分散システムの障害の透過性実現のためのプロセスの状態判別方法の提案

岡田 大輝[1,a)]　串田 高幸[1]

概要：今日の情報化社会において，分散システムは重要なシステム形態である。分散システムの分野においての,「透過性」という概念に焦点を当てる。特に透過性の一つである「障害の透過性」の実現においては死んだプロセスと反応が遅いプロセスの判別方法が難しいとされている。その解決方法として，参加ノードを介してプロセス状態の確認を行う。そして，得られた情報を元に多数決の形を取りリーダーとなるノードで最終的な状態の決定を下して判別する方法を提案する。本稿では提案する方法の設計と同時にその実験方法を提案を示した。そして，現在の提案を元に改善できる点を提示する。

1. はじめに

分散システムとは，それぞれが独立して動作できる自律コンピューティングが集まり処理を共同することによって、ユーザーからは単一のシステムのように見えるシステムの形態である。

この分散システムおよび分散コンピューティング分野の概念の一つに透過性という概念がある。これは使用するユーザーから分散システムのあらゆる側面を隠し、一つのシステムとして見せることである。そしてこの透過性は概念であると同時に、分散システムが満たすべき設計目標でもある。代表的にはアクセス、ロケーション、リロケーション (移転)、ミグレーション (移行)、レプリケーション (複製)、並行性、障害・失敗の 7 つがある [1]。

- **アクセス**
  データ表現とオブジェクトへのアクセス方法の違いを隠す。
- **ロケーション**
  オブジェクトが存在する場所を隠す
- **リロケーション (移転)**
  使用中にオブジェクトを別の場所に移動できることを隠す
- **ミグレーション**
  オブジェクトが別の場所に移動する可能性があることを隠す
- **レプリケーション (複製)**
  オブジェクトが複製されることを隠す
- **並行性**
  オブジェクトが複数の独立したユーザーによって共有される可能性があることを隠す
- **障害**
  オブジェクトの障害と回復を隠す

これらの透過性の中で、障害の透過性は最も実現が困難な概念である。これは、デッドプロセス (死んだプロセス) と反応が遅いプロセスの区別をつけることが困難なことが理由の一つに挙げられる。

この二つのプロセスの状態によって起こされる故障および障害は分散システムにおける故障モデルの内、Crash-Recovery 故障モデルに含まれる。この故障モデルは、完全な故障の他に途中から復活してくることも想定したものであり、本当に故障したのかただ反応が遅いだけなのか最後まで分からないモデルである。今日、分散合意アルゴリズムとして広く使用されている Raft もこの故障モデルまでの耐故障性を持っている [2]。このことから、Crash-Recovery 故障モデルを想定することは実用性を考慮するうえで重要であり、上記の二つのプロセスの判別方法の探求は有意味である。

本稿では、死んだプロセスと反応が遅いプロセスの判別方法を提案する。2 章で分散システムの障害管理や透過性の関連研究を取り上げる。3 章では本稿の提案を説明する。4 章では本提案の評価を行う。 5 章では本提案の考えられる改善点を挙げる。

[1] 東京工科大学コンピュータサイエンス学部
CDSL, TUT, Hachiouji, Tokyo 101–0062, Japan
[a)] C0117071

2. 関連研究

障害の透過性のは分散システムの障害とその回復をユーザーから隠すものだ。故障、障害を扱う概念である以上、障害管理にも深く関係している。これら透過性や分散システムの障害管理については、以前から研究が成されている。

障害を検知する方法として、本提案ではプロセスに着目している。その他の方法として、各プロセス間の通信にタイムアウトを設けるものがある。タイムアウトで指定した時間内に応答が無かった場合故障が発生しているとするものである。関連する研究では、タイムアウトを使用せずに物理クロックを用いる事でフォールトトレランスを実現するアプローチがある [3]。また、これらのフォールトトレランス用の複数のアルゴリズムをサポートする Web サービスのフォールトトレランス用のアーキテクチャの設計及び実装を示した研究がある [4]。しかし，これはあくまで障害の検知であり，通信しているプロセスが死んでいるがために起こった障害なのか，応答が遅れているがために起こっている障害なのかまで判別することが出来ない。

