案アルゴリズムの平均値は PBFT の約 55 %ほどの通信回数で合意に至れる.

また議論においては, 提案したアルゴリズムを実用するために必要な諸項目について提起し, その解決手法を簡略ではあるが述べた. すなわち, リーダー選挙,leader の Byzantine 故障への対応, 合意選挙のより厳密な終了条件についてである. これら提起した項目を満たすことが出来れば, より実用的な分散合意アルゴリズムとして, 分散システムのレプリケーション, 一貫性と整合性の問題の解決手法において有力なものになりえる.

参考文献

[1] Ongaro, D. and Ousterhout, J.: In Search of an Understandable Consensus Algorithm, *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, Berkeley, CA, USA, USENIX Association, pp. 305–320 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2643634.2643666⟩ (2014).

[2] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Commun. ACM*, Vol. 21, No. 7, pp. 558–565 (online), DOI: 10.1145/359545.359563 (1978).

[3] Schneider, F. B.: Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial, *ACM Comput. Surv.*, Vol. 22, No. 4, pp. 299–319 (online), DOI: 10.1145/98163.98167 (1990).

[4] Keeney, M., Kowalski, E., Cappelli, D., Moore, A., Shimeall, T. and Rogers, S.: Insider threat study: Computer system sabotage in critical infrastructure sectors, Technical report, National Threat Assessment Ctr Washington Dc (2005).

[5] Lamport, L., Shostak, R. and Pease, M.: *The Byzantine Generals Problem*, pp. 203–226 (online), available from ⟨https://doi.org/10.1145/3335772.3335936⟩, Association for Computing Machinery (2019).

[6] Kotla, R., Alvisi, L., Dahlin, M., Clement, A. and Wong, E.: Zyzzyva: speculative byzantine fault tolerance, *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pp. 45–58 (2007).

[7] Castro, M. and Loskov, B.: Practical Byzantine Fault Tolerance (1999).

[8] Moniz, H.: The Istanbul BFT Consensus Algorithm (2020).

[9] Saltini, R. and Hyland-Wood, D.: IBFT 2.0: A Safe and Live Variation of the IBFT Blockchain Consensus Protocol for Eventually Synchronous Networks (2019).

[10] Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System, (online), available from ⟨http://www.bitcoin.org/bitcoin.pdf⟩ (2009).

[11] Buchman, E.: Tendermint: Byzantine fault tolerance in the age of blockchains, PhD Thesis, University of Guelph (2016).

[12] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K. and Wylie, J. J.: Fault-Scalable Byzantine Fault-Tolerant Services, *SIGOPS Oper. Syst. Rev.*, Vol. 39, No. 5, pp. 59–74 (online), DOI: 10.1145/1095809.1095817 (2005).

[13] Cowling, J., Myers, D., Liskov, B., Rodrigues, R. and Shrira, L.: HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance, pp. 177–190 (2006).

[14] Martin, J. . and Alvisi, L.: Fast Byzantine Consensus, *IEEE Transactions on Dependable and Secure Computing*, Vol. 3, No. 3, pp. 202–215 (2006).

[15] Chen, J., Zhang, X. and Shangguan, P.: Improved PBFT Algorithm Based on Reputation and Voting Mechanism, *Journal of Physics: Conference Series*, Vol. 1486, No. 3, p. 032023 (online), DOI: 10.1088/1742-6596/1486/3/032023 (2020).

[16] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G. and Abraham, I.: HotStuff: BFT Consensus in the Lens of Blockchain (2018).

[17] Lim, J., Suh, T., Gil, J. and Yu, H.: Scalable and leaderless Byzantine consensus in cloud computing environments, *Information Systems Frontiers*, Vol. 16, No. 1, pp. 19–34 (2014).

[18] Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H. and Toueg, S.: Consensus with Byzantine Failures and Little System Synchrony, *International Conference on Dependable Systems and Networks (DSN'06)*, pp. 147–155 (2006).

[19] Jiang, Y. and Lian, Z.: High Performance and Scalable Byzantine Fault Tolerance, *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, IEEE, pp. 1195–1202 (2019).

[20] Borran, F. and Schiper, A.: A leader-free byzantine consensus algorithm, *International Conference on Distributed Computing and Networking*, Springer, pp. 67–

78 (2010).
[21] Crain, T., Gramoli, V., Larrea, M. and Raynal, M.: Dbft: Efficient leaderless byzantine consensus and its application to blockchains, *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, IEEE, pp. 1–8 (2018).