障害からの透過的な回復に関する研究では，ランタイム回復ポリシーを統合するための建設的なアプローチを提案された [5]。これは，障害からの回復を図る際に，他のプロセッサの動作を中断させない点で透過的な障害回復を提供する。また，分散システムを構成する環境下で起きたプロセッサの障害後にシステム全体の一貫した状態を回復するためのプロトコルとして Optimistic Recovery という手法が提案されている [6]。この提案は分散システムの障害発生は稀であるという楽観的な視点を前提としたものである。本提案ではプロセスに焦点を当て，分散システム中のプロセスの障害発生は大小問わず常に起こるものという前提を置く。その前提の元，プロセス監視の一点に注力した状態判別方法を提案する。

CORBA システムは，アクセスとロケーションの透過性の 2 つをサポートしている。Skrzypczynski は，CORBA システムで障害の透過性をサポートするために動的プロキシパターンに基づいたソリューションを示した [7]。このソリューションはサーバー障害，プロセス障害，ネットワーク障害といった障害の解決を提供することが出来る．しかし，これは CORBA オブジェクトに対してのみ障害透過性を提供する．本提案では，普遍的なプロセスの監視に着目するという点で，あらゆる規格に左右されないものである．

Lowell，Chandra，Chen の 3 人は，オペレーティングシステムの障害の透過性の抽象化を検討した [8]。障害の透過性を提供するための条件として，十分なアプリケーション状態が保存されることを保証することと，アプリケーション状態に影響する障害からの回復を可能にするために十分なアプリケーション状態が失われることを保証することの 2 つを挙げた。本提案では，分散システム中のアプリケーションのプロセスが完全な故障が発生しない場合 (プロセスは正常に動作しているが応答しない状態) でもプロセスの遅延を検知し，透過性の実現をサポートする。

Ravindran，Chanson の 2 人は，障害回復中に RPC(リモートプロシージャコール) レイヤーによって悪用される可能性があるアプリケーションレイヤーの特定の汎用プロパティを反映するモデルを提示した。そして，モデルに基づいた障害に起因する孤児プロセスを発見する手法を提案した [9]。この手法は，孤児プロセスを削除する手法に必要なロールバックを最小限に抑えるものである。

表 1　応答とプロセス状態の整理

通信・呼び出し	プロセス状態
応答あり	生きている
応答なし	死んでいる
	生きている

3. 提案

3.1 前提

まず、外部からの通信や呼び出しに対しての反応、及びプロセスの状態を表 1 のように整理する。外部からの通信や呼び出しは、そのプロセスに対しての通信、呼び出しである。　外部からの通信や呼び出しに対して応答があり、プロセスが生きている場合、これはそのノードは稼働しており、渡された処理に対して正常に結果を返すことが出来る状態にある。

外部からの通信や呼び出しに対して応答がなく、プロセスの状態が死んでいる場合、これはそのノードは参加している分散システム内の故障箇所であり渡された処理に対して結果を返すことが出来ない状態である。この場合はプロセスは死んでいると判断する。

外部からの通信や呼び出しに対して応答がなく、プロセスが生きている場合、このノード内のプロセスは動いているが結果を返すことが出来ない状態にあると仮定する。ネットワークの遅延やプログラムに欠陥があるといった原因が考えられる。この場合、プロセスは処理が遅延しており反応が遅れていると判断する。

プロセスの状態確認の時、少なくとも過半数のノードが正常な状態であることを前提とする。これは、分散システムにおいて、N 個の故障ノードを許容するにはどのくらいのノード数が必要かという指標である 2N+1 の式に従っている。この式は, 健康なノードが過半数以上存在しなければ分散システム自体が正常に機能しないことを示している.

3.2 判別方法

本稿で提案するプロセス状態の判別方法は、プロセスの異常が考えられるノードに対して、正常に動いている各

ノードがそのノードに対して応答を要求する。各ノードが得られた結果を最終決定を下すを役割を担うサーバーで処理し、プロセス状態の判別を行う。これは、独裁的なサーバーによって判断を下さず、多数のノードからの応答を元に一つの状態に決定するという点で代表的な分散合意アルゴリズムとの共通点がある [2][10]。

3.2.1 役割

本提案では、プロセス状態を判別するために以下に示す 2 つの役割を設ける

- **Leader**
  各ノードから送られた応答を元に最終的な状態判別を担当する。
- **Reporter**
  Leader へ監視対象のプロセスの状態を報告する。

Leader の役割は、Reporter への命令とコントロール、送られた応答を元に状態決定を下すことである。この Leader のみ処理を主導する権利を持っており、Reporter への命令は Leader 以外のノードからは送られることはない。Leader は Reporter よりも強い権限を持っているが、Leader のみで状態判別が行われることはない。そして、Reader が直接故障が疑われる対象ノードの Reporter へ応答を要求することはない。

Leader の内部構成は図 1 中の Leader 部分のようになる。Leader の動作は Control から Connecter、Decision に送られることによって行われる。Connecter は他ノードに配置されている Reporter とのプロセス状態の交換を担当する。Reporter から送られる応答を元に Decision にて状態の判別を行う。

この Leader は、分散システムを構成しているノードとは別のサーバーに配置される。本提案のソフトウェア構成図では、別途の監視サーバーの一部として Leader が動作することを想定している。

Reporter は対象プロセスの監視を行う。そして Leader からの命令があった際には対象ノードの Reporter へ応答を要求し、得られたプロセス状態を送信する役割を担っている。Reporter は Leader と異なり、分散システムを構成している各ノードに常駐する。Reporter は以下の 3 パターンの応答を返すことが出来る。

- **Alive**
  プロセスは生きている。
- **Dead**
  プロセスは死んでいる。
- **NoReply**
  応答が返ってこなかった。

Alive は、担当するノードのプロセスが生きている時に外部からのプロセス状態の提供要求に対して返す応答である。この場合、プロセスは稼働自体はしている。しかし、表 1 に示した通り正常に処理を返しているかはこれだけでは判別することは困難である。

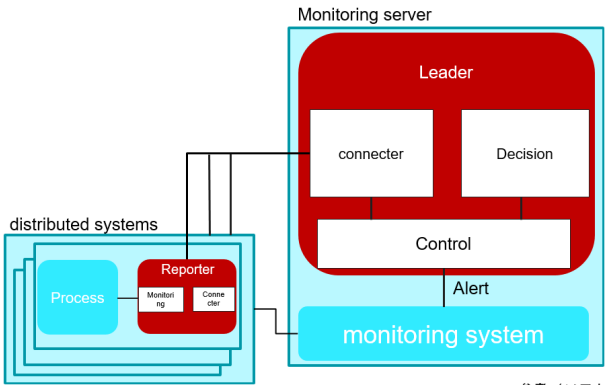

図 1　ソフトウェア構成図

Dead は、プロセスが死んでいる場合に外部からの要求に対して返す応答である。この場合は、プロセスは死んでおり、稼働していない。しかし、状態を確認した段階では死んでいながらもすぐに復活する可能性がある。

NoReply は、対象ノードの Reporter から状態を調べるために応答を要求した Reporter へ応答が無かった場合や Leader へ調査を依頼した Reporter から一定時間内に応答が無かった場合に応答なしとして処理する。

Reporter の内部構成は図 1 中の Reporter 部分のようになる。左白枠の connecter は Leader の connecter とプロセス状態の交換を行う部分である。ここで、Leader からの命令を受け取る。右白枠 Monitoring は実際に対象プロセスの状態を監視する部分である。この connecter と Monitoring が Reporter の中で処理を分業する。

3.2.2 動作

図 1 は本提案のソフトウェア構成図である。図中の monitoring system(以下、モニタリングシステム) はメトリックス監視ソフトウェアを指している。distributed systems(以下、分散システム) は監視対象の分散システムを表している。distributed systems 内の四角形は分散システムを構成しているノードを表している。Monitoring server(以下、モニタリングサーバー) は分散システムのノードとして参加

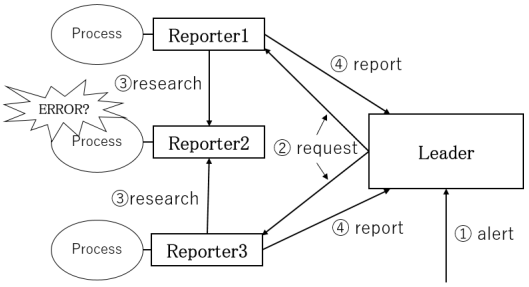

図 2　処理の流れ

していない別途の監視サーバーとしている。図中の水色部分が既存ソフトウェアを表し、赤部分が本提案の新規ソフトウェアを表している。

図 2 に処理の流れを示す。まず、モニタリングシステムからアラートをモニタリングサーバー内の Leader へ送信する。このアラートが本提案のソフトウェアにおいてトリガーの役割を担っている。

アラートは Leader 内の Control で受信される。アラートを受信した Control は Leader 内の connector へ、プロセス異常の可能性があるノード (以下、対象ノード) のホスト名を渡す。connector は渡された対象ノードのを元に、健康なノードに常駐している Reporter へ対象ノードのホスト名と共にプロセス状態の確認を要求する。ここで Reporter から指定時間以内に応答がない場合、NoReply として処理する。

Leader から要求を受けた Reporter は、送られてきた対象ノードのホスト名を受信する。受信後、正常に依頼を受け取った事を知らせるために Leader へ返答を送る。その後、送られてきた対象ノードのホスト名を元に、そのノードに常駐している Reporter へプロセス状態の情報を要求する。この際、乱数を用いて通信のタイミングをずらす。これにより、各ノードが一斉に要求を試みることを防ぐ。ここで対象ノードの Reporter から指定時間以内にない場合、NoReply として Leader へ結果を送信する。返答があった場合、その返答結果を Leader へと送信する。

対象ノード内の Reporter は、各ノードからの要求を受信後、Reporter 内の Monitoring でプロセス状態の確認を行う。そして確認した結果を各ノードを返答として送信する。ここで送れる返信は Alive、もしくは Dead である。

確認をした各ノードの Reporter から送られてきた応答を元に、Leader 内の Decision で状態の決定をする。状態の決定方法は送られた Alive、Dead の二つから多かった方をプロセスの状態として決定する。NoReply は無効票として扱い、状態の決定に反映されない。

3.3 信頼度

最終的な決定を多数決の形で実施する場合、状況次第では一つの決定に定める事が出来ない可能性がある。例えば、6 個のノードを持つ分散システムがあるとする。そのうちの 1 個にプロセスの異常の可能性が発生したとする。この場合、その他の 5 個のノードに常駐している Reporter が状態の確認を実施し状態の判別をする。この時、Alive が 2 票、Dead が 2 票、NoReply が 1 票であった。この場合、Alive と Dead が同数であるため一つに決定する事が出来ないのである。

表 2 信頼度の段階

信頼度	説明
trust	一番高い信頼度
normal	初期値
lowest	一番低い信頼度

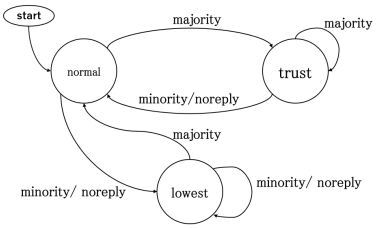

図 3 信頼度の遷移

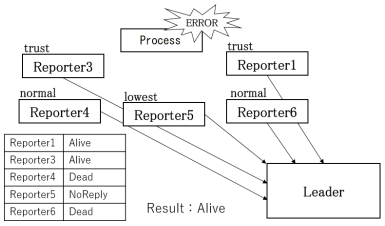

図 4 信頼度導入後のプロセス判別

この状況を防ぐために、各ノードの Reporter に信頼度を設定して重みをつけることを提案する。信頼度はプロセス状態の判別のときに、最終的に決定された状態と同じ状態を返答として送った Reporter の信頼度を上昇させる。決定されなかった状態の方へ投票する、もしくは NoReply と Leader に送った場合、その Reporter の信頼度を下降させる。信頼度は表 2 の 3 段階としている。投票が同数になり決定が出来ないとき、信頼度が高い Reporter からの投票ほど状態の決定の際に優先される。

trust は一番高い信頼度であり、決定が出来なかった際に最優先で判断材料として Leader に返した応答が使用される。trust の Reporter は、NoReply を返答することは低確率であるという仮定の下、奇数個のノードを trust とするべきである。これは、信頼度の制約上、NoReply を高頻度で返す Reporter の信頼度が trust になることはないという根拠の元に仮定している。

normal は 3 段階のうち、中間の信頼度であり各 Reporter に設定される信頼度の初期値である。trust だけでは状態の決定に至れなかった場合に、normal からの投票も有効として再度決定を試みる。

lowest は 3 段階のうち、一番低い信頼度である。高頻度で NoReply を返してしまう。決定の際に少数意見の側であることが多い Reporter である。この信頼度レベルの Reporter はネットワークの遅延や、外部からの干渉による改ざんの可能性がある。lowest からの投票は初回の決定以外では反映されない。

各ノードの Reporter の信頼度を図 4 の状態だとする。図 4 では、Alive と Dead が 2 票ずつで同数となる。しかし、Dead の方は信頼度が normal である Reporter2 個が投票し

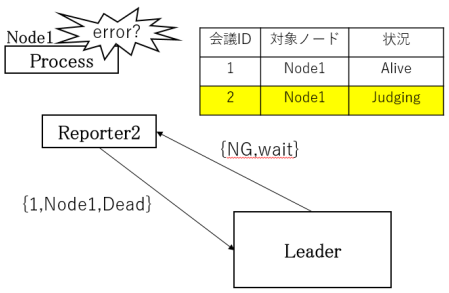

図 5　会議 ID

ている。対して、Alive は信頼度が trust である Reporter2 個が投票している。これにより、Leader はプロセスは生きてると判断する。Alive と Dead が同数での判断の場合、信頼度の変動はない。

3.4　会議 ID

マシン間でデータのやり取りをする関係上、通信が遅延することも考慮しなければならない。一定時間内に返答が無かった場合は NoReply として処理することを 3.2.1 章で示した。しかし、予期せぬタイミングで遅れて返答が届いてしまう可能性がある。例えば、ノード 1 のプロセスの状態判別を実施したとする。その際、Reporter2 から時間内に返答が無かったので NoReply として処理されたと仮定する。ノード 1 のプロセス判別が正常に終了したとする。しかし、再度ノード 1 のプロセスの状態判別を実施されたとする。このときに、遅れていた前のノード 1 のプロセスの状態が Leader に届いた場合、状態判別が正常に行えない。正しい状態判別をサポートするため、一つの状態判別の一連の処理の流れに会議 ID を割り当てる。

会議 ID は固有の値であり、同時に同じ値の ID は存在しない。会議 ID は Leader が管理する。Leader は一つの状態判別の処理が終わるごとに会議 ID を+1 する。Leader と Reporter 間の通信では会議 ID を参照しながら行う。Reporter から最新の会議 ID ではない ID と共に送られてきた応答を受け取った Leader は、図 5 で示すように、処理を拒否して次の処理までその Reporter を待機させる。途中からの参加を認めない。

4.　評価

実験は図 6 のように 5 台の Ubuntu18.04 の VM を使用する。そのうちの 4 台に Nginx を導入する。そして、1 台をロードバランサーとして使用する。残りの 3 台を Web サーバとして使用する。Nginx を導入しない 1 台は監視サーバとして使用する。監視サーバには Leader を配置する。3 台の Web サーバにはそれぞれ Reporter を配置する。ロードバランシングの手法はラウンドロビンとする。

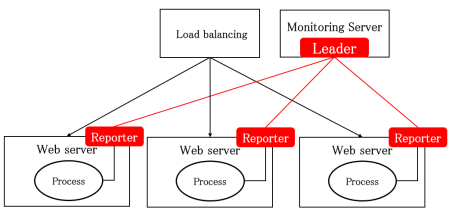

図 6　実験環境

Leader はプロセスに異常が発生したとみられる際にソフトウェアを実行する。しかし、本実験でメトリックスの異常ではなく、実験者による任意のタイミングで動作させる。Reporter は実行されている web サーバーのプロセス状態を監視する。

本実験では意図的にプロセスに異常を発生させる。ここでは、プロセスの死亡は、プロセスの実行状況に関わらずプロセスが終了してしまった場合や Web サーバーがダウンしてしまうこととする。プロセスを殺す方法として、Ubuntu ではプロセスを強制終了させる kill コマンドが存在する。これにより、web サーバーのプロセスを強制終了させる方法がある。

プロセスの応答の遅延は、web ページで動作させる JavaScript の setTimeout 関数を使用する方法、for 文のループにより処理を遅延させる方法がある。本実験では後者の for 文によるループを採用する。

これらのプロセス異常を任意のタイミング、任意の web サーバーで実行させる。異常を発生させた後ソフトウェアを実行する。これにより、本稿の提案がどの程度正しくプロセスを判別できるのかを実験する。

5.　議論

改善点の一つがトリガーとなるメトリクスの選定である。本稿の提案ではメトリクスの指定を行わなかった。候補として、ネットワークのトラフィックがある。トラフィックの量を監視することで、そのサーバーが応答を返すことが出来ているのかを確認できる。これにより、表 1 での「通信・呼び出し」の点を判断することできると予想できる。しかし、サーバ、プロセスに異常が発生する理由は許容量を超えるアクセスや、物理的な障害によるものと多岐にわたる。着目する最適なメトリクスの調査は早急に解決すべき課題の一つである。

信頼度システムにおいて、信頼度に差異が無い状態で Alive,Dead が同数であった場合どのように状態を状態を判別するのかという課題がある。投票が実施された際、意見が同数に割れてしまいプロセス状態を一つに決定できない状況を解決するために信頼度という概念を 3.3 章で提案した。しかし、これは複数回プロセスの状態判別が行われないと信頼度に差が生まれない。

信頼度は初期では全 Reporter は normal として設定されている。この初期値の決定に変更を加えることが解決方法の一つである。例として、Reporter を配置した段階で

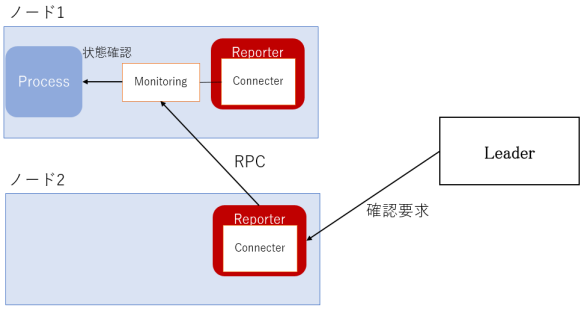

図 7 Connecter と Monitoring の分離

Reporter に Leader 自身のプロセスを状態を調べさせる。この結果と応答速度によって、初期の信頼度を定める。これにより、初期の段階から信頼度に差を設ける。

別の方法として、Reporter が返答として送信する Alive、Dead に重みを設定するものがある。これにより、Alive と Dead が同数、投票した Reporter が同信頼度であった場合にも一つに決定することを可能にする。重みの設定する際、異常が疑われる場合は故障 (Dead) とする悲観的な考え方をとる。もしくは、異常が疑われる場合もプロセスが動いている (Alive) とする楽観的な考え方をとるという 2 種類の考え方がある。

本提案では、Leader の役割を担うノードは単一であり、状態判別までの処理の大部分が Leader の主導により行われる。これは、処理を主導する存在を一つにする事によりプロセスの状態判別までの過程の複雑性を減らす。しかし、単一である Leader が故障した際は全体が停止してしまう。Leader である 1 機が故障してしまうだけでシステム全体が機能できないという点では耐障害性に問題があることは明白である。

耐障害性の課題を解決する方法として、参加する全てのノードが Leader と Reporter の二つ役割を担えるようにする方法を提案する。動作開始時、全てのノードから一つの Leader ノードを決定する。Leader となったノード以外のノードは全て Reporter になり、動作を開始する。Leader が故障した際は、再度 Leader の決定を開始する。Leader の決定については Raft において、すでに実装されている例がある [2]。

Reporter が対象ノードにプロセス状態の確認を実施する際、その対象ノードに常駐している Reporter からプロセス状態を得る。これは、対象ノードの Reporter が正しい返答が出来るという前提があり、結果はそのノードに常駐している Reporter に依存することになる。対象ノードの Reporter に異常が発生して応答できない、不正な改ざんにより正しい返答が出来なくなった場合、確認を行った全てのノードは間違った結果を返す。結果的にシステム全体で間違った判断をしてしまう。

この課題を解決する方法として、プロセス状態の確認を対象ノードの Reporter を介さずに行う。図 7 のように Reporter 内の Connectoer と Monitoring を分離する事により、外部の Reporter が Monitoring を使用して直接プロセス状態を確認できるようにする。これにより、対象ノードの Reporter への依存度を軽減する。

6. おわりに

本稿では、分散システムを構成している各ノードを使用して、プロセスのを判別する方法を提案した。本提案では、監視サーバと対象ノードとの一対一のやり取りでの監視ではなく他ノードを使用してプロセスの状態判別を実施する。これにより、監視サーバから直接確認が行えない状況でもプロセスの状態を確認できるという点に優位性がある。さらに、他ノードを使用した際に起こりうる問題に対しての対策と解決方法を提示した。しかし 5 章で示したように、実用的なシステムへの昇華には解決すべき点や検討すべき点がある事も事実である。これらについて実験を実施し、最適なアプローチを調査すべきである。

参考文献

[1] Steen, M. and Tanenbaum, A. S.: A Brief Introduction to Distributed Systems, *Computing*, Vol. 98, No. 10, pp. 967–1009 (online), DOI: 10.1007/s00607-016-0508-7 (2016).

[2] 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).

[3] Lamport, L.: Using Time Instead of Timeout for Fault-Tolerant Distributed Systems., *ACM Trans. Program. Lang. Syst.*, Vol. 6, No. 2, pp. 254–280 (online), DOI: 10.1145/2993.2994 (1984).

[4] Dialani, V., Miles, S., Moreau, L., Roure, D. D. and Luck, M.: Transparent Fault Tolerance for Web Services Based Architectures, *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, London, UK, UK, Springer-Verlag, pp. 889–898 (online), available from ⟨http://dl.acm.org/citation.cfm?id=646667.700187⟩ (2002).

[5] Kandasamy, N., Hayes, J. P. and Murray, B. T.: Transparent Recovery from Intermittent Faults in Time-triggered Distributed Systems, *IEEE Trans. Comput.*, Vol. 52, No. 2, pp. 113–125 (online), DOI: 10.1109/TC.2003.1176980 (2003).

[6] Strom, R. and Yemini, S.: Optimistic Recovery in Distributed Systems, *ACM Trans. Comput. Syst.*, Vol. 3, No. 3, pp. 204–226 (online), DOI: 10.1145/3959.3962 (1985).

[7] Skrzypczynski, G.: Failure Transparency in Corba Systems, *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, PARELEC '02, Washington, DC, USA, IEEE Computer Society, pp. 228– (online), available from

⟨http://dl.acm.org/citation.cfm?id=824476.825993⟩ (2002).

[8] Lowell, D. E., Chandra, S. and Chen, P. M.: Exploring Failure Transparency and the Limits of Generic Recovery, *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, Berkeley, CA, USA, USENIX Association, (online), available from ⟨http://dl.acm.org/citation.cfm?id=1251229.1251249⟩ (2000).

[9] Ravindran, K. and Chanson, S. T.: Failure Transparency in Remote Procedure Calls, *IEEE Trans. Comput.*, Vol. 38, No. 8, pp. 1173–1187 (online), DOI: 10.1109/12.30871 (1989).

[10] Lamport, L.: The Part-time Parliament, *ACM Trans. Comput. Syst.*, Vol. 16, No. 2, pp. 133–169 (online), DOI: 10.1145/279227.279229 (1998).

